МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГБОУ ВО «Пермский государственный национальный исследовательский университет»

Физический факультет

Кафедра радиоэлектроники и защиты информации

**ДЕТЕКТОР АНОМАЛИЙ ПОВЕДЕНИЯ СЕТЕВОГО ОБОРУДОВАНИЯ**

Курсовая работа

|  |  |
| --- | --- |
|  | Работу выполнил  студент 3 курса специальности  “Информационная безопасность автоматизированных систем”  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.А. Пагин |
| Оценка работы научным руководителем  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (оценка, выставленная руководителем)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата, подпись руководителя) | Научный руководитель:  старший преподаватель кафедры радиоэлектроники и защиты информации  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.И. Моисеев |
| Оценка работы комиссией по защите курсовых работ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (оценка, выставленная комиссией)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата, подпись председателя комиссии) |  |

Пермь, 2017

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc480451912)

[ОБЗОР ВАРИАНТОВ РЕШЕНИЯ 4](#_Toc480451913)

[Выбор протокола. 4](#_Toc480451914)

[Выбор алгоритма. 4](#_Toc480451915)

[Выбор среды разработки. 4](#_Toc480451916)

[ВЫДЕЛЕНИЕ НАИВАЖНЕЙШИХ ПОКАЗАТЕЛЕЙ СОСТОЯНИЯ РОУТЕРА. ПРОТОКОЛ SNMP. 6](#_Toc480451917)

[РАЗРАБОТКА ЭКСПЕРТНОЙ СИСТЕМЫ 7](#_Toc480451918)

[ХРАНЕНИЕ ДАННЫХ. RRD 8](#_Toc480451919)

[ОРГАНИЗАЦИЯ ВОЗМОЖНОСТЕЙ ПРОТОКОЛА SNMP И ВОЗМОЖНОСТЕЙ RRDTOOL В СРЕДЕ PYTHON. 10](#_Toc480451920)

[АЛГОРИТМ РАБОТЫ ОСНОВНОГО МОДУЛЯ. 12](#_Toc480451921)

[ТЕСТИРОВАНИЕ 13](#_Toc480451922)

[ВОЗМОЖНОСТИ ПОСТРОЕНИЯ ГРАФИКОВ И ПРОСМОТРА ДАННЫХ. 16](#_Toc480451923)

[ЗАКЛЮЧЕНИЕ 19](#_Toc480451924)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 20](#_Toc480451925)

[ПРИЛОЖЕНИЕ №1 21](#_Toc480451926)

[ПРИЛОЖЕНИЕ №2 24](#_Toc480451927)

[ПРИЛОЖЕНИЕ №3 25](#_Toc480451928)

[ПРИЛОЖЕНИЕ №4 33](#_Toc480451929)

# ВВЕДЕНИЕ

Двадцатый век уже не представить без глобальной сети Интернета. Вместе с его развитием неизбежно разрастаются и различные локальные сети, организующие доступ в интернет, либо просто для объединения большого количества устройств предприятий в единую сеть. Вместе с этим, безусловно, увеличивается и количество коммутаторов и маршрутизаторов в данных сетях. По мере подключения в сеть все новых и новых конечных пользователей, маршрутизаторам все сложнее справляться с нагрузкой, что ведет к дальнейшему увеличению их количества. Когда сеть разрастается, сетевому администратору становится все сложнее уследить за функционированием всех устройств в сети. На сегодняшний день существует большое количество приложений/дополнений, позволяющих вести мониторинг нагрузки на различные узлы сети, но, в большинстве своем, они дают лишь сухие данные, за которыми нужно постоянно следить, выявляя различные аномалии поведения сетевого оборудования, будь то излишняя нагрузка на процессор роутера или большое количество одновременно подключенных пользователей.

В данной работе будет рассмотрен процесс создания приложения, которое позволит не только вести мониторинг состояния сетевого оборудования, но и в реальном времени отслеживать различные отклонения от нормы, своевременно сообщая об этом сетевому администратору сети.

Таким, образом, цель работы – *разработать модуль для детектирования различного рода аномалий в поведении сетевого оборудования*.

Для достижения данной цели можно выделить следующие задачи:

* Выделить наиболее важные параметры, на основе которых будет происходить детектирование
* Разработать экспертную систему для данных параметров
* Объединить вышеописанные шаги в отдельном программном модуле
* Протестировать модуль в реальных условиях

# ОБЗОР ВАРИАНТОВ РЕШЕНИЯ

## Выбор протокола.

Для работы с оборудованием Cisco был разработал сетевой протокол NetFlow, предназначенный для учета сетевого трафика. Однако использование данного протокола заточено лишь под мониторинг трафика. Различные онлайн-приложения позволяют с помощью данного протокола вести мониторинг трафика, проходящего через устройство, не предлагая при этом каких-либо средств для детектирования аномалий. К тому же данный протокол поддерживается в основном только устройствами Сisco.

Протокол SNMP[1] – это протокол типа «запрос-ответ», который и позволяет посредством простых запросов получать от устройства данные метрики. Данный протокол имеет большую универсальность, будучи поддерживаемым всеми сетевыми устройствами.

Таким образом, выбран протокол SNMP.

## Выбор алгоритма.

В сети можно найти решения в сфере детектирования аномалии поведения сетевого оборудования, основанные на анализе содержимого пакетов посредством нейронных сетей. Данный подход ресурсозатратен, в достаточной степени эмпиричен и не имеет привязки к непосредственным аппаратным средствам сетевого оборудования.

Протокол SNMP позволяет получать как данные о трафике, так и данные о состоянии аппаратных элементов. Объединение данных показателей для детектирования аномалий имеет больший охват, чем подход с анализом трафика. Для детектирования аномалий в данном случае предлагается использовать жесткие алгоритмы посредством разработанной экспертной системы.

Таким образом, выбран анализ на основе данных о трафике и состоянии аппаратных элементов.

## Выбор среды разработки.

Библиотеки для протокола SNMP существуют почти для всех современных языков программирования. Учитывая факт большого количества open-source библиотек и пакетов для \*unix подобных систем, предпочтительнее выбрать операционную систему CentOS 5 вместо привычных Windows-платформ.

\*unix подобные системы поддерживают легкую установку большинства языков программирования. На сегодняшний день наиболее распространены языки с объектно-ориентированной парадигмой - С# и Python. Однако, в среднем объем кода на языке Python меньше, чем для аналогичных алгоритмов на С#. Кроме того, для языка Python существует большее количество open-source библиотек.

Таким образом выбрана ОС CentOS 5 и язык программирования Python v3.4.

# ВЫДЕЛЕНИЕ НАИВАЖНЕЙШИХ ПОКАЗАТЕЛЕЙ СОСТОЯНИЯ РОУТЕРА. ПРОТОКОЛ SNMP.

Современные маршрутизаторы содержат внутри себя достаточно большие производительные мощности – это достаточно мощные процессоры и достаточные объемы оперативной памяти. Эти два аппаратных показателя несут за собой достаточно большой объем информации о состоянии оборудования. Кроме того, необходимо вспомнить и о такой метрике, как число приходящих пакетов за единицу времени. В совокупности три данных значения дают обширных охват для мониторинга состояния роутера/маршрутизатора. Выше мы определились, что будем использовать протокол SNMP.

Для подключения к устройству по SNMP необходимы имя группы и IP-адрес устройства. Для тестирования работы приложения был выдан доступ к коммутатору Cisco Catalyst 4500 series. Подключение организуется в части программного кода (Рисунок 1):

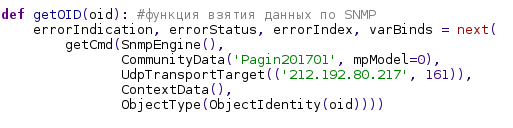


Рисунок 1

Где имя группы – Pagin201701 и адрес устройства – 212.192.80.217.

Идентификация запрашиваемых данных определяется с помощью OID’ов. Например (Рисунок 2):

image (3)

Рисунок 2

Где последовательность «.1.3.6.1.2.1.3.1.1.1.1.3.3» кодирует данные о количестве broadcast пакетов. Последняя цифра последовательности обозначает номер интерфейса, данные о котором запрашиваются. Для данной работы собирались данные с интерфейсов:

* Te 1/1-2
* Gi 2/1-48
* Gi 3/1-12

# РАЗРАБОТКА ЭКСПЕРТНОЙ СИСТЕМЫ

Ранее был сделан акцент на наиболее важных параметрах роутеров, которые мы можем получить по протоколу SNMP:

* Загрузка CPU
* Объем свободной RAM
* Количество пакетов(broadcast/unicast/multicast)

Но имея лишь эти последовательно поступающие данные, разрабатываемая система ничем не будет отличаться от обычных систем мониторинга. Для поиска и выявления различного рода аномалий необходимо разработать алгоритмы, по которым будет детектироваться «необычное» поведение оборудования.

Рассмотрим методы детектирования для значений CPU и RAM.  
Значения CPU и RAM – достаточно показательные значения, для которых можно выделить следующие крайние случаи:

* Загрузка более, чем на 80%. Что говорит о необходимости установки более мощного устройства, т.к. данное устройство вскоре может начать не справляться с предложенной нагрузкой.
* Единовременный скачок загрузки более, чем на 15%. Суточная загрузка процессора имеет некий тренд, однако, он почти не прослеживается (в течение суток на тестовом коммутаторе загрузка CPU была в среднем около 10-15%). А резкое изменение является «необычным» явлением.

Теперь рассмотрим методы детектирования для количества пакетов трафика:

* Увеличение количества пришедших пакетов на 20% в сравнении с количеством пакетом, соответствующего времени прошлой недели. Так как количество пришедших пакетов имеет значительную привязку к недельному тренду, необходимо ориентироваться на прошлые значения.
* Единовременный скачок количества пришедших пакетов более, чем на 10%. Несмотря на то, что первая метрика рассматривает привязку к тренду, необходимо рассматривать и резкие скачки в течение одного промежутка времени.

# ХРАНЕНИЕ ДАННЫХ. RRD

Для обработки и хранения данных, забираемых с сетевого оборудования по протоколу SNMP необходимо было завести какую-либо базу данных. Так как разрабатываемый программный модуль может использоваться неограниченное количество времени, обычные базы данных могут бесконечно разрастаться, что недопустимо. Поэтому были выбраны циклические базы данных RRD. Для всех собираемых данных их структура имеет аналогичный вид (Рисунок 3):

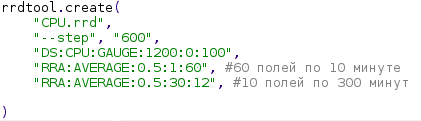


Рисунок 3

Данный пример для хранения значений загруженности CPU. Сначала создается файл для хранения значений – «CPU.rrd». Далее задается шаг для поступления новых значений – 600 секунд (10 минут). Далее задается поле, в которое будут сохраняться значения и тип данных для него – поле «CPU», тип данных – GAUGE, абсолютные значения данных. Последние 2 строки задают тип интерполяции данных – AVERAGE. Когда первое поле будет полностью заполнено, последние 30 значений из него усреднятся и будут перенесены в следующее поле, которое хранит 10 5-часовых усредненных значений. При переполнении базы, последние значения будут откидываться.

СОЗДАНИЕ ВЫБОРКИ ДЛЯ ТРАФИКА.

Как было сказано ранее, один из методов детектирования аномалий поведения сетевого оборудования опирается на данные о пришедших пакетах трафика за предыдущую неделю. Для этого было необходимо сделать отдельную выборку. Для этого был создан отдельный программный модуль (Приложение №1). В течение недели собирались данные о пришедших пакетах за каждые 10 минут. Далее использовался еще один программный модуль (Приложение №2) для приведения данных к необходимому виду. Итоговая выборка имеет вид (Рисунок 4):

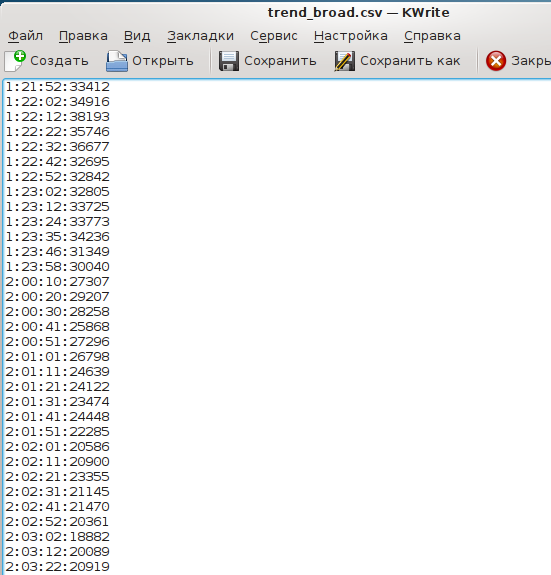


Рисунок 4

В каждой строке первая цифра кодирует день недели (Понедельник – 1, вторник – 2 и т.д.), вторая – часы, третья – минуты, четвертая – количество пакетов, пришедшие за промежуток от предыдущего значения до нового. Аналогичные наборы значений были созданы для всех трех типов трафика – broadcast/unicast/multicast.

# ОРГАНИЗАЦИЯ ВОЗМОЖНОСТЕЙ ПРОТОКОЛА SNMP И ВОЗМОЖНОСТЕЙ RRDTOOL В СРЕДЕ PYTHON.

Для разработки было решено написать консольное приложение на языке Python v.3.4[2]. Установка Python (Рисунок 5):

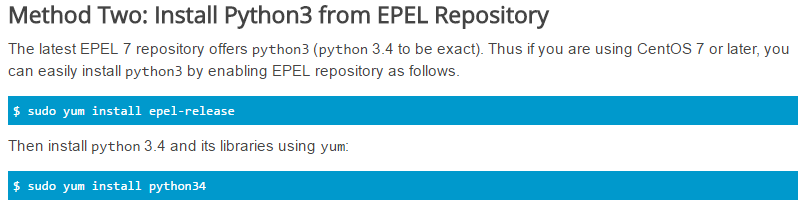


Рисунок 5

Работа с данным языком наиболее удобна в случае разработки детектора аномалий поведения сетевого оборудования, вследствие простоты установки на \*unix-подобные системы и наличии большого количества open-source библиотек.

Рассмотренный протокол SNMP и возможности работы с базами данных RRD, было необходимо перенести в среду разработки. Для этого были выбраны opensource библиотеки: «rrdtool»[4] и «pysnmp»[3].

Установка «rrdtool» происходит следующими командами (Рисунок 6):

Рисунок 6

Установка «pysnmp» происходит следующими командами (Рисунок 7):

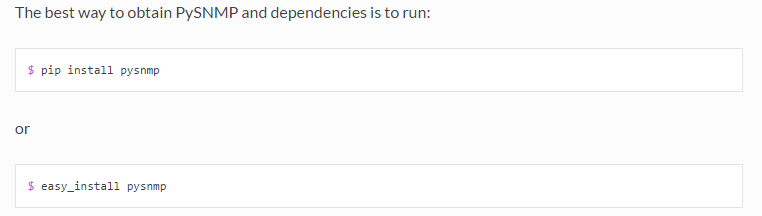


Рисунок 7

Данные библиотеки наиболее оптимизированы и имеют наиболее удобный и интуитивно понятный синтаксис, вследствие чего и были выбраны.

# АЛГОРИТМ РАБОТЫ ОСНОВНОГО МОДУЛЯ.

Основной модуль [Приложение 3] имеет следующий алгоритм работы:

1. Опрос загруженности CPU
2. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
3. Опрос количества свободной RAM
4. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
5. Опрос количества пришедших Broadсast пакетов.
6. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
7. Опрос количества пришедших Multiсast пакетов.
8. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
9. Опрос количества пришедших Uniсast пакетов.
10. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
11. Ожидание времени до следующего опроса
12. Переход к пункту 1.

# ТЕСТИРОВАНИЕ

При запуске приложение [Приложение 3] автоматически создает необходимые RRD базы данных и начинает захват данных по протоколу SNMP. В случае обнаружения какой-либо аномалии, пользователю сообщается об этом в окне приложения и заносится соответствующая записать в файл логирования «logging.log».

Для тестирования данный модуль был запущен. В течение первых 20 минут его работа была при обычных условиях нагрузки на коммутатор. Далее на 3 машинах в данной сети были запущены команды (Рисунок 8):

image (1)

Рисунок 8

Данная команда запускает бесконечный ping-flood пакетами по 3600 байт на тестируемое устройство.

Вывод программы при этом (Рисунок 9):

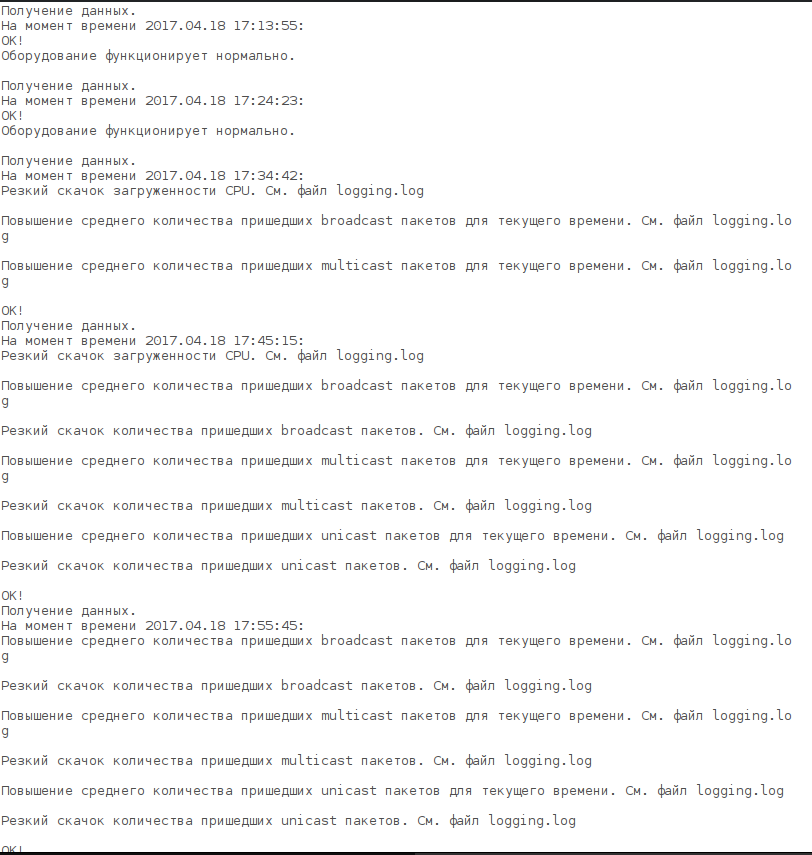


Рисунок 9

Как можно заметить, первые 20 минут работы все было хорошо. Рассмотрим последующие моменты времени отдельно:

* 17:34:42 - программа выявила скачок загруженности CPU и повышенные количества broadcast и multicast пакетов.
* 17:45:15 - все введенные в экспертную систему метрики сработали. Однако, как можно заметить, на загруженность RAM данный тест влияния не имел.
* 17:55:45 – загруженность CPU стабилизировалась и при этом была меньше 85%, поэтому в выводе программы исчезло предупреждение о скачке нагрузки на CPU.

Все «происшествия» вносятся в файл “logging.log”. После данного запуска программы содержимое файла выглядит так (Рисунок 10):

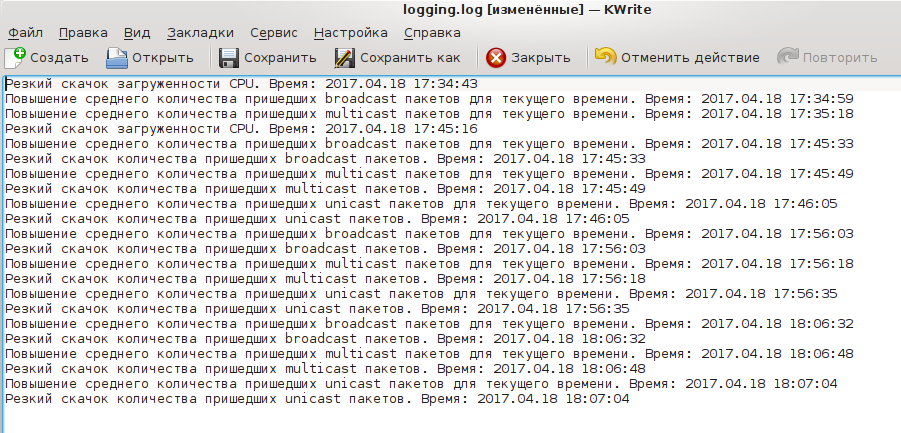


Рисунок 10

Видим, что все аномалии успешно залогированы. Таким образом, разработанный программный модуль успешно детектирует аномалии, введенные в экспертную систему.

# ВОЗМОЖНОСТИ ПОСТРОЕНИЯ ГРАФИКОВ И ПРОСМОТРА ДАННЫХ.

Для дополнительного удобства анализа данных пользователем был создан отдельный модуль [Приложение №4]. При запуске данный модуль выдает содержимое всех RRD баз данных (Рисунок 11):

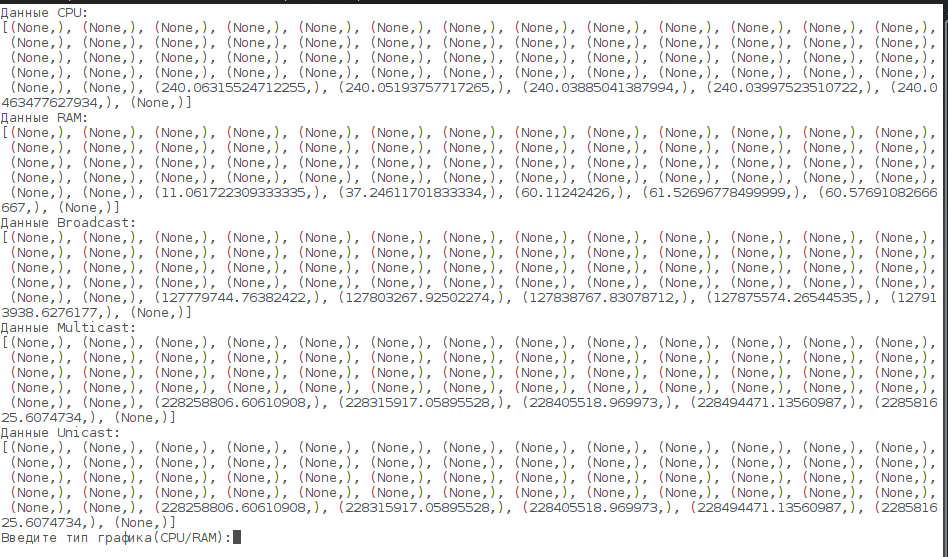


Рисунок 11

Далее организован диалог с пользователем, в котором предлагается выбрать данные CPU или RAM и соответствующее количество часов, за которое необходимо построить график (Рисунок 12):

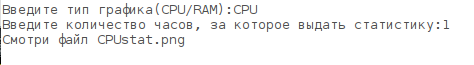


Рисунок 12

При удачном построении, модуль предлагает посмотреть созданный графический файл. Создадим и просмотрим график для данных из блока тестирования для CPU (Рисунок 13) и RAM (Рисунок 14):

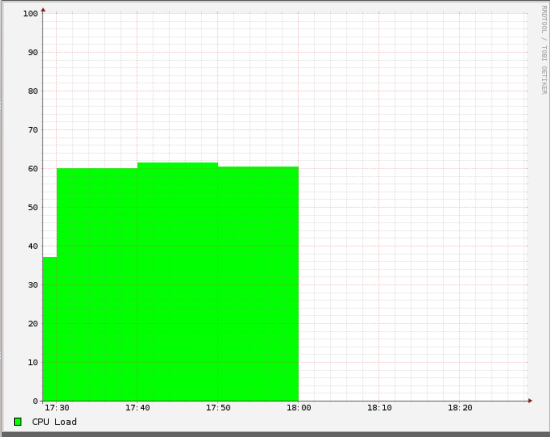


Рисунок 13

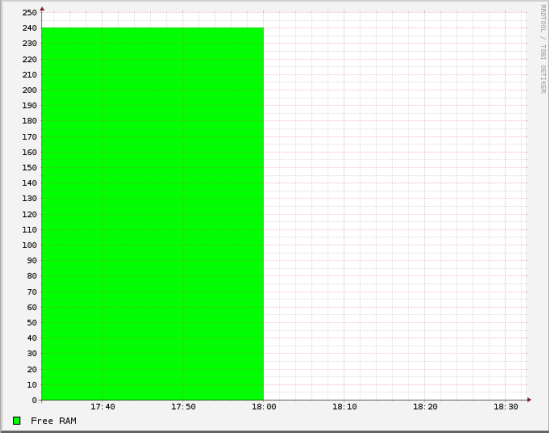


Рисунок 14

В начале графика CPU хорошо видно тот самый скачок загруженности с 36% до 60%, который был задетектирован программой. При этом, как мы видим, загруженность RAM никак не менялась.

Построение графиков успешно функционирует и помогает пользователю наглядно увидеть аномалии в различных параметрах сетевого оборудования.

# ЗАКЛЮЧЕНИЕ

Как было показано, разработанное приложение помогает системному администратору своевременно выявлять те или иные отклонения в поведении сетевого оборудования. Данная разработка в перспективе может сильно расширятся, включая в себя все больше показателей, а также анализ данных посредством нейросетей. В ходе работы были встречены некоторые проблемы, которые могут быть успешно решены более долгим созданием опорной выборки для данных. В целом, данная работа включила в себя немало областей – от анализа данных до устройства сетевого оборудования и программирования на Python.

Как вывод, можно сказать о том, что данное приложение может помочь в детектировании аномалий уже сейчас, что, несомненно, является хорошим окончанием для данной работы.

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. В. Олифер, Н. Олифер «Компьютерные сети», изд. «Питер», 2016, стр. 731
2. How to install Python3 on CentOS [Online] http://ask.xmodulo.com/install-python3-centos.html
3. Pysnmp Quick Start. [Online] http://pysnmp.sourceforge.net/quick-start.html
4. Rrdpython. [Online] http://oss.oetiker.ch/rrdtool/prog/rrdpython.en.html

# ПРИЛОЖЕНИЕ №1

#! -\*- coding: utf-8 -\*-

from pysnmp.hlapi import \*

import time

from datetime import datetime

def getOID(oid):

errorIndication, errorStatus, errorIndex, varBinds = next(

getCmd(SnmpEngine(),

CommunityData('Pagin201701', mpModel=0), #подключение к роутеру

UdpTransportTarget(('212.192.80.217', 161)),

ContextData(),

ObjectType(ObjectIdentity(oid))))

if errorIndication:

print(errorIndication)

elif errorStatus:

print('%s at %s' % (errorStatus.prettyPrint(),

errorIndex and varBinds[int(errorIndex) - 1][0] or '?'))

else:

for varBind in varBinds:

print(' = '.join([x.prettyPrint() for x in varBind]))

return str(varBind[1])

i=0

while i<1:

#Обработка для multicast пакетов

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.3"))

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #

j=j+1

new\_file = open("trand\_multi.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

#Обработка для broadcast пакетов

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.3"))

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

new\_file = open("trand\_broad.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

#Обработка для unicast пакетов

traffic=int(getOID(".1.3.6.1.2.1.2.2.1.11.3"))

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

new\_file = open("trand\_uni.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

time.sleep(580)

# ПРИЛОЖЕНИЕ №2

import time

from datetime import datetime

fileob = open('trand\_uni.csv') #открытие файла, созданного предыдущим модулем

arr = fileob.read().split("\n")

fileob.close()

i=1

print(len(arr))

j=1 #начиная с понедельника

while i<len(arr)-1:

string=arr[i].split(":")

string2=arr[i-1].split(":")

print(string)

new\_file = open("trend\_uni.csv", 'a')

#вычисление и запись числа пришедших пакетов за промежуток времени

new\_file.write(str(j)+":"+string[0]+":"+string[1]+":"+str(int(string[2])-int(string2[2]))+"\n")

razn=int(string2[0][0])-int(string[0][0])

if(razn > 1): #проверка наступления нового дня

j+=1

new\_file.close()

i+=1

print(“ОК!”)

# ПРИЛОЖЕНИЕ №3

#! -\*- coding: utf-8 -\*-

from pysnmp.hlapi import \*

import time

from datetime import datetime

import rrdtool

def getOID(oid): #функция взятия данных по SNMP

errorIndication, errorStatus, errorIndex, varBinds = next(

getCmd(SnmpEngine(),

CommunityData('Pagin201701', mpModel=0),

UdpTransportTarget(('212.192.80.217', 161)),

ContextData(),

ObjectType(ObjectIdentity(oid))))

if errorIndication:

print(errorIndication)

elif errorStatus:

print('%s at %s' % (errorStatus.prettyPrint(),

errorIndex and varBinds[int(errorIndex) - 1][0] or '?'))

return str(varBinds[0][1])

rrdtool.create(

"CPU.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"RAM.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:300",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_broad.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_multi.rrd",

# "--start", "now",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_uni.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

i=0

while i<1:

print("Получение данных.")

start = datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")

print("На момент времени "+start+":")

flag=0

#взять массив элементов CPU.rrd

result = rrdtool.fetch("CPU.rrd", "AVERAGE",'-s', str(rrdtool.first("CPU.rrd")),

'-e', str(rrdtool.last("CPU.rrd")),

'-r', "2")

CPU\_used = getOID(".1.3.6.1.4.1.9.2.1.57.0")

#проверка на скачок заггруженности CPU

if(result[2][-2][0]):

if((abs(int(result[2][-2][0]) - int(CPU\_used)))>10):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок загруженности СPU. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок загруженности СPU. См. файл logging.log\n")

new\_file.close()

#Проверка на повышенную загруженность CPU

if(int(CPU\_used)>85):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Высокий процент загруженности СPU: "+ CPU\_used+"%. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Высокий процент загруженности СPU: "+ CPU\_used+" См. файл logging.log\n")

new\_file.close()

rrdtool.update("CPU.rrd", "N:"+str(CPU\_used))

#взять массив элементов RAM.rrd

result = rrdtool.fetch("RAM.rrd", "AVERAGE",'-s', str(rrdtool.first("RAM.rrd")),

'-e', str(rrdtool.last("RAM.rrd")),

'-r', "2")

RAM\_free\_bt=getOID(".1.3.6.1.4.1.9.2.1.8.0")

Ram\_free\_mb = float(RAM\_free\_bt)/(1024\*1024)

#Проверка на скачок загруженности RAM

if(result[2][-2][0]):

if(abs(int(result[2][-2][0])-int(Ram\_free\_mb))>5):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкое уменьшение размера свободной памяти RAM. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкое уменьшение размера свободной памяти RAM. См. файл logging.log\n")

new\_file.close()

#Проверка на повышенную загруженность RAM

if(Ram\_free\_mb<100):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Малый объем свободной RAM: "+ Ram\_free\_mb+" mb. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Малый объем свободной RAM: "+ Ram\_free\_mb+" mb. См. файл logging.log\n")

new\_file.close()

rrdtool.update("RAM.rrd", "N:"+str(Ram\_free\_mb))

result = rrdtool.fetch("Traffic\_broad.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_broad.rrd")),

'-e', str(rrdtool.last("Traffic\_broad.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.3")) #broadcast

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_broad.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

print(str(normal) + " " + str(int(traffic)-int(result[2][-2][0])))

if((((int(traffic)-int(result[2][-2][0]))-int(normal))/int(normal))>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших broadcast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших broadcast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших broadcast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших broadcast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_broad.rrd", "N:"+str(traffic))

result = rrdtool.fetch("Traffic\_multi.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_multi.rrd")),

'-e', str(rrdtool.last("Traffic\_multi.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.3")) #multicast

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #multi/broad/uni

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #multi/broad/uni

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_multi.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

if((((int(traffic)-int(result[2][-2][0]))-normal)/normal)>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших multicast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших multicast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших multicast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших multicast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_multi.rrd", "N:"+str(traffic))

result = rrdtool.fetch("Traffic\_uni.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_uni.rrd")),

'-e', str(rrdtool.last("Traffic\_uni.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.2.2.1.11.3")) #unicast

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_uni.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

if((((int(traffic)-int(result[2][-2][0]))-normal)/normal)>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших unicast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших unicast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших unicast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших unicast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_uni.rrd", "N:"+str(traffic))

if(flag==0):

print("Оборудование функционирует нормально.")

print("--------------------------------------------")

time.sleep(580)

# ПРИЛОЖЕНИЕ №4

import rrdtool

import tempfile

#извлечение данных RAM

result = rrdtool.fetch("RAM.rrd", "AVERAGE",'-s', str(rrdtool.first("RAM.rrd")),

'-e', str(rrdtool.last("RAM.rrd")),

'-r', "2")

#извлечение данных СPU

result2 = rrdtool.fetch("CPU.rrd", "AVERAGE",'-s', str(rrdtool.first("CPU.rrd")),

'-e', str(rrdtool.last("CPU.rrd")),

'-r', "2")

#извлечение данных Broadcast

result3 = rrdtool.fetch("Traffic\_broad.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_broad.rrd")),

'-e', str(rrdtool.last("Traffic\_broad.rrd")),

'-r', "2")

#извлечение данных Milticast

result4 = rrdtool.fetch("Traffic\_multi.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_multi.rrd")),

'-e', str(rrdtool.last("Traffic\_multi.rrd")),

'-r', "2")

#извлечение данных Unicast

result5 = rrdtool.fetch("Traffic\_uni.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_uni.rrd")),

'-e', str(rrdtool.last("Traffic\_uni.rrd")),

'-r', "2")

print("Данные CPU:")

print(result[2])

print("Данные RAM:")

print(result2[2])

print("Данные Broadcast:")

print(result3[2])

print("Данные Multicast:")

print(result4[2])

print("Данные Unicast:")

print(result4[2])

typeofgraph = input("Введите тип графика(CPU/RAM):")

time = input("Введите количество часов, за которое выдать статистику:")

#Построение графиков

if(typeofgraph=="CPU"):

fd,path=tempfile.mkstemp('.png')

rrdtool.graph("CPUstat.png",

"--imgformat","PNG",

"--height", "480",

"--width", "600",

"--step","60",

"--start", "-%sh" %time,

"--lower-limit","0",

"--upper-limit","100",

"--end","now",

"DEF:CPU=CPU.rrd:CPU:AVERAGE",

"AREA:CPU#00FF00: CPU Load")

print("Смотри файл CPUstat.png")

if(typeofgraph=="RAM"):

fd,path=tempfile.mkstemp('.png')

rrdtool.graph("RAMstat.png",

"--imgformat","PNG",

"--height", "480",

"--width", "600",

"--step","60",

"--start", "-%sh" %time,

"--lower-limit","0",

"--upper-limit","250",

"--end","now",

"DEF:RAM=RAM.rrd:CPU:AVERAGE",

"AREA:RAM#00FF00: Free RAM")

print("Смотри файл RAMstat.png")

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГБОУ ВО «Пермский государственный национальный исследовательский университет»

Физический факультет

Кафедра радиоэлектроники и защиты информации

**ДЕТЕКТОР АНОМАЛИЙ ПОВЕДЕНИЯ СЕТЕВОГО ОБОРУДОВАНИЯ**

Курсовая работа

|  |  |
| --- | --- |
|  | Работу выполнил  студент 3 курса специальности  “Информационная безопасность автоматизированных систем”  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.А. Пагин |
| Оценка работы научным руководителем  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (оценка, выставленная руководителем)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата, подпись руководителя) | Научный руководитель:  старший преподаватель кафедры радиоэлектроники и защиты информации  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.И. Моисеев |
| Оценка работы комиссией по защите курсовых работ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (оценка, выставленная комиссией)  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата, подпись председателя комиссии) |  |

Пермь, 2017

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc480451912)

[ОБЗОР ВАРИАНТОВ РЕШЕНИЯ 4](#_Toc480451913)

[Выбор протокола. 4](#_Toc480451914)

[Выбор алгоритма. 4](#_Toc480451915)

[Выбор среды разработки. 4](#_Toc480451916)

[ВЫДЕЛЕНИЕ НАИВАЖНЕЙШИХ ПОКАЗАТЕЛЕЙ СОСТОЯНИЯ РОУТЕРА. ПРОТОКОЛ SNMP. 6](#_Toc480451917)

[РАЗРАБОТКА ЭКСПЕРТНОЙ СИСТЕМЫ 7](#_Toc480451918)

[ХРАНЕНИЕ ДАННЫХ. RRD 8](#_Toc480451919)

[ОРГАНИЗАЦИЯ ВОЗМОЖНОСТЕЙ ПРОТОКОЛА SNMP И ВОЗМОЖНОСТЕЙ RRDTOOL В СРЕДЕ PYTHON. 10](#_Toc480451920)

[АЛГОРИТМ РАБОТЫ ОСНОВНОГО МОДУЛЯ. 12](#_Toc480451921)

[ТЕСТИРОВАНИЕ 13](#_Toc480451922)

[ВОЗМОЖНОСТИ ПОСТРОЕНИЯ ГРАФИКОВ И ПРОСМОТРА ДАННЫХ. 16](#_Toc480451923)

[ЗАКЛЮЧЕНИЕ 19](#_Toc480451924)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 20](#_Toc480451925)

[ПРИЛОЖЕНИЕ №1 21](#_Toc480451926)

[ПРИЛОЖЕНИЕ №2 24](#_Toc480451927)

[ПРИЛОЖЕНИЕ №3 25](#_Toc480451928)

[ПРИЛОЖЕНИЕ №4 33](#_Toc480451929)

# ВВЕДЕНИЕ

Двадцатый век уже не представить без глобальной сети Интернета. Вместе с его развитием неизбежно разрастаются и различные локальные сети, организующие доступ в интернет, либо просто для объединения большого количества устройств предприятий в единую сеть. Вместе с этим, безусловно, увеличивается и количество коммутаторов и маршрутизаторов в данных сетях. По мере подключения в сеть все новых и новых конечных пользователей, маршрутизаторам все сложнее справляться с нагрузкой, что ведет к дальнейшему увеличению их количества. Когда сеть разрастается, сетевому администратору становится все сложнее уследить за функционированием всех устройств в сети. На сегодняшний день существует большое количество приложений/дополнений, позволяющих вести мониторинг нагрузки на различные узлы сети, но, в большинстве своем, они дают лишь сухие данные, за которыми нужно постоянно следить, выявляя различные аномалии поведения сетевого оборудования, будь то излишняя нагрузка на процессор роутера или большое количество одновременно подключенных пользователей.

В данной работе будет рассмотрен процесс создания приложения, которое позволит не только вести мониторинг состояния сетевого оборудования, но и в реальном времени отслеживать различные отклонения от нормы, своевременно сообщая об этом сетевому администратору сети.

Таким, образом, цель работы – *разработать модуль для детектирования различного рода аномалий в поведении сетевого оборудования*.

Для достижения данной цели можно выделить следующие задачи:

* Выделить наиболее важные параметры, на основе которых будет происходить детектирование
* Разработать экспертную систему для данных параметров
* Объединить вышеописанные шаги в отдельном программном модуле
* Протестировать модуль в реальных условиях

# ОБЗОР ВАРИАНТОВ РЕШЕНИЯ

## Выбор протокола.

Для работы с оборудованием Cisco был разработал сетевой протокол NetFlow, предназначенный для учета сетевого трафика. Однако использование данного протокола заточено лишь под мониторинг трафика. Различные онлайн-приложения позволяют с помощью данного протокола вести мониторинг трафика, проходящего через устройство, не предлагая при этом каких-либо средств для детектирования аномалий. К тому же данный протокол поддерживается в основном только устройствами Сisco.

Протокол SNMP[1] – это протокол типа «запрос-ответ», который и позволяет посредством простых запросов получать от устройства данные метрики. Данный протокол имеет большую универсальность, будучи поддерживаемым всеми сетевыми устройствами.

Таким образом, выбран протокол SNMP.

## Выбор алгоритма.

В сети можно найти решения в сфере детектирования аномалии поведения сетевого оборудования, основанные на анализе содержимого пакетов посредством нейронных сетей. Данный подход ресурсозатратен, в достаточной степени эмпиричен и не имеет привязки к непосредственным аппаратным средствам сетевого оборудования.

Протокол SNMP позволяет получать как данные о трафике, так и данные о состоянии аппаратных элементов. Объединение данных показателей для детектирования аномалий имеет больший охват, чем подход с анализом трафика. Для детектирования аномалий в данном случае предлагается использовать жесткие алгоритмы посредством разработанной экспертной системы.

Таким образом, выбран анализ на основе данных о трафике и состоянии аппаратных элементов.

## Выбор среды разработки.

Библиотеки для протокола SNMP существуют почти для всех современных языков программирования. Учитывая факт большого количества open-source библиотек и пакетов для \*unix подобных систем, предпочтительнее выбрать операционную систему CentOS 5 вместо привычных Windows-платформ.

\*unix подобные системы поддерживают легкую установку большинства языков программирования. На сегодняшний день наиболее распространены языки с объектно-ориентированной парадигмой - С# и Python. Однако, в среднем объем кода на языке Python меньше, чем для аналогичных алгоритмов на С#. Кроме того, для языка Python существует большее количество open-source библиотек.

Таким образом выбрана ОС CentOS 5 и язык программирования Python v3.4.

# ВЫДЕЛЕНИЕ НАИВАЖНЕЙШИХ ПОКАЗАТЕЛЕЙ СОСТОЯНИЯ РОУТЕРА. ПРОТОКОЛ SNMP.

Современные маршрутизаторы содержат внутри себя достаточно большие производительные мощности – это достаточно мощные процессоры и достаточные объемы оперативной памяти. Эти два аппаратных показателя несут за собой достаточно большой объем информации о состоянии оборудования. Кроме того, необходимо вспомнить и о такой метрике, как число приходящих пакетов за единицу времени. В совокупности три данных значения дают обширных охват для мониторинга состояния роутера/маршрутизатора. Выше мы определились, что будем использовать протокол SNMP.

Для подключения к устройству по SNMP необходимы имя группы и IP-адрес устройства. Для тестирования работы приложения был выдан доступ к коммутатору Cisco Catalyst 4500 series. Подключение организуется в части программного кода (Рисунок 1):

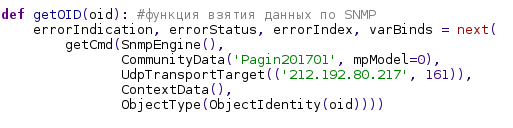


Рисунок 1

Где имя группы – Pagin201701 и адрес устройства – 212.192.80.217.

Идентификация запрашиваемых данных определяется с помощью OID’ов. Например (Рисунок 2):

image (3)

Рисунок 2

Где последовательность «.1.3.6.1.2.1.3.1.1.1.1.3.3» кодирует данные о количестве broadcast пакетов. Последняя цифра последовательности обозначает номер интерфейса, данные о котором запрашиваются. Для данной работы собирались данные с интерфейсов:

* Te 1/1-2
* Gi 2/1-48
* Gi 3/1-12

# РАЗРАБОТКА ЭКСПЕРТНОЙ СИСТЕМЫ

Ранее был сделан акцент на наиболее важных параметрах роутеров, которые мы можем получить по протоколу SNMP:

* Загрузка CPU
* Объем свободной RAM
* Количество пакетов(broadcast/unicast/multicast)

Но имея лишь эти последовательно поступающие данные, разрабатываемая система ничем не будет отличаться от обычных систем мониторинга. Для поиска и выявления различного рода аномалий необходимо разработать алгоритмы, по которым будет детектироваться «необычное» поведение оборудования.

Рассмотрим методы детектирования для значений CPU и RAM.  
Значения CPU и RAM – достаточно показательные значения, для которых можно выделить следующие крайние случаи:

* Загрузка более, чем на 80%. Что говорит о необходимости установки более мощного устройства, т.к. данное устройство вскоре может начать не справляться с предложенной нагрузкой.
* Единовременный скачок загрузки более, чем на 15%. Суточная загрузка процессора имеет некий тренд, однако, он почти не прослеживается (в течение суток на тестовом коммутаторе загрузка CPU была в среднем около 10-15%). А резкое изменение является «необычным» явлением.

Теперь рассмотрим методы детектирования для количества пакетов трафика:

* Увеличение количества пришедших пакетов на 20% в сравнении с количеством пакетом, соответствующего времени прошлой недели. Так как количество пришедших пакетов имеет значительную привязку к недельному тренду, необходимо ориентироваться на прошлые значения.
* Единовременный скачок количества пришедших пакетов более, чем на 10%. Несмотря на то, что первая метрика рассматривает привязку к тренду, необходимо рассматривать и резкие скачки в течение одного промежутка времени.

# ХРАНЕНИЕ ДАННЫХ. RRD

Для обработки и хранения данных, забираемых с сетевого оборудования по протоколу SNMP необходимо было завести какую-либо базу данных. Так как разрабатываемый программный модуль может использоваться неограниченное количество времени, обычные базы данных могут бесконечно разрастаться, что недопустимо. Поэтому были выбраны циклические базы данных RRD. Для всех собираемых данных их структура имеет аналогичный вид (Рисунок 3):

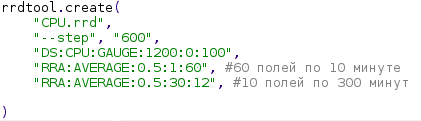


Рисунок 3

Данный пример для хранения значений загруженности CPU. Сначала создается файл для хранения значений – «CPU.rrd». Далее задается шаг для поступления новых значений – 600 секунд (10 минут). Далее задается поле, в которое будут сохраняться значения и тип данных для него – поле «CPU», тип данных – GAUGE, абсолютные значения данных. Последние 2 строки задают тип интерполяции данных – AVERAGE. Когда первое поле будет полностью заполнено, последние 30 значений из него усреднятся и будут перенесены в следующее поле, которое хранит 10 5-часовых усредненных значений. При переполнении базы, последние значения будут откидываться.

СОЗДАНИЕ ВЫБОРКИ ДЛЯ ТРАФИКА.

Как было сказано ранее, один из методов детектирования аномалий поведения сетевого оборудования опирается на данные о пришедших пакетах трафика за предыдущую неделю. Для этого было необходимо сделать отдельную выборку. Для этого был создан отдельный программный модуль (Приложение №1). В течение недели собирались данные о пришедших пакетах за каждые 10 минут. Далее использовался еще один программный модуль (Приложение №2) для приведения данных к необходимому виду. Итоговая выборка имеет вид (Рисунок 4):

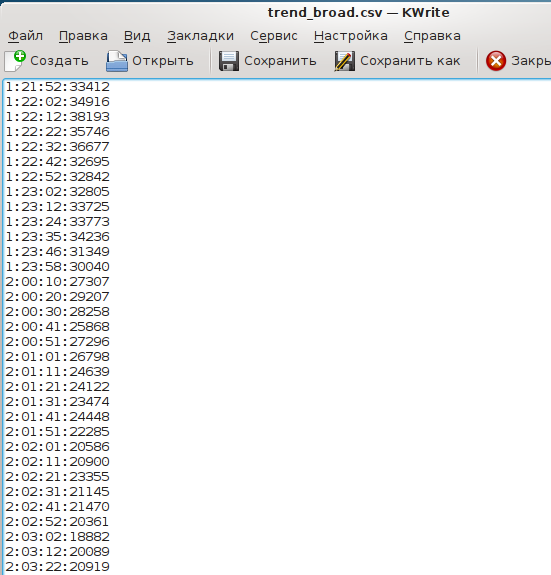


Рисунок 4

В каждой строке первая цифра кодирует день недели (Понедельник – 1, вторник – 2 и т.д.), вторая – часы, третья – минуты, четвертая – количество пакетов, пришедшие за промежуток от предыдущего значения до нового. Аналогичные наборы значений были созданы для всех трех типов трафика – broadcast/unicast/multicast.

# ОРГАНИЗАЦИЯ ВОЗМОЖНОСТЕЙ ПРОТОКОЛА SNMP И ВОЗМОЖНОСТЕЙ RRDTOOL В СРЕДЕ PYTHON.

Для разработки было решено написать консольное приложение на языке Python v.3.4[2]. Установка Python (Рисунок 5):

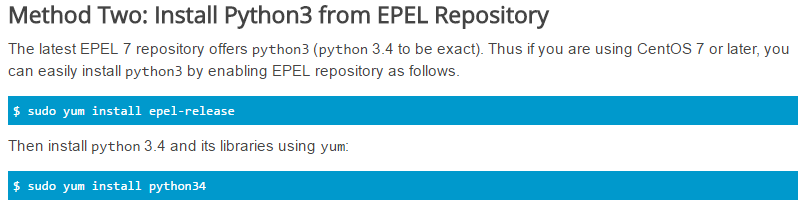


Рисунок 5

Работа с данным языком наиболее удобна в случае разработки детектора аномалий поведения сетевого оборудования, вследствие простоты установки на \*unix-подобные системы и наличии большого количества open-source библиотек.

Рассмотренный протокол SNMP и возможности работы с базами данных RRD, было необходимо перенести в среду разработки. Для этого были выбраны opensource библиотеки: «rrdtool»[4] и «pysnmp»[3].

Установка «rrdtool» происходит следующими командами (Рисунок 6):

Рисунок 6

Установка «pysnmp» происходит следующими командами (Рисунок 7):

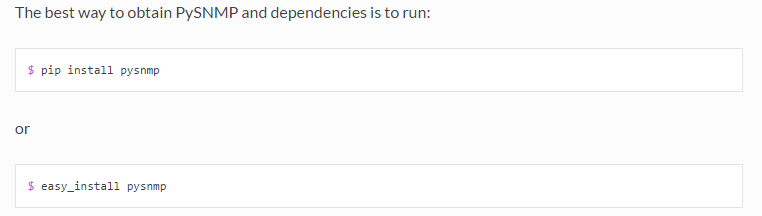


Рисунок 7

Данные библиотеки наиболее оптимизированы и имеют наиболее удобный и интуитивно понятный синтаксис, вследствие чего и были выбраны.

# АЛГОРИТМ РАБОТЫ ОСНОВНОГО МОДУЛЯ.

Основной модуль [Приложение 3] имеет следующий алгоритм работы:

1. Опрос загруженности CPU
2. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
3. Опрос количества свободной RAM
4. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
5. Опрос количества пришедших Broadсast пакетов.
6. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
7. Опрос количества пришедших Multiсast пакетов.
8. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
9. Опрос количества пришедших Uniсast пакетов.
10. Анализ полученных данных и соответствующий вывод в случае положительного результата детектирования аномалии
11. Ожидание времени до следующего опроса
12. Переход к пункту 1.

# ТЕСТИРОВАНИЕ

При запуске приложение [Приложение 3] автоматически создает необходимые RRD базы данных и начинает захват данных по протоколу SNMP. В случае обнаружения какой-либо аномалии, пользователю сообщается об этом в окне приложения и заносится соответствующая записать в файл логирования «logging.log».

Для тестирования данный модуль был запущен. В течение первых 20 минут его работа была при обычных условиях нагрузки на коммутатор. Далее на 3 машинах в данной сети были запущены команды (Рисунок 8):

image (1)

Рисунок 8

Данная команда запускает бесконечный ping-flood пакетами по 3600 байт на тестируемое устройство.

Вывод программы при этом (Рисунок 9):

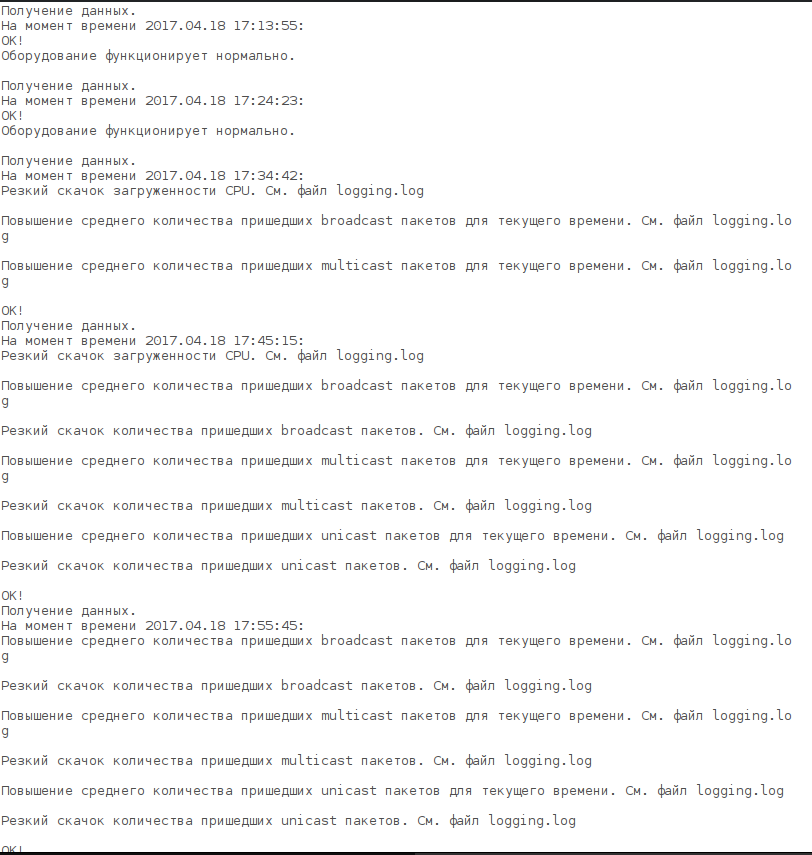


Рисунок 9

Как можно заметить, первые 20 минут работы все было хорошо. Рассмотрим последующие моменты времени отдельно:

* 17:34:42 - программа выявила скачок загруженности CPU и повышенные количества broadcast и multicast пакетов.
* 17:45:15 - все введенные в экспертную систему метрики сработали. Однако, как можно заметить, на загруженность RAM данный тест влияния не имел.
* 17:55:45 – загруженность CPU стабилизировалась и при этом была меньше 85%, поэтому в выводе программы исчезло предупреждение о скачке нагрузки на CPU.

Все «происшествия» вносятся в файл “logging.log”. После данного запуска программы содержимое файла выглядит так (Рисунок 10):

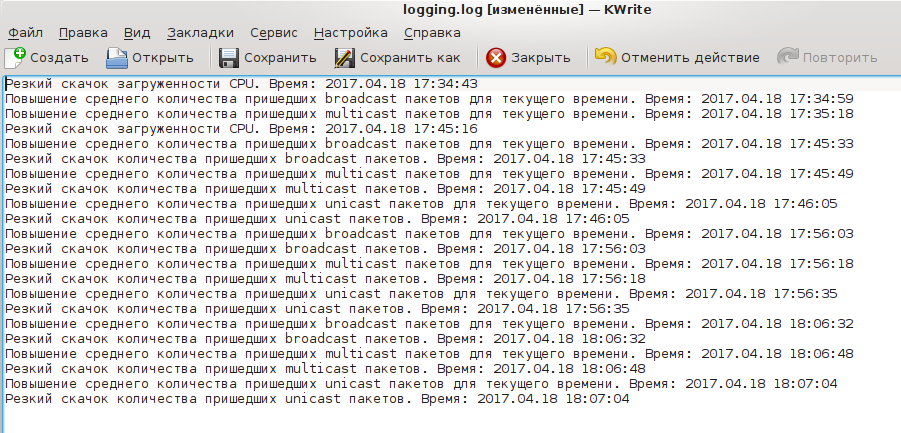


Рисунок 10

Видим, что все аномалии успешно залогированы. Таким образом, разработанный программный модуль успешно детектирует аномалии, введенные в экспертную систему.

# ВОЗМОЖНОСТИ ПОСТРОЕНИЯ ГРАФИКОВ И ПРОСМОТРА ДАННЫХ.

Для дополнительного удобства анализа данных пользователем был создан отдельный модуль [Приложение №4]. При запуске данный модуль выдает содержимое всех RRD баз данных (Рисунок 11):

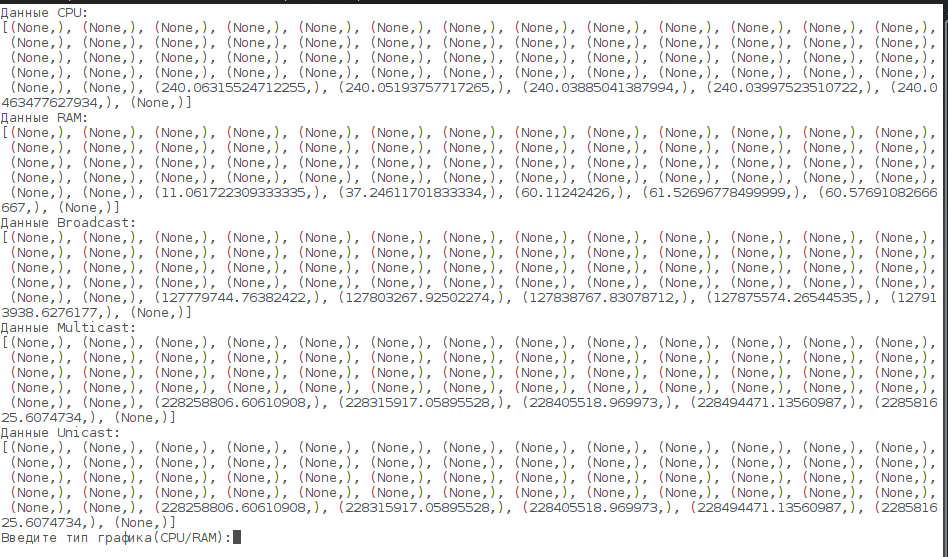


Рисунок 11

Далее организован диалог с пользователем, в котором предлагается выбрать данные CPU или RAM и соответствующее количество часов, за которое необходимо построить график (Рисунок 12):

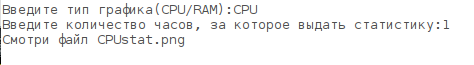


Рисунок 12

При удачном построении, модуль предлагает посмотреть созданный графический файл. Создадим и просмотрим график для данных из блока тестирования для CPU (Рисунок 13) и RAM (Рисунок 14):

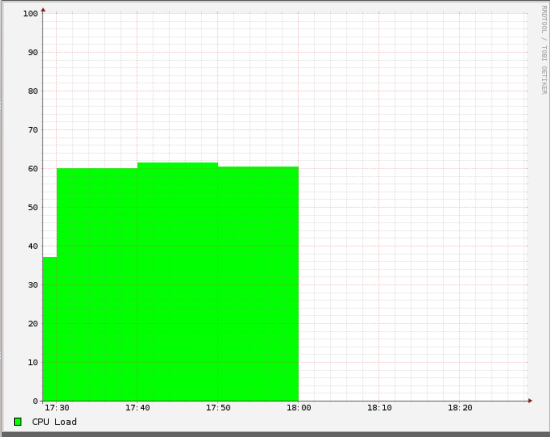


Рисунок 13

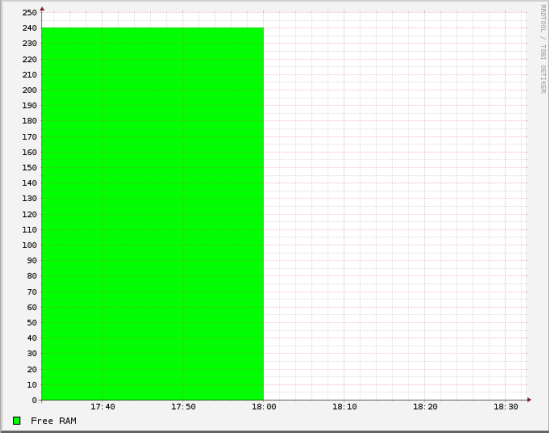


Рисунок 14

В начале графика CPU хорошо видно тот самый скачок загруженности с 36% до 60%, который был задетектирован программой. При этом, как мы видим, загруженность RAM никак не менялась.

Построение графиков успешно функционирует и помогает пользователю наглядно увидеть аномалии в различных параметрах сетевого оборудования.

# ЗАКЛЮЧЕНИЕ

Как было показано, разработанное приложение помогает системному администратору своевременно выявлять те или иные отклонения в поведении сетевого оборудования. Данная разработка в перспективе может сильно расширятся, включая в себя все больше показателей, а также анализ данных посредством нейросетей. В ходе работы были встречены некоторые проблемы, которые могут быть успешно решены более долгим созданием опорной выборки для данных. В целом, данная работа включила в себя немало областей – от анализа данных до устройства сетевого оборудования и программирования на Python.

Как вывод, можно сказать о том, что данное приложение может помочь в детектировании аномалий уже сейчас, что, несомненно, является хорошим окончанием для данной работы.

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. В. Олифер, Н. Олифер «Компьютерные сети», изд. «Питер», 2016, стр. 731
2. How to install Python3 on CentOS [Online] http://ask.xmodulo.com/install-python3-centos.html
3. Pysnmp Quick Start. [Online] http://pysnmp.sourceforge.net/quick-start.html
4. Rrdpython. [Online] http://oss.oetiker.ch/rrdtool/prog/rrdpython.en.html

# ПРИЛОЖЕНИЕ №1

#! -\*- coding: utf-8 -\*-

from pysnmp.hlapi import \*

import time

from datetime import datetime

def getOID(oid):

errorIndication, errorStatus, errorIndex, varBinds = next(

getCmd(SnmpEngine(),

CommunityData('Pagin201701', mpModel=0), #подключение к роутеру

UdpTransportTarget(('212.192.80.217', 161)),

ContextData(),

ObjectType(ObjectIdentity(oid))))

if errorIndication:

print(errorIndication)

elif errorStatus:

print('%s at %s' % (errorStatus.prettyPrint(),

errorIndex and varBinds[int(errorIndex) - 1][0] or '?'))

else:

for varBind in varBinds:

print(' = '.join([x.prettyPrint() for x in varBind]))

return str(varBind[1])

i=0

while i<1:

#Обработка для multicast пакетов

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.3"))

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #

j=j+1

new\_file = open("trand\_multi.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

#Обработка для broadcast пакетов

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.3"))

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

new\_file = open("trand\_broad.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

#Обработка для unicast пакетов

traffic=int(getOID(".1.3.6.1.2.1.2.2.1.11.3"))

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

new\_file = open("trand\_uni.csv", 'a')

new\_file.write(datetime.strftime(datetime.now(), "%H:%M")+":"+str(traffic) +"\n")

new\_file.close()

time.sleep(580)

# ПРИЛОЖЕНИЕ №2

import time

from datetime import datetime

fileob = open('trand\_uni.csv') #открытие файла, созданного предыдущим модулем

arr = fileob.read().split("\n")

fileob.close()

i=1

print(len(arr))

j=1 #начиная с понедельника

while i<len(arr)-1:

string=arr[i].split(":")

string2=arr[i-1].split(":")

print(string)

new\_file = open("trend\_uni.csv", 'a')

#вычисление и запись числа пришедших пакетов за промежуток времени

new\_file.write(str(j)+":"+string[0]+":"+string[1]+":"+str(int(string[2])-int(string2[2]))+"\n")

razn=int(string2[0][0])-int(string[0][0])

if(razn > 1): #проверка наступления нового дня

j+=1

new\_file.close()

i+=1

print(“ОК!”)

# ПРИЛОЖЕНИЕ №3

#! -\*- coding: utf-8 -\*-

from pysnmp.hlapi import \*

import time

from datetime import datetime

import rrdtool

def getOID(oid): #функция взятия данных по SNMP

errorIndication, errorStatus, errorIndex, varBinds = next(

getCmd(SnmpEngine(),

CommunityData('Pagin201701', mpModel=0),

UdpTransportTarget(('212.192.80.217', 161)),

ContextData(),

ObjectType(ObjectIdentity(oid))))

if errorIndication:

print(errorIndication)

elif errorStatus:

print('%s at %s' % (errorStatus.prettyPrint(),

errorIndex and varBinds[int(errorIndex) - 1][0] or '?'))

return str(varBinds[0][1])

rrdtool.create(

"CPU.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"RAM.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:300",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_broad.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_multi.rrd",

# "--start", "now",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

rrdtool.create(

"Traffic\_uni.rrd",

"--step", "600",

"DS:CPU:GAUGE:1200:0:100000000000",

"RRA:AVERAGE:0.5:1:60", #60 полей по 10 минуте

"RRA:AVERAGE:0.5:30:12", #10 полей по 300 минут

)

i=0

while i<1:

print("Получение данных.")

start = datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")

print("На момент времени "+start+":")

flag=0

#взять массив элементов CPU.rrd

result = rrdtool.fetch("CPU.rrd", "AVERAGE",'-s', str(rrdtool.first("CPU.rrd")),

'-e', str(rrdtool.last("CPU.rrd")),

'-r', "2")

CPU\_used = getOID(".1.3.6.1.4.1.9.2.1.57.0")

#проверка на скачок заггруженности CPU

if(result[2][-2][0]):

if((abs(int(result[2][-2][0]) - int(CPU\_used)))>10):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок загруженности СPU. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок загруженности СPU. См. файл logging.log\n")

new\_file.close()

#Проверка на повышенную загруженность CPU

if(int(CPU\_used)>85):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Высокий процент загруженности СPU: "+ CPU\_used+"%. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Высокий процент загруженности СPU: "+ CPU\_used+" См. файл logging.log\n")

new\_file.close()

rrdtool.update("CPU.rrd", "N:"+str(CPU\_used))

#взять массив элементов RAM.rrd

result = rrdtool.fetch("RAM.rrd", "AVERAGE",'-s', str(rrdtool.first("RAM.rrd")),

'-e', str(rrdtool.last("RAM.rrd")),

'-r', "2")

RAM\_free\_bt=getOID(".1.3.6.1.4.1.9.2.1.8.0")

Ram\_free\_mb = float(RAM\_free\_bt)/(1024\*1024)

#Проверка на скачок загруженности RAM

if(result[2][-2][0]):

if(abs(int(result[2][-2][0])-int(Ram\_free\_mb))>5):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкое уменьшение размера свободной памяти RAM. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкое уменьшение размера свободной памяти RAM. См. файл logging.log\n")

new\_file.close()

#Проверка на повышенную загруженность RAM

if(Ram\_free\_mb<100):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Малый объем свободной RAM: "+ Ram\_free\_mb+" mb. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Малый объем свободной RAM: "+ Ram\_free\_mb+" mb. См. файл logging.log\n")

new\_file.close()

rrdtool.update("RAM.rrd", "N:"+str(Ram\_free\_mb))

result = rrdtool.fetch("Traffic\_broad.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_broad.rrd")),

'-e', str(rrdtool.last("Traffic\_broad.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.3")) #broadcast

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.3."+str(j)))

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_broad.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

print(str(normal) + " " + str(int(traffic)-int(result[2][-2][0])))

if((((int(traffic)-int(result[2][-2][0]))-int(normal))/int(normal))>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших broadcast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших broadcast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших broadcast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших broadcast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_broad.rrd", "N:"+str(traffic))

result = rrdtool.fetch("Traffic\_multi.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_multi.rrd")),

'-e', str(rrdtool.last("Traffic\_multi.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.3")) #multicast

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #multi/broad/uni

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.31.1.1.1.2."+str(j))) #multi/broad/uni

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_multi.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

if((((int(traffic)-int(result[2][-2][0]))-normal)/normal)>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших multicast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших multicast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших multicast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших multicast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_multi.rrd", "N:"+str(traffic))

result = rrdtool.fetch("Traffic\_uni.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_uni.rrd")),

'-e', str(rrdtool.last("Traffic\_uni.rrd")),

'-r', "2")

traffic=int(getOID(".1.3.6.1.2.1.2.2.1.11.3")) #unicast

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11.4"))

j=9

while(j<21):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

j=57

while(j<69):

traffic+=int(getOID(".1.3.6.1.2.1.2.2.1.11."+str(j)))

j=j+1

#Проверка на повышенное число пришедших пакетов

if(result[2][-2][0]):

new\_file = open("trend\_uni.csv", 'r')

line=new\_file.readline()

while((line.find(str(datetime.now().weekday()+1)+":"+str(datetime.strftime(datetime.now(), "%H"))))):

line=new\_file.readline()

while(int(datetime.strftime(datetime.now(), "%M"))>int(line[5:7])):

line=new\_file.readline()

normal=int(line[8:])

new\_file.close()

if((((int(traffic)-int(result[2][-2][0]))-normal)/normal)>0.2):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Повышение среднего количества пришедших unicast пакетов для текущего времени. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Повышение среднего количества пришедших unicast пакетов для текущего времени. См. файл logging.log\n")

new\_file.close()

#Проверка на скачок числа пришедших пакетов

if(result[2][-3][0]):

if(((int(traffic)-int(result[2][-2][0]))-(int(result[2][-2][0])-int(result[2][-3][0])))/(int(result[2][-2][0])-int(result[2][-3][0]))>0.1):

flag=1

new\_file = open("logging.log", 'a')

new\_file.write("Резкий скачок количества пришедших unicast пакетов. Время: "+ datetime.strftime(datetime.now(), "%Y.%m.%d %H:%M:%S")+"\n")

print("Резкий скачок количества пришедших unicast пакетов. См. файл logging.log\n")

new\_file.close()

rrdtool.update("Traffic\_uni.rrd", "N:"+str(traffic))

if(flag==0):

print("Оборудование функционирует нормально.")

print("--------------------------------------------")

time.sleep(580)

# ПРИЛОЖЕНИЕ №4

import rrdtool

import tempfile

#извлечение данных RAM

result = rrdtool.fetch("RAM.rrd", "AVERAGE",'-s', str(rrdtool.first("RAM.rrd")),

'-e', str(rrdtool.last("RAM.rrd")),

'-r', "2")

#извлечение данных СPU

result2 = rrdtool.fetch("CPU.rrd", "AVERAGE",'-s', str(rrdtool.first("CPU.rrd")),

'-e', str(rrdtool.last("CPU.rrd")),

'-r', "2")

#извлечение данных Broadcast

result3 = rrdtool.fetch("Traffic\_broad.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_broad.rrd")),

'-e', str(rrdtool.last("Traffic\_broad.rrd")),

'-r', "2")

#извлечение данных Milticast

result4 = rrdtool.fetch("Traffic\_multi.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_multi.rrd")),

'-e', str(rrdtool.last("Traffic\_multi.rrd")),

'-r', "2")

#извлечение данных Unicast

result5 = rrdtool.fetch("Traffic\_uni.rrd", "AVERAGE",'-s', str(rrdtool.first("Traffic\_uni.rrd")),

'-e', str(rrdtool.last("Traffic\_uni.rrd")),

'-r', "2")

print("Данные CPU:")

print(result[2])

print("Данные RAM:")

print(result2[2])

print("Данные Broadcast:")

print(result3[2])

print("Данные Multicast:")

print(result4[2])

print("Данные Unicast:")

print(result4[2])

typeofgraph = input("Введите тип графика(CPU/RAM):")

time = input("Введите количество часов, за которое выдать статистику:")

#Построение графиков

if(typeofgraph=="CPU"):

fd,path=tempfile.mkstemp('.png')

rrdtool.graph("CPUstat.png",

"--imgformat","PNG",

"--height", "480",

"--width", "600",

"--step","60",

"--start", "-%sh" %time,

"--lower-limit","0",

"--upper-limit","100",

"--end","now",

"DEF:CPU=CPU.rrd:CPU:AVERAGE",

"AREA:CPU#00FF00: CPU Load")

print("Смотри файл CPUstat.png")

if(typeofgraph=="RAM"):

fd,path=tempfile.mkstemp('.png')

rrdtool.graph("RAMstat.png",

"--imgformat","PNG",

"--height", "480",

"--width", "600",

"--step","60",

"--start", "-%sh" %time,

"--lower-limit","0",

"--upper-limit","250",

"--end","now",

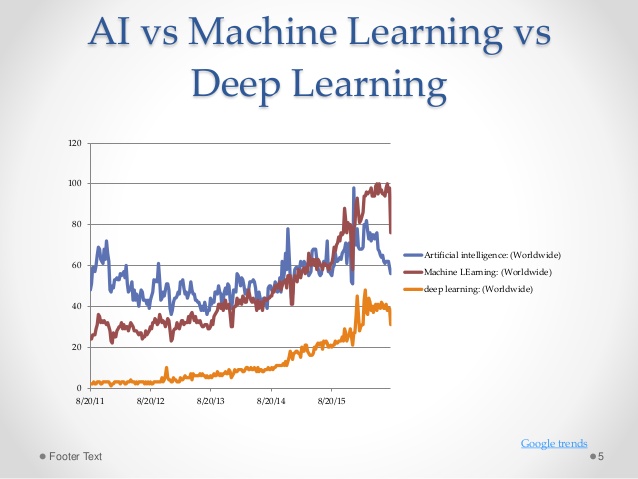
"DEF:RAM=RAM.rrd:CPU:AVERAGE",

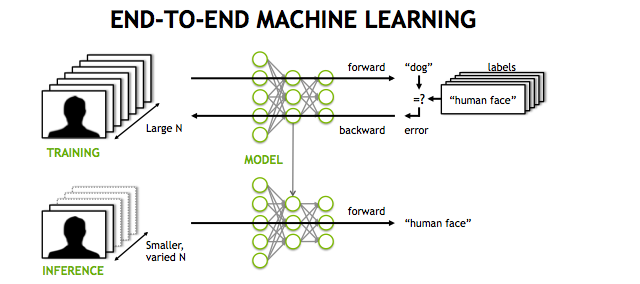
"AREA:RAM#00FF00: Free RAM")

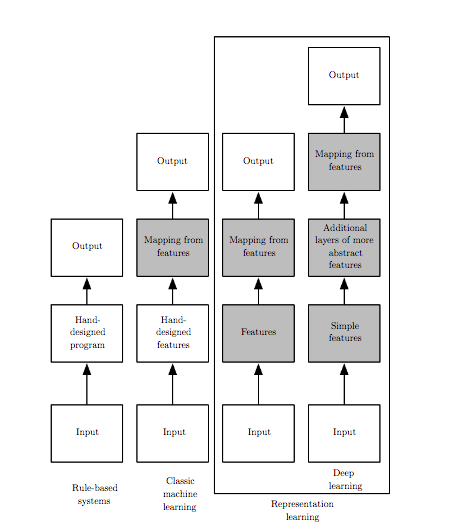
print("Смотри файл RAMstat.png")

https://github.com/mmehas/neurohacker/blob/master/neurohacker.ipynb

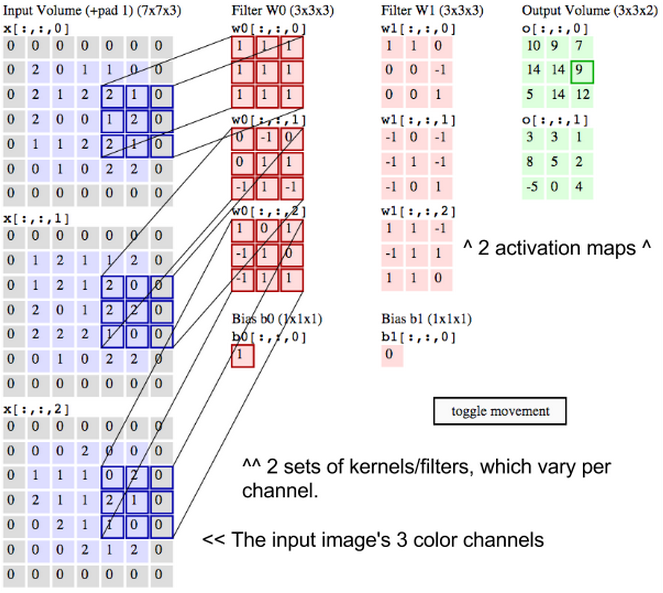
Для обработки и классификации изображений посредством нейронных сетей мы предлагаем использовать свободную библиотеку от Google – TensorFlow. При этом упор будем делать на использование DeepLearning (Глубинное обучение).

Для Аргументации:  


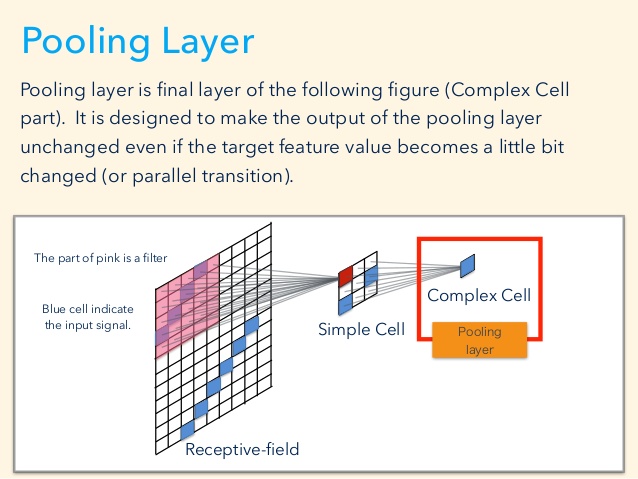




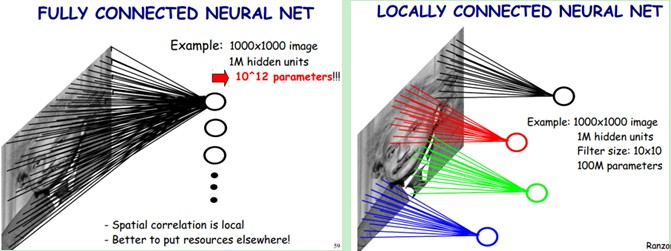
В сети можно найти немало примеров создания подобных нейронных сетей для распознавания лиц, цифр, дорожных знаков. В целом, идея будет той же:  
 Создание 5-6 конвуляционных слоев, каждый из которых будет заостряться на все более крупных элементах(точка->линия->совокупность линий, похожая на перелом). Учитывая одноцветность рентгеновских снимков, предлагается сразу перевести изображения в черно-белую гамму, что так же приведет и к уменьшению требуемых аппаратных ресурсов для обучения.



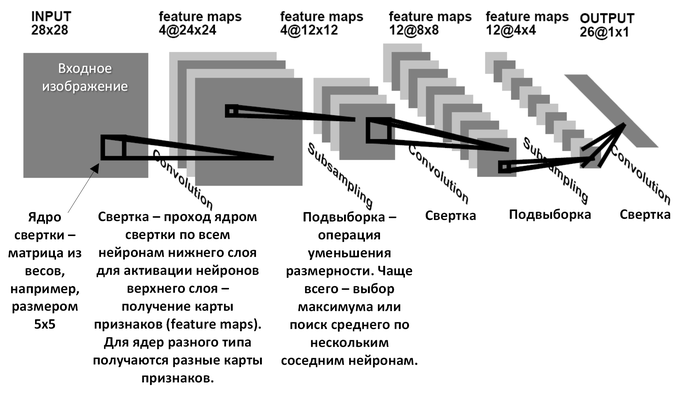
Между конвуляционными слоями необходимо добавлять пулинговые слои, чтобы выделять некоторые особоважные признаки той или иной части изображения



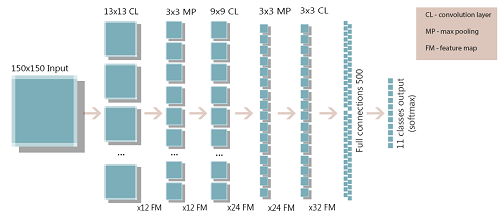
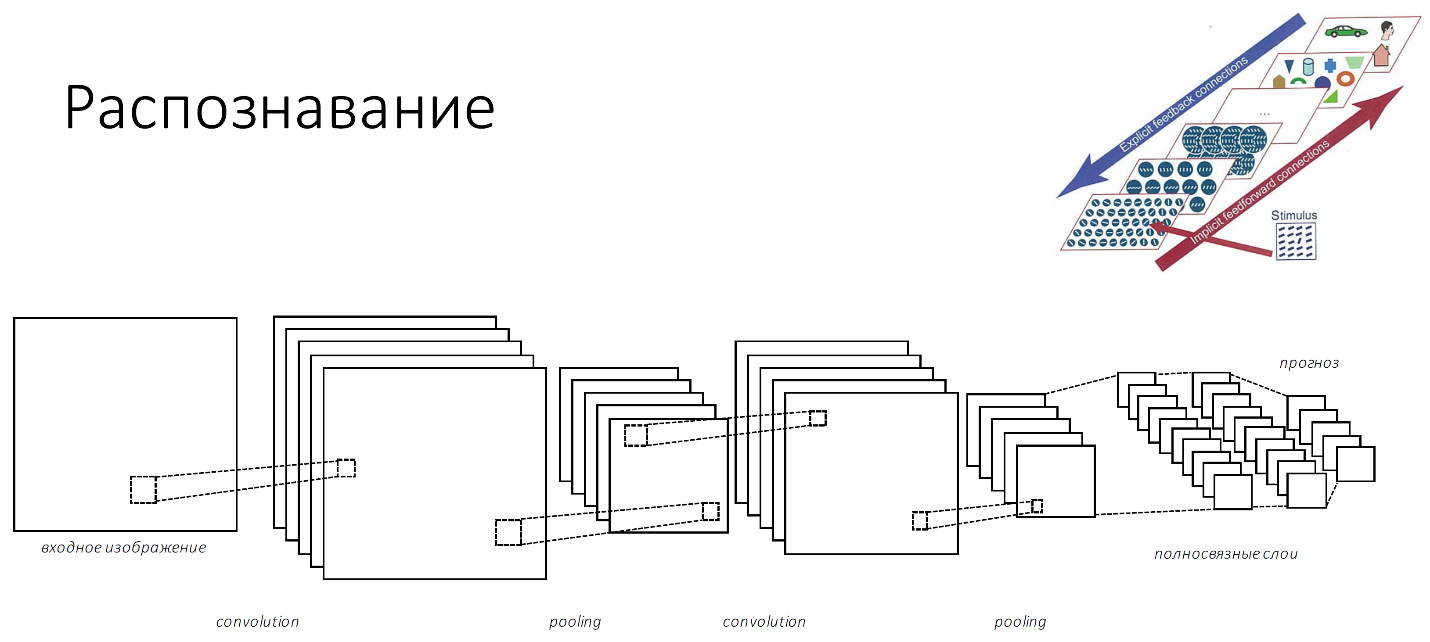
Чтобы соединить все результаты воедино, в конце должны использоваться несколько полносвязных слоев. Перед этим используется функция flatten, помещающая трехмерную абстракцию предыдущих шагов в одну плоскость, что и обеспечивает полную связность выборки.



Появившаяся проблема

Перед обучением, все изображения должны быть приведены к единой размерности. 

https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%91%D1%80%D1%82%D0%BE%D1%87%D0%BD%D0%B0%D1%8F\_%D0%BD%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F\_%D1%81%D0%B5%D1%82%D1%8C

*Layer 1: Convolutional. Input = 200x200x1. Output = 100x100x6(Relu)*

*Layer 2: Pooling. Input = 100x100x6 Output = 60x60x6*

*Layer 3: Convolutional. Input = 60x60x6. Output = 50x50x6*

*Layer 4: Pooling. Input = 50x50x6 Output = 30x30x6*

*Layer 5: Convolutional. Input = 30x30x6 Output = 15x15x10.*

*Layer 6: Pooling. Input = 15x15x10. Output = 10x10x10.*

*Flatten. Input = 10x10x10. Output = 1000.*

*Layer 7: Fully Connected. Input = 360. Output = 128.(sigmoida)*

*Layer 8: Fully Connected. Input = 128. Output = 64.*

*Dropout layer*

*Layer 9: Fully Connected. Input = 64. Output = 43*

Нужен сукперкампуктер. Спасибо геймерам, сейчас суперкампукетер может позводлить себе лох. Спасибо за видеокарты.

->параллельная нейронная сеть, принимающая на вход результаты анализов(числовой вектор) – данная сеть обычная, полносвязная(для девочек, даже с эконома делали). Сигмойда  
->Далее выходные данные этоих сетей складываем с дополнительными показателями(результаты анализов) ReLU + пара слов про эпохи и смещение весов.

Топологию этой сети следует установить эмпирически, т.к. для обучения этой секти требуется гораджздо меньше вычислетеьных мощностей, чем для нейронных сетенй для мальчиков.

**Выборка:**

**Нужна большая хихи**. В машинном обучении выборка имеет большое значение. От того, насколько кач будет выборка , напрямую зависит качество работы всей сети. Проблема – это данные закрытые. + данные надо специально гоотовить. Выборка будет оч дорогьй ефаймать. Нужен спец медик + серьехная работа аналистичекая, кк можно эти анализы классифицировать. Т.е. самое дорогое и вадное в этом проекете- выборка.

Начало- алгоритмы оч часто используются. Фейсбук там с 95 % точностью распощнает лицо. Пермский стартап успешно применяет нейросети в медицинской сфере для анализа данных с мелицинских прборов. Но нейросети не для диагноза, а для прогнозироания метрик тела по косвенным пирзнакам. Точность больше, чем у жеских алгоритмов.   
  
<http://docplayer.ru/35920310-Raspoznavanie-tuberkuleza-na-flyuorograficheskih-snimkah-s-pomoshchyu-glubokih-neyronnyh-setey.html> - успешно определЯет тубиков   
  
презентация там

MYCIN – работает

<http://masters.donntu.org/2009/fvti/andrianova/library/neuro/default.htm>

доступ только у специалистов, чтобы люди не лечились от херни

In [ ]:

file\_url = 'https://www.dropbox.com/s/9u5pmiijvd0mi8p/signs\_dataset.zip'

in the terminal:

git clone <https://github.com/mmehas/neurohacker>

cd neurohacker

wget <https://www.dropbox.com/s/9u5pmiijvd0mi8p/signs_dataset.zip>

## Step 0: Load The Data

In [ ]:

*# Load pickled data*

**import** **pickle**

**import** **os**

*# TODO: Fill this in based on where you saved the training and testing data*

data\_folder='traffic-signs-data'

**import** **zipfile**

zip\_ref = zipfile.ZipFile("signs\_dataset.zip", 'r')

zip\_ref.extractall(data\_folder)

zip\_ref.close()

training\_file = os.path.join(data\_folder, 'train\_2.p')

testing\_file = os.path.join(data\_folder, 'test\_2.p')

**with** open(training\_file, mode='rb') **as** f:

train = pickle.load(f)

**with** open(testing\_file, mode='rb') **as** f:

test = pickle.load(f)

X\_train\_raw, y\_train\_raw = train['features'], train['labels']

X\_test, y\_test = test['features'], test['labels']

## Step 1: Dataset Summary & Exploration

The pickled data is a dictionary with 4 key/value pairs:

* 'features' is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).
* 'labels' is a 1D array containing the label/class id of the traffic sign. The file signnames.csv contains id -> name mappings for each id.
* 'sizes' is a list containing tuples, (width, height) representing the the original width and height the image.
* 'coords' is a list containing tuples, (x1, y1, x2, y2) representing coordinates of a bounding box around the sign in the image. **THESE COORDINATES ASSUME THE ORIGINAL IMAGE. THE PICKLED DATA CONTAINS RESIZED VERSIONS (32 by 32) OF THESE IMAGES**

Complete the basic data summary below.

In [ ]:

n\_train = len(X\_train\_raw)

n\_test = len(X\_test)

image\_shape = (len(X\_train\_raw[0,0]), len(X\_train\_raw[0]))

n\_classes = max(y\_train\_raw) + 1

**print**("Number of training examples =", n\_train)

**print**("Number of testing examples =", n\_test)

**print**("Image data shape =", image\_shape)

**print**("Number of classes =", n\_classes)

Visualize the German Traffic Signs Dataset using the pickled file(s). This is open ended, suggestions include: plotting traffic sign images, plotting the count of each sign, etc.

The [Matplotlib](http://matplotlib.org/) [examples](http://matplotlib.org/examples/index.html) and [gallery](http://matplotlib.org/gallery.html) pages are a great resource for doing visualizations in Python.

**NOTE:** It's recommended you start with something simple first. If you wish to do more, come back to it after you've completed the rest of the sections.

In [ ]:

*### Data exploration visualization goes here.*

*### Feel free to use as many code cells as needed.*

**import** **matplotlib.pyplot** **as** **plt**

**from** **mpl\_toolkits.axes\_grid1** **import** ImageGrid

*# Visualizations will be shown in the notebook.*

%matplotlib inline

### Let's look at a random sample of images to get sense of the dataset

In [ ]:

**import** **numpy** **as** **np**

**def** show\_images(sample):

cnt = 0

fig = plt.figure(1, (30., 30.))

sample\_size = len(sample)

grid = ImageGrid(fig, 111, *# similar to subplot(111)*

nrows\_ncols=(int(sample\_size / 3) + 1, 3), *# creates 2x2 grid of axes*

axes\_pad=0.1, *# pad between axes in inch.*

)

**for** i **in** range(sample\_size):

grid[i].imshow(sample[i]) *# The AxesGrid object work as a list of axes.*

plt.show()

sample\_size = 50

sample = X\_train\_raw[np.random.choice(n\_train, sample\_size, replace=False)]

show\_images(sample)

A sample of images shows that we're having traffic signs under various lightning conditions and various angles. Some are also are looking more blurred than others.

### How many images do we have per class?

In [ ]:

**import** **collections**

class\_count = collections.Counter(y\_train\_raw)

plt.bar(range(n\_classes), sorted(class\_count.values()))

We can see that there are several main classes that about 40 % of classes are occupying a huge portion of a dataset, but anyway we're having at least 200 images per class which generally is not that bad for learning.

## Step 2: Design and Test a Model Architecture

Design and implement a deep learning model that learns to recognize traffic signs. Train and test your model on the [German Traffic Sign Dataset](http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset).

There are various aspects to consider when thinking about this problem:

* Neural network architecture
* Play around preprocessing techniques (normalization, rgb to grayscale, etc)
* Number of examples per label (some have more than others).
* Generate fake data.

Here is an example of a [published baseline model on this problem](http://yann.lecun.com/exdb/publis/pdf/sermanet-ijcnn-11.pdf). It's not required to be familiar with the approach used in the paper but, it's good practice to try to read papers like these.

### Implementation

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

* Try different preprocessing techniques (colors, normalization)
* Data augmentations (shifts, rotations, whitening)
* Various convnet architectures: try various number of layers, change number of filters, etc.

In [ ]:

*# preprocessing*

**def** rgb2gray(rgb):

r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]

gray = 0.2989 \* r + 0.5870 \* g + 0.1140 \* b

**return** gray

**def** preprocess\_image(image):

image = image.astype('float32')

image = rgb2gray(image)

*#image -= image.mean()*

*#image /= image.std()*

**return** np.reshape(image, (image.shape[0], image.shape[1], 1))

X\_train\_preprocessed = np.asarray([preprocess\_image(image) **for** image **in** X\_train\_raw])

X\_test\_preprocessed = np.asarray([preprocess\_image(image) **for** image **in** X\_test])

In [ ]:

**def** augment\_data(X, y):

*## insert augmentations here*

**return** (X, y)

*### Generate additional data (OPTIONAL!)*

*### and split the data into training/validation/testing sets here.*

*### Feel free to use as many code cells as needed.*

**from** **sklearn.cross\_validation** **import** train\_test\_split

X\_train\_augmented, y\_train\_augmented = augment\_data(X\_train\_preprocessed, y\_train\_raw)

**print**(len(X\_train\_augmented))

In [ ]:

*### Define your architecture here.*

*### Feel free to use as many code cells as needed.*

**import** **tensorflow** **as** **tf**

**import** **random**

**from** **sklearn.utils** **import** shuffle

keep\_prob = tf.placeholder(tf.float32)

**from** **tensorflow.contrib.layers** **import** flatten

**def** convnet(x, model\_name):

*# Arguments used for tf.truncated\_normal, randomly defines variables for the weights and biases for each layer*

mu = 0

sigma = 0.1

*# Layer 1: Convolutional. Input = 32x32x1. Output = 28x28x6*

conv1\_W = tf.Variable(tf.truncated\_normal(shape=(5, 5, 1, 6), mean = mu, stddev = sigma))

conv1\_b = tf.Variable(tf.zeros(6))

conv1 = tf.nn.conv2d(x, conv1\_W, strides=[1, 1, 1, 1], padding='VALID') + conv1\_b

conv1 = tf.nn.relu(conv1)

*# Pooling. Input = 28x28x6 Output = 14x14x10.*

conv1 = tf.nn.max\_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

*# Layer 2: Convolutional. Output = 12x12x10.*

conv2\_W = tf.Variable(tf.truncated\_normal(shape=(3, 3, 6, 10), mean = mu, stddev = sigma))

conv2\_b = tf.Variable(tf.zeros(10))

conv2 = tf.nn.conv2d(conv1, conv2\_W, strides=[1, 1, 1, 1], padding='VALID') + conv2\_b

conv2 = tf.nn.relu(conv2)

*# Pooling. Input = 12x12x10. Output = 6x6x10.*

conv2 = tf.nn.max\_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

*# Flatten. Input = 6x6x10. Output = 360.*

fc0 = flatten(conv2)

*# Layer 4: Fully Connected. Input = 360. Output = 128.*

fc1\_W = tf.Variable(tf.truncated\_normal(shape=(6\*6\*10, 128), mean = mu, stddev = sigma))

fc1\_b = tf.Variable(tf.zeros(128))

fc1 = tf.matmul(fc0, fc1\_W) + fc1\_b

fc1 = tf.nn.relu(fc1)

*# Layer 5: Fully Connected. Input = 128. Output = 64.*

fc2\_W = tf.Variable(tf.truncated\_normal(shape=(128, 64), mean = mu, stddev = sigma))

fc2\_b = tf.Variable(tf.zeros(64))

fc2 = tf.matmul(fc1, fc2\_W) + fc2\_b

fc2 = tf.nn.relu(fc2)

*# Dropout layer*

h\_fc2\_drop = tf.nn.dropout(fc2, keep\_prob)

*# Layer 6: Fully Connected. Input = 64. Output = 43*

fc3\_W = tf.Variable(tf.truncated\_normal(shape=(64, 43), mean = mu, stddev = sigma))

fc3\_b = tf.Variable(tf.zeros(43))

logits = tf.matmul(h\_fc2\_drop, fc3\_W) + fc3\_b

**return** logits

In [ ]:

**from** **sklearn.cross\_validation** **import** train\_test\_split

**import** **tensorflow** **as** **tf**

**class** **TrainConfiguration**:

**def** \_\_init\_\_(

self,

name,

model,

optimizer = tf.train.AdamOptimizer(learning\_rate = 0.001),

learning\_rate = 0.001,

epochs = 100,

validation\_size = 0.05,

batch\_size = 128,

X\_train = X\_train\_augmented,

y\_train = y\_train\_augmented,

X\_test = X\_test\_preprocessed,

y\_test = y\_test

):

self.name = name

self.model = model

self.optimizer = optimizer

self.learning\_rate = learning\_rate

self.epochs = epochs

self.batch\_size = batch\_size

self.X\_train, self.X\_validation, self.y\_train, self.y\_validation = \

train\_test\_split(X\_train, y\_train, test\_size = validation\_size, random\_state = 42)

self.X\_test = X\_test

self.y\_test = y\_test

self.x = tf.placeholder(tf.float32, (None, 32, 32, 1))

self.y = tf.placeholder(tf.int32, (None))

self.one\_hot\_y = tf.one\_hot(self.y, 43)

self.predictions = model(self.x, self.name)

cross\_entropy = tf.nn.softmax\_cross\_entropy\_with\_logits(logits = self.predictions, labels = self.one\_hot\_y)

cross\_entropy\_loss = tf.reduce\_mean(cross\_entropy)

optimizer = tf.train.AdamOptimizer(learning\_rate = self.learning\_rate)

self.training\_operation = optimizer.minimize(cross\_entropy\_loss)

In [ ]:

**def** evaluate(config, X\_data, y\_data):

num\_examples = len(X\_data)

total\_accuracy = 0

sess = tf.get\_default\_session()

correct\_prediction = tf.equal(tf.argmax(config.predictions, 1), tf.argmax(config.one\_hot\_y, 1))

accuracy\_operation = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

**for** offset **in** range(0, num\_examples, config.batch\_size):

batch\_x, batch\_y = X\_data[offset:offset+config.batch\_size], y\_data[offset:offset+config.batch\_size]

accuracy = sess.run(accuracy\_operation, feed\_dict={config.x: batch\_x, config.y: batch\_y, keep\_prob: 1.0})

total\_accuracy += (accuracy \* len(batch\_x))

**return** total\_accuracy / num\_examples

**def** train\_and\_evaluate(config):

global\_config = config

**print**("Starting training with config " + config.name)

saver = tf.train.Saver()

**with** tf.Session() **as** sess:

**print**("Running session")

sess.run(tf.global\_variables\_initializer())

num\_examples = len(config.X\_train)

**print**("Training on {} examples...".format(num\_examples))

**for** i **in** range(config.epochs):

X\_train, y\_train = shuffle(config.X\_train, config.y\_train)

**for** offset **in** range(0, num\_examples, config.batch\_size):

end = offset + config.batch\_size

batch\_x, batch\_y = X\_train[offset:end], y\_train[offset:end]

sess.run(config.training\_operation, feed\_dict={config.x: batch\_x, config.y: batch\_y, keep\_prob: 0.5})

validation\_accuracy = evaluate(config, config.X\_validation, config.y\_validation)

**if** i % 10 == 0:

**print**("EPOCH {} ...".format(i+1))

**print**("Validation Accuracy = {:.3f}".format(validation\_accuracy))

test\_accuracy = evaluate(config, config.X\_test, config.y\_test)

**print**("Test Accuracy = {:.3f}".format(test\_accuracy))

saver.save(sess, './' + config.name)

**print**("Model saved")

**return** test\_accuracy

In [ ]:

training\_configurations = [

TrainConfiguration(name = "Final architecture", model = convnet, validation\_size = 0.05, epochs = 100),

]

In [ ]:

*### Train your model here.*

*### Feel free to use as many code cells as needed.*

accuracy = {}

**for** config **in** training\_configurations:

accuracy[config.name] = train\_and\_evaluate(config)

**print**(accuracy)

## Step 3: Test a Model on New Images

Take several pictures of traffic signs that you find on the web or around you (at least five), and run them through your classifier on your computer to produce example results. The classifier might not recognize some local signs but it could prove interesting nonetheless.

You may find signnames.csv useful as it contains mappings from the class id (integer) to the actual sign name.

### Implementation

Use the code cell (or multiple code cells, if necessary) to implement the first step of your project. Once you have completed your implementation and are satisfied with the results, be sure to thoroughly answer the questions that follow.

In [ ]:

*### Load the images and plot them here.*

*### Feel free to use as many code cells as needed.*

**from** **os** **import** listdir

**from** **os.path** **import** isfile, join

**import** **tensorflow** **as** **tf**

**from** **scipy** **import** misc

photos\_dir = 'signs'

onlyfiles = [join(photos\_dir, f) **for** f **in** listdir(photos\_dir) **if** isfile(join(photos\_dir, f))]

raw\_images = []

**for** filename **in** onlyfiles:

image = misc.imread(filename)

resized\_image = misc.imresize(image, (32, 32))

raw\_images.append(resized\_image)

images = np.asarray(raw\_images)

show\_images(images)

In [ ]:

*### Run the predictions here.*

*### Feel free to use as many code cells as needed.*

config = training\_configurations[0]

predict = tf.nn.softmax(config.predictions)

preprocessed\_images = np.asarray([preprocess\_image(image) **for** image **in** images])

**with** tf.Session() **as** sess:

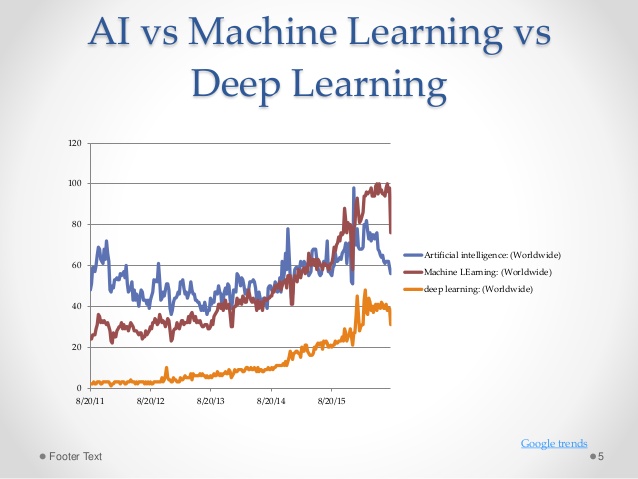
sess.run(tf.global\_variables\_initializer())

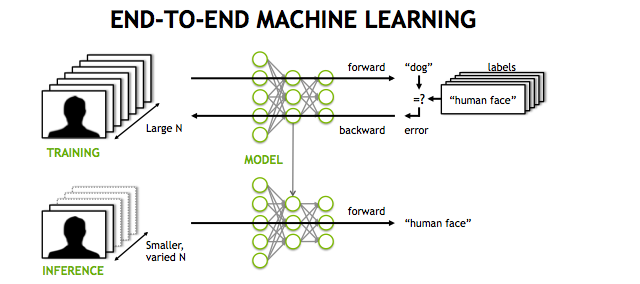
predictions = sess.run(tf.nn.top\_k(predict, k = 5), feed\_dict={config.x: preprocessed\_images, keep\_prob: 1.0})

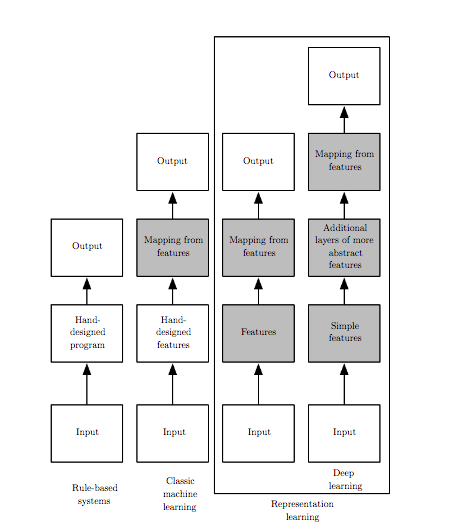
**print**(predictions)

https://github.com/mmehas/neurohacker/blob/master/neurohacker.ipynb

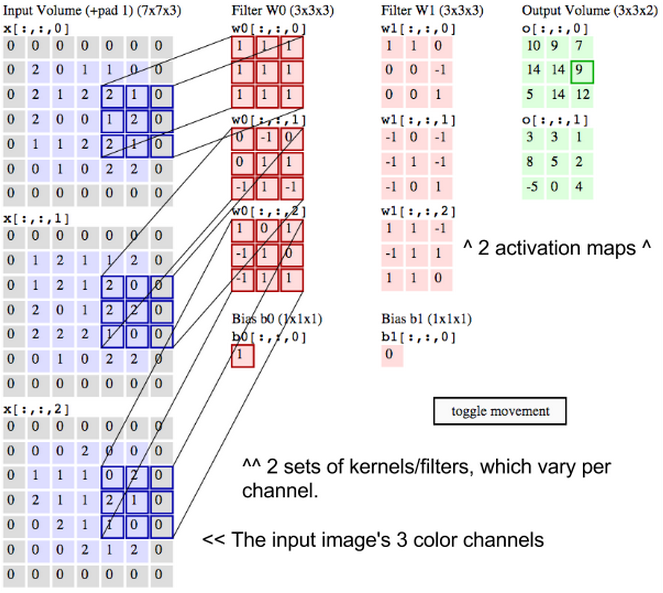
Для обработки и классификации изображений посредством нейронных сетей мы предлагаем использовать свободную библиотеку от Google – TensorFlow. При этом упор будем делать на использование DeepLearning (Глубинное обучение).

Для Аргументации:  






В сети можно найти немало примеров создания подобных нейронных сетей для распознавания лиц, цифр, дорожных знаков. В целом, идея будет той же:  
 Создание 5-6 конвуляционных слоев, каждый из которых будет заостряться на все более крупных элементах(точка->линия->совокупность линий, похожая на перелом). Учитывая одноцветность рентгеновских снимков, предлагается сразу перевести изображения в черно-белую гамму, что так же приведет и к уменьшению требуемых аппаратных ресурсов для обучения.



Между конвуляционными слоями необходимо добавлять пулинговые слои, чтобы выделять некоторые особоважные признаки той или иной части изображения

**Трансформация стратификации современного российского общества в процессе развития демократических и рыночных реформ.**

**Введение:**

(http://studme.org/1600030225401/sotsiologiya/stratifikatsiya\_sovremennogo\_rossiyskogo\_obschestva)

Как говорил Питирим Сорокин в своей статье «Социальная и культурная мобильность» - любая организованная социальная группа всегда социально стратифицирована. Не существовало и не существует ни одной постоянной социальной группы, которая была бы “плоской” и в которой все ее члены были бы равными.( http://socioline.ru/pages/pitirim-sorokin-sotsialnaya-i-kulturnaya-mobilnost)

Как мы знаем, за последние годы изменения в социальной структуре нашей страны достаточно велики и нельзя их не заметить. Времена перестройки полностью изменили виды и размеры старт общества России. К тому же, нельзя с уверенностью сказать, что страты сформированы окончательно и неизменно. Экономическая ситуация имеет прямое отражение на формировании структуры общества. Таким образом, в процессе развития демократических и рыночных реформ социальная стратификация российского общества претерпела значительную трансформацию.

Экономический потенциал разных социальных групп в СССР измерялся мерой их участия во владении, распределении и использовании общественного богатства. По этому критерию выделялись такие группы, как бюрократия, распределявшая дефицитные социальные блага; руководители производств, распоряжавшиеся финансами и продукцией предприятий и обычно причастные к теневой экономике; работники материально-технического снабжения, оптовой и розничной торговли, сферы обслуживания и проч. Однако люди, в той или иной мере причастные к распределительно-обменным процессам, составляли сравнительно небольшую долю населения. Массовые слои общества подобных прав не имели, и их экономическая стратификация определялась уровнем заработков и семейных доходов, зависевших от множества факторов, начиная с характера и содержания труда, сфер и отраслей его приложения, ведомственной принадлежности предприятий и кончая численностью и составом семей. Взаимодействие экономических, социальных, региональных, демографических и иных факторов создавало довольно пеструю картину экономической стратификации населения.

В настоящее время экономический потенциал общественных групп включает три компонента: владение капиталом, производящим доход; причастность к процессам распределения, перемещения и обмена общественного продукта; уровень личных доходов и потребления. Особая роль принадлежит первому компоненту. Активно формируются разнообразные формы негосударственной собственности (индивидуальная, групповая, кооперативная, акционерная, корпоративная и т.д.), возникают разные типы капитала (финансовый, торговый, промышленный). В социальном плане более или менее отчетливо выделились собственники частного капитала. Среди них есть и очень крупные, и средние, и мелкие, относящиеся, соответственно, к разным слоям.(здесь можно нарисовать табличку или пару графиков)

**Изменения:**

1. Во-первых, коренным образом изменился сам характер стратификационной системы. Если в советском обществе преобладали черты этакратической системы, построенной на властных иерархиях и формальных рангах, то в современном российском обществе формулирование стратификационной системы происходит на экономической основе, когда главными критериями становятся уровень доходов, владение собственностью и возможность осуществлять самостоятельную хозяйственную деятельность.
2. Во-вторых, сложился довольно многочисленный предпринимательский слой, высшие представители которого не только составляют существенную часть хозяйственно-экономической элиты, но и в ряде случаев входят в политическую элиту страны. Можно по-разному оценивать сущность, состав и структуру этого слоя, но нельзя не видеть, что переход к рыночной экономике породил качественно новые статусные группы, обладающие экономической свободой и претендующие на самые высокие места в системе общественной иерархии.
3. В-третьих, в ходе реформ появились новые престижные виды деятельности, что заметно изменило социально-профессиональную стратификационную систему. Так, резко возрос престиж предпринимательской, коммерческой, финансово-банковской, управленческой, юридической и некоторых других видов деятельности (реклама, маркетинг, операции с недвижимостью и т.д.).
4. В-четвертых, наметилось полярное расслоение общества, что находит выражение в растущей дифференциации доходов населения. Так, если незадолго до распада советского государства децильный коэффициент (соотношение средних доходов 10% наименее обеспеченных и 10% наиболее обеспеченных слоев населения) равнялся пяти, то в 1997 г. он повысился до двенадцати, а в настоящее время — до двадцати пяти.
5. В-пятых, несмотря на существенную социальную полярность общества, начинает формироваться средний класс, ядро которого образуют высокопродуктивные, инициативные и предприимчивые социальные категории (предприниматели, менеджеры, бизнесмены, фермеры, представители научно-технической интеллигенции, высококвалифицированные рабочие и др.). Средний класс определяет стабильность социальной системы и одновременно обеспечивает ее динамичное развитие. Он заинтересован в осуществлении экономических реформ и выступает субъектом технологической модернизации и политической демократизации общества.

**Вопросы экономики:**

Как говорил Макс Вебер в своей статье «Основные понятия стратификации»( http://socioline.ru/pages/maks-veber-osnovnye-ponyatiya-stratifikatsii) - фактором, создающим “класс”, вне всяких сомнений выступает экономический интерес. Рассмотрим изменения в стратификации современного российского общества, обусловленные экономическими факторами.

Большинство россиян ожидало от перестройки пусть не мгновенного, но достаточно быстрого роста благосостояния и улучшения условий жизни. В действительности, однако, рыночные реформы привели к резкому ухудшению их материального положения. В настоящее время реальные доходы россиян не превышают 40-50% от уровня середины 80-х годов. Причем если и происходит некоторое повышение их среднего уровня, то только за счет наиболее обеспеченной части населения. У основной же его массы доходы снижаются. Не случайно доля россиян, считавших свое материальное положение плохим, повысилась с 42% в 1993 году до 49% в 1995 году, причем доля оценивавших его как "очень плохое" выросла с 8 до 12%. Изменилась и структура денежных доходов населения. В конце 1980-х годов они на 72% формировались за счет заработной платы, на 14% за счет социальных трансфертов и еще на 14% за счет доходов от собственности и предпринимательской деятельности. В 1995 году соотношение названных источников доходов составило 40:16:44. В условиях, когда участие в предпринимательстве принимают примерно 10% населения, это само по себе говорит о поляризации бедности и богатства. (тоже пара графиков).

По данным специальных исследований, в конце 1980-х годов децильный коэффициент дифференциации доходов не превышал 4,5 раза, в 1993 году он составил 7,8, а осенью 1995 года - уже 10,5 раза. Сходную картину рисует и государственная статистика: 1991 год - 5,5 раза, 1993 год - 11 и 1995 год - 13 раз. В 1994 году 10% наиболее обеспеченных граждан располагали 40% совокупных доходов населения, в то время как на долю абсолютного большинства оставалось 60%. Осенью 1995 года разница между уровнями доходов, получаемых 10% наиболее и наименее обеспеченных россиян, составляла почти 25 раз. Воспитанные в эгалитарных ценностях россияне воспринимают поляризацию бедности и богатства как рост социальной несправедливости.

**Варианты стратификации:**

В стратификационной структуре современного российского общества Т. И. Заславская выделила четыре слоя: верхний, средний, базовый и нижний.

1. Верхний слой (6% занятого населения) образуют элитные и субэлитные группы, занимающие важные позиции в системе государственного управления, в экономических и силовых структурах. Это политические лидеры, верхушка государственного аппарата, значительная часть генералитета, руководители промышленных корпораций и банков, преуспевающие предприниматели и бизнесмены, видные деятели науки и культуры. Верхний слой почти на 90% представлен мужчинами молодого и среднего возраста. Это самый образованный слой: две трети его представителей имеют высшее образование. Уровень доходов этого слоя в 10 раз превышает доходы нижнего слоя и в 6—7 раз — доходы базового слоя. Таким образом, верхний слой обладает самым мощным экономическим и интеллектуальным потенциалом и имеет возможность оказывать прямое влияние на процессы реформ.
2. Средний слой (18% занятого населения) состоит из мелких и средних предпринимателей, полупредпринимателей, менеджеров средних и небольших предприятий, представителей среднего звена государственного аппарата, администраторов непроизводственной сферы, старших офицеров, лиц интеллектуальных профессий, фермеров, наиболее квалифицированных рабочих и служащих. Почти 60% из них заняты в негосударственном секторе. Большую часть и здесь составляют мужчины, преимущественно среднего возраста. Уровень образования представителей этого слоя значительно выше, чем в среднем но стране, однако несколько ниже но сравнению с верхним слоем. По уровню доходов средний слой существенно уступает верхнему слою и, соответственно, заметно хуже его социальное самочувствие. Несмотря на то что большинство представителей среднего слоя не обладают ни достаточным капиталом, ни отвечающим в полной мере современным требованиям уровнем профессионализма, ни высоким социальным престижем, социологи рассматривают этот слой российского общества в качестве зародыша среднего класса в его западном понимании.
3. Базовый слой (66% занятого населения) включает лиц, занятых преимущественно в государственном секторе экономики. К нему относятся рабочие индустриального типа, значительная часть интеллигенции (специалисты), полуинтеллигенция (помощники специалистов), служащие из технического персонала, основная масса военнослужащих, работники массовых профессий торговли и сервиса, а также большая часть крестьянства. Около 60% этого слоя составляют женщины, в основном среднего и старшего возраста. Только 25% его представителей имеют высшее образование. Уровень жизни этого слоя, и прежде невысокий, в последние годы постоянно снижается: 44% его представителей живут за чертой бедности. Хотя потребности, интересы и ценностные ориентации групп, составляющих базовый слой, весьма различны, их модель поведения в переходный период достаточно сходна: это приспособление к изменяющимся условиям с целью выжить и по возможности сохранить достигнутый статус.
4. Нижний слой (10% занятого населения) обладает наименьшим профессионально-квалификационным и трудовым потенциалом. К нему относятся работники, занятые простейшими видами труда, не требующими профессиональных знаний (уборщики, лифтеры, вахтеры, курьеры, подсобные рабочие, такелажники и т.д.). Из них более 40% заняты в индустриальных отраслях и 25% — в сфере торговли, обслуживания. Две трети этого слоя составляют женщины, а доля пожилых людей в три раза выше средней по стране. Для этих социальных категорий характерен чрезвычайно низкий уровень жизни: 2/3 живут за чертой бедности, из них четверть — за гранью нищеты. Большинство представителей этого слоя выступают против реформ, а 1/3 считает, что стране нужна диктатура.

Наряду с этими основными слоями Т. И. Заславская отмечает также наличие "социального дна", которое образуют алкоголики, бомжи, бродяги, криминальные элементы и т.д. Однако эмпирически идентифицировать эти группы не удалось, что связано с их десоциализацией, изолированностью от общества, включенностью в различные криминальные и полукриминальные структуры.

Несколько иную модель стратификационной системы современного российского общества предлагает известный социолог М. Н. Римашевская, которая выделяет следующие социально-классовые группы:

1. "общероссийские элитные группы", обладающие крупной собственностью и средствами властного влияния на федеральном уровне;
2. "региональные и корпоративные элиты", обладающие значительной собственностью и влиянием на уровне регионов и секторов экономики;
3. "верхний средний класс", имеющий собственность и доходы, обеспечивающие западные стандарты повеления и притязания на повышение социального статуса;
4. "динамичный средний класс", проявляющий социальную активность и имеющий доходы, обеспечивающие среднероссийские и более высокие стандарты потребления;
5. "аутсайдеры", характеризующиеся низкой социальной активностью, невысоким уровнем доходов и ориентацией на легальные способы их получения;
6. "маргиналы", отличающиеся низкой степенью социальной адаптации, незначительными доходами и неустойчивостью социально-экономического положения;
7. "криминальные элементы", проявляющие высокую социальную активность, но противоречащую моральным и правовым нормам общества.

Приведенные концепции социальной стратификации современного российского общества не исчерпывают многообразия точек зрения по данной проблематике. Интересные результаты получены в ходе исследований, проведенных по вопросам социальной дифференциации в разных регионах страны. Дискуссия продолжается и по проблемам формирования среднего класса в России. И это вполне понятно, ибо стратификационный профиль нашего общества подвижен, меняется в зависимости от множества факторов — подъема или спада производства, структурной перестройки экономики, технологического обновления, появления новых престижных профессий и т.д. Потребность в социологическом объяснении этих бурно развивающихся процессов будет и в дальнейшем стимулировать изучение различных аспектов стратификации российского общества.

**Лабораторная работа\_1. «Графы».**

Граф – совокупность точек плоскости, называемых ***вершинами***, соединенных линиями (отрезками), называемыми ***ребрами*** или дугами.

|  |  |  |
| --- | --- | --- |
| Неориентированный граф |  | из каждой вершины можно пройти в каждую и обратно. |
| Ориентированный граф (орграф) |  | ребра являются направленными, т. е. существует только одно доступное направление между двумя связными вершинами |
|  |  |  |
| Смешанный граф |  | Наличие как ориентированных, так и неориентированных ребер. |
|  |  |  |
| Взвешенные графы | | |
| Взвешенный неориентированный граф |  | ***Вес ребра графа.***  Графы, ребрам которых поставлено в соответствие конкретное значение называются *взвешенными* графами.  Значение – ***вес ребра графа***. |
| Взвешенный ориентированный граф |  |
| Вершинно-взвешенный граф |  |  |
| Реберно-взвешенный граф |  |  |
|  |  |  |
| Смешанный граф с наличием петель |  | ***Петля.***  У ребра оба конца совпадают, т. е. ребро выходит из вершины и входит в нее, то такое ребро называется ***петлей***. |
|  |  |  |

Обозначения, используемые в теории графов:

* G=(V, E), здесь G – граф, V – его вершины, а E – ребра;
* |V| – порядок (число вершин) – обозначим n;
* |E| – размер графа (число рёбер) – обозначим m.

Вершины u и v , связанные между собой ребром (u, v) называют ***смежными***.

***Путь*** или маршрут, от вершины u к вершине v, определяется как последовательность смежных вершин, начинающаяся в u и заканчивающаяся в v.

***Длина пути*** – количество ребер через которые проходит путь.

Если все ребра графа, составляющие путь различны, то такой путь называется ***простым***.

***Для ориентированных*** и ***неориентированных*** графах - понятие ***степень вершины***.

Пример 1.

|  |  |  |  |
| --- | --- | --- | --- |
|  | ***Порядок графа***  |V|=5  ***Размер графа***  |E|=10  ***степень вершины:*** а=4 |  | |V|=4,  |E|=5;  ***степень вершины:***  ***v2 =2; v4 =3*** |

Способы представления графа в памяти компьютера:

1. **Матрица смежности**
2. **Матрица инцидентности**
3. **Список смежности**
4. **Список ребер**
5. **Матрица смежности** *-* квадратная булевская матрица, отражающая смежность вершин. Размер матрицы n×n, где n – число вершин.

А[I,j]= 1, если вершины с номерами i и j смежные.

А[I,j]= 0, если вершины с номерами i и j не смежные.

|  |  |  |
| --- | --- | --- |
| *Граф* | *Матрица смежности* | *Описание* |
| не ориентированный | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 1 | 2 | 3 | 4 | | 1 | 0 | 1 | 0 | 1 | | 2 | 0 | 0 | 1 | 1 | | 3 | 0 | 1 | 0 | 0 | | 4 | 1 | 0 | 1 | 0 | | *MSG-матрица смежности графа* int [,]*MSG* = new int [4,4]  { {0, 1, 0, 1}, {0, 0, 1, 1}, {0, 1, 0, 0}, {1, 0, 1, 0}, }; |
| Ориентированный | |  |  |  |  |  | | --- | --- | --- | --- | --- | | вер  ши  на | A | *B* | *C* | *D* | | *A* | 1 | 1 | 1 | 0 | | *B* | 0 | 0 | 0 | 1 | | *C* | 0 | 0 | 0 | 0 | | *D* | 1 | 0 | 0 | 0 |     Ребра (дуги) |  |

1. **Матрица инцидентности (матрица инциденций) -** каждая из вершин u и v инцидентна ребру g, если g соединяет u и v. Размер матрицы инцидентности – **n×m**, здесь **n** – число вершин графа, **m**– число ребер.

|  |  |  |
| --- | --- | --- |
| *Граф* | *Матрица инцидентности* | *Описание* |
| не ориентированный | Элемент матрицы=1 , если вершина инцидентна ребру  Элемент матрицы=0 , если вершина не инцидентна ребру   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | a | b | c | d | e | | 1 | 1 | 1 | 0 | 0 | 0 | | 2 | 1 | 0 | 1 | 1 | 0 | | 3 | 0 | 0 | 0 | 1 | 1 | | 4 | 0 | 1 | 1 | 0 | 1 | | Аналогично матрице смежности |
| Ориентированный | Элемент матрицы=-1 , если вершина инцидентна ребру ,если *вершина* является его *началом*  Элемент матрицы=0 , если вершина не инцидентна ребру  Элемент матрицы = 1 , если вершина инцидентна ребру ,если *вершина* является его *концом*   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | № вер  шины | Ребра (дуги) | | | | | |  | a | b | c | d | e | | 1 | -1 | 1 | 0 | 0 | 0 | | 2 | 1 | 0 | -1 | -1 | 0 | | 3 | 0 | 0 | 0 | 1 | 1 | | 4 | 0 | -1 | 1 | 0 | -1 | |  |

1. **Список смежности**

|  |  |  |
| --- | --- | --- |
| *Граф* | *Список смежности* |  |
| Не ориентированный граф | **beg[v]** - указатель на начало списка, содержащего вершины, смежные с вершиной **v**. |  |
| Ориентированный граф  http://xn--h1aaalccfe5c.xn--p1ai/2/6_clip_image002_0000.gif | http://xn--h1aaalccfe5c.xn--p1ai/2/6_clip_image003_0001.gif |  |
| Смешанный граф  6 вершин (**|V|**) ,  5 ребер (**|E|**). | |  |  | | --- | --- | | Вершина *выхода* | Вершины *входа* | | 1 | 5 | | 2 | 6 | | 3 | 2, 5 | | 4 | 5 | | 5 | 1, 4 | | 6 | 2 | | **Vmax** - максимально возможное число вершин  **Emax** - максимально возможное число ребер  **int t[Emax]**   – хранит вершины, в которые *входят* ребра;  **int next\_el [Emax]**   – содержит указатели на элементы массива **t**;  **int head [Vmax]** – содержит указатели на такие вершины записанные в массив **t,** с которых начинается процесс перечисления всех вершин смежных одной i-ой вершине.  **const int Vmax=100, Emax=Vmax\*2;** |

1. **Список ребер (список ребер графа) -** в каждой строке записаны *две смежные вершины* и *вес*, соединяющего эти вершины ребра.

Количество строк в списке ребер(высота списка) равно:

**Кол-во ориентированных ребер + (Кол-во не ориентированных ребер)\*2**

|  |  |  |
| --- | --- | --- |
| *Граф* | *Список ребер* |  |
| 5 вершин, 4 ребра; 3 направленных ребра и 1 ненаправленное.  Высота списка ребер: 3+1\*2=5. | |  |  |  | | --- | --- | --- | | Номер вершины *выхода* | Номер вершины *входа* | Вес ребер | | Смежные вершины | |  | | 1 | 3 | 13 | | 1 | 4 | 14 | | 2 | 1 | 12 | | 3 | 1 | 13 | | 5 | 1 | 15 | |  |

**Задания для выполнения в аудитории.**

**Задание №1. 2 балла.**

**Составьте в виде таблицы, опишите на языке программирования С# с обеспечением вывода на экран матриц смежности и инцидентности для следующих графов:**

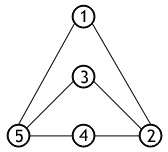
|  |  |
| --- | --- |
| **Вариант 1** | **Вариант 2** |
|  |  |
|  |  |

**Задание №2. 3 балла.**

**Реализуйте обход графа в глубину.**

Данный метод предполагает продвижение вглубь до тех пор, пока это возможно. Невозможность дальнейшего продвижения, означает, что следующим шагом будет переход на последний, имеющий несколько вариантов движения (один из которых исследован полностью), ранее посещенный узел (вершина). Отсутствие последнего свидетельствует об одной из двух возможных ситуаций: либо все вершины графа уже просмотрены, либо просмотрены все те, что доступны из вершины, взятой в качестве начальной, но не все (несвязные и ориентированные графы допускают последний вариант).

Приведенный неориентированный связный граф имеет в сумме 5 вершин.



Сначала необходимо выбрать начальную вершину. Какая бы вершина в качестве таковой не была выбрана, граф в любом случае исследуется полностью, поскольку, это связный граф без единого направленного ребра.

Пусть начало обхода:

- узел 1, тогда порядок последовательности просмотренных узлов следующий: 1 2 3 5 4.

- узел 3, то порядок обхода: 3 2 1 5 4.

Алгоритм поиска в глубину базируется на рекурсии, т. е. функция обхода, по мере выполнения, вызывает сама себя, что делает код компактным.

static void PGL(int st, bool[] visit, int[,] graph)

{

int i;

Console.Write(" "+(st+1));

visit[st] = true;

for (i = 0; i <= n-1; i++)

if ((graph[st,i] != 0) && (!visit[i]))

**PGL(i, visit, graph);**

}

// ==== метод Main =============

int start=2; //номер вершины, с которой нач.обход

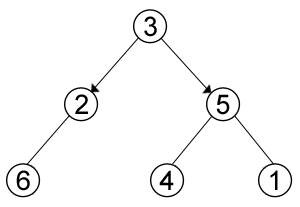
bool[] visit = new bool[n];//массив для хранения посещенных вершин

// заполнение массива visit

// «вывод графа» в виде матрицы смежности

// вызов метода PGL

**ЗАДАНИЯ ДЛЯ СРС**

1. **Задание №1 . 7 баллов**
   1. **Разработайте программу для получения списка смежности графа.** 

*Используйте следующие обозначения:*

**Vmax** - максимально возможное число вершин

**Emax** - максимально возможное число ребер

**int t[Emax]**

 – хранит вершины, в которые *входят* ребра;

**int next\_el [Emax]**

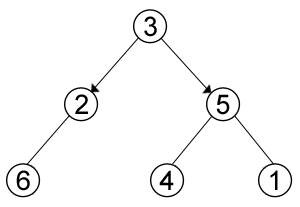
 – содержит указатели на элементы массива **t**;

**int head [Vmax]** – содержит указатели на такие вершины записанные в массив **t,** с которых начинается процесс перечисления всех вершин смежных одной i-ой вершине.

**const int Vmax=100, Emax=Vmax\*2;**

*Работа программы:*

1. Ввод с клавиатуры количество вершин и ребер.
2. В цикле ввод ребер (смежных вершин) **v и u**, например для графа



Вводим ребро от вершины 5 до вершины 4:

**5**

**4**

от вершины 5 до вершины 1:

**5**

**1**

И т.д.

После ввода каждой пары смежных вершин – обращение к процедуре добавления ребер ADD(v,u). Если ребро ориентированное, то ADD(v,u)если не ориентированное, то процедура добавляет два ребра: ADD(v,u) и ADD(u,v).

static void ADD(int v, int u)

{

k=k+1;

t[k]=u;

next\_el[k]=head[v];

head[v]=k;

}

Результат работы программы – список смежности графа. Для графа из данного задания результат следующий:

1=>5

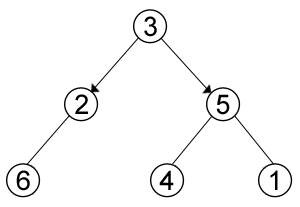
2=>6

3=>2,5

4=>5

5=>1,4

**1.2. Сделайте копию экрана с результатами работы программы для следующих графов:**



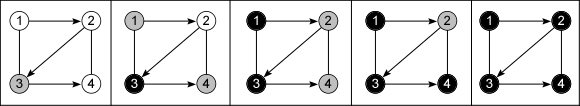
* 1. **= 5 баллов**
  2. **= 2 балла**

**Задание №2. 10 баллов**

**Реализуйте обход графа – поиск в ширину.**

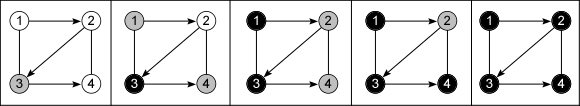
***Пример 1.***

Дано: смешанный граф с |V| = 4 и |E| = 5.



Предположим, что в процессе работы алгоритма каждая из вершин графа может быть окрашена в один из трех цветов: черный, белый или серый. Изначально все вершины белые. По мере обнаружения, окрашивается сначала в серый, а затем в черный цвет. Определенный момент обхода описывает следующие условие: если вершина черная, то все ее потомки окрашены в серый или черный цвет.

Стартовая вершина – узел 3.



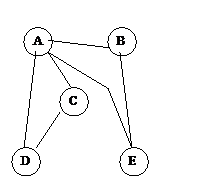
Порядок обхода: 3 1 4 2

Стартовая вершина – узел 4. Порядок обхода: 4 2 3 1

Стартовая вершина – узел 2 Порядок обхода: 2 3 4 1.

Стартовая вершина – узел 1 Порядок обхода: 1 2 3 4

***Пример 2.*** *Для представленного графа «вывод графа» при обходе «в ширину» следующий:*

 B, A, E, С, D

***Реализация алгоритма.***

Последовательность обработки:

1. обрабатываются все вершины, **смежные** с текущей,
2. обрабатываются все **потомки этих вершин.**

**Алгоритм:**

**procedure** ОБХОД В ШИРИНУ(**P**: вершина);   
**begin**   
    Поместить вершину **P** в пустую очередь *OCH*;   
**while** очередь *OCH* не пуста **do**

**begin**  
            Взять первую вершину P из очереди *OCH*;   
**if P** еще не посещалась  **then**

**begin**  
               Посетить вершину **P**

Поместить в очередь *OCH* все вершины, смежные с **P**   
**end**   
**end**   
**end;**

***Необходимые структуры:***

* матрица смежности графа, например GM (целочисленный тип данных);
* очередь, например, queue;
* массив посещенных вершин, например, visited (логический).

*Этапы процесса обхода графа:*

1. массив visited обнуляется, т. е. ни одна вершина графа еще не была посещена;
2. в качестве стартовой, выбирается некоторая вершина s и помещается в очередь (в массив queue);
3. вершина s исследуется (помечается как посещенная), и все смежные с ней вершины помещаются в конец очереди, а сама она удаляется;
4. если на данном этапе очередь оказывается пустой, то алгоритм прекращает свою работу; иначе посещается вершина, стоящая в начале очереди, помечается как посещенная, и все ее потомки заносятся в конец очереди;
5. пункт 4 выполняется пока это возможно.

**ПРИЛОЖЕНИЕ**

### *Изображение графов на плоскости*

При изображении графов чаще всего используется следующая система обозначений: каждой вершине сопоставляется точка на плоскости, и если между вершинами существует ребро, то соответствующие точки соединяются отрезком. В случае ориентированного графа отрезки заменяют стрелками.

Не следует путать изображение графа с собственно графом (абстрактной структурой), поскольку одному графу можно сопоставить не одно графическое представление. Изображение призвано лишь показать, какие пары вершин соединены рёбрами, а какие — нет. Часто на практике бывает трудно ответить на вопрос, являются ли два изображения моделями одного и того же графа или нет. В зависимости от задачи, одни изображения могут давать более наглядную картину, чем другие.

### *Языки описания и программы построения графов*

Для описания графов в целях, пригодных для машинной обработки и одновременно удобном для человеческого восприятия используется несколько стандартизированных языков, среди которых:

* DOT (язык графов)
* GraphML
* Trivial Graph Format
* GML
* GXL
* XGMML
* DGML

Отметим специализированные программы для построения графов. К наиболее удачным относятся коммерческие:

* ILOG
* GoView
* Lassalle AddFlow
* LEDA (есть бесплатная редакция).

Из бесплатных можно отметить Boost Graph Library - Библиотека для работы с графами на языке C++

Для визуализации графов можно использовать:

* Graphviz (по мнению экспертов, она хорошо работает для орграфов)
* LION Graph Visualizer.
* Графоанализатор – это русскоязычная программа, с весьма простым пользовательским интерфейсом.

***Методы теории графов***

Методы теории графов – одно из наиболее активно развивающихся направлений в сегментации изображений.

Александр Вежневец, Ольга Баринова. **Методы сегментации изображений: автоматическая сегментация.***Компьютерная графика и мультимедиа*. Выпуск №4(4)/2006. http://cgm.computergraphics.ru/content/view/147**Выпуск:** [Выпуск №4(4)/2006](http://cgm.computergraphics.ru/issues/issue14)

ДЛЯ ВУЗОВ Т . А . Павловска я Программирование на языке высокого уровня ПИТЕ Р Москв а • Санкт-Петербург • Нижни й Новгород • Вороне ж Ростов-на-Дон у • Екатеринбур г • Самар а • Новосибирс к Кие в • Харько в • Минс к 200 9 Павловская Татьяна Александровна С#. Программировани е н а язык е высоког о уровн я Учебник для вузов Рецензенты: Смирнова Н. Н. — заведующая кафедрой вычислительной техники Балтийского государственного технического университета «Военмех» им. Д. Ф. Устинова, кандидат технических наук, профессор; Трофимов В. В. —заведующий кафедрой информатики Санкт-Петербургского государственного университета экономики и финансов, доктор технических наук, профессор ББК 32.973-018.1 УДК 004.43 Павловская Т. А. П12 С#. Программирование на языке высокого уровня. Учебник для вузов. — СПб.: Питер, 2009. — 432 с: ил. ISBN 978-5-91180-174-8 Задача этой книги — кратко, доступно и строго изложить основы С#, одного из самых перс­ пективных современных языков программирования. Книга содержит описание версии С# 2.0 (2005) и предназначена для студентов, изучающих язык «с нуля», но будет полезна и опытным програм­ мистам, желающим освоить новый язык, не тратя времени на увесистые переводные фолианты. Кроме конструкций языка в книге рассматриваются основные структуры данных, исполь­ зуемые при написании программ, классы библиотеки, а также рекомендации по стилю и техно­ логии программирования. По ключевым темам приводятся задания для выполнения лабораторных работ, каждая из которых содержит по двадцать однотипных вариантов в расчете на учебную группу студентов. Язык С# как средство обучения программированию обладает рядом несомненных достоинств. Он хорошо организован, строг, большинство его конструкций логичны и удобны, а развитые средства диагностики и редактирования кода делают процесс программирования приятным и эф­ фективным. Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки дипломи­ рованных специалистов «Информатика и вычислительная техника». © ОО О «Питер Пресс», 2009 В с е прав а защищены . Никака я част ь данно й книг и н е може т быт ь воспроизведен а в како й б ы т о н и был о форм е бе з письменног о разрешени я владельце в авторски х прав . изведен а како й о ы т о н и был о ISBN 978-5-91180-174-8 О О О «Пите р Пресс» , 198206 , Санкт-Петербург , Петергофско е шоссе , д . 73 , ли т А2 9 Налогова я льгот а - общероссийски й классификато р продукци и О К 005-93 , то м 2 ; 9 5 300 5 - литератур а учебна я Подписан о в печат ь 04.09.08 . Форма т 70x100/16 . Усл . п . л . 34,83 . Тира ж 2700 . Зака з 1102 0 Отпечатан о п о технологи и Q P в ОА О «Печатны й двор » им . А . М . Горького . 197110 , Санкт-Петербург , Чкаловски й пр. , д . 15 . Содержание Предисловие 6 От издательства 7 Глава 1. Первый взгляд на платформу .NET 8 Объектно-ориентированное программировани е 11 Классы ^ 13 Среда Visual Studio.NET ' 14 Рекомендации по программировани ю 21 Глава 2. Основные понятия языка 22 Состав язык а 22 Типы данных 31 Рекомендации по программировани ю 36 Глава 3. Переменные, операции и выражения 38 Переменны е 38 Именованные константы 41 Операции и выражения 42 Линейны е программ ы 5 9 Рекомендации по программировани ю 67 Глава 4. Операторы 69 Выражения , блоки и пустые оператор ы 70 Оператор ы ветвления 70 Оператор ы цикла 75 Базовые конструкци и структурного программировани я 87 Обработк а исключительных ситуаций 89 Операторы checke d и unchecked 95 Рекомендации по программировани ю 95 Глава 5. Классы: основные понятия 100 Присваивание и сравнение объектов 103 Данные : поля и константы 104 Метод ы 106 Ключевое слово this 114 Конструктор ы 114 Свойства 120 Рекомендации по программировани ю 124 Глава 6. Массивы и строки 126 Массив ы 126 Оператор foreach 136 Массив ы объектов 138 Символ ы и строк и 139 Класс Random 148 Рекомендации по программировани ю 150 4 Содержание Глава 7. Классы: подробности 152 Перегрузк а методов 152 Рекурсивные методы 153 Метод ы с переменны м количеством аргументов 154 Мето д Main 156 Индексатор ы 157 Операции класса 161 Деструктор ы 169 Вложенные тип ы 169 Рекомендации по программировани ю 170 Глава 8. Иерархии классов 172 Наследование 172 Виртуальные методы 178 Абстрактные классы 181 Бесплодные классы 182 Класс object 183 Рекомендации по программировани ю 186 Глава 9. Интерфейсы и структурные типы 188 Синтаксис интерфейс а 188 Реализация интерфейс а 190 Работа с объектами через интерфейсы . Операции is и as 194 Интерфейс ы и наследование 195 Стандартные интерфейс ы .NET 198 Структуры 212 Перечисления 215 Рекомендации по программировани ю 219 Глава 10. Делегаты, события и потоки выполнения 220 Делегат ы 220 События 232 Многопоточны е приложения 237 Рекомендации по программировани ю 245 Глава 11. Работа с файлами 246 Потоки байтов 250 Асинхронны й ввод-вывод 253 Потоки символов 255 Двоичны е потоки 260 Консольный ввод-вывод 262 Работа с каталогами и файлами 263 Сохранение объектов (сериапизация ) '. . . 267 Рекомендации по программировани ю 270 Глава 12. Сборки, библиотеки, атрибуты, директивы 272 Сборк и 272 Создание библиотеки 275 Использование библиотеки 278 Рефлексия 279 Атрибуты 283 Пространств а име н 285 Директив ы препроцессор а 287 Рекомендации по программировани ю 290 Содержание 5 Глава 13. Структуры данных, коллекции и классы-прототипы 291 Абстрактные структуры данных 291 Пространство имен System.Collections 295 Классы-прототип ы 299 Частичные тип ы 308 Обнуляемые тип ы 309 Рекомендации по программировани ю 310 Глава 14. Введение в программирование под Windows 311 Событийно-управляемое программировани е 31 2 Шабло н Windows-приложения 314 Класс Control 32 3 Элементы управления 325 Предварительные замечания о формах 337 Класс Form 33 8 Диалоговы е окн а 339 Класс Application 342 Краткое введение в график у 34 4 Рекомендации по программировани ю 346 Глава 15. Дополнительные средства С# 347 Небезопасны й ко д 34 7 Регулярные выражения 355 Документировани е в формат е XML 365 Темы, не рассмотренны е в книг е 366 Заключение 36 9 Лабораторные работы 370 Лабораторна я работа 1. Линейны е программ ы 370 Лабораторна я работа 2 . Разветвляющиеся вычислительные процесс ы ........ . 371 Лабораторна я работа 3. Организация циклов 379 Лабораторна я работа 4. Простейши е классы 381 Лабораторна я работа 5. Одномерны е массив ы 38 5 Лабораторна я работа 6. Двумерны е массив ы 38 9 Лабораторна я работа 7. Строк и 393 Лабораторна я работа 8. Классы и операци и 39 5 Лабораторна я работа 9. Наследование 40 0 Лабораторна я работа 10. Структуры 40 5 Лабораторная работа 11 . Интерфейсы и параметризованны е коллекции 411 Лабораторна я работа 12. Создание Windows-приложений 41 2 Спецификаторы формата для строк С# 423 Список литературы 425 Алфавитный указатель 427 Предисловие Задача этой книги — кратко, доступно и строго изложить основы С#, одного из самых перспективных современных языков программирования. Книга содержит описание версии С# 2.0 (2005) и предназначена для студентов, изучающих язык «с нуля», но будет полезна и опытным программистам, желающим освоить но­ вый язык, не тратя времени на увесистые переводные фолианты. Помимо конструкций языка в книге рассматриваются основные структуры дан­ ных, используемые при написании программ, классы библиотеки, а также ре­ комендации по стилю и технологии программирования. По ключевым темам приводятся задания для лабораторных работ, каждая из которых содержит до двадцати однотипных вариантов в расчете на учебную группу студентов. Язык С# как средство обучения программированию обладает рядом несомнен­ ных достоинств. Он хорошо организован, строг, большинство его конструкций логичны и удобны. Развитые средства диагностики и редактирования кода дела­ ют процесс программирования приятным и эффективным. Мощная библиотека классов платформы .NE T берет на себя массу рутинных операций, что дает возможность решать более сложные задачи, используя готовые «строительные блоки» . Все это позволяет расценивать С# как перспективную замену языков Паскаль, BASIC и С+ + при обучении программированию. Немаловажно, что С# является не учебным, а профессиональным языком, пред­ назначенным для решения широкого спектра задач, и в первую очередь — в бы­ стро развивающейся области создания распределенных приложений. Поэтому базовый курс программирования, построенный на основе языка С#, позволит студентам быстрее стать востребованными специалистами-профессионалами. Мощь языка С# имеет и оборотную сторону: во-первых, он достаточно требовате­ лен к ресурсам компьютера 1 , во-вторых, для осмысленного написания простейшей программы, вычисляющей, «сколько будет дважды два», требуется изучить дос­ таточно много материала, но многочисленные достоинства языка и платформы .NET перевешивают все недостатки. Дополнительным достоинством является то, что компания Microsoft распро­ страняет версию С# Express 2005, по которой можно познакомиться с языком бесплатно (http://msdn.microsoft.com/vstuclio/express/visualCsharp/), что дает нам долгожданную возможность обойтись без пиратских копий программного обес­ печения и почувствовать себя законопослушными гражданами. 1 Например, минимальные требования для С# Express 2005 — Windows ХР/2000, процес­ сор Pentium 600 МГц и 128 Мбайт оперативной памяти. От издательства 7 ПРИМЕЧАНИ Е Название языка вообще-то следовало бы произносить как «Си-диез». Компания Microsoft неоднократно подчеркивала, что, несмотря на некоторое различие в на­ чертании и коде символов «решетки» и «диеза», в названии языка имеется в виду именно музыкальный диез и обыгрывается тот факт, что символом С в музыке обозначается нота «до». Таким образом, создатели языка вложили в его название смысл «на полтона выше С» . Однако в нашей стране укоренилась калька с англий­ ского — «Си-шарн». Поклонники этого произношения должны иметь в виду, что язык С++ они должны называть тогда исключительно «Си-плас-плас», потому что программист прежде всего обязан следовать логике! Все возможности языка С# описаны в соответствии с его спецификацией (http:// www.ecma-international.org/publications/standards/Ecma-334.htm) и сопровождают­ ся простыми примерами, к которым даны исчерпывающие пояснения. Большин­ ство примеров представляют собой консольные приложения, однако в главе 14 дается введение в программирование под Windows, а в главе 15 — введение в соз­ дание веб-форм и веб-служб. Возможности, появившиеся во второй версии языка, рассматриваются не в отдельной главе, а по логике изложения материала, при этом они всегда снабжены соответствующей ремаркой для тех, кто пользуется платформой .NET предыдущих версий. В книге приведено краткое введение в интересные и полезные темы, которые обычно не освещаются в базовом курсе С#, например, многопоточные приложе­ ния, асинхронные делегаты, асинхронный ввод-вывод и регулярные выражения. Определения синтаксических элементов языка выделены в тексте книги полу­ жирным шрифтом. Все ключевые слова, типы, классы и большинство методов, описанных в книге, можно найти по алфавитному указателю, что позволяет ис­ пользовать ее и в качестве справочника. Планируется интернет-поддержка кни­ ги на сайте http://ips.ifmo.ru в виде конспекта лекций, тестовых вопросов и про­ верки выполнения лабораторных работ. Традиционного раздела «Благодарности» в этом предисловии не будет: при напи­ сании этой книги мне никто не помогал, и вся ответственность за ошибки лежит исключительно на мне. У меня нет собаки или жены, чтобы поблагодарить их, как это принято у зарубежных авторов, а мой попугай постоянно мешал мне своими громкими неорганизованными выкриками. Ваши замечания, пожелания, дополнения, а также замеченные ошибки и опечат­ ки не ленитесь присылать по адресу mux@tp2055.spb.edu — и тогда благодаря вам кто-то сможет получить более правильный вариант этой книги. От издательства Ваши замечания, предложения и вопросы отправляйте также по адресу элек­ тронной почты comp@piter.com (издательство «Питер»). Мы будем рады узнать ваше мнение! Подробную информацию о наших книгах вы найдете на веб-сайте издательства www.piter.com. Глава 1 Первый взгляд на платформу .NET Программист пишет программу, компьютер ее выполняет. Программа создается на языке, понятном человеку, а компьютер умеет исполнять только программы, написанные на его языке — в машинных кодах. Совокупность средств, с помо­ щью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают, называют средой разработки, или оболочкой. Среда разработки обычно содержит: • текстовый редактор, предназначенный для ввода и корректировки текста программы; • компилятор, с помощью которого программа переводится с языка, на котором она написана, в машинные коды; • средства отладки и запуска программ; • общие библиотеки, содержащие многократно используемые элементы программ; • справочную систему и другие элементы. Под платформой понимается нечто большее, чем среда разработки для одного язы­ ка. Платформа .NET (произносится «дотнет») включает не только среду разработ­ ки для нескольких языков программирования, называемую Visual Studio.NET, но и множество других средств, например, механизмы поддержки баз данных, электронной почты и коммерции. В эпоху стремительного развития Интернета — глобальной информационной сети, объединяющей компьютеры разных архитектур, важнейшими задачами при соз­ дании програм м становятся : г I riepettocztjvtocTnb — возможност ь выполнени я н а различны х типа х компьютеров ; • безопасность — невозможност ь несанкционированных действий; • надежность — способность выполнять необходимые функции в предопреде­ ленных условиях; средний интервал между отказами; Глава 1 Первый взгляд на платформу .NET Программист пишет программу, компьютер ее выполняет. Программа создается на языке, понятном человеку, а компьютер умеет исполнять только программы, написанные на его языке — в машинных кодах. Совокупность средств, с помо­ щью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают, называют средой разработки, или оболочкой. Среда разработки обычно содержит: • текстовый редактор, предназначенный для ввода и корректировки текста программы; • компилятор, с помощью которого программа переводится с языка, на котором она написана, в машинные коды; • средства отладки и запуска программ; • общие библиотеки, содержащие многократно используемые элементы программ; • справочную систему и другие элементы. Под платформой понимается нечто большее, чем среда разработки для одного язы­ ка. Платформа .NET (произносится «дотнет») включает не только среду разработ­ ки для нескольких языков программирования, называемую Visual Studio.NET, но и множество других средств, например, механизмы поддержки баз данных, электронной почты и коммерции. В эпоху стремительного развития Интернета — глобальной информационной сети, объединяющей компьютеры разных архитектур, важнейшими задачами при соз­ дании программ становятся: • переносимость — возможность выполнения на различных типах компьютеров; • безопасность — невозможность несанкционированных действий; • надежность — способность выполнять необходимые функции в предопреде­ ленных условиях; средний интервал между отказами; Первый взгляд на платформу .NET 9 • использование готовых компонентов — для ускорения разработки; • межъязыковое взаимодействие — возможность применять одновременно не­ сколько языков программирования. Платформа .NET позволяет успешно решать все эти задачи. Для обеспечения пе­ реносимости компиляторы, входящие в состав платформы, переводят программу не в машинные коды, а в промежуточный язык (Microsoft Intermediate Language, MSIL, или просто IL) , который не содержит команд, зависящих от языка, опера­ ционной системы и типа компьютера. Программа на этом языке выполняется не самостоятельно, а под управлением системы, которая называется общеязыковой средой выполнения (Common Language Runtime, CLR) . Среда CL R может быть реализована для любой операционной системы. При выполнении программы CL R вызывает так называемый JIT-компилятор, пере­ водящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. JIT означает «just in time», что можно перевести как «вовремя», то есть компилируются только те части программы, которые требует­ ся выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в кэше 1 для дальнейшего использования. Схема выполнения программы при использовании платформы .NET приведена на рис. 1.1. Сборка (.ехе или .dll) Исходный текст программы Д Базовые классы среды (IL или метаданные) CLR Загрузчик классов t ЛТ-компилятор • Исполняемый код (машинные команды) Выполнение программы — 1 Вызовы нескомпилированных методов Рис. 1.1 . Схема выполнения программы в .NET Компилятор в качестве результата своего выполнения создает так называе­ мую сборку — файл с расширением ехе или dll, который содержит код на языке IL и метаданные. Метаданные представляют собой сведения об объектах, ис­ пользуемых в программе, а также сведения о самой сборке. Они позволяют организовать межъязыковое взаимодействие, обеспечивают безопасность и об­ легчают развертывание приложений, то есть установку программ на компьютеры пользователей. Кэш — область оперативной памяти, предназначенная для временного хранения ин­ формации. 10 Глава 1. Первый взгляд на платформу .NET ПРИМЕЧАНИ Е Сборка может состоять из нескольких модулей. В любом случае она представляет собой программу, готовую для установки и не требующую для этого ни дополни­ тельной информации, ни сложной последовательности действий. Каждая сборка имеет уникальное имя. Во время работы программы среда CL R следит за тем, чтобы выполнялись только разрешенные операции, осуществляет распределение и очистку памяти и обрабатывает возникающие ошибки. Это многократно повышает безопасность и надежность программ. Платформа .NET содержит огромную библиотеку классов\*, которые можно ис­ пользовать при программировании на любом языке .NET. Общая структура биб­ лиотеки приведена на рис. 1.2. Библиотека имеет несколько уровней. На самом нижнем находятся базовые классы среды, которые используются при создании любой программы: классы ввода-вывода, обработки строк, управления безопас­ ностью, графического интерфейса пользователя, хранения данных и пр. Платформа.NET Среда разработки приложений Visual Studio.NET Каркас .NET Framework Библиотека классов каркаса (FCL) Веб-сервисы Интерфейс пользователя Классы для работы с данными и XML Базовые классы среды Общеязыковая среда выполнения (CLR) .NET Enterprise Servers .NET Building Block Services Операционная система Аппаратные средства компьютера Рис. 1.2. Структура платформы .NET Над этим слоем находится набор классов, позволяющий работать с базами дан­ ных и XM L (с XM L вы познакомитесь в последней главе этой книги). Классы самого верхнего уровня поддерживают разработку распределенных приложений, а также веб- и Windows-приложений. Программа может использовать классы любого уровня. Подробное изучение библиотеки классов .NET — необходимая, но и наиболее трудоемкая задача программиста при освоении этой платформы. Библиотека 1 Понятие класса рассматривается в следующем разделе. Пока можно считать, что класс служит для реализации некоторых функций. Объектно-ориентированное программирование 11 классов вместе с CL R образуют каркас (framework), то есть основу платформы. Назначение остальных частей платформы мы рассмотрим по мере изучения материала. ПРИМЕЧАНИ Е Термин «приложение» можно для начала воспринимать как синоним слова «програм­ ма». Например, вместо фразы «программа, работающая под управлением Windows», говорят «Windows-приложение» или просто «приложение». Платформа .NET рассчитана на объектно-ориентированную технологию созда­ ния программ, поэтому прежде чем начинать изучение языка С#, необходимо познакомиться с основными понятиями объектно-ориентированного програм­ мирования (ООП) . Объектно-ориентированное программирование Принципы ОО П проще всего понять на примере программ моделирования. В ре­ альном мире каждый предмет или процесс обладает набором статических и ди­ намических характеристик, иными словами, свойствами и поведением. Поведе­ ние объекта зависит от его состояния и внешних воздействий. Например, объект «автомобиль» никуда не поедет, если в баке нет бензина, а если повернуть руль, изменится положение колес. Понятие объекта в программе совпадает с обыденным смыслом этого слова: объ­ ект представляется как совокупность данных, характеризующих его состояние, и функций их обработки, моделирующих его поведение. Вызов функции на выпол­ нение часто называют посылкой сообщения объекту1 . При создании объектно-ориентированной программы предметная область пред­ ставляется в виде совокупности объектов. Выполнение программы состоит в том, что объекты обмениваются сообщениями. Это позволяет использовать при про­ граммировании понятия, более адекватно отражающие предметную область. При представлении реального объекта с помощью программного необходимо выде­ лить в первом его существенные особенности. Их список зависит от цели модели­ рования. Например, объект «крыса» с точки зрения биолога, изучающего миграции, ветеринара или, скажем, повара будет иметь совершенно разные характеристики. Выделение существенных с той или иной точки зрения свойств называется абст­ рагированием. Таким образом, программный объект — это абстракция. Важным свойством объекта является его обособленность. Детали реализации объ­ екта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты Например, вызов функции «повернуть руль» интерпретируется как посылка сообщения «автомобиль, поверни руль!». 12 Глава 1. Первый взгляд на платформу .NET от пользователя объекта и недоступны для непреднамеренных изменений. Объект используется через его интерфейс — совокупность правил доступа. Скрытие деталей реализации называется инкапсуляцией (от слова «капсула»). Ничего сложного в этом понятии нет: ведь и в обычной жизни мы пользуемся объектами через их интерфейсы. Сколько информации пришлось бы держать в голове, если бы для просмотра новостей надо было знать устройство телевизора! Таким образом, объект является «черным ящиком», замкнутым по отношению к внешнему миру. Это позволяет представить программу в укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объ­ емом информации и успешно отлаживать сложные программы. Сказанное можно сформулировать более кратко и строго: объект — это инкапсу­ лированная абстракция с четко определенным интерфейсом. Инкапсуляция позволяет изменить реализацию объекта без модификации основ­ ной части программы, если его интерфейс остался прежним. Простота модифи­ кации является очень важным критерием качества программы, ведь любой про­ граммный продукт в течение своего жизненного цикла претерпевает множество изменений и дополнений. Кроме того, инкапсуляция позволяет использовать объект в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки объектов для применения во многих программах. Каждый год в мире пишется огромное количество новых программ, и важнейшее значение приобретает возможность многократного использования кода. Преиму­ щество объектно-ориентированного программирования состоит в том, что для объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости не только повторять исходный код родительского объекта, но даже иметь к нему доступ. Наследование является мощнейшим инструментом ОО П и применяется для сле­ дующих взаимосвязанных целей: • исключения из программы повторяющихся фрагментов кода; • упрощения модификации программы; • упрощения создания новых программ на основе существующих. Кроме того, только благодаря наследованию появляется возможность использо­ вать объекты, исходный код которых недоступен, но в которые требуется внести изменения. Наследование позволяет создавать иерархии объектов. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а бо­ лее специализированные — на ветвях и листьях. Наследование облегчает исполь­ зование библиотек объектов, поскольку программист может взять за основу объек­ ты, разработанные кем-то другим, и создать наследников с требуемыми свойствами. Объект, на основании которого строится новый объект, называется родительским объектом, объектом-предком, базовым классом, или суперклассом, а унаследо­ ванный от него объект — потомком, подклассом, или производным классом. Классы 13 О О П позволяет писать гибкие, расширяемые и читабельные программы. Во мно­ гом это обеспечивается благодаря полиморфизму, под которым понимается воз­ можность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа. Чаще все­ го понятие полиморфизма связывают с механизмом виртуальных методов, кото­ рый мы рассмотрим в главе 8. Подводя итог сказанному, сформулирую достоинства ООП. • использование при программировании понятий, близких к предметной области; • возможность успешно управлять большими объемами исходного кода благо­ даря инкапсуляции, то есть скрытию деталей реализации объектов и упроще­ нию структуры программы; • возможность многократного использования кода за счет наследования; • сравнительно простая возможность модификации программ; • возможность создания и использования библиотек объектов. Эти преимущества особенно явно проявляются при разработке программ большого объема и классов программ. Однако ничто не дается даром: создание объектно- ориентированной программы представляет собой весьма непростую задачу, по­ скольку требует разработки иерархии объектов, а плохо спроектированная иерар­ хия может свести к нулю все преимущества объектно-ориентированного подхода. Кроме того, идеи ОО П не просты для понимания и в особенности для практиче­ ского применения. Чтобы эффективно использовать готовые объекты из биб­ лиотек, необходимо освоить большой объем достаточно сложной информации. Неграмотное же применение ОО П способно привести к созданию излишне слож­ ных программ, которые невозможно отлаживать и усовершенствовать. Классы Д ля представления объектов в языках С#, Java, С++ , Delphi и т. п. используется понятие класс, аналогичное обыденному смыслу этого слова в контексте «класс членистоногих», «класс млекопитающих», «класс задач» и т. п. Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых экземплярами класса. «Классический» класс содержит данные, задающие свойства объектов класса, и функции, определяющие их поведение. В последнее время в класс часто добавля­ ется третья составляющая — события, на которые может реагировать объект класса 1 . Все классы библиотеки .NET, а также все классы, которые создает программист в среде .NET, имеют одного общего предка — класс object и организованы в единую иерархическую структуру. Внутри нее классы логически сгруппированы в так называемые пространства имен, которые служат для упорядочивания имен 1 Это оправдано для классов, использующихся в программах, построенных на основе со- бытийно-управляемой модели, например, при программировании для Windows. \ 14 Глава 1. Первый взгляд на платформу .NET классов и предотвращения конфликтов имен: в разных пространствах имена мо­ гут совпадать. Пространства имен могут быть вложенными, их идея аналогична знакомой вам иерархической структуре каталогов на компьютере. Любая программа, создаваемая в .NET, использует пространство имен System. В нем определены классы, которые обеспечивают базовую функциональность, напри­ мер, поддерживают выполнение математических операций, управление памятью и ввод-вывод. Обычно в одно пространство имен объединяют взаимосвязанные классы. Напри­ мер, пространство System. Net содержит классы, относящиеся к передаче данных по сети, System.Windows.Forms — элементы графического интерфейса пользовате­ ля, такие как формы, кнопки и т. д. Имя каждого пространства имен представля­ ет собой неделимую сущность, однозначно его определяющую. Последнее, о чем необходимо поговорить, прежде чем начать последовательное изучение языка С#, — среда разработки Visual Studio.NET. Среда Visual Studio.NET Среда разработки Visual Studio.NET предоставляет мощные и удобные средства написания, корректировки, компиляции, отладки и запуска приложений, исполь­ зующих .NET-совместимые языки. Корпорация Microsoft включила в платфор­ му средства разработки для четырех языков: С#, VB.NET, С+ + и J#. Платформа .NET является открытой средой. Это значит, что компиляторы для нее могут поставляться и сторонними разработчиками. К настоящему времени разработаны десятки компиляторов для .NET, например, Ada, COBOL, Delphi, Eiffel, Fortran, Lisp, Oberon, Perl и Python. Bee .NET-совместимые языки должны отвечать требованиям общеязыковой спе­ цификации (Common Language Specification, CLS), в которой описывается набор общих для всех языков характеристик. Это позволяет использовать для разра­ ботки приложения несколько языков программирования и вести полноценную межъязыковую отладку. Все программы независимо от языка используют одни и те же базовые классы библиотеки .NET. Приложение в процессе разработки называется проектом. Проект объединяет все необходимое для создания приложения: файлы, папки, ссылки и прочие ре­ сурсы. Среда Visual Studio.NET позволяет создавать проекты различных типов, например: • Windows-приложение использует элементы интерфейса Windows, включая формы, кнопки, флажки и пр.; • консольное приложение выполняет вывод «на консоль», то есть в окно команд­ ного процессора; • библиотека классов объединяет классы, которые предназначены для исполь­ зования в других приложениях; Среда Visual Studio.NET 15 • веб-приложение — это приложение, доступ к которому выполняется через браузер (например, Internet Explorer) и которое по запросу формирует веб­ страницу и отправляет ее клиенту по сети; • веб-сервис — компонент, методы которого могут вызываться через Интернет. Несколько проектов можно объединить в решение (solution). Это облегчает со­ вместную разработку проектов. Консольные приложения Среда Visual Studio.NET работает на платформе Windows и ориентирована на соз­ дание Windows- и веб-приложений, однако разработчики предусмотрели работу и с консольными приложениями. При запуске консольного приложения операци­ онная система создает так называемое консольное окно, через которое идет весь ввод-вывод программы. Внешне это напоминает работу в операционной системе в режиме командной строки, когда ввод-вывод представляет собой поток символов. Консольные приложения наилучшим образом подходят для изучения языка, так как в них не используется множество стандартных объектов, необходимых для создания графического интерфейса. В первой части курса мы будем создавать только консольные приложения, чтобы сосредоточить внимание на базовых свойствах языка С#. В следующем разделе рассмотрены самые простые действия в среде: создание и запуск на выполнение консольного приложения на С#. Более полные сведения, необходимые для работы в Visual Studio.NET, можно получить из документации или книг [8], [16]. ПРИМЕЧАНИ Е Большинство примеров, приведенных в книге, иллюстрируют базовые возможно­ сти С# и разрабатывались в интегрированной среде версии 7.1 (библиотека .NET Framework 1.1), однако вы можете работать и в более новых версиях. Программы, ко­ торые используют новые средства языка, появившиеся в спецификации версии 2.0, проверялись в Visual С# 2005 Express Edition (библиотека .NET Framework 2.0). Создание проекта. Основные окна среды Д л я создания проекта следует после запуска Visual Studio.NET1 в главном ме­ ню выбрать команду File • New • Project.... В левой части открывшегося диало­ гового окна нужно выбрать пункт Visual С# Projects, в правой — пункт Console Application. В поле Name можно ввести имя проекта, а в поле Location — место его сохранения на диске, если заданные по умолчанию значения вас не устраивают. После щелчка на кнопке ОК среда создаст решение и проект с указанным име­ нем. Примерный вид экрана приведен на рис. 1.3. В верхней части экрана располагается главное меню (с разделами File, Edit, View и т. д.) и панели инструментов (toolbars). Панелей инструментов в среде великое множество, и если включить их все (View • Toolbars...), они займут половину экрана. 1 Полезно создать для этого ярлык на рабочем столе или на панели быстрого запуска. 16 Глава 1. Первый взгляд на платформу .NET " 3 Рис. 1.3. Примерный вид экрана после создания проекта консольного приложения В верхней левой части экрана располагается окно управления проектом. Solution Explorer (если оно не отображается, следует воспользоваться командой View • Solution Explorer главного меню). В окне перечислены все ресурсы, входящие в проект: ссыл­ ки на библиотеку (System, System.Data, System.XML), файл ярлыка (App.ico), файл с исходным текстом класса (Classl.cs) и информация о сборке1 (Assemblylnfo.cs). В этом же окне можно увидеть и другую информацию, если перейти на вкладку Class View, ярлычок которой находится в нижней части окна. На вкладке Class View представлен список всех классов, входящих в приложение, их элементов и предков. ПРИМЕЧАНИ Е Небезынтересно полюбопытствовать, какие файлы создала среда для поддержки проекта. С помощью проводника Windows можно увидеть, что на заданном диске появилась папка с указанным именем, содержащая несколько других файлов и вло­ женных папок. Среди них — файл проекта (с расширением csproj), файл решения (с расширением sin) и файл с кодом класса (Classl .cs). В нижней левой части экрана расположено окио свойств Properties (если окна не видно, воспользуйтесь командой View • Properties главного меню). В окне свойств Как говорилось в предыдущем разделе, сборка является результатом работы компилято­ ра и содержит код на промежуточном языке и метаданные. Среда Visual Studio.NET 17 отображаются важнейшие характеристики выделенного элемента. Например, чтобы изменить имя файла, в котором хранится класс Classl, надо выделить этот файл в окне управления проектом и задать в окне свойств новое значение свой­ ства FileName (ввод заканчивается нажатием клавиши Enter). Основное пространство экрана занимает окно редактора, в котором располагает­ ся текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист добавляет код по мере необходимости. Ключевые (зарезервированные) слова 1 отображаются синим цветом, коммента­ рии 2 различных типов — серым и темно-зеленым, остальной текст — черным. Слева от текста находятся символы структуры: щелкнув на любом квадратике с минусом, можно скрыть соответствующий блок кода. При этом минус превра­ щается в плюс, щелкнув на котором, можно опять вывести блок на экран. Это средство хорошо визуально структурирует код и позволяет сфокусировать вни­ мание на нужных фрагментах. Заготовка консольной программы Рассмотрим каждую строку заготовки программы (листинг 1.1). Не пытайтесь сра­ зу понять абсолютно все, что в ней написано. Пока что ваша цель — изучить прин­ ципы работы в оболочке, а досконально разбираться в программе мы будем позже. Листин г 1.1 . Заготовка консольной программы using System; namespace ConsoleAppli cati onl { /// /// Summary description for Classl. /// class Classl { /// /// The main entry point for the application. /// [STAThread] static void Main(string[] args) { // // TODO: Add code to start application here // } } } Ключевые слова — это слова, имеющие специальное значение для компилятора. Их мож­ но использовать только в том смысле, в котором они определены. Комментарии предназначены для программиста и позволяют обеспечить документиро­ вание программы. 1 18 Глава 1. Первый взгляд на платформу .NET Директива using System разрешает использовать имена стандартных классов из пространства имен System непосредственно (без указания имени пространства). Ключевое слово namespace создает для проекта собственное пространство имен, названное по умолчанию ConsoleAppl icationl. Это сделано для того, чтобы можно было давать программным объектам имена, не заботясь о том, что они могут сов­ пасть с именами в других пространствах имен. Строки, начинающиеся с двух или трех косых черт, являются комментариями и предназначены для документирования текста программы. С# — объектно-ориентированный язык, поэтому написанная на нем программа представляет собой совокупность взаимодействующих между собой классов. В нашей заготовке программы всего один класс, которому по умолчанию задано имя Classl. Описание класса начинается с ключевого слова class, за которым сле­ дуют его имя и далее в фигурных скобках — список элементов класса (его дан­ ных и функций, называемых также методами). ВНИМАНИ Е Фигурные скобки являются важным элементом синтаксиса. Каждой открывающей скобке соответствует своя закрывающая, которая обычно располагается ниже по тексту с тем же отступом. Эти скобки ограничивают блок, внутри которого могут располагаться другие блоки, вложенные в него, как матрешки. Блок может приме­ няться в любом месте, где допускается отдельный оператор. В данном случае внутри класса только один элемент — метод Main. Каждое при­ ложение должно содержать метод Main — с него начинается выполнение програм­ мы. Все методы описываются по единым правилам. Упрощенный синтаксис метода: [ спецификаторы ] тип имя\_нетода ( [ параметры ] ) { тело метода: действия, выполняемые методом } ВНИМАНИ Е Для описания языка программирования в документации часто используется некото­ рый формальный метаязык, например, формулы Бэкуса—Наура или синтаксические диаграммы. Для наглядности и простоты изложения в этой книге используется широ­ ко распространенный неформальный способ описания, когда необязательные части синтаксических конструкций заключаются в квадратные скобки, текст, который необ­ ходимо заменить конкретным значением, пишется по-русски, а возможность выбо­ ра одного из нескольких элементов обозначается вертикальной чертой. Например: [ void | int ] имя\_метода(): Эта запись означает, что вместо конструкции имя\_метода необходимо указать кон­ кретное имя в соответствии с правилами языка, а перед ним может находиться либо слово voi d, либо слово i nt, либо ничего. Символ подчеркивания используется для связи слов вместо пробела, показывая, что на этом месте должен стоять один син­ таксический элемент, а не два. В тех случаях, когда квадратные скобки являются элементом синтаксиса, это оговаривается особо. Среда Visual Studio.NET 19 Таким образом, любой метод должен иметь тип, имя и тело, остальные части описания являются необязательными, поэтому мы их пока проигнорируем. Мето­ ды подробно рассматриваются в разделе «Методы» (см. с. 106). ПРИМЕЧАНИ Е Наряду с понятием «метод» часто используется другое — функция-член класса. Метод является частным случаем функции — законченного фрагмента кода, кото­ рый можно вызвать по имени. Далее в книге используются оба эти понятия. Среда заботливо поместила внутрь метода Main комментарий: // TODO: Add code to start application here Это означает: «Добавьте сюда код, выполняемый при запуске приложения». Последуем совету и добавим после строк комментария (но не в той же строке!) строку | Console.WriteLine{"Ур-ра! Зар-работало! (с) Кот Матроскин"); Здесь Console — это имя стандартного класса из пространства имен System. Его метод WriteLine выводит на экран заданный в кавычках текст. Как видите, для обращения к методу класса используется конструкция имякласса.имя\_иетода Если вы не сделали ошибку в первом же слове, то сразу после ввода с клавиату­ ры следом за словом Console символа точки среда выведет подсказку, содержа­ щую список всех доступных элементов класса Console. Выбор нужного имени выполняется либо мышью, либо клавишами управления курсором, либо вводом одного или нескольких начальных символов имени. При нажатии клавиши Enter выбранное имя появляется в тексте программы. СОВЕ Т Не пренебрегайте возможностью автоматического ввода — это убережет вас от опе­ чаток и сэкономит время. Если подсказка не появляется, это свидетельствует об ошибке в имени или в месте расположения в программе вводимого текста. Программа должна приобрести вид, приведенный в листинге 1.2 (для того чтобы вы могли сосредоточиться на структуре программы, из нее убраны все коммента­ рии и другие пока лишние для нас детали). Ничего ужасного в этом листинге нет, не правда ли? Обратите внимание на то, что после внесения изменений около имени файла на ярлычке в верхней части окна редактора появился символ \* — это означает, что текст, сохраненный на диске, и текст, представленный в окне редактора, не совпадают. Для сохранения файла воспользуйтесь командой File • Save главного меню или кнопкой Save на панели инструментов (текстовый курсор должен при этом находиться в окне редактора). Впрочем, при запуске программы среда со­ хранит исходный текст самостоятельно. 20 Глава 1. Первый взгляд на платформу .NET Листин г 1.2. Первая программа на С# using System: namespace ConsoleApplicationl { class Classl { static void MainO { Console.WriteLine( "Ур-ра! Зар-работало! (с ) Кот Матроскин" ): } } } На рис. 1.4 приведен экран после создания консольного приложения в Visual С# 2005 Express Edition. Как видите, текст заготовки приложения более лако­ ничен, чем в предыдущей версии, и практически совпадает с листингом 1.1 без учета комментариев. Рис. 1.4. Вид экрана после создания проекта консольного приложения в Visual С# 2005 Express Edition Запуск программы Самый простой способ запустить программу — нажать клавишу F5 (или выбрать в меню команду Debug • Start). Если программа написана без ошибок, то фраза Ур-ра! Зар-работало! (с) Кот Матроскин промелькнет перед вашими глазами в кон­ сольном окне, которое незамедлительно закроется. Это хороший результат, но для того чтобы пронаблюдать его спокойно, следует воспользоваться клавишами Ctrl+FS1 (или выбрать в меню команду Debug • Start Without Debugging). 1 Обозначение Ctrl+F5 означает, что следует, удерживая клавишу Ctrl, нажать F5. Рекомендации по программированию 21 После внесения изменений компилятор может обнаружить в тексте программы синтаксические ошибки. Он сообщает об этом в окне, расположенном в нижней части экрана. Давайте внесем в программу синтаксическую ошибку, чтобы впервые увидеть то, что впоследствии вам придется наблюдать многие тысячи раз. Уберите точку с запятой после оператора Console.WriteLineL.) и запустите программу заново. Вы увидите диалоговое окно с сообщением о том, что при построении приложения обнаружены ошибки, и вопросом, продолжать ли дальше (There were build errors. Continue?). Сурово ответив No, обратимся к окну вывода ошибок за разъяснениями. ПРИМЕЧАНИ Е Если сообщения об ошибке не видно, просмотрите содержимое окна с помощью по­ лосы прокрутки, пытаясь найти в нем слово «error». Двойной щелчок на строке с сообщением об ошибке error CS1002 ; expected (означающее, что здесь ожидалась точка с запятой) подсвечивает неверную стро­ ку в программе. Дл я получения дополнительных пояснений можно нажать кла­ вишу F1, при этом в окне редактора появится новая вкладка с соответствующей страницей из справочной системы. Для возвращения к коду класса следует щелк­ нуть на ярлычке с именем файла в верхней строке редактора. Другой способ получения справки — использовать окно Dynamic Help, распо­ ложенное на том же месте, что и окно свойств Properties (его ярлычок находится в нижней строке окна). Содержимое этого окна динамически меняется в зависи­ мости от того, какой элемент выделен. Для получения справки надо щелкнуть на соответствующей ссылке, и страница справочной системы будет отображена на отдельной вкладке окна редактора (в Visual С# 2005 Express Edition справочная информация появляется в отдельном окне). Теперь, когда вы получили общее представление о платформе .NET, можно, на­ конец, приступить к планомерному изучению языка С#. Рекомендации по программированию В этой главе дается очень краткое введение в интересную и обширную тему — платформу .NET. Для более глубокого понимания механизмов ее функциониро­ вания настоятельно рекомендуется изучить дополнительную литературу (напри­ мер, [5], [19], [26], [27]) и публикации в Интернете. Среда разработки Visual Studio.NET предоставляет программисту мощные и удоб­ ные средства написания, корректировки, компиляции, отладки и запуска приложе­ ний, описанию которых посвящены целые книги. В процессе изучения языка С# желательно постепенно изучать эти возможности, ведь чем лучше вы будете вла­ деть инструментом, тем эффективнее и приятнее будет процесс программирования. Глава 2 Основные понятия языка В этой главе рассматриваются элементарные «строительные блоки» языка С#, вводится понятие типа данных и приводится несколько классификаций типов. Состав языка Язык программирования можно уподобить очень примитивному иностранному языку с жесткими правилами без исключений. Изучение иностранного языка обычно начинают с алфавита, затем переходят к словам и законам построения фраз, и только в результате длительной практики и накопления словарного запа­ са появляется возможность свободно выражать на этом языке свои мысли. При­ мерно так же поступим и мы при изучении языка С#. Алфавит и лексемы Все тексты на языке пишутся с помощью его алфавита. Например, в русском языке один алфавит (набор символов), а в албанском — другой. В С# использу­ ется кодировка символов Unicode. Давайте разберемся, что это такое. Компьютер умеет работать только с числами, и для того чтобы можно было хра­ нить в его памяти текст, требуется определить, каким числом будет представлять­ ся (кодироваться) каждый символ. Соответствие между символами и кодирую­ щими их числами называется кодировкой, или кодовой таблицей (character set). Существует множество различных кодировок символов. Например, в Windows часто используется кодировка ANSI , а конкретно — СР1251. Каждый символ представляется в ней одним байтом (8 бит), поэтому в этой кодировке можно од­ новременно задать только 256 символов. В первой половине кодовой таблицы находятся латинские буквы, цифры, знаки арифметических операций и другие распространенные символы. Вторую половину занимают символы русского алфа- Состав языка 23 вита. Если требуется представлять символы другого национального алфавита (например, албанского), необходимо использовать другую кодовую таблицу. Кодировка Unicode позволяет представить символы всех существующих алфави­ тов одновременно, что коренным образом улучшает переносимость текстов. Каждому символу соответствует свой уникальный код. Естественно, что при этом для хранения каждого символа требуется больше памяти. Первые 128 Unicode- символов соответствуют первой части кодовой таблицы ANSI. Алфавит С# включает: • буквы (латинские и национальных алфавитов) и символ подчеркивания (\_), который употребляется наряду с буквами; • цифры; • специальные символы, например +, \*, { и &; • пробельные символы (пробел и символы табуляции); • символы перевода строки. Из символов составляются более крупные строительные блоки: лексемы, дирек­ тивы препроцессора и комментарии. Лексема (token1 ) —это минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды лексем: • имена {идентификаторы); • ключевые слова; • знаки операций; • разделители; • литералы (константы). Лексемы языка программирования аналогичны словам естественного языка. На­ пример, лексемами являются число 128 (но не его часть 12), имя Vasia, ключевое слово goto и знак операции сложения +. Далее мы рассмотрим лексемы подробнее. Директивы препроцессора пришли в С# из его предшественника — языка С++ . Препроцессором называется предварительная стадия компиляции, на которой формируется окончательный вид исходного текста программы. Например, с по­ мощью директив (инструкций, команд) препроцессора можно включить или вы­ ключить из процесса компиляции фрагменты кода. Директивы препроцессора не играют в С# такой важной роли, как в С++ . Мы рассмотрим их в свое вре­ мя—в разделе «Директивы препроцессора» (см. с. 287). Комментарии предназначены для записи пояснений к программе и формирова­ ния документации. Правила записи комментариев описаны далее в этом разделе. Из лексем составляются выражения и операторы. Выражение задает правило вычисления некоторого значения. Например, выражение а + b задает правило вычисления суммы двух величин. Часто это слово ленятся переводить и пишут просто «токен». 24 Глава 2. Основные понятия языка ПРИМЕЧАНИ Е Компилятор выполняет перевод программы в исполняемый файл на языке IL за две последовательные фазы: лексического и синтаксического анализа. При лексиче­ ском анализе из потока символов, составляющих исходный текст программы, выделя­ ются лексемы (по другим лексемам, разделителям, пробельным символам и переводам строки). На этапе синтаксического анализа программа переводится в исполняемый код. Кроме того, компилятор формирует сообщения о синтаксических ошибках. Оператор задает законченное описание некоторого действия, данных или эле­ мента программы. Например: int а: Это — оператор описания целочисленной переменной а. Идентификаторы Имена в программах служат той же цели, что и имена в мире людей, — чтобы об­ ращаться к программным объектам и различать их, то есть идентифицировать. Поэтому имена также называют идентификаторами. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, sysop, SySoP и SYS0P — три разных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Длина идентификатора не ограничена. Пробелы внутри имен не до­ пускаются. В идентификаторах С# разрешается использовать помимо латинских букв бук­ вы национальных алфавитов. Например, Пёсик или gg являются правильными идентификаторами1 . Более того, в идентификаторах можно применять даже так называемые escape-последовательности Unicode, то есть представлять символ с помощью его кода в шестнадцатеричном виде с префиксом \и, например, \u00F2. ПРИМЕЧАНИ Е Примеры неправильных имен: 21 ate, Big gig, Б#г; первое начинается с цифры, вто­ рое и третье содержат недопустимые символы (пробел и #). Имена даются элементам программы, к которым требуется обращаться: перемен­ ным, типам, константам, методам, меткам и т. д. Идентификатор создается на этапе объявления переменной (метода, типа и т. п.), после.этого его можно ис­ пользовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее: • идентификатор не должен совпадать с ключевыми словами (см. следующий раздел)2 ; 1 Другое дело, захочется ли вам при вводе текста программы все время переключать ре­ гистр. Пример программы, в которой в именах используется кириллица, приведен на с. 182. 2 Впрочем, для использования ключевого слова в качестве идентификатора его достаточ­ но предварить символом @. Например, правильным будет идентификатор @i f. Состав языка 25 • не рекомендуется начинать идентификаторы с двух символов подчеркивания, поскольку такие имена зарезервированы для служебного использования. Д ля улучшения читабельности программы следует давать объектам осмысленные имена, составленные в соответствии с определенными правилами. Понятные и согласованные между собой имена — основа хорошего стиля программирова­ ния. Существует несколько видов так называемых нотаций — соглашений о пра­ вилах создания имен. В нотации Паскаля каждое слово, составляющее идентификатор, начинается с прописной буквы, например, MaxLength, MyFuzzyShooshpanchik. Венгерская нотация (ее предложил венгр по национальности, сотрудник компа­ нии Microsoft) отличается от предыдущей наличием префикса, соответствующе­ го типу величины, например, i MaxLength, IpfnMyFuzzyShooshpanchik. Согласно нотации Camel, с прописной буквы начинается каждое слово, составляю­ щее идентификатор, кроме первого, например, maxLength, myFuzzyShooshpanchik. Человеку с богатой фантазией абрис имени может напоминать верблюда, откуда и произошло название этой нотации. Еще одна традиция — разделять слова, составляющие имя, знаками подчеркива­ ния: max\_length, my\_fuzzy\_shooshpanchik, при этом все составные части начинаются со строчной буквы. В С# для именования различных видов программных объектов чаще всего ис­ пользуются две нотации: Паскаля и Camel. Многобуквенные идентификаторы в примерах этой книги соответствуют рекомендациям, приведенным в специ­ фикации языка. Кроме того, в примерах для краткости часто используются одно- буквенные имена. В реальных программах такие имена можно применять только в ограниченном наборе случаев. Ключевые слова Ключевые слова — это зарезервированные идентификаторы, которые имеют специ­ альное значение для компилятора. Их можно использовать только в том смыс­ ле, в котором они определены. Список ключевых слов С# приведен в табл. 2.1. Знаки операций и разделители Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Например, в вы­ ражении а += b знак += является знаком операции, а а и b — операндами. Симво­ лы, составляющие знак операций, могут быть как специальными, например, &&, | и < > ? ++ | | «..> > тт\_ ]=<=>=+ = -= \*= /= %= &= | = А = « = » = -> Литералы Литералами, или константами, называют неизменяемые величины. В С# есть логические, целые, вещественные, символьные и строковые константы, а также константа null. Компилятор, выделив константу в качестве лексемы, относит ее к одному из типов данных по ее внешнему виду. Программист может задать тип константы и самостоятельно 1 . Описание и примеры констант каждого типа приведены в табл. 2.2. Примеры, иллюстрирующие наиболее часто употребляемые формы констант, выделены полу­ жирным шрифтом (при первом чтении можно обратить внимание только на них). Определение типа будет введено чуть позже в этой главе, а пока можно использовать обыденное значение этого слова. Состав языка 27 Таблица 2.2. Константы в С# Константа Описание Примеры Логическая true (истина) или false (ложь) Целая Десятичная: последовательность десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за которой может следовать суффикс (U, u, L, 1, UL, Ul, uL, ul, LU, Lu, 1U, lu) Шестнадцатеричная: символы Ox или OX, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, А, В, С, D, Е, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, 1, UL, Ul, uL, ul, LU, Lu, 1U, lu) Вещественная С фиксированной точкой1 : [цифры] [.] [цифры] [суффикс] Суффикс — один из символов F, f, D, d, М, m С порядком: [цифры][.][цифры]{Е|е}[+|-][цифры] [суффикс] Суффикс — один из символов F, f, D, d, М, m Символьная Символ, заключенный в апострофы Строковая Последовательность символов, заключенная в кавычки Константа null Ссылка, которая не указывает ни на какой объект true false 8 8u О 199226 OLu 199226L ОхА 0x188 0X00FF OxAU 0xlB8LU 0X00FF1 5.7 .001 35 5.7F .OOld 35 5F .OOlf 35m 0.2E6 0.2E6D 5E10 .lle+3 .lle-3 5E-10 'A' V '\* ' '\0' '\n' '\xF' '\x74' '\uA81B' "Здесь был Vasia" "иЗначение г = \0xF5 \n" "Здесь был \u0056\u0061" "С: WtempWfilel.txt" @"С:\temp\filel.txt" null Рассмотрим табл. 2.2 более подробно. Логических литералов всего два. Они широ­ ко используются в качестве признаков наличия или отсутствия чего-либо. Целые литералы могут быть представлены либо в десятичной, либо в шестнадца- теричной системе счисления, а вещественные — только в десятичной системе, но Напомню, что квадратные скобки при описании означают необязательность заключенной в них конструкции. 28 Глава 2. Основные понятия языка в двух формах: с фиксированной точкой и с порядком. Вещественная константа с порядком представляется в виде мантиссы и порядка. Мантисса записывается слева от знака экспоненты (Е или е), порядок — справа от знака. Значение кон­ станты определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10 (например, 1.3е2 = 1,3 • 102 = 130). При записи вещест­ венного числа могут быть опущены либо целая часть, либо дробная, но, конечно, не обе сразу. ВНИМАНИ Е Пробелы внутри числа не допускаются. Для отделения целой части от дробной ис­ пользуется не запятая, а точка. Символ Е не представляет собой знакомое всем из математики число е, а указывает, что далее располагается степень, в которую нужно возвести число 10. Если требуется сформировать отрицательную целую или вещественную констан­ ту, то перед ней ставится знак унарной операции изменения знака (-) , например: -218, -022, -ОхЗС, -4.8, -0.1е4. Когда компилятор распознает константу, он отводит ей место в памяти в соответст­ вии с ее видом и значением. Если по каким-либо причинам требуется явным обра­ зом задать, сколько памяти следует отвести под константу, используются суффик­ сы, описания которых приведены в табл. 2.3. Поскольку такая необходимость возникает нечасто, эту информацию можно при первом чтении пропустить. Таблица 2.3. Суффиксы целых и вещественных констант Суффикс Значение L, 1 Длинное целое (long) U, и Беззнаковое целое (unsigned) F, f Вещественное с одинарной точностью (float) D, d Вещественное с двойной точностью (double) М, m Финансовое десятичного типа (decimal) Допустимые диапазоны значений целых и вещественных констант в зависимо­ сти от префикса мы рассмотрим немного позже в этой главе. Символьная константа представляет собой любой символ в кодировке Unicode. Символьные константы записываются в одной из четырех форм, представлен­ ных в табл. 2.2: • «обычный» символ, имеющий графическое представление (кроме апострофа и символа перевода строки), — 'А', 'ю', '\*' ; • управляющая последовательность — '\0', '\п'; • символ в виде шестнадцатеричного кода — 'VxF', '\х74'; • символ в виде escape-последовательности Unicode — '\uA81B'. Состав языка 29 Управляющей последовательностью, или простой escape-последовательностью, называют определенный символ, предваряемый обратной косой чертой. Управ­ ляющая последовательность интерпретируется как одиночный символ и исполь­ зуется для представления: • кодов, не имеющих графического изображения (например, \п — переход в на­ чало следующей строки); • символов, имеющих специальное значение в строковых и символьных лите­ ралах, например, апострофа '. В табл. 2.4 приведены допустимые значения последовательностей. Если непосред­ ственно за обратной косой чертой следует символ, не предусмотренный таблицей, возникает ошибка компиляции. Таблица 2.4. Управляющие последовательности в С# Вид Наименование \а Звуковой сигнал \Ь Возврат на шаг \f Перевод страницы (формата) \п Перевод строки \г Возврат каретки \t Горизонтальная табуляция \v Вертикальная табуляция \\ Обратная косая черта V Апостроф V Кавычка \0 Нуль-символ Символ, представленный в виде шестнадцатеричного кода, начинается с пре­ фикса \0х, за которым следует код символа. Числовое значение должно нахо­ диться в диапазоне от 0 до 216 - 1, иначе возникает ошибка компиляции. Escape-последовательности Unicode служат для представления символа в ко­ дировке Unicode с помощью его кода в шестнадцагеричном виде с префиксом \и или \U, например, \u00F2, \U00010011. Коды в диапазоне от \U10000 до \U10FFFF представляются в виде двух последовательных символов; коды, превышающие \U10FFFF, не поддерживаются. Управляющие последовательности обоих видов могут использоваться и в стро­ ковых константах, называемых иначе строковыми литералами. Например, если требуется вывести несколько строк, можно объединить их в один литерал, отде­ лив одну строку от другой символами \п: "Никто не доволен своейЛпвнешностью, но каждый доволен\псвоим умом" 30 Глава 2. Основные понятия языка Этот литерал при выводе будет выглядеть так: Никто не доволен своей внешностью, но каждый доволен своим умом Другой пример: если внутри строки требуется использовать кавычку, ее предва­ ряют косой чертой, по которой компилятор отличает ее от кавычки, ограничи­ вающей строку: "Издательский дом \"Питер\"" Как видите, строковые литералы с управляющими символами несколько теряют в читабельности, поэтому в С# введен второй вид литералов — дословные лите­ ралы (verbatim strings). Эти литералы предваряются символом @, который отклю­ чает обработку управляющих последовательностей и позволяет получать строки в том виде, в котором они записаны. Например, два приведенных выше литерала в дословном виде выглядят так: @"Никто не доволен своей внешностью, но каждый доволен своим умом" @"Издательский дом "Питер"" Чаще всего дословные литералы применяются в регулярных выражениях и при задании полного пути файла, поскольку в нем присутствуют символы обратной косой черты, которые в обычном литерале пришлось бы представлять с помо­ щью управляющей последовательности. Сравните два варианта записи одного и того же пути: "С: WappWbi nWdebugWa. exe" @"С:\app\bin\debug\a.exe" Строка может быть пустой (записывается парой смежных двойных кавычек "") , пустая символьная константа недопустима. Константа nul 1 представляет собой значение, задаваемое по умолчанию для вели­ чин так называемых ссылочных типов, которые мы рассмотрим далее в этой главе. Комментарии Комментарии предназначены для записи пояснений к программе и формирова­ ния документации. Компилятор комментарии игнорирует. Внутри комментария можно использовать любые символы. В С# есть два вида комментариев: одно­ строчные и многострочные. Однострочный комментарий начинается с двух символов прямой косой черты (// ) и заканчивается символом перехода на новую строку, многострочный заклю­ чается между символами-скобками /\* и \*/ и может занимать часть строки, целую строку или несколько строк. Комментарии не вкладываются друг в друга: симво­ лы // и /\* не обладают никаким специальным значением внутри комментария. I данных 31 Me того, в языке есть еще одна разновидность комментариев, которые начи- !тся с трех подряд идущих символов косой черты (///) . Они предназначены формирования документации к программе в формате XML . Компилятор лекает эти комментарии из программы, проверяет их соответствие правилам шисывает их в отдельный файл. Правила задания комментариев этого вида рассмотрим в главе 15. /ты данных шые, с которыми работает программа, хранятся в оперативной памяти. Есте- енно, что компилятору необходимо точно знать, сколько места они занимают, : именно закодированы и какие действия с ними можно выполнять. Все это ается при описании данных с помощью типа. г данных однозначно определяет: внутреннее представление данных, а следовательно, и множество их возмож­ ных значений; допустимые действия над данными (операции и функции). пример, целые и вещественные числа, даже если они занимают одинаковый ьем памяти, имеют совершенно разные диапазоны возможных значений; целые ела можно умножать друг на друга, а, например, символы — нельзя. ждое выражение в программе имеет определенный тип. Величин, не имеющих какого типа, не существует. Компилятор использует информацию о типе при оверке допустимости описанных в программе действий. 1мять, в которой хранятся данные во время выполнения программы, делится две области: стек (stack) и динамическая область, или хип (heap)1 . Стек ис- льзуется для хранения величин, память под которые выделяет компилятор, в динамической области память резервируется и освобождается во время вы- шнения программы с помощью специальных команд. Основным местом для мнения данных в С# является хип. лассификация типов юбая информация легче усваивается, если она «разложена по полочкам». По- ому, прежде чем перейти к изучению конкретных типов языка С#, рассмотрим i классификацию. Типы можно классифицировать по разным признакам. ели принять за основу строение элемента, все типы можно разделить на простые ie имеют внутренней структуры) и структурированные (состоят из элементов эугих типов). По своему «создателю» типы можно разделить на встроенные В русскоязычной литературе для этого термина часто используют романтичный сино­ ним «куча». А в куче встречается «мусор», что означает не то, что вы могли подумать, а «неиспользуемые величины» — впрочем, пока мы оставим мусор без внимания! 32 (стандартные) и определяемые программистом (рис. 2.1). Дл я данных статиче­ ского типа память выделяется в момент объявления, при этом ее требуемый объ­ ем известен. Для данных динамического типа размер данных в момент объявле­ ния может быть неизвестен, и память под них выделяется по запросу в процессе выполнения программы. Рис. 2.1 . Различные классификации типов данных С# Встроенные типы Встроенные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Если же программист определяет собственный тип данных, он описывает его характеристики и сам дает ему имя, которое затем применяется точно так же, как имена стандартных типов. Описание собственного типа дан­ ных должно включать всю информацию, необходимую для его использования, а именно внутреннее представление и допустимые действия. Встроенные типы С# приведены в табл. 2.5. Они однозначно соответствуют стан­ дартным классам библиотеки .NET, определенным в пространстве имен System Как видно из таблицы, существуют несколько вариантов представления целы? и вещественных величин. Программист выбирает тип каждой величины, исполь зуемой в программе, с учетом необходимого ему диапазона и точности представ ления данных. Целые типы, а также символьный, вещественные и финансовый типы можн объединить под названием арифметических типов. Внутреннее представление величины целого типа — целое число в двоично коде. В знаковых типах старший бит числа интерпретируется как знаковый (О положительное число, 1 — отрицательное). Отрицательные числа чаще все представляются в так называемом дополнительном коде. Дл я преобразован! числа в дополнительный код все разряды числа, за исключением знаковог иинертируются, затем к числу прибавляется единица, и знаковому биту то/ присваивается единица. Беззнаковые типы позволяют представлять тольк положительные числа, поскольку старший разряд рассматривается как част! кода числа. 33 i. Встроенные типы С# Ключевое слово Тип .NET Диапазон значений Описание Размер, битов тип bool Boolean true, false [ sbyte SByte От-128 до 127 Со знаком 8 byte Byte От 0 до 255 Без знака 8 short Intl6 От -32 768 до 32 767 Со знаком 16 ushort UIntl6 От 0 до 65 535 Без знака 16 int Int32 От -2 • 109 до 2 • 109 Со знаком 32 uint UInt32 От 0 до 4 • 10» Без знака 32 long Int64 От -9 х 1018 до 9 • 1018 Со знаком 64 ulong UInt64 От 0 до 18 • 1018 Без знака 64 й char Char От U+0000 до U+ffff Unicode- символ 16 ые 1 float Single От 1.5 • 10"45 до 3.4 • 1038 7 цифр 32 double Double От 5.0 • 10324 до 1.7 • 10308 15-16 цифр 64 й decimal Decimal От 1.0 • 10~28 до 7.9 • 1028 28-29 цифр 128 string String Длина ограничена объемом доступной памяти Строка из Unicode- символов object Object Можно хранить все что угодно Всеобщий предок Ч ИЕ под величину отведено я двоичных разрядов, то в ней можно представить 2" 1чных сочетаний нулей и единиц. Если старший бит отведен под знак, то диа- 1 возможных значений величины — [-2""1 ,2 п ~ 1 - 1], а если все разряды исполь- ся для представления значения, диапазон смещается в область положительных 1 и равен [0, 2я - 1] (см. табл. 2.5). ные типы, или типы данных с плавающей точкой, хранятся в памяти а иначе, чем целочисленные. Внутреннее представление вещественного гит из двух частей — мантиссы и порядка, каждая часть имеет знак. Дли- ы определяет точность числа, а длина порядка — его диапазон. В первом ши это можно представить себе так: например, для числа 0,381 • 104 ственных и финансового типов в таблице приведены абсолютные величины ных и максимальных значений. 34 Глава 2. Основные понятия язы хранятся цифры мантиссы 381 и порядок 4, для числа 560,3 • 102 — мантис 5603 и порядок 5 (мантисса нормализуется), а число 0,012 представлено как 12 и Конечно, в этом примере не учтены система счисления и другие особенност Все вещественные типы могут представлять как положительные, так и отрив тельные числа. Чаще всего в программах используется тип double, поскольку е диапазон и точность покрывают большинство потребностей. Этот тип имеют i щественные литералы и многие стандартные математические функции. ПРИМЕЧАНИ Е Обратите внимание на то, что при одинаковом количестве байтов, отводимых г величины типа f 1 oat и i nt, диапазоны их допустимых значений сильно различав ся из-за внутренней формы представления. То же самое относится к 1 ong и doub Тип decimal предназначен для денежных вычислений, в которых критичны они ки округления. Как видно из табл. 2.5, тип float позволяет хранить однов} менно всего 7 значащих десятичных цифр, тип double — 15-16. При вычислени ошибки округления накапливаются, и при определенном сочетании значений ; даже может привести к результату, в котором не будет ни одной верной значат цифры! Величины типа decimal позволяют хранить 28-29 десятичных разряд Тип decimal не относится к вещественным типам, у них различное внутрею представление. Величины денежного типа даже нельзя использовать в одном ъ ражении с вещественными без явного преобразования типа. Использование личин финансового типа в одном выражении с целыми допускается. Любой встроенный тип С# соответствует стандартному классу библиотеки .N1 определенному в пространстве имен System. Везде, где используется имя встроен] го типа, его можно заменить именем класса библиотеки. Это значит, что у вст: енных типов данных С# есть методы и поля. С их помощью можно, наприм получить минимальные и максимальные значения для целых, символьных, < данных определенного типа. Во время выполнения программы значение г менной можно изменять. Все переменные, используемые в программе, дол быть описаны явным образом. При описании для каждой переменной зада] ее имя и тип. Пример описания целой переменной с именем а и вещественной перемени int а; float х; Имя переменной служит для обращения к области памяти, в которой хран значение переменной. Имя дает программист. Оно должно соответствовать вилам именования идентификаторов С#, отражать смысл хранимой вели 1 и быть легко распознаваемым. Например, если в программе вычисляется i чество каких-либо предметов, лучше назвать соответствующую переме! quantity или, на худой конец, kolich, но не, скажем, A, tl7\_xz или prikol. СОВЕ Т Желательно, чтобы имя не содержало символов, которые можно перепутат! с другом, например 1 (строчная буква L) и I (прописная буква i) . эменные 39 переменной выбирается, исходя из диапазона и требуемой точности пред- вления данных. Например, нет необходимости заводить переменную вещест- ного типа для хранения величины, которая может принимать только целые «ния , — хотя бы потому, что целочисленные операции выполняются гораздо трее. I объявлении можно присвоить переменной некоторое начальное значение, !сть инициализировать ее, например: a, b = 1; it х = 0.1. у = 0.If: сь описаны: теременная а типа i nt, начальное значение которой не присваивается; геременная b типа i nt, ее начальное значение равно 1; теременные х и у типа f 1 oat, которым присвоены одинаковые начальные значе- И1я 0.1. Разница между ними состоит в том, что для инициализации перемен- юй х сначала формируется константа типа double (это тип, присваиваемый по шолчанию литералам с дробной частью), а затем она преобразуется к типу 'loat; переменной у значение 0.1 присваивается без промежуточного преобра- ювания. г инициализации можно использовать не только константу, но и выражение — шое, чтобы на момент описания оно было вычисляемым, например: b = 1, а = 100: х = b \* а + 25; щиализировать переменную прямо при объявлении не обязательно, но перед , как ее использовать в вычислениях, это сделать все равно придется, иначе пилятор сообщит об ошибке. 1 МАНИ Е Рекомендуется всегда инициализировать переменные при описании. ючем, иногда эту работу делает за программиста компилятор, это зависит от гонахождения описания переменной. Как вы помните из главы 1, программа Z# состоит из классов, внутри которых описывают методы и данные. Пере- ные, описанные непосредственно внутри класса, называются полями класса. автоматически присваивается так называемое «значение по умолчанию» — правило, это 0 соответствующего типа. «менные, описанные внутри метода класса, называются локальными перемен- ш. Их инициализация возлагается на программиста. 1МЕЧАНИ Е В этой главе рассматриваются только локальные переменные простых встроенных типов данных. 40 Глава 3. Переменные, операции и выражен Так называемая область действия переменной, то есть область программы, г, можно использовать переменную, начинается в точке ее описания и длит< до конца блока, внутри которого она описана. Блок — это код, заключеннь в фигурные скобки. Основное назначение блока — группировка операторо В С# любая переменная описана внутри какого-либо блока: класса, метода ш блока внутри метода. Имя переменной должно быть уникальным в области ее действия. Область де ствия распространяется на вложенные в метод блоки, из этого следует, что i вложенном блоке нельзя объявить переменную с таким же именем, что и в охв тывающем его, например: class X // начало описания класса X { int А; // поле А класса X int В; // поле В класса X void Y() // метод Y класса { int С; // локальная переменная С, область действия - метод Y int А; // локальная переменная А (НЕ конфликтует с полем А) { // ============ вложенный блок 1 ============ int D; // локальная переменная D, область действия - этот блок // int А: недопустимо! Ошибка компиляции, конфликт с локальной // переменной А С = В: // присваивание переменной С поля В класса X (\*\* ) С = this.А: // присваивание переменной С поля А класса X (\*\*\* ) } // ============ конец блока 1 =============== { // ============ вложенный блок 2 ============ int D; // локальная переменная D, область действия - этот блок } // ============ конец блока 2 =============== } // конец описания метода Y класса } // конец описания класса X Давайте разберемся в этом примере. В нем описан класс X, содержащий TJ элемента: поле А, поле В и метод Y. Непосредственно внутри метода Y заданы д. локальные переменные — С и А. Внутри метода класса можно описывать переменную с именем, совпадаюнн с полем класса, потому что существует способ доступа к полю класса с пом щью ключевого слова this (это иллюстрирует строка, отмеченная символами \*\*4 Таким образом, локальная переменная А не «закрывает» поле А класса X, а вот п пытка описать во вложенном блоке другую локальную переменную с тем > именем окончится неудачей (эти строки закомментированы). Если внутри метода нет локальных переменных, совпадающих с полями класс к этим полям можно обратиться в методе непосредственно (см. строку, помеченн} Именованные константы 41 символами \*\*) . Две переменные с именем D не конфликтуют между собой, по­ скольку блоки, в которых они описаны, не вложены один в другой. СОВЕ Т Как правило, переменным с большой областью действия даются более длинные имена, а для переменных, вся «жизнь» которых — несколько строк исходного тек­ ста, хватит и одной буквы с комментарием при объявлении. В листинге 3.1 приведен пример программы, в которой описываются и выводят­ ся на экран локальные переменные. Листин г 3.1 . Описание переменных using System; namespace ConsoleAppl icationl { class Classl { static void MainO { int i = 3; double у = 4.12; decimal d = 600m; string s = "Вася"; Console.Write( "i = " ) Console.WriteC "y = " ) Console.Write( "d = " ) Console.Write( "s - " ) } } } Как вы догадались, метод Write делает то же самое, что и WriteLine, но не перево­ дит строку. Более удобные способы вывода рассмотрены в конце этой главы в разделе «Простейший ввод-вывод» (см. с. 59). ВНИМАНИ Е . Переменные создаются при входе в их область действия (блок) и уничтожаются при выходе. Это означает, что после выхода из блока значение переменной не сохра­ няется. При повторном входе в этот же блок переменная создается заново. Именованные константы Можно запретить изменять значение переменной, задав при ее описании ключе­ вое слово const, например: const int b = 1; const float x = 0:1, у = 0.If: // const распространяется на обе переменные Consol е. Writel\_ine( i ) Consolе.WriteLine( у ) Console.Writel\_ine( d ) Console.Writel\_ine( s ) 42 Глава 3. Переменные, операции и выражен!- Такие величины называют именованными константами, или просто констаь тами. Они применяются для того, чтобы вместо значений констант можно был использовать в программе их имена. Это делает программу более понятной и об легчает внесение в нее изменений, поскольку изменить значение достаточн только в одном месте программы. ПРИМЕЧАНИ Е Улучшение читабельности происходит только при осмысленном выборе имен кон стант. В хорошо написанной программе вообще не должно встречаться иных 4nce j кроме 0 и 1, все остальные числа должны задаваться именованными константам] с именами, отражающими их назначение. Именованные константы должны обязательно инициализироваться при описа нии. При инициализации можно использовать не только константу, но и выра жение — главное, чтобы оно было вычисляемым на этапе компиляции, например const int b = 1. а = 100: const int х = b \* а + 25: Операции и выражения Выражение — это правило вычисления значения. В выражении участвуют one ранды, объединенные знаками операций. Операндами простейшего выражение могут быть константы, переменные и вызовы функций. Например, а + 2 — это выражение, в котором + является знаком операции, а а и 2 - операндами. Пробелы внутри знака операции, состоящей из нескольких сим­ волов, не допускаются. По количеству участвующих в одной операции опе­ рандов операции делятся на унарные, бинарные и тернарную. Операции С# при­ ведены в табл. 3.11 . Таблица 3.1 . Операции С# Категория Знак операции Название Описание Первичные Доступ к элементу С. 105 х () Вызов метода или делегата С. 108, 221 х [] Доступ к элементу • С. 127 Х++ Постфиксный инкремент С. 47 Х-- Постфиксный декремент С. 47 new Выделение памяти С. 48 typeof Получение типа С. 280 checked Проверяемый код С. 46 unchecked Непроверяемый код С. 46 1 В этой таблице символ х призван показать расположение операнда и не является частью знака операции. !рации и выражения 43 тегория Знак операции Название Описание арные + Унарный плюс - Унарный минус С. 48 (арифметическое отрицание) j Логическое отрицание С. 48 - Поразрядное отрицание С. 48 ++Х Префиксный инкремент С. 47 --Х Префиксный декремент С. 47 (тип ) х Преобразование типа С. 49 ^льтипликативные \* Умножение С. 50 та умножения) / Деление С. 50 1 Остаток от деления С. 50 щитивные (типа + Сложение С. ожения) - Вычитание С. 53 Сбита « Сдвиг влево С. 54 » Сдвиг вправо С. 54 гношения < Меньше С. 54 проверки тина > Больше С. 54 <= Меньше или равно С. 54 >= Больше или равно С. 54 is Проверка принадлежности типу С. 194 as Приведение типа С. 194 роверки на — Равно С. 54 [венство 1 = Не равно С. 54 оразрядные & Поразрядная конъюнкция (И ) С. 55 )гические А. Поразрядное исключающее ИЛИ С. 55 1 Поразрядная дизъюнкция (ИЛИ ) С. 55 словные логические && Логическое И С. 56 I I Логическое ИЛ И С. 56 словная ? ; Условная операция С. 56 рисваивания = Присваивание С. 56 \*= Умножение с присваиванием / = Деление с присваиванием 1- Остаток отделения с присваиванием += Сложение с присваиванием продолжение & 44 Глава 3. Переменные, операции и выражен Таблица 3.1 (продолжение) Категория Знак операции Название Описани< &• « = » = Вычитание с присваиванием Сдвиг влево с присваиванием Сдвиг вправо с присваиванием Поразрядное И с присваиванием Поразрядное исключающее ИЛ И с присваиванием Поразрядное ИЛИ с присваиванием ПРИМЕЧАНИ Е В версию С# 2.0 введена операция объединения ??, которая рассматривается в г: ве 13 (см. раздел «Обнуляемые типы», с. 309). Операции в выражении выполняются в определенном порядке в соответств] с приоритетами, как и в математике. В табл. 3.1 операции расположены по уб; ванию приоритетов, уровни приоритетов разделены в таблице горизонтальны? линиями. Результат вычисления выражения характеризуется значением и типом. Напр мер, пусть а и b — переменные целого типа и описаны так: int а = 2, b = 5; Тогда выражение а + b имеет значение 7 и тип i nt, а выражение а = b имеет знач ние, равное помещенному в переменную а (в данному случае — 5), и тип, совп дающий с типом этой переменной. Если в одном выражении соседствуют несколько операций одинакового пр оритета, операции присваивания и условная операция выполняются спра налево, остальные — слева направо. Дл я изменения порядка выполнения on раций используются круглые скобки, уровень их вложенности практически ограничен. Например, а + b + с означает (а + Ь) + с, а а = b = с означает а = (Ь = с). То ее сначала вычисляется выражение b = с, а затем его результат становится правь операндом для операции присваивания переменной а. ПРИМЕЧАНИ Е Часто перед выполнением операции требуется вычислить значения операнде Например, в выражении F(i) + G(i++) \* H(i) сначала вызываются функции F и Н, а затем выполняются умножение и сложение. Операнды всегда вычисляют слева направо независимо от приоритетов операций, в которых они участвуя Кстати, в приведенном примере метод Н вызывается с новым значением i (увел ченным на 1). Тип результата выражения в общем случае формируется по правилам, котор! описаны в следующем разделе. перации и выражения 45 1реобразования встроенных арифметических ипов-значений [ри вычислении выражений может возникнуть необходимость в преобразовании япов. Если операнды, входящие в выражение, одного типа и операция для этого шпа определена, то результат выражения будет иметь тот же тип. ели операнды разного типа и/или операция для этого типа не определена, пе- ед вычислениями автоматически выполняется преобразование типа по прави- ам, обеспечивающим приведение более коротких типов к более длинным для охранения значимости и точности. Автоматическое (неявное) преобразование озможно не всегда, а только если при этом не может случиться потеря значимости. ;сли неявного преобразования из одного типа в другой не существует, програм- ист может задать явное преобразование типа с помощью операции (тип)х. Его езультат остается на совести программиста. Явное преобразование рассматри- ается в этой главе немного позже. НИМАНИ Е Арифметические операции не определены для более коротких, чем i nt, типов. Это означает, что если в выражении участвуют только величины типов sbyte, byte, short и ushort, перед выполнением операции они будут преобразованы в i nt. Таким образом, результат любой арифметической операции имеет тип не менее int. [равила неявного преобразования иллюстрирует рис. 3.1. Если один из операн- ов имеет тип, изображенный на более низком уровне, чем другой, то он приво- ится к типу второго операнда при наличии пути между ними. Если пути нет, озникает ошибка компиляции. Если путей несколько, выбирается наиболее ко- откий, не содержащий пунктирных линий. Преобразование выполняется не по- ледовательно, а непосредственно из исходного типа в результирующий. double Z [ float decimal ] long ulong Рис. 3.1 . Неявные арифметические преобразования типов 46 Глава 3. Переменные, операции и выражения Преобразование более коротких, чем i nt, типов выполняется при присваивании. Обратите внимание на то, что неявного преобразования из float и double в decimal не существует. ПРИМЕЧАНИ Е Преобразование из типов int, ui nt и long в тип float и из типа long в тип double может вызвать потерю точности, но не потерю значимости. В процессе других вари­ антов неявного преобразования никакая информация не теряется. Введение в исключения При вычислении выражений могут возникнуть ошибки, например, переполне­ ние, исчезновение порядка или деление на ноль. В С# есть механизм, который позволяет обрабатывать подобные ошибки и таким образом избегать аварийного завершения программы. Он так и называется: механизм обработки исключитель­ ных ситуаций (исключений). Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью специального действия, называемого выбрасыванием (генерированием) исключения. Каждому типу ошибки соответствует свое исключение. Поскольку С# — язык объектно-ориентированный, исключения являются классами, которые имеют общего предка — класс Excepti on, определенный в пространстве имен System. Например, при делении на ноль будет выброшено (сгенерировано) исключение с длинным, но понятным именем DivideByZeroException, при недостатке памяти — ис­ ключение OutOfMemoryException, при переполнении — исключение OverflowException. ПРИМЕЧАНИ Е Стандартных исключений очень много, тем не менее программист может создавать и собственные исключения на основе класса Exception. Программист может задать способ обработки исключения в специальном блоке ко­ да, начинающемся с ключевого слова catch («перехватить»), который будет автома­ тически выполнен при возникновении соответствующей исключительной ситуации. Внутри блока можно, например, вывести предупреждающее сообщение или скор­ ректировать значения величин и продолжить выполнение программы. Если этот блок не задан, система выполнит действия по умолчанию, которые обычно заключа­ ются в выводе диагностического сообщения и нормальном завершении программы. Процессом выбрасывания исключений, возникающих при переполнении, можно управлять. Для этого служат ключевые слова checked и unchecked. Слово checked включает проверку переполнения, слово unchecked выключает. При выключенной проверке исключения, связанные с переполнением, не генерируются, а результат операции усекается. Проверку переполнения можно реализовать для отдельного выражения или для целого блока операторов, например: а = checked (b + с); // для выражения unchecked { //для блока операторов а = b + с; } Операции и выражения 47 Проверка не распространяется на функции, вызванные в блоке. Если проверка переполнения включена, говорят, что вычисления выполняются в проверяемом контексте, если выключена — в непроверяемом. Проверку переполнения вы­ ключают в случаях, когда усечение результата операции необходимо в соответ­ ствии с алгоритмом. Можно задать проверку переполнения во всей программе с помощью ключа ком­ пилятора /checked, это полезно при отладке программы. Поскольку подобная проверка несколько замедляет работу, в готовой программе этот режим обычно не используется. Мы подробно рассмотрим исключения и их обработку в разделе «Обработка ис­ ключительных ситуаций» (см. с. 89). Основные операции С# В этом разделе кратко описаны синтаксис и применение всех операций С#, кро­ ме некоторых первичных, которые рассматриваются в последующих главах при изучении соответствующего материала. Инкремент и декремент Операции инкремента (++) и декремента (--) , называемые также операциями увеличения и уменьшения на единицу, имеют две формы записи — префиксную, когда знак операции записывается перед операндом, и постфиксную. В префикс­ ной форме сначала изменяется операнд, а затем его значение становится резуль­ тирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется. Листинг 3.2 ил­ люстрирует эти операции. Листин г 3.2. Операции инкремента и декремента using System; namespace ConsoleApplicationl { class Classl { static void MainO { int x = 3. у = 3; Console.Write( "Значение префиксного выражения: " ); Consolе.WriteLineC ++х ); Console.Write( "Значение х после приращения: " ); Console.WriteLineC х ); Console.WriteC "Значение постфиксного выражения: " ); Console.WriteLineC у++ ); Console.WriteC "Значение у после приращения: " ); Console.WriteLineC у ); 48 Глава 3. Переменные, операции и выражения Результат работы программы: Значение префиксного выражения: 4 Значение х после приращения: 4 Значение постфиксного выражения: 3 Значение у после приращения: 4 Стандартные операции инкремента существуют для целых, символьных, вещест­ венных и финансовых величин, а также для перечислений. Операндом может быть переменная, свойство или индексатор (мы рассмотрим свойства и индекса­ торы в свое время, в главах 5 и 7). Операция new Операция new служит для создания нового объекта. Формат операции: new тип ( [ аргументы ] ) С помощью этой операции можно создавать объекты как ссылочных, так и зна­ чимых типов, например: object z = new objectO: int i = new int(): II то же самое, что int i = 0; Объекты ссылочного типа обычно формируют именно этим способом, а пере­ менные значимого типа чаще создаются так, как описано ранее в разделе «Пере­ менные». При выполнении операции new сначала выделяется необходимый объем памяти (для ссылочных типов в хипе, для значимых — в стеке), а затем вызывается так называемый конструктор по умолчанию, то есть метод, с помощью которого ини­ циализируется объект. Переменной значимого типа присваивается значение по умолчанию, которое равно нулю соответствующего типа. Для ссылочных типов стандартный конструктор инициализирует значениями по умолчанию все поля объекта. Если необходимый для хранения объекта объем памяти выделить не удалось, генерируется исключение OutOfMemoryException. Операции отрицания Арифметическое отрицание (унарный минус -) меняет знак операнда на проти­ воположный. Стандартная операция отрицания определена для типов int, long, float, double и decimal. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам (см. рис. 3.1). Тип результа­ та соответствует типу операции. ПРИМЕЧАНИ Е Для значений целого и финансового типов результат достигается вычитанием ис­ ходного значения из нуля. При этом может возникнуть переполнение. Будет ли при этом выброшено исключение, зависит от контекста. Операции и выражения 49 Логическое отрицание (! ) определено для типа bool. Результат операции — значение false, если операнд равен true, и значение true, если операнд равен false. Поразрядное отрицание (~) , часто называемое побитовым, инвертирует каждый разряд в двоичном представлении операнда типа int, uint, long или ulong. Операции отрицания представлены в листинге 3.3. Листин г 3.3. Операции отрицания BSing System; namespace ConsoleApplicationl { class Ciassl { static void MainO sbyte a = 3, b = -63, с = 126; bool d = true; Console.WriteLineC -a ) Console.WriteLineC -c ) Console.WriteLineC !d ) Console.WriteLineC ~a ) Console.WriteLineC ~b ) Console.WriteLineC ~c ) // Результат -3 // Результат -126 // Результат false // Результат -4 // Результат 62 // Результат -127 Явное преобразование типа Операция используется, как и следует из ее названия, для явного преобразо­ вания величины из одного типа в другой. Это требуется в том случае, когда не­ явного преобразования не существует. При преобразовании из более длинного типа в более короткий возможна потеря информации, если исходное значение выходит за пределы диапазона результирующего типа 1 . Формат операции: ( тип ) выражение Здесь тип — это имя того типа, в который осуществляется преобразование, а вы­ ражение чаще всего представляет собой имя переменной, например: long b = 300; int а = (int) b; byte d = (byte) a; // данные не теряются // данные теряются Преобразование типа часто применяется для ссылочных типов при работе с ие­ рархиями объектов. Эта потеря никак не диагностируется, то есть остается на совести программиста. 50 Глава 3. Переменные, операции и выражения Умножение, деление и остаток от деления Операция умножения (\* ) возвращает результат перемножения двух операндов. Стандартная операция умножения определена для типов int, ui nt, long, ulong, float, double и decimal. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам (см. рис. 3.1). Тип результа­ та операции равен «наибольшему» из типов операндов, но не менее int. Если оба операнда целочисленные или типа decimal и результат операции слиш­ ком велик для представления с помощью заданного типа, генерируется исключе­ ние System. OverflowExcepti on 1 . Все возможные значения для вещественных операндов приведены в табл. 3.2. Символами х и у обозначены конечные положительные значения, символом z — результат операции вещественного умножения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению «бесконечность»2 , если слишком мал, он принимается за 0. NaN (not a number) означает, что результат не является числом. Таблица 3.2. Результаты вещественного умножения \* +У -у +0 -0 + 00 —00 NaN +х + Z - z +0 -0 + 00 - o o NaN -х - z + Z -0 +0 - o o + o o NaN +0 +0 -0 +0 -0 NaN NaN NaN -0 -0 +0 -0 +0 NaN NaN NaN + 00 + 00 - 00 NaN NaN + 00 - 00 NaN —00 —00 + о о NaN NaN - 00 + o o NaN NaN NaN NaN NaN NaN NaN NaN NaN Операция деления (I) вычисляет частное от деления первого операнда на второй. Стандартная операция деления определена для типов int, uint, long, ulong, float, double и decimal. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата определяется правилами преобразования (см. рис. 3.1), но не меньше int. Если оба операнда целочисленные, результат операции округляется вниз до ближайшего целого числа. Если делитель равен нулю, генерируется исключение System. Di vi deByZeroExcept i on. Если хотя бы один из операндов вещественный, дробная часть результата деления не отбрасывается, а все возможные значения приведены в табл. 3.3. Символами х и у обозначены конечные положительные значения, символом z — результат 1 В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбра­ сываются избыточные биты. 2 Об «особых» значениях вещественных величин упоминалось на с. 34. Операции и выражения 51 операции вещественного деления. Если результат слишком велик для представ­ ления с помощью заданного типа, он принимается равным значению «бесконеч­ ность», если слишком мал, он принимается за 0. Таблица 3.3. Результаты вещественного деления / +У -у +0 -0 + 0 0 —00 NaN +х + Z - z + о о - 0 0 +0 -0 NaN -х - z + Z - 0 0 + o o -0 +0 NaN +0 +0 -0 NaN NaN +0 -0 NaN -0 -0 +0 NaN NaN -0 +0 NaN + 0 0 + 0 0 - о о + o o —00 NaN NaN NaN - о о —00 + 0 0 —oo + o o NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN Для финансовых величин (тип decimal) при делении на 0 и переполнении генериру­ ются соответствующие исключения, при исчезновении порядка результат равен 0. Операция остатка от деления (%) также интерпретируется по-разному для це­ лых, вещественных и финансовых величин. Если оба операнда целочисленные, результат операции вычисляется по формуле х - (х / у) \* у. Если делитель равен нулю, генерируется исключение System.DivideByZeroException. Тип результата опе­ рации равен «наибольшему» из типов операндов, но не менее int (см. рис. 3.1). Если хотя бы один из операндов вещественный, результат операции вычисляется по формуле х - п \* у, где п — наибольшее целое, меньшее или равное результа­ ту деления х на у. Все возможные комбинации значений операндов приведены в табл. 3.4. Символами х и у обозначены конечные положительные значения, символом z — результат операции остатка от деления. Таблица 3.4. Результаты вещественного остатка от деления о/ /о +У -у +0 -0 + 0 0 —00 NaN + Х + Z z NaN NaN X X NaN - X - z - z NaN NaN -x -x NaN +0 +0 +0 NaN NaN +0 +0 NaN -0 -0 -0 NaN NaN -0 -0 NaN + 0 0 NaN NaN NaN NaN NaN NaN NaN - 0 0 NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN Д ля финансовых величин (тип decimal) при получении остатка от деления на О и при переполнении генерируются соответствующие исключения, при исчезно­ вении порядка результат равен 0. Знак результата равен знаку первого операнда. 52 Глава 3. Переменные, операции и выражени Пример применения операциий умножения, деления и получения остатка прел ставлен в листинге 3.4. Листин г 3.4. Операции умножения, деления и получения остатка using System; namespace Consol eApplicationl { class Classl { static void MainO { int x = 11, у = 4; float z = 4; Console.WriteLineC z \* у ) ; // Результат 16 Console.WriteLineC z \* le308 ); // Результат "бесконечность Console.WriteLineC x / у ) ; // Результат 2 Console. WriteLineC x / z ) ; // Результат 2.75 Console.WriteLineC x I у ) : // Результат 3 Console.WriteLineC le-324 / le-324 ); // Результат NaN } } } Еще раз обращаю ваше внимание на то, что несколько операций одного приори тета выполняются слева направо. Для примера рассмотрим выражение 2 / х • у Деление и умножение имеют один и тот же приоритет, поэтому сначала 2 делит ся на х, а затем результат этих вычислений умножается на у. Иными словамр это выражение эквивалентно формуле 2 -•у- х Если же мы хотим, чтобы выражение х • у было в знаменателе, следует заклю чить его в круглые скобки или сначала поделить числитель на х, а потом на у, т есть записать как 2 / (х • у) или 2 / х / у. , Сложение и вычитание Операция сложения (+ ) возвращает сумму двух операндов. Стандартная one рация сложения определена для типов int, ui nt, long, ulong, float, double и decimal К величинам других типов ее можно применять, если для них существует неяв ное преобразование к этим типам (см. рис. 3.1). Тип результата операции раве] «наибольшему» из типов операндов, но не менее i nt. Если оба операнда целочисленные или типа decimal и результат операции слиш ком велик для представления с помощью заданного типа, генерируется исключе ние System.OverflowException1 . 1 В проверяемом контексте. В непроверяемом исключение не выбрасывается, зато отбра сываются избыточные биты. Операции и выражения 53 Все возможные значения для вещественных операндов приведены в табл. 3.5. Символами х и у обозначены конечные положительные значения, символом z — результат операции вещественного сложения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значе­ нию «бесконечность», если слишком мал, он принимается за 0. Таблица 3.5. Результаты вещественного сложения + У +0 -0 + 0 0 - o o NaN X z X X + 0 0 - 00 NaN +0 У +0 +0 + 00 - 00 NaN -0 У +0 -0 + 00 - o o NaN + 00 + 00 + 00 + 00 + 00 NaN NaN —00 —оо —00 —00 NaN - 00 NaN NaN NaN NaN NaN NaN NaN NaN Операция вычитания (- ) возвращает разность двух операндов. Стандартная опера­ ция вычитания определена для типов int, ui nt, long, ulong, float, double и decimal. К величинам других типов ее можно применять, если для них существует неяв­ ное преобразование к этим типам (см. рис. 3.1). Тип результата операции равен «наибольшему» из типов операндов, но не менее i nt. Если оба операнда целочисленные или типа decimal и результат операции слиш­ ком велик для представления с помощью заданного типа, генерируется исключе­ ние System.OverflowException. Все возможные значения результата вычитания для вещественных операндов приведены в табл. 3.6. Символами х и у обозначены конечные положительные значения, символом z — результат операции вещественного вычитания. Если х и у равны, результат равен положительному нулю. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значе­ нию «бесконечность» с тем же знаком, что и х - у, если слишком мал, он прини­ мается за 0 с тем же знаком, что их - у. Таблица 3.6. Результаты вещественного вычитания - У +0 -0 + 00 —00 NaN X г X X —00 + 0 0 NaN +0 -у +0 +0 - 00 + 00 NaN -0 -у -0 +0 - 0 0 + 00 NaN + 00 + 0 0 + 0 0 + 00 NaN + o o NaN —00 —00 —00 —00 —00 NaN NaN NaN NaN NaN NaN NaN NaN NaN 54 Глава 3. Переменные, операции и выражения Операции сдвига Операции сдвига ( « и » ) применяются к целочисленным операндам. Они сдви­ гают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом 1 . При сдвиге влево (« ) освободившиеся разряды обнуляются. При сдвиге вправо ( » ) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа (то есть выполняется логический сдвиг), и знаковым разрядом — в против­ ном случае (выполняется арифметический сдвиг). Операции сдвига никогда не приводят к переполнению и потере значимости. Стандартные операции сдвига определены для типов int, ui nt, long и ulong. Пример применения операций сдвига представлен в листинге 3.5. Листин г 3.5. Операции сдвига using System; namespace ConsoleApplicationl { class Classl { static void MainO Операции отношения и проверки на равенство Операции отношения (<=, >, >=, ==, != ) сравнивают первый операнд со вторым Операнды должны быть арифметического типа. Результат операции — логической типа, равен true или false. Правила вычисления результатов приведены в табл. 3.7 Таблица 3.7. Результаты операций отношения Операция Результат х == у true, если х равно у, иначе fal se х ! = у true, если х не равно у, иначе fal se х < у true, если х меньше у, иначе fal se х > у true, если х больше у, иначе fal se х <= у true, если х меньше или равно у, иначе fal se х >= у true, если х больше или равно у, иначе fal se 1 Фактически, учитывается только 5 младших битов второго операнда, если первый имее тип int или ui nt, и 6 битов, если первый операнд имеет тип long или ulong. byte а = 3, b = 9; sbyte с = 9, d = -9; Console.WriteLineC a « 1 ) Console.WriteLineC a « 2 ) Console.WriteLineC b » 1 ) Console.WriteLineC с » 1 ) Console.WriteLineC d » 1 ) // Результат 6 // Результат 12 // Результат 4 // Результат 4 // Результат -5 Операции и выражения 55 ПРИМЕЧАНИ Е Обратите внимание на то, что операции сравнения на равенство и неравенство име­ ют меньший приоритет, чем остальные операции сравнения. Очень интересно формируется результат операций отношения для особых случаев вещественных значений. Например, если один из операндов равен NaN, результатом для всех операций, кроме !=, будет false (для операции != результат равен true). Очевиден факт, что для любых операндов результат операции х ! = у всегда равен результату операции ! (х == у), однако если один или оба операнда равны NaN, для операций , <= и >= этот факт не подтверждается. Например, если х или у равны NaN, то х < у даст false, а ! (х >= у) — true. Другие особые случаи рассматриваются следующим образом: • значения +0 и -0 равны; • значение -с о меньше любого конечного значения и равно другому значению -оо ; • значение +о о больше любого конечного значения и равно другому значению +со . Поразрядные логические операции Поразрядные логические операции (&, |, Л) применяются к целочисленным опе­ рандам и работают с их двоичными представлениями. При выполнении опера­ ций операнды сопоставляются побитно (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго и т. д.). Стандартные операции определены для типов int, ui nt, long и ulong. При поразрядной конъюнкции, или поразрядном И (операция обозначается &), бит ре­ зультата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1. При поразрядной дизъюнкции, или поразрядном ИЛИ (операция обозначается |) , бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из опе­ рандов равен 1. При поразрядном исключающем ИЛИ (операция обозначается А) бит результата ра­ вен 1 только тогда, когда соответствующий бит только одного из операндов равен 1. Пример применения поразрядных логических операций представлен в листинге 3.6. Листин г 3.6. Поразрядные логические операции using System: namespace ConsoleApplicationl { class Classl { static void MainO Console.WriteLineC 6 & 5) Console.WriteLineC 6 | 5 ) Console.WriteLineC 6 A 5 ) // Результат 4 // Результат 7 // Результат 3 56 Глава 3. Переменные, операции и выражени! Условные логические операции Условные логические операции И (&&) и ИЛ И (11) чаще всего используются с one рандами логического типа. Результатом логической операции является true шп false. Операции вычисляются по сокращенной схеме. Результат операции логическое И имеет значение true, только если оба операнд; имеют значение true. Результат операции логическое ИЛИ имеет значение true если хотя бы один из операндов имеет значение true. ВНИМАНИ Е Если значения первого операнда достаточно, чтобы определить результат опера ции, второй операнд не вычисляется. Например, если первый операнд операции \ равен false, результатом операции будет false независимо от значения второй операнда, поэтому он не вычисляется. Пример применения условных логических операций представлен в листинге 3.7. Листин г 3.7. Условные логические операции using System; namespace ConsoleApplicationl { class Classl { static void MainO { Console WriteLineC true & & true ) ; // Результат true Console WriteLineC true & & false ) ; // Результат false Console WriteLineC true 1 1 true ); // Результат true Console WriteLineC true J 1 false ) ; // Результат true } } } Условная операция Условная операция (? :) — тернарная, то есть имеет три операнда. Ее формат операнд\_1 ? операнд\_2 : операндЗ Первый операнд — выражение, для которого существует неявное преобразовани к логическому типу. Если результат вычисления первого операнда равен true, TI результатом условной операции будет значение второго операнда, иначе — третье го операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип мо жет различаться. Тип результата операции зависит от типа второго и третьего операндов: • если операнды одного типа, он и становится типом результата операции 1 ; • иначе, если существует неявное преобразование типа от операнда 2 к операн ду 3, но не наоборот, то типом результата операции становится тип операнда ; Это наиболее часто используемый вариант применения тернарной операции. Операции и выражения 57 • иначе, если существует неявное преобразование типа от операнда 3 к операн­ ду 2, но не наоборот, то типом результата операции становится тип операнда 2; • иначе возникает ошибка компиляции. Условную операцию часто используют вместо условного оператора if (он рас­ сматривается в следующей главе) для сокращения текста программы. Пример применения условной операции представлен в листинге 3.8. Листин г 3.8. Условная операция using System; namespace ConsoleApplicationl { class Classl { static void MainO { int a = 11. b = 4; int max = b > a ? b : a; Console.WriteLineC max ); // Результат 11 } И ш Другой пример применения условной операции: требуется, чтобы некоторая це­ лая величина увеличивалась на 1, если ее значение не превышает п, а иначе при­ нимала значение 1. Это удобно реализовать следующим образом: i = (i < n) ? i + 1: 1; Условная операция правоассоциативна, то есть выполняется справа налево. На­ пример, выражение а ? b : с ? d : е вычисляется как aTb^cîd^) 1 . , Операции присваивания Операции присваивания (=, +=, -=, \*- и т. д.) задают новое значение переменной2 . Эти операции могут использоваться в программе как законченные операторы. Формат операции простого присваивания (=): переменная = выражение Механизм выполнения операции присваивания такой: вычисляется выражение и его результат заносится в память по адресу, который определяется именем пере­ менной, находящейся слева от знака операции. То, что ранее хранилось в этой об­ ласти памяти, естественно, теряется. Схематично это полезно представить себе так: Переменная <- Выражение Напомню, что константа и переменная являются частными случаями выражения. 1 Строго говоря, такой «хитрой» записи следует избегать. Программа не должна напоми­ нать шараду. 1 А также свойству, событию или индексатору, которые мы рассмотрим в свое время. 58 Глава 3. Переменные, операции и выра» ^ Примеры операторов присваивания: а = b + с / 2: b = а: а = Ь: х = 1; х = х + 0.5: Обратите внимание: Ь = аи а = Ь — это совершенно разные действия! ПРИМЕЧАНИ Е Чтобы не перепутать, что чему присваивается, запомните мнемоническое прави. присваивание — это передача данных «налево». Начинающие часто делают ошибку, воспринимая присваивание как аналог { венства в математике. Чтобы избежать этой ошибки, надо понимать ме> низм работы оператора присваивания. Рассмотрим для этого последний прим (х = х + 0.5). Сначала из ячейки памяти, в которой хранится значение переме ной х, выбирается это значение. Затем к нему прибавляется 0.5, после чего по./: чившийся результат записывается в ту же самую ячейку, а то, что хранило там ранее, теряется безвозвратно. Операторы такого вида применяются в щ граммировании очень широко. Для правого операнда операции присваивания должно существовать неявное щ образование к типу левого операнда. Например, выражение целого типа мож присвоить вещественной переменной, потому что целые числа являются подмг жеством вещественных, и информация при таком присваивании не теряется: вещественнаяпеременная := целоевыражение; Правила преобразований перечислены в разделе «Преобразования встроенн] арифметических типов-значений» (см. с. 45). Результатом операции присваивания является значение, записанное в лев] операнд. Тип результата совпадает с типом левого операнда. В сложных операциях присваивания ( +=, \*=, /= и т. п.) при вычислении вырая ния, стоящего в правой части, используется значение из левой части. Наприм! при сложении с присваиванием ко второму операнду прибавляется первый, и j зультат записывается в первый операнд, то есть выражение а += b является 6oi компактной записью выражения а = а + Ь. Результатом операции сложного присваивания является значение, записанн в левый операнд. ПРИМЕЧАНИ Е В документации написано, что тип результата совпадает с типом левого операн если он не менее i nt. Это означает, что если, например, переменные а и b имеют т byte, присваивание а += b недопустимо и требуется преобразовать тип явным об] зом: а += (byte)Ь. Однако на практике компилятор ошибку не выдает. Линейные программы 59 Напомню, что операции присваивания правоассоциативны, то есть выполняют­ ся справа налево, в отличие от большинства других операций (а = Ь = с озна­ чает а = (Ь = с)) . Не рассмотренные в этом разделе операции будут описаны позже. Линейные программы Линейной называется программа, все операторы которой выполняются последова­ тельно в том порядке, в котором они записаны. Простейшим примером линейной программы является программа расчета по заданной формуле. Она состоит из трех этапов: ввод исходных данных, вычисление по формуле и вывод результатов. Для того чтобы написать подобную программу, нам пока не хватает знаний о том, как организовать ввод и вывод на языке С#. Подробно этот вопрос рассматрива­ ется в главе 11, а здесь приводятся только минимально необходимые сведения. Простейший ввод-вывод Любая программа при вводе исходных данных и выводе результатов взаимодей­ ствует с внешними устройствами. Совокупность стандартных устройств ввода и вывода, то есть клавиатуры и экрана, называется консолью. Обмен данными с консолью является частным случаем обмена с внешними устройствами, кото­ рый подробно рассмотрен в главе 11. В языке С#, как и во многих других, нет операторов ввода и вывода. Вместо них для обмена с внешними устройствами применяются стандартные объекты. Для работы с консолью в С# применяется класс Console, определенный в пространст­ ве имен System. Методы этого класса Write и WriteLine уже использовались в на­ ших программах. Поговорим о них подробнее, для чего внесем некоторые изме­ нения в листинг 3.1. Результаты этих изменений представлены в листинге 3.9. Листин г 3.9. Методы вывода using System: namespace ConsoleApplicationl { class Classl { static void MainO { int i = 3; double у = 4.12: decimal d = 600m: string s = "Вася"; Console WriteLineC "i - " + i ) ; // 1 Console WriteLineC "y = {0} \nd - {1}". y. d ) ; // 2 Console WriteLineC "s = " + s ) : // 3 } } } 60 Глава 3. Переменные, операции и выражени: Результат работы программы: 1 = 3 У = 4,12 d = 600 s = Вася До сих пор мы использовали метод WriteLine для вывода значений переменных и ли тералов различных встроенных типов. Это возможно благодаря тому, что в клас се Console существует несколько вариантов методов с именами Write и WriteLine предназначенных для вывода значений различных типов. Методы с одинаковыми именами, но разными параметрами называются перегру женными. Компилятор определяет, какой из методов вызван, по типу иередавае мых в него величин. Методы вывода в классе Consol е перегружены для всех встро енных типов данных, кроме того, предусмотрены варианты форматного вывода Листинг 3.9 содержит два наиболее употребительных варианта вызова методо) вывода. Сначала обратите внимание на способ вывода пояснений к значениял переменных в строках 1 и 3. Пояснения представляют собой строковые литера лы. Если метод WriteLine вызван с одним параметром, он может быть любой встроенного типа, например, числом, символом или строкой. Нам же требуете) вывести в каждой строке не одну, а две величины: текстовое пояснение и значе ние переменной, — поэтому прежде чем передавать их для вывода, их требуется «склеить» в одну строку с помощью операции +. Перед объединением строки с числом надо преобразовать число из его внутрен ней формы представления в последовательность символов, то есть в строку. Пре образование в строку определено во всех стандартных классах С# — для этоп служит метод ToStringO. В данном случае он выполняется неявно, но можно вы звать его и явным образом: Console.WriteLineC "i = " + i.ToStringO ); Оператор 2 иллюстрирует форматный вывод. В этом случае используется друпи вариант метода WriteLine, который содержит более одного параметра. Первым па раметром методу передается строковый литерал, содержащий помимо обычны: символов, предназначенных для вывода на консоль, параметры в фигурных скоб ках, а также управляющие последовательности (они были рассмотрены в главе 2) Параметры нумеруются с нуля, перед выводом они заменяются значениями со ответствующих переменных в списке вывода: нулевой параметр заменяется зна чением первой переменной (в данном примере — у), первый параметр — второ! переменной (в данном примере — d) и т. д. ПРИМЕЧАНИ Е Для каждого параметра можно задать ширину поля вывода и формат вывода Мы рассмотрим эти возможности в разделе «Форматирование строк» (см. с. 146] Из управляющих последовательностей чаще всего используются символы пере вода строки (\п) и горизонтальной табуляции (\t). Линейные программы 61 Рассмотрим простейшие способы ввода с клавиатуры. В классе Console определе­ ны методы ввода строки и отдельного символа, но нет методов, которые позволя­ ют непосредственно считывать с клавиатуры числа. Ввод числовых данных вы­ полняется в два этапа: 1. Символы, представляющие собой число, вводятся с клавиатуры в строковую переменную. 2. Выполняется преобразование из строки в переменную соответствующего типа. Преобразование можно выполнить либо с помощью специального класса Convert, определенного в пространстве имен System, либо с помощью метода Parse, имею­ щегося в каждом стандартном арифметическом классе. В листинге 3.10 исполь­ зуются оба способа. Листин г 3.10 . Методы ввода using System; namespace ConsoleApplicationl class Classl { static void MainO Console.WriteLineC "Введите строку string s = Console.ReadLineO: Console.WriteLineC "s = " + s ); ) ; // 1 Console.WriteLineC "Введите символ" ); char с = (char)Console.ReadO; // 2 Console.ReadLineC); // 3 Console.WriteLineC "c = " + с ); string buf; // строка - буфер для ввода чисел Console.WriteLineC "Введите целое число" ); buf = Consolе.ReadLineC); int i = Convert.ToInt32( buf ); 1/4 Console.WriteLineC i ); Console.WriteLineC "Введите вещественное число" ); buf = Console.ReadLineO: double x = Convert.ToDoubleС buf ); // Console.WriteLineC x ); // 5 Console.WriteLineC "Введите вещественное число" ); buf = Console.ReadLineO; double у - double.ParseC buf ); // Console.WriteLineC у ); // б Console.WriteLineC "Введите вещественное число" ); buf - Console.ReadLineO; decimal z = decimal.ParseC buf ); // Console.WriteLineC z ); // 7 } } 62 Глава 3. Переменные, операции и выражения К этому примеру необходимо сделать несколько пояснений. Ввод строки выпол­ няется в операторе 1. Длина строки не ограничена, ввод выполняется до символа перевода строки. Ввод символа выполняется с помощью метода Read, который считывает один символ из входного потока (оператор 2). Метод возвращает значение типа int, представляющее собой код символа, или -1 , если символов во входном потоке нет (например, пользователь нажал клавишу Enter). Поскольку нам требуется не int, a char, а неявного преобразования от int к char не существует, приходится применить операцию явного преобразования типа, которая описана в разделе «Явное преобразование типа» (см. с. 49). За оператором 2 записан оператор 3, который считывает остаток строки и нику­ да его не передает. Это необходимо потому, что ввод данных выполняется через буфер — специальную область оперативной памяти. Фактически, данные сначала заносятся в буфер, а затем считываются оттуда процедурами ввода. Занесение в буфер выполняется по нажатию клавиши Enter вместе с ее кодом. Метод Read, в отличие от ReadLine, не очищает буфер, поэтому следующий после него ввод будет выполняться с того места, на котором закончился предыдущий. В операторах 4 и 5 используются методы класса Convert, в операторах 6 и 7 — методы Parse классов Double и Decimal библиотеки .NET, которые используются здесь через имена типов С# double и decimal. ВНИМАНИ Е При вводе вещественных чисел дробная часть отделяется от целой с помощью запя­ той, а не точки. Иными словами, при вводе используются правила операционной системы, а не языка программирования. Допускается задавать числа с порядком, например, 1,95е-8. Если вводимые с клавиатуры символы нельзя интерпретировать как веществен­ ное число, генерируется исключение 1 . Ввод-вывод в файлы При отладке даже небольших программ может потребоваться их выполнить не раз, не два и даже не десять. При этом ввод исходных данных может стать утомительным и испортить все удовольствие от процесса 2 . Удобно заранее под­ готовить исходные данные в текстовом файле и считывать их в программе. Кро­ ме того, это дает возможность не торопясь продумать, какие исходные данные требуется ввести для полной проверки программы, и заранее рассчитать, что должно получиться в результате. 1 При этом система задает вопрос, хотите ли вы воспользоваться услугами отладчика. От этого предложения следует отказаться, поскольку на данном уровне освоения языка пре­ доставляемые отладчиком сведения будут бесполезны. 2 А удовольствие — необходимое условие написания хорошей программы! Линейные программы 63 Вывод из программы тоже бывает полезно выполнить не на экран, а в текстовый файл для последующего неспешного анализа и распечатки. Работа с файлами подробно рассматривается в главе 11, а здесь приводятся лишь образцы для использования в программах. В листинге 3.11 приведена версия программы из листинга 3.9, выполняющая вывод не на экран, а в текстовый файл с именем output.txt. Файл создается в том же каталоге, что и исполняемый файл програм­ мы, по умолчанию — ...\ConsoleApplication1\bin\Debug. Листин г 3.11 . Вывод в текстовый файл using System; using System.10: namespace ConsoleApplicationl { class Classl { static void MainO { StreamWriter f = new StreamWriterC "output.txt" ) int i = 3; double у = 4.12; decimal d = 600m; string s = "Вася"; f.WriteLineC "i = " + i ); //3 f.WriteLineC "y = {0} \nd = {1}". y, d ); // 4 f.WriteLineC "s = " + s ): li b f.CloseO; // 6 } } } Д ля того чтобы использовать в программе файлы, необходимо: 1. Подключить пространство имен, в котором описываются стандартные классы для работы с файлами (оператор 1). 2. Объявить файловую переменную и связать ее с файлом на диске (оператор 2). 3. Выполнить операции ввода-вывода (операторы 3-5). 4. Закрыть файл (оператор 6). СОВЕ Т При отладке программы бывает удобно выводить одну и ту же информацию и на экран, и в текстовый файл. Для этого соответствующие операторы дублируют. Ввод данных из файла выполняется аналогично. В листинге 3.12 приведена про­ грамма, аналогичная листингу 3.10, но ввод выполняется из файла с именем input.txt, расположенного в каталоге D:\C#. Естественно, из программы убраны все приглашения к вводу. // 1 // 2 64 Глава 3. Переменные, операции и выражения Текстовый файл можно создать с помощью любого текстового редактора, но удоб­ нее использовать Visual Studio.NET. Для этого следует выбрать в меню команду File • New • File... и в появившемся диалоговом окне выбрать тип файла Text File. Листин г 3.12 . Ввод из текстового файла using System; using System.10; namespace ConsoleApplicationl { class Classl { static void MainO { StreamReader f = new StreamReader( "d:\\C#\\input.txt" ); string s = f.ReadLineC); Console.WriteLineC "s = " + s ); char с = (char)f.ReadO; f.ReadLineC); Console.WriteLineC "c = " + с ); string buf; buf = f.ReadLineC); int i = Convert.ToInt32( buf ); Console.WriteLineC i ); buf = f.ReadLineC); double x = Convert.ToDoubleC buf ); Console.WriteLineC x ); buf = f.ReadLineO; double у = double.ParseC buf ); Console.WriteLineC у ); buf = f.ReadLineO; decimal z = decimal.ParseC buf ); Console.WriteLineC z ); f.CloseC); Математические функции — класс Math В выражениях часто используются математические функции, например синус или возведение в степень. Они реализованы в классе Math, определенном в простран­ стве имен System. С помощью методов этого класса можно вычислить: • тригонометрические функции: Sin, Cos, Tan; • обратные тригонометрические функции: ASin, ACos, ATan, ATan2; Линейные программы 65 • гиперболические функции: Tanh, Sinh, Cosh; • экспоненту и логарифмические функции: Exp, Log, LoglO; • модуль (абсолютную величину), квадратный корень, знак: Abs, Sqrt, Sign; • округление: Ceiling, Floor, Round; • минимум, максимум: Mi n, Max; • степень, остаток: Pow, IEEEReminder; • полное произведение двух целых величин: BigMul; • деление и остаток от деления: Di vRem. Кроме того, у класса есть два полезных поля: число к и число е. Описание мето­ дов и полей приведено в табл. 3.8. Таблица 3.8. Основные поля и статические методы класса Math Имя Описание Результат Пояснения Abs Модуль Перегружен1 |x| записывается как Abs(x) Acos Арккосинус2 double Acos(double x) Asin Арксинус double Asin(double x) Atan Арктангенс double Atan(double x) Atan2 Арктангенс double Atan2(double x, double у) — угол, тангенс которого есть результат деления у на х BigMul Произведение long BigMul (int х, int у) Ceiling Округление до большего целого double Ceiling(double х) Cos Косинус double Cos(double х) Cosh Гиперболический косинус double CoshCdouble х) DivRem Деление и остаток Перегружен DivRem(x, у, rem) E. База натурального логарифма (число ё) double 2,71828182845905 Exp Экспонента double ex записывается как Ехр(х) Floor Округление до меньшего целого double Floor(double х) IEEERemainder Остаток от деления double IEEERemainder(double х, double у) Log Натуральный логарифм double \ogex записывается как Log(x) продолжение & Это означает, что существует несколько версий метода для различных типов данных. 2 Угол задается в радианах. 66 Глава 3. Переменные, операции и выражения В листинге 3.13 приведен пример применения двух методов класса Math. Осталь­ ные методы используются аналогично. Листин г 3.13 . Применение методов класса Math usin g System; namespace ConsoleApplication l { clas s Class l { stati c voi d MainO { Console.Write ( "Введите x: " ); strin g buf = Console.ReadLine(); double x = double.Parse ( bu f ); Console.WriteLine ( "Значение sin ; " + Math.Sin(x ) ); Console.Write ( "Введите у; " ); b u f = Console.ReadLine(); double у = double.Pars e bu f ); Console.WriteLine ( "Максимум : " + Math.Max(x , y ) ); } } } В качестве примера рассмотрим программу расчета по заданной формуле у = Vîtv\*-е°'2 " Га + 2tg2a + l,6103 log 1 0 x 2 . 67 з формулы видно, что исходными данными для программы являются две вели- 1ны — х и а. Поскольку их тип и точность представления в условии не оговоре- >i, выберем для них тин double. Программа приведена в листинге 3.14. 1стин г 3.14 . Программа расчета по заданной формуле i ng System: mespace ConsoleApplication l clas s Class l { stati c voi d Main() ( strin g buf : Console.WriteLineC "Введите x" ): b uf = Console.ReadLineO : double x = Convert.ToDouble( bu f ): Console.WriteLineC "Введите aifa " ); b uf = Console.ReadLineO: double a = double.ParseC buf ); double у = Math.Sqrt( Math.PI \* x ) - Math.ExpC 0. 2 \* Math.Sqrt(a ) ) + 2 \* Math.TanC 2 \* a ) + 1.6e3 \* Math.Log]0i' Math.PowCx. 2) ); Console.WriteLineC "Для x « {0} и a!fa - {!)" , x. a }; Console.WriteLineC "Результат = " + у ); } } 1так, к настоящему моменту у вас накопилось достаточно сведений, чтобы писать :а С# простейшие линейные программы, выполняющие вычисления по форму- ам. В следующей главе мы займемся изучением операторов, позволяющих реа- .изовывать более сложные алгоритмы. Рекомендации по программированию 1риступая к написанию программы, четко определите, что является ее исходны­ ми данными и что требуется получить в результате. Выбирайте тип переменных • учетом диапазона и требуемой точности представления данных. Давайте переменным имена, отражающие их назначение. Правильно выбранные шена могут сделать программу в некоторой степени самодокументированной. Неудачные имена, наоборот, служат источником проблем. В именах следует 1збегат ь сокращений. Они делают программу менее понятной, к тому же часто 1егко забыть, как именно было сокращено то или иное слово. 68 Общая тенденция такая: чем больше область действия переменной, тем более длинное у нее имя. Перед таким именем можно поставить префикс типа (однз или несколько букв, по которым можно определить тип переменной). Напротив для переменных, вся «жизнь » которых проходит на протяжении нескольких строк кода, лучше обойтись однобуквенными именами-типа i или к. Имена переменных логического типа, используемые в качестве флагов, должны быть такими, чтобы по ним можно было судить о том, что означают значения tru e и false . Например, признак «пусто» лучше описать не как bool flag , а как bool empty. ПРИМЕЧАНИ Е В С# принято называть классы, методы и константы в соответствии с нотацией Паскаля, а локальные переменные — в соответствии с нотацией Camel (см. раздел «Идентификаторы», с. 24). Переменные желательно инициализировать при их объявлении, а объявлять как можно ближе к месту их непосредственного использования. С другой стороны, удобно все объявления локальных переменных метода располагать в начале бло­ ка так, чтобы их было просто найти. При небольших размерах методов оба эти пожелания довольно легко совместить. Избегайте использования в программе чисел в явном виде. Константы должны иметь осмысленные имена, заданные с помощью ключевого слова const. Символическое имя делает программу более понятной, а кроме того, при необходимости изме­ нить значение константы потребуется изменить программу только в одном мес­ те. Конечно, этот совет не относится к константам 0 и 1. Ввод с клавиатуры предваряйте приглашением, а выводимые значения — поясне­ ниями. Для контроля сразу же после ввода выводите исходные данные на дис­ плей (по крайней мере, в процессе отладки). До запуска программы подготовьте тестовые примеры, содержащие исходные данные и ожидаемые результаты. Отдельно проверьте реакцию программы на неверные исходные данные. При записи выражений обращайте внимание на приоритет операций. Если в одном выражении соседствует несколько операций одинакового приоритета, операции присваивания и условная операция выполняются справа налево, остальные — слева направо. Для изменения порядка выполнения операций используйте круг­ лые скобки. Тщательно форматируйте текст программы так, чтобы его было удобно читать. Ставьте пробелы после знаков препинания, отделяйте пробелами знаки опе­ раций, не пишите много операторов в одной строке, используйте комментарии и пустые строки для разделения логически законченных фрагментов программы. Глава 4 Операторы Операторы языка вместе с его типами данных определяют круг задач, которые можно решать с помощью этого языка. С# реализует типичный набор операто­ ров для языка программирования общего назначения. В этой главе рассматрива­ ются основные операторы С#, составляющие так называемые базовые конструк­ ции структурного программирования. Структурное программирование — это технология создания программ, позво­ ляющая путем соблюдения определенных правил сократить время разработки и уменьшить количество ошибок, а также облегчить возможность модификации программы. Структурный подход, сформировавшийся в 60-70-х годах прошлого столетия, позволил довольно успешно создавать достаточно крупные проекты, но слож­ ность программного обеспечения продолжала возрастать, и требовались все более развитые средства ее преодоления. Идеи структурного программирования по­ лучили свое дальнейшее развитие в объектно-ориентированном программирова­ нии — технологии, позволяющей достичь простоты структуры и управляемости очень больших программных систем. Несмотря на то что язык С# реализует объектно-ориентированную парадигму, принципы структурного программирования лежат в основе кодирования каждо­ го метода, каждого фрагмента алгоритма. Не существует единственного самого лучшего способа создания программ. Для решения задач разного рода и уровня сложности требуется применять разные технологии программирования. В простейших случаях достаточно освоить азы структурного написания программ. Для создания же сложных проектов требует­ ся не только свободно владеть языком в полном объеме, но и иметь представле­ ние о принципах проектирования и отладки программ, возможностях библиоте­ ки и т. д. Как правило, чем сложнее задача, тем больше времени требуется на освоение инструментов, необходимых для ее решения. Глава 4. Операторы Выражения, блоки и пустые операторы Любое выражение, завершающееся точкой с занятой, рассматривается как опера­ тор, выполнение которого заключается в вычислении выражения. Частным слу­ чаем выражения является пустой оператор : (он используется, когда по синтак­ сису оператор требуется, а по смыслу — нет). Примеры: 1-й; //' выполняется операция инкремента а \*« b + с; // выполняется умножение с присваиванием fu!)( i. k ): // выполняется вызов функции while ( tru e ): / / цикл из пустого оператора (бесконечный) Блок, или составной оператор, — это последовательность описаний и операторов, заключенная в фигурные скобки. Блок воспринимается компилятором как один оператор и может использоваться всюду, где синтаксис требует одного оператора, а алгоритм — нескольких. Блок может содержать один оператор или быть пустым. Операторы ветвления Операторы ветвления if и switch применяются для тою чтобы в зависимости от конкретных значений исходных данных обеспечить выполнение разных после­ довательностей операторов. Оператор if обеспечивает передачу управления на одну из двух ветвей вычислений, а оператор switch — на одну из произвольного чиста ветвей. Условный оператор if Условный оператор if используется для разветвления процесса вычислений па два направления. Структурная схема оператора приведена на рис. 4.1. ^ Рис. 4.1 . Структурная схема условного оператора Формат оператора: if ( логическоевыражение ) оператор\_1; [ else оператора; ] 71 "начала вычисляется логическое выражение. Если оно имеет значение trjt , вы- олняется первый оператор, иначе - второй. После этого управление передается а оператор, следующий за условным. Ветвь els e может отсутствовать. 1РИМЕЧАНИ Е Операторы, входящие в условный, не должны иметь метку и не могут быт ь описа­ ниями. 7.ти в какой-либо ветви требуется выполнить несколько операторов, их необхо- л мо заключить в блок, иначе компилятор не сможет понять, где заканчивается етвление. Блок может содержать любые операторы, в том числе описания и другие словные операторы (но не может состоять из одних описаний). Необходимо учи ывать, что переменная, описанная в блоке, вне блока не существует. Трпмеры условных операторов: а < 0 ) b = 1 : ,7 1 а < b && ( а > d 11 а == 0 ) ) b++: els e { b \*= а: а = 0: } // 2 а < b ) if ( а < с ) m = a: els e т = с: if ( b < с ) m = b: els e m = с: /.' 3 - b > a ) max = b: els e max - a: // 4 } примере 1 отсутствует ветвь else . Подобная конструкция реализует пропуск ператора, поскольку присваивание либо выполняется, либо пропускается в за- иснмости от выполнения условия. Если требуется проверить несколько условий, их объединяют знаками логиче- ких условных операций. Например, выражение в примере 2 будет истинно в том лучае, если выполнится одновременно условие а < b и одно из условий в скоб­ ах. Если опустить внутренние скобки, будет выполнено сначала логическое И, . потом — ИЛИ . Оператор в примере 3 вычисляет наибольшее значение из трех неременных. Об­ ратите внимание на то, что компилятор относит часть els e к ближайшему ключе- iONi y слову 1 f. \онструкции, подобные оператору в примере 4 (вычисляется наибольшее значе­ ние из двух переменных), проще и нагляднее записывать в виде условней) опера щи, в данном случае следующей: •;.=b>a?b:a ; ПРИМЕЧАНИ Е Распространенная ошибка начинающих — неверная запись проверки на принадлеж­ ность диапазону. Например, чтобы проверить условие 0 < х < 1, нельзя записать его в условном операторе непосредственно, так как каждая операция отношении долж­ на иметь два операнда. Правильный способ записи: if( 0 < X && х < 1)... 3 качестве примера подсчитаем количество очков после выстрела по мишени, тзображенной на рис. 4.2. 72 Глава 4. Операторы / очко \ \ \ \ Рис. 4.2. Мишень Как уже говорилось, тип переменных выбирается, исходя из их назначения. Ко­ ординаты выстрела нельзя представить целыми величинами, так как это приве­ дет к потере точности результата, а счетчик очков не имеет смысла описывать как вещественный. Программа приведена в листинге 4.1. Листин г 4.1 . Выстрел п о мишени using System; namespace ConsoleApplicationl { class Classl { static void MainO Console.WriteLineC "Введите координату x" ); string buf = Console.ReadLineO; double x = Convert.ToDoubleC buf ); Console.WriteLineC "Введите координату у" ); buf = Console.ReadLineO: double у = double.ParseC buf ); int kol = 0: i f (x\* x + y\*y< 4 ) kol = 1; Console.WriteLineC "Результат = {0} очков", kol ); Даже такую простую программу можно еще упростить с помощью промежуточ­ ной переменной и записи условия в виде двух последовательных, а не вложен­ ных, операторов if: 73 -f ( temp < 1 ) kol = 2; Tonsole.WriteLineC- "Результат = {0} очков" , kol ); Обратите внимание на то, что в первом варианте программы второй оператор i f выполняется только в случае невыполнения условия в первом условном опера­ торе, а во втором варианте оба условных оператора выполняются последователь­ н о , один за другим. Следует избегать проверки вещественных величин на равенство, вместо этого лучше сравнивать модуль их разности с некоторым малым числом. Это связано с погрешностью представления вещественных значений в памяти: -""oat а. Ь; ... •\* С а == b ) ... // не рекомендуется! - f ( Math.AbsCa - b) < ie- 6 ) ... // надежно! Значение величины, с которой сравнивается модуль разности, следует выбирать з зависимости от решаемой задачи и точности участвующих в выражении пере­ менных. Снизу эта величина ограничена определенной в классах Singl e и Double константой Epsilo n (это минимально возможное значение переменной такое, ч т о 1.0 + Epsilo n != 1.0). Оператор выбора switch Оператор switc h (переключатель) предназначен для разветвления процесса вы­ числений на несколько направлений. Структурная схема оператора приведена на рис. 4.3. switch Выражение case 1 case 2 Операторы 1 Операторы 2 case п default > Операторы п Операторы Рис. 4.3. Структурная схема оператора switch Формат оператора: switch С выражение ){ case константное\_выражение\_1: [ список\_операторов\_1 ] case константное\_выражение\_2: [ список\_операторов\_2 ] case константноевыражениеп: [ списокоператоровп ] [ default: операторы ] 74 Глава 4. Операто| Выполнение оператора начинается с вычисления выражения. Тип выражент чаще всего целочисленный (включая char) или строковый1 . Затем управлетп перелается первому оператору из списка, помеченному константным выражен ем, значение которого совпало с вычисленным. Все константные выражения должны быть неявно приводимы к типу выражен] в скобках. Если совпадения не произошло, выполняются операторы, расположе ные после слова defaul t (а при его отсутствии управление передается следующе? за switc h оператору). Каждая ветвь переключателя должна заканчиваться явным оператором перех да, а именно оператором break, got o или return : • оператор break выполняет выход из самого внутреннего из объемлющих е операторов switch , for, whil e и do (см. раздел «Оператор break», с. 84); • оператор got o выполняет переход на указанную после него метку, обычно э метка case одной из нижележащих ветвей оператора switc h (см. раздел «От ратор goto», с. 83); • оператор retur n выполняет выход из функции, в теле которой он записан (с раздел «Оператор return», с. 87). Оператор goto обычно используют для последовательного выполнения нескот ких ветвей переключателя, однако поскольку это нарушает читабельность щ граммы, такого решения следует избегать. В листинге 4.2 приведен пример программы, реализующей простейший калы лятор на четыре действия. Листин г 4.2. Калькулятор на четыре действия using System; namespace ConsoleAppl i cation ! : clas s Class l { stati c voi d Main() i strin g buf ; double ă. b. res , Corisole. Writpi jne ' "Введите первый операнв: " ): buf" = Console ReadLineC): a = double . Parsef bu f i; Console.WriieLineC "Введите знак операции" ); char op - (char)Console.ReadO : Console.ReaciLmec); Console.WriteLineC "Введите второй операнд:" ) . b uf = Console.ReadLineO ; b = double.ParseC buf ); В общем случае выражение может быть любого типа, для которого существует неяв преобразование к указанным, а также перечисляемого тина. Операторы цикла 75 bool ok = true : switc h (op) I i case '+ 1 : res = a + b; break : case '- 1 : res = a - b: break ; case '\* ' : res - a \* b: break : case '/ ' : res = a / b: break : defaul t : res = double.NaN: ok = false : break : i if (ok ) Console.WriteLineC "Результат. " + res ): els e Console WriteLineC "Недопустимая операция" СОВЕТ Хотя наличие ветви defaul t и не обязательно, рекомендуется всегда обрабатывать случай, когда значение выражения не совпадает ни с одной из констант. Это облег чает поиск ошибок при отладке программы. Оператор switch предпочтительнее оператора if в тех случаях, когда в программе требуется разветвить вычисления на количество направлений большее двух и вы­ ражение, по значению которого производится переход на ту или иную ветвь, не является вещественным. Часто это справедливо даже для двух ветвей, поскольку новы шает наглядность программы. Операторы цикла Операторы цикла используются для вычислений, повторяющихся многократно. В С# имеется четыре вида циклов: цикл с предусловием whi 1е, цикл с постусло­ вием repeat, цикл с параметром for и цикл перебора foreach. Каждый из них со­ стоит из определенной последовательности операторов. Блок, ради выполнения которого и организуется цикл, называется телам цикла. Остальные операторы служат для управления процессом повторения вычис­ лений: это начальные установки, проверка условия продолжения цикла и модн• фикация параметра цикла (рис, 4.4). Один проход цикла называется итерацией. Начальные установки служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются. Прочерка условия продолжения цикла выполняется на каждой итерации либо до ^ела цикла (тогда говорят о цикле с предусловием, схема которого показана на : пс. 4.4, а), либо после тела цикла (цикл с постусловием, рис. 4.4, 6). Разница между ними состоит в том. что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Про­ верка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу. 76 Глава 4. Операторы Начальные установки Модификация параметра цикла Рис. 4.4. Структурные схемы операторов цикла Параметром цикла называется переменная, которая используется при проверке условия продолжения цикла и принудительно изменяется на каждой итерации, причем, как правило, на одну и ту же величину. Если параметр цикла целочис­ ленный, он называется счетчиком цикла. Количество повторений такого цикла можно определить заранее. Параметр есть не у всякого цикла. В так называемом итеративном цикле условие продолжения содержит переменные, значения ко­ торых изменяются в цикле по рекуррентным формулам1 . Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы break, continue, return и goto (см. раздел «Операторы передачи управления», с. 83). Передавать управление извне внутрь цикла запре­ щается (при этом возникает ошибка компиляции). Цикл с предусловием while Формат оператора прост: while ( выражение ) оператор Выражение должно быть логического типа. Например, это может быть операция от­ ношения или просто логическая переменная. Если результат вычисления выраже­ ния равен true, выполняется простой или составной оператор (блок). Эти действия повторяются до того момента, пока результатом выражения не станет значение fa 1 se. После окончания цикла управление передается на следующий за ним оператор. Рекуррентной называется формула, в которой новое значение переменной вычисляется с использованием ее предыдущего значения. Операторы цикла 77 Выражение вычисляется перед каждой итерацией цикла. Если при первой про­ верке выражение равно false, цикл не выполнится ни разу. ВНИМАНИ Е Если в теле цикла необходимо выполнить более одного оператора, необходимо за­ ключить их в блок с помощью фигурных скобок. В качестве примера рассмотрим программу, выводящую для аргумента х, изме­ няющегося в заданных пределах с заданным шагом, таблицу значений следую­ щей функции: 't, х <= Xk ) li b { У = t; 1/2 продолжение 78 Глава 4. Оператор/ Листин г 4,3 (продолжение) i f ( X >= О И X < 10 ) у = t \* х; i f ( х >= 10 ) у = 2 \* t ; Console.WriteLineC "| {0.6 } j {1. 6 х += dX: {1.6 } I", x. у ): // 2 11 2 //' 3 // 4 ПРИМЕЧАНИ Е При выводе использованы дополнительные поля спецификаций формата, опред( ляющие желаемую ширину поля вывода под переменную (в данном случае — 6 не зиций). В результате колонки при выводе таблицы получаются ровными. Описанг спецификаций формата приведено в приложении. Параметром этого цикла, то есть переменной, управляющей его выполнением является х. Блок модификации параметра цикла представлен оператором, вь полняющимся на шаге 4. Для перехода к следующему значению аргумента tck j щее значение наращивается на величину шага и заносится в ту же иеременнув Начинающие часто забывают про модификацию параметра, в результате програи ма «зацикливается». Если с вами произошла такая неприятность, попробуйт для завершения программы нажать клавиши Ctrl+Break, а впредь перед запуске программы проверяйте: • присвоено ли параметру цикла верное начальное значение; • изменяется ли параметр цикла на каждой итерации цикла; • верно ли записано условие продолжения цикла. Распространенным приемом программирования является организация бесконе1 ного цикла с заголовком whil e (true ) и принудительным выходом из тела цик; по выполнению какого-либо условия с помощью операторов передачи управл' ния. В листинге 4.4 приведен пример использования бесконечного цикла для О] ганизации меню программы. Листин г 4.4. Организация меню usin g System; namespace ConsolеАрр1ication l { clas s Class ! { stati c voi d MainO strin g buf : whil e ( tru e ) { Console.WriteLineC "1 - пункт\_1 . 2 - пункт\_\_2. 3 - выход" ): b u f = Console.ReadLineO: switc h С buf ) case "1 " : // Вставить код обработки пункта 1 Console.WriteLineC "отладка - пункт\_1 " ): Операторы цикла 79 break : case "2 Console.WriteLineC break ; // Вставить код обработки nyiKi a 2 ne( "oi;idaxa - пункг\_2 " ), case "3" return ; defaul t : Console.WriteLineC "не попал по клавише!" break ; Цикл с постусловием do Цикл с постусловием реализует структурную схему, приведенную на рис. 4.4, б, и имеет вид do оператор while выражение; Сначала выполняется простой или составной оператор, образующий тело цикла, а затем вычисляется выражение (оно должно иметь тип bool) . Если выражение истинно, тело цикла выполняется еще раз и проверка повторяется. Цикл завер­ шается, когда выражение станет равным fals e или в теле цикла будет выполнен какой-либо оператор передачи управления. Этот вид цикла применяется в тех случаях, когда тело цикла необходимо обя­ зательно выполнить хотя бы один раз, например, если в цикле вводятся данные и выполняется их проверка. Если же такой необходимости нет, предпочтитель­ нее пользоваться циклом с предусловием. Пример программы, выполняющей проверку ввода, приведен в листинге 4.5. Листин г 4.5. Проверка ввода .sin g System: -amespace ConsolеАррIication i clas s Class l { stati c voi d MainO char answer: do Console.WriteLineC "Купи слоника, a?" ): answer = (char) Console.Read О; Console.ReadLineO; whli e ( answer !=• у' ); 80 Глава 4. Операторы Рассмотрим еще один пример применения цикла с постусловием — программу, определяющую корень уравнения cos(x) = х методом деления пополам с точно­ стью 0,0001. Исходные данные для этой задачи — точность, результат — число, представляю­ щее собой корень уравнения 1 . Оба значения имеют вещественный тип. Суть метода деления пополам очень проста. Задается интервал, в котором есть ровно один корень (следовательно, на концах этого интервала функция имеет значения разных знаков). Вычисляется значение функции в середине этого ин­ тервала. Если оно того же знака, что и значение на левом конце интервала, зна­ чит, корень находится в правой половине интервала, иначе — в левой. Процесс повторяется для найденной половины интервала до тех пор, пока его длина не станет меньше заданной точности. В приведенной далее программе (листинг 4.6) исходный интервал задан с помо­ щью констант, значения которых взяты из графика функции. Дл я уравнений, имеющих несколько корней, можно написать дополнительную программу, опре­ деляющую (путем вычисления и анализа таблицы значений функции) интерва­ лы, содержащие ровно один корень 2 . Листин г 4.6 . Вычисление корня нелинейного уравнения using System; namespace ConsoleApplicationi { class Classl { static void MainO x = ( left + right ) / 2; if ( ( Math.Cos(x) - x ) \* ( Math.Cos(left) - left ) < 0 ) right = x; else left = x; } while ( Math.Abs( right - left ) < le-4 ); Console.WriteLineC "Корень равен " + x ); double x, left = 0, right = 1; do В эту программу для надежности очень полезно добавить подсчет количества выполненных итераций и принудительный выход из цикла при превышении их разумного количества. 81 Цикл с параметром for Цикл с параметром имеет следующий формат: for ( инициализация; выражение; модификации ) оператор; Инициализация служит для объявления величин, используемых в цикле, и при­ своения им начальных значений. В этой части можно записать несколько опера­ торов, разделенных запятой, например: - э г ( in t i - 0. j « 20: .. . •nt k. m; ""эг ( k = 1 , m = 0: .. . Областью действия переменных, объявленных в части инициализации цикла, яв­ ляется цикл. Инициализация выполняется один раз в начале исполнения цикла. Выражение типа bool определяет условие выполнения цикла: если его результат равен true , цикл выполняется. Цикл с параметром реализован как цикл с преду­ словием. Модификации выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую, например: -~:г ( in t i = 0, j = 20; i < 5 && j > 10; i++ , j- - ) ... Простой или составной оператор представляет собой тело цикла. Любая из час­ тей оператора fo r может быть опущена (но точки с запятой надо оставить на сво­ их местах!). Для примера вычислим сумму чисел от 1 до 100: • " t 5 - 0: - V ( in t i - 1 , i < - 100; i++ ) s += i : В листинге 4.7 приведена программа, выводящая таблицу значений функции из листинга 4.3. Листин г 4.7. Таблица значений функции, полученных с использованием цикла for .51ng System: -amespace ConsoleApplication l clas s Class l { stati c voi d MainO double Xn = -2 , Xk = 12. dX = 2. t = 2, y; Console.WriteLineC "| x | у |"; // заголовок таблицы f o r ( double x = Xn; x <= Xk: x += dX ) // 1 , 4 , 5 У - t; i f ( x >= 0 && x < 10 ) у = t \* x ; // 2 // 2 продолжение & 82 Глава 4. Оператс Листин г 4.7 (продолжение) Любой цикл whil e может быть приведен к эквивалентному ему циклу fo r и оборот. Например, два следующих цикла эквивалентны: Цикл for : f o r ( bl ; Ь2: ЬЗ ) оператор; Цикл whi 1е: Ы : whil e ( Ь2 ) { оператор; ЬЗ } Цикл перебора foreach Цикл foreac h используется для просмотра всех объектов из некоторой труп данных, например массива, списка или другого контейнера. Он будет рассм рен, когда у нас появится в нем необходимость, а именно в разделе «Онера1 foreach» (см. с. 136). Рекомендации по выбору оператора цикла Операторы цикла взаимозаменяемы, но можно привести некоторые рекомен ции по выбору наилучшего в каждом конкретном случае. Оператор do whil e обычно используют, когда цикл требуется обязательно вып нить хотя бы раз, например, если в цикле производится ввод данных. Оператором whil e удобнее пользоваться в тех случаях, когда либо число ите ций заранее неизвестно, либо очевидных параметров цикла нет, либо модифи цию параметров удобнее записывать не в конце тела цикла. i f ( х >= 10 ) у = 2 \* t ; // 2 Console.WriteLineC "| {0,6 } | {1.6 } |". х, у ): // 3 } } } } Как видите, в этом варианте программы все управление циклом сосредоточь в его заголовке. Это делает программу понятней. Кроме того, областью дейстг служебной переменной х является цикл, а не вся функция, как это было в л тинге 4.3, что предпочтительнее, поскольку переменная х вне цикла не требует СОВЕ Т В общем случае надо стремиться к минимизации области действия перемени Это облегчает поиск ошибок в программе. 83 Оператор foreac h применяют для просмотра элементов различных коллекций объектов. Оператор fo r предпочтительнее в большинстве остальных случаев. Однозначно — для организации циклов со счетчиками, то есть с целочисленными переменны­ ми, которые изменяют свое значение при каждом проходе цикла регулярным об­ разом (например, увеличиваются на 1). Начинающие часто делают ошибки при записи циклов. Чтобы избежать этих ошибок, рекомендуется: • проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого правильные начальные значе­ ния (а также возможно ли выполнение других операторов); • проверить, изменяется ли в цикле хотя бы одна переменная, входящая в усло­ вие выхода из цикла; • предусмотреть аварийный выход из цикла по достижении некоторого коли­ чества итераций (пример приведен в следующем разделе); • и, конечно, не забывать о том, что если в теле цикла требуется выполнить бо­ лее одного оператора, нужно заключать их в фигурные скобки. Операторы передачи управления В С# есть пять операторов, изменяющих естественный порядок выполнения вы­ числений: • оператор безусловного перехода goto ; • оператор выхода из цикла break; • оператор перехода к следующей итерации цикла continue ; • оператор возврата из функции return ; • оператор генерации исключения throw. Эти операторы могут передать управление в пределах блока, в котором они ис­ пользованы, и за его пределы. Передавать управление внутрь другого блока за­ прещается. Первые четыре оператора рассматриваются в этом разделе, а оператор throw — далее в этой главе на с. 93. Оператор goto Оператор безусловного перехода goto используется в одной из трех форм: goto метка; goto case константное\_выражение; goto default; В теле той же функции должна присутствовать ровно одна конструкция вида метка: оператор; 84 Глава 4. Операторь Оператор goto метка передает управление на помеченный оператор. Метка — этс обычный идентификатор, областью видимости которого является функция, в теле которой он задан. Метка должна находиться в той же области видимости, чтс и оператор перехода. Использование этой формы оператора безусловного пере­ хода оправдано в двух случаях: • принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей; • переход из нескольких точек функции вниз по тексту в одну точку (например, если перед выходом из функции необходимо всегда выполнять какие-либо действия). В остальных случаях для записи любого алгоритма существуют более подходя­ щие средства, а использование оператора goto приводит только к усложнению структуры программы и затруднению отладки. Применение этого оператора на­ рушает принципы структурного и модульного программирования, по которым все блоки, образующие программу, должны иметь только один вход и один выход. Вторая и третья формы оператора goto используются в теле оператора выбора switch. Оператор goto case константное\_выражение передает управление на соответ­ ствующую константному выражению ветвь, а оператор goto default — на ветвь default. Надо отметить, что реализация оператора выбора в С# на редкость не­ удачна, и наличие в нем оператора безусловного перехода затрудняет понимание программы, поэтому лучше обходиться без него. Оператор break Оператор break используется внутри операторов цикла или выбора для перехода в точку программы, находящуюся непосредственно за оператором, внутри кото­ рого находится оператор break. Д ля примера рассмотрим программу вычисления значения функции ch х (гипер­ болический косинус) с точностью в = 10~6 с помощью бесконечного ряда Тейлора по формуле х 2 х 4 х 6 х 2 " у = 1 + + + +.. . + +.. . 2! 4! 6! 2я! Этот ряд сходится при | х\ < со. Для достижения заданной точности требуется суммировать члены ряда, абсолютная величина которых больше е. Для сходяще­ гося ряда модуль члена ряда Сп при увеличении п стремится к нулю. При неко­ тором п неравенство | Сп \ > е перестает выполняться и вычисления прекращаются. Алгоритм решения задачи выглядит так: задать начальное значение суммы ряда, а затем многократно вычислять очередной член ряда и добавлять его к ранее найденной сумме. Вычисления заканчиваются, когда абсолютная величина оче­ редного члена ряда станет меньше заданной точности. До выполнения программы предсказать, сколько членов ряда потребуется просум­ мировать, невозможно. В цикле такого рода есть опасность, что он никогда не за­ вершится — как из-за возможных ошибок в вычислениях, так и из-за ограниченной 85 области сходимости ряда (данный ряд сходится на всей числовой оси, но суще­ ствуют ряды Тейлора, которые сходятся только для определенного интервала значений аргумента). Поэтому для надежности программы необходимо преду­ смотреть аварийный выход из цикла с печатью предупреждающего сообщения по достижении некоторого максимально допустимого количества итераций. Для выхода из цикла применяется оператор break. Прямое вычисление члена ряда по приведенной общей формуле, когда х возво­ дится в степень, вычисляется факториал, а затем числитель делится на знамена­ тель, имеет два недостатка, которые делают этот способ непригодным. Первый недостаток — большая погрешность вычислений. При возведении в степень и вы­ числении факториала можно получить очень большие числа, при делении кото­ рых друг на друга произойдет потеря точности, поскольку количество значащих цифр, хранимых в ячейке памяти, ограничено 1 . Второй недостаток связан с эф­ фективностью вычислений: как легко заметить, при вычислении очередного чле­ на ряда нам уже известен предыдущий, поэтому вычислять каждый член ряда «от печки» нерационально. Д ля уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через преды­ дущий: Cn+i = C,i ' ^ где Т — некоторый множитель. Подставив в эту формулу Сп и С„+1, получим вы­ ражение для вычисления Т: т С, • \_ 2п\ -х2™ = х 2 ~Сп "х2п - (2(п + 1))! (2и + 1)(2я + 2)' В листинге 4.8 приведен текст программы с комментариями. Листин г 4.8 . Вычисление суммы бесконечного ряда jsin g System: •'aniespace Conso 1 eAppl i cat i on l { clas s Class ! { stati c voi d MainO { double e = le- 6; cons t in t Maxlte r = 500; / / ограничитель количества итераций Console.WriteLineC "Введите аргумент: " ); strin g buf = Console.ReadLineO; double x = Convert.ToDoubleC buf ); bool done = true ; // признак достижения точности double ch = 1, у = ch : f o r ( in t n = 0: Math.Abs(ch) > e; n++ ) { продолжение Кроме того, большие числа могут переполнить разрядную сетку. Глава 4. Операторы Листин г 4.8 (продолжение) ch\*=x\*x/(2\* n + l)/(2\* n + 2) ; у += ch ; // добавление члена ряда к сумге if ( n > Maxlte r ) { done = false ; break : } } if ( done ) Console.WriteLineC "Сумма ряда - " + у ); els e Console.WriteLineC "Ряд расходится" ); } } } Получение суммы бесконечного ряда — пример вычислений, которые принципи­ ально невозможно выполнить точно, В данном случае мы задавали желаемую погрешность вычислений с помощью значения е. Это значение не может быт ь меньше, чем самое малое число, представимое с помощью переменной типа double, но при задании такого значения точность результата фактически будет горазд о ниже из-за погрешностей, возникающих при вычислениях. Они связаны с коне ч - ностью разрядной сетки. В общем случае погрешность результата складывается из нескольких части"!: • погрешность постановки задачи (возникает при упрощении задачи); • начальная погрешность (точность представления исходных данных) ; • погрешность метода (при использовании приближенных методо в решени я задачи); • погрешности округления и вычисления (поскольку величины хранятся в ог ­ раниченном количестве разрядов). Специфика машинных вычислений состоит в том, что алгоритм, безупречный с точки зрения математики, при реализации без учета возможных погрешностей може т привести к получению результатов, не содержащих н и одно й верно й зна ­ чащей цифры! Это происходит, например, при вычитании двух близких значе­ ний или при работе с очень большими или очень малыми числами. Оператор continue Оператор перехода к следующей итерации текущего цикла continu e пропускает все операторы, оставшиеся до конца тела цикла, и передает управление н а нача ­ ло следующей итерации. Перепишем основной цикл листинга 4.8 с применением оператор а continue : f o r С in t n = 0; Math.Abs(ch) > e; n++ ) { ch \*= x \* x / ( 2 \* n + 1 ) / ( 2 \* n + 2 ); у += ch ; if ( n <= Maxlte r ) continue ; done = false : break ; } 87 Оператор return Оператор возврата из функции return завершает выполнение функции и переда­ ет управление в точку ее вызова. Синтаксис оператора: return [ выражение ]; Тип выражения должен иметь неявное преобразование к типу функции. Если тип возвращаемого функцией значения описан как void, выражение должно от­ сутствовать. Базовые конструкции структурного программирования Главное требование, которому должна удовлетворять программа, — работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена точно к заявленному сроку и допускать оперативное внесение необходимых изменений н дополнений. 11ными словами, современные критерии качества программы — это, прежде все­ г о, надежность, а также возможность точно планировать производство програм­ мы и ее сопровождение. Для достижения этих целей программа должна иметь простую структуру, быть читабельной и легко модифицируемой. Технология структурного программирования позволяет создавать как раз такие программы небольшого и среднего объема. Для разработки более сложных комплексов тре­ буется применять объектно-ориентированное программирование. В С# идеи структурного программирования используются на самом низком уровне — при написании методов объектов. Доказано, что любой алгоритм мож­ но реализовать только из трех структур, называемых базовыми конструкциями структурного программирования, — это следование, ветвление и цикл. Следованием называется конструкция, реализующая последовательное выполне­ ние двух или более операторов (простых или составных). Ветвление задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. Цикл реализует многократное выполнение оператора. Базо­ вые конструкции приведены на рис. 4.5. Следование Цикл Ветвление Рис. 4.5. Базовые конструкции структурного программирования 88 Глава 4. Операторы Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следование из двух ветвлений, каждое из которых включает вложенные циклы (рис. 4.6). Цикл! 1 J ;\_\_ Цикл! i J j j j j Ветвление ! ! j цикл \_ \_ ^—nrm— '\_ ! Рис. 4.6. Вложение базовых конструкций Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать (а программы чаще прихо­ дится читать, чем писать), отлаживать и при необходимости вносить в нее изме­ нения. В большинстве языков высокого уровня существует несколько реализаций ба­ зовых конструкций; в С# есть четыре вида циклов и два вида ветвлений (на два и на произвольное количество направлений). Они введены для удобства програм­ мирования, и в каждом случае надо выбирать наиболее подходящие средства. Главное, о чем нужно помнить даже при написании самых простых программ, — они должны состоять из четкой последовательности блоков строго определен­ ной структуры. «Кто ясно мыслит, тот ясно излагает» — практика давно показа­ ла, что программы в стиле «поток сознания» нежизнеспособны, не говоря о том, что они просто некрасивы. Обработка исключительных ситуаций 89 Обработка исключительных ситуаций В языке С# есть операторы, позволяющие обнаруживать и обрабатывать ошибки (исключительные ситуации), возникающие в процессе выполнения программы. Об этом уже упоминалось в разделе «Введение в исключения» (см. с. 46), а сей­ час мы рассмотрим механизм обработки исключений более подробно. Исключительная ситуация, или исключение, — это возникновение аварийного со­ бытия, которое может порождаться некорректным использованием аппаратуры пли неправильной работой программы, например делением на ноль или пере­ полнением. Обычно эти события приводят к завершению программы с систем­ ным сообщением об ошибке. С# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение. Исключения С# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например нажатие клавиш Ctrl+C. Меха­ низм исключений предназначен только для событий, которые могут произойти в результате работы самой программы и указываются явным образом. Исключе­ ния возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное. Исключения позволяют логически разделить вычислительный процесс на две час­ ти — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главное то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использую­ щий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функ­ ций и программ, состоящих из многих модулей. Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое зна­ чение или параметры, поэтому заголовки функций не разрастаются. ПРИМЕЧАНИ Е В принципе, ничто не мешает рассматривать в качестве исключений не только ошибки, но и нормальные ситуации, возникающие при обработке данных, но это не имеет преимуществ перед другими решениями, не улучшает структуру программы и не делает ее понятнее. Исключения генерирует либо среда выполнения, либо программист с помощью оператора throw . В табл. 4.1 приведены наиболее часто используемые стандарт­ ные исключения, генерируемые средой. Они определены в пространстве имен System. Все они являются потомками класса Exception , а точнее, потомками его потомка SystemException. Исключения обнаруживаются и обрабатываются в операторе try . 90 Глава 4. Операторы Таблица 4.1 . Часто используемые стандартные исключения И м я Описани е Ari thmeti cExcepti on Ошибка в арифметических операциях или преобразованиях (является предком DivideBeZeroException и OverFlowException) ArrayTypeMi smatchExcepti on Попытка сохранения в массиве элемента несовместимого тина Di vi deByZeroExcepti on Попытка деления на ноль FormatException Попытка передать в метод аргумент неверного формата IndexOutOfRangeException Индекс массива выходит за границы диапазона InvalidCastException Ошибка преобразования типа OutOfMemoryExcepti on Недостаточно памяти для создания нового объекта OverFlowException Переполнение при выполнении арифметических операций StackOverR owExcepti on Переполнение стека Оператор try Оператор try содержит три части: • контролируемый блок — составной оператор, предваряемый ключевым сло­ вом try. В контролируемый блок включаются потенциально опасные операто­ ры программы. Все функции, прямо или косвенно вызываемые из блока, так­ же считаются ему принадлежащими; • один или несколько обработчиков исключений — блоков catch, в которых опи­ сывается, как обрабатываются ошибки различных типов; • блок завершения finally выполняется независимо от того, возникла ошибка в контролируемом блоке или нет. Синтаксис оператора try: try блок [ блоки catch ] [ блок finally ] Отсутствовать могут либо блоки catch, либо блок finally, но не оба одновре­ менно. Рассмотрим, каким образом реализуется обработка исключительных ситуаций. 1. Обработка исключения начинается с появления ошибки. Функция или опе­ рация, в которой возникла ошибка, генерирует исключение. Как правило, ис­ ключение генерируется не непосредственно в блоке try, а в функциях, прямо или косвенно в него вложенных. 2. Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление. 3. Выполняется блок final 1у, если он присутствует (этот блок выполняется и в том случае, если ошибка не возникла). 91 4. Если обработчик не найден, вызывается стандартный обработчик исключе­ ния. Его действия зависят от конфигурации среды. Обычно он выводит на эк­ ран окно с информацией об исключении и завершает текущий процесс. ПРИМЕЧАНИ Е Подобные окна не предназначены для пользователей программы, поэтому все ис­ ключения, которые могут возникнуть в программе, должны быть перехвачены и об­ работаны. Обработчики исключений должны располагаться непосредственно за блоком try . Они начинаются с ключевого слова catch , за которым в скобках следует тип об­ рабатываемого исключения. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Блоки catc h просматри­ ваются в том порядке, в котором они записаны, пока не будет найден соответст­ вующий типу выброшенного исключения. Синтаксис обработчиков напоминает определение функции с одним парамет­ ром — типом исключения. Существуют три формы записи: catch( тип имя ) { ... /\* тело обработчика \*/ } catch( тип ) {.../\* тело обработчика \*/ } catch { ... /\* тело обработчика \*/ } Первая форма применяется, когда имя параметра используется в теле обработ­ чика для выполнения каких-либо действий, например вывода информации об исключении. Вторая форма не предполагает использования информации об исключении, иг­ рает роль только его тип. Третья форма применяется для перехвата всех исключений. Так как обработ­ чики просматриваются в том порядке, в котором они записаны, обработчик третьего типа (он может быть только один) следует помещать после всех осталь­ ных. Пример: -'•у { . . . // Контролируемый блок catch ( OvertlowExceptio n е ) { // Обработка исключений класса OverflowExceptio n (переполнение) catch ( DivideByZeroExceptîon ) { . . . .// Обработка исключений класса DivideByZeroException (деление на 0) catch { /7 Обработка всех остальных исключений Если исключение в контролируемом блоке не возникло, все обработчики про­ пускаются. 92 В любом случае, произошло исключение или нет, управление передается в блок завершения finally (если он существует), а затем — первому оператору, находяще­ муся непосредственно за оператором try. В завершающем блоке обычно записы­ ваются операторы, которые необходимо выполнить независимо от того, возник­ ло исключение или нет, например, закрытие файлов, с которыми выполнялась работа в контролируемом блоке, или вывод информации. В листинге 4.9 приведена программа, вычисляющая силу тока по заданным напря­ жению и сопротивлению. Поскольку вероятность неверного набора веществен­ ного числа довольно высока, оператор ввода включен в контролируемый блок. ПРИМЕЧАНИ Е Исключение, связанное с делением на ноль, для вещественных значений возник­ нуть не может, поэтому не проверяется. При делении на ноль будет выдан резуль­ тат «бесконечность». Листин г 4.9 . Использование исключений для проверки ввода using System; namespace ConsoleApplicationi { class Program { static void MainO { string buf; double u, i, r; try { Console.WriteLineC "Введите напряжение:" ); buf = Console.ReadLineO; u = double.ParseC buf ); Console.WriteLineC "Введите сопротивление:" ): buf = Console.ReadLineO; r- = double. ParseC buf ); i = u / r; Console.WriteLineC "Сила тока - " + i ); } catch ( FormatException ) { Console.WriteLineC "Неверный формат ввода!" ); } catch // общий случай { Console.WriteLineC "Неопознанное исключение" ); } } } } Обработка исключительных ситуаций 93 Операторы try могут многократно вкладываться друг в друга. Исключение, ко­ торое возникло во внутреннем блоке try и не было перехвачено соответствую­ щим блоком catch, передается на верхний уровень, где продолжается поиск под­ ходящего обработчика. Этот процесс называется распространением исключения. Распространение исключений предоставляет программисту интересные возмож­ ности. Например, если на внутреннем уровне недостаточно информации для того, чтобы провести полную обработку ошибки, можно выполнить частичную обра­ ботку и сгенерировать исключение повторно, чтобы оно было обработано на верх­ нем уровне. Генерация исключения выполняется с помощью оператора throw. Оператор throw До сих пор мы рассматривали исключения, которые генерирует среда выпол­ нения С#, но это может сделать и сам программист. Дл я генерации исключе­ ния используется оператор throw с параметром, определяющим вид исключе­ ния. Параметр должен быть объектом, порожденным от стандартного класса System.Exception. Этот объект используется для передачи информации об исклю­ чении его обработчику. Оператор throw употребляется либо с параметром, либо без него: throw [ выражение ]; Форма без параметра применяется только внутри блока catch для повторной ге­ нерации исключения. Тип выражения, стоящего после throw, определяет тип ис­ ключения, например: :hrow new DivideByZeroExceptionO; Здесь после слова throw записано выражение, создающее объект стандартного класса «ошибка при делении на 0» с помощью операции new. При генерации исключения выполнение текущего блока прекращается и проис­ ходит поиск соответствующего обработчика с передачей ему управления. Обра­ ботчик считается найденным, если тип объекта, указанного после throw, либо тот же, что задан в параметре catch, либо является производным от него. Рассмотрим пример, приведенный в спецификации С#: jsing System; class Test static void F() { try { GO; // функция, в которой может произойти исключение } catch ( Exception е ) { Console.WriteLineC "Exception in F: " + e.Message ); e = new Exception( "F" ); throw; // повторная генерация исключения } Глава 4. Операторы } static void GO { throw new Exception( "G" ); // моделирование исключительной ситуации } static void MainO { try { FO; } catch ( Exception e ) { Console.WriteLineC "Exception in Main: " + e.Message ); В методе F выполняется промежуточная обработка исключения, которая заклю­ чается в том, что на консоль выводится поле Message перехваченного объекта е (об элементах класса Exception рассказывается в следующем разделе). После этого исключение генерируется заново. Несмотря на то что в обработчике исключения создается новый объект класса Exception с измененной строкой информации, передаваемой в исключении, выбрасывается не этот объект, а тот, который был перехвачен обработчиком, поэтому результат работы программы следующий: Exception in F: G Exception in Main: G Заменим оператор throw таким оператором: throw e; В этом случае в обработчике будет выброшено исключение, созданное в преды­ дущем операторе, и вывод программы изменится: Exception in F: G Exception in Main: F С обработкой исключений мы еще не раз встретимся в примерах программ, при­ веденных в следующих главах. Класс Exception Класс Exception содержит несколько полезных свойств, с помощью которых мож­ но получить информацию об исключении. Они перечислены в табл. 4.2. Таблица 4.2 . Свойства класса System.Exception Свойство Описание HelpLink U R L файла справки с описанием ошибки Message Текстовое описание ошибки. Устанавливается при создании объекта. Свойство доступно только для чтения Source И мя объекта или приложения, которое сгенерировало ошибку 95 Свойство Описание STаскTrac e Последовательность вызовов, которые привели к возникновению ошибки. Свойство доступно только для чтения " не r Except io n Содержит ссылку на исключение, послужившее причиной генерации текущего исключения ~r.rget.Site Метод, выбросивший исключение Операторы checked и unchecked К а к уж е упоминалось в главе 3, процессом генерации исключений, возникающих п р и переполнении , можн о управлять с помощью ключевых слов checked и unchecked, которы е употребляются как операции, если они используются в выражениях, и как операторы , если они предваряют блок, например: з - checked (b + с) ; // для выражения (проверка включена) ,^checked { // для блока операторов (проверка выключена) а - b + с: П поверка не распространяется на функции, вызванные в блоке. Рекомендации по программированию К настоящему моменту вы изучили основные операторы С# и можете писать на этом язык е простые программы, используя класс, состоящий из одного метода. Даже на это м уровн е важно придерживаться определенных правил, следование которым поможе т ва м избежать многих распространенных ошибок. Конечно, на все случаи жизн и советы дать невозможно, ведь не зря многие считают программирование ис­ кусством. С приобретением опыта вы добавите к приведенным далее правилам мно­ жеств о своих, выстраданных в результате долгих бдений над «последней» ошибкой. ВНИМАНИЕ — Глазная цель, к которой нужно стремиться, — получить понятную программу как можно более простой структуры. В конечном счете, все технологии программирова­ ния направлены на достижение именно этой цели, поскольку только так можно до­ биться надежности программы и легкости ее модификации. Создани е программ ы надо начинать с определения ее исходных данных и резуль­ татов. Пр и этом задумываются не только о смысле каждой величины, но и о том, какое множество значений она может принимать. В соответствии с этим выбира­ ются тип ы переменных. Следующий шаг — записать на естественном языке (возможно, с применением обобщенных блок-схем), что именно и как должна делать программа. Если вы не 96 Глава 4. Операторы можете сформулировать алгоритм по-русски, велика вероятность того, что он плохо продуман (естественно, я не имею в виду, что надо «проговаривать» все на уровне от­ дельных операторов, например, «изменяя индекс от 0 до 100 с шагом 1..»). Описание алгоритма полезно по нескольким причинам: оно помогает в деталях продумать алго­ ритм, найти на самой ранней стадии некоторые ошибки, разбить программу на ло­ гическую последовательность блоков, а также обеспечить комментарии к программе. При кодировании 1 программы необходимо помнить о принципах структурного программирования: программа должна состоять из четкой последовательности блоков — базовых конструкций, каждая из которых имеет один вход и один вы­ ход. Конструкции могут вкладываться, но не пересекаться. Программа должна быть «прозрачна». Если какое-либо действие можно запрограм­ мировать разными способами, то предпочтение должно отдаваться не наиболее компактному и даже не наиболее эффективному, а более понятному. Особенно это важно тогда, когда пишут программу одни, а сопровождают другие, что является широко распространенной практикой. «Непрозрачное» программирование может повлечь за собой огромные издержки, связанные с поиском ошибок при отладке. Для записи каждого фрагмента алгоритма необходимо использовать наиболее подходящие средства языка. Например, ветвление на насколько направлений по значению целой или строковой переменной эффектнее записать с помощью од­ ного оператора switch, а не нескольких операторов if . Для просмотра массива лучше пользоваться циклом for или foreach. Оператор goto применяют весьма редко, например, в операторе выбора switch или для принудительного выхода из нескольких вложенных циклов, а в большинстве других ситуаций лучше исполь­ зовать другие средства языка, такие как break или return. Для организации циклов пользуйтесь наиболее подходящим оператором. Цикл do применяется только в тех случаях, когда тело в любом случае потребуется вы­ полнить хотя бы один раз, например, при проверке ввода. При использовании циклов надо стремиться объединять инициализацию, проверку условия выхода и приращение в одном месте. Рекомендации по выбору наиболее подходящего оператора цикла были приведены на с. 82. При записи итеративных циклов (в которых для проверки условия выхода ис­ пользуются соотношения переменных, формирующихся в теле цикла), необхо­ димо предусматривать аварийный выход по достижении заранее заданного мак­ симального количества итераций. Это повышает надежность программы. Более короткую ветвь if лучше поместить сначала, иначе вся структура может не поместиться на экране, что затруднит отладку. Бессмысленно использовать проверку на равенство true или false: 1 Кодированием называют процесс написания текста программы, чтобы отличить его о более общего понятия «программирование», включающего также этапы проектирование и отладки программы. bool busy: if ( busy == true ) { ... } if ( busy == false ) { ... } // плохо! Лучше if ( busy ) // плохо! Лучше if ( !busy ) 97 Следует избегать лишних проверок условий. Например: if ( а < b ) с - 1 ; else if ( а > b ) с = 2; else if ( а == b ) с = 3: Вместо этих операторов можно написать: if ( а < b ) с = 1 : else if ( а > b ) с = 2: else с = 3; Пли даже так: с = 3: if ( а < b ) с = 1 ; if ( а > b ) с - 2; Если первая ветвь оператора i f обеспечивает передачу управления, использовать •зетвъ else нет необходимости: if ( i > 0 ) break: // здесь i <= О 3 некоторых случаях условная операция лучше условного оператора: if ( z == 0 ) i = j; else i = k; // лучше так: i = z == 0 ? j : k; Необходимо предусматривать печать сообщений или генерацию исключения в тех точках программы, куда управление при нормальной работе программы передавать­ ся не должно. Именно это сообщение вы с большой вероятностью получите при первом же запуске программы. Например, полезно, если оператор switch имеет зетвь default, реализующую обработку ситуации по умолчанию, если в ней пере­ числены все возможные значения переключателя. 3 программе полезно предусматривать реакцию на неверные входные параметры. -Это может быть генерация исключения (предпочтительно), печать сообщения или формирование признака результата с его последующим анализом. Сообщение об гмибке должно быть информативным и подсказывать пользователю, как ее ис­ править. Например, при вводе неверного значения в сообщении должен быть указан допустимый диапазон. Генерируйте исключения в тех случаях, когда в месте возникновения ошибки не­ достаточно данных, чтобы ее обработать. Заключайте потенциально опасные фрагменты программы в проверяемый блок -.-: и обрабатывайте хотя бы исключение типа Exception, а лучше — все исключе­ ния, которые могут в нем возникнуть, по отдельности. Конечно, все усилия по комментированию программы пропадут втуне, если сама она написана путано, неряшливо и непродуманно. Поэтому после написания про­ грамму следует тщательно отредактировать — убрать ненужные фрагменты, : группировать описания, оптимизировать проверки условий и циклы, проверить, : лтнмально ли разбиение на методы, и т. д. С первого раза без помарок хороший .гкст не напишешь, будь то сочинение, статья или программа. Однако не следует 98 Глава 4. Операторь пытаться оптимизировать все, что «попадается под руку», поскольку главных принцип программиста тот же, что и врача: «Не навреди!». Если программа ра ботает недостаточно эффективно, надо в первую очередь подумать о том, каки< алгоритмы в нее заложены. Подходить к написанию программы нужно таким образом, чтобы ее можно бьш в любой момент передать другому программисту. Полезно дать почитать свок программу кому-нибудь из друзей или коллег (а еще лучше - врагов или завист ников) и в тех местах, которые они не смогут понять без устных комментариев внести их прямо в текст программы. Комментарии имеют очень важное значение, поскольку программист, как ни стран но, чаще читатель, чем писатель. Даже если сопровождающим программистов является автор программы, разбираться через год в плохо документированное тексте сомнительное удовольствие. Дальнейшие советы касаются комментари ев и форматирования текста программы, что является неотъемлемой частью про цесса программирования. Программа, если она используется, живет не один год, потребность в каких-п ее новых свойствах проявляется сразу же после ввода в эксплуатацию, и сопровож дение программы занимает гораздо больше времени, чем ее написание. Основна часть документации должна находиться в тексте программы. Хорошие коммента рии написать почти так же сложно, как и хорошую программу. В С# есть мощны средства документирования программ, речь о которых пойдет в главе 15, но сам содержание комментариев никто, кроме вас, задать не сможет. Комментарии должны представлять собой правильные предложения- без сокраще ний и со знаками препинания 1 и не должны подтверждать очевидное (коммента рии в этой книге не могут служить образцом, поскольку они предназначены дл обучения, а не сопровождения). Например, бессмысленны фразы типа «вызов ме года f» или «описание переменных». Если комментарий к фрагменту программ! занимает несколько строк, разместить его лучше перед фрагментом, чем справ от него, и выровнять по вертикали. Вложенные блоки должны иметь отступ в 3-5 символов, причем блоки од ног уровня вложенности должны быть выровнены по вертикали. Форматируйте теш по столбцам везде, где это возможно, это делает программу гораздо понятнее: strin g but = "qwerty"; double ex = 3.1234; i n t number = 12; byt e z =0 ; if ( done ) Console.WriteLine ! "Сумма ряда - " + у ); els e Console.WriteLineC "Ряд расходится" ): i f ( x >= 0 && x < 10 ) у = t \* x ; els e if ( x >= 10 ) у = 2 \* t: els e у = x; 1 Совсем хорошо, если они при этом не будут содержать орфографических ошибок. Рекомендации по программированию 99 В последних трех строках показано, что иногда большей ясности можно добить­ ся, если не следовать правилу отступов буквально. Абзацный отступ комментария должен соответствовать отступу комменти­ руемого блока: II Комментарий, описывающий, // что происходит в следующем ниже // блоке программы, { /\* Непонятный блок программы \*/ } Для разделения методов и других логически законченных фрагментов пользуй­ тесь пустыми строками или комментарием вида / / - Не следует размещать в одной строке много операторов. Как и в русском языке, после знаков препинания должны использоваться пробелы: f=a+b; // плохо! Лучше f - а + Ь; Помечайте конец длинного составного оператора, например: whil e ( tru e ) { whil e ( х < у ) { f o r ( i = 0; i < 10; ++i ) { f o r ( j = 0; j < 10; ++j ) { } / // две страницы кода /end fo r ( j = 0; j < 10; ++j ) } // end fo r ( i = 0; i < 10; ++i ) } II end whil e ( x < у ) } // end whil e ( tru e ) Конструкции языка C# в основном способствуют хорошему стилю программи­ рования, тем не менее, и на этом языке легче легкого написать запутанную, не­ надежную, некрасивую программу, которую проще переписать заново, чем вне­ сти в нее требуемые изменения. Только постоянные тренировки, самоконтроль и стремление к совершенствованию помогут вам освоить хороший стиль, без ко­ торого невозможно стать квалифицированным программистом 1 . В заключение порекомендую тем, кто предпочитает учиться программированию не только на своих ошибках, очень полезные книги [2], [6]. Вообще говоря, этот не блещущий новизной совет можно отнести практически к любому роду человеческой деятельности. Глава 5 Классы: основные понятия Понятие о классах вы получили 1 в разделах «Классы» (см. с. 13) и «Заготовка консольной программы» (см. с. 17). Все программы, приведенные в этой книге ранее, состояли из одного класса с одним-единственным методом Main. Сейчас настало время подробнее изучить состав, правила создания и использования классов. По сути, отныне все, что мы будем рассматривать, так или иначе связа­ но с этим ключевым средством языка. Класс является типом данных, определяемым пользователем. Он должен пред­ ставлять собой одну логическую сущность, например, являться моделью реаль­ ного объекта или процесса. Элементами класса являются данные и функции, предназначенные для их обработки. Описание класса содержит ключевое слово class , за которым следует его имя, а да­ лее в фигурных скобках - тело класса, то есть список его элементов. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных ат­ рибутов и спецификаторов, определяющих различные характеристики класса: [ атрибуты ] [ спецификаторы ] class имякласса [ : предки ] тело-класса Как видите, обязательными являются только ключевое слово class , а также имя и тело класса. Имя кчасса задается программистом по общим правилам С#. Тело класса — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может выглядеть так: clas s Demo {} ПРИМЕЧАНИ Е Необязательные атрибуты задают дополнительную информацию о классе. Посколь­ ку наша задача пока состоит в том, чтобы освоить основные понятия, мы отложим знакомство с атрибутами до главы 12. 1 А может быть, и не получили. Классы: основные понятия 101 Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечисле­ ны в табл. 5.1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложен­ ным. В зависимости от места описания класса некоторые из этих спецификато­ ров могут быть запрещены. Таблица 5.1 . Спецификаторы класса Спецификатор Описание 1 new Используется для вложенных классов. Задает новое описание класса взамен унаследованного от предка. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 175) •") public Доступ не ограничен 3 protected Используется для вложенных классов. Доступ только из элементов данного и производных классов L -i internal Доступ только из данной программы (сборки)' 5 protected internal Доступ только из данного и производных классов или из данной программы (сборки) 6 private Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс t abstract Абстрактный класс. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 181) s sealed Бесплодный класс. Применяется в иерархиях объектов, рассматривается в главе 8 (см. с. 182) 9 static Статический класс. Введен в версию языка 2.0. Рассматривается в разделе «Конструкторы» (см. с. 114) Спецификаторы 2-6 называются спецификаторами доступа. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут присутствовать в описании только в вариантах, приведенных в таб­ лице, а также могут комбинироваться с остальными спецификаторами. В этой главе мы будем изучать классы, которые описываются в пространстве имен непосредственно (то есть не вложенные классы). Для таких классов до­ пускаются только два спецификатора: public и internal. По умолчанию, то есть если ни один спецификатор доступа не указан, подразумевается специфика­ тор internal. Класс является обобщенным понятием, определяющим характеристики и пове­ дение некоторого множества конкретных объектов этого класса, называемых эк­ земплярами, или объектами, класса. Понятие сборки было введено в главе 1 на с. 9. 110 Глава 5. Классы: основные понятия Листин г 5.3 (продолжение) short tl = 3. t2 = 4: i n t у = Max( tl , tZ ) : Console.WriteLineC у ): i n t z = Max( a + tl . t l / 2 Console.WriteLineC z ); b ): // вызов метода Max // результат : 4 // вызов метода Max // результат : 5 2 В классе описан метод Мах, который выбирает наибольшее из двух переданных ему значений. Параметры описаны как а и Ь. В методе Main выполняются три вы­ зова Мах. В результате первого вызова методу Мах передаются два аргумента того же типа, что и параметры, во втором вызове — аргументы совместимого типа, в третьем - выражения. ВНИМАНИ Е — Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должн ы записыватьс я в том же порядке , что и в заголовке метола, и должно существоват ь неявное преобразование типа каждого аргумента к типу со­ ответствующего параметра. Количество аргументов должно соответствовать количеству параметров. На име­ на никаких ограничений не накладывается: имена аргументов могут как совпа­ дать, так и не совпадать с именами параметров. Существуют два способа передачи параметров: по значению и но ссылке. При передаче по значению метод получает копии значений аргументов, и опера­ торы метода работают с этими копиями. Доступа к исходным значениям аргу­ ментов у метода нет. а следовательно, нет и возможности их изменить. При передаче по ссылке (по адресу) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять ис­ ходные значения аргументов, модифицируя параметры. В С# для обмена данными между вызывающей и вызываемой функциями преду­ смотрено четыре тина параметров: • параметры-значения; • параметры-ссылки — описываются с помощью ключевого слова ref; • выходные параметры — описываются с помощью ключевого слова out: • параметры-массивы — описываются с помощью ключевого слова params. Ключевое слово предшествует описанию типа параметра. Если оно опущено, па­ раметр считается параметром-значением. Параметр-массив может быть только один и должен располагаться последним в списке, например: publi c in t Calculate ( in t a. re f in t b. out in t c, params intf ] d ) ... 200 Глава 9. Интерфейсы и структурные типы Листин г 9.1 (продолжение) { Monster temp = (Monster) obj; if ( this.health > temp.health ) return 1; if ( this.health < temp.health ) return -1; return 0; } string name; int health, ammo; } class CI ass 1 { static void MainO { const int n = 3; Monster[] stado = new Monster[n]; stado[0] = new Monster( 50, 50, "Вася" ); stado[l] = new Monster( 80, 80, "Петя" ); stado[2] = new Monster( 40, 10, "Маша" )'; Array.Sort( stado ); // сортировка стала возможной foreach ( Monster el em in stado ) elem.Passport(); } } } Результат работы программы: Monster Маша health = 40 ammo = 10 Monster Вася health = 50 ammo = 50 Monster Петя health = 80 ammo = 80 Если несколько объектов имеют одинаковое значение критерия сортировки, их относительный порядок следования после сортировки не изменится. Во многих алгоритмах требуется выполнять сортировку объектов по различным критериям. В С# для этого используется интерфейс I Comparer, который рассмот­ рен в следующем разделе. Сортировка по разным критериям (интерфейс IComparer) Интерфейс IComparer определен в пространстве имен System. Col lections. Он со­ держит один метод CompareTo, возвращающий результат сравнения двух объек­ тов, переданных ему в качестве параметров: interface IComparer { int Compare ( object obi, object ob2 ) Стандартные интерфейсы .NET 201 Принцип применения этого интерфейса состоит в там, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализую­ щий этот интерфейс. Объект этого класса передается в стандартный метод сор­ тировки массива в качестве второго аргумента (существует несколько перегру­ женных версий этого метода). Пример сортировки массива объектов из предыдущего листинга по именам (свойство Name, класс SortByName) и количеству вооружений (свойство Ammo, класс EortByAmmo) приведен в листинге 9.2. Классы параметров сортировки объявлены вложенными, поскольку они требуются только объектам класса Monster. Листин г 9.2. Сортировка по двум критериям .sin g System; .sin g System.Col lections ; 'amespace ConsoleApplication i clas s Monster { publi c Monster( in t health , in t ammo, strin g name ) { this.healt h = health ; this.ammo = ammo; this.nam e = name; } publi c in t Ammo { g e t { retur n ammo; } s e t { if (valu e > 0) ammo = value ; els e ammo = 0; } } publi c strin g Name { g e t { retur n name; } } virtua l publi c voi d Passport O { Console.WriteLineC "Monster {0} \ t healt h = {1} ammo = {2}" , name, health , ammo ); } publi c clas s SortByName : IComparer // { i n t IComparer.Compare( objec t obi , objec t ob2 ) продолжение ^ 202 Глава 9. Интерфейсы и структурные типы Листин г 9.2 (продолжение) Monster ml = (Monster) obi : Monster m2 = (Monster) ob2; retur n String.Compare( m1.Name, m2.Name ); int. IComparer.Compare( objec t obi , objec t ob2 ) { Monster ml - (Monster) obi : Monster ni2 - (Monster) ob2; if ( ml.Ammo > m2.Ammo ) retur n 1; if ( ml.Ammo < m2.Ammo ) retur n -1 ; retur n 0. cons t in t n=3 . MonsterГ3 stado « new Monster] , stado[0 ] = new Monster( 50. 50. "Вася" ); stado[l ] = new Monster( 80. 80. "Петя" ); stado[2 ] = new Monster( 40. 10. "Маша" ): Console.WriteLine ( "Сортировка по имени: " ); Array.Sort ( stado , new Monster SortByNameО j ; foreac h ( Monster el em in stado ) el em. Passport','). Console.WriteLine ( "Сортировке по вооружению " %:: Array.Sort ( stado . new Monster.SortByAmmoO ); foreac h ( Monster elem in stado ) elem.Passport(); publi c clas s SortByAmmo : IComparer strin g name: m t health , ammo; Class Class1 { stati c void Mam() Результат работы программы: Сортировка по имени: Monster Вася healt h » 50 ammo Monster Маша healt h = 40 ammo Monster Петя healt h = 80 ammo 50 10 30 Стандартные интерфейсы .NET 203 Сортировка по вооружению: Monster Маша healt h Monster Вася healt h Monster Петя healt h healt h = 40 ammo healt h = 50 ammo healt h = 80 ammo 10 50 80 Перегрузка операций отношения Если класс реализует интерфейс IComparable, его экземпляры можно сравнивать между собой по принципу больше или меньше. Логично разрешить использовать для этого операции отношения, перегрузив их. Операции должны перегружаться парами: < и >, <= и >=, == и ! =. Перегрузка операций обычно выполняется путем де­ легирования, то есть обращения к переопределенным методам Сотра г е т о и тана Is. ПРИМЕЧАНИ Е Если класс реализует интерфейс IComparable, требуется переопределить метод fcqual s и связанный с ним метод GetHashCode. Оба метода унаследованы от базового класса object . Пример перегрузки был приведен в разделе «Клас с object\* (см. с. 183). В листинге 9.3 операции отношения перегружены для класса Monster. В качестве критерия сравнения объектов по принципу больше или меньше выступает поле health , а при сравнении на равенство реализуется значимая семантика, то есть попарно сравниваются все поля объектов Листин г 9 . 3 . Перегрузка операций отношения using System; namespace ConsoleApplication i clas s Monster : IComparable { publi c Monster( in t health , in t ammo, strin g name ) { this.healt h = health ; this.ammo = ammo; this.nam e = name; publi c overrid e bool Equals( objec t ob j ) { if ( ob j == nul l II GetTypeO != obj.GetType() ) retur n false ; Monster temp = (Monster) obj ; retur n healt h == temp.healt h && ammo == temp.ammo && name == temp.name; publi c overrid e in t GetHashCode() продолжение iP 204 Глава 9. Интерфейсы и структурные типы Листин г 9.3 (продолжение) { return name.GetHashCode(); } public static bool operator == ( Monster a. Monster b ) { return a.Equals( b ); } // вариант: // public static bool operator == ( Monster a, Monster b ) // { // return ( a.CompareTo( b ) == 0 j; / / } public static bool operator != ( Monster a, Monster b ) { return ! a.Equals( b ); } // вариант: // public static bool operator != ( Monster a, Monster b ) // { // return ( a.CompareTo( b ) !- 0 ); / / } public static bool operator < ( Monster a, Monster b ) return ( a.CompareTo( b ) < 0 ); public static bool operator > ( Monster a. Monster b ) return ( a.CompareTo( b ) > 0 ); public static bool operator <= ( Monster a. Monster b ) return ( a.CompareTo( b ) <= 0 ); public static bool operator >= ( Monster a, Monster b ) return ( a.CompareTo( b ) >= 0 ); public int CompareTo( object obj ) Стандартные интерфейсы .NET 205 Monster temp = (Monster) obj ; if ( this.healt h > temp.healt h ) retur n 1; if ( this.healt h < temp.healt h ) retur n -1 : retur n 0; } strin g name; i n t health , ammo; } clas s Class l { stati c voi d Main() { Monster Вася = new Monster( 70, 80, "Вася" ); Monster Петя = new Monster( 80, 80, "Петя" ); if ( Вася > Петя ) Console.WriteLine ( "Вася больше Пети" ); els e if ( Вася = Петя ) Console.WriteLine ( "Вася — Петя" ); els e Console.WriteLine ( "Вася меньше Пети" ); } } Результат работы программы не разочаровывает: Зася меньше Пети Клонирование объектов (интерфейс ICIoneable) Клонирование — это создание копии объекта. Копия объекта называется кло­ ном. Как вам известно, при присваивании одного объекта ссылочного типа дру­ гому копируется ссылка, а не сам объект (рис. 9.1, а). Если необходимо скопи­ ровать в другую область памяти поля объекта, можно воспользоваться методом w emberwiseClone, который любой объект наследует от класса object . При этом объ­ екты, на которые указывают поля объекта, в свою очередь являющиеся ссылка­ ми, не копируются (рис. 9.1, б). Это называется поверхностным клонированием. а б Рис. 9.1 . Клонирование объектов 206 Глава 9. Интерфейсы и структурные типа Д ля создания полностью независимых объектов необходимо глубокое клонирова­ ние, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта, поля полей и т. д. (рис. 9.1, в). Алгоритм глубокого клонирования весьма сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей. Объект, имеющий собственные алгоритмы клонирования, должен объявляться как наследник интерфейса ICloneable и переопределять его единственный метод С1ог т В листинге 9.4 приведен пример создания поверхностной копии объекта класса Monster с помощью метода MemberwiseClone, а также реализован интерфейс ICloneab'r В демонстрационных целях в имя клона объекта добавлено слово «Клон» . Обратите внимание на то, что метод MemberwiseClone можно вызвать только из методов класса. Он не может быть вызван непосредственно, поскольку объявлен в классе objec t как защищенный (protected) . Листин г 9.4. Клонирование объектов using System: namespace ConsoleApplIcation i { clas s Monster : ICloneabl e { publi c Monster( in t health , in t ammo, strin g name ) { this.healt h = health : this.ammo = ammo: this.nam e = name; } publi c Monster ShallowClone() / / поверхностная копия { retur n (Monster)this,MemberwiseClone(); } publi c objec t Clone O / / пользовательская копия { retur n new Monster( this.health , this.ammo. "Клон " + this.nam e ); } virtua l publi c voi d Passport O { Console.WriteLineC "Monster {0} \t healt h = {1} ammo = {2}" , name, health , ammo ); } strin g name; i n t health , ammo; } clas s Class l { stati c voi d MainO Стандартные интерфейсы .NET { Monster Вася = new Monster( 70. 80. "Вася" ); Monster X = Вася; Monster Y = Вася.ShallowClone(); Monster Z = (Моп51ег)Вася.CIone(); } } } Объект X ссылается на ту же область памяти, что и объект Вася. Следовательно, если мы внесем изменения в один из этих объектов, это отразится на другом. Объекты Y и Z, созданные путем клонирования, обладают собственными копия­ ми значений полей и независимы от исходного объекта. Перебор объектов (интерфейс I Enumerable) и итераторы Оператор foreach является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы IEnumerable и IEnumerator. Для применения оператора foreach к пользовательскому типу дан­ ных требуется реализовать в нем эти интерфейсы. Давайте посмотрим, как это делается. Интерфейс IEnumerable (перечислимый) определяет всего один метод — GetEnuraerator, возвращающий объект типа IEnumerator (перечислитель), который можно исполь зовать для просмотра элементов объекта. Интерфейс IEnumerator задает три элемента: • свойство Current, возвращающее текущий элемент объекта; • метод MoveNext, продвигающий перечислитель на следующий элемент объекта; • метод Reset, устанавливающий перечислитель в начало просмотра. Цикл foreach использует эти методы для перебора элементов, из которых состо­ ит объект. Таким образом, если требуется, чтобы для перебора элементов класса мог приме­ няться цикл foreach, необходимо реализовать четыре метода: GetEnumerator, Current, MoveNext и Reset. Например, если внутренние элементы класса организованы в мас­ сив, потребуется описать закрытое поле класса, хранящее текущий индекс в мас­ сиве, в методе MoveNext задать изменение этого индекса на 1 с проверкой выхода за границу массива, в методе Current — возврат элемента массива по текущему индексу и т. д. Это не интересная работа, а выполнять ее приходится часто, поэтому в версию 2.0 были введены средства, облетающие выполнение перебора в объекте — итераторы. Итератор представляет собой блок кода, задающий последовательность перебо­ ра элементов объекта. На каждом проходе цикла foreach выполняется один шаг 207 208 Глава 9. Интерфейсы и структурные типы итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова yield. Рассмотрим создание итератора на примере (листинг 9.5). Пусть требуется соз­ дать объект, содержащий боевую группу экземпляров типа Monster, неоднократно использованного в примерах этой книги. Для простоты ограничим максималь­ ное количество бойцов в группе десятью. Листин г 9.5. Класс с итератором using System; using System.Col lections; namespace ConsoleApplicationi { class Monster { .. . } class Daemon { .. . } class Stado ; IEnumerable // 1 { private Monster[] mas; private int n; public StadoO { mas = new Monster[10]; n = 0; } public IEnumerator GetEnumeratoK) { for ( int i = 0; i < n; ++i ) yield return mas[i]; // 2 } public void Add( Monster m ) { if ( n >= 10 ) return; mas[n] = m: ++n; class Classl { static void MainO { Stado s = new StadoO; s.Add( new Monster() ); s.Add( new Monster("Вася") ): s.Add( new Daemon О ); foreach ( Monster m in s ) m.Passport(); Стандартные интерфейсы .NET 209 Все, что требуется сделать в версии 2.0 для поддержки перебора, — указать, что класс реализует интерфейс IEnumerable (оператор 1), и описать итератор (опера­ тор 2). Доступ к нему может быть осуществлен через методы MoveNext и Current интерфейса IEnumerator. За кодом, приведенным в листинге 9.5, стоит большая внутренняя работа компи­ лятора. На каждом шаге цикла foreach для итератора создается «оболочка» — служебный объект, который запоминает текущее состояние итератора и выпол­ няет все необходимое для доступа к просматриваемым элементам объекта. Ины­ ми словами, код, составляющий итератор, не выполняется так, как он выглядит — в виде непрерывной последовательности, а разбит на отдельные итерации, между которыми состояние итератора сохраняется. В листинге 9.6 приведен пример итератора, перебирающего четыре заданных строки. Листин г 9.6. Простейший итератор using System; using System.Collections; namespace ConsoleApplicationi { class Num : IEnumerable { public IEnumerator GetEnumerator() { yield return "one"; yield return "two"; yield return "three"; yield return "oops"; } } class Classl { static void MainO { foreach ( string s in new NumO ) Console.WriteLineC s ); } Результат работы программы: one two three oops Следующий пример демонстрирует перебор значений в заданном диапазоне (от 1 до 5): using System; using System.Col lections; Методы 111 О параметрах-массивах мы будем говорить позже, в главе 7 (см. с, 154), а сейчас рассмотрим остальные типы параметров. Параметры-значения Параметр-значение описывается в заголовке метода следующим образом: тип имя Пример заголовка метода, имеющего один параметр-значение целого типа: .oi d Р( in t х ) Имя параметра может быть произвольным. Параметр х представляет собой .ло­ кальную переменную, которая получает свое значение из вызывающей функции при вызове (метода. В метод передается копия значения аргумента. Механизм передачи следующий: из ячейки памяти, в которой хранится неремен­ ная, передаваемая в метод, берется ее значение и копируется в специальную об­ ласть памяти — область параметров. Метод работает с этой копией, следователь­ но, доступа к ячейке, где хранится сама переменная, не имеет. По завершении работы метода область параметров освобождается. Таким образом, для парамет­ ров-значений используется, как вы догадались, передача но значению. Ясно, что этот способ годится только для величин, которые не должны измениться после выполнения метода, то есть для его исходных данных. При вызове метода на месте параметра, передаваемого по значению, может нахо­ диться выражение, а также, конечно, его частные случаи - переменная или кон­ станта. Должно существовать неявное преобразование типа выражения к типу параметра\*. Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения: - nt X = 1; ;;yt e с = 1 : .short у = 1: Тогда следующие вызовы метода Р, заголовок которого был описан ранее, будут синтаксически правильными: ; х ): Р( с ): Р( У ): Р( 200 ); Р( х / 4 + 1 ); Параметры-ссылки Во многих методах все величины, которые метод должен получить в качестве ис­ ходных данных, описываются в списке параметров, а величина, которую вычис­ ляет метод как результат своей работы, возвращается в вызывающий код с помо­ щью оператора return. Очевидно, что если метод должен возвращать более одной величины, такой способ не годится. Еще одна проблема возникает, если в метоле О неявных преобразованиях рассказывалось в разделе «Преобразования встроенных арифметических типов-значений» (см. с. 45). 210 Глава 9. Интерфейсы и структурные типы namespace ConsoleApplicationi { class Classl { public static IEnumerable Count( int from, int to ) { from = 1; while ( from <= to ) yield return from++: } static void MainO { foreach ( int i in Count( 1, 5 ) ) Console.WriteLineC i ); } } } Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. В листинге 9.7 описаны две дополнительные стратегии перебора элементов класса Stado, введен­ ного в листинге 9.5, — перебор в обратном порядке и выборка только тех объек­ тов, которые являются экземплярами класса Monster (для этого использован ме­ тод получения типа объекта GetType, унаследованный от базового класса object). Листин г 9.7. Реализация нескольких стратегий перебора using System; using System.Col lections; using MonsterLib: namespace ConsoleAppl icationi { class Monster { .. . } class Daemon { .. . } class Stado ; IEnumerable { private Monster[] mas; private int n; public StadoO mas = new Monster[10]; n = 0; public IEnumerator GetEnumeratorO for ( int i =0 ; i < n; ++i ) yield return mas[i]; public IEnumerable BackwardsO // в обратном порядке for ( int i = n - 1; i >= 0; --i ) yield return mas[i]: public IEnumerable MonstersOnlyO // только монстры 112 требуется изменить значение каких-либо передаваемых в него величин. В этих случаях используются параметры-ссылки. Признаком параметра-ссылки является ключевое слово re f перед описанием па­ раметра: ref тип имя Пример заголовка метода, имеющего один параметр-ссылку целого типа: voi d Р( re f in t х ) При вызове метода в область параметров копируется не значение аргумента, а его адрес, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Таким образом, параметры-ссылки передаются по адресу (чаще употребляется термин «передача по ссылке»). Метод работает непосредственно с перемен­ ной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке. ВНИМАНИ Е При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово ref . Исходные данные передавать в метод по ссылке не рекомендуется, чтобы исклю­ чить возможность их непреднамеренного изменения. Проиллюстрируем передачу параметров-значений и параметров-ссылок на при­ мере (листинг 5.4). а» % ж Листин г 5.4. Параметры-значения и параметры-ссылки usin g System; namespace ConsoleApplication i { clas s Class l { stati c voi d P( in t a, re f in t b ) { a = 44; b = 33; Console.WriteLine ( "внутри метода {0} {1}", a, b ); } stati c voi d Main() { i n t a = 2. b = 4; Console.WriteLineC "до вызова {0} {1}", a, b ); P( a, re f b ); Console.WriteLineC "после вызова {0} {!}" , a, b ): 113 Результаты работы этой программы: :;. вызовз 2 4 —утри метода 44 33 —еле вызова 2 33 Как видите, значение переменной а в функции Main не изменилось, поскольку пе- оеменная передавалась по значению, а значение переменной b изменилось пото- му, что она была передана по ссылке. Несколько иная картина получится, если передавать в метод не величины значи­ мых типов, а экземпляры классов, то есть величины ссылочных типов. Как вы помните, переменная-объект на самом деле хранит ссылку на данные, располо­ женные в динамической памяти, и именно эта ссылка передается в метод либо по адресу, либо по значению. В обоих случаях метод получает в свое распоряже­ ние фактический адрес данных и, следовательно, может их изменить. СОВЕ Т Д л я простоты можно считать, что объекты всегда передаются по ссылке. Разница между передачей объектов по значению и по ссылке состоит в том, что з последнем случае можно изменить саму ссылку, то есть после вызова метода она может указывать на другой объект. Выходные параметры Довольно часто возникает необходимость в методах, которые формируют не­ сколько величин, например, если в методе создаются объекты или инициализи­ руются ресурсы. В этом случае становится неудобным ограничение параметров- ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор out. Параметру, имеющему этот специфика­ тор, должно быть обязательно присвоено значение внутри метода, компилятор за этим следит. Зато в вызывающем коде можно ограничиться описанием пере­ менной без инициализации. Изменим описание второго параметра в листинге 5.4 так, чтобы он стал выход­ ным (листинг 5.5). Листин г 5.5. Выходные параметры ,sin g System; "smespace ConsoleApplication i clas s Class l { stati c voi d P( in t a. out in t b ) { a = 44; b = 33: Console.WriteLineC "внутри метода {0} {1}" . a, b ); } stati c voi d MainO ^ продолжение & 114 Глава 5. Классы: основные понятия Листин г 5.5 (продолжение) int а = 2, Ь; Р( a, out b ) : Console.WriteLineC "после вызова {0} {!}" , a, b ); При вызове метода перед соответствующим параметром тоже указывается клю­ чевое слово out. СОВЕ Т В списке параметров записывайте сначала все входные параметры, затем — все ссылки и выходные параметры. Давайте параметрам имена, по которым можно по­ лучить представление об их назначении. Ключевое слово this Каждый объект содержит свой экземпляр полей класса. Методы находятся в па­ мяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с по­ лями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию экземпляр. В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например: '< class Demo double у; public Demo TO // метод возвращает ссылку на экземпляр У- return this; • \*\* 1 •Ц public void Sety( double у ) this.у = у; // полю у присваивается значение параметра у Конструкторы Конструктор предназначен для инициализации объекта. Он вызывается авто­ матически при создании объекта класса с помощью операции new. Имя конст­ руктора совпадает с именем класса. Ниже перечислены свойства конструкторов: • Конструктор не возвращает значение, даже типа void. «чОнструкторы 115 • Класс может иметь несколько конструкторов с разными параметрами для раз­ ных видов инициализации. • Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение null . "J Конструктор, вызываемый без параметров, называется конструктором по умолчанию. До сих пор мы задавали начальные значения полей класса при описании класса | см., например, листинг 5.1). Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании юъектов требуется присваивать полю разные значения, это следует делать в кон- ;трукторе. В листинге 5.6 в класс Demo добавлен конструктор, а поля сделаны закрытыми (ненужные в данный момент элементы опущены). В программе соз­ даются два объекта с различными значениями полей. Листин г 5.6. Класс с конструктором ..s-ng System; "=mespace ConsoleApplication i clas s Demo { publi c Demo( in t a, double у ) // конструктор с параметрами { this. а = а; this. у = у : } publi c double Gety O // метод получения поля у { retur n у ; } i n t а; double у ; } clas s Class l { stati c voi d Main O { Demo a = new Demo( 300, 0.002 ) Console.WriteLineC a.Gety O ): Demo b = new Demo( 1, 5.71 ); Console.WriteLineC b.Gety O ); } } } // вызов конструктора // результат ; 0,002 // вызов конструктора / / результат : 5,71 116 Глава 5. Классы: основные понятия Часто бывает удобно задать в классе несколько конструкторов, чтобы обес­ печить возможность инициализации объектов разными способами. Следую­ щий пример несколько «притянут за уши», но тем не менее иллюстрирует этот тезис: class Demo { public DemoC int a ) // конструктор 1 { this.a = a: this.у = 0.002; } public Demo( double у ) // конструктор 2 { this.a = 1: this.у = у; } Demo x = new Demo( 300 ); // вызов конструктора 1 Demo у = new DemoC 5.71 ); // вызов конструктора 2 Все конструкторы должны иметь разные сигнатуры. Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, удобно вызвать первый конструктор из второго. Для этого используется уже известное вам ключевое слово this в дру­ гом контексте, например: class Demo { public DemoC int a ) // конструктор 1 { this.a = a; } public DemoC int a, double у ) : thisC a ) // вызов конструктора 1 { this.у = у; } } Конструкция, находящаяся после двоеточия, называется инициализатором, то есть тем кодом, который исполняется до начала выполнения тела конст­ руктора. Как вы помните, все классы в С# имеют общего предка — класс object. Конст­ руктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка. Это можно сделать и явным образом с помощью < a.Length; ++i ) Console.WriteC "\t" + a[i] ); Console.WriteLineC); } } ) Методы Sort, IndexOf и BinarySearch являются статическими, поэтому к ним об­ ращаются через имя класса, а не экземпляра, и передают в них имя массива. Двоичный поиск можно применять только для упорядоченных массивов. Он выполняется гораздо быстрее, чем линейный поиск, реализованный в методе IndexOf. В листинге поиск элемента, имеющего значение 18, выполняется обоими этими способами. ПРИМЕЧАНИ Е Рассмотренные методы имеют по несколько версий (в этом случае употребляется термин «перегруженные методы»), что позволяет выполнять поиск, сортировку и копирование как в массиве целиком, как и в его указанном диапазоне. В классе Classl описан вспомогательный статический метод PrintArray, предназна­ ченный для вывода массива на экран. В него передаются два параметра: строка заголовка header и массив. Количество элементов массива определяется внутри метода с помощью свойства Length. Таким образом, этот метод можно использо­ вать для вывода любого целочисленного одномерного массива. Результат работы программы: /сходный массив: 24 50 18 3 16 -7 9 -1 "орядоченный массив: -7 -1 3 9 16 18 24 50 Для того чтобы применять метод PrintArray к массивам, состоящим из элементов другого типа, можно описать его второй параметр как Array. Правда, при этом значение элемента массива придется получать с помощью метода Get Value, посколь­ ку доступ по индексу для класса Array не предусмотрен. Обобщенный метод вы­ зола массива выглядит так: :.:ii c static void PrintArrayC string header, Array a ) Console.WriteLineC header ); for ( int i = 0; i < a.Length; ++i ) Console.WriteC "\t" + a.GetValue(i) ); Console.WriteLineC); 136 Глава 6. Массивы и строки В листинге 6.4 продемонстрировано применение элементов класса Array при ра­ боте со ступенчатым массивом. Листин г 6.4. Использование методов класса Array со ступенчатым массивом using System; namespace ConsoleApplicationi { class Classl / №?£ Ш МЮ { int[][ ] a = new int[3][]; a[0] = new int [5] { 24. 50. 18. 3. 16 }; a[l] = new int [3] { 7, 9. -1 }; a[2] = new int [4] { 6. 15, 3, 1 }; Console.WriteLineC "Исходный массив:" ); for ( int i = 0; i < a.Length; ++i ) { for ( int j = 0; j < a[i].Length; ++j ) Console.WriteC "\t" + a[i][j ] ); Console.WriteLineC); } Console.WriteLineC Array.IndexOf( a[0], 18 ) ); Обратите внимание на то, как внутри цикла по строкам определяется длина каж­ дого массива. Результат работы программы: Исходный массив: 24 50 18 3 16 7 9 -1 6 15 3 1 2 Оператор foreach Оператор foreach применяется для перебора элементов в специальным обра­ зом организованной группе данных. Массив является именно такой группой1 . Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять их перебор по индексу: мы про­ сто указываем на необходимость перебрать все элементы группы. Синтаксис оператора: foreach ( тип имя in выражение ) телоцикла 1 Отчего это так, вы поймете, изучив главу 9. Оператор foreach 137 Имя задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива выражение (в качестве выражения чаще всего применяется имя массива или другой группы данных). В простом или составном операторе, представляющем собой тело цикла, выполняются дей­ ствия с переменной цикла. Тип переменной должен соответствовать типу эле­ мента массива. Например, пусть задан массив: int[] а = {24 , 50, 18, 3, 16, -7, 9, -1 }; Вывод этого массива на экран с помощью оператора foreach выглядит следую­ щим образом: foreach ( int х in а ) Console.WriteLineC х ); Этот оператор выполняется так: на каждом проходе цикла очередной элемент массива присваивается переменной х и с ней производятся действия, записанные в теле цикла. Ступенчатый массив из листинга 6.4 вывести на экран с помощью оператора foreach немного сложнее, чем одномерный, но все же проще, чем с помощью цикла for: foreach ( int[] х in а ) { foreach ( int у in х ) Console.WriteC "\t" + у ): Console.WriteLineC); } В листинге 6.5 решается та же задача, что и в листинге 6.1, но с использованием цикла foreach. Обратите внимание на то, насколько понятнее стала программа. Листин г 6.5. Работа с одномерным массивом с использованием цикла foreach using System; namespace ConsoleApplicationi { class Classl { static void MainO { int[] a = { 3, 12, 5, -9, 8, -4 }; Console.WriteLineC "Исходный массив:" ); foreach ( int el em in a ) Console.WriteC "\t" + elem ); Console.WriteLineC); long sum = 0; //сумма отрицательных элементов int num =0 ; // количество отрицательных элементов foreach С int elem in а ) if ( elem < 0 ) ^ продолжение & 138 Глава 6. Массивы и строки Листин г 6.5 (продолжение) sum += elem; ++num; } Console.WriteLineC "sum = " + sum ); Console.WriteLineC "num = " + num ); int max = a[0]; // максимальный элемент foreach ( int elem in a ) if ( elem > max ) max = elem; Console.WriteLineC "max = " + max ); } } } А вот как можно переписать метод вывода массива из листинга 6.3: public static void PrintArrayC string header, Array a ) { Console.WriteLineC header ); foreach ( object x in a ) Console.WriteC "\t" + x ); Console.WriteLineC); } Такая запись становится возможной потому, что любой объект может быть неяв­ но преобразован к типу его базового класса, а тип object, как вы помните, являет­ ся корневым классом всей иерархии. Когда вы продвинетесь в изучении С# до раздела «Виртуальные методы» (см. с. 178), вы поймете механизм выполнения этого кода, а пока можете просто им пользоваться. ВНИМАНИ Е Ограничением оператора foreach является то, что с его помощью можно только просматривать значения в группе данных, но не изменять их. Массивы объектов При создании массива, состоящего из элементов ссылочного типа, память выде­ ляется только под ссылки на элементы, а сами элементы необходимо разместить в хипе явным образом. В качестве примера создадим массив из объектов некото­ рого класса Monster: using System; namespace ConsoleApplicationi { class Monster { .. . } class Classl Символы и строки 139 { static void MainO { Random rnd = new Random(); const int n = 5; Monster[] stado = new Monster[n]: // 1 for ( int i = 0; i < n; ++i ) 112 { stado[i] = new Monster( rnd.Next( 1, 100 ), rnd.Next( 1, 200 ), "Crazy" + i .ToStringO ): } foreach ( Monster x in stado ) x.Passport(); // 3 } } Результат работы программы: 'Onster CrazyO health = 18 ammo = 94 ••'onster Crazyl health = 85 ammo = 75 '•'onster Crazy2 health = 13 ammo = 6 •Onster Crazy3 health = 51 ammo = 104 Onster Crazy4 health = 68 ammo = 114 В программе для получения случайных значений использован стандартный класс Random, который описан далее в этой главе (см. с. 148). В операторе 1 выде­ ляется пять ячеек памяти под ссылки на экземпляры класса Monster?, и эти ссыл­ ки заполняются значением null. В цикле 2 создаются пять объектов: операция -ew выделяет память в хипе необходимого для хранения полей объекта объема, а конструктор объекта заносит в эти поля соответствующие значения (выполня­ ется версия конструктора с тремя параметрами). Цикл 3 демонстрирует удобство применения оператора foreach для работы с массивом. Символы и строки Обработка текстовой информации является, вероятно, одной из самых распро- .траненных задач в современном программировании, и С# предоставляет для ее г-ешения широкий набор средств: отдельные символы, массивы символов, изме­ няемые и неизменяемые строки и регулярные выражения. Символы Символьный тип char предназначен для хранения символов в кодировке Unicode. Способы представления символов рассматривались в разделе «Литералы» (см. с. 26). Символьный тип относится к встроенным типам данных С# и соответствует стан­ дартному классу Char библиотеки .NET из пространства имен System. В этом классе определены статические методы, позволяющие задать вид и категорию символа, 104 Глава 5. Классы: основные понятия одной величины ссылочного типа, это может отразиться на другой (в данном слу­ чае, если изменить объект через ссылку с, объект b также изменит свое значение). Объекты Рис. 5.2. Присваивание объектов Аналогичная ситуация с операцией проверки на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке объекты b и с равны, но а не равно b даже при равенстве их значений или если они обе равны nullI). Данные: поля и константы Данные, содержащиеся в классе, могут быть переменными или константами и задаются в соответствии с правилами, рассмотренными в разделе «Перемен­ ные» (см. с. 38) и «Именованные константы» (см. с. 41). Переменные, описанные в классе, называются полями класса. При описании элементов класса можно также указывать атрибуты и специфика­ торы, задающие различные характеристики элементов. Синтаксис описания эле­ мента данных приведен ниже: [ атрибуты ] [ спецификаторы ] [ const ] тип имя [ = начальное\_значение ] До атрибутов мы доберемся еще не скоро, в главе 12, а возможные специфика­ торы полей и констант перечислены в табл. 5.2. Для констант можно использо­ вать только спецификаторы 1-6. Таблица 5.2. Спецификаторы полей и констант класса № Спецификатор Описание new public protected internal Новое описание поля, скрывающее унаследованный элемент класса Доступ к элементу не ограничен Доступ только из данного и производных классов Доступ только из данной сборки 140 Глава 6. Массивы и строки 5 1 I -sir S i а также преобразовать символ в верхний или нижний регистр и в число. Основ­ ные методы приведены в табл. 6.2. Таблица 6.2. Основные методы класса System.Char Метод Описание GetNumericValue GetUnicodeCategory IsControl IsDigit IsLetter IsLetterOrDigit IsLower IsNumber IsPunctuation IsSeparator Isllpper IsWhiteSpace Parse ToLower ToUpper MaxValue, MinValue Возвращает числовое значение символа, если он является цифрой, и -1 в противном случае Возвращает категорию Unicode-символа1 Возвращает true, если символ является управляющим Возвращает true, если символ является десятичной цифрой Возвращает true, если символ является буквой Возвращает true, если символ является буквой или цифрой Возвращает true, если символ задан в нижнем регистре Возвращает true, если символ является числом (десятичным или шестнадцатеричным) Возвращает true, если символ является знаком препинания Возвращает true, если символ является разделителем Возвращает true, если символ записан в верхнем регистре Возвращает true, если символ является пробельным (пробел, перевод строки и возврат каретки) Преобразует строку в символ (строка должна состоять из одного символа) Преобразует символ в нижний регистр Преобразует символ в верхний регистр Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления) В листинге 6.6 продемонстрировано использование этих методов. Листин г 6.6. Использование методов класса System.Char using System; namespace ConsoleApplicationi { class Classl { static void MainO { try char b = с = 0x63' , d = Ou0032'; / / 1 1 Bee Unicode-символы разделены на категории, например, десятичные цифры (DecimalDigitNumber), римские цифры (LetterNumber), разделители строк (LineSeparator), буквы в ниж­ нем регистре (LowercaseLetter) и т. д. Символы и строки 141 Console.WriteLine( "{0} {1} {2}". b. с. d ); Console.WriteLine( "{0} {1} {2}", char.ToLower(b). char.Tol)pper(c), char.GetNumericValue(d) ); char a; do // 2 { Console.Write( "Введите символ: " ); a = char.Parse( Console.ReadLine() ); Console.WriteLine( "Введен символ {0}. его код - {1}". а. Сint)а ); if (char.IsLetter(a)) Console.WriteLine("Буква"); if (char. IsUpper(a)) Console.WriteLine("Bepxий peг."); if (char.IsLower(a)) Console.WriteLine("Нижний peг."); if (char.IsControl (a)) Console.WriteLine("Упpaвляющий"); if (char.IsNumber(a)) Console.WriteLine("Число"); if (char.IsPunctuation(a)) Console.WriteLine("Paздeлитeль"); } while (a != 'q') : catch Console.WriteLine( "Возникло исключение" ); return; } В операторе 1 описаны три символьных переменных. Они инициализируются символьными литералами в различных формах представления. Далее выполня­ ются вывод и преобразование символов. В цикле 2 анализируется вводимый с клавиатуры символ. Можно вводить и управ­ ляющие символы, используя сочетание клавиши Ctrl с латинскими буквами. При вводе использован метод Parse, преобразующий строку, которая должна содер­ жать единственный символ, в символ типа char. Поскольку вводится строка, ввод каждого символа следует завершать нажатием клавиши Enter. Цикл выполняет­ ся, пока пользователь не введет символ q. Вывод символа сопровождается его кодом в десятичном виде 1 . Для вывода кода используется явное преобразование к целому типу. Явное преобразование из сим­ волов в строки и обратно в С# не существует, неявным же образом любой объект, в том числе и символ, может быть преобразован в строку 2 , например: string s = 'к' + 'о' + О' ; // результат - строка "кот" При вводе и преобразовании могут возникать исключительные ситуации, напри­ мер, если пользователь введет пустую строку. Для «мягкого» завершения про­ граммы предусмотрена обработка исключений. 1 Обратите внимание на коды русских букв: как далеко от латинских они оказались в ко­ дировке Unicode! 2 Об этом рассказывалось в разделе «Простейший ввод-вывод» (см. с. 59). 142 Глава 6. Массивы и строки Массивы символов Массив символов, как и массив любого иного типа, построен на основе базового класса Array, некоторые свойства и методы которого были перечислены в табл. 6.1. Применение этих методов позволяет эффективно решать некоторые задачи. Про­ стой пример приведен в листинге 6.7. Листин г 6.7. Работа с массивом символов using System; namespace ConsoleApplicationi { class Classl { static void MainO { char[] a = { 'm', 'a', 's', 's', 'i' , V }; // 1 char[] b = "а роза упала на лапу азора" .ToCharArrayO; // 2 PrintArray( "Исходный массив а:", а ); int pos = Array.IndexOf( a, 'm' ); I a[pos] = 'M'; \*T PrintArrayC "Измененный массив a:", a ); "3 4 » PrintArray( "Исходный массив b:", b ); if Array.Reverse! b ); j£ PrintArrayC "Измененный массив b:", b ); "? # public static void PrintArrayC string header, Array a ) I £ Console.WriteLineC header ); • foreach ( object x in a ) Console.WriteC x ); f£ Console.WriteLineC "An" ); Результат работы программы: Исходный массив а: massiv Измененный массив а: Massi V Исходный массив Ь: а роза упала на лапу азора Измененный массив Ь: ароза упал ан алапу азор а Символы и строки 143 Символьный массив можно инициализировать, либо непосредственно задавая его элементы (оператор 1), либо применяя метод ToCharArray класса string, который разбивает исходную строку на отдельные символы (оператор 2). Строки типа string Тип string, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом С# . Ему соответствует базовый класс System.String библиотеки .NET. Создать строку можно несколькими способами: string s; // инициализация отложена string t = "qqq"; // инициализация строковым литералом string u = new stringO '. 20); // конструктор создает строку из 20 пробелов char[] а = { '0' , '0' , '0 ' }; // массив для инициализации строки string V = new string( а ); // создание из массива символов Д ля строк определены следующие операции: • присваивание (=); • проверка на равенство (==); • проверка на неравенство (!=) ; • обращение по индексу ([]) ; • сцепление (конкатенация) строк (+). Несмотря на то что строки являются ссылочным типом данных, на равенство и неравенство проверяются не ссылки, а значения строк. Строки равны, если име­ ют одинаковое количество символов и совпадают посимвольно. Обращаться к отдельному элементу строки по индексу можно только для полу­ чения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к так называемым неизменяемым типам данныхК Методы, из­ меняющие содержимое строки, на самом деле создают новую копию строки. Не­ используемые «старые» копии автоматически удаляются сборщиком мусора. В классе System.String предусмотрено множество методов, полей и свойств, поз­ воляющих выполнять со строками практически любые действия. Основные эле­ менты класса приведены в табл. 6.3. Таблица 6.3. Основные элементы класса System.String Название Вид Описание Compare Статический метод Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и особенностей национального представления дат и т. д. продолжение # 1 Наследовать от этого класса также запрещается. 144 Глава 6. Массивы и строки Таблица 6.3 (продолжение) Название Вид Описание CompareOrdi nal Статический метод Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки CompareTo Метод Сравнение текущего экземпляра строки с другой строкой Concat Статический метод Конкатенация строк. Метод допускает сцепление произвольного числа строк Copy Статический метод Создание копии строки Empty Статическое поле Пустая строка (только для чтения) Format Статический метод Форматирование в соответствии с заданными спецификаторами формата (см. далее) IndexOf, Методы Определение индексов первого и последнего IndexOf Any, вхождения заданной подстроки или любого LastlndexOf, символа из заданного набора LastIndexOfAny Insert Метод Вставка подстроки в заданную позицию Intern, Статические методы Возвращает ссылку на строку, если такая Islnterned уже существует. Если строки нет, Intern добавляет строку во внутренний пул, Is Intern возвращает nul I Join Статический метод Слияние массива строк в единую строку. Между элементами массива вставляются разделители (см. далее) Length Свойство Длина строки (количество символов) PadLeft, Методы Выравнивание строки по левому или правому PadRight краю путем вставки нужного числа пробелов в начале или в конце строки Remove Метод Удаление подстроки из заданной позиции Replace Метод Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом Split Метод Разделяет строку на элементы, используя заданные разделители. Результаты помещаются в массив строк StartsWith, Методы Возвращает true или false в зависимости от EndsWith того, начинается или заканчивается строка заданной подстрокой Substring Метод Выделение подстроки, начиная с заданной позиции ToCharArray Метод Преобразование строки в массив символов ToLower, Методы Преобразование символов строки к нижнему Tollpper и ли верхнему регистру Символы и строки 145 Название Вид Описание Tim, T rimStart , "rimEnd Методы Удаление пробелов в начале и конце строки или только с одного ее конца (обратные по отношению к методам PadLeft и PadRight действия) Пример применения методов приведен в листинге 6.8. Листин г 6.8. Работа со строками типа string jsin g System: namespace ConsoleApplication i clas s Class l { stati c voi d MainO { strin g s = "прекрасная королева Изольда": Console.WriteLineC s ); strin g sub = s.Substring C 3 ).Remove( 12, 2 ); III Console.WriteLineC sub ); string[ ] mas = s.Split O ') ; 112 strin g joine d = string.Jom ( " ! " , mas ); Console.WriteLineC joine d ); Console.WriteLineC "Введите строку " ); strin g x = Console.ReadLineO; // 3 Console.WriteLineC "Вы ввели строку " + x ); double a = 12.234; i n t b = 29; Console.WriteLineC " a = {0,6;C } b = {1,2:X}" , a, b ); II A Console.WriteLineC " a = {0,6:0.# } a = {1,5:0. # ' руб. '}" • a, b ); li b } } Результат работы программы: прекрасная королева Изольда красная корова Изольда •рекрасная! королева! Изольда Введите строк у че хочу! Вы ввели строк у не хочу! ; = 12,23р. b = 10 В операторе 1 выполняются два последовательных вызова методов: метод Substrin g возвращает подстроку строки s, которая содержит символы исходной строки, на­ чиная с третьего. Для этой подстроки вызывается метод Remove, удаляющий из нее два символа, начиная с 12-го. Результат работы метода присваивается пере­ менной sub. 146 Глава 6. Массивы и строки Аргументом метода Split (оператор 2) является разделитель, в данном случае — символ пробела. Метод разделяет строку на отдельные слова, которые заносятся в массив строк mas. Статический метод Join (он вызывается через имя класса) объ­ единяет элементы массива mas в одну строку, вставляя между каждой парой слов строку "! ". Оператор 3 напоминает вам о том, как вводить строки с клавиатуры. Форматирование строк В операторе 4 из листинга 6.8 неявно применяется метод Format, который заменя­ ет все вхождения параметров в фигурных скобках значениями соответствующих переменных из списка вывода. После номера параметра можно задать минималь­ ную ширину поля вывода, а также указать спецификатор формата, который оп­ ределяет форму представления выводимого значения. В общем виде параметр задается следующим образом: {п [,т[:спецификатор\_формата]]} Здесь п — номер параметра. Параметры нумеруются с нуля, нулевой параметр заменяется значением первой переменной из списка вывода, первый параметр — второй переменной и т. д. Параметр m определяет минимальную ширину поля, которое отводится под выводимое значение. Если выводимому числу достаточно меньшего количества позиций, неиспользуемые позиции заполняются пробела­ ми. Если числу требуется больше позиций, параметр игнорируется. Спецификатор формата, как явствует из его названия, определяет формат вы­ вода значения. Например, спецификатор С (Currency) означает, что параметр должен форматироваться как валюта с учетом национальных особенностей пред­ ставления, а спецификатор X (Hexadecimal) задает шестнадцатеричную форму представления выводимого значения. ПРИМЕЧАНИ Е До настоящего момента мы пользовались сокращенной записью, задавая только но­ мера параметров. Список спецификаторов формата приведен в приложении. В операторе 5 используются так называемые пользовательские шаблоны формати­ рования. Если приглядеться, в них нет ничего сложного: после двоеточия задается вид выводимого значения посимвольно, причем на месте каждого символа может стоять либо #, либо 0. Если указан знак #, на этом месте будет выведена цифра числа, если она не равна нулю. Если указан 0, будет выведена любая цифра, в том числе и 0. В табл. 6.4 приведены примеры шаблонов и результатов вывода. Таблица 6.4. Примеры применения пользовательских шаблонов форматирования Число Шаблон Вид 1,243 1,243 0,1 0,1 00.00 01,24 #.## 1,24 00.00 00,10 #.## ,1 Символы и строки 147 Пользовательский шаблон может также содержать текст, который в общем слу­ чае заключается в апострофы. Строки типа StringBuilder Возможности, предоставляемые классом string, широки, однако требование не­ изменности его объектов может оказаться неудобным. В этом случае для работы со строками применяется класс StringBuilder, определенный в пространстве имен System.Text и позволяющий изменять значение своих экземпляров. При создании экземпляра обязательно использовать операцию new и конструк- тор, например: StringBuilder а = new StringBuilder( : // 1 StringBuilder b = new StringBuilder( "qwerty" ); // 2 StringBuilder с = new StringBuilder! 100 ); // 3 StringBuilder d = new StringBuilder! "qwerty", 100 ).; // 4 StringBuilder e = new StringBuilder! "qwerty", 1, 3, 100 ); / / 5 В конструкторе класса указываются два вида параметров: инициализирующая строка или подстрока и объем памяти, отводимой под экземпляр (емкость буфе­ ра). Один или оба параметра могут отсутствовать, в этом случае используются их значения по умолчанию. Если применяется конструктор без параметров (оператор 1), создается пустая строка размера, заданного по умолчанию (16 байт). Другие виды конструкторов задают объем памяти, выделяемой строке, и/или ее начальное значение. На­ пример, в операторе 5 объект инициализируется подстрокой длиной 3 символа, начиная с первого (подстрока "wer"). Основные элементы класса StringBuilder приведены в табл. 6.5. Таблица 6.5. Основные элементы класса System.Text.StringBuilder Название Вид Описание Append Метод Добавление в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки типа string AppendFormat Метод Добавление форматированной строки в конец строки Capacity Свойство Получение или установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, генерируется исключение ArgumentOutOfRangeExcepti on Insert Метод Вставка подстроки в заданную позицию Length Свойство Длина строки (количество символов) MaxCapacity Свойство Максимальный размер буфера Remove Метод Удаление подстроки из заданной позиции продолжение & 148 Глава 6. Массивы и строки Таблица 6.5 (продолжение) Название Вид Описание Replace Метод Замена всех вхождений заданной подстроки или символа новой подстрокой или символом ToString Метод Преобразование в строку типа string Пример применения методов приведен в листинге 6.9. Листин г 6.9. Работа со строками типа StringBuilder using System; using System.Text; namespace ConsoleAppl icationi { class Classl { static void MainO { Console.WriteC "Введите зарплату: " ); double salary = double.ParseC Console.ReadLineO ); StringBuilder a = new StringBuiIder(); a.Appendt "зарплата " ); a.AppendFormatC "{0. 6:C} - в год {1 , 6:C}". salary, salary \* 12 ); Console.WriteLineC a ); a.ReplaceC "p.", "тыс.$" ); Console.WriteLineC "А лучше было бы: " + a ); } } } Результат работы программы: Введите зарплату: 3500 зарплата 3 500.00р. - в год 42 000,00р. А лучше было бы: зарплата 3 500,ООтыс.$ - в год 42 000.ООтыс.$ Емкость буфера не соответствует количеству символов в строке и может увеличи­ ваться в процессе работы программы как в результате прямых указаний програм­ миста, так и вследствие выполнения методов изменения строки, если строка в ре­ зультате превышает текущий размер буфера. Программист может уменьшить размер буфера с помощью свойства Capacity, чтобы не занимать лишнюю память. Класс Random При отладке программ, использующих массивы, удобно иметь возможность ге­ нерировать исходные данные, заданные случайным образом. В библиотеке С# на этот случай есть класс Random, определенный в пространстве имен System. < n; ++i ) Console.Write ! "{ 0 ,6:0.##}" , a.NextDouble() ); Console.WriteLine ! "An Диапазон [0 , 1000]: " ); f o r ( in t l = 0; i < n; ++i ) Console.Write ! " " + b.Next( 1000 ) ); Console.WriteLine ! "\n Диапазон [-10 , 10]: " ); продолжение Данные: поля и константы 105 Спецификатор Описание 5 protecte d interna l Доступ только из данного и производных классов и из данной сборки 6 privat e Доступ только из данного класса - stati c Одно поле для всех экземпляров класса S readonl y Пол е доступно только для чтения 9 volatil e Пол е может изменяться другим процессом или системой По умолчанию элементы класса считаются закрытыми (private) . Для полей клас­ са этот вид доступа является предпочтительным, поскольку ноля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям. ВНИМАНИ Е Поля, описанные со спецификатором static , а также константы существуют в един­ ственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Если класс содержит только статические эле­ менты, экземпляр класса создавать не требуется. Именно этим фактом мы пользо­ вались во всех предыдущих листингах. Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева — имя экземпляра для обычных нолей или имя класса для статических. В листинге 5.1 приведены пример простого класса Demo и два способа обращения к его полям. Листин г 5.1 . Класс Demo, содержащий поля и константу .sin g System. "amespace Consol eAppl ication i clas s Demo { publi c in t a = 1; publi c cons t double с = 1.66; publi c stati c strin g s = "Demo" double y ; // поле данных // константа II статическое поле класса // закрытое поле данных clas s Class l { stati c voi d MainO Demo x = new Demo(); Console.WriteLine ! x. a ); Console.WriteLine ! Demo.с ): Console.WriteLine ! Demo.s ); // создание экземпляра класса Demo // x. a - обращение к полю класса // Demo.с - обращение к константе // обращение к статическому полю 150 Глава 6. Массивы и строки Листин г 6.10 {продолжение) for ( int i = 0; i < n; ++i ) Console.WriteC " " + a.NextC-10, 10) ); Console.WriteLineC "\n Массив [0, 255]:" ); byte[] mas = new byte[n]; a.NextBytesC mas ): for (int i = 0; i < n; ++i) Console.WriteC " " + mas[i] ); Результат работы программы: Диапазон [0, 1]: 0,02 0,4 0,24 0,55 0,92 0,84 Диапазон [0, 1000]: 248 ПО 467 771 657 432 Диапазон [-10, 10]: -8 9 -6 -10 7 4 9 -5 Массив [0, 255]: 181 105 60 50 70 77 9 28 133 150 0,9 0,78 0,78 0,74 354 943 101 642 -2 -1 Рекомендации по программированию Используйте для хранения данных массив, если количество однотипных элемен­ тов, которые требуется обработать в вашей программе, известно или, по крайней мере, известно максимальное количество таких элементов. В последнем случае память под массив выделяется «по максимуму», а фактическое количество эле­ ментов хранится в отдельной переменной, которая вычисляется в программе. При работе с массивом нужно обязательно предусматривать обработку ис­ ключения IndexOutOfRangeException, если индекс для обращения к массиву вы­ числяется в программе по формулам, а не задается с помощью констант или счетчиков циклов for. СОВЕ Т При отладке программ, использующих массивы, исходные данные удобно подго­ товить в текстовом файле. Это позволит продумать, какие значения элементов необходимо задать, чтобы протестировать выполнение каждой ветви программы. Например, исходные данные для программы, вычисляющей в одномерном массиве номер первого элемента, равного нулю, должны включать варианты, когда такой элемент встречается в массиве ни одного, один и более раз. Если количество элементов, обрабатываемых программой, может быть произволь­ ным, удобнее использовать не массив, а другие структуры данных, например па­ раметризованные коллекции, которые рассматриваются в главе 13. Рекомендации по программированию 151 При работе со строками необходимо учитывать, что в С# строка типа string яв­ ляется неизменяемым типом данных, то есть любая операция изменения строки на самом деле возвращает ее копию. Для изменения строк используется тип OnngBuilder. Прежде чем описывать в программе какое-либо действие со стро­ ками, полезно посмотреть, нет ли в списке элементов используемого класса под­ ходящих методов и свойств. Для эффективного поиска и преобразования текста в соответствии с заданными шаблонами используются так называемые регулярные выражения, которые рас­ смотрены в главе 15. Глава 7 Классы: подробности В этой главе мы продолжим знакомство с элементами классов, начатое в главе 5. Сначала мы рассмотрим дополнительные возможности методов, не описанные в главе 5, а затем перейдем к новым элементам класса — индексаторам, операци­ ям и деструкторам. Перегрузка методов Часто бывает удобно, чтобы методы, реализующие один и тот же алгоритм для различных типов данных, имели одно и то же имя. Если имя метода является осмысленным и несет нужную информацию, это делает программу более по­ нятной, поскольку для каждого действия требуется помнить только одно имя. Использование нескольких методов с одним и тем же именем, но различными типами параметров называется перегрузкой методов. ПРИМЕЧАНИ Е Мы уже использовали перегруженные версии методов стандартных классов и даже сами создавали перегруженные методы, когда описывали в классе несколько конструкторов. Компилятор определяет, какой именно метод требуется вызвать, по типу факти­ ческих параметров. Этот процесс называется разрешением (resolution) перегруз­ ки. Тип возвращаемого методом значения в разрешении не участвует 1 . Механизм разрешения основан на достаточно сложном наборе правил, смысл которых сво­ дится к тому, чтобы использовать метод с наиболее подходящими аргументами и выдать сообщение, если такой не найдется. Допустим, имеется четыре варианта метода, определяющего наибольшее значение: 1 Если последний параметр метода имеет модификатор params, о котором рассказывается далее в этой главе, этот параметр также не учитывается. Рекурсивные методы 153 // Возвращает наибольшее из двух целых: i n t max( in t a, in t b ) // Возвращает наибольшее из трех целых: i n t max( in t a, in t b, in t с ) // Возвращает наибольшее из первого параметра и длины второго: i n t max ( in t a. strin g b ) // Возвращает наибольшее из второго параметра и длины первого: i n t max ( strin g b, in t a ) Console.WriteLineC max( 1, 2 ) ); Console.WriteLineC max( 1, 2. 3 ) ); Console.WriteLineC max( 1, "2 " ) ): Console.WriteLineC max( "1" , 2 ) ); При вызове метода max компилятор выбирает вариант метода, соответствующий типу передаваемых в метод аргументов (в приведенном примере будут последо­ вательно вызваны все четыре варианта метода). Если точного соответствия не найдено, выполняются неявные преобразова­ ния типов в соответствии с общими правилами, например, bool и char в int , floa t в double и т. п. Если преобразование невозможно, выдается сообщение об ошиб­ ке. Если соответствие на одном и том же этапе может быть получено более чем одним способом, выбирается «лучший» из вариантов, то есть вариант, содержа­ щий меньшие количество и длину преобразований в соответствии с правилами, описанными в разделе «Преобразования встроенных арифметических типов-зна­ чений» (см. с. 45). Если существует несколько вариантов, из которых невозмож­ но выбрать лучший, выдается сообщение об ошибке. Вам уже известно, что все методы класса должны различаться сигнатурами. Это понятие было введено в разделе «Методы» (см. с. 106). Перегруженные методы имеют одно имя, но должны различаться параметрами, точнее, их типами и спо­ собами передачи (ou t или ref) . Например, методы, заголовки которых приведены ниже, имеют различные сигнатуры и считаются перегруженными: i n t maxC in t a. in t b ) i n t max( in t a, re f in t b ) Перегрузка методов является проявлением полиморфизма, одного из основных свойств ООП . Программисту гораздо удобнее помнить одно имя метода и ис­ пользовать его для работы с различными типами данных, а решение о том, какой вариант метода вызвать, возложить на компилятор. Этот принцип широко ис­ пользуется в классах библиотеки .NET. Например, в стандартном классе Console метод WriteLin e перегружен 19 раз для вывода величин разных типов. Рекурсивные методы Рекурсивным называется метод, который вызывает сам себя. Такая рекурсия назы­ вается прямой. Существует еще косвенная рекурсия, когда два или более метода вызывают друг друга. Если метод вызывает себя, в стеке создается копия значений 154 Глава 7. Классы: подробности его параметров, как и при вызове обычного метода, после чего управление пере­ дается первому исполняемому оператору метода. При повторном вызове этот процесс повторяется. Ясно, что для завершения вычислений каждый рекурсивный метод должен содер­ жать хотя бы одну нерекурсивную ветвь атгоритма, заканчивающуюся оператором возврата. При завершении метода соответствующая часть стека освобождается и управление передается вызывающему методу, выполнение которого продолжа­ ется с точки, следующей за рекурсивным вызовом. Классическим примером рекурсивной функции является функция вычисления факториала (это не означает, что факториал следует вычислять именно так). Для того чтобы получить значение факториала числа п, требуется умножить на п фак­ ториал числа (п - 1). Известно также, что 0 != 1 и 1 != 1: long facte long n ) { if ( n == 0 11 n == 1 ) return 1; // нерекурсивная ветвь return ( n \* fact( n - 1 ) ); // рекурсивная ветвь } To же самое можно записать короче: long factt long n ) { return ( n > 1 ) ? n \* fact e n - 1 ) : 1 : } Рекурсивные методы чаще всего применяют для компактной реализации рекур­ сивных алгоритмов, а также для работы со структурами данных, описанными ре­ курсивно, например, с двоичными деревьями (понятие о двоичном дереве дается в главе 13). Любой рекурсивный метод можно реализовать без применения ре­ курсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно. К достоинствам рекурсии можно отнести компактность записи, к недостат­ кам — расход времени и памяти на повторные вызовы метода и передачу ему ко­ пий параметров, а главное, опасность переполнения стека. Методы с переменным количеством аргументов Иногда бывает удобно создать метод, в который можно передавать разное ко­ личество аргументов. Язык С# предоставляет такую возможность с помощью ключевого слова params. Параметр, помеченный этим ключевым словом, раз­ мещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например: public int Calculate( int a, out int c. params int[] d ) ... В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество Методы с переменным количеством аргументов 155 элементов массива получают с помощью его свойства Length. В качестве примера рассмотрим метод вычисления среднего значения элементов массива (листинг 7.1). Листин г 7.1 . Метод е переменным числом параметров .sing System; 'smespace ConsoleApplicationi class Classl { public static double Average! params int[] a ) { if ( a.Length == 0 ) throw new Exception! "Недостаточно аргументов в методе" ); double av = 0; foreach ( int elem in a ) av += elem; return av / a.Length; } static void Main!) { try { int[] a = { 10, 20, 30 }; Console.WriteLine! Average! a ) ); III intC] b = { -11. -4, 12, 14, 32, -1, 28 }; Console.WriteLine! Average! b ) ); /12 short z = 1. e = 12; byte v = 100; Console.WriteLine! Average! z, e, v ) ); //3 Console.WriteLine! Average!) ); // 4 } catch! Exception e ) { Console.WriteLine! e.Message ); return; } } } Результат работы программы: -^достаточно аргументов в методе ПРИМЕЧАНИ Е В данном алгоритме отсутствие аргументов при вызове метода Average является ошибкой. Этот случай обрабатывается генерацией исключения. Если не обработать эту ошибку, результат вычисления среднего будет равен «не числу » (NaN) вследст­ вие деления на ноль в операторе возврата из метода. 156 Глава 7. Классы: подробности Параметр-массив может быть только один и должен располагаться последним в списке. Соответствующие ему аргументы должны иметь типы, для которых воз­ можно неявное преобразование к типу массива. Метод Main Метод, которому передается управление после запуска программы, должен иметь имя Main и быть статическим. Он может принимать параметры из внешнего окру­ жения и возвращать значение в вызвавшую среду. Предусматривается два вари­ анта метода — с параметрами и без параметров: // без параметров: static тип MainO { ... } static void MainO { ... } // с параметрами: static тип Main( string[] args ){/\*...\*/ } static void Main( string[] args ){/\*...\*/ } Параметры, разделяемые пробелами, задаются при запуске программы из ко­ мандной строки после имени исполняемого файла программы. Они передаются в массив args. ПРИМЕЧАНИ Е Имя параметра в программе может быть любым, но принято использовать имя args. Если метод возвращает значение, оно должно быть целого типа, если не возвра­ щает, он должен описываться как void. В этом случае оператор возврата из Main можно опускать, а вызвавшая среда автоматически получит нулевое значение, означающее успешное завершение. Ненулевое значение обычно означает аварий­ ное завершение, например: static int Main( string[] args ) { if ( .. . /\* все пропало \*/ ) return 1; if ( .. . /\* абсолютно все пропало \*/ ) return 100; } Возвращаемое значение анализируется в командном файле, из которого запускает­ ся программа. Обычно это делается для того, чтобы можно было принять решение, выполнять ли командный файл дальше. В листинге 7.2 приводится пример мето­ да Main, который выводит свои аргументы и ожидает нажатия любой клавиши. Листин г 7.2. Параметры метода Main using System; namespace ConsoleApplicationi { class Classl Индексаторы 157 { static void Main( string[] args ) { foreach( string arg in args ) Console.WriteLineC arg ); Console.ReadO; } } Пусть исполняемый файл программы имеет имя ConsoleApplication1.exe и вызы­ вается из командной строки: ; cs\ConsoleApplicationl\bin\Debug\ConsoleApplicationl.exe one two three Тогда на экран будет выведено: " л О Если параметр содержит специальные символы или пробелы, его заключают в ка- зычки. ПРИМЕЧАНИ Е Для запуска программы из командной строки можно воспользоваться, к примеру, ко­ мандой Выполнить меню Пуск или командой Пуск • Программы • Стандартные • Командная строка. Индексаторы Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках. Иными словами, индексатор — это такой «ум- ш й » индекс для объектов. Синтаксис индексатора аналогичен синтаксису свойства: грибуты спецификаторы тип this [ список\_параметров ] get коддоступа set коддоступа ВНИМАНИ Е В данном случае квадратные скобки являются элементом синтаксиса, а не указани­ ем на необязательность конструкции. [трибуты мы рассмотрим позже, в главе 12, а спецификаторы аналогичны пецификаторам свойств и методов. Индексаторы чаще всего объявляются со 158 Глава 7. Классы: подробности спецификатором public, поскольку они входят в интерфейс объекта1 . Атрибуты и спецификаторы могут отсутствовать. Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке значения (set) элемента массива. Может отсутст­ вовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения (read-only), если отсутствует часть get, индексатор доступен только для записи (write-only). Список параметров содержит одно или несколько описаний индексов, по кото­ рым выполняется доступ к элементу. Чаще всего используется один индекс це­ лого типа. Индексаторы в основном применяются для создания специализированных мас­ сивов, на работу с которыми накладываются какие-либо ограничения. В листин­ ге 7.3 создан класс-массив, элементы которого должны находиться в диапазоне [О, 100]. Кроме того, при доступе к элементу проверяется, не вышел ли индекс за допустимые границы. Листин г 7.3. Использование индексаторов using System; namespace ConsoleApplicationi { class SafeArray { public SafeArray( int size ) // конструктор класса a = new int[size]; length = size; public int Length // свойство - размерность get { return length; } public int this[int i] // индексатор get { if ( i >= 0 && i < length ) return a[i] ; else { error = true; return 0; } } set { if ( i >= 0 && i < length && value >= 0 && value <= 100 ) a[i] = value; 1 В версии C# 2.0 допускается раздельное указание спецификаторов доступа для блоков по­ лучения и установки индексатора, аналогично свойствам (раздел «Свойства», см. с. 120). 159 else error = true; l } public bool error = false; int[] a; int length; } class Classl { static void MainO { int n = 100; SafeArray sa = new SafeArray( n ); // создание объекта for ( int i = 0; i < n; ++i ) { sa[i] = i \* 2; III использование индексатора Console.WriteC sa[i] ); 112 использование индексатора } if ( sa.error ) Console.WriteC "Были ошибки!" ); 11з листинга видно, что индексаторы описываются аналогично свойствам. Благо­ даря применению индексаторов с объектом, заключающим в себе массив, можно работать так же, как с обычным массивом. Если обращение к объекту встречает­ ся в левой части оператора присваивания (оператор 1), автоматически вызывает­ ся метод get. Если обращение выполняется в составе выражения (оператор 2), вызывается метод set. В классе SafeArray принята следующая стратегия обработки ошибок: если при попытке записи элемента массива его индекс или значение заданы неверно, зна­ чение элементу не присваивается; если при попытке чтения элемента индекс не зходит в допустимый диапазон, возвращается 0; в обоих случаях формируется значение открытого поля error, равное true. ПРИМЕЧАНИ Е Проверка значения поля error может быть выполнена после какого-либо блока, предназначенного для работы с массивом, как в приведенном примере, или после каждого подозрительного действия с массивом (это может замедлить выполнение программы). Такой способ обработки ошибок не является единственно возмож­ ным. Правильнее обрабатывать подобные ошибки с помощью механизма исключе­ ний, соответствующий пример будет приведен далее в этой главе. Вообще говоря, индексатор не обязательно должен быть связан с каким-либо знутренним полем данных. В листинге 7.4 приведен пример класса Pow2, единст­ венное назначение которого — формировать степень числа 2. // открытый признак ошибки // закрытый массив // закрытая размерность 106 Глава 5. Классы: основные понятия Пол е у вывести на экран аналогичны м образо м не удастся: он о являетс я закры­ тым, т о есть недоступн о извне (и з класс а Classl). Поскольк у значение этом у пол ю явны м образо м не присвоено, среда присваивает ему значение ноль . ВНИМАНИ Е Все поля сначала автоматически инициализируются нулем соответствующего типа (например, полям типа int присваивается 0, а ссылкам на объекты — значение nu 11) . После этого полю присваивается значение, заданное при его явной инициа­ лизации. Задание начальных значений для статических полей выполняется при инициализации класса, а обычных — при создании экземпляра. Пол я со спецификатором readonly предназначены тольк о дл я чтения. Установит ь значение такого пол я можн о либ о при его описании, либ о в конструкторе (кон­ структоры рассматриваются дале е в этой главе). Методы Метод —• это функциональны й элемент класса, который реализуе т вычислени я > ил и другие действия, выполняемы е классо м ил и экземпляром . Метод ы опреде­ ляю т поведение класса. Мето д представляет собо й законченный фрагмент кода, к котором}' можн о обра­ титьс я но имени. О н описываетс я один раз, а вызыватьс я може т стольк о раз, скольк о необходимо . Оди н и тот же метод може т обрабатыват ь различные дан- ные, переданные ем у в качестве аргументов . ^ Синтаксис метода: £ [ атрибут ы ] [ спецификаторы ] ти п имя\_метода ( [ параметры ] ) тело метода ' •р Рассмотри м основны е элементы описания метода. Перва я строка представляе т собой заголовок метода. Тело метода, задающе е действия, выполняемы е методом, ^ чаще всего представляе т собой бло к — последовательност ь операторов в фигур- н ых скобках1 . П р и описании методо в можн о использоват ь спецификатор ы 1-7 из табл. 5.2, имеющи е тот же смысл , что и дл я полей , а также спецификаторы virtual, sealed, override, abstrac t и extern, которые буду т рассмотрены по мере необходимости. Чаш е всего дл я методо в задается спецификато р доступ а public, ведь метод ы составляю т интерфейс класса - то, с чем работает пользователь , поэтому они должн ы быт ь доступны. ВНИМАНИ Е Статические (static) методы, или методы класса, можно вызывать, не создавая эк­ земпляр объекта. Именно таким образом используется метод Main. Позже мы увидим, точка с запятой. что для методов некоторых типов вместо тела используется просто 160 Глава 7. Классы: подробней Листин г 7.4 . Индексатор без массива using System; namespace ConsoleApplicationi { class Pow2 { public ulong this[int i] { get { if ( i >= 0 ) { ulong res = 1; for ( int k = 0; k < i; k++ ) // цикл получения степени unchecked { res \*= 2; } III return res; } else return 0; } } } class Classl { static void MainO { int n = 13; Pow2 pow2 = new Pow2(); for ( int i = 0; i < n; ++i ) Console.WriteLineC "{0}\t{l} O i, pow2[i] ); Оператор 1 выполняется в непроверяемом контексте 1 , для того чтобы исклют ние, связанное с переполнением, не генерировалось. В принципе, данная програ ма работает и без этого, но если поместить класс Pow2 в проверяемый контек при значении, превышающем допустимый диапазон для типа ulong, возник? исключение. Результат работы программы: 0 1 1 2 2 4 3 8 4 16 5 32 6 64 Это понятие рассмотрено в разделе «Введение в исключения» (см. с. 46). Операции класса 161 128 : 256 г 512 Ю24 2048 12 4096 Язык С# допускает использование многомерных индексаторов. Они описывают­ ся аналогично обычным и применяются в основном для контроля за занесением данных в многомерные массивы и выборке данных из многомерных массивов, оформленных в виде классов. Например: •-t[,] а: Если внутри класса объявлен такой двумерный массив, то заголовок индексато­ ра должен иметь вид coli c int this[int i, int j] Операции класса С? позволяет переопределить действие большинства операций так, чтобы при использовании с объектами конкретного класса они выполняли заданные функ­ ции. Это дает возможность применять экземпляры собственных типов данных з составе выражений таким же образом, как стандартных, например: "..Object .а, Ь. с; : = а + Ь; // используется операция сложения для класса MyObject Определение собственных операций класса часто называют перегрузкой опера­ ций. Перегрузка обычно применяется для классов, описывающих математические или физические понятия, то есть таких классов, для которых семантика опера­ ций делает программу более понятной. Если назначение операции интуитивно не понятно с первого взгляда, перегружать такую операцию не рекомендуется. Операции класса описываются с помощью методов специального вида (функ­ ций-операций). Перегрузка операций похожа на перегрузку обычных методов. Синтаксис операции: Г атрибуты ] спецификаторы объявительоперации тело Атрибуты рассматриваются в главе 12, в качестве спецификаторов одновремен­ но используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю (extern). Объявитель операции содержит ключевое слово operator, по которому и опозна­ ется описание операции в классе. Тело операции определяет действия, которые выполняются при использовании операции в выражении. Тело представляет со­ бой блок, аналогичный телу других методов. 162 Глава 7. Классы: подробности ВНИМАНИ Е Новые обозначения для собственных операций вводить нельзя. Для операций класса сохраняются количество аргументов, приоритеты операций и правила ассоциации (справа налево или слева направо), используемые в стандартных типах данных. При описании операций необходимо соблюдать следующие правила: • операция должна быть описана как открытый статический метод класса (спе­ цификаторы public static); • параметры в операцию должны передаваться по значению (то есть не должны предваряться ключевыми словами ref или out); • сигнатуры всех операций класса должны различаться; • типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции). В С# существуют три вида операций класса: унарные, бинарные и операции пре­ образования типа. Унарные операции Можно определять в классе следующие унарные операции: + - ! - ++ -- true false Синтаксис объявителя унарной операции: тип operator унарнаяоперация ( параметр ) Примеры заголовков унарных операций: public static int operator +( MyObject m ) public static MyObject operator -- ( MyObject m ) public static bool operator true( MyObject m ) Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать: • для операций +, -, ! и - величину любого типа; • для операций ++ и - - величину типа класса, для которого она определяется; • для операций true и false величину типа bool. Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и пе­ редать его в качестве результата. ПРИМЕЧАНИ Е Префиксный и постфиксный инкременты не различаются (для них может сущест­ вовать только одна реализация, которая вызывается в обоих случаях). Операции класса 163 ПРИМЕЧАНИ Е Операции true и fal se обычно определяются для логических типов SQL, обладаю­ щих неопределенным состоянием, и не входят в число тем, рассматриваемых в этой книге. В качестве примера усовершенствуем приведенный в листинге 7.3 класс SafeArray для удобной и безопасной работы с массивом. В класс внесены следующие изме­ нения: • добавлен конструктор, позволяющий инициализировать массив обычным мас­ сивом или серией целочисленных значений произвольного размера; • добавлена операция инкремента; • добавлен вспомогательный метод Print вывода массива; • изменена стратегия обработки ошибок выхода за границы массива; • снято требование, чтобы элементы массива принимали значения в заданном диапазоне. Текст программы приведен в листинге 7.5. Листин г 7.5. Определение операции инкремента для класса SafeArray using System; namespace ConsoleApplicationi class SafeArray { public SafeArray( int size ) // конструктор a = new int[size]; length - size; public SafeArray( params int[] arr ) // новый конструктор length = arr.Length; a = new intDength]; for ( int i = 0; i < length; ++i ) a[i] = arr[i]; public static SafeArray operator ++( SafeArray x ) // ++ SafeArray temp = new SafeArray( x.length ): for ( int i = 0; i < x.length; ++i ) temp[i] = ++x.a[i]; return temp; public int this [int i] // индексатор get продолжение ^ 164 Глава 7. Классы: подробности Листин г 7.5 (продолжение) { if ( i >= 0 && i < length ) return a[i] : else throw new IndexOutOfRangeException(); // исключение } set { if ( i >= 0 && i < length ) a[i] = value; else throw new IndexOutOfRangeException(); // исключение public void Print( string name ) // вывод на экран { Console.WriteLine( name + ":" ); for ( int i = 0; i < length; ++i ) Console.Write( "\t" + a[i] ); Console.WriteLine(); } int[] a; // закрытый массив int length; // закрытая размерность class Classl { static void MainO { try { SafeArray al = new SafeArray( 5. 2, -1. 1, -2 ); al.Print("Массив 1" ); al++; al.Print( "Инкремент массива 1" ); } catch ( Exception e ) // обработка исключения { Console.WriteLine( e.Message ); } } } } Бинарные операции Можно определять в классе следующие бинарные операции: +-\*/%& « » = = !=><>=< = ВНИМАНИ Е Операций присваивания в этом списке нет. Операции класса 165 Синтаксис объявителя бинарной операции: тип operator бинарная\_операция (параметра параметр2) Примеры заголовков бинарных операций: Cjblic static MyObject operator + ( MyObject ml, MyObject m2 ) Cublic static bool operator == ( MyObject ml, MyObject m2 ) Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция может возвращать величину любого типа. Операции == и ! =, > и = и <= определяются только парами и обычно возвраща­ ют логическое значение. Чаще всего в классе определяют операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение объектов, а не их ссылок, как определено по умолчанию для ссылочных типов. Перегрузка опера­ ций отношения требует знания интерфейсов, поэтому она рассматривается поз­ же, в главе 9 (см. с. 203). Пример определения операции сложения для класса SafeArray, описанного в пре­ дыдущем разделе, приведен в листинге 7.6. В зависимости от операндов опера­ ция либо выполняет поэлементное сложение двух массивов, либо прибавляет значение операнда к каждому элементу массива. Листин г 7.6. Определение операции сложения для класса SafeArray using System; ramespace ConsoleApplicationl class SafeArray { public SafeArray( int size ) a = new int[size]; length = size; public SafeArray( params int[] arr ) length = arr.Length; a = new int[length]; for ( int i = 0; i < length: ++i ) a[i] = arr[i]; public static SafeArray operator + ( SafeArray x. SafeArray у ) // + int I en = x.length < y.length ? x.length : y.length; SafeArray temp = new SafeArray den); for ( int i = 0; i < Ten; ++i ) temp[i] = x[i] + y[i] : return temp; продолжение з £ " 166 Глава 7. Классы: подробности Листин г 7.6 (продолжение) public static SafeArray operator + ( SafeArray x, int у ) // + SafeArray temp = new SafeArrayCx.length); for ( int i = 0; i < x.length: ++i ) temp[i] = x[i] + y, return temp; public static SafeArray operator + ( int x, SafeArray у ) // + SafeArray temp = new SafeArray(y.length); for ( int i = 0; i < y.length; ++i ) temp[i] = x + y[i] ; return temp; public static SafeArray operator ++ ( SafeArray x ) // ++ SafeArray temp = new SafeArrayU.length); for ( int i = 0: i < x.length; ++i ) temp[i] = ++x.a[ij; return temp; public int this[int i] II [] get { if ( i >= 0 && i < length ) return a[i] : else throw new IndexOutOfRangeExceptionC); set { if ( i >= 0 && i < length ) a[i] = value; else throw new IndexOutOfRangeExceptionC); public void PrintC string name ) { Console.WriteLineC name + ":" ); for ( int i = 0; i < length; ++i ) Console.WriteC "\t" + a[i] Console.WriteLineC); int[] a; int length; // закрытый массив // закрытая размерность class Classl { static void MainO Операции класса 167 try SafeArray al = new SafeArrayC 5, 2, -1, 1, -2 ); al.Print( "Массив 1" ); SafeArray a2 = new SafeArray( 1, 0, 3 ); a2.Print( "Массив 2" ); SafeArray a3 = al + a2; a3.Print( "Сумма массивов 1 и 2" ); al = al + 100; // 1 al.Print( "Массив 1 + 100" ); al = 100 + al; // 2 al.PrintC "100 + массив 1" ); a2 += ++a2 +1 ; 113 оторвать руки! a2.Print( "++a2, a2 + a2 + 1" ); catch ( Exception e ) { Console.WriteLine( e.Message ); Результат работы программы: Массив 1: 5 2 -1 Насси i 2: 1 0 3 Сумма массивов 1 и 2: 7 3 3 Масси J 1 + 100: 106 103 100 100 + массив 1: 206 203 200 ++а2. а2 + а2 + 5 1: 3 9 102 99 202 199 Обратите внимание: чтобы обеспечить возможность сложения с константой, опе­ рация сложения перегружена два раза для случаев, когда константа является первым и вторым операндом (операторы 2 и 1). Сложную операцию присваивания += (оператор 3) определять не требуется, да это и невозможно. При ее выполнении автоматически вызываются сначала операция 168 Глава 7. Классы: подробности сложения, а потом присваивания. В целом же оператор 3 демонстрирует недопусти­ мую манеру программирования, поскольку результат его выполнения неочевиден. ПРИМЕЧАНИ Е — — В перегруженных методах для объектов применяется индексатор. Для повышения например: temp.a[i] = х + y.a[i]. Операции преобразования типа Операции преобразования типа обеспечивают возможность явного и неявног преобразования между пользовательскими типами данных. Синтаксис объявите ля операции преобразования типа: implicit operator тип ( параметр ) // неявное преобразование explicit operator тип ( параметр ) // явное преобразование Эти операции выполняют преобразование из типа параметра в тип, указанны в заголовке операции. Одним из этих типов должен быть класс, для которог определяется операция. Таким образом, операции выполняют преобразовани либо типа класса к другому типу, либо наоборот. Преобразуемые типы не дол » ны быть связаны отношениями наследования 1 . Примеры операций преобразовг ния типа для класса Monster, описанного в главе 5: public static implicit operator int( Monster m ) return m.health: } public static explicit operator Monster( int h ) { return new Monster( h. 100, "Fromlnt" ); Ниже приведены примеры использования этих преобразований в программе. Н надо искать в них смысл, они просто иллюстрируют синтаксис: Monster Masha = new Monster( 200, 200, "Masha" ); Неявное преобразование выполняется автоматически: • при присваивании объекта переменной целевого типа, как в примере; • при использовании объекта в выражении, содержащем переменные целевого тип; • при передаче объекта в метод на место параметра целевого типа; • при явном приведении типа. int i = Masha; Masha = (Monster) 500; // неявное преобразование // явное преобразование 1 Следовательно, нельзя определять преобразования к типу object и наоборот, впрочел они уже определены без нашего участия. Вложенные типы 169 Явное преобразование выполняется при использовании операции приведения типа. Все операции класса должны иметь разные сигнатуры. В отличие от других ви­ дов методов, для операций преобразования тип возвращаемого значения вклю­ чается в сигнатуру, иначе нельзя было бы определять варианты преобразования данного типа в несколько других. Ключевые слова implicit и explicit в сигнату­ ру не включаются, следовательно, для одного и того же преобразования нельзя определить одновременно явную и неявную версии. Неявное преобразование следует определять так, чтобы при его выполнении не возникала потеря точности и не генерировались исключения. Если эти ситуации возможны, преобразование следует описать как явное. Деструкторы В С# существует специальный вид метода, называемый деструктором. Он вызы­ вается сборщиком мусора непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие корректность последую­ щего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т. п.). Синтаксис деструктора: [ атрибуты ] [ extern ] -имяклассаО тело Как видно из определения, деструктор не имеет параметров, не возвращает зна­ чения и не требует указания спецификаторов доступа. Его имя совпадает с име­ нем класса и предваряется тильдой (~) , символизирующей обратные по отноше­ нию к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой, если деструктор определен как внешний (extern). Сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соот­ ветствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, не­ возможно гарантировать, что деструктор будет обязательно вызван в процессе работы программы. Следовательно, его лучше использовать только для гарантии освобождения ресурсов, а «штатное» освобождение выполнять в другом месте программы. Применение деструкторов замедляет процесс сборки мусора. Вложенные типы В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержа­ щим их классом. Механизм вложенных типов позволяет скрыть ненужные детали Методы 107 Пример простейшего метола: oubli c double Gety O // метод для получения поля у из лис г ин га { return у; } Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин «метод возвращает значение», поскольку после выполне­ ния метода происходит возврат в то место вызывающей функции, откуда был вызван метод, и передача туда значения выражения, записанного в операторе 'Stum (рис. 5.3). Если метод не возвращает никакого значения, в его заголовке задается тип void, а оператор return отсутствует. Вызывающая функция Вызов метода Метод return .. Возврат значения Рис. 5.3. Вызов метода Параметры используются для обмена информацией с метолом. Параметр пред­ ставляет собой локальную переменную, которая при вызове метода принима­ ет значение соответствующего аргумента. Область действия параметра - весь метод. Например, чтобы вычислить значение синуса для вещественной величины х, мы передаем ее в качестве аргумента в метод Sin класса Math, а чтобы вывести значе ние этой переменной на экран, мы передаем ее в метод WnteLioe класса Console ;cuble х = 0.1 : :;аЫе у - Math .Sin(x): 2i'isole.Wri teLine(x); При этом метод Sin возвращает в точку своего вызова вещественное значение си­ нуса, которое присваивается переменной у, а метод Wntel ine ничего не возвращает. ВНИМАНИ Е Метод, не возвращающий значение, вызывается отдельным оператором, а метол, возвращающий значение, — в составе выражения в правой части оператора при­ сваивания. Параметры, описываемые в заголовке метода, определяют множество значений аргументов, которые можно передавать в метод. Список аргументов при вызо­ ве как бы накладывается на список параметров, поэтому они должны попарно 170 Глава 7. Классы: подробности и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточ­ нения). Дл я вложенных типов можно использовать те же спецификаторы, что и для полей класса 1 . Например, введем в наш класс Monster вспомогательный класс Gun. Объекты этого класса без «хозяина» бесполезны, поэтому его можно определить как внут­ ренний: using System; namespace ConsoleApplicationi { class Monster { class Gun { } } } Помимо классов вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления. Мы рассмотрим их в главе 9. Рекомендации по программированию Как правило, класс как тип, определенный пользователем, должен содержать скрытые (private) поля и следующие функциональные элементы: • конструкторы, определяющие, как инициализируются объекты класса; • набор методов и свойств, реализующих характеристики класса; • классы исключений, используемые для сообщений об ошибках путем генера­ ции исключительных ситуаций. Классы, моделирующие математические или физические понятия, обычно так­ же содержат набор операций, позволяющих копировать, присваивать, срав­ нивать объекты и производить с ними другие действия, требующиеся по сути класса. Перегруженные операции класса должны иметь интуитивно понятный общепри­ нятый смысл (например, не следует заставлять операцию + выполнять что-либо, кроме сложения или добавления). Если какая-либо операция перегружена, сле­ дует перегрузить и аналогичные операции, например + и ++ (компилятор этого автоматически не сделает). При этом операции должны иметь ту же семантику, что и их стандартные аналоги. 1 Спецификаторы описаны в разделе «Данные: поля и константы» (см. с. 104). Рекомендации по программированию 171 В подавляющем большинстве классов для реализации действий с объектами класса предпочтительнее использовать не операции, а методы, поскольку им можно дать осмысленные имена. Перегруженные методы, в отличие от операций, применяются в классах повсе­ местно — как минимум, используется набор перегруженных конструкторов для создания объектов различными способами. Методы с переменным числом параметров реализуются менее эффективно, чем обычные, поэтому если, к примеру, требуется передавать в метод два, три или четыре параметра, возможно, окажется более эффективным реализовать не один метод с параметром params, а три перегруженных варианта с обычными пара­ метрами. Глава 8 Иерархии классов Управлять большим количеством разрозненных классов довольно сложно. С этой проблемой можно справиться путем упорядочивания и ранжирования классов, то есть объединяя общие для нескольких классов свойства в одном классе и ис­ пользуя его в качестве базового. Эту возможность предоставляет механизм наследования, который является мощ­ нейшим инструментом ООП . Он позволяет строить иерархии, в которых классы- потомки получают свойства классов-предков и могут дополнять их или изменять. Таким образом, наследование обеспечивает важную возможность многократного использования кода. Написав и огладив код базового класса, можно, не изменяя его, за счет наследования приспособить класс для работы в различных ситуаци­ ях. Это экономит время разработки и повышает надежность программ. Классы, расположенные ближе к началу иерархии, объединяют в себе общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных особенностей. Итак, наследование применяется для следующих взаимосвязанных целей: • исключения из программы повторяющихся фрагментов кода; • упрощения модификации программы; • упрощения создания новых программ на основе существующих. Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести из­ менения. Наследование Класс в С# может иметь произвольное количество потомков и только одного предка. При описании класса имя его предка записывается в заголовке класса Наследование 173 после двоеточия. Если имя предка не указано, предком считается базовый класс всей иерархии System.Object: [ атрибуты ] [ спецификаторы ] class имякласса [ : предки ] тело класса ПРИМЕЧАНИ Е Обратите внимание на то, что слово «предки» присутствует в описании класса во множественном числе, хотя класс может иметь только одного предка. Причина в том, что класс наряду с единственным предком может наследовать от интерфейсов — специального вида классов, не имеющих реализации. Интерфейсы рассматриваются в следующей главе. Рассмотрим наследование классов на примере. В разделе «Свойства» (см. с. 120) был описан класс Monster, моделирующий персонаж компьютерной игры. Допус­ тим, нам требуется ввести в игру еще один тип персонажей, который должен об­ ладать свойствами объекта Monster, а кроме того уметь думать. Будет логично сделать новый объект потомком объекта Monster (листинг 8.1). Листин г 8.1 . Класс Daemon, потомок класса Monster using System; namespace ConsoleApplicationi class Monster class Daemon Monster public Daemon О brain = 1: public Daemon( string name, int brain ) : base( name ) // 1 this.brain = brain; public Daemon( int health, int ammo, string name, int brain ) : base( health, ammo, name ) // 2 this.brain = brain; new public void PassportO // 3 продолжение & 174 Глава 8. Иерархии классов Листин г 8.1 (продолжение) { Console.WriteLineC "Daemon {0} \t health = {1} ammo = {2} brain = {3}". Name, Health, Ammo, brain ): } public void ThinkO // 4 { Console.Write( Name + " is" )•; for ( int i = 0; i < brain; ++i ) Console.WriteC " thinking" ); Console.WriteLine( ); } int brain; // закрытое поле } class Classl { static void MainO { Daemon Dima = new Daemon( "Dima", 3 ) Dima. Passport ()••; Dima.ThinkO; Dima.Health -= 10; Dima.PassportO; } } } В классе Daemon введены закрытое поле brain и метод Think, определены собствен­ ные конструкторы, а также переопределен метод Passport. Все поля и свойства класса Monster наследуются в классе Daemon1 . Результат работы программы: Daemon Dima health = 100 ammo = 100 brain = 3 Dima is thinking thinking thinking... Daemon Dima health = 90 ammo = 100 brain = 3 Как видите, экземпляр класса Daemon с одинаковой легкостью использует как собственные (операторы 5-7), так и унаследованные (оператор 8) элементы класса. Рассмотрим общие правила наследования, используя в качестве примера листинг 8.1. Конструкторы не наследуются, поэтому производный класс должен иметь соб­ ственные конструкторы. Порядок вызова конструкторов определяется приведен­ ными далее правилами: 1 Если бы в классе Monster были описаны методы, не переопределенные в Daemon, они бы также были унаследованы. // 5 // 6 // 7 // 8 Наследование 175 • Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров. Это правило использовано в первом из конструкторов класса Daemon. • Для иерархии, состоящей из нескольких уровней, конструкторы базовых клас­ сов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким об­ разом, каждый конструктор инициализирует свою часть объекта. • Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке ини­ циализации (это продемонстрировано в конструкторах, вызываемых в опера­ торах 1 и 2). Вызов выполняется с помощью ключевого слова base. Вызывает­ ся та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова base. Поля, методы и свойства класса наследуются, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать ком­ пилятору свое намерение с помощью ключевого слова new 1 . В листинге 8.1 таким образом переопределен метод вывода информации об объекте Passport. Другой способ переопределения методов рассматривается далее в разделе «Виртуальные методы». Метод Passport класса Daemon замещает соответствующий метод базового класса, однако возможность доступа к методу базового класса из метода производного класса сохраняется. Для этого перед вызовом метода указывается все то же вол­ шебное слово base, например: case.PassportO: СОВЕ Т Вызов одноименного метода предка из метода потомка всегда позволяет сохранить функции предка и дополнить их, не повторяя фрагмент кода. Помимо уменьшения объема программы это облегчает ее модификацию, поскольку изменения, внесен­ ные в метод предка, автоматически отражаются во всех его потомках. В конструкто­ рах метод предка вызывается после списка параметров и двоеточия, а в остальных методах — в любом месте с помощью приведенного синтаксиса. Вот, например, как выглядел бы метод Passport, если бы мы в классе Daemon хоте­ ли не полностью переопределить поведение его предка, а дополнить его: \*e»v public void PassportO base.PassportO; Console.WriteLineC " brain = {1}", brain ); Можно этого не делать, но тогда компилятор выдаст предупреждение. Предупреждение (warning) — это не ошибка, оно не препятствует успешной компиляции, но тем не менее... 176 Глава 8. Иерархии классов Элементы базового класса, определенные как private, в производном классе не­ доступны. Поэтому в методе Passport для доступа к полям name, health и ammo при­ шлось использовать соответствующие свойства базового класса. Другое решение заключается в том, чтобы определить эти поля со спецификатором protected, в этом случае они будут доступны методам всех классов, производных от Monster. Оба решения имеют свои достоинства и недостатки. ВНИМАНИ Е Важно понимать, что на этапе выполнения программы объект представляет собой единое целое, не разделенное на части предка и потомка. Во время выполнения программы объекты хранятся в отдельных переменных, массивах или других коллекциях 1 . Во многих случаях удобно оперировать объ­ ектами одной иерархии единообразно, то есть использовать один и тот же про­ граммный код для работы с экземплярами разных классов. Желательно иметь возможность описать: • объект, в который во время выполнения программы заносятся ссылки на объ­ екты разных классов иерархии; • контейнер, в котором хранятся объекты разных классов, относящиеся к од­ ной иерархии; • метод, в который могут передаваться объекты разных классов иерархии; • метод, из которого в зависимости от типа вызвавшего его объекта вызывают­ ся соответствующие методы. Все это возможно благодаря тому, что объекту базового класса можно присвоить объект производного класса2 . Давайте попробуем описать массив объектов базового класса и занести туда объ­ екты производного класса. В листинге 8.2 в массиве типа Monster хранятся два объекта типа Monster и один — типа Daemon. Листин г 8.2. Массив объектов разных типов using System; namespace .ConsoleApplicationl { class Monster { } class Daemon : Monster 1 Коллекция — объект, предназначенный для хранения данных и предоставляющий мето­ ды доступа к ним. Например, массив предоставляет прямой доступ к любому его элемен­ ту по индексу. Коллекции библиотеки .NET рассматриваются в главе 13. 2 Еще раз напомню, что объекты относятся к ссылочному типу, следовательно, присваива­ ется не сам объект, а ссылка на него. Наследование 177 { . . . //см. листинг 8.1 } class Classl { static void MainO { const int n = 3; Monster[] stado = new Monsterln]; stado[0] = new Monster( "Monia" ); stado[l] = new Monster( "Monk" ); stado[2] = new Daemon ( "Dimon", 3 ); foreach ( Monster elem in stado ) elem.PassportO; // 1 for ( int i = 0; i < n; ++i ) stado[i].Ammo =0 ; 111 Console.WriteLineC); foreach ( Monster elem in stado ) el em. PassportO; // 3 } } } Результат работы программы: Monster Monia health = 100 ammo = 100 Monster Monk health = 100 ammo = 100 Monster Dimon health - 100 ammo = 100 Monster Monia health = 100 ammo = 0 Monster Monk health = 100 ammo = 0 Monster Dimon health = 100 ammo = 0 Результат радует нас только частично: объект типа Daemon действительно можно поместить в массив, состоящий из элементов типа Monster, но для него вызыва­ ются только методы и свойства, унаследованные от предка. Это устраивает нас в операторе 2, а в операторах 1 и 3 хотелось бы, чтобы вызывался метод Passport, переопределенный в потомке. Итак, присваивать объекту базового класса объект производного класса можно, но вызываются для него только методы и свойства, определенные в базовом классе. Иными словами, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает. Это и понятно: ведь компилятор должен еще до выполнения программы решить, какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод (этот процесс называется ранним связыванием). При этом компилятор может руководствоваться только типом переменной, для которой вызывается ме­ тод или свойство (например, stadoli ] .Ammo). То, что в этой переменной в разные 178 Глава 8. Иерархии классов моменты времени могут находиться ссылки на объекты разных типов, компиля­ тор учесть не может. Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения про­ граммы, а точнее — до момента вызова метода, когда уже точно известно, на объ­ ект какого типа указывает ссылка. Такой механизм в С# есть — он называется поздним связыванием и реализуется с помощью так называемых виртуальных методов, которые мы незамедлительно и рассмотрим. Виртуальные методы При раннем связывании программа, готовая для выполнения, представляет со­ бой структуру, логика выполнения которой жестко определена. Если же тре­ буется, чтобы решение о том, какой из одноименных методов разных объектов иерархии использовать, принималось в зависимости от конкретного об'ьекта, для которого выполняется вызов, то заранее жестко связывать эти методы с осталь­ ной частью кода нельзя. Следовательно, надо каким-то образом дать знать компилятору, что эти методы будут обрабатываться по-другому. Для этого в С# существует ключевое слово virtual. Оно записывается в заголовке метода базового класса, например: virtual public void Passport!) .. . Слово virtual в переводе с английского значит «фактический». Объявление ме­ тода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения про­ граммы. Этот механизм называется поздним связыванием. Д ля его реализации необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет в любой момент воспользоваться, поэтому компилятор формирует для этих методов таблицу виртуальных методов (Virtual Method Table. VMT) . В нее записываются адреса виртуальных методов (в том числе унаследо­ ванных) в порядке описания в классе. Для каждого класса создается одна таблица. Каждый объект во время выполнения должен иметь доступ к VMT . Обеспечение этой связи нельзя поручить компилятору, так как она должна устанавливаться во время выполнения программы при создании объекта. Поэтому связь экземп­ ляра объекта с VM T устанавливается с помощью специального кода, автомати­ чески помещаемого компилятором в конструктор объекта. Если в производном классе требуется переопределить виртуальный метод, ис­ пользуется ключевое слово override, например: override public void PassportO .. . Переопределенный виртуальный метод должен обладать таким же набором па­ раметров, как и одноименный метод базового кчасса. Это требование вполне ес­ тественно, если учесть, что одноименные методы, относящиеся к разным клас­ сам, могут вызываться из одной и той же точки программы. Виртуальные методы 179 Добавим в листинг 8.2 два волшебных слова — virtual и override — в описания методов Passport, соответственно, базового и производного классов (листинг 8.3). Листин г 8.3. Виртуальные методы using System; namespace ConsoleApplicationi class Monster { virtual public void PassportO { Console.WriteLineC "Monster {0} \t health = {1} ammo = {2}", name, health, ammo ); } } class Daemon ; Monster { override public void PassportO { Console.WriteLineC "Daemon {0} \t health = {1} ammo = {2} brain = {3}", Name. Health, Ammo, brain ); } } class Classl { static void MainO { const int n = 3; Monster[] stado = new Monster[n]; stado[0] = new MonsterC "Monia" ); stado[l] = new MonsterC "Monk" ): stado[2] = new Daemon ( "Dimon", 3 ); foreach ( Monster elem in stado ) elem.PassportO; for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0; Console.WriteLineC): foreach ( Monster elem in stado ) elem.PassportO; } } } 108 соответствовать друг другу. Правила соответствия подробно рассматриваются в следующих разделах. Д ля каждого параметра должны задаваться его тип и имя. Например, заголовок метода Sin выглядит следующим образом: publi c stati c double Sin ( double a ): Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой сигнатуру метода — то, по чему один метод отличают от других. В классе не должно быть методов с одинаковыми сигнатурами. В листинге 5.2 в класс Demo добавлены методы установки и получения значения поля у (на самом деле для подобных целей используются не методы, а свойства, которые рассматриваются чуть позже). Кроме того, статическое поле s закрыто, то есть определено по умолчанию как private , а для его получения описан метод Gets, являющий собою пример статического метода. Листин г 5.2. Простейшие методы usin g System; namespace ConsoleApplication l { clas s Demo i pub'i c in t э=1 ; publi c cons t double с - 1.66; stati c strin g s = "Demo"; double y ; publi c double Gety O // метод получения поля у { retur n у ; } publi c voi d SetyC double y\_ ) // метод установки поля у { У = У\_: } publi c stati c strin g Gets О // метод получения поля s { retur n s ; clas s Class l { stati c voi d MainO { Demo x = new Demo(); x.Sety(0.12) : // вызов метода установки поля у 180 Глава 8. Иерархии классов Результат работы программы: Monster Monia health = 100 ammo = 100 Monster Monk health = 100 ammo = 100 Daemon Dimon health = 100 ammo = 100 brain = 3 Monster Monia health = 100 ammo = 0 Monster Monk health = 100 ammo = 0 Daemon Dimon health = 100 ammo = 0 brain = 3 Как видите, теперь в циклах 1 и 3 вызывается метод Passport, соответствующий типу объекта, помещенного в массив. Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых вир­ туальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод на­ следуется. Вызов виртуального метода выполняется так: из объекта берется адрес его таб­ лицы VMT , из VM T выбирается адрес метода, а затем управление передается этому методу. Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта. ПРИМЕЧАНИ Е Вызов виртуального метода, в отличие от обычного, выполняется через дополни­ тельный этап получения адреса метода из таблицы VMT , что несколько замедляет выполнение программы. С помощью виртуальных методов реализуется один из основных принципов объ­ ектно-ориентированного программирования — полиморфизм. Это слово в пере­ воде с греческого означает «много форм», что в данном случае означает «один вызов — много методов». Применение виртуальных методов обеспечивает гиб­ кость и возможность расширения функциональности класса. Виртуальные методы незаменимы и при передаче объектов в методы в качестве параметров. В параметрах метода описывается объект базового типа, а при вызо­ ве в нее передается объект производного класса. В этом случае виртуальные ме­ тоды, вызываемые для объекта из метода, будут соответствовать типу аргумента, а не параметра. При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому. Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод. ПРИМЕЧАНИ Е Все сказанное о виртуальных методах относится также к свойствам и индексаторам. Абстрактные классы 181 Абстрактные классы При создании иерархии объектов для исключения повторяющегося кода часто быва­ ет логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что ни­ какие реальные объекты им не соответствуют. Такие классы называют абстрактными. Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по- своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором abstract. ПРИМЕЧАНИ Е Абстрактный класс может содержать и полностью определенные методы, в отличие от сходного с ним по предназначению специального вида класса, называемого ин­ терфейсом. Интерфейсы рассматриваются в следующей главе. Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например: abstract class Spirit г i public abstract void PassportO: } class Monster : Spirit { override public void PassportO { Console.Writel\_ine( "Monster {0} \t health = {1} ammo = {2}", name, health, ammo ); } } class Daemon : Monster { override public void PassportO { Console. Writel\_ine( "Daemon {0} \t health = {1} ammo = {2} brain = {3}", Name, Health, Ammo, brain ); } . . . // полный текст этих классов приведен в главе 12 182 Глава 8. Иерархии классов Абстрактные классы используются при работе со структурами данных, предна­ значенными для хранения объектов одной иерархии, и в качестве параметров методов. Если класс, производный от абстрактного, не переопределяет все абст­ рактные методы, он также должен описываться как абстрактный. Можно создать метод, параметром которого является абстрактный класс. На место этого параметра при выполнении программы может передаваться объект любого производного класса. Это позволяет создавать полиморфные методы, работаю­ щие с объектом любого типа в пределах одной иерархии. Полиморфизм в раз­ личных формах является мощным и широко применяемым инструментом ООП . ПРИМЕЧАНИ Е Мы уже использовали полиморфизм в разделе «Оператор foreach» (см. с. 136) для того, чтобы метод PrintArra y мог работать с массивом любого типа. Еще один при­ мер применения абстрактных и виртуальных методов имеется в главе 10. Бесплодные классы В С# есть ключевое слово sealed, позволяющее описать класс, от которого, в про­ тивоположность абстрактному, наследовать запрещается: sealed clas s Spiri t i } // clas s Monster : Spiri t { .. . } ошибка! Большинство встроенных типов данных описано как sealed. Если необходимо ис­ пользовать функциональность бесплодного класса, применяется не наследование, а вложение, или включение: в классе описывается поле соответствующего типа. Вложение классов, когда один класс включает в себя поля, являющиеся класса­ ми, является альтернативой наследованию при проектировании. Например, если есть объект «двигатель», а требуется описать объект «самолет», логично сделать двигатель полем этого объекта, а не его предком. Поскольку поля класса обычно закрыты, возникает вопрос, как же пользоваться методами включенного объекта. Общепринятый способ состоит в том, чтобы описать метод объемлющего класса, из которого вызвать метод включенного класса. Такой способ взаимоотношений классов известен как модель включения- делегирования. Пример приведен в листинге 8.4. Листин г 8.4. Модель включения-делегирования usin g System; namespace ConsoleApplication i { clas s Двигатель { publi c voi d Запуске) Класс object 183 { Console.WriteLine( "вжжжж!!" ); } } class Самолет { public Самолет() { левый = new ДвигательО ; правый = new ДвигательО; } public void Запустить\_двигатели() { левый.Запуск(); правый.Запуске); } Двигатель левый, правый; } class Classl { static void MainO { Самолет AH24\_1 = new Самолете); АН24\_1.3апустить\_двигатели(); } } Результат работы программы: зжжжж!! зжжжж!! В методе Запустить\_двигатели запрос на запуск двигателей передается, или, как принято говорить, делегируется вложенному классу. В отличие от наследования, когда производный класс «является» (is а) разно­ видностью базового, модель включения-делегирования реализует отношение «имеет» (has а). При проектировании классов следует выбирать модель, наибо­ лее точно отражающую смысл взаимоотношений классов, например, моделируе­ мых объектов предметной области. Класс object Корневой класс System.Object всей иерархии объектов .NET, называемый в С# object, обеспечивает всех наследников несколькими важными методами. Про­ изводные классы могут использовать эти методы непосредственно или пере­ определять их. 184 Глава 8. Иерархии классов Класс object часто используется и непосредственно при описании типа парамет­ ров методов для придания им общности, а также для хранения ссылок на объек­ ты различного типа — таким образом реализуется полиморфизм. Открытые методы класса System.Object перечислены ниже. • Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Син­ таксис: public virtual bool Equals( object obj ); • Метод Equals с двумя параметрами возвращает значение true, если оба пара­ метра ссылаются на одну и ту же область памяти. Синтаксис: public static bool Equals( object obi, object ob2 ); • Метод GetHashCode формирует хеш-код объекта и возвращает число, однознач­ но идентифицирующее объект. Это число используется в различных структу­ рах и алгоритмах библиотеки. Если переопределяется метод Equals, необходи­ мо перегрузить и метод GetHashCode. Подробнее о хеш-кодах рассказывается в разделе «Абстрактные структуры данных» (см. с. 291). Синтаксис: public virtual int GetHashCodeO; • Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвра­ щаемое значение имеет тип Туре. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения 1 . Синтаксис: public Type Get Туре О; • Метод ReferenceEquals возвращает значение true, если оба параметра ссыла­ ются на одну и ту же область памяти. Синтаксис: public static bool( object obi, object ob2 ); • Метод ToString по умолчанию возвращает для ссылочных типов полное имя класса в виде строки, а для значимых — значение величины, преобразованное в строку. Этот метод переопределяют для того, чтобы можно было выводить информацию о состоянии объекта. Синтаксис: public virtual string ToStringO В производных объектах эти методы часто переопределяют. Например, можно j переопределить метод Equals для того, чтобы задать собственные критерии срав- ! нения объектов, потому что часто бывает удобнее использовать для сравнения не j ссылочную семантику (равенство ссылок), а значимую (равенство значений). Пример применения и переопределения методов класса object для класса Monster \* приведен в листинге 8.5. 1 Мы рассмотрим этот тип в главе 12 (см. с. 279). Класс object 185 Листин г 8.5. Перегрузка методов класса object using System; namespace ConsoleApplicationl class Monster { public Monster( int health, int ammo, string name ) { this.health = health; this.ammo = ammo; this.name = name; } public override bool Equals( object obj ) { if ( obj == null || GetTypeO != obj.GetTypeC) ) return false; Monster temp = (Monster) obj; return health == temp.health && ammo == temp.ammo && name == temp.name; } public override int GetHashCode() { return name.GetHashCode(); } public override string ToStringO { return string.Format( "Monster {0} \t health = {1} ammo = {2}" , name, health, ammo ); } string name; int health, ammo; } class Classl { static void Main() { Monster X = new Monster( 80, 80, "Вася" ); Monster Y = new Monster( 80, 80, "Вася" ); Monster Z = X; if ( X == Y ) Console.WriteLine( X == Y ") ; else Console.WriteLine( X != Y ") ; продолжение & 186 Глава 8. Иерархии классов Листин г 8.5 (продолжение) if ( X == Z ) Console.WriteLineC X == Z ") ; else Console.WriteLineC X != Z ") ; if ( X.Equals(Y) ) Console.WriteLineC " X Equals Y " ); else Console.WriteLineC " X not Equals Y " ); Console.Wri teLi ne(X.GetTypeC)); } } } Результат работы программы: X != Y X « \* Z X Equals Y ConsoleAppli cati onl.Monster В методе Equals сначала проверяется переданный в него аргумент. Если он равен nul 1 или его тип не соответствует типу объекта, вызвавшего метод, возвращается значение false. Значение true формируется в случае попарного равенства всех полей объектов. Метод GetHashCode просто делегирует свои функции соответствующему методу одного из полей. Метод ToString формирует форматированную строку, содержа­ щую значения полей объекта. Анализируя результат работы программы, можно увидеть, что в операции срав­ нения на равенство сравниваваются ссылки, а в перегруженном методе Equals — значения. Для концептуального единства можно переопределить и операции от­ ношения, соответствующий пример приведен в разделе «Перегрузка операций отношения» (см. с. 203). Рекомендации по программированию Наследование классов предоставляет программисту богатейшие возможности организации кода и его многократного использования. Выбор наиболее подходя­ щих средств для целей конкретного проекта основывается на знании механизма их работы и взаимодействия. Наследование класса Y от класса X означает, что Y представляет собой разновид­ ность класса X, то есть более конкретную, частную концепцию. Базовый класс 1 является более общим понятием, чем Y1 . Везде, где можно использовать X, мож­ но использовать и Y, но не наоборот (вспомните, что на место базового класса можно передавать любой из производных). Необходимо помнить, что во время 1 Например, каждый программист — человек, но не каждый человек — программист. 187 выполнения программы не существует иерархии классов и передачи сообщений объектам базового класса из производных — есть только конкретные объекты классов, поля которых формируются на основе иерархии на этапе компиляции. Главное преимущество наследования состоит в том, что на уровне базового класса можно написать универсальный код, с помощью которого работать также с объ­ ектами производного класса, что реализуется с помощью виртуальных методов. Как виртуальные должны быть описаны методы, которые выполняют во всех классах иерархии одну и ту же функцию, но, возможно, разными способами. Пусть, например, все объекты иерархии должны уметь выводить информацию о себе. Поскольку эта информация хранится в различных полях производных классов, функцию вывода нельзя реализовать в базовом классе. Естественно на­ звать ее во всех классах одинаково и объявить как виртуальную с тем, чтобы ее можно было вызывать в зависимости от фактического типа объекта, с которым работают через базовый класс. Д ля представления общих понятий, которые предполагается конкретизировать в производных классах, используют абстрактные классы. Как правило, в абст­ рактном классе задается набор методов, то есть интерфейс, который каждый из потомков будет реализовывать по-своему. Обычные (не виртуальные) методы переопределять в производных классах не рекомендуется, поскольку производные классы должны наследовать свойства базовых, а спецификатор new, с помощью которого переопределяется обычный метод, «разрывает» отношение наследования на уровне метода. Иными словами, невиртуальный метод должен быть инвариантен относительно специализации, то есть должен сохранять свойства, унаследованные из базового класса незави­ симо от того, как конкретизируется (специализируется) производный класс. Специализация производного класса достигается добавлением новых методов и переопределением существующих виртуальных методов. Альтернативным наследованию механизмом использования одним классом друго­ го является вложение, когда один класс является полем другого. Вложение пред­ ставляет отношения классов «Y содержит X» или «Y реализуется посредством X». Для выбора между наследованием и вложением служит ответ на вопрос о том, может ли у Y быть несколько объектов класса X («Y содержит X»). Кроме того, вложение используется вместо наследования тогда, когда про классы X и Y нельзя сказать, что Y является разновидностью X, но при этом Y использует часть функ­ циональности X («Y реализуется посредством X»). Глава 9 Интерфейсы и структурные типы В этой главе рассматриваются специальные виды классов — интерфейсы, струк­ туры и перечисления. Синтаксис интерфейса Интерфейс является «крайним случаем» абстрактного класса. В нем задается набор абстрактных методов, свойств и индексаторов, которые должны быть реализова­ ны в производных классах 1 . Иными словами, интерфейс определяет поведение, которое поддерживается реализующими этот интерфейс классами. Основная идея использования интерфейса состоит в том, чтобы к объектам таких классов можно было обращаться одинаковым образом. Каждый класс может определять элементы интерфейса по-своему. Так достига­ ется полиморфизм: объекты разных классов по-разному реагируют на вызовы одного и того же метода. Синтаксис интерфейса аналогичен синтаксису класса: [ атрибуты ] [ спецификаторы ] interface имя\_интерфейса [ : предки ] телоинтерфейса [ ; ] Д ля интерфейса могут быть указаны спецификаторы, new, publ i с, protected, internal и private. Спецификатор new применяется для вложенных интерфейсов и имеет такой же смысл, как и соответствующий модификатор метода класса. Остальные спецификаторы управляют видимостью интерфейса. В разных контекстах опре­ деления интерфейса допускаются разные спецификаторы. По умолчанию интер­ фейс доступен только из сборки, в которой он описан (internal). 1 Кроме того, в интерфейсе можно описывать события, которые мы рассмотрим в главе 10. Синтаксис интерфейса 189 Интерфейс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую. Тело интерфейса составляют абстрактные методы, шаблоны свойств и индексаторов, а также события. ПРИМЕЧАНИ Е Методом исключения можно догадаться, что интерфейс не может содержать кон­ станты, поля, операции, конструкторы, деструкторы, типы и любые статические элементы. В качестве примера рассмотрим интерфейс I Act ion, определяющий базовое пове­ дение персонажей компьютерной игры, встречавшихся в предыдущих главах. Допустим, что любой персонаж должен уметь выводить себя на экран, атаковать и красиво умирать: interface IAction { void Draw(); int Attack(int a); void Die(); int Power { get; } } В интерфейсе IAction заданы заголовки трех методов и шаблон свойства Power, доступного только для чтения. Как легко догадаться, если бы требовалось обес­ печить еще и возможность установки свойства, в шаблоне следовало указать ключевое слово set, например: int Power { get; set; } В интерфейсе имеет смысл задавать заголовки тех методов и свойств, которые будут по-разному реализованы различными классами разных иерархий. ВНИМАНИ Е Если некий набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии. То, что работа­ ет в пределах иерархии одинаково, предпочтительно полностью определить в базо­ вом классе (примерами таких действий являются свойства Health, Ammo и Name из иерархии персонажей игры). Интерфейсы же чаще используются для задания общих свойств объектов различных иерархий. Отличия интерфейса от абстрактного класса: • элементы интерфейса по умолчанию имеют спецификатор доступа publ i с и не могут иметь спецификаторов, заданных явным образом; • интерфейс не может содержать полей и обычных методов — все элементы ин­ терфейса должны быть абстрактными; Методы 109 Console.WriteLineC x.Gety O ); // вызов метода получения поля у Console.WriteLine t Demo.Gets С) ); // вызов метода получения поля s // Console.WriteLine t Gets O ); // вариант вызова } } } Как видите, методы класса имеют непосредственный доступ к его закрытым полям. Метод, описанный со спецификатором static , должен обращаться только к статическим полям класса. Обратите внимание на то, что статический метод вызывается через имя класса, а обычный — через имя экземпляра. ПРИМЕЧАНИ Е При вызове метода из другого метода того же класса имя класса/экземпляра можно не указывать. Параметры методов Рассмотрим более подробно, каким образом метод обменивается информацией с вызвавшим его кодом. При вызове метода выполняются следующие действия: 1. Вычисляются выражения, стоящие на месте аргументов. 2. Выделяется память под параметры метода в соответствии с их типом. 3. Каждому из параметров сопоставляется соответствующий аргумент (аргумен­ ты как бы накладываются на параметры и замещают их). 4. Выполняется тело метода. 5. Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип void , управление передается на оператор, следующий после вызова. При этом проверяется соответствие типов аргументов и параметров и при необ­ ходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение. Листинг 5.3 иллюстрирует этот процесс. Листин г 5.3. Передача параметров методу .sin g System; -amespace ConsoleApplication i clas s Class l { stati c in t MaxCint a, in t b) / / метод выбора максимального значения { if ( а > b ) retur n а; els e retur n b; } stati c voi d MainO { i n t a = 2, b = 4 : i n t x = Max( a, b ); // вызов метода Max Console.WriteLineC x ); // результат : 4 продолжение 190 Глава 9. Интерфейсы и структурные типы • класс, в списке предков которого задается интерфейс, должен определять все его элементы, в то время как потомок абстрактного класса может не переоп­ ределять часть абстрактных методов предка (в этом случае производный класс также будет абстрактным); • класс может иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы. ПРИМЕЧАНИ Е В библиотеке .NET определено большое количество стандартных интерфейсов, ко­ торые описывают поведение объектов разных классов. Например, если требуется сравнивать объекты по принципу больше или меньше, соответствующие классы должны реализовать интерфейс IComparable. Мы рассмотрим наиболее употреби­ тельные стандартные интерфейсы в последующих разделах этой главы. Реализация интерфейса В списке предков класса сначала указывается его базовый класс, если он есть, а затем через запятую — интерфейсы, которые реализует этот класс. Таким обра­ зом, в С# поддерживается одиночное наследование для классов и множествен­ ное — для интерфейсов. Это позволяет придать производному классу свойства нескольких базовых интерфейсов, реализуя их по своему усмотрению. Например, реализация интерфейса IAction в классе Monster может выглядеть сле­ дующим образом: using System: namespace ConsoleAppl icationl { interface IAction { void DrawO; int Attack( int a ): void DieO: int Power {get ; } } class Monster : IAction { public void DrawO { Console.WriteLineC "Здесь был " + name ): } public int Attack( int ammo\_ ) { ammo -= ammo\_; if ( ammo > 0 ) Console.WriteLineC "Ба-бах!" ); else ammo = 0: 191 retur n зптао; } publi c voi d Oie O { Console.WriteLineC "Monster " + name + " RIP" ); healt h = 0: } publi c in t Power I ciet { retur n ammo \* health ; } Естественно , чт о сигнатур ы методо в в интерфейс е и реализаци и должн ы полно ­ сть ю совпадать . Дл я реализуемы х элементо в интерфейс а в класс е следуе т указы ­ ват ь спецификато р publ ic . К эти м элемента м можн о обращатьс я ка к чере з объек т класса , та к и чере з объек т тип а соответствующег о интерфейса 1 : Monster Vasia = new MonsterC 50. 50, "Вася" ); // объект класса Monster Vasia.DrawC): // результат : Здесь был Вася IArtio n Acto r = new MonsterC 10, 10, "Маша" ); // объект типа интерфейса Actor.DrawC); // результат : Здесь был Маша Удобств о второг о способ а проявляетс я пр и присваивани и объекта м тип а I Actio n ссыло к н а объект ы различны х классов , поддерживающи х это т интерфейс . На ­ пример , легк о себе представит ь мето д с параметро м тип а интерфейса . Н а мест о этог о параметр а можн о передават ь любо й объект , реализующи й интерфейс : stati c voi d Act С IActio n A ) { A.DrawC); ) stati c voi d Maine) { Monster Vasia = new MonsterC 50, 50, "Вася" ); ActC Vasia ); 1 Этот объект должен содержать ссылку на класс, поддерживающий интерфейс. Естест­ венно, что объекты типа интерфейса, так же как и объекты абстрактных классов, созда­ вать нельзя. 192 Глава 9. Интерфейсы и структурные типы Существует второй способ реализации интерфейса в классе: явное указание име­ ни интерфейса перед реализуемым элементом. Спецификаторы доступа при этом не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейса, например: class Monster • IAction { int IAction.Power { get { return ammo \* health; } } void IAction.DrawO { Console.WriteLineC "Здесь был " + name ); } } IAction Actor = new Monster( 10, 10, "Mama" ); Actor.Draw(); // обращение через объект типа интерфейса // Monster Vasia = new MonsterC 50, 50, "Вася" ); // Vasia.DrawO; ошибка! Таким образом, при явном задании имени реализуемого интерфейса соответст­ вующий метод не входит в интерфейс класса. Это позволяет упростить его в том случае, если какие-то элементы интерфейса не требуются конечному пользовате­ лю класса. Кроме того, явное задание имени реализуемого интерфейса перед именем метода позволяет избежать конфликтов при множественном наследовании, если эле­ менты с одинаковыми именами или сигнатурой встречаются более чем в одном интерфейсе1 . Пусть, например, класс Monster поддерживает два интерфейса: один для управления объектами, а другой для тестирования: interface ITest { void DrawO; } interface IAction 1 Методы с одинаковыми именами, но с различными сигнатурами к конфликту не приво­ дят, они просто считаются перегруженными. Реализация интерфейса 193 void Draw(); int Attack( int a ); void Die(); int Power { get; } lass Monster : IAction, ITest void ITest.DrawO { Console.Writel\_ine( "Testing " + name ); void IAction.DrawO { Console.WriteLineC "Здесь был " + name ); Оба интерфейса содержат метод Draw с одной и той же сигнатурой. Различать их помогает явное указание имени интерфейса. Обращаться к этим методам можно, используя операцию приведения типа, например: Monster Vasia = new MonsterC 50, 50, "Вася" ); :(ITest)Vasia).DrawO; // результат: Здесь был Вася ;(IAction)Vasia).DrawO; //результат: Testing Вася Впрочем, если от таких методов не требуется разное поведение, можно реали­ зовать метод первым способом (со спецификатором public), компилятор не воз­ ражает: slass Monster : IAction. ITest public void DrawO { Console.WriteLineC "Здесь был " + name ): К методу Draw, описанному таким образом, можно обращаться любым способом: через объект класса Monster, через интерфейс IAction или ITest. Конфликт возникает в том случае, если компилятор не может определить из контекста обращения к элементу, элемент какого именно из реализуемых ин­ терфейсов требуется вызвать. При этом всегда помогает явное задание имени интерфейса. 1 94 Глава 9. Интерфейсы и структурные типы Работа с объектами через интерфейсы. Операции is и as При работе с объектом через объект типа интерфейса бывает необходимо убедить­ ся, что объект поддерживает данный интерфейс. Проверка выполняется с помо­ щью бинарной операции is. Эта операция определяет, совместим ли текущий тип объекта, находящегося слева от ключевого слова is, с типом, заданным справа. Результат операции равен true, если объект можно преобразовать к заданному типу, и false в противном случае. Операция обычно используется в следующем контексте: 1f ( объект is тип ) { // выполнить преобразование "объекта" к "типу" /'/ выполнить действия с преобразованным объектом Допустим, мы оформили какие-то действия с объектами в виде метода с пара­ метром типа object. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Дл я безопасного преобразования следу­ ет проверить, возможно ли оно, например так: static vo'd Act( object A ) { if ( A is IAction ) { IAction Actor = (IAction) A; Actor.Draw(); 1 } В метод Act можно передавать любые объекты, но на экран будут выведены толь­ ко те, которые поддерживают интерфейс IAction. Недостатком использования операции is является то, что преобразование фак­ тически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — as. Она выполняет преобра­ зование к заданному типу, а если это невозможно, формирует результат null, например: static void Act( object A ) I IAction Actor = A as IAction; if ( Actor != null ) Actor.Draw(); Обе рассмотренные операции применяются как к интерфейсам, так и к классам. Интерфейсы и наследование 195 Интерфейсы и наследование Интерфейс может не иметь или иметь сколько угодно интерфейсов-предков, в по­ следнем случае он наследует все элементы всех своих базовых интерфейсов, начиная с самого верхнего уровня. Базовые интерфейсы должны быть доступны в не меньшей степени, чем их потомки. Например, нельзя использовать интер­ фейс, описанный со спецификатором private или internal, в качестве базового для открытого (publ ic ) интерфейса'. Как и в обычной иерархии классов, базовые интерфейсы определяют общее по­ ведение, а их потомки конкретизируют и дополняют его. В интерфейсе-потомке можно также указать элементы, переопределяющие унаследованные элементы с такой же сигнатурой. В этом случае перед элементом указывается ключевое слово new, как и в аналогичной ситуации в классах. С помощью этого слова со­ ответствующий элемент базового интерфейса скрывается. Вот пример из до­ кументации С#: interface 1 Base i void F( int i ): } interface I left : IBase { new void F( int i ); // переопределение метода F interface Inght ; IBase R \ void G(): interface Iderived : I Left, I Right {} class A void Test( IDerived d ) { d.F( 1 ); // Вызывается I Left F ((IBase)d).F( 1 ): // Вызывается IBase.F ((ILeft )d). F( 1 ); //' Вызывается I Left. F v(IRight)d).F( 1 ); /7 Вызывается IBase.F Метод F из интерфейса IBase скрыт интерфейсом ILeft, несмотря на то что в це­ почке IDerived — IRight — IBase он не переопределялся. Класс, реализующий интерфейс, должен определять все его элементы, в том числе унаследованные. Если при этом явно указывается имя интерфейса, оно Естественно, интерфейс не может быть наследником самого себя. 196 Глава 9. Интерфейсы и структурные типы должно ссылаться на тот интерфейс, в котором был описан соответствующий элемент, например: class А : IRight { IRight.GO { .. . } IBase.F( int i ) { .. . } // IRight.F( int i ) - нельзя } Интерфейс, на собственные или унаследованные элементы которого имеется яв­ ная ссылка, должен быть указан в списке предков класса, например: class В : А { // IRight.GO { .. . } нельзя! } class С : A. IRight { IRight.GO { ... } II можно IBase.F( int i ) { .. . } // можно } Класс наследует все методы своего предка, в том числе те, которые реализовывали интерфейсы. Он может переопределить эти методы с помощью спецификатора new, но обращаться к ним можно будет только через объект класса. Если использо­ вать для обращения ссылку на интерфейс, вызывается не переопределенная версия: interface IBase { void АО : } class Base : IBase { public void AO { .. . } } class Derived: Base { new public void AO { .. . } } Derived d = new Derived () ; d.AO; // вызывается Derived.AO; IBase id = d; id.AO; // вызывается Base.AO; Однако если интерфейс реализуется с помощью виртуального метода класса, по­ сле его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату: Интерфейсы и наследование 197 interface IBase { void АО; } class Base : IBase { public virtual void AO { .. . } } class Derived: Base { public override void AO { .. . } } Derived d = new Derived () : d.AO; // вызывается Derived.AO; IBase id = d; id.AO; // вызывается Derived.AO; Метод интерфейса, реализованный явным указанием имени, объявлять виртуаль­ ным запрещается. При необходимости переопределить в потомках его поведение пользуются следующим приемом: из этого метода вызывается другой, защищен­ ный метод, который объявляется виртуальным. В приведенном далее примере метод А интерфейса IBase реализуется посредством защищенного виртуального метода А\_, который можно переопределять в потомках класса Base: interface IBase { void AO; class Base : IBase { void IBase.AO { A\_(); } protected virtual void A\_() { .. . } class Derived: Base protected override void A\_() { .. . } Существует возможность повторно реализовать интерфейс, указав его имя в спи­ ске предков класса наряду с классом-предком, уже реализовавшим этот интер­ фейс. При этом реализация переопределенных методов базового класса во вни­ мание не принимается: interface IBase { void АО; 1 98 Глава 9. Интерфейсы и структурные типы } clas s Base : IBase I voi d IBase.A() { .. . } //н е используется в Derived clas s Derived : Base, IBase publi c voi d AO { .. . } i i Если класс наследует от класса и интерфейса, которые содержат методы с одина­ ковыми сигнатурами, унаследованный метод класса воспринимается как реали­ зация интерфейса, например: interfac e Interface l г voi d FO : } clas s Class l publi c voi d F() { .. . } pub1 ic voi d GO { .. . } } clas s Class-2 : Classl . Interface l F new publi c voi d GO { .. . } j Здесь класс Class2 наследует от класса Class l метод F. Интерфейс Interface l так­ же содержит метод F. Компилятор не выдает ошибку, потому что класс Class2 со­ держит метод, подходящий для реализации интерфейса. Вообще при реализации интерфейса учитывается наличие «подходящих» мето­ дов в классе независимо от их происхождения. Это могут быть методы, описан­ ные в текущем или базовом классе, реализующие интерфейс явным или неяв­ ным образом. Стандартные интерфейсы .NET В библиотеке классов ."NET определено множество стандартных интерфейсов, задающих желаемое поведение объектов. Например, интерфейс IComparable зада­ ет метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку. Реализация интерфейсов lEnumerable и enumerat e дает возможность просматривать содержимое объекта с помощью конструкции foreach , а реализация интерфейса ICloneabl e — клонировать объекты. 199 Стандартные интерфейсы поддерживаются многими стандартными классами биб­ лиотеки. Например, работа с массивами с помощью цикла foreac h возможна имен­ но потому, что тип Arra y реализует интерфейсы IEnumerable и IEnumerator. Можно создавать и собственные классы, поддерживающие стандартные интерфейсы, что позволит использовать объекты этих классов стандартными способами. Сравнение объектов (интерфейс IComparable) Интерфейс IComparable определен в пространстве имен System. Он содержит всего один метод CompareTo, возвращающий результат сравнения двух объектов — теку­ щего и переданного ему в качестве параметра: interfac e IComparable i n t CompareTo( objec t ob j ) Метод должен возвращать: • 0, если текущий объект и параметр равны; • отрицательное число, если текущий объект меньше параметра; • положительное число, если текущий объект больше параметра. Реализуем интерфейс IComparable в знакомом нам классе Monster. В качестве кри­ терия сравнения объектов выберем поле health . В листинге 9.1 приведена про­ грамма, сортирующая массив монстров по возрастанию величины, характери­ зующей их здоровье (элементы класса, не используемые в данной программе, не приводятся). Листин г 9.1 . Пример реализации интерфейса IComparable jsin g System; 'iamespace Consol eAppl ication i clas s Monster : IComparable { publi c MonsterC int. health , in t ammo, strin g name ) { this.healt h = health ; this.ammo = ammo: this.nam e = name; } virtua l publi c voi d Passport O { Console.WriteLineC "Monster {0} \t healt h = {1} ammo = {2}" , name, health , ammo ); } publi c in t CompareToC objec t ob j ) // реализация интерфейса продолжение Стандартные интерфейсы .NET 211 { for ( int i = 0; i < n; ++i ) if ( mas[i] .GetTypeO .Name == "Monster" ) yield return mas[i]; } public void Add( Monster m ) { if ( n >= 10 ) return: mas[n] = m; ++n: class Classl { static void MainO { Stado s = new StadoO; s.Add( new Monster() ); s.Add( new Monster("Bacя") ): s.Add( new Daemon() ); foreach ( Monster i in s ) i.PassportO foreach ( Monster i in s.Backwards О ) i.Passport О foreach ( Monster i in s.MonstersOnly() ) i.PassportO } Теперь, когда вы получили представление об итераторах, рассмотрим их более формально. Блок итератора синтаксически представляет собой обычный блок и может встре­ чаться в теле метода, операции или части get свойства, если соответствующее возвращаемое значение имеет тип IEnumerable или IEnumerator1 . В теле блока итератора могут встречаться две конструкции: • yield return формирует значение, выдаваемое на очередной итерации; • yield break сигнализирует о завершении итерации. Ключевое слово yield имеет специальное значение для компилятора только в этих конструкциях. Код блока итератора выполняется не так, как обычные блоки. Компилятор фор­ мирует служебный объект-перечислитель, при вызове метода MoveNext которого выполняется код блока итератора, выдающий очередное значение с помощью ключевого слова yield. Следующий вызов метода MoveNext объекта-перечислите­ ля возобновляет выполнение блока итератора с момента, на котором он был при­ остановлен в предыдущий раз. А также тип их параметризованных двойников IEnumerable или IEnumerator из про­ странства имен System. Collections .Generic, описанного в главе 13. 212 Глава 9. Интерфейсы и структурные типы Структур а — -с\та камш. , ^«^отдаадли к тьдакх ^ , лк^яювук л р^Ч\ «ш ш crţnvi чий от него: Q стоуклурл является значимым, a -fte ссылочным титло м дашх, то е.с\ъ •этаело ляр страт у ръ\ хранит значения своих элементов, а не ссылки на них, и расп лагается в стеке, а не в хипе; • структура не может участвовать в иерархиях наследования, она может толы реализовывать интерфейсы; • в структуре запрещено определять конструктор по умолчанию, поскольку с определен неявно и присваивает всем ее элементам значения по умолчани (нули соответствующего типа); • в структуре запрещено определять деструкторы, поскольку это бессмысленн ПРИМЕЧАНИ Е Строго говоря, любой значимый тип С# является структурным. Отличия от классов обусловливают область применения структур: типы данны имеющие небольшое количество полей, с которыми удобнее работать как со зн чениями, а не как со ссылками. Накладные расходы на динамическое выделен! памяти для небольших объектов могут весьма значительно снизить быстродейств! программы, поэтому их эффективнее описывать как структуры, а не как класс! ПРИМЕЧАНИ Е С другой стороны, передача структуры в метод по значению требует и дополю тельного времени, и дополнительной памяти. Синтаксис структуры: [ атрибуты ] [ спецификаторы ] struct имяструктуры [ : интерфейсы ] тело\_структуры [ ; ] Спецификаторы структуры имеют такой же смысл, как и для класса, причем у спецификаторов доступа допускаются только public, internal и private (после; ний — только для вложенных структур). Интерфейсы, реализуемые структурой, перечисляются через запятую. Тело струк туры может состоять из констант, полей, методов, свойств, событий, индексатс ров, операций, конструкторов и вложенных типов. Правила их описания и ис пользования аналогичны соответствующим элементам классов, за исключение: некоторых отличий, вытекающих из упомянутых ранее: • поскольку структуры не могут участвовать в иерархиях, для их элементов н могут'использоваться спецификаторы protected и protected internal; • структуры не могут быть абстрактными (abstract), к тому же по умолчанш они бесплодны (sealed); Структуры 213 • методы структур не могут быть абстрактными и виртуальными; • переопределяться (то есть описываться со спецификатором override ) могут только методы, унаследованные от базового класса object ; • параметр thi s интерпретируется как значение, поэтому его можно использо­ вать для ссылок, но не для присваивания; • при описании структуры нельзя задавать значения полей по умолчанию 1 — это будет сделано в конструкторе по умолчанию, создаваемом автоматически (конструктор присваивает значимым полям структуры нули, а ссылочным — значение nul 1). В листинге 9.8 приведен пример описания структуры, представляющей комплекс­ ное число. Дл я экономии места из всех операций приведено только описание сложения. Обратите внимание на перегруженный метод ToString : он позволяет выводить экземпляры структуры на консоль, поскольку неявно вызывается в ме­ тоде Console.WriteLine . Использованные в методе спецификаторы формата опи­ саны в приложении. Листин г 9.8. Пример структуры using System; namespace ConsoleApplicat i on l i struc t Complex { publi c double re , im; publi c Complex( double re\_ , double im\_ ) re = re\_ ; im = im\_ ; / / можно использовать this.re , this.i m publi c stati c Complex operato r + ( Complex a. Complex b ) retur n new Complex( a.r e + b.re , a.im + b.im ); publi c overrid e strin g ToString O retur n ( string.Format ( "({0.2:0.##};{1,2:0.##})" . re , im ) ); } clas s Class l { stati c voi d MainO { Complex a = new Complex! 1.2345, 5.6 ); продолжение i 1 К статическим полям это ограничение не относится. 214 Глава 9. Интерфейсы и структурные типы Листин г 9.8 (продолжение) Console.WriteLineC "а = " + а ); Complex b: b.re = 10; b.im = 1; Console.WriteLineC "b = " + b ); Complex с = new ComplexO; Console.WriteLineC "c = " + с ); с = a + b; Console.WriteLineC "c = " + с ); } } } Результат работы программы: а = (1.23:5,6) b = (10; 1) с = ( 0; 0) с = (11,23:6.6) При выводе экземпляра структуры на консоль выполняется упаковка, то есть не­ явное преобразование в ссылочный тип. Упаковка (это понятие было введено в разделе «Упаковка и распаковка», см. с. 36) применяется и в других случаях, когда структурный тип используется там, где ожидается ссылочный, например, при преобразовании экземпляра структуры к типу реализуемого ею интерфейса. При обратном преобразовании — из ссылочного типа в структурный — выполня­ ется распаковка. Присваивание структур имеет, что естественно, значимую семантику, то есть при присваивании создается копия значений полей. То же самое происходит и при передаче структур в качестве параметров по значению. Дл я экономии ре­ сурсов ничто не мешает передавать структуры в методы по ссылке с помощью ключевых слов ref или out. Особенно значительный выигрыш в эффективности можно получить, используя массивы структур вместо массивов классов. Например, для массива из 100 эк­ земпляров класса создается 101 объект, а для массива структур — один объ­ ект. Пример работы с массивом структур, описанных в предыдущем листинге: Complex [] mas = new Complex[4]; for ( int i = 0; i < 4; ++i ) { mas[i].re = i; mas[i].im = 2 \* i; } foreach ( Complex elem in mas ) Console.WriteLine( elem ); Перечисления 215 Если поместить этот фрагмент вместо тела метода Main в листинге 9.5, получим следующий результат: При написании программ часто возникает потребность определить несколько связанных между собой именованных констант, при этом их конкретные значе­ ния могут быть не важны. Для этого удобно воспользоваться перечисляемым ти­ пом данных, все возможные значения которого задаются списком целочислен­ ных констант, например: enum Menu { Read, Write , Append, Exi t } enum Радуга { Красный, Оранжевый, Желтый, Зеленый, Синий, Фиолетовый } Д ля каждой константы задается ее символическое имя. По умолчанию констан­ там присваиваются последовательные значения типа i nt, начиная с 0, но можно задать и собственные значения, например: enum Nums { two = 2, three , four , te n = 10, eleven , fift y = te n + 40 }: Константам thre e и fou r присваиваются значения 3 и 4, константе eleven — 11. Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать. Преимущество перечисления перед описанием именованных констант состоит в том, что связанные константы нагляднее; кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список. Синтаксис перечисления: [ атрибуты ] [ спецификаторы ] enum имя\_перечисления [ : базовыйтип ] тело\_перечисления [ ; ] Спецификаторы перечисления имеют такой же смысл, как и для класса, причем допускаются только спецификаторы new, public , protected , interna l и private . Базовый тип — это тип элементов, из которых построено перечисление. По умол­ чанию используется тип int , но можно задать тип и явным образом, выбрав его среди целочисленных типов (кроме char), а именно: byte , sbyte , short , ushort, int , uint , long и ulong. Необходимость в этом возникает, когда значения констант не­ возможно или неудобно представлять с помощью типа i nt. Тело перечисления состоит из имен констант, каждой из которых может быть присвоено значение. Если значение не указано, оно вычисляется прибавлением единицы к значению предыдущей константы. Константы по умолчанию имеют спецификатор доступа public . ( 0 ( 1 ( 2 ( 3 0) 2) 4) 6) Перечисления 216 Глава 9. Интерфейсы и структурные типы Перечисления часто используются как вложенные типы, идентифицируя значе­ ния из какого-либо ограниченного набора. Пример такого перечисления приве­ ден в листинге 9.9. Листин г 9.9. Пример перечисления using System; namespace ConsoleApplicationl { struct Боец { public enum Воинское\_Звание { Рядовой, Сержант, Майор, Генерал } public string Фамилия; public Воинское\_3вание Звание; } class Classl { static void MainO { Боец x; x.Фамилия = "Иванов"; x.Звание = Боец.Воинское\_Звание.Сержант; Console.WriteLineC х.Звание + " " + х.Фамилия ); } } } Результат работы программы: Сержант Иванов Перечисления удобно использовать для представления битовых флагов, например: enum Flags ; byte { ЬО, Ы. Ь2, ЬЗ = 0x04, Ь4 = 0x08, Ь5 = 0x10, Ь6 = 0x20, Ь7 = 0x40 } Операции с перечислениями С переменными перечисляемого типа можно выполнять арифметические операции (+, -, ++, --) , логические поразрядные операции (\* , &, |, ~) , сравнивать их с помо­ щью операций отношения (<=, >, >=, ==, != ) и получать размер в байтах (sizeof). При использовании переменных перечисляемого типа в целочисленных выражени­ ях и операциях присваивания требуется явное преобразование типа. Переменной перечисляемого типа можно присвоить любое значение, представимое с помощью Перечисления 217 оазового типа, то есть не только одно из значении, входящих в тело перечисления. Присваиваемое значение становится новым элементом перечисления. Пример: Flags а = Flags.Ь2 | Flags.Ь4; Console.WriteLineC "а = {0} {0,2:Х}", а ); ++а; Console.WriteLineC "а = {0} {0.2:X}". а ): int х = (int) а; Console.WriteLine( "x = {0} {0.2:X}", x ); Flags b = (Flags) 65; Console.WriteLine( "b = {0} {0.2:X}", b ); Результат работы этого фрагмента программы ({0,2:Х} обозначает шестнадцате- ричный формат вывода): а = 10 0А а = 11 0В х = 11 В D = 65 41 Другой пример использования операций с перечислениями приведен в листинге 9.10. Листин г 9.10 . Операции с перечислениями using System; namespace ConsoleApplicationi { struct Боец { public enum Воинское\_Звание { Рядовой, Сержант, Лейтенант, Майор, Полковник, Генерал } public string Фамилия; public Воинское\_Звание Звание; } class Classl { static void MainO { Боец x; x.Фамилия = "Иваново x.Звание = Боец.Воинское\_Звание.Сержант; for ( int i = 1976; i < 2006; i += 5 ) { if ( x.Звание < Боец.Воинское\_Звание.Генерал ) ++х.Звание; продолжение ^ 218 Глава 9. Интерфейсы и структурные типы Листин г 9.10 (продолжение) Console.WriteLineC "Год: {0} {1} {2}" . 1. х.Звание. х.Фамилия Результат работы программы: Год Год Год Год Год Год 1976 Лейтенант Иванов 1981 Майор Иванов 1986 Полковник Иванов 1991 Генерал Иванов 1996 Генерал Иванов 2001 Генерал Иванов Базовый класс System.Enum Все перечисления в С# являются потомками базового класса System.Enum, кото­ рый снабжает их некоторыми полезными методами. Статический метод GetName позволяет получить символическое имя константы по ее номеру, например: Console.WriteLineC Enum.GetName(typeof( Flags ). 8) ); // fc>4 Console.WriteLineC Enum.GetNameCtypeof( Боец.Воинское\_Звание ),!)) : // Сержант ПРИМЕЧАНИ Е Операция typeof возвращает тип своего аргумента (см. раздел «Рефлексия» в главе 12). Статические методы GetNames и GetValues формируют, соответственно, массивы имен и значений констант, составляющих перечисление, например: Array names = Enum.GetNames( typeof(Flags) ); Console.WriteLineC "Количество элементов в перечислении: " + names.Length ); foreach ( string elem in names ) Console.WriteC " " + elem ); Array values = Enum.GetValuesC typeof(Flags) ); foreach ( Flags elem in values ) Console.WriteC " " + (byte) elem ); Статический метод IsDefined возвращает значение true, если константа с задан­ ным символическим именем описана в указанном перечислении, и false в про­ тивном случае, например: if ( Enum.IsDefinedC typeof( Flags ), "Ь5" ) ) Console.WriteLine( "Константа с именем Ь5 существует" ); else Console.WriteLine( "Константа с именем Ь5 не существует" ); Статический метод GetUnderlyingType возвращает имя базового типа, на котором по­ строено перечисление. Например, для перечисления Flags будет получено System.Byte: Console.WriteLine( Enum.GetUnderlyingType( typeof(Flags) ) ); Рекомендации по программированию 219 Рекомендации по программированию Интерфейсы чаще всего используются для задания общих свойств объектов раз­ личных иерархий. Основная идея интерфейса состоит в том, что к объектам классов, реализующих интерфейс, можно обращаться одинаковым образом, при этом каждый класс может определять элементы интерфейса по-своему. Если некий набор действий имеет смысл только для какой-то конкретной иерар­ хии классов, реализующих эти действия разными способами, уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии. В С# поддерживается одиночное наследование для классов и множественное — для интерфейсов. Это позволяет придать производному классу свойства несколь­ ких базовых интерфейсов. Класс должен определять все методы всех интерфей­ сов, которые имеются в списке его предков. В библиотеке .NET определено большое количество стандартных интерфейсов. Реализация стандартных интерфейсов в собственных классах позволяет исполь­ зовать для объектов этих классов стандартные средства языка и библиотеки. Например, для обеспечения возможности сортировки объектов стандартными ме­ тодами следует реализовать в соответствующем классе интерфейсы IComparable или IComparer. Реализация интерфейсов IEnumerable и IEnumerator дает возможность просматривать содержимое объекта с помощью конструкции foreach, а реализа­ ция интерфейса ICloneable — клонировать объекты. Использование итераторов упрощает организацию перебора элементов и позво­ ляет задать для одного и того же класса различные стратегии перебора. Область применения структур — типы данных, имеющие небольшое количест­ во полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для экземплятров не­ больших классов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры. Преимущество использования перечислений для описания связанных между собой значений состоит в том, что это более наглядно и инкапсулировано, чем россыпь именованных констант. Кроме того, компилятор выполняет проверку типов, а ин­ тегрированная среда разработки подсказывает возможные значения констант, выводя их список. Глава 10 Делегаты, события и потоки выполнения - В этой главе рассматриваются делегаты и события — два взаимосвязанных сред­ ства языка С#, позволяющие организовать эффективное взаимодействие объек­ тов. Во второй части главы приводятся начальные сведения о разработке много­ поточных приложений. Делегаты Делегат — это вид класса, предназначенный для хранения ссылок на методы. Де­ легат, как и любой другой класс, можно передать в качестве параметра, а зате\ вызвать инкапсулированный в нем метод. Делегаты используются для поддерж ки событий, а также как самостоятельная конструкция языка. Рассмотрим снача ла второй случай. Описание делегатов Описание делегата задает сигнатуру методов, которые могут быть вызваны с ег< помощью: [ атрибуты ] [ спецификаторы ] delegate тип имяделегата ( [ параметры ] ) Спецификаторы делегата имеют тот же смысл, что и для класса, причем допуска ются только спецификаторы new, public, protected, internal и private. Тип описывает возвращаемое значение методов, вызываемых с помощью делега та, а необязательными параметрами делегата являются параметры этих методо! Делегат может хранить ссылки на несколько методов и вызывать их поочередж естественно, что сигнатуры всех методов должны совпадать. Глава 10 Делегаты, события и потоки выполнения - В этой главе рассматриваются делегаты и события — два взаимосвязанных сред­ ства языка С#, позволяющие организовать эффективное взаимодействие объек­ тов. Во второй части главы приводятся начальные сведения о разработке много­ поточных приложений. Делегаты Делегат — это вид класса, предназначенный для хранения ссылок на методы. Де­ легат, как и любой другой класс, можно передать в качестве параметра, а зате\ вызвать инкапсулированный в нем метод. Делегаты используются для поддерж ки событий, а также как самостоятельная конструкция языка. Рассмотрим снача ла второй случай. Описание делегатов Описание делегата задает сигнатуру методов, которые могут быть вызваны с ег< помощью: [ атрибуты ] [ спецификаторы ] delegate тип имяделегата ( [ параметры ] ) Спецификаторы делегата имеют тот же смысл, что и для класса, причем допуска ются только спецификаторы new, public, protected, internal и private. Тип описывает возвращаемое значение методов, вызываемых с помощью делега та, а необязательными параметрами делегата являются параметры этих методо! Делегат может хранить ссылки на несколько методов и вызывать их поочередж естественно, что сигнатуры всех методов должны совпадать. Делегаты 221 Пример описания делегата: public delegate void D ( int i ); Здесь описан тип делегата, который может хранить ссылки на методы, возвра­ щающие void и принимающие один параметр целого типа. ПРИМЕЧАНИ Е Делегат, как и всякий класс, представляет собой тип данных. Его базовым классом является класс System.Delegate, снабжающий своего «отпрыска» некоторыми полез­ ными элементами, которые мы рассмотрим позже. Наследовать от делегата нельзя, да и нет смысла. Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса. Использование делегатов Для того чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы. Делегаты применяются в основном для следующих целей: • получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы; • обеспечения связи между объектами по типу «источник — наблюдатель»; • создания универсальных методов, в которые можно передавать другие методы; • поддержки механизма обратных вызовов. Все эти варианты подробно обсуждаются далее. Рассмотрим сначала пример реа­ лизации первой из этих целей. В листинге 10.1 объявляется делегат, с помощью которого один и тот же оператор используется для вызова двух разных методов (С001 и Hack). Листин г 10.1 . Простейшее использование делегата using System: namespace ConsoleApplicationi delegate void Del ( ref string s ); // объявление делегата class Classl public static void C001 ( ref string s ) // метод 1 string temp = ""; for ( int i = 0: i < s.Length: ++i ) продолжение 222 Глава 10. Делегаты, события и потоки выполнения Листин г 10.1 (продолжение) if ( s[i] == 'о' 11 s[i] — '0') temp += 'O'; else if ( s[i] == 'I ' ) temp += 'Г ; else temp += s[ij ; } s = temp; } public static void Hack ( ref string s ) // метод 2 { string temp = ""; for ( int i = 0; i < s.Length; ++i ) if ( i / 2 \* 2 == i ) temp += char.ToUpper( s[i] ); else temp += s[i] ; } s = temp; static void MainO { string s = "cool hackers"; Del d; // экземпляр делегата for ( int i = 0; i < 2; ++i ) { d = new Del( COOl ); // инициализация методом 1 if ( i == 1 ) d = new Del(Hack); // инициализация методом 2 d( ref s ); // использование делегата для вызова методов Console.WriteLine( s ); } Результат работы программы: cOOl hackers COOl hAcKeRs Использование делегата имеет тот же синтаксис, что и вызов метода. Если деле­ гат хранит ссылки на несколько методов, они вызываются последовательно в том орядке, в котором были добавлены в делегат. Добавление метода в список выполняется либо с помощью метода Combine, унас­ ледованного от класса System.Delegate, либо, что удобнее, с помощью перегружен­ ной операции сложения. Вот как выглядит измененный метод Main из предыду­ щего листинга, в котором одним вызовом делегата выполняется преобразование исходной строки сразу двумя методами: V Делегаты 223 static void MainO string s = "cool hackers"; Del d = new Del ( C001 ) ; d += new Del( Hack ); // добавление метода в делегат d( ref s ); Console.WriteLineC s ); // результат: C001 hAcKeRs При вызове последовательности методов с помощью делегата необходимо учи­ тывать следующее: • сигнатура методов должна в точности соответствовать делегату; • методы могут быть как статическими, так и обычными методами класса; • каждому методу в списке передается один и тот же набор параметров; • если параметр передается по ссылке, изменения параметра в одном методе от­ разятся на его значении при вызове следующего метода; • если параметр передается с ключевым словом out или метод возвращает зна­ чение, результатом выполнения делегата является значение, сформированное последним из методов списка (в связи с этим рекомендуется формировать спи­ ски только из делегатов, имеющих возвращаемое значение типа voi d); • если в процессе работы метода возникло исключение, не обработанное в том же методе, последующие методы в списке не выполняются, а происходит по­ иск обработчиков в объемлющих делегат блоках; • попытка вызвать делегат, в списке которого нет ни одного метода, вызывает генерацию исключения System. Null Ref erenceExcepti on. Паттерн «наблюдатель» Рассмотрим применение делегатов для обеспечения связи между объектами по типу «источник — наблюдатель». В результате разбиения системы на множество совместно работающих классов появляется необходимость поддерживать согла­ сованное состояние взаимосвязанных объектов. При этом желательно избежать жесткой связанности классов, так как это часто негативно сказывается на воз­ можности многократного использования кода. Для обеспечения гибкой, динамической связи между объектами во время выпол­ нения программы применяется следующая стратегия. Объект, называемый ис­ точником, при изменении своего состояния, которое может представлять инте­ рес для других объектов, посылает им уведомления. Эти объекты называются наблюдателями. Получив уведомление, наблюдатель опрашивает источник, что­ бы синхронизировать с ним свое состояние. Примером такой стратегии может служить связь объекта с различными его пред­ ставлениями, например, связь электронной таблицы с созданными на ее основе диаграммами. 224 Глава 10. Делегаты, события и потоки выполнения Программисты часто используют одну и ту же схему организации и взаимодей­ ствия объектов в разных контекстах. За такими схемами закрепилось название паттерны, или шаблоны проектирования. Описанная стратегия известна под на­ званием паттерн «наблюдатель». Наблюдатель (observer) определяет между объектами зависимость типа «один ко многим», так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются. Рассмотрим пример (листинг 10.2), в котором демонстрируется схема оповещения источни­ ком трех наблюдателей. Гипотетическое изменение состояния объекта модели­ руется сообщением «OOPS!» . Один из методов в демонстрационных целях сде­ лан статическим. Листин г 10.2 . Оповещение наблюдателей с помощью делегата using System; namespace ConsoleApplicationi { public delegate void Del( object о ); // объявление делегата class Subj // класс-источник Del dels: // объявление экземпляра делегата public void Register( Del d ) // регистрация делегата dels += d: public void OOPS О // что-то произошло Console.WriteLineC "OOPS!" ): if ( dels != null ) dels( this ); // оповещение наблюдателей class ObsA // класс-наблюдатель public void Do( object о ) // реакция на событие источника Console.WriteLineC "Вижу, что OOPS!" ): class ObsB // класс-наблюдатель public static void SeeC object о ) // реакция на событие источника Console.WriteLineC "Я тоже вижу, что OOPS!" ): аты 225 \ :lass Classl { static void MainO { Subj s = new SubjO; ObsA ol = new ObsA(); ObsA o2 - new ObsAO; s.Register( new Del( ol.Do ) ); s.Register( new Del( o2.Do ) ); s.Register( new Del( ObsB.See ) ); s.OOPSO; } } источнике объявляется экземпляр делегата, в этот экземпляр заносятся мето- тех объектов, которые хотят получать уведомление об изменении состояния гочника. Этот процесс называется регистрацией делегатов. При регистрации (я метода добавляется к списку. Обратите внимание: для статического метода азывается имя класса, а для обычного метода — имя объекта. При наступлении аса X» все зарегистрированные методы поочередно вызываются через делегат. :зультат работы программы: PS! жу. что OOPS! жу, что OOPS! тоже вижу, что OOPS! ля обеспечения обратной связи между наблюдателем и источником делегат объ- влен с параметром типа object, через который в вызываемый метод передается ;ылка на вызывающий объект. Следовательно, в вызываемом методе можно по­ учать информацию о состоянии вызывающего объекта и посылать ему сообще- ия (то есть вызывать методы этого объекта). Звязь «источник — наблюдатель» устанавливается во время выполнения про- раммы для каждого объекта по отдельности. Если наблюдатель больше не хочет юлучать уведомления от источника, можно удалить соответствующий метод из писка делегата с помощью метода Remove или перегруженной операции вычита- гая, например: jublic void UnRegister( Del d ) // удаление делегата dels -= d; } // объект класса-источника // объекты // класса-наблюдателя // регистрация методов // наблюдателей в источнике // ( экземпляры делегата ) // инициирование события 226 Глава 10. Делегаты, события и потоки выполнения Операции Делегаты можно сравнивать на равенство и неравенство. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке. Сравнивать можно даже делега­ ты различных типов при условии, что они имеют один и тот же тип возвращае­ мого значения и одинаковые списки параметров. ПРИМЕЧАНИ Е Делегаты, различающиеся только именами, считаются имеющими разные типы. С делегатами одного типа можно выполнять операции простого и сложного при­ сваивания, например: Del dl = new Del( ol.Do ): // ol.Do Del d2 = new Del( o2.Do ); // o2.Do Del d3 = dl + d2; // ol.Do и o2.Do d3 += dl : // ol.Do. o2.Do и ol.Do d3 -= d2; // Ol.Do и ol.Do Эти операции могут понадобиться, например, в том случае, если в разных об­ стоятельствах требуется вызывать разные наборы и комбинации наборов ме­ тодов. Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии уда­ ляется сборщиком мусора. Передача делегатов в методы Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается функциональная параметризация: в ме­ тод можно передавать не только различные данные, но и различные функции их обработки. Функциональная параметризация применяется для создания универ­ сальных методов и обеспечения возможности обратного вызова. В качестве простейшего примера универсальною метода можно привести метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции. Этот пример приво­ дится далее. Обратный вызов (callback) представляет собой вызов функции, передаваемое в другую функцию в качестве параметра. Рассмотрим рис. 10.1. Допустим, в биб лиотеке описана функция А, параметром которой является имя другой функции В вызывающем коде описывается функция с требуемой сигнатурой (В) и переда ется в функцию А. Выполнение функции А приводит к вызову В, то есть управле ние передается из библиотечной функции обратно в вызывающий код. Делегаты 227 - ь - • в Вызов А (В)- А(Х) Вызов X ч у Вызывающий код /' \ Библиотека V У Ч У Рис. 10.1 . Механизм обратного вызова Механизм обратного вызова широко используется в программировании. Напри­ мер, он реализуется во многих стандартных функциях Windows. Пример передачи делегата в качестве параметра приведен в листинге 10.3. Про­ грамма выводит таблицу значений функции на заданном интервале с шагом, равным единице. Листин г 10.3. Передача делегата через список параметров using System: namespace ConsoleApplicationl { public delegate double Fun( double x ); // объявление делегата class Classl { . public static void Tablet Fun F, double x, double b ) { Console.WriteLineC " X Y " ); while (x <= b) { Console.WriteLineC "| {0,8:0.000} | {1,8:0.000} |". x, F(x)); x += 1: } Console.WriteLineC " " ); public static double SimpleC double x ) { return 1: } static void MainO { Console.WriteLineC " Таблица функции Sin " ); продолжение 2 2 8 Глава 10. Делегаты, события и потоки выполнения Листин г 10.3 (продолжение) Tablet new Fun( Math.Sin ), -2, 2 ); Console.WriteLineC " Таблица функции Simple " ); TableC new Fun( Simple ), 0. 3 ); } } } Результат работы программы: Таблица функции Sin X Y I -2.000 I -0,909 I I -1.000 I -0,841 j I 0.000 I 0,000 j I 1,000 I 0.841 I I 2,000 j 0,909 I Таблица функции Simple X Y I 0,000 I 1,000 I I 1.000 I 1,000 I I 2,000 I 1,000 j I 3,000 I 1.000 I В среде Visual Studio 2005, использующей версию 2.0 языка C#, можно при­ менять упрощенный синтаксис для делегатов. Первое упрощение заключается в том, что в большинстве случаев явным образом создавать экземпляр делегата не требуется, поскольку он создается автоматически по контексту. Второе упро­ щение заключается в возможности создания так называемых анонимных мето­ дов — фрагментов кода, описываемых непосредственно в том месте, где исполь­ зуется делегат. В листинге 10.4 использованы оба упрощения для реализации тех же действий, что и листинге 10.3. Листин г 10.4 . Передача делегата через список параметров (версия 2.0) using System; namespace ConsoleAppl icationi { public delegate double FunC double x ); // объявление делегата class Classl { public static void TableC Fun F. double x, double b ) { Console.WriteLine( " X Y " ): while ( x <= b ) Делегаты 229 { Console.WriteLineC "| {0.8:0.000} | {1,8:0.000} |", x. FCx)): x += 1; } Console.WriteLineC " " ): } static void MainO { Console.WriteLineC " Таблица функции Sin " ): TableC Math.Sin. -2. 2 ); // упрощение 1 Console.WriteLineC " Таблица функции Simple " ); TableC delegate (double x ){ return 1: }, 0. 3 ); // упрощение 2 } } } В первом случае экземпляр делегата, соответствующего функции Sin, создается автоматически 1 . Чтобы это могло произойти, список параметров и тип возвра­ щаемого значения функции должны быть совместимы с делегатом. Во втором случае не требуется оформлять простой фрагмент кода в виде отдельной функции Simple, как это было сделано в предыдущем листинге, — код функции оформля­ ется как анонимный метод и встраивается прямо в место передачи. Альтернативой использованию делегатов в качестве параметров являются вирту­ альные методы. Универсальный метод вывода таблицы значений функции можно реализовать с помощью абстрактного базового класса, содержащего два метода: метод вывода таблицы и абстрактный метод, задающий вид вычисляемой функции. Для вывода таблицы конкретной функции необходимо создать производный класс, переопределяющий этот абстрактный метод. Реализация метода вывода таблицы с помощью наследования и виртуальных методов приведена в листинге 10.5. Листин г 10.5 . Альтернатива параметрам-делегатам using System; namespace ConsoleAppli cationl { abstract class TableFun { public abstract double F( double x ); public void TableC double x, double b ) { Console.WriteLineC " X Y while ( x <= b ) 1 В результате в 2005 году язык С# в этой части вплотную приблизился к синтаксису ста­ рого доброго Паскаля, в котором передача функций в качестве параметров была реализо­ вана еще в 1992 году, если не раньше. продолжение & 230 Листин г 10.5 (продолжение) { Console.WriteLineC "| {0.8:0.000} | {1.8:0.000} |". х, FCx)); х += 1; } Console.WriteLineC " " ); ' } } class SimpleFun : TableFun { public override double F( double x ) { return 1; } } class SinFun : TableFun { public override double F( double x ) { return Math.Sin(x); } } class Classl { static void MainO { \ TableFun a = new SinFunO: Console.WriteLineC " Таблица функции Sin " ); a.TableC -2. 2 ): a = new SimpleFunO; Console.WriteLineC " Таблица функции Simple " ): a.TableC 0, 3 ); } } } Результат работы этой программы такой же, как и предыдущей, но, на мой взгляд, в данном случае применение делегатов предпочтительнее. Обработка исключений при вызове делегатов Ранее говорилось о том, что если в одном из методов списка делегата генериру­ ется исключение, следующие методы не вызываются. Этого можно избежать, если обеспечить явный перебор всех методов в проверяемом блоке и обрабатывать возникающие исключения. Все методы, заданные в экземпляре делегата, можно Делегаты 231 получить с помощью унаследованного метода GetlnvocationList. Этот прием иллю­ стрирует листинг 10.6, представляющий собой измененный вариант листинга 10.1. Листин г 10.в . Перехват исключений при вызове делегата using System; namespace ConsoleApplicationi { delegate void Del ( ref string s ); class Classl { public static void C001 ( ref string s ) { Console.WriteLineC "вызван метод C001" ); string temp = ""; for ( int i = 0; i < s.Length; ++i ) { if ( s[i] == 'o' || s[i] == 'O') temp += '0'; else if ( s[i] == '1 ' ) temp += 'Г ; else temp += s[i]; } s = temp; } public static void Hack ( ref string s ) { Console.WriteLineC "вызван метод Hack" ); string temp = ""; for ( int i = 0; i < s.Length; ++i ) if ( i / 2 \* 2 == i ) temp += char.ToUpperC s[i] ); else temp += s[i]; s = temp; public static void BadHack ( ref string s ) { Console.WriteLineC "вызван метод BadHack" ); throw new Exception(); // имитация ошибки static void MainO { string s = "cool hackers"; Del d = new Del С C001 ) ; d += new Del( BadHack ); d += new DelC Hack ); // создание экземпляра делегата // дополнение списка методов // дополнение списка методов продолжение г? 232 Глава 10. Делегаты, события и потоки выполнения Листин г 10. 6 (продолжение) foreach ( Del fun in d.GetlnvocationListO ) { try { fun( ref s ); // вызов каждого метода из списка } catch ( Exception е ) { Console.WriteLineC е.Message ); Console.WriteLineC "Exception in method " + fun.Method.Name); } } Console.WriteLineC "результат - " + s ); } } } Результат работы программы: вызван метод COOl вызван метод BadHack Exception of type System.Exception was thrown. Exception in method BadHack вызван метод Hack результат - COOl hAcKeRs В этой программе помимо метода базового класса GetlnvocationList использова­ но свойство Method. Это свойство возвращает результат типа Method Info. Класс Method Info содержит множество свойств и методов, позволяющих получить пол­ ную информацию о методе, например его спецификаторы доступа, имя и тип возвращаемого значения. Мы рассмотрим этот интересный класс в разделе «Реф­ лексия» главы 12. События Событие — это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, являющих­ ся наблюдателями события, активизируются методы-обработчики этого собы­ тия. Обработчики должны быть зарегистрированы в объекте-источнике события. Таким образом, механизм событий формализует на языковом уровне паттерн «наблюдатель», который рассматривался в предыдущем разделе. Механизм событий можно также описать с помощью модели «публикация — под­ писка»: один класс, являющийся отправителем (sender) сообщения, публикует события, которые он может инициировать, а другие классы, являющиеся получа­ телями (receivers) сообщения, подписываются на получение этих событий. События 233 События построены на основе делегатов: с помощью делегатов вызываются ме­ тоды-обработчики событий. Поэтому создание события в классе состоит из сле­ дующих частей: • описание делегата, задающего сигнатуру обработчиков событий; • описание события; • описание метода (методов), инициирующих событие. Синтаксис события похож на синтаксис делегата 1 : [ атрибуты ] [ спецификаторы ] event тип имясобытия Для событий применяются спецификаторы new, public, protected, internal, private, static, virtual, sealed, override, abstract и extern, которые изучались при рассмот­ рении методов классов. Например, так же как и методы, событие может быть ста­ тическим (static), тогда оно связано с классом в целом, или обычным — в этом случае оно связано с экземпляром класса. Тип события — это тип делегата, на котором основано событие. Пример описания делегата и соответствующего ему события: public delegate void Del( object о ); // объявление делегата class А { public event Del Oops; // объявление события } Обработка событий выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответст­ вует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод. Как видите, это в точности тот же самый механизм, который рассматривался в пре­ дыдущем разделе. Единственное отличие состоит в том, что при использовании событий не требуется описывать метод, регистрирующий обработчики, посколь­ ку события поддерживают операции += и -=, добавляющие обработчик в список и удаляющие его из списка. ПРИМЕЧАНИ Е Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата. В листинге 10.7 приведен код из листинга 10.2, переработанный с использовани­ ем событий. 1 Приводится упрощенный вариант. В общем случае имеется возможность задавать не­ сколько имен событий, инициализаторы и методы добавления и удаления событий. 234 Глава 10. Делегаты, события и потоки выполнения Листин г 10.7 . Оповещение наблюдателей с помощью событий using System; namespace ConsoleApplicationl { public delegate void DelO; class Subj { public event Del Oops; publi { c void CryOopsO // объявление делегата // класс-источник // объявление события // метод, инициирующий событие Console.WriteLineC "OOPS!" ); if ( Oops != null ) OopsO: } class ObsA { public void DoO ; // класс-наблюдатель // реакция на событие источника Console.WriteLineC "Вижу, что OOPS!" ); } class ObsB { public static void SeeO { Console.WriteLineC "Я тоже вижу, что OOPS!" ) } // класс-наблюдатель // реакция на событие источника class Classl { static void MainO { Subj s = new Subj(); / / ' объект класса-источника ObsA ol = new ObsAC): / / объекты ObsA o2 = new ObsAC); / / класса-наблюдателя s.Oops += new DelC ol.Do ); / / добавление s.Oops += new DelC o2.Do ); // обработчиков s.Oops += new DelC ObsB.See ); // к событию s. CryOopsO; / / инициирование события События 235 Внешний код может работать с событиями единственным образом: добавлять об­ работчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от опе­ раций сложного присваивания для арифметических типов. Иного способа досту­ па к списку обработчиков нет. Внутри класса, в котором описано событие, с ним можно обращаться, как с обыч­ ным полем, имеющим тип делегата: использовать операции отношения, присваи­ вания и т. д. Значение события по умолчанию — null. Например, в методе CryOops выполняется проверка на nul 1 для того, чтобы избежать генерации исключения System.Nul1ReferenceException. В библиотеке .NET описано огромное количество стандартных делегатов, пред­ назначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам: • имя делегата заканчивается суффиксом EventHandler; • делегат получает два параметра: О первый параметр задает источник события и имеет тип object; О второй параметр задает аргументы события и имеет тип EventArgs или про­ изводный от него. Если обработчикам события требуется специфическая информация о событии, то для этого создают класс, производный от стандартного класса EventArgs, и до­ бавляют в него необходимую информацию. Если делегат не использует такую информацию, можно не описывать делегата и собственный тип аргументов, а обой­ тись стандартным классом делегата System.EventHandler. Имя обработчика события принято составлять из префикса On и имени события. В листинге 10.8 приведен пример из листинга 10.7, оформленный в соответствии со стандартными соглашениями .NET. Найдите восемь отличий! Листин г 10.8. Использование стандартного делегата EventHandler using System; namespace ConsoleApplicationi { class Subj { public event EventHandler Oops; public void CryOops() { Console.WriteLineC "OOPS!" ); if ( Oops != null ) OopsC this, null ); } } class ObsA ^ продолжение & 236 Глава 10. Делегаты, события и потоки выполнения Листин г 10. 8 (продолжение) public void OnOopsC object sender. EventArgs e ) { Console.WriteLineC "Вижу, что OOPS!" ); } } class ObsB { public static void OnOopsC object sender, EventArgs e ) { Console.WriteLineC "Я тоже вижу, что OOPS!" ): } } class Classl { static void MainO { Subj s = new SubjC); ObsA ol = new ObsAC); ObsA o2 = new ObsAC); s.Oops += new EventHandlerC ol.OnOops ): s.Oops += new EventHandlerC o2.0nOops ): s.Oops +- new EventHandlerC ObsB.OnOops ); s. CryOopsO: } } } Те, кто работает с С# версии 2.0, могут упростить эту программу, используя но­ вую возможность неявного создания делегатов при регистрации обработчиков событий. Соответствующий вариант приведен в листинге 10.9. В демонстраци­ онных целях в код добавлен новый анонимный обработчик — еще один меха­ низм, появившийся в новой версии языка. Листин г 10.9 . Использование делегатов и анонимных методов (версия 2.0) using System; namespace ConsoleApplicationl { class Subj { public event EventHandler Oops; public void CryOopsO { Console.WriteLineC "OOPS!" ) Многопоточные приложения 237 if ( Oops != null ) Oops( this, null ); } class ObsA { public void OnOopsC object sender, EventArgs e ) { Console.WriteLineC "Вижу, что OOPS!" ); } } class ObsB { public static void OnOopsC object sender, EventArgs e ) { Console.WriteLineC "Я тоже вижу, что OOPS!" ): class Classl { static void MainO { Subj s = new SubjC); ObsA ol ' new ObsAC); ObsA o2 = new ObsAC); s.Oops += ol.OnOops; s.Oops += o2.0n0ops; s.Oops += ObsB.OnOops; s.Oops += delegate ( object sender, EventArgs e ) { Console.WriteLineC "Я с вами!" ); }; } } s.CryOopsO; } События включены во многие стандартные классы .NET, например, в классы пространства имен Windows.Forms, используемые для разработки Windows-прило­ жений. Мы рассмотрим эти классы в главе 14. Многопоточные приложения Приложение .NET состоит из одного или нескольких процессов. Процессу при­ надлежат выделенная для него область оперативной памяти и ресурсы. Каждый процесс может состоять из нескольких доменов (частей) приложения, ресурсы 238 Глава 10. Делегаты, события и потоки выполнения которых изолированы друг от друга. В рамках домена может быть запущено не­ сколько потоков выполнения. Поток (thread1 ) представляет собой часть испол­ няемого кода программы. В каждом процессе есть первичный поток, исполняющий роль точки входа в приложение. Для консольных приложений это метод Main. Многопоточные приложения создают как для многопроцессорных, так и для од­ нопроцессорных систем. Основной целью при этом являются повышение общей производительности и сокращение времени реакции приложения. Управление потоками осуществляет операционная система. Каждый поток получает некото­ рое количество квантов времени, по истечении которого управление передается другому потоку. Это создает у пользователя однопроцессорной машины впечат­ ление одновременной работы нескольких потоков и позволяет, к примеру, вы­ полнять ввод текста одновременно с длительной операцией по передаче данных. Недостатки многопоточности: • большое количество потоков ведет к увеличению накладных расходов, свя­ занных с их переключением, что снижает общую производительность системы; • в многопоточных приложениях возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков (например, если один поток начинает измене­ ние общих данных, а отведенное ему время истекает, доступ к этим же дан­ ным может получить другой поток, который, изменяя данные, необратимо их повреждает). Поддержка многопоточности осуществляется в .NET в основном с помощью пространства имен System.Threading. Некоторые типы этого пространства описа­ ны в табл. 10.1. Таблица 10.1 . Некоторые типы пространства имен System.Threading Класс Thread Тип Описание Interlocked Класс, обеспечивающий синхронизированный доступ к переменным, которые используются в разных потоках Monitor Класс, обеспечивающий синхронизацию доступа к объектам Mutex Класс-примитив синхронизации, который используется также для синхронизации между процессами Класс, определяющий блокировку, поддерживающую один доступ на запись и несколько — на чтение ReaderWriterLock Thread Класс, который создает поток, устанавливает его приоритет, получает информацию о состоянии 1 Иногда этот термин переводится буквально — «нить», чтобы отличить его от потоков ввода-вывода, которые рассматриваются в следующей главе. Поэтому в литературе мож­ но встретить и термин «многонитевые приложения». ^Многолоточные приложения 239 Тип Описание ThreadPool Класс, используемый для управления набором взаимосвязанных потоков — пулом потоков Timer Класс, определяющий механизм вызова заданного метода в заданные интервалы времени для пула потоков WaitHandle Класс, инкапсулирующий объекты синхронизации, которые ожидают доступа к разделяемым ресурсам IOCompletionCallback Класс, получающий сведения о завершившейся операции ввода-вывода ThreadStart Делегат, представляющий метод, который должен быть выполнен при запуске потока TimerCallback Делегат, представляющий метод, обрабатывающий вызовы от класса Timer WaitCaПback Делегат, представляющий метод для элементов класса ThreadPool ThreadPriority Перечисление, описывающее приоритет потока ThreadState Перечисление, описывающее состояние потока Первичный поток создается автоматически. Для запуска вторичных потоков ис­ пользуется класс Thread. При создании объекта-потока ему передается делегат, определяющий метод, выполнение которого выделяется в отдельный поток: Thread t = new Thread ( new ThreadStartC имя\_метода ) ); После создания потока заданный метод начинает в нем свою работу, а первич­ ный поток продолжает выполняться. В листинге 10.10 приведен пример одно­ временной работы двух потоков. Листин г 10.10 . Создание вторичного потока using System: using System.Threading: namespace ConsoleAppl icationi { class Program { static public void HedgehogO // метод для вторичного потока { for ( int i = 0; i < 6: ++i ) { Console.WriteLineC i ); Thread.SleepC 1000 ); } } static void MainO { Console.WriteLineC "Первичный поток " + Thread.CurrentThread.GetHashCodeO ); продолжение 240 Глава 10. Делегаты, события и потоки выполнения Листин г 10.1 0 (продолжение) Thread ta = new ThreadC new ThreadStart(Hedgehog) ); Console.WriteLineC "Вторичный поток " + ta.GetHashCodeC) ); ta.StartC); for ( int i = 0; i > -6: --i ) { . Console.WriteC " " + i ); Thread.SleepC 400 ); } } } } Результат работы программы: Первичный поток 1 Вторичный поток 2 0 0 -1 -2 1 -3 -4 2 -5 3 4 5 В листинге используется метод Sleep, останавливающий функционирование пото­ ка на заданное количество миллисекунд. Как видите, оба потока работают одно­ временно. Если бы они работали с одним и тем же файлом, он был бы испорчен так же, как и приведенный вывод на консоль, поэтому такой способ распаралле­ ливания вычислений имеет смысл только для работы с различными ресурсами. В табл. 10.2 перечислены основные элементы класса Thread. Таблица 10.2 . Основные элементы класса Thread Элемент Вид Описание CurrentThread Статическое Возвращает ссылку на выполняющийся поток (только свойство для чтения) IsAlive Свойство Возвращает true или false в зависимости от того, запущен поток или нет IsBackground Свойство Возвращает или устанавливает значение, которое показывает, является ли этот поток фоновым Name Свойство Установка текстового имени потока Priority Свойство Получить/установить приоритет потока (используются значения перечисления ThreadPrority) ThreadState Свойство Возвращает состояние потока (используются значения перечисления ThreadState) Abort Метод Генерирует исключение ThreadAbortException. Вызов этого метода обычно завершает работу потока GetData, Статические Возвращает (устанавливает) значение для указанного SetData методы слота в текущем потоке GetDomain, Статические Возвращает ссылку на домен приложения (идентификатор GetDomainID методы домена приложения), в рамках которого работает поток Многопоточные приложения 241 Элемент Вид Описание GetHashCode Метод Возвращает хеш-код для потока Sleep Статический метод Приостанавливает выполнение текущего потока на заданное количество миллисекунд Interrupt Метод Прерывает работу текущего потока Join Метод Блокирует вызывающий поток до завершения другого потока или указанного промежутка времени и завершает поток Resume Метод Возобновляет работу после приостановки потока Start Метод Начинает выполнение потока, определенного делегатом ThreadStart Suspend Метод Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется Можно создать несколько потоков, которые будут совместно использовать один и тот же код. Пример приведен в листинге 10.11. Листин г 10.11 . Потоки, использующие один объект using System; using System.Threading; namespace ConsoleApplicationi { class Classl { public void Do() { for ( int i - 0: i < 4; ++i ) { Console.Write( " " + i ); Thread.SIeep( 3 ); } } } class Program { static void MainO { Classl a = new ClassK); Thread tl = new Thread( new ThreadStartC a.Do ) ); tl.Name = "SecondО Console.WriteLineC "Поток " + tl.Name ); tl.StartO; Thread t2 = new ThreadC new ThreadStartC a.Do ) ); t2.Name = "Third": Console.WriteLineC "Поток " + t2.Name ); t2.Start(); } } } 242 Глава 10. Делегаты, события и потоки выполнения Результат работы программы: Поток Second Поток Third 0 0 1 1 2 2 3 3 Варианты вывода могут несколько различаться, поскольку один поток прерыва­ ет выполнение другого в неизвестные моменты времени. Для того чтобы блок кода мог использоваться в каждый момент только одним потоком, применяется оператор lock. Формат оператора: lock ( выражение ) блок\_операторов Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово this, для статиче­ ских — typeof (класс). Блок операторов задает критическую секцию кода, которую требуется заблокировать. Например, блокировка операторов в приведенном ранее методе Do выглядит сле­ дующим образом: public void Do() { lockC this ) { for ( int i = 0; i < 4; ++i ) { Console.Write( " " + i ): Thread.SIeep( 30 ): } } } Для такого варианта метода результат работы программы изменится: Поток Second Поток Third 0 1 2 3 0 1 2 3 Асинхронные делегаты Делегат можно вызвать на выполнение либо синхронно, как во всех приведенных ранее примерах, либо асинхронно с помощью методов Beginlnvoke и Endlnvoke. При вызове делегата с помощью метода Beginlnvoke среда выполнения создает для исполнения метода отдельный поток и возвращает управление оператору, следующему за вызовом. При этом в исходном потоке можно продолжать вычис­ ления. • Если при вызове Beginlnvoke был указан метод обратного вызова, этот метод вызы­ вается после завершения потока. Метод обратного вызова также задается с помо­ щью делегата, при этом используется стандартный делегат AsyncCal 1 back. В методе, обратного вызова для получения возвращаемого значения и выходных парамет­ ров применяется метод Endlnvoke. ' j j Многопоточные приложения 243 Есл и метод обратного вызова не бы л указан в параметрах метода Beginlnvoke, ме­ тод End Invoke можн о использовать в потоке, инициировавше м запрос. В листинге 10.11 приводятся два пример а асинхронного вызова метода, выпол­ няющег о разложение числа на множители. Листин г приводится по документа­ ц и и Visual Studio с некоторым и изменениями. Класс Factorizer содержит метод Factorize, выполняющи й разложение на мно­ жители. Этот метод асинхронно вызывается двум я способами: в методе Numl ме­ тод обратного вызова задается в Beginlnvoke, в методе Num2 имею т место ожидани е завершения потока и непосредственный вызов End Invoke. Листин г 10.11 . Асинхронные делегаты using System: using System.Threading; using System.Runtime.Remoting.Messaging; // асинхронный делегат public delegate bool AsyncDelegate ( int Num. out int ml, out int m2 ); // класс, выполняющий разложение числа на множители public class Factorizer { public bool FactorizeC int Num. out int ml, out int m2 ) { ml = 1; m2 = Num; for ( int i = 2; i < Num; i++ ) if ( 0 == (Num % i) ) { ml = i; m2 = Num / i; break; } if (1 == ml ) return false; else return true; } } // класс, получающий делегата и результаты public class PNum { private int Number; public PNum( int number ) { Number = number; } [OneWayAttributeO] // метод, получающий результаты public void Res( IAsyncResult ar ) { int ml, m2; // получение делегата из AsyncResult AsyncDelegate ad = (AsyncDelegate)((AsyncResult)ar).AsyncDelegate; // получение результатов выполнения метода Factorize ad.EndInvoke( out ml, out m2, ar ); продолжение & 244 Глава 10. Делегаты, события и потоки выполнения Листин г 10.1 1 (продолжение) II вывод результатов Console.WriteLineC "Первый способ : множители {0} : {1} {2}", Number, ml. m2 ); } } // демонстрационный класс public class Simple { // способ 1: используется функция обратного вызова public void NumlO { Factorizer f = new FactorizerO; AsyncDelegate ad = new AsyncDelegate ( f.Factorize ); int Num = 1000589023, tmp; // создание экземпляра класса, который будет вызван // после завершения работы метода Factorize PNum n = new PNum( Num ); // задание делегата метода обратного вызова AsyncCallback callback = new AsyncCallbackC n.Res ): // асинхронный вызов метода Factorize IAsyncResult ar = ad.BeginlnvokeC Num, out tmp, out tmp, callback, null ); // // здесь - выполнение неких дальнейших действий // ... } // способ 2: используется ожидание окончания выполнения public void Num2() { Factorizer f = new FactorizerO; AsyncDelegate ad = new AsyncDelegate ( f.Factorize ): int Num = 1000589023. tmp; // создание экземпляра класса, который будет вызван // после завершения работы метода Factorize PNum n = new PNum( Num ); // задание делегата метода обратного вызова AsyncCallback callback = new AsyncCallbackC n.Res ); // асинхронный вызов метода Factorize IAsyncResult ar = ad.BeginlnvokeC Num, out tmp, out tmp, null, null ); // ожидание завершения ar.AsyncWaitHandle.WaitOne( 100.00. false ); Рекомендации по программированию 245 if ( ar.IsCompleted ) { int ml. m2; // получение результатов выполнения метода Factorize ad.Endlnvoket out ml, out m2. ar ); // вывод результатов Console.WriteLine( "Второй способ : множители {0} : {1} {2}", Num, ml. m2 ); } } public static void MainO { Simple s = new SimpleO; s.NumlO; S.Num2(); } } Результат работы программы: Первый способ : множители 1000589023 : 7 142941289 Второй способ : множители 1000589023 : 7 142941289 ПРИМЕЧАНИ Е Атрибут [OneWayAttributeO] помечает метод как не имеющий возвращаемого зна­ чения и выходных параметров. Рекомендации по программированию Делегаты широко применяются в библиотеке .NET как самостоятельно, так и для поддержки механизма событий, который имеет важнейшее значение при про­ граммировании под Windows. Делегат представляет собой особый вид класса, несколько напоминающий ин­ терфейс, но, в отличие от него, задающий только одну сигнатуру метода. В языке С + + аналогом делегата является указатель на функцию, но он не обладает безо­ пасностью и удобством использования делегата. Благодаря делегатам становит­ ся возможной гибкая организация взаимодействия, позволяющая поддерживать согласованное состояние взаимосвязанных объектов. Начиная с версии 2.0, в С# поддерживаются возможности, упрощающие процесс программирования с применением делегатов — неявное создание делегатов при регистрации обработчиков событий и анонимные обработчики. Основной целью создания многопоточных приложений является повышение общей производительности программы. Однако разработка многопоточных при­ ложений сложнее, поскольку при этом возникают проблемы синхронизации данных, связанные с потенциальной возможностью доступа к одним и тем же данным со стороны нескольких потоков. Глава 11 Работа с файлами Под файлом обычно подразумевается именованная информация на внешнем носи­ теле, например на жестком или гибком магнитном диске. Логически файл можно представить как конечное количество последовательных байтов, поэтому такие устройства, как дисплей, клавиатура и принтер, также можно рассматривать как частные случаи файлов. Передача данных с внешнего устройства в оперативную память называется чтением, или вводом, обратный процесс — записью, или выводом. Ввод-вывод в С# выполняется с помощью подсистемы ввода-вывода и классов библиотеки .NET. В этой главе рассматривается обмен данными с файлами и их частным случаем — консолью. Обмен данными реализуется с помощью потоков. Поток (stream)1 — это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. Потоки обеспечивают надежную работу как со стандартными, так и с определенными пользователем типами данных, а также единообразный и понятный синтаксис. Поток определяется как последователь­ ность байтов и не зависит от конкретного устройства, с которым производится обмен (оперативная память, файл на диске, клавиатура или принтер). Обмен с потоком для повышения скорости передачи данных производится, как правило, через специальную область оперативной памяти — буфер. Буфер выде­ ляется для каждого открытого файла. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь буфер не запол­ нится. Только после этого или после специальной команды сброса происходит передача данных на внешнее устройство. При чтении из файла данные вначале считываются в буфер, причем не столько, сколько запрашивается, а сколько по­ мещается в буфер. Механизм буферизации позволяет более быстро и эффективно обмениваться информацией с внешними устройствами. 1 Не путать с потоком выполнения, описанным в предыдущей главе. Впрочем, и в обыч­ ной речи мы часто обозначаем одним и тем же словом совершенно разные вещи! Работа с файлами 247 Для поддержки потоков библиотека .NET содержит иерархию классов, основная часть которой представлена на рис. 11.1. Эти классы определены в пространстве имен System. 10. Помимо классов там описано большое количество перечислений для задания различных свойств и режимов. Рис. 11.1 . Классы библиотеки .NET для работы с потоками Классы библиотеки позволяют работать в различных режимах с файлами, ката­ логами и областями оперативной памяти. Краткое описание классов приведено в табл. 11.1. Таблиц а 11.1 . Основные классы пространства имен System.Ю Класс Описание BinaryReader, BinaryWriter BufferedStream Di rectory, Directorylnfo, File, Filelnfo FileStream MemoryStream Чтение и запись значений простых встроенных типов (целочисленных, логических, строковых и т. п.) во внутренней форме представления Временное хранение потока байтов (например, для последующего переноса в постоянное хранилище) Работа с каталогами или физическими файлами: создание, удаление, получение свойств. Возможности классов Fi 1 е и Di rectory реализованы в основном в виде статических методов. Аналогичные классы Directorylnfo и Filelnfo используют обычные методы Произвольный (прямой) доступ к файлу, представленному как поток байтов Произвольный доступ к потоку байтов в оперативной памяти продолжение & 248 Глава 11. Работа с файлами Таблица 11.1 (продолжение) Класс Описание StreamWriter, Чтение из файла и запись в файл текстовой информации StreamReader (произвольный доступ не поддерживается) StringWriter, Работа с текстовой информацией в оперативной памяти StringReader Как можно видеть из таблицы, выполнять обмен с внешними устройствами мож­ но на уровне: • двоичного представления данных (BinaryReader, BinaryWriter); • байтов (FileStream); • текста, то есть символов (StreamWriter, StreamReader). В .NET используется кодировка Unicode, в которой каждый символ кодируется двумя байтами. Классы, работающие с текстом, являются оболочками классов, использующих байты, и автоматически выполняют перекодирование из байтов в символы и обратно. Двоичные и байтовые потоки хранят данные в том же виде, в котором они пред­ ставлены в оперативной памяти, то есть при обмене с файлом происходит поби­ товое копирование информации. Двоичные файлы применяются не для про­ смотра их человеком, а для использования в программах. Доступ к файлам может быть последовательным, когда очередной элемент можно прочитать (записать) только после аналогичной операции с предыдущим элемен­ том, и произвольным, или прямым, при котором выполняется чтение (запись) произвольного элемента по заданному адресу. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода. Прямой доступ в сочетании с отсутствием преобразований обеспечивает высо­ кую скорость получения нужной информации. ПРИМЕЧАНИ Е Методы форматированного ввода, с помощью которых можно выполнять ввод с кла виатуры или из текстового файла значений арифметических типов, в С# не под держиваются. Для преобразования из символьного в числовое представлен» используются методы класса Convert или метод Parse, рассмотренные в раздел- «Простейший ввод-вывод» (см. с. 59). ПРИМЕЧАНИ Е Форматированный вывод, то есть преобразование из внутренней формы предста! ления числа в символьную, понятную человеку, выполняется с помощью перегр] женных методов ToString, результаты выполнения которых передаются в метод текстовых файлов. Помимо перечисленных классов в библиотеке .NET есть классы XmlTextReadf и XmlTextWriter, предназначенные для формирования и чтения кода в форма' XML. Понятие об XM L дается в главе 15. Работа с файлами 249 Рассмотрим простейшие способы работы с файловыми потоками. Использова­ ние классов файловых потоков в программе предполагает следующие операции: 1. Создание потока и связывание его с физическим файлом. 2. Обмен (ввод-вывод). 3. Закрытие файла. Каждый класс файловых потоков содержит несколько вариантов конструкторов, с помощью которых можно создавать объекты этих классов различными спосо­ бами и в различных режимах. Например, файлы можно открывать только для чтения, только для записи или для чтения и записи. Эти режимы доступа к файлу содержатся в перечислении FileAccess, определенном в пространстве имен System. 10. Константы перечисле­ ния приведены в табл. 11.2. Таблица 11.2 . Значения перечисления FileAccess Значение Описание Read Открыть файл только для чтения ReadWrite Открыть файл для чтения и записи Write Открыть файл только для записи Возможные режимы открытия файла определены в перечислении FileMode (табл. 11.3). Таблица 11.3 . Значения перечисления FileMode Значение Описание Append Открыть файл, если он существует, и установить текущий указатель в конец файла. Если файл не существует, создать новый файл Create Создать новый файл. Если в каталоге уже существует файл с таким же именем, он будет стерт CreateNew Создать новый файл. Если в каталоге уже существует файл с таким же именем, возникает исключение IOException Open " Открыть существующий файл OpenOrCreate Открыть файл, если он существует. Если нет, создать файл с таким именем Truncate Открыть существующий файл. После открытия он должен быть обрезан до нулевой длины Режим FileMode.Append можно использовать только совместно с доступом типа FileAccess.Write, то есть для файлов, открываемых для записи. Режимы совместного использования файла различными пользователями опреде­ ляет перечисление FileShare (табл. 11.4). 250 Глава 11. Работа с файлами Таблиц а 11.4 . Значения перечисления FileShare Значение Описание None Совместное использование открытого файла запрещено. Запрос на открытие данного файла завершается сообщением об ошибке Read Позволяет открывать файл для чтения одновременно нескольким пользователям. Если этот флаг не установлен, запросы на открытие файла для чтения завершаются сообщением об ошибке ReadWrite Позволяет открывать файл для чтения и записи одновременно нескольким пользователям Wri te Позволяет открывать файл для записи одновременно нескольким пользователям Потоки байтов Ввод-вывод в файл на уровне байтов выполняется с помощью класса Fi I eStream, который является наследником абстрактного класса Stream, определяющего набор стандартных операций с потоками. Элементы класса Stream описаны в табл. 11.5. Таблица 11.5 . Элементы класса Stream Элемент Описание BeginRead, Начать асинхронный ввод или вывод BeginWrite CanRead, Свойства, определяющие, какие операции поддерживает поток: чтение, CanSeek, прямой доступ и/или запись CanWrite СЛ ose Закрыть текущий поток и освободить связанные с ним ресурсы (сокеты, указатели на файлы и т. п.) EndRead, Ожидать завершения асинхронного ввода; закончить асинхронный вывод EndWrite Flush Записать данные из буфера в связанный с потоком источник данных и очистить буфер. Если для данного потока буфер не используется, то этот метод ничего не делает Length Возвратить длину потока в байтах Position Возвратить текущую позицию в потоке Read, Считать последовательность байтов (или один байт) из текущего потока ReadByte и переместить указатель в потоке на количество считанных байтов Seek Установить текущий указатель потока на заданную позицию SetLength Установить длину текущего потока Wri te, Записать последовательность байтов (или один байт) в текущий поток WriteByte и переместить указатель в потоке на количество записанных байтов Потоки байтов 251 Класс FileStream реализует эти элементы для работы с дисковыми файлами. Для определения режимов работы с файлом используются стандартные перечис­ ления FileMode, FileAccess и FileShare. Значения этих перечислений приведены в табл. 11.2—11.4. В листинге 11.1 представлен пример работы с файлом. В при­ зере демонстрируются чтение и запись одного байта и массива байтов, а также позиционирование в потоке. Листин г 11.1 . Пример использования потока байтов using System; using System.10; namespace ConsoleApplication1 { class Classl { static void MainO . { FileStream f = new FileStream( "test.txt", FileMode.Create, FileAccess.ReadWrite ); f.WriteByte( 100 ); II ъ начало файла записывается число 100 byte[] х = new byte[10]; for ( byte i = 0; i < 10; ++i ) { x[1] = (byteM 10 - i ); f.WriteByte(i); // записывается 10 чисел от 0 до 9 } f.WriteC x, 0, 5 ); // записывается 5 элементов массива byte[] у = new byte[20]; f.Seek( 0, SeekOrigin.Begin ); // текущий указатель - на начало f.Read( у,. 0. 20 ); // чтение из файла в массив foreach С byte elem in у ) Consolе.WriteC " " + elem ); Console.WriteLineC); f.Seek(5, SeekOrigin.Begin); // текущий указатель - на 5-й элемент int а = f.ReadByteO; // чтение 5-го элемента Console.WriteLineC а ); а = f.ReadByteO; // чтение 6-го элемента Console.WriteLineC а ); Console.WriteLineC "Текущая позиция в потоке" + f.Position ); f.CloseO; } } } 252 Глава 11. Работа с файлами Результат работы программы: 100 0123 4 5678 9 1 0 9876000 0 4 5 Текущая позиция в потоке 7 Текущая позиция в потоке первоначально устанавливается на начало файла (для любого режима открытия, кроме Append) и сдвигается на одну позицию при записи каждого байта. Для установки желаемой позиции чтения используется метод Seek, имеющий два параметра: первый задает смещение в байтах относительно точки отсчета, зада­ ваемой вторым. Точки отсчета задаются константами перечисления SeekOrigin: начало файла — Begin, текущая позиция — Current и конец файла — End. В данном примере файл создавался в текущем каталоге. Можно указать и пол­ ный путь к файлу, при этом удобнее использовать дословные литералы, речь о которых шла в разделе «Литералы» (см. с. 30), например: FileStream f = new FileStreamC @"D:\CJ\test.txt", FileMode.Create, FileAccess.ReadWrite ): В дословных литералах не требуется дублировать обратную косую черту. Операции по открытию файлов могут завершиться неудачно, например, при ошиб­ ке в имени существующего файла или при отсутствии свободного места на дис­ ке, поэтому рекомендуется всегда контролировать результаты этих операций. В случае непредвиденных ситуаций среда выполнения генерирует различные ис­ ключения, обработку которых следует предусмотреть в программе, например: • FileNotFoundException, если файла с указанным именем в указанном каталоге не существует; • DirectoryNotFoundException, если не существует указанный каталог; • Argument Except ion, если неверно задан режим открытия файла; • IOException, если файл не открывается из-за ошибок ввода-вывода. Возможны и другие исключительные ситуации. Удобно обрабатывать наиболее вероятные ошибки раздельно, чтобы предоста­ вить пользователю программы в выводимом сообщении наиболее точную инфор­ мацию. В приведенном далее примере отдельно перехватывается ошибка в имени файла, а затем обрабатываются все остальные возможные ошибки: try { FileStream f = new FileStreamC @"d:\C#\test.tx", Fi1eMode.Open. FileAccess.Read ); // действия с файлом f.CloseO; } catchC FileNotFoundException e ) Асинхронный ввод-вывод 253 { Console.WriteLineC е.Message ); Console.WriteLineC "Проверьте правильность имени файла!" ); return; } catch( Exception e ) { Console.WriteLineC "Error: " + e.Message ); return; } При закрытии файла освобождаются все связанные с ним ресурсы, например, для файла, открытого для записи, в файл выгружается содержимое буфера. По­ этому рекомендуется всегда закрывать файлы после окончания работы, в осо­ бенности файлы, открытые для записи. Если буфер требуется выгрузить, не за­ крывая файл, используется метод Flush. Асинхронный ввод-вывод Класс Stream (и, соответственно, FileStream) поддерживает два способа выполнения операций ввода-вывода: синхронный и асинхронный. По умолчанию файлы от­ крываются в синхронном режиме, то есть последующие операторы выполняются только после завершения операций ввода-вывода. Для длительных файловых опе­ раций более эффективно выполнять ввод-вывод асинхронно, в отдельном потоке выполнения 1 . При этом в первичном потоке можно выполнять другие операции. Для асинхронного ввода-вывода необходимо открыть файл в асинхронном режи­ ме, для этого используется соответствующий вариант перегруженного конструк­ тора. Асинхронная операция ввода инициируется с помощью метода BeginRead. Помимо характеристик буфера, в который выполняется ввод, в этот метод пере­ дается делегат, задающий метод, выполняемый после завершения ввода. Этот метод может инициировать обработку полученной информации, возобно­ вить операцию чтения или выполнить любые другие действия, например, прове­ рить успешность ввода и сообщить о его завершении. Обычно в этом методе вы­ зывается метод EndRead, который завершает асинхронную операцию. Аналогично выполняется и асинхронный вывод. В листинге 11.2 приведен при­ мер асинхронного чтения из файла большого объема и параллельного выполне­ ния диалога с пользователем. ПРИМЕЧАНИ Е Вообще говоря, существуют различные способы завершения асинхронных опера­ ций, и здесь демонстрируется только один из них. 1 Работа с потоками выполнения рассматривалась в предыдущей главе. 254 Глава 11. Работа с файлами Листин г 11.2 . Асинхронный ввод using System; using System.10; using System.Threading; namespace ConsoleApplicationl { class Demo { public void UserlnputO // диалог с пользователем { string s; do { Console.WriteLineC "Введите строку. Enter для завершения" ); s = Console.ReadLineO; } while (s.Length != 0 ); } public void OnCompletedReadC IAsyncResult ar ) .// 1 { int bytes = f.EndRead( ar ); Console.WriteLineC "Считано " + bytes ); } public void AsyncReadO { f = new FileStreamC "D:\\verybigfile". FileMode.Open. FileAccess.Read, FileShare.Read. buf.Length, true ); 111 callback = new AsyncCallbackC OnCompletedRead ): 1/3 f.BeginReadC buf, 0. buf.Length, callback, null ); // 4 } FileStream f; byte[] buf = new byte[66666666]; AsyncCallback callback; } class Program { static void MainO { Demo d = new DemoO; d. AsyncReadO; d.UserlnputO; } } } Потоки символов 255 Для удобства восприятия операции чтения из файла и диалога с пользователем оформлены в отдельный класс Demo. Метод OnCompletedRead (оператор 1) должен получать один параметр стандартно­ го типа IAsyncResult, содержащий сведения о завершении операции, которые пе­ редаются в метод EndRead. Файл открывается в асинхронном режиме, об этом говорит значение true по­ следнего параметра конструктора (оператор 2). В операторе 3 создается экзем­ пляр стандартного делегата AsyncCallback, который инициализируется методом OnCompletedRead. С помощью этого делегата метод OnCompletedRead передается в метод'BeginRead (опе­ ратор 4), который создает отдельный поток, начинает асинхронный ввод и воз­ вращает управление в вызвавший поток. Обратный вызов метода OnCompletedRead происходит при завершении операции ввода. При достаточно длинном файле verybigfile можно убедиться, что приглашение к вводу в методе Userlnput выдается раньше, чем сообщение о завершении операции ввода из метода OnCompletedRead. ПРИМЕЧАНИ Е Пример, приведенный в листинге 11.2, максимально упрощен для демонстрации методов BeginRead и EndRead, поэтому в нем нет необходимых в любой программе проверок наличия файла, успешности считывания и т. д. Потоки символов Символьные потоки StreamWriter и StreamReader работают с Unicode-символами1 , следовательно, ими удобнее всего пользоваться для работы с файлами, предна­ значенными для восприятия человеком. Эти потоки являются наследниками классов TextWriter и TextReader соответственно, которые обеспечивают их боль­ шей частью функциональности. В табл. 11.6 и 11.7 приведены наиболее важные элементы этих классов. Как видите, произвольный доступ для текстовых файлов не поддерживается. Таблица 11.6 . Наиболее важные элементы базового класса TextWriter Элемент Описание Close Закрыть файл и освободить связанные с ним ресурсы. Если в процессе записи используется буфер, он будет автоматически очищен Fl ush Очистить все буферы для текущего файла и записать накопленные в них данные в место их постоянного хранения. Сам файл при этом не закрывается продолжение & 1 Существует возможность изменить используемую кодировку с помощью объекта System. Text.Encoding. 256 Глава 11. Работа с файлами Таблиц а 11. 6 {продолжение) Элемент Описание NewLine Используется для задания последовательности символов, означающих начало новой строки. По умолчанию используется последовательность «возврат каретки» — «перевод строки» (\г\п) Write Записать фрагмент текста в поток Wri teLi пе Записать строку в поток и перейти на другую строку Таблиц а 11.7 . Наиболее важные элементы класса TextReader Элемент Описание Peek Возвратить следующий символ, не изменяя позицию указателя в файле Read Считать данные из входного потока ReadBlock Считать из входного потока указанное пользователем количество символов и записать их в буфер, начиная с заданной позиции ReadLine Считать строку из текущего потока и возвратить ее как значение типа stri ng. Пустая строка (nul 1) означает конец файла (EOF) ReadToEnd Считать все символы до конца потока, начиная с текущей позиции, и возвратить считанные данные как одну строку типа stri ng Вы уже знакомы с некоторыми методами, приведенными в этих таблицах: на протяжении всей книги постоянно использовались методы чтения из текстовых потоков и записи в текстовые потоки, но не для дисковых файлов, а для консоли, которая является их частным случаем. В листинге 11.3 создается текстовый файл, в который записываются две строки. Вторая строка формируется из преобразованных численных значений пере­ менных и поясняющего текста. Содержимое файла можно посмотреть в любом текстовом редакторе. Файл создается в том же каталоге, куда среда записывает исполняемый файл. По умолчанию это каталог ...\ConsoleApplication1\bin\Debug. Листин г 11.3 . Вывод в текстовый файл using System; using System.10; namespace ConsoleApplicationl { class Classl { static void MainO { try { StreamWriter f = new StreamWriterC "text.txt" ); f.WriteLineC "Вывод в текстовый файл:" ); double а = 12.234; Потоки символов 257 int b = 29: f.WriteLineC " a = {0.6:C} b = {1.2:X}\ a. b ); f.CloseO; catch( Exception e ) { Console.WriteLineC "Error: " + e.Message ); return; } } В листинге 11.4 файл, созданный в предыдущем листинге, выводится на экран. Листин г 11.4 . Чтение текстового файла using System; using System.10; namespace ConsoleApplicationi { class Classl { static void MainO { try { StreamReader f = new StreamReaderC "text.txt" ); string s = f.ReadToEndO; Console.WriteLine(s); f.CloseO: catchC FileNotFoundException e ) Console.WriteLineC e.Message ); Console.WriteLineC " Проверьте правильность имени файла!" ): return; catch( Exception e ) Console.WriteLineC "Error: return; + e.Message ); 258 Глава 11. Работа с файлами В этой программе весь файл считывается за один прием с помощью метода ReadToEnd. Чаще возникает необходимость считывать файл построчно, такой при­ мер приведен в листинге 11.5. Каждая строка при выводе предваряется номером. Листин г 11.5 . Построчное чтение текстового файла using System; using System.10; namespace ConsoleApplicationl { class Classl { static void MainO { try { StreamReader f = new StreamReaderC "text.txt" ); string s; long i = 0: while ( ( s = f.ReadLineO ) != null ) Console.WriteLineC "{0}: {1}". ++i, s ); f.CloseO; } catch( FileNotFoundException e ) { Console.WriteLineC e.Message ); Console.WriteLineC "Проверьте правильность имени файла!" ); return; } catch С Exception e ) { Console.WriteLineC "Error: " + e.Message ); return; } } } } Пример преобразования чисел, содержащихся в текстовом файле, в их внутре] нюю форму представления приведен в листинге 11.6. В программе вычисляет\* сумма чисел в каждой строке. На содержимое файла накладываются весьма строгие ограничения: числа дол: ны быть разделены ровно одним пробелом, после последнего числа в строке пр бела быть не должно, файл не должен заканчиваться символом перевода стро\* Методы разбиения строки и преобразования в целочисленное представлен рассматривались ранее. Потоки символов 259 Листин г 11.6 . Преобразования строк в числа using System; using System.10; namespace ConsoleApplicationi { class Classl { static void MainO { try { StreamReader f = new StreamReaderC "numbers.txt" ); string s: const int n = 20; int[] a = new int[n]; string[] buf; while С ( s = f.ReadLineO ) != null ) { buf = s.SplitO ') ; long sum = 0; for ( int i = 0; i < buf.Length; ++i ) { a[i ] = Convert.ToInt32( buf[i] ); sum += a[i] ; } Console.WriteLineC "{0} сумма; {1}0 s. sum ); } f.CloseO; } catch( FileNotFoundException e ) { Console.WriteLineC e.Message ); Console.WriteLineC " Проверьте правильность имени файла!" ): return; } catch С Exception e ) { Console.WriteLineC "Error; " + e.Message ): return; } } } } Результат работы программы: 12 4 сумма: 7 3 44 -3 б сумма: 50 8 11 сумма: 10 260 Глава 11. Работа с файлами Двоичные потоки Двоичные файлы хранят данные в том же виде, в котором они представлены в опе­ ративной памяти, то есть во внутренней форме представления. Двоичные файлы применяются не для просмотра их человеком, а для использования в программах. Выходной поток BinaryWriter поддерживает произвольный доступ, то есть име­ ется возможность выполнять запись в произвольную позицию двоичного файла. Двоичный файл открывается на основе базового потока, в качестве которого чаще всего используется поток FileStream. Входной двоичный поток содержит пере­ груженные методы чтения для всех простых встроенных типов данных. Основные методы двоичных потоков приведены в табл. 11.8 и 11.9. Таблиц а 11.8 . Наиболее важные элементы класса BinaryWriter Элемент Описание BaseStream Базовый поток, с которым работает объект BinaryWriter Close Закрыть поток Flush Очистить буфер I Seek Установить позицию в текущем потоке j Write Записать значение в текущий поток I Таблиц а 11.9 . Наиболее важные элементы класса BinaryReader Элемент Описание BaseStream Базовый поток, с которым работает объект BinaryReader Close Закрыть поток PeekChar Возвратить следующий символ без перемещения внутреннего указателя в потоке ' Read Считать поток байтов или символов и сохранить в массиве, передаваемом как входной параметр ReadXXXX Считать из потока данные определенного типа (например, ReadBoolean, ReadByte, Readlnt32 и т. д.) В листинге 11.7 приведен пример формирования двоичного файла. В файл заи сывается последовательность вещественных чисел, а затем для демонстрац произвольного доступа третье число заменяется числом 8888. Листин г 11.7 . Формирование двоичного файла using System; using System.10; namespace ConsoleApplicationl { class Classl { static void MainO Двоичные потоки try { BinaryWriter tout = new BinaryWriter( new FileStreamC @"D:\C#\binary", FileMode.Create) ); double d = 0; while ( d < 4 ) { fout.WriteC d ); d += 0.33; }: fout.Seek( 16. SeekOrigin.Begin ); // второй элемент файла fout.Write( 8888d ); fout.CloseC); } catchC Exception e ) { Console.WriteLineC "Error: " + e.Message ); return: } } } } При создании двоичного потока в него передается объект базового потока. При установке указателя текущей позиции в файле учитывается длина каждого зна­ чения типа doubl е — 8 байт. Попытка просмотра сформированного программой файла в текстовом редакторе весьма медитативная, но не информативная, поэтому в листинге 11.8 приводится программа, которая с помощью экземпляра BinaryReader считывает содержимое файла в массив вещественных чисел, а затем выводит этот массив на экран. При чтении принимается во внимание тот факт, что метод ReadDoubl е при обна­ ружении конца файла генерирует исключение EndOfStreamException. Поскольку в данном случае это не ошибка, тело обработчика исключений пустое. Листин г 11.8 . Считывание двоичного файла using System; using System.10; namespace ConsoleAppl icationi { class Classl { static void MainO { продолжение ^ try 262 Глава 11. Работа с файлами Листин г 11. 8 {продолжение) ' { FileStream f = new FileStreamC @"D:\C#\binary", Fi1eMode.Open ); BinaryReader fin = new BinaryReaderC f ); long n = f.Length / 8; // количество чисел в файле doublet] х = new double[n]: long i = 0: try { whileC true ) x[i++] = fin.ReadDoubleO; // чтение } catch ( EndOfStreamException e ) {} foreachC double d in x ) Console.WriteC " " + d ); // вывод fin.CloseO; f.CloseO; catch ( FileNotFoundException e ) Console.WriteLineC e.Message ); Console.WriteLineC "Проверьте правильность имени файла!" ); return; catch ( Exception e ) Console.WriteLineC "Error: " + e.Message ); return; } } } Результат работы программы: 0 0,33 8888 0,99 1.32 1,65 1,98 2,31 2,64 2,97 3,3 3,63 3,96 Консольный ввод-вывод Консольные приложения имеют весьма ограниченную область применения, са­ мой распространенной из которых является обучение языку программирования. Для организации ввода и вывода используется известный вам класс Console, оп­ ределенный в пространстве имен System. В этом классе определены три стандартных Работа с каталогами и файлами 263 потока: входной поток Console. In класса TextReader и выходные потоки Consol е. Out и Console.Error класса TextWriter. По умолчанию входной поток связан с клавиатурой, а выходные — с экраном, однако можно перенаправить эти потоки на другие устройства с помощью мето­ дов Set In и SetOut или средствами операционной системы (перенаправление с по­ мощью операций и ») . При обмене с консолью можно применять методы указанных потоков, но чаще используются методы класса Console — Read, ReadLine, Write и WriteLine, которые просто передают управление методам нижележащих классов In, Out и Error. Использование не одного, а двух выходных потоков полезно при желании разде­ лить нормальный вывод программы и её сообщения об ошибках. Например, нор­ мальный вывод программы можно перенаправить в файл, а сообщения об ошиб­ ках — на консоль или в файл журнала. Работа с каталогами и файлами В пространстве имен System. 10 есть четыре класса, предназначенные для рабо­ ты с физическими файлами и структурой каталогов на диске: Directory, File, Directorylnfo и Filelnfo. С их помощью можно выполнять создание, удаление, перемещение файлов и каталогов, а также получение их свойств. Классы Directory и File реализуют свои функции через статические методы. Directorylnfo и Filelnfo обладают схожими возможностями, но они реализу­ ются путем создания объектов соответствующих классов. Классы Directorylnfo и Filelnfo происходят от абстрактного класса FileSystemlnfo, который снабжает их базовыми свойствами, описанными в табл. 11.10. Таблица 11.10 . Свойства класса FileSystemlnfo Свойство Описание Attributes Получить или установить атрибуты для данного объекта файловой системы. Для этого свойства используются значения перечисления FileAttributes CreationTime Получить или установить время создания объекта файловой системы Exists Определить, существует ли данный объект файловой системы Extension Получить расширение файла Full Name Возвратить имя файла или каталога с указанием полного пути LastAccessTime Получить или установить время последнего обращения к объекту файловой системы LastWriteTime Получить или установить время последнего внесения изменений в объект файловой системы Name Возвратить имя файла. Это свойство доступно только для чтения. Для каталогов возвращает имя последнего каталога в иерархии, если это возможно. Если нет, возвращает полностью определенное имя 264 Глава 11. Работа с файлами Класс Directorylnfo содержит элементы, позволяющие выполнять необходимьн действия с каталогами файловой системы. Эти элементы перечислены в табл. 11.11 Таблиц а 11.11 . Элементы класса Directorylnfo Элемент Описание Create, CreateSubDirectory Delete GetDirectories GetFiles MoveTo Parent Создать каталог или подкаталог по указанному пути в файловой системе Удалить каталог со всем его содержимым Возвратить массив строк, представляющих все подкаталоги Получить файлы в текущем каталоге в виде массива объектов класса Filelnfo Переместить каталог и все его содержимое на новый адрес в файловой системе Возвратить родительский каталог В листинге 11.9 приведен пример, в котором создаются два каталога, выводит информация о них и предпринимается попытка удаления каталога. Листин г 11.9 . Использование класса Directorylnfo using System; using System.10; namespace ConsoleApplicationl { class Classl { static void DirlnfoC Directorylnfo di { // Вывод информации Console.WriteLineC Console.WriteLineC Console.WriteLineC Console.WriteLineC Console.WriteLineC Console.WriteLineC , Console.WriteLineC Console.WriteLineC ) о каталоге ===== Directory Info ): Full Name: " Name: " Parent: " Creation: " Attributes: Root: " + di.FullName ); + di.Name ); + di.Parent ); + di.CreationTime ); + di.Attributes ); + di.Root ); ===== ); } static void MainO { Directorylnfo di1 Directorylnfo di2 try { // Создать каталоги dil.CreateO: new DirectoryInfo( @"c:\MyDir" ); new DirectorylnfoC @"c:\MyDir\temp" ); Работа с каталогами и файлами 265 di2.Create(); // Вывести информацию о каталогах Dirlnfo(dil): DirInfo(di2); // Попытаться удалить каталог Console.WriteLineC "Попытка удалить {0}." , dil.Name ) : dil.DeleteO; } catch С Exception ) { Console.WriteLineC "Попытка не удалась " ); } } } } Результат работы программы: ===== Directory Info ===== Full Name: c:\MyDir Name: MyDir Parent: Creation: 30.04.2006 17:14:44 Attributes: Directory Root: c:\ ===== Directory Info ===== Full Name: c:\MyDir\temp Name: temp Parent: MyDir Creation: 30.04.2006 17:14:44 Attributes: Directory Root: c:\ Попытка удалить MyDir. Попытка не удалась Каталог не пуст, поэтому попытка его удаления не удалась. Впрочем, если ис­ пользовать перегруженный вариант метода Delete с одним параметром, задаю­ щим режим удаления, можно удалить и непустой каталог: dil.DeleteC true ); // удаляет непустой каталог Обратите внимание на свойство Attributes. Некоторые его возможные значения, заданные в перечислении FileAttributes, приведены в табл. 11.12. 266 Глава 11. Работа с файлами Таблица 11,12 . Некоторые значения перечисления FileAttributes Значение Описание Archive Используется приложениями при выполнении резервного копирования, а в некоторых случаях — при удалении старых файлов Compressed Файл является сжатым Directory Объект файловой системы является каталогом Encrypted Файл является зашифрованным Hidden Файл является скрытым Normal Файл находится в обычном состоянии, и для него установлены любые другие атрибуты. Этот атрибут не может использоваться с другими атрибутами Offline Файл, расположенный на сервере, кэширован в хранилище на клиентском компьютере. Возможно, что данные этого файла уже устарели Readonly Файл доступен только для чтения System Файл является системным Листинг 11.10 демонстрирует использование класса Filelnfo для копирования всех файлов с расширением jpg из каталога d:\foto в каталог d:\temp. Метод Exists позволяет проверить, существует ли исходный каталог. Листин г 11.10 . Копирование файлов using System; using System.10; namespace Consol eApplicationi { •class Classl { static void MainO { try { string DestName = @"d:\tempV; Directorylnfo dest = new DirectoryInfо( DestName ): dest.CreateO; // создание целевого каталога Directorylnfo dir = new DlrectorylnfoC @"d:\foto" ); if ( ! dir.Exists ) // проверка существования каталога { Console.WriteLineC "Каталог " + dir.Name + " не существует" ); return; } FilelnfoD files = dir.GetFiles( "\*.jpg" ); //список файлов Сохранение объектов (сериализация) 267 foreachC Filelnfo f in files ) f.CopyToC dest + f.Name ); // копирование файлов Console.WriteLineC "Скопировано " + fi1es.Length + " jpg-файлов" ); } catch ( Exception e ) { Console.WriteLineC "Error: " + e.Message ); }• } } } Использование классов Fi 1 e и Di rectory аналогично, за исключением того, что их методы являются статическими и, следовательно, не требуют создания объектов. Сохранение объектов (сериализация) В С# есть возможность сохранять на внешних носителях не только данные при­ митивных типов, но и объекты. Сохранение объектов называется сериализацией, а восстановление сохраненных объектов — десериализацией. При сериализации объект преобразуется в линейную последовательность байтов. Это сложный процесс, поскольку объект может включать множество унаследованных полей и ссылки на вложенные объекты, которые, в свою очередь, тоже могут состоять из объектов сложной структуры. К счастью, сериализация выполняется автоматически, достаточно просто пометить класс как сериализуемый с помощью атрибута [Serial izable]. Атрибуты рассмат­ риваются в главе 12, пока же достаточно знать, что атрибуты — это дополнитель­ ные сведения о классе, которые сохраняются в его метаданных. Те поля, которые сохранять не требуется, помечаются атрибутом [NonSerialized], например: [Serializable] class Demo { public int a = 1: [NonSerialized] public double y: public Monster X. Y; } Объекты можно сохранять в одном из двух форматов: двоичном или SOAP (в виде XML-файла). В первом случае следует подключить к программе пространство имен System.Runtime.Serialization.Formatters.Binary, во втором — пространство System.Runtime.Seri alizati on.Formatters.Soap. 268 Глава 11. Работа с файлами Рассмотрим сохранение объектов в двоичном формате. Для этого используется класс BinaryFormatter, в котором определены два метода: Serialize( поток, объект ); DeserializeC поток ): Метод Serialize сохраняет заданный объект в заданном потоке, метод Deserialize восстанавливает объект из заданного потока. В листинге 11.11 объект приведенного ранее класса Demo сохраняется в файле на диске с именем Demo.bin. Этот файл можно просмотреть, открыв его, к примеру, в Visual Studic.NET. Листин г 11.11 . Сериализация объекта using System: using System.10; us i ng System.Runti me.Seri a1i zat i on.Formatters.Bi na ry; namespace ConsoleApplicationi { [Serializable] abstract class Spirit { public abstract void PassportO; } [Serializable] class Monster : Spirit { public Monster( int health, int ammo, string name ) { this.health = health; this.ammo - ammo; this.name = name: } override public void PassportO { Console.WriteLineC "Monster {0} \t health = {1} ammo = {2}", name, health, ammo ); } string name; int health, ammo; } [Serial izable] class Demo { public int a = 1; Сохранение объектов (сериализация) 269 [NonSerialized] public double b; public Monster X, Y; } class Classl • { static void MainO { Demo d = new DemoO; d.X = new Monster( 100. 80. "Вася" ); d.Y = new MonsterC 120, 50, "Петя" ); d.a - 2; d.b = 2; d.X. PassportO; d. Y. PassportO; Console.WriteLineC d.a ); Console.WriteLineC d.b ); FileStream f = new FileStreamC "Demo.bin", FileMode.Create ); BinaryFormatter bf = new BinaryFormatterO; bf.Serialize( f, d ); // сохранение объекта d в потоке f f.CloseO: } } } В программе не приведены неиспользуемые методы и свойства классов. Обрати­ те внимание на то, что базовые классы сохраняемых объектов также должны быть помечены как сохраняемые. Результат работы программы: Monster Вася health = 100 ammo = 80 Monster Петя health = 120 ammo = 50 2 2 Итак, для сохранения объекта в двоичном формате необходимо: 1. Подключить к программе пространство имен System.Runtime.Serialization. Formatters.Binary. 2. Пометить сохраняемый класс и связанные с ним классы атрибутом [Serializable]. 3. Создать поток и связать его с файлом на диске или с областью оперативной памяти. 4. Создать объект класса BinaryFormatter. 5. Сохранить объекты в потоке. 6. Закрыть файл. В листинге 11.12 сохраненный объект считывается из файла. 270 Глава 11. Работа с файлами Листин г 11.12 . Десериапизация объекта using System; using System.10; using System.Runtime.Serialization.Formatters.Binary; namespace Consol eAppl icationi { class Classl { static void MainO { FileStream f = new FileStream( "Demo.bin", FileMode.Open ); BinaryFormatter bf = new BinaryFormatterO; Demo d = (Demo) bf.DeserializeC f ); // восстановление объекта d.X. PassportO; d.Y. Passport О ; Console.WriteLineC d.a ); Console.WriteLineC d.b ); f.CloseO; } } } Результат работы программы: Monster Вася health = 100 ammo = 80 Monster Петя health = 120 ammo = 50 2 0 Как видите, при сериализации сохраняется все дерево объектов. Обратите вни­ мание на то, что значение поля у не было сохранено, поскольку оно было помече­ но как несохраняемое. ПРИМЕЧАНИ Е Сериализация в формате SOAP выполняется аналогично с помощью класса SoapFormatter. Программист может задать собственный формат сериализации, для этого ему придется реализовать в своих классах интерфейс ISeri al i zabl e и специ­ альный вид конструктора класса. Рекомендации по программированию Большинство программ тем или иным образом работают с внешними устройства­ ми, в качестве которых могут выступать, например, консоль, файл на диске или сетевое соединение. Взаимодействие с внешними устройствами организуется с по­ мощью потоков, которые поддерживаются множеством классов библиотеки .NET. Рекомендации по программированию 271 Поток определяется как последовательность байтов и не зависит от конкретного устройства, с которым производится обмен. Классы библиотеки позволяют ра­ ботать с потоками в различных режимах и на различных уровнях: на уровне дво­ ичного представления данных, байтов и текста. Двоичные и байтовые потоки хранят данные во внутреннем представлении, текстовые — в кодировке Unicode. Поток можно открыть в синхронном или асинхронном режиме для чтения, записи или добавления. Доступ к файлам может быть последовательным и произволь­ ным. Текстовые файлы позволяют выполнять только последовательный доступ, в двоичных и байтовых потоках можно использовать оба метода. Прямой доступ в сочетании с отсутствием преобразований обеспечивает высокую скорость обмена. Методы форматированного ввода для значений арифметических типов в С# не поддерживаются. Для преобразования из символьного в числовое представление используются методы класса Convert или метод Parse. Форматированный вывод выполняется с помощью перегруженного метода ToString, результат выполнения которого передается в методы текстовых файлов. Рекомендуется всегда проверять успешность открытия существующего файла, пе­ рехватывать исключения, возникающие при преобразовании значений арифмети­ ческих типов, и явным образом закрывать файл, в который выполнялась запись. Длительные операции с файлами более эффективно выполнять в асинхронном режиме. Для сохранения объектов (сериализации) используется атрибут [Serializable]. Объекты можно сохранять в одном из двух форматов: двоичном или SOAP (в виде XML-файла). Глава 12 Сборки, библиотеки, атрибуты, директивы Все наши предыдущие приложения состояли из одного физического файла. Для больших проектов это неудобно и чаще всего невозможно, да и в других случаях бывает удобнее поместить связанные между собой типы в библиотеку и исполь­ зовать их по мере необходимости. В этой главе мы рассмотрим вопросы созда­ ния и использования библиотек, способы получения и подготовки информации о типах, пространства имен и препроцессор. Эти сведения необходимы для ус­ пешной разработки реальных программ. Сборки В результате компиляции в среде .NET создается сборка — файл с расширением ехе или dll, который содержит код на промежуточном языке, метаданные типов, ма­ нифест и ресурсы (рис. 12.1). Понятие сборки было введено в главе 1 (см. с. 9), а сейчас мы рассмотрим ее составные части более подробно. Манифест Метаданные Код на языке IL Ресурсы (не обязательно) Рис. 12.1 . Сборка, состоящая из одного файла борки 273 Тромежуточный язык (Intermediate Language, IL ) не содержит инструкций, ависящих от операционной системы и типа компьютера, что обеспечивает две юновные возможности: 1 выполнение приложения на любом типе компьютера, для которого существу­ ет среда выполнения CLR; 2 повторное использование кода, написанного на любом .NET-совместимом языке. [L-код можно просмотреть с помощью дизассемблера ILDasm.exe, который нахо­ дится в папке ...\SDK\bin\ каталога размещения Visual Studio.NET. После за- туска ILDasm можно открыть любой файл среды .NET с расширением ехе или dll ; помощью команды File • Open. В окне программы откроется список всех эле­ ментов сборки, сведения о каждом можно получить двойным щелчком. При этом открывается окно, в котором для методов выводится доступный для восприятия дисассемблированный код. • Метаданные типов — это сведения о типах, используемых в сборке. Компилятор создает метаданные автоматически. В них содержится информация о каждом типе, имеющемся в программе, и о каждом его элементе. Например, для каждого класса описываются все его поля, методы, свойства, события, базовые классы и интерфейсы. Среда выполнения использует метаданные для поиска определений типов и их элементов в сборке, для создания экземпляров объектов, проверки вызова ме­ тодов и т. д. Компилятор, редактор кода и средства отладки также широко ис­ пользуют метаданные, например, для вывода подсказок и диагностических со­ общений. Манифест — это набор метаданных о самой сборке, включая информацию обо всех файлах, входящих в состав сборки, версии сборки, а также сведения обо всех внешних сборках, на которые она ссылается. Манифест создается компи­ лятором автоматически, программист может дополнять его собственными ат­ рибутами. Чаще всего сборка состоит из единственного файла, однако она может вклю­ чать и несколько физических файлов (модулей). В этом случае манифест либо включается в состав одного из файлов, либо содержится в отдельном файле. Многофайловые сборки используются для ускорения загрузки приложения — это имеет смысл для сборок большого объема, работа с которыми производится удаленно. На логическом уровне сборка представляет собой совокупность взаимосвязан­ ных типов — классов, интерфейсов, структур, перечислений, делегатов и ресур­ сов. Библиотека .NET представляет собой совокупность сборок, которую исполь­ зуют приложения. Точно так же можно создавать и собственные сборки, которые можно будет задействовать либо в рамках одного приложения (частные сборки), либо совместно различными приложениями (открытые сборки). По умолчанию все сборки являются частными. 274 Глава 12. Сборки, библиотеки, атрибуты, директивы Манифест сборки содержит: • идентификатор версии; • список всех внутренних модулей сборки; • список внешних сборок, необходимых для нормального выполнения сборки; • информацию о естественном языке, используемом в сборке (например, рус­ ском); • «сильное» имя (strong name) — специальный вариант имени сборки, исполь­ зуемый для открытых сборок; • необязательную информацию, связанную с безопасностью; • необязательную информацию, связанную с хранением ресурсов внутри сбор­ ки (подробнее о форматах ресурсов .NET см. [27]). Идентификатор версии относится ко всем элементам сборки. Он позволяет из­ бегать конфликтов имен и поддерживать одновременное существование и ис­ пользование различных версий одних и тех же сборок. Идентификатор версии состоит из двух частей: информационной версии в виде текстовой строки и версии совместимости в виде четырех чисел, разделенных точками: • основной номер версии (major version); • дополнительный номер версии (minor version); • номер сборки (build number); • номер ревизии (revision number). Среда выполнения применяет идентификатор версий для определения того, какие из открытых сборок совместимы с требованиями клиента. Например, если клиент запрашивает сборку 3.1.0.0, а присутствует только версия 3.4.0.0, сборка не будет опознана как подходящая, поскольку считается, что в дополнительных версиях могут произойти изменения в типах и их элементах. Разные номера ревизии допускают, но не гарантируют совместимость. Номер сборки на совместимость не влияет, так как чаще всего он изменяется при установке заплатки, или патча (patch). Идентификатор версии формируется автоматически, но при желании можно за­ дать его вручную с помощью атрибута [AssemblyVersion], который рассматривает­ ся далее на с. 285. Информация о безопасности позволяет определить, предоставить ли клиенту доступ к запрашиваемым элементам сборки. В манифесте сборки определены ог­ раничения системы безопасности. Ресурсы представляют собой, например, файлы изображений, помещаемых на форму, текстовые строки, значки приложения и т. д. Хранение ресурсов внут­ ри сборки обеспечивает их защиту и упрощает развертывание приложения. Среда Visual Studio.NET предоставляет возможности автоматического внедре­ ния ресурсов в сборку. Открытые и частные сборки различаются по способам размещения на компьюте­ ре пользователя, именованию и политике версий. Частные сборки должны нахо­ диться в каталоге приложения, использующего сборку, или в его подкаталогах. Создание библиотеки 275 Открытые сборки размещаются в специальном каталоге, который называется глобальным кэшем сборок (Global Assembly Cache, GAC) . Для идентификации открытой сборки используется уже упоминавшееся сильное имя (strong name), которое должно быть уникальным. "' Создание библиотеки Для создания библиотеки следует при разработке проекта в среде Visual Studio.NET выбрать шаблон Class Library (библиотека классов). В главе 8 была создана про­ стая иерархия классов персонажей компьютерной игры. В этом разделе мы офор­ мим ее в виде библиотеки, то есть сборки с расширением dll. Для сборки задано имя MonsterLib (рис. 12.2). Рис. 12.2 . Создание библиотеки Текст модуля приведен в листинге 12.1. По сравнению с модулем из главы 8 в него добавлены спецификаторы доступа public для всех трех классов, входящих в биб­ лиотеку. Листин г 12.1 . Библиотека монстров namespace MonsterLib { using System; public abstract class Spirit { public abstract void PassportO; продолжение & 276 Глава 121 . Сборки, библиотеки, атрибуты, директивы Листин г 12.1 (продолжение) public class Monster : Spirit { public Monster'O this.health = 100; this.ammo = 100: this.name = "Noname"; public MonsterC string name ) : thisO { this.name = name; } public Monster( int health, int ammo, string name ) { this.health = health; this.ammo = ammo; this.name = name; } public int Health { get { return health; } set { if ( value > 0 ) health = value; else health -= 0; } } public int Ammo { get { return ammo; } set { if ( value > 0 ) ammo = value; else ammo = 0; } } public string Name { дание библиотеки 277 pet { • return inSTfe- , } } override public void PassportO { Console.WriteLineC "Monster {0} \t health = {1} ammo = {2}" , name, health, ammo ); } string name; int health, ammo; } public class Daemon : Monster { public Daemon() brain =1 ; public DaemonC string name, int brain ) : baseC name ) this.brain = brain; public DaemonC int health, int ammo, string name, int brain ) : baseC health, ammo, name ) this.brain = brain; override public void PassportO Console.WriteLineC "Daemon {0} \t health = {1} ammo = {2} brain = {3}" . Name, Health, Ammo, brain ); public void ThinkO Console.WriteC Name + " is" ); for ( int i = 0; i < brain; ++i ) Console.WriteC " thinking" ); Console.WriteLineC "...") ; int brain; } 278 Глава 12. Сборки, библиотеки, атрибуты, директивы Скомпилировав библиотеку, вы обнаружите файл MonsterLib.dll в каталогах ...\bin\ Debug и ...\obj\Debug. Открыв файл MonsterLib.dll с помощью программы ILDasm.exe, можно получить полную информацию о созданной библиотеке (рис. 12.3). Рис. 12.3. Просмотр библиотеки с помощью дизассемблера ILDasm.exe Использование библиотеки Любая библиотека — это сервер, предоставляющий свои ресурсы клиентам. Соз­ дадим клиентское приложение, выполняющее те же функции, что и приложение из раздела «Виртуальные методы» (см. с. 178), но с использованием библиотеки MonsterLib.dll. Для того чтобы компилятор мог ее обнаружить, необходимо после создания проекта (как обычно, это — консольное приложение) подключить ссылку на библиотеку с помощью команды Project • Add Reference (Добавить ссылку). Для поиска каталога, содержащего библиотеку, следует использовать кнопку Browse. После подключения библиотеки можно пользоваться ее открытыми элементами таким же образом, как если бы они были описаны в том же модуле. Текст прило­ жения приведен в листинге 12.2. Рефлексия 279 Листин г 12.2 . Клиентское приложение using System; - namespace ConsoleApplicationi { using MonsterLib: class Classl { static void MainO { const int n = 3; MonsterC] stado = new Monster[n]: stado[0] = new MonsterC "Monia" ); stadoll] = new MonsterC "Monk" ); stado[2] = new Daemon ( "Dimon", 3 ); foreach ( Monster elem in stado ) elem. PassportO; for ( int i = 0; i < n; ++i ) stado[i].Ammo = 0; Console.writeLineO; foreach ( Monster elem in stado ) elem.PassportO; } } } Результаты работы программы совпадают с полученными в листинге 8.3. Ана­ лиз каталога ...\bin\Debug показывает, что среда создала в нем копию библиотеки MonsterLib.dll, то есть поместила библиотеку в тот же каталог, что и исполняемый файл. Если скопировать эти два файла в другое место, программа не потеряет своей работоспособности — главное, чтобы оба файла находились в одном каталоге. Допускается также, чтобы частные сборки находились в подкаталогах основного каталога приложения. ПРИМЕЧАНИ Е \* Преимущество .NET состоит в том, что благодаря стандартным соглашениям мож­ но использовать библиотеки независимо от языка, на котором они были написаны. Таким образом, можно было бы написать клиентское приложение, например, на языке VB.NET. Рефлексия Рефлексия — это получение информации о типах во время выполнения програм­ мы. Например, можно получить список всех классов и интерфейсов сборки, спи­ сок элементов каждого из классов, список параметров каждого метода и т. д. Вся информация берется из метаданных сборки. Для использования рефлексии не­ обходимы класс System.Туре и типы пространства имен System.Reflection. 280 Глава 12. Сборки, библиотеки, атрибуты, директивы В классе Туре описаны методы, которые позволяют получить информацию о ти­ пах. В пространстве имен System.Reflection описаны типы, поддерживающие Туре, а также классы, которые служат для организации позднего связывания и дина­ мической загрузки сборок. Наиболее важные свойства и методы класса Туре приведены в табл. 12.1. Таблица 12.1 . Элементы класса Туре Элемент Описание IsAbstract, IsArray, IsNestedPublic, IsClass, IsNestedPrivate, IsCOMObject, IsEnum, Islnterface, IsPrimitive, IsSealed, IsValueType GetConstructors, GetEvents, GetFields, Get Interfaces, GetMethods, GetMembers, GetNestedTypes, GetProperties FindMembers GetType InvokeMember Свойства, позволяющие получить соответствующие характеристики конкретного типа в программе (например, является ли он абстрактным, является ли он массивом, классом и т. п.). Приведены не все свойства Методы, возвращающие массив с набором соответствующих элементов (конструкторов, событий, полей и т. п.). Возвращаемое значение соответствует имени метода, например, GetFields возвращает массив типа Fieldlnfo, GetMethods — массив типа Method Info. Для каждого из методов есть парный ему (без символа S в конце имени), который предназначен для работы с одним заданным в параметре элементом (например, GetMethod и GetMethods) Метод возвращает массив типа Member Info на основе заданных критериев поиска Метод возвращает объект типа Туре по имени, заданному в виде строки Метод используется для позднего связывания заданного элемента Воспользоваться этими методами можно после создания экземпляра класса Туре. Поскольку это абстрактный класс, обычный способ создания объектов с помо­ щью операции new неприменим, зато существуют три других способа: 1. В базовом классе object описан метод GetType, которым можно воспользовать­ ся для любого объекта, поскольку он наследуется. Метод возвращает объект типа Туре, например: Monster X = new MonsterC); Type t = X.GetTypeO; 2. В классе Type описан статический метод GetType с одним параметром строко­ вого типа, на место которого требуется передать имя класса (типа), например: Type t = Type.GetTypeC "Monster" ); 3. Операция typeof возвращает объект класса Туре для типа, заданного в качест­ ве параметра, например: Type t = typeofС Monster ); Рефлексия 281 При использовании второго и третьего способов создавать экземпляр исследуе­ мого класса нет необходимости. Как видно из табл. 12.1, многие методы класса Туре возвращают экземпляры стан­ дартных классов (например, Memberlnfo). Эти классы описаны в пространстве имен System. Ref lection. Наиболее важные из этих классов перечислены в табл. 12.2. Таблица 12.2. Некоторые классы пространства имен System.Reflection Т и п Описани е Assembly Содержит методы для получения информации о сборке, а также для загрузки сборки и выполнения с ней различных операций AssemblyName Позволяет получать информацию о сборке (имя, версия, совместимость, естественный язык и т. п.) EventInfo Хранит информацию о событии Fieldlnfo Хранит информацию о поле Memberlnfo • Абстрактный базовый класс, определяющий общие элементы для классов Eventlnfo, Fieldlnfo, Methodlnfo и Propertylnfo Methodlnfo Хранит информацию о методе Module Позволяет обратиться к модулю в многофайловой сборке Parameterlnfo Хранит информацию о параметре Propertylnfo Хранит информацию о свойстве В листинге 12.3 приведены примеры использования рассмотренных методов и классов для получения информации о классах библиотеки из листинга 12.1. Листин г 12.3 . Получение информации о классах using System; using System.Reflection; namespace ConsoleApplicationl { using MonsterLib; class Classl { static void InfoC Type t ) { Console.WriteLineC "=== = Класс " + t.FullName ) ; if ( t.IsAbstract ) Console.WriteLineC "абстрактный" ); if ( t.IsClass ) Console.WriteLineC "обычный" ); if ( t.IsEnum ) Console.WriteLineC "перечисление" ); Console.WriteLineC "базовый класс " + t.BaseType ); MethodInfo[] met = t.GetMethodsO; foreach ( Methodlnfo m in met ) Console.WriteLineC m ); Console.WriteLineC): продолжение •. 282 Глава 12. Сборки, библиотеки, атрибуты, директивы Листин г 12.3 (продолжение) PropertyInfo[] prs = t.GetPropertiesO; , foreach ( PropertyInfo p in prs ) Console.WriteLineC p ); Console.WriteLineC); } static void MainO { Type t = typeof( Spirit ); InfoC t ); t = typeof( Monster ): InfoC t ); t = typeof( Daemon ); InfoC t ); } } } Результат работы программы: ======= Класс MonsterLib.Spirit абстрактный обычный базовый класс System.Object Void PassportO Int32 GetHashCodeO Boolean Equals(System.Object) System.String ToStringO System.Type Get TypeО ======= Класс MonsterLib.Monster обычный базовый класс MonsterLib.Spirit Void PassportO Int32 GetHashCodeO Boolean Equals(System.Object) System.String ToStringO Int32 get\_Health() Void set\_Health(Int32) Int32 get\_Ammo() Void set\_Ammo(Int32) System.String get\_Name() System.Type GetType() Int32 Health Int32 Ammo System.String Name Атрибуты 283 ======= Класс MonsterLib.Daemon обычный базовый класс MonsterLib.Monster Void PassportO Int32 GetHashCodeO Boolean Equals(System.Object) System.String ToStringO Void Think О Int32 getJealthO Void set\_Health(Int32) Int32 get\_Ammo() Void set\_Ammo(Int32) System.String get\_Name() System.Type GetTypeO Int32 Health Int32 Ammo System.String Name Можно продолжить исследования дальше, например, получить параметры и воз­ вращаемое значение каждого метода. Думаю, что принцип вам уже ясен. Напом­ ню, что вся эта информация берется из метаданных сборки. Атрибуты Атрибуты — это дополнительные сведения об элементах программы (классах, мето­ дах, параметрах и т. д.). С помощью атрибутов можно добавлять информацию в ме­ таданные сборки и затем извлекать ее во время выполнения программы. Атрибут яв­ ляется специальным видом класса и происходит от базового класса System. Attribute. Атрибуты делятся на стандартные и пользовательские. В библиотеке .NET преду­ смотрено множество стандартных атрибутов, которые можно использовать в про­ граммах. Если всего разнообразия стандартных атрибутов не хватит, чтобы удовле­ творить прихотливые требования программиста, он может описать собственные классы атрибутов, после чего применять их точно так же, как стандартные. При использовании (спецификации) атрибутов они задаются в секции атрибутов, располагаемой непосредственно перед элементом, для описания которого они предназначены. Секция заключается в квадратные скобки и может содержать не­ сколько атрибутов, перечисляемых через запятую. Порядок следования атрибу­ тов произвольный. Для каждого атрибута задаются имя, а также необязательные параметры и тип элемента сборки, к которому относится атрибут. Простейший пример атрибута: [Serializable] class Monster { 284 Глава 12. Сборки, библиотеки, атрибуты, директивы [NonSerialized] string name; int health, ammo; } Атрибут [Serializable], означающий, что объекты этого класса можно сохранять во внешней памяти, относится ко всему классу Monster. При этом поле name поме­ чено атрибутом [NonSerialized], что говорит о том, что это поле сохраняться не должно. Сохранение объектов рассматривалось в главе 10. Обычно из контекста понятно, к какому элементу сборки относится атрибут, однако в некоторых случаях могут возникнуть неоднозначности. Для их устра­ нения перед именем атрибута записывается тип элемента сборки — уточняющее ключевое слово, отделяемое от атрибута двоеточием. Ключевые слова и соответ­ ствующие элементы сборки, к которым могут относиться атрибуты, перечислены в табл. 12.3. Таблица 12.3 . Типы элемента сборки, задаваемые для атрибутов Ключевое слово Описание assembly Атрибут относится ко всей сборке field Атрибут относится к полю event Атрибут относится к событию method Атрибут относится к методу pa ram Атрибут относится к параметрам метода property Атрибут относится к свойству return Атрибут относится к возвращаемому значению type Атрибут относится к классу или структуре Пусть, например, перед методом описан гипотетический атрибут ABC: [ABC] public void Do() { .. . } По умолчанию он относится к методу. Чтобы указать, что атрибут относится не к методу, а к его возвращаемому значению, следует написать: [return:ABC] public void Do() { ... } Атрибут может иметь параметры. Они записываются в круглых скобках через запятую после имени атрибута и бывают позиционными и именованными. Име­ нованный параметр указывается в форме имя = значение, для позиционного про­ сто задается значение. Например, для использованного в следующем фрагмен­ те кода атрибута CLSCompliant задан позиционный параметр true. Атрибуты, Пространства имен 285 относящиеся к сборке, должны располагаться непосредственно после директив using, например: using System; [assembly:CLSCompliant(true)] namespace ConsoleApplicationl { ... Атрибут [CLSCompliant] определяет, удовлетворяет программный код соглашени­ ям CLS (Common Language Specification) или нет. Стандартные атрибуты, как и другие типы классов, имеют набор конструкторов, которые определяют, каким образом использовать (специфицировать) атрибут. Фактически, при использовании атрибута указывается наиболее подходящий конструктор, а величины, не указанные в конструкторе, задаются через имено­ ванные параметры в конце списка параметров. Стандартный атрибут [STAThread], старательно удаленный из всех листингов в этой книге, относится к методу, перед которым он записан. Он имеет значение только для приложений, использующих модель СОМ , и задает модель потоков в рамках модели СОМ . Пример применения еще одного стандартного атрибута, ["Conditional!, приведен далее в разделе «Директивы препроцессора». Атрибуты уровня сборки хранятся в файле Assemblylnfo.cs, автоматически созда­ ваемом средой для любого проекта. Для явного задания номера версии сборки можно записать атрибут [AssemblyVersion], например: [assembly: AssemblyVersion("1.0.0.0")] Создание пользовательских атрибутов выходит за рамки темы этого учебника. Интересующиеся могут обратиться к книге [27]. Пространства имен Пространство имен — это контейнер для типов, определяющий область их види­ мости. Пространства имен предотвращают конфликты имен и используются для двух взаимосвязанных целей: • логического группирования элементов программы, расположенных в различ­ ных физических файлах; • группирования имен, предоставляемых сборкой в пользование другим модулям. Во всех программах, созданных ранее, использовалось пространство имен, созда­ ваемое по умолчанию. Реальные программы чаще всего разрабатываются груп­ пой программистов, каждый из которых работает со своим набором физических файлов (единиц компиляции), хранящих элементы создаваемого приложения. Если в разных файлах описать пространства имен с одним и тем же именем, то при построении приложения, состоящего из этих файлов, будет скомпоновано единое пространство имен. 286 Глава 12. Сборки, библиотеки, атрибуты, директивы1 Пространства имей могут быть вложенными, например: namespace State { namespace City { ••• } } Такое объявление аналогично следующему: namespace State.City { ••• } Вложенные пространства имен, как вы наверняка успели заметить, широко при­ меняются в библиотеке .NET. Существует три способа использования типа, определенного в каком-либо про­ странстве имен: 1. Использовать полностью квалифицированное имя. Например, в пространстве имен System.Runtime.Serialization.Formatters.Binary описан класс BinaryFormatter. Создание объекта этого класса с помощью квалифицированного имени вы­ глядит так: System.Runtime.Serialization.Formatters.Binary.BinaryFormatter bf = new System.Runtime.Seri ali zati on.Formatters.Bi nary.Bi naryFormatter(); 2. Использовать директиву using, с помощью которой импортируются все имен; из заданного пространства. В этом случае предыдущий пример примет ви, using System.Runtime.Serialization.Formatters.Binary; BinaryFormatter bf = new BinaryFormatterO; ВНИМАНИ Е Директива using должна располагаться вне или внутри пространства имен, но ; любых описаний типов. 3. Использовать псевдоним типа. Это делается с помощью второй формы дире тивы using: using BinF = System.Runtime.Seri ali zati on.Formatters.Bi nary.Bi naryFormatter; BinF bf .= new BinFO; Первый способ применяется при однократном использовании имени типа «неглубоко» вложенных пространств имен, второй — в большинстве остальн случаев, что мы и делали во всех примерах, а третий можно рекомендовать г многократном использовании длинного имени типа. ЗКТИ8 Ы препроцессора 287 ерсию языка С # 2.0 введена возможность применять псевдоним пространст- шен с помощью операции ::, например: ss Program static void MainO { SI0::Stream s = new MIO: .-Empty St ream О; // использование псевдонимов пользование псевдонима для пространства имен гарантирует, что последую- (е подключения других пространств имен к этой сборке не повлияют на суще- дующие определения. Слева от операции :: можно указать идентификатор )bal. Он гарантирует, что поиск идентификатора, расположенного справа от ерации, будет выполняться только в глобальном пространстве имен. Цель по­ льзования этого идентификатора та же: не допустить изменений существую- та определений при разработке следующих версий программы, в которых в нее пут быть добавлены новые пространства имен, содержащие элементы с такими ; именами. жим образом, сборки обеспечивают физическое группирование типов, а про- ранства имен — логическое. В мире сетевого программирования, когда про- )аммисту доступны десятки тысяч классов, пространства имен совершенно юбходимы как для классификации и поиска, так и для предотвращения кон- ликтов имен типов. репроцессором в языке С+ + называется предварительный этап компиляции, ормирующий окончательный вариант текста программы. В языке С#, потомке -++, препроцессор практически отсутствует, но некоторые директивы сохрани- ись. Назначение директив — исключать из процесса компиляции фрагменты ода при выполнении определенных условий, выводить сообщения об ошибках предупреждения, а также структурировать код программы. каждая директива располагается на отдельной строке и не заканчивается точкой запятой, в отличие от операторов языка. В одной строке с директивой может /асполагаться только комментарий вида //. Перечень и краткое описание дирек- ив приведены в табл. 12.4. 'ассмотрим более подробно применение директив условной компиляции. Они ^пользуются для того, чтобы исключить компиляцию отдельных частей про- раммы. Это бывает полезно при отладке или, например, при поддержке несколь- сих версий программы для различных платформ. тд SI0 = System.10: ig MIO = MyLibrary.IO; // псевдоним пространства имен // псевдоним пространства имен Директивы препроцессора 288 Глава 12. Сборки, библиотеки, атрибуты, директивы Таблиц а 12.4 . Директивы препроцессора Наименование Описание fdefine, fundef #if,#elif, felse, fendif #line terror, fwarning #region, fendregion fpragma Определение (например, #def i ne DEBUG) и отмена определения (fundef DEBUG) символьной константы, которая используется директивами условной компиляции. Директивы размещаются до первой лексемы единицы компиляции. Допускается повторное определение одной и той же константы Директивы условной компиляции. Код, находящийся в области их действия, компилируется или пропускается в зависимости от того, была ли ранее определена символьная константа (см. далее) Задание номера строки и имени файла, о котором выдаются сообщения, возникающие при компиляции (например, #1 iпе 200 "ku\_ku.txt"). При этом в диагностических сообщениях компилятора имя компилируемого файла заменяется указанным, а строки нумеруются, начиная с заданного первым параметром номера 1 Вывод при компиляции сообщения, указанного в строке директивы. После выполнения директивы f error компиляция прекращается (например, ferror Дальше компилировать нельзя). После выполнения директивы fwarning компиляция продолжается Определение фрагмента кода, который можно будет свернуть или развернуть средствами редактора кода. Фрагмент располагается между этими директивами Введена в версии С# 2.0. Позволяет отключить (#pragma warning disable) или восстановить (#pragma warning restore ) выдачу всех или перечисленных в директиве предупреждений компилятора Формат директив: #if константноевыражение [ #e1if константноевыражение ...] [ #elif константноевыражение ... ] [ #else ... ] #endif Количество директив #elif произвольно. Исключаемые блоки кода могут со­ держать как описания, так и исполняемые операторы. Константное выражение может содержать одну или несколько символьных констант, объединенных знаками операций ==, ! =, !, && и 11. Также допускаются круглые скобки. Кон­ станта считается равной true, если она была ранее определена с помощью ди­ рективы #def i ne. 1 «...Натуре находится много вещей, неизъяснимых даже для обширного ума». — Н. В. Гоголь. Директивы препроцессора 289 Пример применения директив приведен в листинге 12.4. Листин г 12.4 . Применение директив условной компиляции // #define VAR1 // #define VAR2 using System; namespace ConsoleApplicationi { class Classl { #if VAR1 static void F() { Console.WriteLine( "Вариант 1" ): } • #elif VAR2 static void F(){ Console.WriteLineC "Вариант 2" ); } #el se static void F() { Console.WriteLineC "Основной вариант" ); } #endif static void MainO { FO: } } } В зависимости от того, определение какой символьной константы раскомменти­ ровать, в компиляции будет участвовать один из трех методов F. Директива #def i пе применяется не только в сочетании с директивами условной компиляции. Можно применять ее вместе со стандартным атрибутом Conditional для условного управления выполнением методов. Метод будет выполняться, если константа определена. Пример приведен в листинге 12.5. Обратите внима­ ние на то, что для применения атрибута необходимо подключить пространство имен System.Diagnostics. Листин г 12.5 . Использование атрибута Conditional // #define VAR1 fdefine VAR2 using System; using System.Diagnostics: namespace ConsoleApplicationi { class Classl { [Conditional ("VAR1")] static void A() { Console.WriteLineC "Выполняется метод A" ); } [Conditional ("VAR2")] продолжение & 290 Глава 12. Сборки, библиотеки, атрибуты, директивы Листин г 12,5 (продолжение) static void В(){ Console.WriteLineC "Выполняется метод В" ); } static void MainO { АО: BO; •} } } В методе Main записаны вызовы обоих методов, однако в данном случае будет выполнен только метод В, поскольку символьная константа VAR1 не определена. Рекомендации по программированию В этой главе приведено краткое введение в средства, использующиеся при про­ фессиональной разработке программ: библиотеки, рефлексия типов, атрибуты, пространства имен и директивы препроцессора. Для реального использования этих возможностей в программах необходимо изучать документацию и дополни­ тельную литературу, например [20], [21], [26], [27], [30]. лава 13 Структуры данных, коллекции л классы-прототипы 3 этой главе приводится обзор основных структур данных, используемых в про- раммировании, и их реализация в библиотеке .NET в виде коллекций. Кроме гого, описываются средства, добавленные в версию С# 2.0, — классы-прототипы [generics), частичные типы (partial types) и обнуляемые типы (nullable types)1 . Абстрактные структуры данных Любая программа предназначена для обработки данных, от способа организации которых зависит ее алгоритм. Для разных задач необходимы различные способы хранения и обработки данных, поэтому выбор структур данных должен предше­ ствовать созданию алгоритмов и основываться на требованиях к функциональ­ ности и быстродействию программы. Наиболее часто в программах используются следующие структуры данных: мас­ сив, список, стек, очередь, бинарное дерево, хеш-таблица, граф и множество. Да­ лее даны краткие характеристики каждой их этих структур. Массив — это конечная совокупность однотипных величин. Массив занимает не­ прерывную область памяти и предоставляет прямой, или произвольный, доступ к своим элементам по индексу. Память под массив выделяется до начала работы с ним и впоследствии не изменяется. В списке каждый элемент связан со следующим и, возможно, с предыдущим. В пер­ вом случае список называется односвязным, во втором — двусвязным. Также приме­ няются термины однонаправленный и двунаправленный. Если последний элемент 1 Остальные средства, появившиеся в версии 2.0, были рассмотрены ранее. Например, итераторы — в главе 9, анонимные методы — в главе 10. 292 Глава 13. Структуры данных, коллекции и классы-прототипы связать указателем с первым, получается кольцевой список. Количество элемен­ тов в списке может изменяться в процессе работы программы. Каждый элемент списка содержит ключ, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью данных, хранящихся в каждом элементе списка. В качестве ключа в процессе работы со списком могут выступать разные части данных. Например, если создается спи­ сок из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа: при упорядочивании списка по алфавиту ключом будет фамилия, а при поиске, например, ветеранов труда клю­ чом можно сделать стаж. Ключи разных элементов списка могут совпадать. Над списками можно выполнять следующие операции: • добавление элемента в конец списка; • чтение элемента с заданным ключом; • вставка элемента в заданное место списка; • удаление элемента с заданным ключом; • упорядочивание списка по ключу. Список не обеспечивает произвольный доступ к элементу, поэтому при выпол­ нении операций чтения, вставки и удаления выполняется последовательный пе­ ребор элементов, пока не будет найден элемент с заданным ключом. Для списков большого объема перебор элементов может занимать значительное время, по­ скольку среднее время поиска элемента пропорционально количеству элементов в списке. Стек — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вер­ шиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (Last In — First Out, последним пришел — первым ушел). Стек проще всего представить себе как закрытую с одного конца узкую трубу, в которую бросают мячики. Достать первый брошенный мячик можно только после того, как выну­ ты все остальные. Стеки широко применяются в системном программном обес­ печении, компиляторах, в различных рекурсивных алгоритмах. Очередь — это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца. Другие опе­ рации с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (First In — First Out, первым пришел — первым ушел). Очередь проще всего представить себе, постояв в ней час-другой. В программировании очереди применяются, например, в моделировании, диспетчеризации задач операционной системой, буферизован­ ном вводе-выводе. Бинарное дерево — это динамическая структура данных, состоящая из узлов, каж­ дый из которых содержит помимо данных не более двух ссылок на различные бинарные поддеревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева. Абстрактные структуры данных 293 Пример бинарного дерева приведен на рис. 13.1 (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие — потомками. Высота дерева определяется ко­ личеством уровней, на которых располагаются его узлы. Рис. 13.1 . Пример бинарного дерева поиска Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется деревом поиска. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму ко­ личества узлов. Впрочем, скорость поиска в значительной степени зависит от по­ рядка формирования дерева: если на вход подается упорядоченная или почти упорядоченная последовательность ключей,,дерево вырождается в список. Для ускорения поиска применяется процедура балансировки дерева, формирующая дерево, поддеревья которого различаются не более чем на один элемент. Бинар­ ные деревья применяются для эффективного поиска и сортировки данных. Хеш-таблица, ассоциативный массив, или словарь — это массив, доступ к эле­ ментам которого осуществляется не по номеру, а по некоторому ключу. Можно 294 Глава 13. Структуры данных, коллекции и классы-прототипы сказать, что это таблица, состоящая из пар «ключ—значение» (табл. 13.1). Хеш- таблица эффективно реализует операцию поиска значения по ключу. При этом ключ преобразуется в число (хеш-код), которое используется для быстрого нахож­ дения нужного значения в хеш-таблице. Таблиц а 13.1 . Пример хеш-таблицы Ключ Значение boy мальчик girl девочка dog собачка ' Преобразование выполняется с помощью хеш-функции, или функции расстановки. Эта функция обычно производит какие-либо преобразования внутреннего пред­ ставления ключа, например, вычисляет среднее арифметическое кодов составляющих его символов (английское слово «hash» означает перемалывать, перемешивать). Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов массива, то доступ к элементу по ключу выполняется почти так же быстро, как в массиве. Если хеш-функция генерирует для разных ключей оди­ наковые хеш-коды, время поиска возрастает и становится сравнимым со временем поиска в списке. ПРИМЕЧАНИ Е Грубый пример хеш-функции:, когда мы обращаемся к англо-русскому словарю, мы можем использовать в качестве хеш-кода первую букву английского слова (ключа). Открыв словарь на странице, с которой начинаются слова на эту букву, выполняем последовательный поиск ключа в списке. Смысл хеш-функции состоит в том, чтоб отобразить более широкое множество ключей в более узкое множество индексоэ. При этом неизбежно возникают так называемые коллизии, когда хеш-функция формирует для двух разных элементов один и тот же хеш-код. В разных реализациях хеш-таблиц используются различ­ ные стратегии борьбы с коллизиями. Граф — это совокупность узлов и ребер, соединяющих различные узлы. Напри­ мер, можно представить себе карту автомобильных дорог как граф с городами в качестве узлов и шоссе между городами в качестве ребер. Множество реальных практических задач можно описать в терминах графов, что делает их структурой данных, часто используемой при написании программ. Множество — это неупорядоченная совокупность элементов. Для множеств опреде­ лены операции проверки принадлежности элемента множеству, включения и ис­ ключения элемента, а также объединения, пересечения и вычитания множеств. Описанные структуры данных называются абстрактными, поскольку в них не задается реализация допустимых операций. В библиотеках большинства современных объектно-ориентированных языков программирования представлены стандартные классы, реализующие основные Пространство имен System.Collections 295 абстрактные структуры данных. Такие классы называются коллекциями, или кон­ тейнерами. Для каждого типа коллекции определены методы работы с ее эле­ ментами, не зависящие от конкретного типа данных, которые хранятся в коллек­ ции, поэтому один и тот же вид коллекции можно использовать для хранения данных различных типов. Использование коллекций позволяет сократить сроки разработки программ и повысить их надежность. ВНИМАНИ Е : Каждый вид коллекции поддерживает свой набор операций над данными, и быстро­ действие этих операций может быть разным. Выбор вида коллекции зависит от того, что требуется делать с данными в программе и какие требования предъявля­ ются к ее быстродействию. Например, при необходимости часто вставлять и уда­ лять элементы из середины последовательности следует использовать список, а не массив, а если включение элементов выполняется главным образом в конец или на­ чало последовательности — очередь. Поэтому изучение возможностей стандартных коллекций и их грамотное применение являются необходимыми условиями созда­ ния эффективных и профессиональных программ. В библиотеке .NET определено множество стандартных классов, реализующих большинство перечисленных ранее абстрактных структур данных. Основные про­ странства имен, в которых описаны эти классы, — System.Collections, System. Col 1 ecti ons. Speci al i zed и System. Col 1 ecti ons. Generi с (начиная с версии 2.0). В сле­ дующих разделах кратко описаны основные элементы этих пространств имен. ПРИМЕЧАНИ Е Класс System. Array, представляющий базовую функциональность массива, описан в разделе «Класс System.Array» (см. 6. 133). Пространство имен System.Collections В пространстве имен System.Collections определены наборы стандартных коллек­ ций и интерфейсов, которые реализованы в этих коллекциях. В табл. 13.2 при­ ведены наиболее важные интерфейсы, часть из которых мы уже изучали в разделе «Стандартные интерфейсы .NET» (см. с. 198). Таблиц а 13.2. Интерфейсы пространства имен System.Collections Интерфей с Назначени е ICol 1 ecti on Определяет общие характеристики (например, размер) для набора элементов I Compare г Позволяет сравнивать два объекта IDictionary Позволяет представлять содержимое объекта в виде пар «имя—значение» IDictionaryEnumerator Используется для нумерации содержимого объекта, поддерживающего интерфейс IDictionary продолжение ^ 296 Глава 13. Структуры данных, коллекции и классы-прото Таблица 13. 2 (продолжение) Интерфейс Назначение IEnumerable IEnumerator IHashCodeProvider I Li st Возвращает интерфейс IEnumerator для указанного объекта Обычно используется для поддержки оператора foreach в отношении объектов Возвращает хеш-код для реализации типа с применением выбранного пользователем алгоритма хеширования Поддерживает методы добавления, удаления и индексиров элементов в списке объектов В табл. 13.3 перечислены основные коллекции, определенные в пространс System. Collections1 . Таблица 13.3. Коллекции пространства имен System.Collections Класс Назначение Важнейшие из реализованных интерфейс ArrayList Массив, динамически изменяющий свой размер BitArray Компактный массив для хранения битовых значений Hashtable Хеш-таблица2 Queue Очередь SortedList Коллекция, отсортированная по ключам. Доступ к элементам — по ключу или по индексу Stack Стек IList, ICollection, IEnumerable, ICloneabl I Col lection, IEnumerable, ICloneable IDictionary, ICollection, IEnumerable, ICloneable ICollection, ICloneable, IEnumerable IDictionary, ICollection, IEnumerable, ICloneable ICollection, IEnumerable Пространство имен System.Collections.Specialized включает специализиро ные коллекции, например коллекцию строк StringCollection и хеш-таблицу строковыми ключами StringDictionary. В качестве примера стандартной коллекции рассмотрим класс ArrayLi st. Класс ArrayList Основным недостатком обычных массивов является то, что объем памяти, обходимый для хранения их элементов, должен быть выделен до начала раб с массивом. Класс ArrayList позволяет программисту не заботиться о выделе памяти и хранить в одном и том же массиве элементы различных типов. 1 Таблицы 13.2 и 13.3 приводятся по [27] и документации. 2 У типов, экземпляры которых предназначены для хранения в объекте класса Hashtabl должен быть замещен метод System.Object.GetHashCode. Пространство имен System.Collections 297 По умолчанию при создании объекта типа ArrayLi st строится массив из 16 элементов типа object. Можно задать желаемое количество элементов в массиве, передав его в конструктор или установив в качестве значения свойства Capacity, например: ArrayList arrl = new ArrayList(); // создается массив из 16 элементов ArrayLi st arr2 = new ArrayList(1000); // создается массив из 1000 элементов ArrayList аггЗ = new ArrayList(); arr3.Capacity = 1000; // количество элементов задается Основные методы и свойства класса ArrayLi st перечислены в табл. 13.4. Таблица 13.4 . Основные элементы класса ArrayList Элемент Вид Описание Capacity Свойство Емкость массива (количество элементов, которые могут храниться в массиве) Count Свойство Фактическое количество элементов массива Item Свойство Получить или установить значение элемента по заданному индексу Add Метод Добавление элемента в конец массива AddRange Метод Добавление серии элементов в конец массива BinarySearch Метод Двоичный поиск в отсортированном массиве или его части Clear Метод Удаление всех элементов из массива Clone Метод Поверхностное копирование 1 элементов одного массива в другой массив CopyTo Метод Копирование всех или части элементов массива в одномерный массив t GetRange Метод Получение значений подмножества элементов массива в виде объекта типа ArrayLi st IndexOf Метод Поиск первого вхождения элемента в массив (возвращает индекс найденного элемента или -1, если элемент не найден) Insert Метод Вставка элемента в заданную позицию (по заданному индексу) InsertRange Метод Вставка группы элементов, начиная с заданной позиции LastIndexOf Метод Поиск последнего вхождения элемента в одномерный массив Remove Метод Удаление первого вхождения заданного элемента в массив RemoveAt Метод Удаление элемента из массива по заданному индексу RemoveRange Метод Удаление группы элементов из массива Reverse Метод Изменение порядка следования элементов на обратный SetRange Метод Установка значений элементов массива в заданном диапазоне Sort Метод Упорядочивание элементов массива или его части TrimToSize Метод Установка емкости массива равной фактическому количеству элементов Это понятие рассматривалось в разделе «Клонирование объектов» (см. с. 205). 298 Глава 13. Структуры данных, коллекции и классы-прототипы Класс ArrayLi st реализован через класс Array, то есть содержит закрытое поле этого класса. Поскольку все типы в С# являются потомками класса object, мас­ сив может содержать элементы произвольного типа. Даже если в массиве хра­ нятся обычные целые числа, то есть элементы значимого типа, внутренний класс является массивом ссылок на экземпляры типа object, которые представляют собой упакованный тип-значение. Соответственно, при занесении в массив вы­ полняется упаковка, а при извлечении — распаковка элемента. Это не может не, сказаться на быстродействии алгоритмов, использующих ArrayLi st. Если при добавлении элемента в массив оказывается, что фактическое количест­ во элементов массива превышает его емкость, она автоматически удваивается, то есть происходит повторное выделение памяти и переписывание туда всех суще­ ствующих элементов. Пример занесения элементов в экземпляр класса ArrayLi st: arrl.AddC 123 ); arrl.Add( -2 ): arrl.AddC "Вася" ); Доступ к элементу выполняется по индексу, однако при этом необходимо явным образом привести полученную ссылку к целевому типу, например: int а = (int) arrl[0] int b = (int) arrl[l ] string s = (string) arrl[2] Попытка приведения к типу, не соответствующему хранимому в элементе, вызы вает генерацию исключения InvalidCastException. Для повышения надежности программ применяется следующий прием: экзе ляр класса ArrayLi st объявляется закрытым полем класса, в котором необход" хранить коллекцию значений определенного типа, а затем описываются мето работы с этой коллекцией, делегирующие свои функции методам ArrayLi st. Э~ способ иллюстрируется в листинге 13.1, где создается класс для хранения объек тов типа Monster и производных от него. По сравнению с аналогичным листин из раздела «Виртуальные методы» (см. с. 178), в котором используется обыч~ массив, у нас появилась возможность хранить в классе Stado произвольное ко чество элементов. Листин г 13.1 . Коллекция объектов using System; using System.Col lections; namespace ConsoleApplicationl { class Monster { .. . } class Daemon : Monster { ... } class Stado : IEnumerable { Классы-прототи п ы 299 private ArrayList list; public StadoO { list = new ArrayListO; } public void Add( Monster m ) { list.Add( m ) ; } public void RemoveAtC int i ) { list.RemoveAtt i ); } public void ClearO { list.ClearO; } public IEnumerator GetEnumeratorO { return list.GetEnumeratorO; } } class Classl { static void MainO { Stado stado = new StadoO; stado.Add( new Monster( "Monia" ) ,) ; stado.Add( new MonsterC "Monk" ) ); stado.Add( new Daemon ( "Dimon", 3 ) •); stado. RemoveAU 1 ); foreach ( Monster x in stado ) x. PassportO; } } } Результат работы программы: Monster Monia health = 100 ammo = 100 Daemon Dimon health = 100 ammo = 100 brain = 3 Недостатком этого решения является то, что для каждого метода стандартной коллекции приходится описывать метод-оболочку, вызывающий стандартный метод. Хотя это и несложно, но несколько неизящно. В С#, начиная с версии 2.0, появились классы-прототипы (generics), позволяющие решить эту проблему. Мы рассмотрим их в следующем разделе. Классы-прототипы Многие алгоритмы не зависят от типов данных, с которыми они работают. Про­ стейшими примерами могут служить сортировка и поиск. Возможность отделить алгоритмы от типов данных предоставляют классы-прототипы (generics) — классы, имеющие в качестве параметров типы данных. Чаще всего эти классы применяются для хранения данных, то есть в качестве контейнерных классов, или коллекций. ПРИМЕЧАНИ Е Английский термин «generics» переводится в нашей литературе по-разному: уни­ версальные классы, родовые классы, параметризованные классы, обобщенные классы, шаблоны, классы-прототипы и даже просто генерики. Наиболее адекватны­ ми мне кажутся варианты «классы-прототипы» и «параметризованные классы», поэтому в последующем изложении в основном используются именно эти терми­ ны, хотя и их точными не назовешь. 300 Глава 13. Структуры данных, коллекции и классы-прототипы Во вторую версию библиотеки .NET добавлены параметризованные коллекции для представления основных структур данных, применяющихся при создании программ, — стека, очереди, списка, словаря и т. д. Эти коллекции, расположен­ ные в пространстве имен System.Collections.Generic, дублируют аналогичные коллекции пространства имен System.Collections, рассмотренные в разделе «Про­ странство имен System.CoIlections» (см. с. 295). В табл. 13.5 приводится соот­ ветствие между обычными и параметризованными коллекциями библиотеки .NET (параметры, определяющие типы данных, хранимых в коллекции, указаны в угловых скобках). Таблиц а 13.5 . Параметризованные коллекции библиотеки .NET версии 2.0 Класс-прототип (версия 2.0) Обычный класс Comparer Comparer Dictionary HashTable LinkedList - List ArrayList Queue Queue SortedDi cti ona ry SortedList Stack Stack У коллекций, описанных в библиотеке .NET версий 1.0 и 1.1, есть два основных недостатка, обусловленных тем, что в них хранятся ссылки на тип object: • в одной и той же коллекции можно хранить элементы любого типа, следова­ тельно, ошибки при помещении в коллекцию невозможно проконтролиро­ вать на этапе компиляции, а при извлечении элемента требуется его явное преобразование; • при хранении в коллекции элементов значимых типов выполняется большой объем действий по упаковке и распаковке элементов, что в значительной сте­ пени снижает эффективность работы. Параметром класса-прототипа является тип данных, с которым он работает. Это избавляет от перечисленных недостатков. В качестве примера рассмотрим при­ менение универсального «двойника» класса ArrayLi st — класса List — для хранения коллекции объектов классов Monster и Daemon, которые разрабатывались в главах 5 и 8, а также для хранения целых чисел 1 . Листин г 13.2 . Использование универсальной коллекции List using System; using System.Collections.Generic; using System.Text: Полное описание этих классов приведено в разделе «Создание библиотеки» (см. с. 275) Классы-прототипы 301 namespace ConsoleAppl icationi { using MonsterLib; class Program { static void MainO { List stado = new List(); stado.Add( new MonsterC "Monia" ) ); stado.Add( new MonsterC "Monk" ) ); stado.AddC new Daemon ( "Dimon", 3 ) ); foreach ( Monster x in stado ) x.PassportO; List lint = new List(); lint.AddC 5 ); lint.AddC 1 ); lint.AddC 3 ); lint.SortO; int a = lint[2]; Console.WriteLineC a ); foreach ( int x in lint ) Console.WriteC x + " " ); } } } Результат работы программы: Monster Monia health = 100 ammo Monster Monk health = 100 ammo Daemon Dimon health = 100 ammo 5 1 3 5 В листинге 13.2 две коллекции. Первая (stado) содержит элементы пользователь­ ских классов, которые находятся в библиотеке MonsterL.ib.clll, созданной в преды­ дущей главе. В коллекции, для которой объявлен тип элементов Monster, благода­ ря полиморфизму можно хранить элементы любого производного класса, но не элементы других типов. Казалось бы, по сравнению с обычными коллекциями это ограничение, а не уни­ версальность, однако на практике коллекции, в которых действительно требует­ ся хранить значения различных, не связанных межу собой типов, почти не ис­ пользуются. Достоинством же такого ограничения является то, что компилятор может выполнить контроль типов во время компиляции, а не выполнения про­ граммы, что повышает ее надежность и упрощает поиск ошибок. Коллекция 1 i nt состоит из целых чисел, причем для работы с ними не требуются ни операции упаковки и распаковки, ни явные преобразования типа при получении элемента из коллекции, как это было в обычных коллекциях (см. листинг 13.1). Классы-прототипы называют также родовыми или шаблонными, поскольку они представляют собой образцы, по которым во время выполнения программы = 100 = 100 = 100 brain = 3 302 Глава 13. Структуры данных, коллекции и классы-прототипы строятся конкретные классы. При этом сведения о классах, которые являются параметрами классов-прототипов, извлекаются из метаданных. ВНИМАНИ Е Использование стандартных параметризованных коллекций для хранения и обра­ ботки данных является хорошим стилем программирования, поскольку позволяет сократить сроки разработки программ и повысить их надежность. Рекомендуется тщательно изучить по документации свойства и методы этих классов и выбирать наиболее подходящие в зависимости от решаемой задачи. В листинге 13.3 приведен еще один пример применения параметризованных коллекций. Программа считывает содержимое текстового файла, разбивает его на слова и подсчитывает количество повторений каждого слова в тексте. Для хранения слов и числа их повторений используется словарь Dictionary. У этого класса два параметра: тип ключей и тип значений, хранимых в словаре. В качестве ключей используются слова, считанные из файла, а значения пред­ ставляют собой счетчики целого типа, которые увеличиваются на единицу, когда слово встречается в очередной раз. Листин г 13.3 . Формирование частотного словаря using System: using System.Collections.Generic; using System.Text; using System.10; namespace ConsoleApplicationl class Program { static void MainO StreamReader f = new StreamReader( @"d:\C#\text.txt" ); string s = f.ReadToEndO; / / / / 1 2 char[] separators = { ' ', '! ' }; List words = new List( s.Split(separators) ); Dictionary map = new Dictionary(); // / / / / 3 4 5 foreach ( string w in words ) // 6 if ( map.ContainsKeyC w ) ) map[w]++; else map[w] = 1; foreach ( string w in map.Keys ) Console.WriteLineC "{0}\t{l}" , w. map[w] ) ; // 7 } Классы-прототипы 303 Пусть исходный файл text. txt содержит строки Ехал Грека через реку. Видит Грека, в реке рак. Сунул Грека в реку руку, рак за руку Греку цап! Тогда результат работы программы будет выглядеть так: Ехал 1 Грека 3 через 1 реку 2 4 Видит 1 в 2 реке 1 рак 2 Сунул 1 руку 2 за 1 Греку 1 цап 1 Несколько пояснений к программе. В операторе 1 открывается текстовый файл, длина которого не должна превышать 32 767 символов, потому что в операторе 2 все его содержимое считывается в отдельную строку. ПРИМЕЧАНИ Е Конечно, для реальной работы такой способ не рекомендуется. Кроме того, для файлов, открываемых для чтения, программа обязательно должна обрабатывать ис­ ключение FileNotFoundException (см. главу 11). В операторе 3 задается массив разделителей, передаваемый в качестве параметра методу Spl it, формирующему массив строк, каждая из которых содержит отдель­ ное слово исходного файла. Этот массив используется для инициализации эк­ земпляра words класса List. Применение стандартного класса позволяет не заботиться о выделении места под массив слов. Оператор 5 описывает словарь, а в цикле 6 выполняется его заполнение путем просмотра списка слов words. Если слово встречается впервые, в значение, соот­ ветствующее слову как ключу, заносится единица. Если слово уже встречалось, значение увеличивается на единицу. В цикле 7 выполняется вывод словаря путем просмотра всех его ключей (для этого используется свойство словаря Keys, возвращающее коллекцию ключей) и вы­ борки соответствующих значений. Метод Split не очень интеллектуален: он рассматривает пробел, расположенный после знака препинания, как отдельное слово. Для более точного разбиения на сло­ ва используются регулярные выражения, которые рассматриваются в главе 15. 304 Глава 13. Структуры данных, коллекции и классы-прототипы ПРИМЕЧАНИ Е — Обратите внимание на то, насколько использование стандартных коллекций сокра­ щает исходный текст программы. Конечно, на тщательное изучение их возможно­ стей требуется много времени, однако это окупается многократно. Для полноты картины следует добавить, что наряду с параметризованными клас­ сами в пространстве имен System.Collections.Generic описаны параметризованные интерфейсы, перечисленные в табл. 13.6. Таблиц а 13.6 . Параметризованные интерфейсы библиотеки .NET версии 2.0 Параметризованный интерфейс (версия 2.0) Обычный интерфейс ICollection ICollection IComparable IComparable IDi ct i ona ry I Enumerable IEnumerator I Enumerator IList IList Создание класса-прототипа Язык C# позволяет создавать собственные классы-прототипы и их разновидно­ сти — интерфейсы, структуры, делегаты и события, а также обобщенные (generic) методы обычных классов. Рассмотрим создание класса-прототипа на примере стека, приведенном в специ­ фикации С#. Параметр типа данных, которые хранятся в стеке, указывается в уг­ ловых скобках после имени класса, а затем используется таким же образом, как и обычные типы: public class Stack { Т [ ] items: int count; public void PushC T item ){... } // помещение в стек public T PopO { .. . } // извлечение из стека } При использовании этого класса на место параметра Т подставляется реальный тип, например int: Stack stack - new Stack(); stack.PushC 3 ); int x = stack.PopO; Тип Stack называется сконструированным типом (constructed type). Этот тип создается во время выполнения программы при его первом упоминании Классы-прототипы 305 в программе. Если в программе встретится класс Stack с другим значимым ти­ пом, например doubl е, среда выполнения создаст другую копию кода для этого типа. Напротив, для всех ссылочных типов будет использована одна и та же копия кода, поскольку работа с указателями на различные типы выполняется одинако­ вым образом. Создание конкретного экземпляра класса-прототипа называется инстанцированием. Класс-прототип может содержать произвольное количество параметров типа. Для каждого из них могут быть заданы ограничения (constraints), указывающие, каким требованиям должен удовлетворять аргумент, соответствующий этому па­ раметру, например, может быть указано, что это должен быть значимый тип или тип, который реализует некоторый интерфейс. Синтаксически ограничения задаются после ключевого слова where, например: public class Stack where T : struct { ... Здесь с помощью слова struct записано ограничение, что элементы стека долж­ ны быть значимого типа. Для ссылочного типа употребляется ключевое слово class. Для каждого типа, являющегося параметром класса, может быть задана одна строка ограничений, которая может включать один класс, а за ним — произ­ вольное количество интерфейсов, перечисляемых через запятую. Указание в качестве ограничения имени класса означает, что соответствующий аргумент при инстанцировании может быть именем либо этого класса, либо его потомка. Указание имени интерфейса означает, что тип-аргумент должен реа- лизовывать данный интерфейс — это позволяет использовать внутри класса-про­ тотипа, например, операции по перечислению элементов, их сравнению и т. п. Помимо класса и интерфейсов в ограничениях можно задать требование, чтобы тип-аргумент имел конструктор по умолчанию без параметров, что позволяет создавать объекты этого типа в теле методов класса-прототипа. Это требование записывается в виде выражения new(), например: public class EntityTablе where К: IComparable, IPersistable where E: Entity. new() { public void Add( К key. E entity ) { ••• if ( key.CompareToC x ) < 0 ) { ... } } } В этом примере на первый аргумент класса EntityTable накладываются два огра­ ничения по интерфейсам, а для второго аргумента задано, что он может быть только классом Entity или его потомком и иметь конструктор без параметров. 306 Глава 13. Структуры данных, коллекции и классы-прототипы Задание ограничений позволяет компилятору выполнять более строгий контроль типов и, таким образом, избежать многих ошибок, которые иначе проявились бы только во время выполнения программы. Для задания значений по умолчанию типизированным элементам класса-прото­ типа используется ключевое слово default. С его помощью элементам ссылочного типа присваивается значение null, а элементам значимого типа — 0. Иногда удобно иметь отдельный метод, параметризованный каким-либо типом данных. Рассмотрим этот случай на примере метода сортировки. Известно, что «самого лучшего» алгоритма сортировки не существует. Для раз­ личных объема, диапазона, степени упорядоченности данных и распределения ключей оптимальными могут оказаться разные алгоритмы. Стандартные методы сортировки реализуют алгоритмы, которые хороши для большинства примене­ ний, но не для всех, поэтому может возникнуть необходимость реализовать соб­ ственный метод. В листинге 13.4 приведен пример сортировки методом выбора. Алгоритм состо­ ит в том, что сначала выбирается наименьший элемент массива и меняется мес­ тами с первым элементом, затем просматриваются элементы, начиная со второ­ го, и наименьший из них меняется местами со вторым элементом, и т. д., всего п - 1 раз. На последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива. Листин г 13.4 . Сортировка выбором using System; using System.Collecti ons.Generi с; using System.Text; namespace ConsoleApplicationl { class Program T buf; int n = a.Length; for ( int i = 0; i < n - 1: ++i ) { int im = i; for ( int j = i + 1; j < n; ++j ) if ( a[j].CompareTo(a[im]) < 0 ) im = j; II Ъ buf = a[i] ; a[i ] = a[im]; a[im] = buf; } } static void MainO Обобщенные методы static void Sort ( ref T[ ] a ) where T : IComparable { // 1 // 2 Классы-прототипы 307 int:] а = { 1, 6. 4, 2, 7. 5, 3 } ; Sort( ref а ); foreach ( int elem in aa ) ) Console Console.WriteLine . C elem ); // 4 double[] b = { 1.1, 5 SortC ref b ); foreach ( double elem 2, 5.21, 2, 7, 6, 3 } : // 5 in b ) Console.WriteLineC elem ); string[] s = { "qwe", SortC ref s ); foreach ( string elem in s ) Console.WriteLineC elem ); qwer", "df", "asd" }; // 6 Метод параметризован типом, Hai который накладывается ограничение (опера­ тор 2), чтобы объекты класса-аргумента можно было сравнивать друг с другом с помощью метода CompareTo, использованного в операторе 3. В главной программе (методе Main) метод Sort вызывается двумя способами: с яв­ ным указанием параметра-типа (оператор 4) и без указания параметра (операто­ ры 5 и 6). Во втором случае компилятор по типу переданного в метод параметра самостоятельно определяет, какой именно тип используется при инстанцировании. ПРИМЕЧАНИ Е Параметры-типы могут использоваться в списке параметров, возвращаемом типе и в теле универсального метода. Итак, параметризованные типы и методы позволяют: • описывать способы хранения и алгоритмы обработки данных независимо от типов данных; • выполнять контроль типов во время компиляции, а не исполнения программы; • увеличить скорость обработки данных за счет исключения операций упаков­ ки, распаковки и преобразования типа. Как уже упоминалось, помимо классов-прототипов и обобщенных методов мож­ но описать параметризованные интерфейсы, структуры и делегаты. В помощью параметризованных интерфейсов можно определить список функ­ ций, которые могут быть реализованы различным образом для разных классов, реализующих эти интерфейсы. Параметризованные интерфейсы можно реализо- вывать в классе-прототипе, используя в качестве аргументов интерфейса пара­ метры типа, реализующего интерфейс, или в обычном классе, подставляя в каче­ стве параметров интерфейса конкретные типы. Параметризованные делегаты позволяют создавать обобщенные алгоритмы, логи­ ку которых можно изменять передаваемыми в качестве параметров делегатами. 308 Глава 13. Структуры данных, коллекции и классы-прототипы Частичные типы Во вторую версию языка введена возможность разбивать описание типа на части и хранить их в разных физических файлах, создавая так называемые частичные типы (partial types). Это может потребоваться для классов большого объема или, что более актуально, для отделения части кода, сгенерированной средствами среды, от написанной программистом вручную. Кроме того, такая возможность облег­ чает отладку программы, позволяя отделить отлаженные части класса от новых. Для описания отдельной части типа используется модификатор partial. Он мо­ жет применяться к классам, структурам и интерфейсам, например: public partial class А { } public partial class A { } После совместной компиляции этих двух частей получается такой же код, как если бы класс был описан обычным образом. Все части одного и того же частич­ ного типа должны компилироваться одновременно, иными словами, добавление новой части к уже скомпилированным не допускается. Модификатор partial не является ключевым словом и должен стоять непосред­ ственно перед одним из ключевых слов class, struct или interface в каждой из частей. Все части определения одного класса должны быть описаны в одном и том же пространстве имен. ПРИМЕЧАНИ Е Если модификатор parti al указывается для типа, описание которого состоит толь­ ко из одной части, это не является ошибкой. Модификаторы доступа для всех частей типа должны быть согласованными. Если хотя бы одна из частей содержит модификатор abstract или sealed, класс считается соответственно абстрактным или бесплодным. Класс-прототип также может объявляться по частям, в этом случае во всех час­ тях должны присутствовать одни и те же параметры типа с одними и теми же ог­ раничениями. Если частичный тип является наследником нескольких интерфейсов, в каждой части не требуется перечислять все интерфейсы: обычно в одной части объяв­ ляется один интерфейс и описывается его реализация, в другой части — другой интерфейс и т. д. Набором базовых интерфейсов для типа, объявленного в не­ скольких частях, является объединение базовых интерфейсов, определенных в каждой части. Обнуляемые типы 309 Обнуляемые типы В программировании давно существует проблема, каким образом задать, что пе­ ременной не присвоено никакого значения. Эта проблема решается разными способами. Один из способов заключается в том, чтобы присвоить переменной какое-либо значение, не входящее в диапазон допустимых для нее. Например, если величина может принимать только положительные значения, ей присваива­ ется -1. Ясно, что для многих случаев этот подход неприменим. Другой способ — хранение логического признака, по которому можно определить, присвоено ли переменной значение. Этот способ не может использоваться, например, для зна­ чений, возвращаемых из метода. В версии С# 2.0 указанная проблема решается введением типов специального вида, называемых обнуляемыми (nullable). Обнуляемый тип представляет собой структуру, хранящую наряду со значением величины (свойство Value) логиче­ ский признак, по которому можно определить, было ли присвоено значение этой величине (свойство HasValue). Если значение величине было присвоено, свойство HasValue имеет значение true. Если значение величины равно null, свойство HasValue имеет значение false, а по­ пытка получить значение через свойство Value вызывает генерацию исключения. Обнуляемый тип строится на основе базового типа, за которым следует символ ?, например: int? х = 123; int? у = null; if ( х.HasValue ) Console.WriteLineC x ); // вместо x можно записать x.Value if ( y.HasValue ) Console.WriteLineC у ); Существуют явные и неявные преобразования из обнуляемых типов в обычные и обратно, при этом выполняется контроль возможности получения значения, например: int i = 123: int? х = i: //int --> int? double? у = x; // int? --> double? int? z = (int?) y; // double? --> int? int j = (int) z; // int? --> int Для величин обнуляемых типов определены операции отношения . Операции == и != возвращают значение true, если обе величины имеют значение null. Естест­ венно, что значение nul 1 считается не равным любому ненулевому значению. Операции , <= и >= дают в результате false, если хотя бы один из операндов имеет значение nul 1. Арифметические операции с величинами обнуляемых типов дают в результате nul 1, если хотя бы один из операндов равен nul 1, например: int? х = null; int? у = х + 1; // у = null 310 Глава 13. Структуры данных, коллекции и классы-прототипы Для величин обнуляемых типов введена еще одна операция — объединения ?? (null coalescing operator). Это бинарная операция, результат которой равен пер­ вому операнду, если он не равен null, и второму в противном случае. Иными словами, эта операция предоставляет замещаемое значение для null, например: int? х = null: int у = х ?? 0; // у = О х = 1: у = х ?? 0: // у = 1 Обнуляемые типы удобно использовать при работе с базами данных и XML. Рекомендации по программированию Алгоритм работы программы во многом зависит от способа организации ее данных, поэтому очень важно до начала разработки алгоритма выбрать опти­ мальные структуры данных, основываясь на требованиях к функциональности и быстродействию программы. Для разных задач необходимы различные способы хранения и обработки данных, поэтому необходимо хорошо представлять себе как характеристики и области применения абстрактных структур данных, так и их конкретную реализацию в виде коллекций библиотеки. Изучение возможностей стандартных коллекций и их грамотное применение является необходимым условием создания эффек­ тивных и профессиональных программ, позволяет сократить сроки разработки программ и повысить их надежность. Недостатками коллекций первых версий библиотеки .NET является отсутствие контроля типов на этапе компиляции и неэффективность при хранении элемен­ тов значимых типов. Параметризованные коллекции, появившиеся в версии 2.0 библиотеки, избавлены от этих недостатков, поэтому в программах рекомендует­ ся использовать именно коллекции версии 2.0, выбирая наиболее подходящие классы в зависимости от решаемой задачи. Для реализации алгоритмов, независимых от типов данных, следует использо­ вать классы-прототипы и обобщенные методы. Они не снижают эффективность программы по сравнению с обычными классами и методами, поскольку код для конкретного типа генерируется средой CLR во время выполнения программы. Помимо классов-прототипов и обобщенных методов можно описать параметри­ зованные интерфейсы, структуры и делегаты. Частичные типы удобно использовать при разработке объемных классов груп­ пой программистов и для упрощения отладки программ. Обнуляемые типы при­ меняют для работы с данными, для которых необходимо уметь определять, было ли им присвоено значение. Глава 14 Введение в программирование под Windows В предыдущих главах мы изучали возможности языка С# на примере консольных приложений, чтобы не распылять свое внимание на изучение классов библиотеки .NET, поддерживающих стандартный графический интерфейс пользователя. Сей­ час пришло время освоить принципы создания Windows-приложений и получить представление об основных классах библиотеки, которые при этом используются. В первую очередь рассмотрим основные особенности Windows как характерного и широко распространенного представителя современных операционных систем. • Многозадачность — это возможность одновременно выполнять несколько при­ ложений. Операционная система обеспечивает разделение ресурсов: каждому приложению выделяется свое адресное пространство, распределяется процес­ сорное время и организуются очереди для доступа к внешним устройствам. Внутри приложения также можно организовать параллельное выполнение нескольких фрагментов, называемых потоками. В разных версиях Windows используются различные механизмы диспетчеризации. • Независимость программ от аппаратуры. Для управления аппаратными сред­ ствами любое приложение обращается к операционной системе, что обеспе­ чивает независимость от конкретных физических характеристик устройств: при смене устройства никакие изменения в программу вносить не требуется. Управление внешними устройствами обеспечивается с помощью драйверов. • Стандартный графический интерфейс с пользователем. Основное взаимо­ действие с пользователем осуществляется в графическом режиме. Каждое приложение выполняет вывод в отведенную ему прямоугольную область экра­ на, называемую окном. Окно состоит из стандартных элементов. Это упроща­ ет освоение программ пользователем и облегчает работу программиста, по­ скольку в его распоряжение предоставляются библиотеки интерфейсных компонентов. Такие библиотеки входят в состав систем программирования. 312 Глава 14. Введение в программирование под Windows Интерфейсные компоненты обращаются к аппаратуре не непосредственно, а через функции операционной системы, называемые API (Application Program Interface — программный интерфейс приложения). API-функции находятся в динамических библиотеках (Dynamic Link Library, DLL) , разделяемых всеми j приложениями. Эти библиотеки называются динамическими потому, что на­ ходящиеся в них функции не подключаются к каждому исполняемому файлу до выполнения программы, а вызываются в момент обращения к ним. В основе пользовательского интерфейса Windows лежит представление экра­ на как рабочего стола, на котором располагаются «листы бумаги» — окна \ приложений, ярлыки и меню. Одни элементы могут полностью или частично перекрывать другие, пользователь может изменять их размеры и перемещать их. Приложение может иметь несколько окон, одно из них является главным. При закрытии главного окна приложение завершается. | • Поддержка виртуального адресного пространства для каждого приложения. | Каждому приложению доступно пространство адресов оперативной памяти \ размером до 4 Гбайт1 . Операционная система отображает его на физические , адреса и обеспечивает защиту приложений друг от друга. В разных версиях \ Windows защита выполняется с различной степенью надежности, например, системы Windows 95/98 гораздо менее надежны, чем Windows NT/2000. • Возможность обмена данными между приложениями. Приложения могут об- ; мениваться данными через буфер обмена или используя другие механизмы, например OLE (Object Linking and Embedding — связывание и внедрение объектов) или именованные программные каналы. • Возможность запуска старых программ. В 32-разрядных версиях Windows можно выполнять 16-разрядные Windows-программы, а также программы, на­ писанные под MS-DOS. Последние запускаются в так называемой виртуаль­ ной DOS-машине, которая создает у программы полное «впечатление» того, что она выполняется под управлением MS-DO S в монопольном режиме. • Принцип событийного управления (рассматривается в следующем разделе). Событийно-управляемое программирование В основу Windows положен принцип событийного управления. Это значит, что и сама система, и приложения после запуска ожидают действий пользователя и реагируют на них заранее заданным образом. Любое действие пользователя (нажатие клавиши на клавиатуре, щелчок кнопкой мыши, перемещение мыши) называется событием. Структура программы, управляемой событиями, изобра­ жена на рис. 14.1. 1 Имеются в виду 32-разрядные приложения, то есть приложения, в которых использует­ ся адрес длиной 32 разряда. Событийно-управляемое программирование 313 Инициализация События Обработчик событий Подпрограмма 1 Подпрограмма п Завершение Рис. 14.1 . Структура программы, управляемой событиями Событие воспринимается Windows и преобразуется в сообщение — запись, содержащую необходимую информацию о событии (например, какая клавиша была нажата, в каком месте экрана произошел щелчок мышью). Сообщения могут поступать не только от пользователя, но и от самой системы, а также от активного или других приложений. Определен достаточно широкий круг стан­ дартных сообщений, образующий иерархию, кроме того, можно определять соб­ ственные сообщения. Сообщения поступают в общую очередь, откуда распределяются по очередям приложений. Каждое приложение содержит цикл обработки сообщений, кото­ рый выбирает сообщение из очереди и через операционную систему вызывает подпрограмму, предназначенную для его обработки (рис. 14.2). Таким образом, Windows-приложение состоит из главной программы, обеспечивающей инициа­ лизацию и завершение приложения, цикла обработки сообщений и набора обра­ ботчиков событий. Рис. 14.2. Структура Windows-приложения Среда Visual Studio.NET содержит удобные средства разработки Windows-при­ ложений, выполняющие вместо программиста рутинную работу — создание шаб­ лонов приложения и форм, заготовок обработчиков событий, организацию цик­ ла обработки сообщений и т. д. Рассмотрим эти средства. 314 Глава 14. Введение в программирование под Windows Шаблон Windows-приложения Создадим новый проект (File • New • Project), выбрав шаблон Windows Application (рис. 14.3). После более длительных раздумий, чем для консольного приложе­ ния, среда сформирует шаблон Windows-приложения. Первое отличие, которое бросается в глаза, — вкладка заготовки формы Form1.cs[Design], расположенная в основной части экрана. Форма представляет собой окно и предназначена для размещения компонентов (элементов управления) — меню, текста, кнопок, спи­ сков, изображений и т. д. Рис. 14.3 . Выбор шаблона проекта Среда создает не только заготовку формы, но и шаблон текста приложения. Пе­ рейти к нему можно, щелкнув в окне Solution Explorer (View • Solution Explorer) правой кнопкой мыши на файле Forml .cs и выбрав в контекстном меню команду View Code. При этом откроется вкладка с кодом формы, который, за исключе­ нием комментариев, приведен в листинге 14.1. Представлять себе, что написано в вашей программе, весьма полезно, поэтому давайте внимательно рассмотрим этот текст. Листин г 14.1 . Шаблон Windows-приложения using System; using System.Drawing: using System.Col lections; using System.ComponentModel; using System.Windows.Forms; using System.Data; 1блон Windows-приложения 315 nespace WindowsApplicationl public class Forml : System.Windows.Forms.Form { private System.ComponentModel.Container components = null; public Forml() { InitializeComponent () ; } protected override void Dispose( bool disposing ) { i f ( disposing ) { if (components != null) { components.Di spose(); } } base.Dispose( disposing ); } #region Windows Form Designer generated code private void InitializeComponentО • { this.components = new System.ComponentModel.ContainerO; this.Size = new System.Drawing.SizeOOO.300): this.Text = "Forml"; } #endregion static void MainO { Application. Run (new FormlO); } } Приложение начинается с директив использования пространств имен библиоте­ ки .NET. Для пустой формы, не содержащей ни одного компонента, необходи­ мыми являются только две директивы: jsing System; jsing System.Windows.Forms; Остальные директивы добавлены средой «на вырост». С пространством имен System вы уже знакомы. Пространство System.Windows.Forms содержит огромное количество типов, являющихся строительными блоками Windows-приложений. 316 Глава 14. Введение в программирование под Windows Список наиболее употребительных элементов этого пространства имен приведен в табл. 14.1, а часть иерархии — на рис. 14.4. Таблица 14.1 . Основные типы Windows.Forms Класс Назначение Application ButtonBase, Button, CheckBox, ComboBox, OataGrid, GroupBox, ListBox, LinkLabel, PictureBox Form ColorDialog, FileDialog, FontDialog, PrintPreviewDialog Menu, MainMenu, Menultem, ContextMenu Clipboard, Help, Timer, Screen, ToolTip, Cursors StatusBar, Splitter, ToolBar, ScrollBar Класс Windows-приложения. При помощи методов этого класса можно обрабатывать Windows-сообщения, запускать и прекращать работу приложения и т. п. Примеры классов, представляющих элементы управления (компоненты): базовый класс кнопок, кнопка, флажок, комбинированный список, таблица, группа, список, метка с гиперссылкой, изображение Класс формы — окно Windows-приложения Примеры стандартных диалоговых окон для выбора цветов, файлов, шрифтов, окно предварительного просмотра Классы выпадающих и контекстных меню Вспомогательные типы для организации графических интерфейсов: буфер обмена, помощь, таймер, экран, подсказка, указатели мыши Примеры дополнительных элементов управления, размещаемых на форме: строка состояния, разделитель, панель инструментов и т. д. Мы будем изучать типы пространства имен Wi ndows. Forms по мере необходимости. Продолжим рассмотрение листинга 14.1. В нем описан класс Forml, производный от класса Form. Он наследует от своего предка множество элементов, которые мы рассмотрим в следующих разделах. В самом классе Forml описано новое закрытое поле components — контейнер для хранения компонентов, которые можно доба­ вить в класс формы. Конструктор формы вызывает закрытый метод InitializeComponent, автоматиче­ ски формируемый средой (код метода скрыт между директивами препроцессора fregion и fendregion). Этот метод обновляется средой при добавлении элементов управления на форму, а также при изменении свойств формы и содержащихся на ней элементов. Например, если изменить цвет фона формы с помощью окна; свойств (Properties), в методе появится примерно такая строка: this.BackColor = System.Drawing.SystemColors.AppWorkspace: Метод освобождения ресурсов Di spose вызывается автоматически при закрытии] формы. Точка входа в приложение, метод Mai п, содержит вызов статического мето^ Run класса Application. Метод Run запускает цикл обработки сообщений и выво, на экран форму, новый экземпляр которой передан ему в качестве параметра Шаблон Windows-приложения 317 Рис. 14.4 . Элементы управления Windows.Forms ПРИМЕЧАНИ Е Запуск приложения, для создания которого мы пока не написали ни строчки кода, можно выполнить с помощью команды меню Debu g • Star t или клавиши F5. На эк­ ран выводится пустая форма, обладающая стандартной функциональностью окна Windows-приложения: например, она умеет изменять свои размеры и реагировать на действия с кнопками развертывания и закрытия. Процесс создания Windows-приложения состоит из двух основных этапов: 1. Визуальное проектирование, то есть задание внешнего облика приложения. 2. Определение поведения приложения путем написания процедур обработки со­ бытий. Визуальное проектирование заключается в помещении на форму компонентов (элементов управления) и задании их свойств и свойств самой формы с помо­ щью окна свойств. Если его не видно, можно воспользоваться командой меню 318 Глава 14. Введение в программирование под Windows View • Properties Window. Свойства отображаются либо в алфавитном порядке, либо сгруппированными по категориям. Способ отображения выбирается с помо­ щью кнопок, расположенных в верхней части окна свойств (рис. 14.5). Рис. 14.5 . Окно свойств Самый простой способ размещения компонента — двойной щелчок на соот­ ветствующем значке палитры компонентов Toolbox (если ее не видно, можно воспользоваться командой меню View • Toolbox), при этом компонент помеща­ ется на форму. Затем компонент можно переместить и изменить его размеры с помощью мыши. Можно также сделать один щелчок на палитре и еще один щелчок в том месте формы, где планируется разместить компонент. В окне свойств отображаются свойства выбранного в данный момент компонента (он окружен рамкой). Задание свойств выполняется либо выбором имеющихся в списке вариантов, либо вводом требуемого значения с клавиатуры. Если около имени свойства стоит значок +, это означает, что свойство содержит другие свойства. Они становятся доступными после щелчка на значке. Пример формы, на которой размещены ком­ поненты с вкладки палитры Windows Forms, показан на рис. 14.6. Эти компонен­ ты описаны в разделе «Элементы управления» (см. с. 325). Определение поведения программы начинается с принятия решений, какие дейст­ вия должны выполняться при щелчке на кнопках, вводе текста, выборе пунктов меню ит. д., иными словами, по каким событиям будут выполняться действия, реализующие функциональность программы. аблон Windows-приложения 319 Рис. 14.6 . Форм а с размещенными на ней компонентами ЗНИМАНИ Е ; Интерфейс программы должен быть интуитивно понятным и по возможности про­ стым. Часто повторяющиеся действия не должны требовать от пользователя вы­ полнения сложных последовательностей операций. Заготовка шаблона обработчика события формируется двойным щелчком на поле, расположенном справа от имени соответствующего события на вкладке Events окна свойств, при этом появляется вкладка окна редактора кода с заготов­ кой соответствующего обработчика (вкладка Events отображается в окне свойств при щелчке на кнопке с желтой молнией, как показано на рис. 14.5). Для каждого класса определен свой набор событий, на которые он может реаги­ ровать. Наиболее часто используемые события: • Acti vated — получение формой фокуса ввода; • Click, Doubleclick — одинарный и двойной щелчки мышью; • Closed — закрытие формы; • Load — загрузка формы; • KeyDown, KeyUp — нажатие и отпускание любой клавиши и их сочетаний; • Keypress — нажатие клавиши, имеющей ASCII-код; • MouseDown, MouseUp — нажатие и отпускание кнопки мыши; • MouseMove — перемещение мыши; • Paint — возникает при необходимости прорисовки формы. В листинге 14.2 приведен текст программы, сформированной средой после раз­ мещения на форме компонентов, представленных на рис. 14.5, и выборе для кнопки события Click, а для поля ввода — события KeyPress. Из листинга для удобства восприятия удалены лишние элементы. Листин г 14.2 . Шаблон приложения с двумя компонентами и заготовками обработчиков событий using System; у using System.Drawing: using System.Collections: продолжение & 320 Глава 14. Введение в программирование под Windows Листин г 14.2 {продолжение) using System.ComponentModel; using System.Windows.Forms; using System.Data; namespace WindowsApplicationi { public class Forml : System.Windows.Forms.Form // 0 { private System.Windows.Forms.TextBox textBoxl; // 1 private System.Windows.Forms.Button buttonl; // 2 private System.ComponentModel.Container components = null; public FormlО { 1л i t i a1izeComponent(): protected override void DisposeC bool disposing ){... } #region Windows Form Designer generated code private void InitializeComponentO { this.textBoxl = new System.Windows.Forms.TextBoxO: // 3 this.buttonl = new System.Windows.Forms.ButtonO; // 4 this.SuspendLayoutO; // 5 // // textBoxl // this.textBoxl.Location = new System.Drawing.Point(24, 16); this.textBoxl.Name = "textBoxl": this.textBoxl.Size = new System.Drawing.Size(240. 20); this.textBoxl.Tablndex = 0; this.textBoxl.Text = "textBoxl"; this.textBoxl.KeyPress += new // 6 System.Wi ndows.Forms.KeyPressEventHandler(thi s.textBoxl\_KeyPress); // // buttonl // this.buttonl.Location = new System.Drawing.Point(192. 80); this.buttonl.Name = "buttonl"; this.buttonl.Tablndex =1 ; this.buttonl.Text = "buttonl"; this.buttonl.Click += new 111 System.EventHandler(this.buttonl\_Cl ick); // // Forml // this.AutoScaleBaseSize = new System.Drawing.Size(5. 13); this.ClientSize = new System.Drawing.Size(292, 126): this.Controls.Add(this.buttonl); // 8 Шаблон Windows-приложения 321 thi s.Controls.Add(thi s.textBoxl) this.Name = "Forml"; this.Text = "Forml"; this.ResumeLayout(false); } lendregion static void MainO Application.Run(new Forml 0) ; private void buttonl\_Click(object sender. System.EventArgs e) // 11 private void textBoxl\_KeyPress(object sender. System.Windows.Forms.KeyPressEventArgs e) // 12 } } Рассмотрим текст программы. В операторах 1 и 2 в класс формы добавляются два закрытых поля: строка ввода типа TextBox и кнопка типа Button. Эти типы опреде­ лены в пространства имен System.Windows .Forms, которое подключается в соответ­ ствующей директиве using, поэтому можно записать эти и соседние строки проще: public class Forml : Form //О { private TextBox textBoxl; // 1 private Button buttonl; 111 private Container components = null; Чаще всего программист изменяет имена, присваиваемые элементам средой, более осмысленными. Это можно сделать, задав новое значение свойства Name элемента. ПРИМЕЧАНИ Е — Того же самого эффекта можно достичь и с помощью вкладки Class View (View • Class View), выбрав в списке нужный элемент и изменив значение его свойства Name в окне свойств. Обратите внимание на то, что при этом в нем отображаются свойства кнопки buttonl не как элемента интерфейса, а как поля класса. Можно из­ менить имена и вручную, но это более трудоемко и чревато ошибками. Самое важное происходит в методе InitializeComponent. В операторах 3 и 4 создаются экземпляры компонентов, затем для каждого из них задаются свойства, определяющие их положение, размер, вид и т. д. Обратите // 9 // 10 322 Глава 14. Введение в программирование под Windows внимание на операторы 6 и 7. В них регистрируются обработчики соответствую­ щих событий. Механизм обработки событий тот же, что мы рассматривали в гла­ ве 10 (см. с. 232) — он описывается моделью «публикация — подписка». Например, для кнопки buttonl, в составе которой есть событие С! ick, регистриру­ ется обработчик buttonl\_Click, являющийся закрытым методом класса Forml. Это значит, что при наступлении события нажатия на кнопку (об этом сообщит опе­ рационная система) будет вызван соответствующий обработчик. Имя обработчика сформируется средой автоматически из имени экземпляра компо­ нента и имени события. Обратите внимание на то, что обработчикам передаются два параметра: объект-источник события и запись, соответствующая типу события. ПРИМЕЧАНИ Е При задании обработчика можно задать и другое имя, для этого оно записывается справа от имени соответствующего события на вкладке Events окна свойств. После создания экземпляров компонентов и настройки их свойств компоненты заносятся в коллекцию, доступ к которой выполняется через свойство Controls (операторы 8 и 9). Если этого не сделать, компоненты не будут отображаться на форме. Коллекция поддерживает методы добавления и удаления компонентов (Add и Remove). Таким образом, для размещения компонента на форме необходимо выполнить три действия: 1. Создать экземпляр соответствующего класса. 2. Настроить свойства экземпляра, в том числе зарегистрировать обработчик событий. 3. Поместить экземпляр в коллекцию компонентов формы. Операторы 5 и 10 используются для того, чтобы все изменения в свойства эле­ ментов вносились одновременно. Для этого в операторе 5 внесение изменений «замораживается», а в операторе 10 разрешается. В теле обработчиков событий (операторы 11 и 12) программист может наконец- то самостоятельно написать код, который будет выполняться при наступлении события. Добавим в эти строки операторы, выводящие окна сообщений с соответ­ ствующим текстом: Здесь используется простейший вариант статического метода Show класса MessageBox, определенного в пространстве имен System.Windows.Forms. Существуют более деся­ ти перегруженных вариантов этого метода, позволяющих варьировать вид выво­ димой информации, например задать заголовок окна и наименования отображае­ мых на нем кнопок. Прежде чем приступить к изучению форм и элементов управления, размещаемых\* на формах, необходимо рассмотреть их общего предка — класс Control. MessageBox.Show("Нажата кнопка buttonl"); MessageBox.Show("Нажата клавиша " + e.KeyChar); // 11 // 12 Класс Control 323 Класс Control Класс Control является базовым для всех отображаемых элементов, то есть эле­ ментов, которые составляют графический интерфейс пользователя, например кнопок, списков, полей ввода и форм. Класс Control реализует базовую функцио­ нальность интерфейсных элементов. Он содержит методы обработки ввода поль­ зователя с помощью мыши и клавиатуры, определяет размер, положение, цвет фона и другие характеристики элемента. Для каждого объекта можно опреде­ лить родительский класс, задав свойство Parent, при этом объект будет иметь, на­ пример, такой же цвет фона, как и его родитель 1 . Наиболее важные свойства класса Control церечислены в табл. 14.2, методы — в табл. 14.3. Таблица 14.2. Основные свойства класса Control Свойство Описание Anchor BackColor, Backgroundlmage, Font, ForeColor, Cursor Bottom, Right Top, Left Bounds CIientRectangle ContextMenu Dock Location Height, Width Size Определяет, какие края элемента управления будут привязаны к краям родительского контейнера. Если задать привязку всех краев, элемент будет изменять размеры вместе с родительским Определяют параметры отображения рабочей области формы: цвет фона, фоновый рисунок, шрифт, цвет текста, вид указателя мыши Координаты нижнего правого угла элемента. Могут устанавливаться также через свойство Size Координаты верхнего левого угла элемента. Эквивалентны свойству Location Возвращает объект типа Rectangle (прямоугольник), который определяет размеры элемента управления Возвращает объект Rectangle, определяющий размеры рабочей области элемента Определяет, какое контекстное меню будет выводиться при щелчке на элементе правой кнопкой мыши Определяет, у какого края родительского контейнера будет отображаться элемент управления Координаты верхнего левого угла элемента относительно верхнего левого угла контейнера, содержащего этот элемент, в виде структуры типа Pol nt. Структура содержит свойства X и Y Высота и ширина элемента Высота и ширина элемента в виде структуры типа Size. Структура содержит свойства Height и Width продолжение & 1 Речь идет не о наследовании, а о взаимоотношениях объектов во время выполнения про­ граммы. Например, если на форме размещена кнопка, форма является родительским объектом по отношению к кнопке. 324 Глава 14. Введение в программирование под Windows Таблиц а 14. 2 {продолжение) Свойство Описание Created, Disposed, Enabled, Focused, Visible Handle ModifierKeys MouseButtons Opacity Parent Region Tablndex, TabStop Возвращают значения типа bool, определяющие текущее состояние элемента: создан, удален, использование разрешено, имеет фокус ввода, видимый Возвращает дескриптор элемента (уникальное целочисленное значение, сопоставленное элементу) Статическое свойство, используемое для проверки состояния модифицирующих клавиш (Shift, Control, Alt). Возвращает результат в виде объекта типа Keys Статическое свойство, проверяющее состояние клавиш мыши. Возвращает результат в виде объекта типа MouseButtons Определяет степень прозрачности элемента управления. Может изменяться от 0 (прозрачный) до 1 (непрозрачный) Возвращает объект, родительский по отношению к данному (имеется в виду не базовый класс, а объект-владелец) Определяет объект Region, при помощи которого можно управлять очертаниями и границами элемента управления Используются для настройки последовательности перемещения с помощью клавиши Tab по элементам управления, расположенным на форме Таблиц а 14.3 . Основные методы класса Control Метод Описание Focus GetStyle, SetStyle Hide, Show Invalidate QnXXXX Refresh SetBounds, SetLocation, SetClientArea Установка фокуса ввода на элемент 1 Получение и установка флагов управления стилем элемента. Используются значения перечисления Control Styles (см. далее) Управление свойством Visible (Hide — скрыть элемент, Show — отобразить элемент) Обновление изображения элемента путем отправки соответствующего сообщения в очередь сообщений. Метод перегружен таким образом, чтобы можно было обновлять не всю область, занимаемую элементом, а лишь ее часть Методы-обработчики событий (OnMouseMove, OnKeyDown, OnResize, OnPaint и т. п.), которые могут быть замещены в производных классах Обновление элемента и всех его дочерних элементов Управление размером и положением элемента 1 В элемент, имеющий фокус ввода, направляется ввод пользователя с клавиатуры. Элементы управления 325 Перечисление Control Styles задает возможные значения стиля формы в виде бито­ вых флагов, поэтому можно использовать их комбинации. Значения всех констант перечисления можно посмотреть в электронной документации, а для первого знакомства достаточно одного — ResizeRedraw. Этот стиль определяет, что при из­ менении размеров формы она будет автоматически перерисована. По умолчанию перерисовка не выполняется, и если на форме есть какое-либо изображение, ре­ зультат изменения размеров формы может сильно озадачить. В табл. 14.4 перечислена небольшая часть событий, определенных в классе Control. Таблица 14.4 . Некоторые события класса Control Событие Click, Doubleclick, MouseEnter, MouseLeave, MouseDown, MouseUp, MouseMove, MouseWheel KeyPress, KeyUp, KeyDown BackColorChanged, ContextMenuChanged, FontChanged, Move, Paint, Resize GotFocus, Leave, LostFocus Описание События от мыши События от клавиатуры События изменения элемента События получения и потери фокуса ввода Применение наиболее важных элементов, описанных в таблицах, рассматривает­ ся в следующих разделах. При написании приложений применяются два способа обработки событий: • замещение стандартного обработчика; • задание собственного обработчика. В большинстве случаев применяется второй способ. Среда разработки создает заготовку обработчика по двойному щелчку на поле, расположенном справа от имени соответствующего события на вкладке Events окна свойств. При этом в код приложения автоматически добавляется строка, регистрирующая этот об­ работчик. Первый способ, то есть переопределение виртуальных методов OnXXXX (OnMouseMove, OnKeyDown, OnResize, OnPaint и т. п.), применяется в основном тогда, когда перед ре­ акцией на событие требуется выполнить какие-либо дополнительные действия. За подробностями интересующиеся могут обратиться к [27]. Элементы управления Элементы управления, или компоненты, помещают на форму с помощью панели инструментов ToolBox (View • ToolBox). В этом разделе кратко описаны простей­ шие компоненты и приведены примеры их использования. Обычно компоненты применяют в диалоговых окнах для получения данных от пользователя или его информирования. 326 Глава 14. Введение в программирование под Windows Метка Label Метка предназначена для размещения текста на форме. Текст хранится в свойстве Text. Можно задавать шрифт (свойство Font), цвет фона (BackColor), цвет шрифта (ForeColor) и выравнивание (TextAlign) текста метки. Метка может автоматиче­ ски изменять размер в зависимости от длины текста (AutoSize = True). Можно разместить на метке изображение (Image) и задать прозрачность (установить для свойства BackColor значение Col or. Transparent). Метка не может получить фокус ввода, следовательно, не обрабатывает сообще­ ния от клавиатуры. Пример использования меток приведен далее (см. с. 327). Кнопка Button Основное событие, обрабатываемое кнопкой, — щелчок мышью (Click). Кроме того, кнопка может реагировать на множество других событий — нажатие клавиш на кла­ виатуре и мыши, изменение параметров и т. д. Нажатие клавиши Enter или пробела, если при этом кнопка имеет фокус ввода, эквивалентно щелчку мышью на кнопке. Можно изменить начертание и размер шрифта текста кнопки, который хранится в свойстве Text, задать цвет фона и фоновое изображение так же, как и для метки. Если занести имя кнопки в свойство AcceptButton формы, на которой расположе­ на кнопка, то нажатие клавиши Enter вызывает событие Click, даже если кнопка не имеет фокуса ввода. Такая кнопка имеет дополнительную рамку и называется кнопкой по умолчанию. Аналогично, если занести имя кнопки в свойство Cancel Button формы, на которой расположена кнопка, то нажатие клавиши Esc вызывает событие Click для этой кнопки. Кнопки часто используются в диалоговых окнах. Как видно из названия, такое окно предназначено для диалога с пользователем и запрашивает у него какие- либо сведения (например, какой выбрать режим или какой файл открыть). Диа­ логовое окно обладает свойством модальности. Это означает, что дальнейшие действия с приложением невозможны до того момента, пока это окно не будет закрыто. Закрыть окно можно, либо подтвердив введенную в него информацию щелчком на кнопке ОК (или Yes), либо отменив ее с помощью кнопки закрытия окна или, например, кнопки Cancel. Таблица 14.5 . Значения перечисления DialogResult Значение Описание Значение Описание None Окно не закрывается Ignore Нажата кнопка Ignore OK Нажата кнопка ОК Yes Нажата кнопка Yes Cancel Нажата кнопка Cancel No Нажата кнопка No Abort Нажата кнопка Abort Retry Нажата кнопка Retry Элементы управления 327 Для сохранения информации о том, как было закрыто окно, у кнопки определяют свойство DialogResult. Это свойство может принимать стандартные значения из пе­ речисления DialogResult, определенного в пространстве имен System. Wi ndows. Forms. Значения перечисления приведены в табл. 14.5. Пример использования кнопок приведен в следующем разделе. Поле ввода TextBox Компонент TextBox позволяет пользователю вводить и редактировать текст, кото­ рый запоминается в свойстве Text. Можно вводить строки практически неограни­ ченной длины (приблизительно до 32 ООО символов), корректировать их, а также вводить защищенный текст (пароль) путем установки маски, отображаемой вме­ сто вводимых символов (свойство PasswordChar). Для обеспечения возможности ввода нескольких строк устанавливают свойства Multiline, Scrol 1 Bars и Wordwrap. Доступ только для чтения устанавливается с помощью свойства Readonly. Элемент содержит методы очистки (Clear), выделения (Select), копирования в буфер (Сору), вставки из него (Paste) и др., а также реагирует на множество со­ бытий, основными из которых являются KeyPress и KeyOown. Рассмотрим создание простого приложения, в котором использованы компонен­ ты типа Label, Button и TextBox. Главное окно приложения показано на рис. 14.7. Рис. 14.7 . Окно приложения «Диагностика кармы» Пользователь вводит число и нажимает клавишу Enter, после чего введенное зна­ чение сравнивается с числом, «задуманным» генератором случайных чисел. Если пользователь не угадал, выводится сообщение «Н е угадали!». Каждое следующее сообщение для наглядности немного сдвигается вправо относительно предыдуще­ го. После совпадения выводится итоговое сообщение, и фокус ввода передается на кнопку Еще раз (рис. 14.8). «Коэффициент невезучести» определяется как ко­ личество попыток, деленное на максимально возможное значение числа. Создание приложения начинается с проектирования его интерфейса. На форме располагаются метки, поля ввода и кнопка. В окне свойств их свойству Text зада­ ются перечисленные в табл. 14.6 значения (в окне отображаются свойства выде­ ленного элемента; элемент выделяется щелчком мыши). 328 Глава 14. Введение в программирование под Windows Рис. 14.8 . Окно приложения после ввода правильного ответа Таблиц а 14.6 . Значение свойства Text для компонентов приложения Компонент Значение свойства Text Forml Диагностика кармы Введите число Пустое значение 1 Еще раз label 1 label2, label3, textBoxl buttonl Поведение приложения задается в обработчиках событий. Число надо загадывать до того, как пользователь введет первый ответ, а затем после каждого щелчка на кнопке Еще раз. Следовательно, эти действия должны выполняться при загрузке формы (событие Load) и щелчке на кнопке (событие CI i ck). Ответ пользователя анализируется при нажатии клавиши Enter после ввода числа (событие KeyPress элемента textBoxl). Вызов редактора программ выполняется двойным щелчком рядом с соответст­ вующим событием элемента на вкладке Events окна свойств (см. рис. 14.5). Для хранения загаданного числа и ответа пользователя применяются поля i и к. В кон­ станте max хранится максимально возможное число. Переменная rnd представля­ ет собой экземпляр генератора случайных чисел 2 . Сокращенный текст программы с комментариями приведен в листинге 14.3. Листин г 14.3 . Программа «Угадай число» using System; using System.Drawing; using System.Collections; using System.ComponentModel; using System.Windows.Forms; using System.Data; 1 Необходимо стереть значение, заданное по умолчанию. 2 Класс Random описан в одноименном разделе (см. с. 148). Элементы управления 329 namespace WindowsApplicationl { public class Forml : Form { private Label labell: private TextBox textBoxl; private Button buttonl; private Label label2; private Label label3; private Container components = null; public FormlО { ... } protected override void Dispose( bool disposing ){... } Windows Form Designer generated code { ... } const int max - 10; // максимальное значение загадываемого числа Random rnd; // экземпляр генератора случайных чисел int i, k; II загадка и отгадка static void MainO { ... } private void Forml\_Load(object sender, EventArgs e) { rnd = new RandomO; i = rnd.Next(max); // загадано число в диапазоне от 0 до max } private void textBoxl\_KeyPress(object sender, KeyPressEventArgs e) { int n; if ( e.KeyChar != (char)13 ) return; // если нажата не клавиша Enter, выйти try // преобразование ввода пользователя в число { n = Convert.ТоInt32(textBoxl.Text); } catch { n = -1; // если введено не число, принять за неверный ответ } if ( n != i ) // ========== пользователь не угадал { label 2.Left += 5; label2.Text = "He угадали!"; textBoxl. ClearO; k++; // увеличение счетчика попыток buttonl.Visible = false; } продолжение # 330 Глава 14. Введение в программирование под Windows Листин г 14.3 (продолжение) else { пользователь угадал 1abel2.Left = 32: // восстановить начальное положение метки 1abel2.Text = "Коэффициент невезучести"; double koef = 1.0 \* k / max: label3.Text = koef.ToStringO; buttonl.Visible = true; } private void buttonl\_Click(object sender. System.EventArgs e) { i = rnd.Next(max) k = 0; textBoxl.CI ear() textBoxl.Focus О label 2.Text = "" 1abel3.Text = "" //загадано число в диапазоне от 0 до max // обнуление количества попыток ответа // поле ввода очищено // курсор установлен на поле ввода // метки очищены Меню MainMenu и ContextMenu Главное меню MainMenu размещается на форме таким же образом, как и другие компоненты: двойным щелчком на его значке на панели Toolbox. При этом зна­ чок располагается под заготовкой формы, а среда переходит в режим редакти­ рования пунктов меню. Каждый пункт меню представляет собой объект типа Menultem, и при вводе пункта меню мы задаем его свойство Text. Переход к зада­ нию заголовка следующего пункта меню выполняется либо щелчком мыши, либо нажатием клавиши Enter и клавиш со стрелками. Обычно, чтобы сделать программу понятнее, изменяют также свойства Name каждого пункта так, чтобы они соответствовали названиям пунктов. Пункт меню может быть запрещен или разрешен (свойство Enabled), видим или невидим (Visible), отмечен или не отмечен (Checked). Заготовка обработчика со­ бытий формируется двойным щелчком на пункте меню. Любое приложение обычно содержит в меню команду Exit, при выборе которой приложение завершается. Для закрытия приложения можно воспользоваться либо методом Close класса главной формы приложения, либо методом Exit клас­ са Appl i cati on, например: private void Exit\_Click(object sender, EventArgs e) { // имя пункта меню - Exit CloseO: // или: // Application.ExitO; } Элементы управления 331 Контекстное меню. — это меню, которые вызывается во время выполнения про­ граммы по нажатию правой кнопки мыши на форме или элементе управления. Обычно в этом меню размещаются пункты, дублирующие пункты главного меню, или пункты, определяющие специфические для данного компонента действия. Контекстное меню ContextMenu создается и используется аналогично главному (значок контекстного меню появляется на панели инструментов, если восполь­ зоваться кнопкой прокрутки). Для привязки контекстного меню к компоненту следует установить значение свойства ContextMenu этого компонента равным име­ ни контекстного меню. Флажок CheckBox Флажок используется для включения-выключения пользователем какого-либо режима. Для проверки, установлен ли флажок, анализируют его свойство Checked, принимающее значение true или false. Флажок может иметь и третье состоя­ ние — «установлен, но не полностью». Как правило, это происходит в тех случаях, когда устанавливаемый режим определяется несколькими «подрежимами», часть из которых включена, а часть выключена. В этом случае используют свойство CheckState, которое может принимать значения Checked, Unchecked и Intermediate. Кроме того, флажок обладает свойством ThreeState, которое управляет возмож­ ностью установки третьего состояния пользователем с помощью мыши. Для флаж­ ка можно задать цвет фона и фоновое изображение так же, как и для метки. Свой­ ство Appearance управляет отображением флажка: либо в виде собственно флажка (Normal), либо в виде кнопки (Button), которая «залипает» при щелчке на ней мышью. Флажки используются в диалоговых окнах как поодиночке, так и в группе, при­ чем все флажки устанавливаются независимо друг от друга. Пример приложе­ ния с флажками приведен далее в этой главе. Переключатель RadioButton Переключатель позволяет пользователю выбрать один из нескольких предложен­ ных вариантов, поэтому переключатели обычно объединяют в группы. Если один из них устанавливается (свойство Checked), остальные автоматически сбрасыва­ ются. Программист может менять стиль и цвет текста, связанного с переключа­ телем, и его выравнивание. Для переключателя можно задать цвет фона и фоно­ вое изображение так же, как и для метки. Переключатели можно поместить непосредственно на форму, в этом случае все они составят одну группу. Если на форме требуется отобразить несколько групп переключателей, их размещают внутри компонента Group или Panel. Свойство Appearance управляет отображением переключателя: либо в традици­ онном виде (Normal), либо в виде кнопки (Button), которая «залипает» при щелч­ ке на ней мышью 1 . Пример использования переключателей приведен далее в этой главе. 1 При установке для свойства AutoCheck значения false кнопка не «залипает». 332 Глава 14. Введение в программирование под Windows Панель GroupBox Панель GroupBox служит для группировки элементов на форме, например для того, чтобы дать общее название и визуально выделить несколько переключа­ телей или флажков, обеспечивающих выбор связанных между собой режимов. Приведенная в листинге 14.4 программа запрашивает у пользователя, массив ка­ кой длины он хочет создать, и создает целочисленный массив с помощью генера­ тора случайных чисел. Пользователь может выбрать диапазон значений элемен­ тов.'либо [-10; 10], либо [-100; 100]. После создания массива можно вычислить его максимальный элемент и/или количество положительных элементов. Окно приложения показано на рис. 14.9. Для удобства имена (свойство Name) большин­ ства компонентов изменены, как указано на рисунке. Переключатели размещены на панели типа GroupBox. Установлено свойство Checked компонента radioButtonl, очищены все поля ввода, а для полей maxtextBox и numPosittextBox установлено свойство Readonly (только для чтения). " Вычнс ieни я в масснвс И!П!!х| Количество \_ рЗнмацпдамвилж j j { < 0 ) n = -n; // если введено отрицательное число агг = new int[n]; // создание массива for ( int i = 0; i < n; ++i ) { arr[i ] = rnd.Next(a. b); // задание элемента массива arraytextBox.Text += " " + arr[i.]; // вывод массива } } private void calcbutton\_Click(object sender, EventArgs e) { int max = arr[0]; int numPosit = 0; for ( int i = 0; i < arr.Length; ++i ) { if ( arr[i ] > max ) max = arr[i] ; // поиск максимума if ( arr[i ] > 0 ) ++numPosit; // количество положительных } if ( maxcheckBox.Checked ) maxtextBox. Text = max.ToStringO; else maxtextBox.Text = ""; if ( numPositcheckBox.Checked ) numPosittextBox.Text = numPosit.ToStringO; else numPosittextBox.Text = ""; } } } Список ListBox Список служит для представления перечней элементов, в которых пользователь может выбрать одно (свойство Sel ecti onMode равно One) или несколько значений (свойство Sel ecti onMode равно Multi Simple или Multi Extended). Если значение свой­ ства SelectionMode равно MultiSimple, щелчок мышью на элементе выделяет его или снимает выделение. Значение Multi Extended позволяет использовать при вы­ делении диапазона строк клавишу Shift, а при добавлении элемента — клавишу Ctrl, аналогично проводнику Windows. Запретить выделение можно, установив значение свойства SelectionMode, равное None. Чаще всего используются списки строк, но можно выводить и произвольные изо­ бражения. Список может состоять из нескольких столбцов (свойство MultiColumn) и быть отсортированным в алфавитном порядке (Sorted = True). Элементы списка нумеруются с нуля. Они хранятся в свойстве Items, представляю­ щем собой коллекцию. В Items можно добавлять элементы с помощью методов шементы управления 335 wjd, AddRange и Insert. Для удаления элементов служат методы Remove и RemoveAt, /даляющие заданный элемент и элемент по заданному индексу соответственно. Выделенные элементы можно получить с помощью свойств Selectedltems и SelectedIndices, предоставляющих доступ к коллекциям выделенных элементов и их ин­ дексов. В листинге 14.5 приведен пример приложения, которое отображает в списке гипа Li stBox строки, считанные из входного файла, а затем по щелчку на кнопке Запис ь выводит выделенные пользователем строки в выходной файл. Вид окна приложения приведен на рис. 14.10. " Form l | Л \_ П | [ У SystemObjec t SystemMashalByRefObjec t System. ComporrertModeLComponen t System. Windows.Forms.Conttol System. Windows.Forms.ScrollableControl System.Windows.Foims.ContainerContiol System. Windows.Forms.For m Загчсь J Рис. 14.10 . Окно приложения для работы со списко м строк Листин г 14.5 . Работа со списко м строк using System; using System.10; using System.Drawing; using System.Col lections; using System.ComponentModel; using System.Windows.Forms; using System.Data; using System.Collections.Specialized; namespace WindowsApplicationi { public class Forml : Form { private ListBox listBoxl; private Button buttonl; private Container components = null; public FormlO { .. . } protected override void DisposeC bool disposing ){... } Windows Form Designe 3 1 r generated code { ... } л ' продолжение & 336 Глава 14. Введение в программирование под Windows Листин г 14. 5 (продолжение) static void MainO { ... } private void Forml\_Load(object sender. EventArgs e) { try { StreamReader f = new StreamReaderC "input.txt" ); string buf; while ( ( buf = f.ReadLineO ) != null ) // чтение из файла listBoxl.Items.Add(buf); // занесение в список } catch ( FileNotFoundException exc ) { MessageBox.Show( exc.Message ); return; } } private void buttonl\_Click(object sender. EventArgs e) { StreamWriter f = new StreamWriter( "output.txt" ); foreach ( string item in listBoxl.Selectedltems ) f.WriteLine(item); // запись в файл f.CloseO; } } } На панели инструментов расположено множество компонентов, не рассмотренных в этой книге. Для их изучения следует обратиться к справочной системе .NET. Она организована понятно и единообразно, и даже те, кто не изучал английский язык, смогут при некотором навыке извлечь оттуда необходимую информацию1 . СОВЕ Т При изучении компонента рекомендуется следующая последовательность действий. Разместите компонент на форме, выделите его, нажмите клавишу F1 и перейдите по ссылке ...overview (обзор). Изучите разделы Remarks и Example, затем перейдите по ссылке ...Members (элементы класса), расположенной в верхней части окна Попы­ тайтесь получить представление о возможностях изучаемого класса, выделив главные из его свойств и открытых методов. После этого можно вернуться к заготовке при­ ложения и начать экспериментировать со свойствами, а затем — с методами класса В следующей части этой главы будут коротко описаны основные элементы клас сов Form и Application, входящих в состав любого приложения. 1 Но лучше, конечно, хотя бы немного знать английский язык, это полезно само по себ< арительные замечания о формах 337 мэдварительные замечания о формах ;с Form наследует от длинной цепочки своих предков множество элемен- определяющих вид и поведение окон различного типа. Генеалогическое ю класса Form выглядит так: Object-^MarshalByRefObject-»Component->Control-» HableControl—>ContainerControl. ia приложения могут иметь различные вид и назначение. Все окна можно телить на модальные и немодальные. Модальное окно не позволяет пользова- ю переключаться на другие окна того же приложения, пока не будет заверше- работа с текущим окном 1 . Как уже отмечалось, в виде модальных обычно эрмляют диалоговые окна, требующие от пользователя ввода какой-либо ин- рмации. Модальное окно можно закрыть щелчком на кнопке наподобие ОК, утверждающей введенную информацию, на кнопке закрытия окна или на опке вроде Cancel, отменяющей ввод пользователя. Примером модального на может служить окно сообщений MessageBox, упоминавшееся в разделе «Шаб- н Windows-приложения» (см. с. 322). модальное окно позволяет переключаться на другие окна того же приложения, емодальные окна являются, как правило, информационными. Они использу- гся в тех случаях, когда пользователю желательно предоставить свободу выбо- i — оставлять на экране какую-либо информацию или нет. аждое приложение содержит одно главное окно. Класс главного окна приложе- ия содержит точку входа в приложение (статический метод Main). При закры­ ли главного окна приложение завершается. i случае использования многодокументного интерфейса (Multiple Document nterface, MDI ) одно родительское окно может содержать другие окна, называе- ше дочерними. При закрытии родительского окна дочерние окна закрываются штоматически. Вид окна определяет его функциональность, например, окно с оди- гарной рамкой не может изменять свои размеры. Рассмотрим наиболее интересных предков класса формы. Их элементы наследу­ ются не только формой, но и другими компонентами, такими как поля ввода или кнопки. Класс MarshalByRefObject наделяет своих потомков некой особенностью, благода­ ря которой обращение к ним выполняется по ссылке, то есть локальная копия объекта не создается. Класс Component обеспечивает потомков способностью взаимодействовать с кон- ТСЙНером, В котором они расположены. Кроме того, в нем определен метод Di spose, который автоматически вызывается, когда экземпляр класса более не использу­ ется. Поэтому для освобождения ресурсов, связанных с приложением, обычно переопределяют этот метод. 1 Особый вид модального окна - системное модальное окно - не позволяет переключать­ ся даже на окна других приложений. 338 Глава 14. Введение в программирование под Windows Класс Control, являющийся предком всех интерфейсных элементов, рассмотрен в этой главе ранее. В классе ScrollableControl определены элементы, позволяющие компоненту иметь горизонтальную и вертикальную полосы прокрутки. Свойства AutoScrol 1 и AutoScrol IMinSize обеспечивают автоматическое появление полос про­ крутки в тех случаях, когда выводимая информация не помещается в компоненте. Класс ContainerControl обеспечивает своих потомков возможностью управлять раз­ мещенными внутри них дочерними компонентами. Например, на форме обычно располагаются несколько кнопок, меток и т. п., а на панели — несколько флажков или переключателей. Свойства и методы класса позволяют установить фокус ввода на элемент или получать информацию о том, какой элемент имеет фокус ввода, а также управлять порядком получения фокуса с помощью свойств TabStop и Tablndex. Класс Form Класс Form представляет собой заготовку формы, от которой наследуются классы форм приложения. Помимо множества унаследованных элементов, в этом классе определено большое количество собственных элементов, наиболее употребитель­ ные из которых приведены в табл. 14.7-14.9. Таблиц а 14.7 . Некоторые свойства класса Form Свойство Описание AcceptButton ActiveMDIChild, IsMDIChild, IsMDIContainer AutoScale FormBorderStyle Cancel Button Control Box Menu, MergedMenu MaximizeBox, MinimizedBox Позволяет задать кнопку или получить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Enter Свойства предназначены для использования в приложениях с многодокументным интерфейсом (MDI ) Позволяет установить или получить значение, определяющее, будет ли форма автоматически изменять свои размеры, чтобы соответствовать высоте шрифта, используемого на форме, или размерам размещенных на ней компонентов Позволяет установить или получить стиль рамки вокруг формы (используются значения перечисления FormBorderStyle) Позволяет задать кнопку или подучить информацию о кнопке, которая будет активизирована при нажатии пользователем клавиши Esc Позволяет установить или получить значение, определяющее, будет ли присутствовать стандартная кнопка системного меню в верхнем левом углу заголовка формы Используются для установки или получения информации о меню на форме Определяют, будут ли на форме присутствовать стандартные кнопки восстановления и свертывания в правом верхнем углу заголовка формы овые окна 339 :тво Описание nTaskbar Определяет, будет ли форма отображаться на панели задач Windows Position Позволяет получить или установить значение, определяющее исходное положение формы в момент выполнения программы (используются значения перечисления FormStartPosition). Например, для отображения формы в центре экрана устанавливается значение CenterScreen wState Определяет состояние отображаемой формы при запуске (используются значения перечисления FormWi ndowState) и ц а 14.8 . Методы класса Form од Описание vate Активизирует форму и помещает в нее фокус ввода ;е Закрывает форму :erToScreen Помещает форму в центр экрана xitMDI Размещает все дочерние формы на родительской в соответствии со значениями перечисления LayoutMDI esize Может быть замещен для реагирования на событие Resize w Отображает форму (унаследовано от Control) wDialog Отображает форму как диалоговое окно (подробнее о диалоговых окнах рассказывается в следующем разделе) >лица 14.9 . Некоторые, события класса Form •бытие Описание ti vate Происходит при активизации формы (когда она выходит в активном приложении на передний план) osed, Closing Происходят во время закрытия формы HChildActive Возникает при активизации дочернего окна б использовании некоторых элементов, перечисленных в таблицах, рассказы- штся в следующем разделе. Диалоговые окна 5 библиотеке .NET нет специального класса для представления диалоговых «он. Вместо этого устанавливаются определенные значения свойств в классе )бычной формы. В диалоговом окне можно располагать те же элементы управле- 1ия, что и на обычной форме. Диалоговое окно характеризуется: 2 неизменяемыми размерами (FormBorderStyle = FixedDialog); • отсутствием кнопок восстановления и свертывания в правом верхнем углу за­ головка формы (MaximizeBox = False, MinimizedBox = False); 340 Глава 14. Введение в программирование под Windows • наличием кнопок наподобие ОК, подтверждающей введенную информацию, и Cancel, отменяющей ввод пользователя, при нажатии которых окно закры­ вается (AcceptButton = имя\_кнопки\_ОК, Cancel Button = имя\_кнопки\_Сапсе1); • установленным значением свойства DialogResult для кнопок, при нажатии ко­ торых окно закрывается. Для отображения диалогового окна используется метод ShowModal, который фор­ мирует результат выполнения из значений перечисления DialogResult, описан­ ных в табл. 14.5. Если пользователь закрыл диалоговое окно щелчком на кнопке наподобие ОК, введенную им информацию можно использовать в дальнейшей работе. Закрытие окна щелчком на кнопке вроде Cancel отменяет все введенные данные. Диалоговое окно обычно появляется при выборе пользователем некото­ рой команды меню на главной форме. В листинге 14.6 приведен пример простого приложения, содержащего две формы. На главной форме (рис. 14.11) расположено меню из двух команд — Dialog и Exit При выборе команды Dialog появляется диалоговое окно, включающее метку Введите информацию, поле ввода текста и две кнопки, ОК и Cancel (рис. 14.12) Рис. 14.11 . Главное окно приложения с меню и диалоговым окном Рис. 14.12 . Диалоговое окно Если пользователь введет текст в диалоговое окно и закроет его щелчком кнопке ОК, этот текст отобразится в поле метки в главном окне. Если диало вое окно будет закрыто щелчком на кнопке Cancel, поле метки останется не менным. Добавление в проект второй формы выполняется выбором в меню кома\* Project > Add Windows Form. Начнем визуальное проектирование с этой фор Значения свойств компонентов, установленных с помощью окна свойств, п< числены в табл. 14.10. Диалоговые окна 341 Таблица 14.10 . Значения свойств элементов формы Элемент Свойство Значение Метка Text Введите информацию TextAli gn MiddleCenter Поле ввода Text Пустое значение Кнопка Name btnOK DialogResult OK Text OK Кнопка Name btnCancel DialogResult Cancel Text Cancel Форма AcceptButton btnOK Cancel Button btnCancel FormBorderStyle FixedDialog MaximizeBox False MinimizeBox False StartPosition CenterParent Пользователь вводит информацию в поле ввода textBoxl, которое является за­ крытым элементом формы. Чтобы получить значение свойства Text поля ввода, добавим в описание класса свойство Info. Это единственное дополнение, которое вносится в шаблон текста класса формы: public class Form2 : Form { private Label label 1: private TextBox textBoxl: private Button btnOK: private Button btnCancel; private Container components = null; public string Info { get { return textBoxl.Text: } } } Визуальное проектирование главной формы еще проще, чем первой, и сводит­ ся к размещению главного меню и метки. Для наглядности метка окружена 342 Глава 14. Введение в программирование под Windows рамкой (BorderStyle = FixedSingle). Вот как выглядят обработчики событий для пунктов меню: private void menuIteml\_Click( object sender, EventArgs e ) { Form2 f = new Form2(); // создание экземпляра класса окна if ( f.ShowDialogO == DialogResult.OK ) // отображение окна 1abel1.Text = f.Info; } private void menuItem2\_Click( object sender, EventArgs e ) { CloseO; // закрытие главного окна } Как видите, для отображения диалогового окна следует создать экземпляр объекта соответствующей формы, а затем вызвать для этого объекта метод ShowDialog. При подтверждении ввода текст пользователя можно получить с помощью свой­ ства Info, доступного только для чтения. При необходимости передавать инфор­ мацию не только из диалогового окна, но и в него, можно добавить в описание свойства часть set. ПРИМЕЧАНИ Е Следует различать процесс создания формы — объекта класса Form или его наследни­ ка—от процесса вывода формы на экран. Форма, как и любой объект, создается при выполнении операции new с вызовом конструктора. Для вывода формы служит метод Show или ShowDi al од класса Form, вызываемый соответствующим объектом. Для скры­ тия формы используется метод Hide. Фактически, методы Show и Hide изменяют свой­ ство Visible объекта. Разница между скрытием формы методом Hi de и ее закрытием методом CI ose состоит в том, что первый из них делает форму невидимой, но не изменя­ ет сам объект, а метод CI ose делает объект недоступным и закрывает все его ресурсы. Класс Application Класс Application, описанный в пространстве имен System.Windows.Forms, содер­ жит статические свойства, методы и события, предназначенные для управления приложением в целом и получения его общих характеристик. Наиболее важные элементы класса Application перечислены в табл. 14.11. Таблица 14.11 . Основные элементы класса Application Элемент класса Тип Описание AddMessageFiIter, Методы Позволяют перехватывать сообщения и выполнять RemoveMessageFi Iter с этими сообщениями нужные предварительные действия. Для того чтобы добавить фильтр сообщений, необходимо указать класс, реализующий интерфейс IMessageFiIter1 1 Подробности см. в [27]. Класс Application 343 Элемент класса Тип Описание DoEvents Метод Обеспечивает способность приложения обрабатывать сообщения из очереди сообщений во время выполнения какой-либо длительной операции Exit Метод Завершает работу приложения ExitThread Метод Прекращает обработку сообщений для текущего потока и закрывает все окна, владельцем которых является этот поток Run Метод Запускает стандартный цикл обработки сообщений для текущего потока CommonAppOataRegi stry Свойство Возвращает параметр системного реестра, который хранит общую для всех пользователей информацию о приложении CompanyName Свойство Возвращает имя компании CurrentCulture Свойство Позволяет задать или получить информацию о естественном языке, для работы с которым предназначен текущий поток CurrentInputlanguage Свойство Позволяет задать или получить информацию о естественном языке для ввода информации, получаемой текущим потоком ProductName Свойство Позволяет получить имя программного продукта, которое ассоциировано с данным приложением ProductVersion Свойство Позволяет получить номер версии программного продукта StartupPath Свойство Позволяет определить имя выполняемого файла для работающего приложения и путь к нему в операционной системе ApplicationExit Событие Возникает при завершении приложения Idle Событие Возникает, когда все текущие сообщения в очереди обработаны и приложение переходит в режим бездействия ThreadExit Событие Возникает при завершении работы потока в приложении. Если работу завершает главный поток приложения, это событие возникает до события Appl i cati onExi t Многие свойства класса Appl i cati on позволяют получить метаданные сборки (на­ пример, номер версии или имя компании), не используя типы пространства имен System.Reflection. Программист не часто работает непосредственно с клас­ сом Application, поскольку большую часть необходимого кода среда формирует автоматически. 344 Глава 14. Введение в программирование под Windows Краткое введение в графику Для вывода линий, геометрических фигур, текста и изображений необходимо соз­ дать экземпляр класса Graphics, описанного в пространстве имен System.Drawing. Существуют различные способы создания этого объекта. Первый способ состоит в том, что ссылку на объект Graphics получают из пара­ метра PaintEventArgs, передаваемого в обработчик события Paint, возникающего при необходимости прорисовки формы или элемента управления: private void Forml\_Paint( object sender, PaintEventArgs e ) { Graphics g = e.Graphics; // использование объекта } Второй способ — использование метода CreateGraphics, описанного в классах формы и элемента управления: Graphics g; g = this.CreateGraphicsO; Третий способ — создание объекта с помощью объекта-потомка Image. Этот спо­ соб используется для изменения существующего изображения: Bitmap bm = new Bitmap( "d:\\picture.bmp" ); Graphics g = Graphics.FromImage( bm ); После создания объекта типа Graphics его можно применять для вывода линий, геометрических фигур, текста и изображений. Основными объектами, которые при этом используются, являются объекты классов: • Реп — рисование линий и контуров геометрических фигур; • Brush — заполнение областей; • Font — вывод текста; • Col or — цвет. В листинге 14.6 представлен код приложения, в котором на форму выводятся линия, эллипс и текст. Вид формы приведен на рис. 14.13. Листин г 14.6 . Работа с графическими объектами using System; using System.Drawing; using System.Windows.Forms; namespace WindowsApplicationl { public partial class Forml : Form { public FormlО { InitializeComponent(); } private void Forml\_Paint( object sender, PaintEventArgs e ) { using ( Graphics g = e.Graphics ) // 1 { Краткое введение в графику 345 using ( Pen pen = new Pen( Col or. Red ) ) 112 { g.DrawLine( pen. 0, 0, 200. 100 ); g.DrawEllipse( pen. new Rectangle(50, 50, 100, 150) ): } string s = "Sample Text"; Font font = new Font( "Arial", 18 ); /7 3 SolidBrush brush = new SolidBrush( Col or. Black ); //4 float x = 100.OF; float у = 20.OF; g.DrawString( s, font, brush, x. у ); font.DisposeO; // 5 brush.DisposeO: // б Рис. 14.13 . Форм а с графикой Графические объекты потребляют системные ресурсы, поэтому рекомендуется вызывать для них метод освобождения ресурсов Dispose. Для упрощения работы с такими объектами в С# есть оператор using1 со следующим синтаксисом: using ( выделениересурса ) оператор Под ресурсом здесь понимается объект, который реализует интерфейс System. IDisposable, включающий метод Dispose. Код, использующий ресурс, вызовом этого метода сигнализирует о том, что ресурс больше не требуется. Если метод Dispose не был вызван, освобождение ресурса выполняется в процессе сборки мусора. Оператор using неявным образом вызывает метод Dispose в,случае успешного создания и использования объекта. Этот способ применен в операторах 1 и 2. 1 Ключевое слово using используется в С# в двух не связанных между собой случаях: как директива и как оператор. Директива using была рассмотрена в главе 12. 346 Глава 14. Введение в программирование под Windows В операторах 3 и 4 объекты создаются обычным образом, поэтому для них требу­ ется явный вызов Di spose, что и происходит в операторах 5 и 6. Как видно даже из этого простого листинга, для вывода графики требуется кропот­ ливое изучение множества свойств и методов множества стандартных классов, описание которых, во-первых, очень объемное, во-вторых, невыносимо скучное, а в-третьих, не входит в задачу учебника по основам программирования. Рекомендации по программированию Процесс создания Windows-приложения состоит из двух основных этапов, кото­ рые могут чередоваться между собой: это визуальное проектирование приложе­ ния и определение его поведения. При задании внешнего облика приложения следует обратить внимание на стан­ дарты интерфейса Windows-приложений: компания Microsoft, в свое время за­ имствовавшая идею стандартного графического интерфейса у компании Apple, довела эту идею до совершенства, детально регламентировав вид окон, располо­ жение, цветовую гамму и пропорции компонентов. Основная сложность для начинающих заключается в разработке алгоритма: по каким событиям будут выполняться действия, реализующие функциональность программы, какие действия должны выполняться при щелчке на кнопках, вводе текста, выборе пунктов меню и т. д. Интерфейс программы должен быть интуитивно понятным и по возможности простым. Часто повторяющиеся действия не должны требовать от пользователя выполнения сложных последовательностей операций. Команды меню и компо­ ненты, которые не имеет смысла использовать в данный момент, рекомендуете\* делать неактивными. Вопросы, задаваемые пользователю программы, должнь быть ненавязчивыми («Нет, а все-таки Вы действительно хотите удалить это' файл?») и немногословными, но при этом не допускать двояких толкований Эта глава получилась самой длинной из-за большого количества информации справочного характера. Несмотря на это .приведенных сведений совершенно не достаточно для создания реальных Windows-приложений. К сожалению, мош библиотеки .NET имеет оборотную сторону: для освоения необходимой инфор мации требуется много времени и упорства, однако это единственный путь дл тех, кто хочет заниматься программированием профессионально. Конечно, пытаться запомнить все методы и свойства классов нет смысла, до< таточно изучить состав используемых пространств имен, представлять се( возможности их элементов и знать, как быстро найти требуемую информаци] Для дальнейшего изучения возможностей библиотеки можно рекомендовать д кументацию и дополнительную литературу [17], [18], [20], [31]. И последш совет: не следует считать себя программистом только на том основании, что i умеете размещать компоненты на форме! Глава 15 Дополнительные средства С# В этой главе описаны дополнительные средства языка С# и среды Visual Studio: указатели, регулярные выражения и документация в формате XML. В конце гла­ вы дается краткое введение в основные области профессионального применения С #: ASP.NET (веб-формы и веб-службы) и ADO.NET (базы данных). Указатели, без которых не мыслят свою жизнь программисты, использующие С и С++ , в языке С# рекомендуется применять только в случае необходимости, по­ скольку они сводят на нет многие преимущества этого языка. Документирование кода в формате XM L и регулярные выражения применяются шире, но относятся скорее к дополнительным возможностям языка, поэтому не были рассмотрены ранее. Напротив, веб-формы, веб-службы и работа с базами данных являются одними из основных областей применения С#, но не рассматриваются в этой книге из-за того, что подобные темы обычно не входят в базовый курс программирования, поскольку для их полноценного освоения требуется иметь базовые знания в об­ ласти сетей, баз данных, протоколов передачи данных и т. п. Небезопасный код Одним из основных достоинств языка С# является его схема работы с памятью: автоматическое выделение памяти под объекты и автоматическая уборка мусора. При этом невозможно обратиться по несуществующему адресу памяти или вый­ ти за границы массива, что делает программы более надежными и безопасными и исключает возможность появления целого класса ошибок, доставляющих мас­ су неудобств при написании программ на других языках. Однако в некоторых случаях возникает необходимость работать с адресами па­ мяти непосредственно, например, при взаимодействии с операционной системой, написании драйверов или программ, время выполнения которых критично. Такую возможность предоставляет так называемый небезопасный (unsafe) код. 348 Глава 15. Дополнительные средства С# Небезопасным называется код, выполнение которого среда CLR не контролиру­ ет. Он работает напрямую с адресами областей памяти посредством указателей и должен быть явным образом помечен с помощью ключевого слова unsafe, кото­ рое определяет так называемый небезопасный контекст выполнения. Ключевое слово unsafe может использоваться либо как спецификатор, либо как оператор. В первом случае его указывают наряду с другими спецификаторами при описании класса, делегата, структуры, метода, поля и т. д. — везде, где допус­ тимы другие спецификаторы. Это определяет небезопасный контекст для опи­ сываемого элемента, например: public unsafe struct Node { public int Value; . public Node\* Left; public Node\* Right; } Вся структура Node помечается как небезопасная, что делает возможным исполь­ зование в ней указателей Left и Right. Можно применить и другой вариант опи­ сания, в котором небезопасными объявляются только соответствующие поля структуры: public struct Node { public int Value; public, unsafe Node\* Left; public unsafe Node\* Right; } Оператор unsafe имеет следующий синтаксис: unsafe блок Все операторы, входящие в блок, выполняются в небезопасном контексте. ПРИМЕЧАНИ Е Компиляция кода, содержащего небезопасные фрагменты, должна производить! с ключом /unsafe. Этот режим можно установить путем настройки среды Visu Studio (Project • Properties • Configuration Properties • Build • Allow Unsafe Cod\* Синтаксис указателей Указатели предназначены для хранения адресов областей памяти. Синтакс объявления указателя: тип\* переменная; Здесь тип — это тип величины, на которую указывает пережиная, то есть ве. чины, хранящейся по записанному в переменной адресу. Тип не может 6i классом, но может быть структурой, перечислением, указателем, а также од\* Небезопасный код 349 из стандартных типов: sbyte, byte, short, ushort, int, uint, long, uiong, char, float, double, decimal, bool и void. Последнее означает, что указатель ссылается на пере­ менную неизвестного типа. Указатель на тип voi d применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов). Указателю на тип voi d можно присвоить значение указателя любого типа, а так­ же сравнивать его с любыми указателями, но перед выполнением каких-либо действий с областью памяти, на которую он ссылается, требуется преобразовать его к конкретному типу явным образом. Примеры объявления указателей: int\* а; // указатель на int Node\* pNode; // указатель на описанную ранее структуру Node void\* р; // указатель на неопределенный тип int\*[] m; // одномерный массив указателей на int int\*\* d; // указатель на указатель на int В одном операторе можно описать несколько указателей одного и того же типа, например: int\* а. Ь, с: // три указателя на int Указатели являются отдельной категорией типов данных. Они не наследуются от типа object, и преобразование между типом object и типами указателей невоз­ можно. В частности, для них не выполняется упаковка и распаковка. Однако до­ пускаются преобразования между разными типами указателей, а также указате­ лями и целыми. ПРИМЕЧАНИ Е Именно потому что указатели могут содержать адрес любой области памяти и, сле­ довательно, изменить ее, операции с ними и называются небезопасными. Величины типа указателя могут являться локальными переменными, полями, параметрами и возвращаемым значением функции. Эти величины подчиняются общим правилам определения области действия и времени жизни. Преобразования указателей Для указателей поддерживаются неявные преобразования из любого типа указа­ теля к типу void\*. Любому указателю можно присвоить константу null. Кроме того, допускаются явные преобразования: • между указателями любого типа; • между указателями любого типа и целыми типами (со знаком и без знака). 350 Глава 15. Дополнительные средства С# Корректность преобразований лежит на совести программиста. Преобразования никак не влияют на величины, на которые ссылаются указатели, но при попытке получения значения по указателю несоответствующего типа поведение програм­ мы не определено 1 . Инициализация указателей Ниже перечислены способы присваивания значений указателям: 1. Присваивание указателю адреса существующего объекта: О с помощью операции получения адреса: int а = 5: // целая переменная int\* р = &а; // в указатель записывается адрес а ПРИМЕЧАНИ Е — Программист должен сам следить за тем, чтобы переменные, на которые ссылается указатель, были правильно инициализированы. О с помощью значения другого инициализированного указателя: int\* Г = р; О с помощью имени массива, которое трактуется как адрес: int[] b = new int[] {10. 20. 30, 50}; // массив fixed ( int\* t = b ) { ... }; // присваивание адреса начала массива fixed ( int\* t = &b[0] ){...} ; //то же самое ПРИМЕЧАНИ Е — Оператор f i xed рассматривается позже. 2. Присваивание указателю адреса области памяти в явном виде: char\* V = (char \*)0xl2F69E; Здесь 0xl2F69E — шестнадцатеричная константа, (char \*) — операция приведе­ ния типа: константа преобразуется к типу указателя на char. ПРИМЕЧАНИ Е Использовать этот способ можно только в том случае, если адрес вам точно извес­ тен, в противном случае может возникнуть исключение. 3. Присваивание нулевого значения: i nt\* хх = nul 1 : 1 Как правило, если в документации встречается оборот «неопределенное поведение» (undefined behavior), ничего хорошего это не сулит. Небезопасный код 351 4. Выделение области памяти в стеке и присваивание ее адреса указателю: int\* s = stackalloc int [10]; Здесь операция stackalloc выполняет выделение памяти под 10 величин типа int (массив из 10 элементов) и записывает адрес начала этой области памяти в переменную s, которая может трактоваться как имя массива. ПРИМЕЧАНИ Е Специальных операций для выделения области динамической памяти (хипа) в С# не предусмотрено. В случае необходимости можно использовать, например, сис­ темную функцию НеарАПос (пример см. в спецификации языка). Операции с указателями Все операции с указателями выполняются в небезопасном контексте. Они пере­ числены в табл. 15.1. Таблица 15.1 . Операции с указателями Операция Описание \* Разадресация — получение значения, которое находится по адресу, хранящемуся в указателе -> Доступ к элементу структуры через указатель [] Доступ к элементу массива через указатель & Получение адреса переменной ++, -- Увеличение и уменьшение значения указателя на один адресуемый элемент +, - Сложение с целой величиной и вычитание указателей ==, !=,<>,<=,> = Сравнение адресов, хранящихся в указателях. Выполняется как сравнение беззнаковых целых величин stackalloc Выделение памяти в стеке под переменную, на которую ссылается указатель Рассмотрим примеры применения операций. Если в указатель занесен адрес объ­ екта, получить доступ к этому объекту можно с помощью операций разадреса- ции и доступа к элементу. Операция разадресации, или разыменования, предназначена для доступа к вели­ чине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины, например: int а = 5; // целая переменная int\* р = &а; // инициализация указателя адресом а Console.WriteLineC \*р ): // операция разадресации, результат: 5 Console.WriteLineC ++(\*р) ) ; // результат: б int[ ] b = new int[] {10, 20, 30, 50}; // массив 352 Глава 15. Дополнительные средства С# fixed ( int\* t = b ) // инициализация указателя адресом начала массива int\* z = t; // инициализ for (int i = 0; i < b.Length; ++i ) // инициализация указателя значением другого указателя t[i] += 5; \*z += 5; ++z; // доступ к элементу массива (увеличение на 5) // доступ с помощью разадресации (увеличение еще на 5) // инкремент указателя Console.WriteLineC &t[5] - t ); // операция вычитания указателей Оператор fixed фиксирует объект, адрес которого заносится в указатель, для того чтобы его не перемещал сборщик мусора и, таким образом, указатель остался корректным. Фиксация происходит на время выполнения блока, который записан после круглых скобок. В приведенном примере доступ к элементам массива выполняется двумя спосо­ бами: путем индексации указателя t и путем разадресации указателя z, значение которого инкрементируется при каждом проходе цикла для перехода к следую­ щему элементу массива. Конструкцию \*переменная можно использовать в левой части оператора при­ сваивания, так как она определяет адрес области памяти. Для простоты эту кон­ струкцию можно считать именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа. Арифметические операции с указателями (сложение с целым, вычитание, инкре­ мент и декремент) автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и име­ ют смысл в основном при работе со структурами данных, элементы которых раз- м$щены в памяти последовательно, например, с массивами. Инкремент перемещает указатель к следующему элементу массива, декремент — к предыдущему. Фактически значение указателя изменяется на величину sizeof (тип), где sizeof — операция получения размера величины указанного типа (в бай­ тах). Эта операция применяется только в небезопасном контексте, с ее помощью можно получать размеры не только стандартных, но и пользовательских типов данных. Для структуры результат может быть больше суммы длин составляю­ щих ее полей из-за выравнивания элементов. Если указатель на определенный тип увеличивается или уменьшается на кон­ станту, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например: short\* р; ... р++; // значение р увеличивается на 2 long\* q; q++; • // значение q увеличивается на 4 Небезопасный код 353 Разность двух указателей — это разность их значений, деленная на размер типа в байтах. Так, результат выполнения последней операции вывода в приведенном примере равен 5. Суммирование двух указателей не допускается. При записи выражений с указателями следует обращать внимание на приорите­ ты операций. В качестве примера рассмотрим последовательность действий, за­ данную в операторе \*р++ =10; Поскольку инкремент постфиксный, он выполняется после выполнения опера­ ции присваивания. Таким образом, сначала по адресу, записанному в указателе р, будет записано значение 10, а затем указатель увеличится на количество бай­ тов, соответствующее его типу. То же самое можно записать подробнее: \*р = 10; Р++; Выражение (\*р)++, напротив, инкрементирует значение, на которое ссылается указатель. В следующем примере каждый байт беззнакового целого числа х выводится на консоль с помощью указателя t: uint х = 0xAB10234F; byte\* t = (byte\*)&x; for ( int i = 0; i < 4: ++i ) Console.Write("{0:X} ", \*t++ ) ; // результат: 4F 23 10 AB Как видите, первоначально указатель t был установлен на младший байт перемен­ ной х. Листинг 15.1 иллюстрирует доступ к полю класса и элементу структуры: Листин г 15.1 . Доступ к полю класса и элементу структуры с помощью указателей using System; namespace ConsoleApplicationl { class A { public int value -. 20; } struct В { public int a; } class Program { unsafe static void MainO { A n = new AO; fixed ( int\* pn = &n.value ) ++(\*pn); Console.WriteLineC "n = " + n.value ); // результат: 21 В b; В\* pb = &b; продолжение x 354 Глава 15. Дополнительные средства С# Листин г 15. 1 {продолжение) pb->a - 100; Console.WriteLineC b.a ); // результат: 100 } ' } } Операция stackalloc позволяет выделить память в стеке под заданное количест­ во величин заданного типа: stackalloc тип [ количество ] Количество задается целочисленным выражением. Если памяти недостаточно, генерируется исключение System.StackOverflowException. Выделенная память ни­ чем не инициализируется и автоматически освобождается при завершении бло­ ка, содержащего эту операцию. Пример выделения памяти под пять элементов типа int и их заполнения числами от 0 до 4: int\* р = stackalloc int [5]; for С int i = 0; i < 5: ++i ) { pC1]-1 : Console.WriteC p[i] + " " ): // результат: 0 1 2 3 4 . } В листинге 15.2 приведен пример работы с указателями, взятый из специфика­ ции С#. Метод IntToString преобразует переданное ему целое значение в строку символов, заполняя ее посимвольно путем доступа через указатель. Листин г 15.2 . Пример работы с указателями: перевод числа в строку using System: class Test { static string IntToString ( int value ) { int n - value >- 0 ? value : -value; unsafe { char\* buffer = stackalloc char[16]; char\* p = buffer + 16; do { \*--p = (charX n % 10 + '0' ); n /- 10; } while ( n !- 0 ); if ( value < 0 ) \*--p - '-': return new stringC p. 0, (int)C buffer + 16 - p ) ); } } static void MainO { Console.WriteLineC IntToStringC 12345 ) ); Console.WriteLineC IntToStringC -999 ) ); } } -улярные выражения 355 егулярные выражения гулярные выражения предназначены дл я обработки текстовой информаци и обеспечивают: эффективны й поиск в тексте по заданному шаблону; редактирование, замену и удаление подстрок; формировани е итоговых отчетов по результатам работы с текстом. помощь ю регулярных выражени й удобно обрабатывать, например, файл ы в фор - ате HTML , файл ы журнало в ил и длинны е текстовые файлы . Дл я поддержк и егулярных выражени й в библиотеку .NE T включен ы классы, объединенны е пространство име н System.Text.RegularExpressions. Метасимволы 'егулярное выражени е — это шабло н (образец), по которому выполняется поиск оответствующего ему фрагмента текста. Язы к описания регулярных выражени й юстоит из символо в двух видов: обычны х и метасимволов. Обычны й симво л федставляет в выражени и сам себя, а метасимвол — некоторый класс символов, щприме р любу ю цифр у ил и букву. Например, регулярное выражени е для поиска в тексте фрагмента «Вася» запл­ аваетс я с помощь ю четырех обычны х символов Вася, а выражени е дл я поиска двух цифр , идущи х подряд, состоит из двух метасимволов \d\d. С помощь ю комбинаци й метасимволов можн о описывать сложны е шаблон ы дл я поиска. Например , можн о описать шабло н дл я IP-адреса, адреса электронной почты, различных форматов даты, заголовков определенного вида и т. д. ПРИМЕЧАНИ Е Синтаксис регулярных выражений .NET в основном позаимствован из языка Perl 5. Неподготовленного человека вид сложного регулярного выражения может привес­ ти в замешательство, но при вдумчивом изучении он обязательно почувствует его красоту и очарование. Пожалуй, регулярные выражения более всего напоминают заклинания, по которым волшебным образом преобразуется текст. Ошибка все­ го в одном символе делает заклинание бессильным, зато, верно составленное, оно творит чудеса! В табл. 15.2 описан ы наиболее употребительные метасимволы, представляющие собой классы символов. Метасимволы , перечисленные в табл. 15.3, уточняют позици ю в строке, в кото­ р ой следует искать совпадение с регулярным выражением, например, только в на­ чале ил и в конце строки. Эт и метасимвол ы являются мнимыми , то есть в тексте им не соответствует никакой реальный символ. 356 Глава 15. Дополнительные средства С# Таблица 15.2 . Классы символов Класс символов Описание Пример Любой символ, кроме \ri Выражение с. t соответствует фрагментам cat, cut, clt, c{t и т. д. [ ] Любой одиночный символ из Выражение c[aul]t соответствует последовательности, записанной фрагментам cat, cut и clt, внутри скобок. Допускается а выражение c[a-z]t — использование диапазонов фрагментам cat, cbt, cct, символов cdt,cz t п Любой одиночный символ, не Выражение c[\*aul]t входящий в последовательность, соответствует фрагментам cbt, записанную внутри скобок. c2t, cXt и т. д., а выражение Допускается использование c[\*a-zA-Z]t — фрагментам cnt, диапазонов символов clt, c4t, c3t и т. д. \w Любой алфавитно-цифровой Выражение c\wt соответствует символ, то есть символ из фрагментам cat, cut, clt, ClOt множества прописных и строчных и т. д., но не соответствует букв и десятичных цифр фрагментам c{t, c;t и т. д. \W Любой не алфавитно-цифровой Выражение c\Wt соответствует символ, то есть символ, фрагментам c{t, c;t, С t и т. д., не входящий в множество но не соответствует фрагментам прописных и строчных букв cat, cut, clt, clOt и т. д. и десятичных цифр \s Любой пробельный символ, Выражение \s\w\w\w\s например символ пробела, соответствует любому слову табуляции (\t, \v), перевода из трех букв, окруженному строки (\п, \г), новой пробельными символами страницы (\f ) \S Любой не пробельный символ, Выражение \s\S\S\s то есть символ, не входящий соответствует любым двум в множество пробельных непробельным символам, окруженным пробельными \d Любая десятичная цифра Выражение c\dt соответствует фрагментам clt, c2t,c9 t \D Любой символ, не являющийся Выражение C\Dt не соответствует десятичной цифрой фрагментам clt, c2t c9t Таблица 15.3 . Уточняющие метасимволы Метасимвол Описание А. Фрагмент, совпадающий с регулярным выражением, следует искать только в начале строки $ Фрагмент, совпадающий с регулярным выражением, следует искать только в конце строки \А Фрагмент, совпадающий с регулярным выражением, следует искать только в начале многострочной строки Регулярные выражения 357 Метасимвол . Описание \Z Фрагмент, совпадающий с регулярным выражением, следует искать только в конце многострочной строки \b Фрагмент, совпадающий с регулярным выражением, начинается или заканчивается на границе слова (то есть между символами, соответствующими метасимволам \w и \W) \ в Фрагмент, совпадающий с регулярным выражением, не должен встречаться на границе слова Например , выражени е ~cat соответствует символа м cat, встречающимс я в начале строки, выражени е cat$ — символа м cat, встречающимс я в конце строки (то есть за ним и идет симво л перевода строки), а выражени е \*$ представляет пусту ю строку, то есть начало строки, за которы м сразу же следует ее конец. В регулярных выражения х часто используют повторители. Повторители — это метасимволы, которые располагаются непосредственно после обычного символа и л и класса символов и задают количество его повторений в выражении . Напри ­ мер, если требуется записать выражени е дл я поиска в тексте пяти идущи х под­ р яд цифр , вместо метасимволов \d\d\d\d\d можн о записать \d{5}. Таком у выра­ жени ю будут соответствовать фрагмент ы 11111, 12345, 53332 и т. д. Наиболе е употребительные повторители перечислены в табл. 15.4. Таблиц а 15.4 . Повторители Метасимвол Описание Пример \* Ноль или более повторений Выражение ca\*t соответствует фрагментам предыдущего элемента ct, cat, caat, caaaaaaaaaaat и т. д. + Одно или более повторений Выражение ca+t соответствует фрагментам предыдущего элемента cat, caat, caaaaaaaaaaat и т. д. 7 Ни одного или одно Выражение ca?t соответствует фрагментам повторение предыдущего ct и cat элемента {п} Ровно п повторений Выражение ca{3}t соответствует фрагменту предыдущего элемента caaat, а выражение (cat){2} — фрагменту catcat1 {". } По крайней мере п Выражение са{3,}t соответствует повторений предыдущего фрагментам caaat, caaaat, элемента caaaaaaaaaaaat и т. д. {п.т} От п до m повторений Выражение ca{2,4}t соответствует предыдущего элемента фрагментам caat, caaat и caaaat Помим о рассмотренных элементов регулярных выражени й можн о использовать конструкцию выбора из нескольких элементов. Вариант ы выбор а перечисляются 1 Круглые скобки служат для группировки символов. 358 Глава 15. Дополнительные средства С# через вертикальную черту. Например , если требуется определить, присутствует ли в тексте хотя бы оди н элемент из списка «cat>, «dog» и «horse», можн о ис­ пользовать выражени е cat|dog|horse П р и поиске используется так называемы й «ленивый » алгоритм, по которому по­ и ск прекращается пр и нахождени и самого короткого из возможны х фрагментов, совпадающи х с регулярны м выражением. Пример ы простых регулярных выражений : • целое число (возможно, со знаком): t-+]?\d+ • вещественное число (может иметь знак и дробну ю часть, отделенную точкой): C-+]?\d+\.?\d\* • российский номер автомобиля (упрощенно): [A-Z]\d{3}[A-Z]{2}\d\dRUS ВНИМАНИ Е Если требуется описать в выражении обычный символ, совпадающий с каким-либо метасимволом, его предваряют обратной косой чертой. Так, для поиска в тексте символа точки следует записать \., а для поиска косой черты — \\. Например , дл я поиска в тексте имен и файл а cat.doc следует использовать ре­ гулярное выражени е cat\.doc. Симво л «точка» экранируется обратной косой чертой, ДДя того, чтоб ы он воспринималс я не как метасимвол «любо й символ» (в то м числе и точка!), а непосредственно 1 . Д л я группирования элементов выражени я используются круглые скобки. Группи­ рование применяется во многих случаях, например, если требуется задать повто­ ритель не дл я отдельного символа, а дл я последовательности (это использовано в предыдуще й таблице). Кром е того, группироваиие служи т дл я запоминани я в некоторой переменно й фрагмента текста, совпавшего с выражением , заклю­ ченны м в скобки. Им я переменной задается в угловых скобках ил и апострофах: (?фрагмент\_выражения) Фрагмен т текста, совпавши й пр и поиске с фрагментом регулярного выражения, заносится в переменну ю с заданны м именем. Пусть, например, требуется выде­ лить из текста номера телефонов, записанных в виде nnn-rm-nn. Регулярное выра­ жени е дл я поиска номера можн о записать так: (?\d\d\d-\d\d-\d\d) Обратите внимание на то, что метасимволы регулярных выражений не совпадают с метг символами, которые используются в шаблонах имен файлов, таких как \*.doc. Регулярные выражения 359 П р и анализе текста в переменну ю с имене м num будут последовательно записы­ ваться найденные номера телефонов. Рассмотрим ещ е оди н вариант применени я группирования — дл я формировани я обратных ссылок. Все конструкции, заключенны е в круглые скобки, автоматиче­ с ки нумеруются, начиная с 1. Эт и номера, предваренные обратной косой чертой, можн о использовать дл я ссылок на соответствующую конструкцию. Например , выражени е (\w)M используется для поиска сдвоенных символов в словах (wa//, mass, cooperate)1 . Круглы е скобки могут быть вложенными , пр и этом номе р конструкции опреде­ ляется порядком открывающе й скобки в выражении . Примеры : (Вася)\s+(должен)\s+(?\d+)\spy6\.\s+Hy что же ты. \1 В этом выражени и три подвыражения, заключенны х в скобки. Ссылк а на первое из них выполняется в конце выражения. С этим выражение м совпадут, напри­ мер, фрагмент ы Вася должен 5 руб. Ну что же ты, Вася Вася должен 53459 руб. Ну что же ты. Вася Выражение , задающее IP-адрес: ((\d{1.3}\.){3}\d{1.3}) Адрес состоит из четырех групп цифр, разделенных точками. Кажда я группа може т включать от одной до трех цифр. Пример ы IP-адресов: 212.46.197.69, 212.194.5.106, 209.122.173.160. Перва я группа, заключенная в скобки, задает весь адрес. Ей при­ сваивается номер 1. В нее вложен ы вторые скобки, определяющи е границ ы для повторителя {3}. Переменную , им я которой задается внутри выражени я в угловых скобках, также можн о использовать дл я обратных ссылок в последующе й части выражения . На ­ пример, поиск двойны х символов в словах можн о выполнить с помощь ю выра­ жени я (?~~\w)\k, где s — им я переменной, в которой запоминается символ, \к — элемент синтаксиса. В регулярное выражени е можн о помещат ь комментарии. Поскольку выражени я обычн о прощ е писать, чем читать, это — очень полезная возможность. Коммента ­ р и й либ о помещается внутрь конструкции (?# ), либ о располагается, начиная от символа # до конца строки 2 . Классы библиотеки .NET для работы с регулярными выражениями Класс ы библиотеки .NE T дл я работы с регулярным и выражениям и объединены в пространство име н System.Text.RegularExpressions. 1 Если написать просто \w\w, будут найдены все пары алфавитно-цифровых символов. 2 Для распознавания этого вида комментария должен быть включен режим х RegexOptions. IgnorePatternWhitespace. 360 Глава 15. Дополнительные средства С# Начнем с класса Regex, представляющего собственно регулярное выражение. Класс является неизменяемым, то есть после создания экземпляра его корректи­ ровка не допускается. Для описания регулярного выражения в классе определе­ но несколько перегруженных конструкторов: • Regex () — создает пустое выражение; • Regex(String) — создает заданное выражение; • RegexCString, RegexOptions) — создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (напри­ мер, различать или не различать прописные и строчные буквы). Пример конструктора, задающего выражение для поиска в тексте повторяющих­ ся слов, расположенных подряд и разделенных произвольным количеством про­ белов, независимо от регистра: Regex гх = new Regex( @"\b(?\w+)\s+(\k)\b". RegexOpti ons.IgnoreCase ); Поиск фрагментов строки, соответствующих заданному выражению, выполняет­ ся с помощью методов IsMatch, Match и Matches. Метод IsMatch возвращает true, если фрагмент, соответствующий выражению, в заданной строке найден, и false в противном случае. В листинге 15.3 приведен пример поиска повторяющихся слов в двух тестовых строках. В регулярное выражение, приведенное ранее, добавлен фрагмент, позволяющий распознавать знаки препинания. Листин г 15.3 . Поиск в строке дублированных слов (методом IsMatch) using System; usi ng System.Text.RegularExpressions; public class Test { public static void MainO { Regex r = new Regex( @"\b(?\w+)[.,:;!? ]\s\*(\k)\b", RegexOptions.IgnoreCase ); string tstl = "Oh. oh! Give me more!"; if ( r.IsMatch( tstl ) ) Console.WriteLine( " tstl yes" ); else Console.WriteLine( " tstl no" ); string tst2 = "Oh give me. give me more!"; if ( r.IsMatch( tst2 ) ) Console.WriteLineC " tst2 yes" ); else Console.WriteLine( " tst2 no" ); } } Результат работы программы: tstl yes tst2 no Регулярные выражения 361 Повторяющиеся слова в строке tst2 располагаются не подряд, поэтому она не со­ ответствует регулярному выражению. Для поиска повторяющихся слов, рас­ положенных в произвольных позициях строки, в регулярном выражении нужно всего-навсего заменить пробел (\s) «любым символом» (.) : Regex г = new Regex( <a"\b(?\w+)[..:;!? ].\*(\k)\b". RegexOptions.IgnoreCase ); Метод Match класса Regex, в отличие от метода IsMatch, не просто определяет, про­ изошло ли совпадение, а возвращает объект класса Match — очередной фраг­ мент, совпавший с образцом. Рассмотрим листинг 15.4, в котором используется этот метод. Листин г 15.4 . Выделение из строки слов и чисел (методом Match) using System; using System. Text.RegularExpressions; public class Test { public static void MainO { string text = "Салат - $4, борщ - $3, одеколон - $10."; string pattern = @"(\w+) - \$(\d+)[..]" ; Regex r = new Regex( pattern ); Match m = r.Match( text ); int total = 0; while ( m.Success ) { Console.WriteLine( m ); total += int.Parse( m.Groups[2].ToStringO ); m = m.NextMatchO; } Console.WriteLine( "Итого: $" + total ); } } Результат работы программы: Салат - $4, борщ - $3, одеколон - $10. Итого: $17 При первом обращении к методу Match возвращается первый фрагмент строки, совпавший с регулярным выражением pattern. В классе Match определено свой­ ство Groups, возвращающее коллекцию фрагментов, совпавших с подвыраже­ ниями в круглых скобках. Нулевой элемент коллекции содержит весь фрагмент, первый элемент — фрагмент, совпавший с подвыражением в первых скобках, второй элемент — фрагмент, совпавший с подвыражением во вторых скобках, и т. д. Если при определении выражения задать фрагментам имена, как это было 362 Глава 15. Дополнительные средства С# сделано в предыдущем листинге, можно будет обратиться к ним по этим именам, например; string pattern - @"(?'name'\w+) - \$(?'price'\d+)С.,]"; total += int.ParseC m.Groups["price"].ToString() ); ПРИМЕЧАНИ Е Метод NextMatch класса Match продолжает поиск в строке с того места, на котором закончился предыдущий поиск. Метод Matches класса Regex возвращает объект класса MatchCollection — коллек­ цию всех фрагментов заданной строки, совпавших с образцом. Рассмотрим теперь пример применения метода Split класса Regex. Этот метод разбивает заданную строку на фрагменты в соответствии с разделителями, задан­ ными с помощью регулярного выражения, и возвращает эти фрагменты в массиве строк. В листинге 15.5 строка из листинга 15.4 разбивается на отдельные слова. Листин г 15.5. Разбиение строки на слова (методом Split) using System; usi ng System.Col 1ecti ons.Generi с; using System.Text.RegularExpressions; public class Test { public static void MainO { string text = "Салат - $4, борщ -$3. одеколон - $10."; string pattern = "[ - ..]+" ; Regex r ' = new Regex( pattern ); List words = new List( r.SplitC text ) ); foreach ( string word in words ) Console.WriteLineC word ); i } ' ' ' } i Результат работы программы: Салат $4 борщ $3 одеколон $10 Метод Repl асе класса Regex позволяет выполнять замену фрагментов текста. Опре­ делено несколько перегруженных версий этого метода. Вот как выглядит пример простейшего применения метода в его статическом варианте, заменяющего все вхождения символа $ символами у. е.: string text = "Салат - $4, борщ -$3, одеколон - $10."; string textl = Regex.Replасе( text, @"\$", "y.e." ); Регулярные выражения 363 Другие версии метода позволяют задавать любые действия по замене с помощью делегата MatchEvaluator, который вызывается для каждого вхождения фрагмента, совпавшего с заданным регулярным выражением. ПРИМЕЧАНИ Е Помимо классов Regex и Match в пространстве имен System.Text.RegularExpressions определены вспомогательные классы, например, класс Capture — фрагмент, совпав­ ший с подвыражением в круглых скобках; класс CaptureCol 1 ecti on — коллекция фраг­ ментов, совпавших со всеми подвыражениями в текущей группе; класс Group содер­ жит коллекцию Capture для текущего совпадения с регулярным выражением и т. д. 8 качестве более реального примера применения регулярных выражений рас­ смотрим программу анализа файла журнала веб-сервера. Это текстовый файл, каждая строка которого содержит информацию об одном соединении с сервером. Четыре строки файла приведены ниже: ppp-48.pool-113.spbnit.ru - - [31/Мау/2002:02:08:32 +0400] "GET / НТТР/1.1" .200 2434 "http://www.price.ru/bin/price/firmlnfo\_f?fid=10922&where=01&base=2" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)" 81.24.130.7 - - [31/May/2002:08:13:17 +0400] "GET /swf/menu.swf HTTP/1.Г~200 4682 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)" 81.24.130.7 - - [31/May/2002:08:13:17 +0400] "GET /swf/header.swf HTTP/1.1" 200 21244 "-" "Mozilla/4.0 (compatible; MSIE 5.01; Windows 98)" gate.solvo.ru - - [31/May/2002:10:43:03 +0400] "GET / HTTP/1.0" 200 2422 "http://www.price.ru/bin/price/firminfo\_f?fid=10922&where»0I&base=l" "Mozilla/4.0 (compatible; MSIE 5.0: Windows NT; DigExt)" Подобные файлы могут иметь весьма значительный объем, поэтому составление итогового отчета в удобном для восприятия формате имеет важное значение. Если рассматривать каждую строку файла как совокупность полей, разделенных пробелами, то поле номер 0 содержит адрес, с которого выполнялось соединение с сервером, поле номер 5 — операцию (GET при загрузке информации), поле 8 — признак успешности выполнения операции (200 — успешно) и, наконец, поле 9 — количество переданных байтов. Приведенная в листинге 15.6 программа формирует в формате HTM L итоговый отчет, содержащий таблицу адресов, с которых выполнялось обращение к серве­ ру, и суммарное количество переданных байтов для каждого адреса. Листин г 15.6 . Анализ журнала веб-сервера using System; using System.10; usi ng System.Col 1ecti ons.Generi с; using System.Text.RegularExpressions: public class Test { public static void MainO ^ продолжение & 364 Глава 15. Дополнительные средства С# Листин г 15. 6 (продолжение) StreamReader f = new StreamReaderC "accessjog" ); StreamWriter w = new StreamWriterC "report.htm" ): Фрагмент результата работы программы показан на рис. 15.1. Regex Regex get г new RegexC "GET" ); new RegexC " " ); j?>block209-209-002. octetgroup.net j 162781 j J81.24.130.7 174756 jgate. sohro. ru {113692} [212.113.108.164 " ~ ~ 12611991 Рис. 15.1 . Фрагмент журнала веб-сервера Файл accessjog считывается построчно, каждая строка разбивается на поля, которые заносятся в массив items. Затем, если загрузка прошла успешно, о чем свидетельствуют значения GET и 200 полей 5 и 8, количество переданных байтов Документирование в формате XML 365 (поле 9) преобразуется в целое и заносится в хеш-таблицу по ключу, которым служит адрес, хранящийся в поле 0. Для формирования HTML-файла report.htm используются соответствующие теги. Файл можно просмотреть, например, с помощью Internet Explorer. Как ви­ дите, программа получилась весьма компактной за счет использования стандарт­ ных классов библиотеки .NET1 . Документирование в формате XML XM L (extensible Markup Language) — это язык разметки текста. Разметкой яв­ ляется все, что не относится к содержанию, или, как модно говорить, контенту: структура документа, формат, вид и т. д. Разметка осуществляется с помощью тегов — управляющих элементов, заключенных в угловые скобки. Теги в XM L всегда парные: открывающий тег записывается перед размечаемым фрагментом, а закрывающий — после него. Закрывающий тег выглядит так же, как открываю­ щий, но предваряется косой чертой, например: Класс для работы с регулярными выражениями В тегах могут присутствовать атрибуты — элементы вида имя = значение, уточ­ няющие и дополняющие описание элемента. Язык XM L широко распространен в Интернете благодаря его универсальности и переносимости. Корпоративные приложения используют XM L как основной формат для обмена данными. Строго говоря, XM L является не языком, а систе­ мой правил для описания языков. ПРИМЕЧАНИ Е Многие составляющие технологии .NET неразрывно связаны с XML, поэтому в про­ странстве имен System. Xml библиотеки .NET описано множество классов, под­ держивающих XML. Объем и задача учебника не позволяют описать эти классы и технологии. Любой программный продукт требуется документировать. Соответствие версий документации и программы — серьезная проблема, которая решается в .NET встраиванием документации прямо в код программы с помощью комментари­ ев и XML-тегов. Комментарии, используемые для построения файлов документации, начина­ ются с символов ///и размещаются перед соответствующим элементом про­ граммы. Внутри комментариев записываются теги, относящиеся к комменти­ руемому элементу — классу, методу,- параметру метода и т. п. Теги перечислены в табл. 15.5. Справедливости ради надо отметить, что аналогичная программа на языке Perl пример­ но в два раза короче. 366 Глава 15. Дополнительные средства С# Таблиц а 15.5 . Теги документирования Тег Описание Форматирование как фрагмента кода Многострочный код (используется в секции ) Пример использования класса или метода Исключение, генерируемое классом < Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся на гласные буквы. Вариант 11 Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел. Вариант 12 Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы. Вариант 13 Написать программу, которая считывает английский текст из файла и выво­ дит его на экран, заменив прописной каждую первую букву слов, начинающихся с гласной буквы. Вариант 14 Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 словами «ноль», «один», «девять», начиная каждое предложение с новой строки. Вариант 15 Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте. Вариант 16 Написать программу, которая считывает текст из файла и выводит на экран сна­ чала вопросительные, а затем восклицательные предложения. Вариант 17 Написать программу, которая считывает текст из файла и выводит его на экран, после каждого предложения добавляя, сколько раз встретилось в нем введенное с клавиатуры слово. Лабораторная работа 8. Классы и операции 395 Вариант 18 Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке. Вариант 19 Написать программу, которая считывает текст из файла и выводит на экран сна­ чала предложения, начинающиеся с однобуквенных слов, а затем все остальные. Вариант 20 Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации. Лабораторная работа 8. Классы и операции Теоретический материал: глава 7. Каждый разрабатываемый класс должен, как правило, содержать следующие эле­ менты: скрытые поля, конструкторы с параметрами и без параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интер­ фейс класса. При возникновении ошибок должны выбрасываться исключения. В программе должна выполняться проверка всех разработанных элементов класса. Вариант 1 Описать класс для работы с одномерным массивом целых чисел (вектором). Обеспечить следующие возможности: • задание произвольных целых границ индексов при создании объекта; • обращение к отдельному элементу массива с контролем выхода за пределы массива; • выполнение операций поэлементного сложения и вычитания массивов с оди­ наковыми границами индексов; • выполнение операций умножения и деления всех элементов массива на скаляр; • вывод на экран элемента массива по заданному индексу и всего массива. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 2 Описать класс для работы с одномерным массивом строк фиксированной длины. Обеспечить следующие возможности: 396 Лабораторные работы • задание произвольных целых границ индексов при создании объекта; • обращение к отдельной строке массива по индексу с контролем выхода за пределы массива; • выполнение операций поэлементного сцепления двух массивов с образовани­ ем нового массива; • выполнение операций слияния двух массивов с исключением повторяющих­ ся элементов; • вывод на экран элемента массива по заданному индексу и всего массива. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 3 Описать класс многочленов от одной переменной, задаваемых степенью много­ члена и массивом коэффициентов. Обеспечить следующие возможности: • вычисление значения многочлена для заданного аргумента; • операции сложения, вычитания и умножения многочленов с получением но­ вого объекта-многочлена; • получение коэффициента, заданного по индексу; • вывод на экран описания многочлена. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 4 Описать класс, обеспечивающий представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматри­ цы любого размера и всей матрицы, доступа по индексам к элементу матрицы. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 5 Описать класс для работы с восьмеричным числом, хранящимся в виде строки символов. Реализовать конструкторы, свойства, методы и следующие операции: • операции присваивания, реализующие значимую семантику; • операции сравнения; • преобразование в десятичное число; • форматный вывод; • доступ к заданной цифре числа по индексу. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 6 Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (по авто- бораторная работа 8. Классы и операции 397 ', по году издания или категории), добавления книг в библиотеку, удаления гиг из нее, доступа к книге по номеру. аписать программу, демонстрирующую все разработанные элементы класса. ариант 7 писать класс «записная книжка». Предусмотреть возможность работы с произ- эльным числом записей, поиска записи по какому-либо признаку (например, по амилии, дате рождения или номеру телефона), добавления и удаления записей, зртировки по фамилии и доступа к записи по номеру. [аписать программу, демонстрирующую все разработанные элементы класса. (ариант 8 )писать класс «студенческая группа». Предусмотреть возможность работы с пе- >еменным числом студентов, поиска студента по какому-либо признаку (напри- iep, по фамилии, имени, дате рождения), добавления и удаления записей, сорти- ювки по разным полям, доступа к записи по номеру. 1аписать программу, демонстрирующую все разработанные элементы класса. Зариант 9 Эписать класс, реализующий тип данных «вещественная матрица» и работу ; ними. Класс должен реализовывать следующие операции над матрицами: • сложение, вычитание (как с другой матрицей, так и с числом); 3 комбинированные операции присваивания (+=, -=) ; • операции сравнения на равенство/неравенство; • операции вычисления обратной и транспонированной матрицы; • доступ к элементу по индексам. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 10 Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами: • умножение, деление (как на другую матрицу, так и на число); • комбинированные операции присваивания (\*=, /=) ; • операцию возведения в степень; • методы вычисления детерминанта и нормы; • доступ к элементу по индексам. Написать программу, демонстрирующую все разработанные элементы класса. 398 Лабораторные работы Вариант 11 Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами: • методы, реализующие проверку типа матрицы (квадратная, диагональная, ну­ левая, единичная, симметричная, верхняя треугольная, нижняя треугольная); • операции сравнения на равенство/неравенство; • доступ к элементу по индексам. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 12 Описать класс «множество», позволяющий выполнять основные операции: до­ бавление и удаление элемента, пересечение, объединение и разность множеств. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 13 Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых,это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть воз­ можность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 14 Описать класс «автостоянка» для хранения сведений об автомобилях. Для каж­ дого автомобиля записываются госномер, цвет, фамилия владельца и признак присутствия на стоянке. Обеспечить возможность поиска автомобиля по разным критериям, вывода списка присутствующих и отсутствующих на стоянке авто­ мобилей, доступа к имеющимся сведениям по номеру места. Написать программу, демонстрирующую все разработанные элементы класса. Вариант 15 Описать класс «колода карт», включающий закрытый массив элементов класса «карта». В карте хранятся масть и номер. Обеспечить возможность вывода кар­ ты по номеру, вывода всех карт, перемешивания колоды и выдачи всех карт из колоды поодиночке и по 6 штук в случайном порядке. Написать программу, демонстрирующую все разработанные элементы классов. Вариант 16 Описать класс «поезд», содержащий следующие закрытые поля: • название пункта назначения; Лабораторная работа 8. Классы и операции 399 • номер поезда (может содержать буквы и цифры); • время отправления. Предусмотреть свойства для получения состояния объекта. Описать класс «вокзал», содержащий закрытый массив поездов. Обеспечить сле­ дующие возможности: • вывод информации о поезде по номеру с помощью индекса; • вывод информации о поездах, отправляющихся после введенного с клавиату­ ры времени; • перегруженную операцию сравнения, выполняющую сравнение времени от­ правления двух поездов; • вывод информации о поездах, отправляющихся в заданный пункт назначения. Информация должна быть отсортирована по времени отправления. Написать про­ грамму, демонстрирующую все разработанные элементы классов. Вариант 17 Описать класс «товар», содержащий следующие закрытые поля: • название товара; • название магазина, в котором продается товар; • стоимость товара в рублях. Предусмотреть свойства для получения состояния объекта. Описать класс «склад», содержащий закрытый массив товаров. Обеспечить сле­ дующие возможности: • вывод информации о товаре по номеру с помощью индекса; • вывод на экран информации о товаре, название которого введено с клавиату­ ры; если таких товаров нет, выдать соответствующее сообщение; • сортировку товаров по названию магазина, по наименованию и по цене; • перегруженную операцию сложения товаров, выполняющую сложение их цен. Написать программу, демонстрирующую все разработанные элементы классов. Вариант 18 Описать класс «самолет», содержащий следующие закрытые поля: • название пункта назначения; • шестизначный номер рейса; • время отправления. Предусмотреть свойства для получения состояния объекта. Описать класс «аэропорт», содержащий закрытый массив самолетов. Обеспечить следующие возможности: • вывод информации о самолете по номеру рейса с помощью индекса; 400 Лабораторные работы • вывод информации о самолетах, отправляющихся в течение часа после вве­ денного с клавиатуры времени; • вывод информации о самолетах, отправляющихся в заданный пункт назначе­ ния; • перегруженную операцию сравнения, выполняющую сравнение времени от­ правления двух самолетов. Информация должна быть отсортирована по времени отправления. Написать программу, демонстрирующую все разработанные элементы классов. Вариант 19 Описать класс «запись», содержащий следующие закрытые поля: • фамилия, имя; • номер телефона; • дата рождения (массив из трех чисел). Предусмотреть свойства для получения состояния объекта. Описать класс «записная книжка», содержащий закрытый массив записей. Обес­ печить следующие возможности: • вывод на экран информации о человеке, номер телефона которого введен с клавиатуры; если такого нет, выдать на дисплей соответствующее сообщение; • поиск людей, день рождения которых сегодня или в заданный день; • поиск людей, день рождения которых будет на следующей неделе; • поиск людей, номер телефона которых начинается на три заданных цифры. Написать программу, демонстрирующую все разработанные элементы классов. Вариант 20 Описать класс «англо-русский словарь», обеспечивающий возможность хране­ ния нескольких вариантов перевода для каждого слова. Реализовать доступ по строковому индексу — английскому слову. Обеспечить возможность вывода всех значений слов по заданному префиксу. Лабораторная работа 9. Наследование Теоретический материал: глава 8. В программах требуется описать базовый класс (возможно, абстрактный), в кото­ ром с помощью виртуальных или абстрактных методов и свойств задается интер­ фейс для производных классов. Целью лабораторной работы является максималь­ ное использование наследования, даже если для конкретной задачи оно не дает Лабораторная работа 9. Наследование 401 выигрыша в объеме программы. Во всех классах следует переопределить метод Equals, чтобы обеспечить сравнение значений, а не ссылок. Функция Mai п должна содержать массив из элементов базового класса, заполнен­ ный ссылками на производные классы. В этой функции должно демонстриро­ ваться использование всех разработанных элементов классов. Вариант 1 Создать класс Point (точка). На его основе создать классы ColoredPoint и Line (линия). На основе класса Line создать классы ColoredLine и PolyLine (много­ угольник). В классах описать следующие элементы: • конструкторы с параметрами и конструкторы по умолчанию; • свойства для установки и получения значений всех координат, а также для изменения цвета и получения текущего цвета; • для линий — методы изменения угла поворота линий относительно первой точки; • для многоугольника — метод масштабирования. Вариант 2 Создать абстрактный класс Vehicle (транспортное средство). На его основе реа­ лизовать классы Plane (самолет), Саг (автомобиль) и Ship (корабль). Классы должны иметь возможность задавать и получать координаты и параметры средств передвижения (цена, скорость, год выпуска и т. п.) с помощью свойств. Для самолета должна быть определена высота, для самолета и корабля — коли­ чество пассажиров, для корабля — порт приписки. Динамические характеристи­ ки задать с помощью методов. Вариант 3 Описать базовый класс Строка. Обязательные поля класса: • поле для хранения символов строки; • значение типа word для хранения длины строки в байтах. Реализовать обязательные методы следующего назначения: Q конструктор без параметров; • конструктор, принимающий в качестве параметра строковый литерал; • конструктор, принимающий в качестве параметра символ; • метод получения длины строки; • метод очистки строки (сделать строку пустой). Описать производный от Строка класс Комплексное\_число. Строки данного класса состоят из двух полей, разделенных символом i. 402 Лабораторные работы Первое поле задает значение действительной части числа, второе — значение мнимой. Каждое из полей может содержать только символы десятичных цифр и символы - и +, задающие знак числа. Символы - или + могут находиться толь­ ко в первой позиции числа, причем символ + может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки бу­ дут встречены любые символы, отличные от допустимых, класс Комплексноечисло принимает нулевое значение. Примеры строк: 33il2 -7И00 +5i - 21 Для класса Комплексное\_число определить следующие методы: • проверка на равенство; • сложение чисел; • умножение чисел. Вариант 4 Описать базовый класс Строка в соответствии с вариантом 3. Описать производный от Строка класс Десятичная\_строка. Строки данного класса могут содержать только символы десятичных цифр и сим­ волы - и +, задающие знак числа. Символы - или + могут находиться только в первой позиции числа, причем символ + может отсутствовать, в этом случае число считается положительным. Если в составе инициализирующей строки бу­ дут встречены любые символы, отличные от допустимых, класс Десятичнаястрока принимает нулевое значение. Содержимое данных строк рассматривается как десятичное число. Для класса определить следующие методы: • конструктор, принимающий в качестве параметра число; • арифметическая разность строк; • проверка на больше (по значению); • проверка на меньше (по значению). Вариант 5 Описать базовый класс Строка в соответствии с вариантом 3. Описать производный от Строка класс Битовая\_строка. Строки данного класса могут содержать только символы ' 0' или ' 1'. Если в со­ ставе инициализирующей строки будут встречены любые символы, отличные от допустимых, класс Битоваястрока принимает нулевое значение. Содержимое дан­ ных строк рассматривается как двоичное число. Отрицательные числа хранятся в дополнительном коде. Лабораторная работа 9. Наследование 403 Для класса Битовая\_строка определить следующие методы: • конструктор, принимающий в качестве параметра строковый литерал; • деструктор; • изменение знака на противоположный (перевод числа в дополнительный код); • присваивание; • вычисление арифметической суммы строк; • проверка на равенство. В случае необходимости более короткая битовая строка расширяется влево зна­ ковым разрядом. Вариант 6 1. Описать базовый класс Элемент. Закрытые поля: О имя элемента (строка символов); О количество входов элемента; О количество выходов элемента. Методы: О конструктор класса без параметров; О конструктор, задающий имя и устанавливающий равным 1 количество входов и выходов; О конструктор, задающий значения всех полей элемента. Свойства: О имя элемента (только чтение); О количество входов элемента; О количество выходов элемента. 2. На основе класса Элемент описать производный класс Комбинационный, представ­ ляющий собой комбинационный элемент (двоичный вентиль), который мо­ жет иметь несколько входов и один выход. Поле — массив значений входов. Методы: О конструкторы; О метод, задающий значение на входах экземпляра класса; О метод, позволяющий опрашивать состояние отдельного входа экземпляра класса; О метод, вычисляющий значение выхода (по варианту задания). 3. На основе класса Элемент описать производный класс Память, представляю­ щий собой триггер. Триггер имеет входы, соответствующие типу триггера 404 Лабораторные работы (см. далее вариант задания), и входы установки и сброса. Все триггеры счита­ ются синхронными, сам синхровход в состав триггера не включается. Поля: О массив значений входов объекта класса, в массиве учитываются все входы (управляющие и информационные); О состояние на прямом выходе триггера; О состояние на инверсном выходе триггера. Методы: О конструктор (по умолчанию сбрасывает экземпляр класса); О конструктор копирования; О метод, задающий значение на входах экземпляра класса; О методы, позволяющие опрашивать состояния отдельного входа экземпля­ ра класса; О метод, вычисляющий состояние экземпляра класса (по варианту задания) в зависимости от текущего состояния и значений на входах; О метод, переопределяющий операцию == для экземпляров класса. 4. Создать класс Регистр, используя класс Память как вложенный класс. Поля: О состояние входа «Сброс» — один для экземпляра класса; О состояние входа «Установка» — один для экземпляра класса; О массив типа Память заданной в варианте размерности; О массив (массивы), содержащий значения на соответствующих входах эле­ ментов массива типа Память. Методы: О метод, задающий значение на входах экземпляра класса; О метод, позволяющий опрашивать состояние отдельного выхода экземпля­ ра класса; О метод, вычисляющий значение нового состояния экземпляра класса. Все поля классов Элемент, Комбинационный и Память должны быть описаны с клю­ чевым словом pri vate. В задании перечислены только обязательные члены и методы класса. Можно задавать дополнительные члены и методы, если они не отменяют обязатель­ ные и обеспечивают дополнительные удобства при работе с данными классами, например, описать функции вычисления выхода/состояния как виртуальные. 5. Для проверки функционирования созданных классов написать программу, использующую эти классы. В программе должны быть продемонстрированы все свойства созданных классов. Конкретный тип комбинационного элемента, тип триггера и разрядность регист­ ра выбираются в соответствии с вариантом задания: Лабораторная работа 10. Структуры 405 Вариант Комбинационный элемент Число входов Триггер Разрядность регистра 1 И-НЕ 4 RS 8 2 ИЛИ 5 RST 10 3 МОД2-НЕ 6 D 12 4 И 8 Т 8 5 ИЛИ-НЕ 8 V 9 6 И 4 RS 10 7 ИЛИ-НЕ 5 JK 11 8 МОД2 5 D 8 9 И 4 Т 10 10 И ЛИ 3 JK 8 11 И-НЕ 3 RS 12 12 ИЛИ-НЕ 4 RST 4 13 МОД2 5 D 10 14 МОД2-НЕ 6 Т 10 15 ИЛИ-НЕ 8 V 10 16 И 8 JK 6 17 И-НЕ 8 RS 10 18 И ЛИ 8 Т ю 19 МОД2 6 JK 8 20 МОД2-НЕ 5 V 10 Лабораторная работа 10. Структуры Теоретический материал: глава 9. Вариант 1 Описать структуру с именем STUDENT, содержащую следующие поля: • фамилия и инициалы; • номер группы; • успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT (записи должны быть упорядочены по возрастанию номера группы); • вывод на экран фамилий и номеров групп для всех студентов, включенных, в массив, если средний балл студента больше 4,0 (если таких студентов нет, вывести соответствующее сообщение). 406 Лабораторные работы Вариант 2 Описать структуру с именем STUDENT, содержащую следующие поля: • фамилия и инициалы; • номер группы; • успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий йз десяти структур типа STUDENT (записи должны быть упорядочены по возрастанию среднего балла); • вывод на экран фамилий и номеров групп для всех студентов, имеющих оцен­ ки 4 и 5 (если таких студентов нет, вывести соответствующее сообщение). Вариант 3 Описать структуру с именем STUDENT, содержащую следующие поля: • фамилия и инициалы; • номер группы; • успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT (записи должны быть упорядочены по алфавиту); • вывод на экран фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2 (если таких студентов нет, вывести соответствующее сообщение). Вариант 4 Описать структуру с именем AER0FL0T, содержащую следующие поля: • название пункта назначения рейса; • номер рейса; • тип самолета. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из семи элементов типа AER0FL0T (записи должны быть упорядочены по возрастанию номера рейса); • вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт на­ значения, название которого совпало с названием, введенным с клавиатуры (если таких рейсов нет, вывести соответствующее сообщение). Вариант 5 Описать структуру с именем AER0FL0T, содержащую следующие поля: • название пункта назначения рейса; • номер рейса; • тип самолета. Лабораторная работа 10. Структуры 4 0 7 Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из семи элементов типа AER0FL0T (записи должны быть размещены в алфавитном порядке по названиям пунк­ тов назначения); • вывод на экран пунктов назначения и номеров рейсов, обслуживаемых само­ летом, тип которого введен с клавиатуры (если таких рейсов нет, вывести со­ ответствующее сообщение). Вариан т б Описать структуру с именем WORKER, содержащую следующие поля: • фамилия и инициалы работника; • название занимаемой должности; • год поступления на работу. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из десяти структур типа WORKER (записи должны быть упорядочены по алфавиту); • вывод на экран фамилий работников, стаж работы которых превышает значе­ ние, введенное с клавиатуры (если таких работников нет, вывести соответст­ вующее сообщение). Вариан т 7 Описать структуру с именем TRAIN, содержащую следующие поля: • название пункта назначения; • номер поезда; • время отправления. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN (записи должны быть размещены в алфавитном порядке по названиям пунк­ тов назначения); • вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени (если таких поездов нет, вывести соответствующее со­ общение). Вариан т 8 Описать структуру с именем TRAIN, содержащую следующие поля: • название пункта назначения; • номер поезда; • время отправления. 408 Лабораторные работы Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из шести элементов типа TRAIN (записи должны быть упорядочены по времени отправления поезда); • вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры (если таких поездов нет, вывести соответст­ вующее сообщение). Вариант 9 Описать структуру с именем TRAIN, содержащую следующие поля: • название пункта назначения; • номер поезда; • время отправления. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN (записи должны быть упорядочены по номерам поездов); • вывод на экран информации о поезде, номер которого введен с клавиатуры (если таких поездов нет, вывести соответствующее сообщение). Вариант 10 Описать структуру с именем MARSH, содержащую следующие поля: • название начального пункта маршрута; • название конечного пункта маршрута; • номер маршрута. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH (записи должны быть упорядочены по номерам маршрутов); • вывод на экран информации о маршруте, номер которого введен с клавиату­ ры (если таких маршрутов нет, вывести соответствующее сообщение). Вариант 11 Описать структуру с именем MARSH, содержащую следующие поля: • название начального пункта маршрута; • название конечного пункта маршрута; • номер маршрута. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH (записи должны быть упорядочены по номерам маршрутов); • вывод на экран информации о маршрутах, которые начинаются или оканчи­ ваются в пункте, название которого введено с клавиатуры (если таких мар­ шрутов нет, вывести соответствующее сообщение). 1абораторная работа 10. Структуры 409 Вариант 12 Описать структуру с именем NOTE, содержащую следующие поля: • фамилия, имя; • номер телефона; • дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть упорядочены по дате рождения); • вывод на экран информации о человеке, номер телефона которого введен с кла­ виатуры (если такого нет, вывести соответствующее сообщение). Вариант 13 Описать структуру с именем NOTE, содержащую следующие поля: • фамилия, имя; • номер телефона; • дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть размещены по алфавиту); • вывод на экран информации о людях, чьи дни рождения приходятся на ме­ сяц, значение которого введено с клавиатуры (если таких нет, вывести соот­ ветствующее сообщение). Вариант 14 Описать структуру с именем NOTE, содержащую следующие поля: • фамилия, имя; • номер телефона; • дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE (записи должны быть упорядочены по трем первым цифрам номера телефона); • вывод на экран информации о человеке, чья фамилия введена с клавиатуры (если такого нет, вывести соответствующее сообщение). Вариант 15 Описать структуру с именем ZNAK, содержащую следующие поля: • фамилия, имя; • знак Зодиака; • дата рождения (массив из трех чисел). 410 Лабораторные работы Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по дате рождения); • вывод на экран информации о человеке, чья фамилия введена с клавиатуры (если такого нет, вывести соответствующее сообщение). Вариант 16 Описать структуру с именем ZNAK, содержащую следующие поля: • фамилия, имя; • знак Зодиака; • дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по дате рождения); • вывод на экран информации о людях, родившихся под знаком, название кото­ рого введено с клавиатуры (если таких нет, вывести соответствующее сооб­ щение). Вариант 17 Описать структуру с именем ZNAK, содержащую следующие поля: • фамилия, имя; • знак Зодиака; • дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK (записи должны быть упорядочены по знакам Зодиака); • вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры (если таких нет, вывести соответствующее сообщение). Вариант 18 Описать структуру с именем PRICE, содержащую следующие поля: • название товара; • название магазина, в котором продается товар; • стоимость товара в рублях. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE (записи должны быть упорядочены в алфавитном порядке по названи­ ям товаров); Лабораторная работа 11. Интерфейсы и параметризованные коллекции 411 • вывод на экран информации о товаре, название которого введено с клавиату­ ры (если таких товаров нет, вывести соответствующее сообщение). Вариант 19 Описать структуру с именем PRICE, содержащую следующие поля: • название товара; • название магазина, в котором продается товар; • стоимость товара в рублях. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE (записи должны быть упорядочены в алфавитном порядке по названи­ ям магазинов); • вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры (если такого магазина нет, вывести соответст­ вующее сообщение). Вариант 20 Описать структуру с именем ORDER, содержащую следующие поля: • расчетный счет плательщика; • расчетный счет получателя; • перечисляемая сумма в рублях. Написать программу, выполняющую следующие действия: • ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ORDER (записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков); • вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры (если такого расчетного счета нет, вывести соответ­ ствующее сообщение). Лабораторная работа 11. Интерфейсы и параметризованные коллекции Теоретический материал: главы 9, 13. Выполнить задания лабораторной работы 9, используя для хранения экземпля­ ров разработанных классов стандартные параметризованные коллекции. Во всех классах реализовать интерфейс IComparable и перегрузить операции отношения для реализации значимой семантики сравнения объектов по какому-либо полю на усмотрение студента. 412 Лабораторные работы Лабораторная работа 12. Создание Windows-приложений Теоретический материал: глава 14. Задание 1. Диалоговые окна Общая часть задания: написать Windows-приложение, заголовок главного окна которого содержит Ф. И. О., группу и номер варианта. В программе должна быть предусмотрена обработка исключений, возникающих из-за ошибочного ввода пользователя Вариант 1 Создать меню с командами Input, Calc и Exit. При выборе команды Input открывается диалоговое окно, содержащее: • три поля типа TextBox для ввода длин трех сторон треугольника; • группу из двух флажков (Периметр и Площадь) типа CheckBox; • кнопку типа Button. Обеспечить возможность: • ввода длин трех сторон треугольника; • выбора режима с помощью флажков: подсчет периметра и/или площади треугольника. При выборе команды Calc открывается диалоговое окно с результатами. При вы­ боре команды Exit приложение завершается. Вариант 2 Создать меню с командами Size, Color, Paint, Quit. Команда Paint недоступна. При выборе команды Quit приложение завершается. При выборе команды Size открывается диалоговое окно, содержащее: • два поля типа TextBox для ввода длин сторон прямоугольника; • группу из трех флажков (Red, Green, Blue) типа CheckBox; • кнопку типа Button. Обеспечить возможность: • ввода длин сторон прямоугольника в пикселах в поля ввода; • выбора его цвета с помощью флажков. После задания параметров команда Paint становится доступной. При выборе команды Paint в главном окне приложения выводится прямоуголь­ ник заданного размера и сочетания цветов или выдается сообщение, если вве­ денные размеры превышают размер окна. Лабораторная работа 12. Создание Windows-приложений 413 Вариант 3 Создать меню с командами Input, Work, Exit. При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно, содержащее: • три поля ввода типа TextBox с метками Radius, Height, Density; • группу из двух флажков (Volume, Mass) типа CheckBox; • кнопку типа Button. Обеспечить возможность: • ввода радиуса, высоты и плотности конуса; • выбора режима с помощью флажков: подсчет объема и/или массы конуса. При выборе команды Work открывается окно сообщений с результатами. Вариант 4 Создать меню с командами Input, Calc, Draw, Exit. При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно, содержащее: • поле ввода типа TextBox с меткой Radius; • группу из двух флажков (Square, Length) типа CheckBox; • кнопку типа Button. Обеспечить возможность: • ввода радиуса окружности; • выбора режима с помощью флажков: подсчет площади круга (Square) и/или длины окружности (Length). При выборе команды Calc открывается окно сообщений с результатами. При вы­ боре команды Draw в центре главного окна выводится круг введенного радиуса или выдается сообщение, что рисование невозможно (если диаметр превышает размеры рабочей области). Вариант 5 Создать меню с командами input, Calc, About. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Input открывается диалоговое окно, содержащее: • три поля ввода типа TextBox с метками Number 1, Number 2, Number 3; • группу из двух флажков (Summ, Least multiple) типа CheckBox; • кнопку типа Button. Обеспечить возможность ввода трех чисел и выбора режима вычислений с помо­ щью флажков: подсчет суммы трех чисел (Summ) и/или наименьшего общего кратного двух первых чисел (Least multiple). При выборе команды Calc открыва­ ется диалоговое окно с результатами. 414 Лабораторные работы Вариант 6 Создать меню с командами Input, Calc, Quit. Команда Calc недоступна. При выборе команды Quit приложение завершается. При выборе команды Input открывается диалоговое окно, содержащее: • два поля ввода типа TextBox с метками Number 1, Number 2; • группу из трех флажков (Summa, Max divisor, Multiply) типа CheckBox; • кнопку типа Button. Обеспечить возможность: • ввода двух чисел; • выбора режима вычислений с помощью флажков (можно вычислять в любой комбинации такие величины, как сумма, наибольший общий делитель и про­ изведение двух чисел). При выборе команды Calc открывается окно сообщений с результатами. Вариант 7 Создать меню с командами Begin, Help, About. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее: • поле ввода типа TextBox с меткой input; • метку типа Label для вывода результата; • группу из трех переключателей (2, 8, 16) типа RadioButton; • две кнопки типа Button — Do и ОК. Обеспечить возможность: • ввода числа в десятичной системе в поле input; • выбора режима преобразования с помощью переключателей: перевод в дво­ ичную, восьмеричную или шестнадцатеричную систему счисления. При щелчке на кнопке Do должен появляться результат перевода. Вариант 8 Создать меню с командами Input color, Change, Exit, Help. При выборе команды Exit приложение завершает работу. При выборе команды Input color открывается диалоговое окно, содержащее: • три поля ввода типа TextBox с метками Red, Green, Blue; • группу из двух флажков (Left, Right) типа CheckBox; • кнопку типа Button. Обеспечить возможность ввода RGB-составляющих цвета. При выборе команды Change цвет главного окна изменяется на заданный (левая, правая или обе поло­ вины окна в зависимости от установки флажков). Лабораторная работа 12. Создание Windows-приложений 415 Вариант 9 Создать меню с командами Input size, Choose, Change, Exit. При выборе команды Exit приложение завершает работу. Команда Change недос­ тупна. При выборе команды Input size открывается диалоговое окно, содержащее: • два поля ввода типа TextBox с метками Size х, Size у; • кнопку типа Button. При выборе команды Choose открывается диалоговое окно, содержащее: • группу из двух переключателей (Increase, Decrease) типа RadloButton; • кнопку типа Button. Обеспечить возможность ввода значений в поля Size х и Size у. Значения интерпре­ тируются как количество пикселов, на которое надо изменить размеры главного окна (увеличить или уменьшить в зависимости от положения переключателей). После ввода значений команда Change становится доступной. При выборе этой команды размеры главного окна увеличиваются или уменьшаются на введенное количество пикселов. Вариант 10 Создать меню с командами Begin, Work, About. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее: • поле ввода типа TextBox с меткой Input word; • группу из двух переключателей (Upper case, Lower case) типа RadioButton; • кнопку типа Button. Обеспечить возможность ввода слова и выбора режима перевода в верхний или нижний регистр в зависимости от положения переключателей. При выборе ко­ манды Work открывается диалоговое окно с результатом перевода. Вариант 11 Создать меню с командами Input color, Change, Clear. При выборе команды Input color открывается диалоговое окно, содержащее: • группу из двух флажков (Up, Down) типа CheckBox; • группу из трех переключателей (Red, Green, Blue) типа RadioButton; • кнопку типа Button. Обеспечить возможность: • выбора цвета с помощью переключателей; • ввода режима, определяющего, какая область закрашивается: все окно, его верхняя или нижняя половина. При выборе команды Change цвет главного окна изменяется на заданный (верх­ няя, нижняя или обе половины в зависимости от введенного режима). При вы­ боре команды Clear восстанавливается первоначальный цвет окна. 416 Лабораторные работы Вариант 12 Создать меню с командами Translate, Help, About, Exit. При выборе команды Exit приложение завершает работу. При выборе команды Translate открывается диалоговое окно, содержащее: • поле ввода типа TextBox с меткой Binary number; • поле ввода типа TextBox для вывода результата (read-only); • группу из трех переключателей (8, 10, 16) типа RadioButton; • кнопку Do типа Button. Обеспечить возможность: • ввода числа в двоичной системе в поле Binary number; • выбора режима преобразования с помощью переключателей: перевод в вось­ меричную, десятичную или шестнадцатеричную систему счисления. При щелчке на кнопке Do должен появляться результат перевода. Вариант 13 Создать меню с командами Reverse, About, Exit. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Reverse открывается диалоговое окно, содержащее: • поле ввода типа TextBox с меткой Input; • группу из двух переключателей (Upper case, Reverse) типа CheckBox; • кнопку ОК типа Button. Обеспечить возможность ввода фразы и выбора режима: перевод в верхний регистр и/или изменение порядка следования символов на обратный в зависи­ мости от состояния переключателей. Результат преобразования выводится в ис­ ходное поле ввода. Вариант 14 Создать меню с командами Input, Show и Exit. При выборе команды Exit приложение завершает работу. При выборе команды Input открывается диалоговое окно вида: Лабораторная работа 12. Создание Windows-приложений 417 Обеспечивается возможность ввода координат двух точек и выбора режима с по­ мощью флажков length и koef: подсчет длины отрезка, соединяющего эти точки, и/или углового коэффициента. При выборе команды Show открывается окно сообщений с результатами под­ счета. Вариант 15 Создать меню с командами Input, About и Exit. При выборе команды Exit приложение завершает работу. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Input открывается диалоговое окно вида: Обеспечивается возможность ввода суммы в рублях и перевода ее в евро и долла­ ры по обычному или льготному курсу. Поля Euro и $ доступны только для чтения. Вариант 16 Создать меню с командами Begin, Work, About. При выборе команды About открывается окно с информацией о разработчике. При выборе команды Begin открывается диалоговое окно, содержащее: • два поля ввода типа TextBox; • группу из двух переключателей (First letter, All letters) типа RadioButton; • кнопку типа Button. Обеспечить возможность ввода предложения и выбора режима его преобразова­ ния: либо начинать с прописной буквы каждое слово (First letter), либо перевести все буквы в верхний регистр (All letters). При выборе команды Work открывается диалоговое окно с результатом преобразования. Вариант 17 Написать анализатор текстовых файлов, выводящий информацию о количе­ стве слов в тексте, а также статистическую информацию о введенной пользо­ вателем букве. 418 Лабораторные работы Создать следующую систему меню: • Файл О Загрузить текст О Выход - • Анализ О Количество слов О Повторяемость буквы При выборе файла для загрузки использовать объект типа OpenFileDialog. При выборе команды Количество слов программа должна вывести в окно сообщений количество слов в тексте. При выборе команды Повторяемость буквы программа предлагает пользователю ввести букву, а затем выводит количество ее повторений без учета регистра в окно сообщений. Вариант 18 Создать редактор текстовых файлов с возможностью сохранения текста в фор­ мате HTML. Создать следующую систему меню: • Файл О Загрузить текст О Сохранить как текст О Сохранить как HTML • Выход При выборе файла для загрузки использовать объект OpenFi leDialog. При выборе файла для сохранения использовать объект SaveFileDlalog. Для редактирования текста использовать объект Memo. При сохранении текста в формате HTM L текст записывать в файл с заменой: • всех пробелов на символы  ; • всех символов перевода строки на символы   
; • всех символов < на символы <; • всех символов > На символы >:; • всех символов & на символы &атр;; • всех символов " (двойные кавычки) на символы Squot;. Вариант 19 Создать меню с командами Input, Draw, Clear. При выборе команды Input открывается диалоговое окно, содержащее: • четыре поля для ввода координат двух точек; • группу из трех переключателей (Red, Green, Blue) типа RadioButton; • кнопку типа Button. Лабораторная работа 12. Создание Windows-приложений 419 При выборе команды Draw в главное окно выводится отрезок прямой выбранно­ го цвета с координатами концов отрезка, заданными в диалоговом окне. При вы­ боре команды Clear отрезок стирается. Вариант 20 Создать меню с командами Input, Change, Exit. При выборе команды Exit приложение завершает работу. Команда Change недос­ тупна. В центре главного окна выведен квадрат размером 100 х 100 пикселов. При выборе команды Input открывается диалоговое окно, содержащее: • два поля ввода типа TextBox с метками Size х, Size у; • группу из двух переключателей (Increase, Decrease) типа RadioButton; • кнопку типа Button. Обеспечить возможность ввода значений в поля Size х и Size у. Значения интер­ претируются как количество пикселов, на которое надо изменить размеры квад­ рата, выведенного в главное окно (увеличить или уменьшить в зависимости от положения переключателей). После ввода значений команда Change становится доступной. При выборе этой команды размеры квадрата увеличиваются или уменьшаются на введенное коли­ чество пикселов. Если квадрат выходит за пределы рабочей области окна, выда­ ется сообщение. Вариант 21 Написать Windows-приложение, которое по заданным в файле исходным дан­ ным выводит информацию о компьютерах. Создать меню с командами Choose, Show, Quit. Команда Sho w недоступна. Команда Quit завершает работу приложения. При запуске приложения из файла читаются исходные данные. Файл необходи­ мо сформировать самостоятельно. Каждая строка файла содержит тип компью­ тера, цену (price) и емкость жесткого диска (hard drive). При выборе команды Choose открывается диалоговое окно, содержащее: • поле типа TextBox для ввода минимальной емкости диска; • поле типа TextBox для ввода максимальной приемлемой цены; • группу из двух переключателей (Hard drive, Price) типа RadioButton; • OK, Cancel — кнопки типа Button. После ввода всех данных команда меню Sho w становится доступной. Команда Sho w открывает диалоговое окно, содержащее список компьютеров, удовле­ творяющий введенным ограничениям и упорядоченный по отмеченной харак­ теристике. 422 Лабораторные работы Из выбранного файла читаются исходные данные для сортировки (сформировать самостоятельно не менее трех файлов различной длины с данными целого типа). После чтения данных становится доступной команда Animate. При выборе команды Animate в главном окне приложения отображается процесс сортировки в виде столбиковой диаграммы. Каждый элемент представляется столбиком соответствующего размера. На каждом шаге алгоритма два элемента меняются местами. Окно должно содержать заголовок. Изображение должно за­ нимать все окно. Вариант 5 Написать Windows-приложение — графическую иллюстрацию аппроксимации методом наименьших квадратов зависимости у = а-х + Ь-х + с -log2 х. Создать меню с командами Open, Coefficients, Show, About, Exit. Команда Exit завершает работу приложения. Команда About открывает окно с ин­ формацией о разработчике. Для выбора файла исходных данных (команда Open) использовать объект OpenFileDialog. Исходные данные для аппроксимации — массивы экспериментальных значений аргумента х и функции у(х) — сформи­ ровать самостоятельно. При выборе команды Coefficients выводится окно сообщений с вычисленными коэффициентами а,Ьис. При выборе команды Show в главном окне приложения отображаются график зависимости и исходные данные в виде точек. Окно должно содержать заголовок. Изображение должно занимать все окно. Приложение Спецификаторы формата для строк С# Спецификаторы используются для форматирования арифметических типов при их преобразовании в строковое представление. Спецификатор Описание С или с Вывод значений в денежном (currency) формате. По умолчанию перед выводимым значением подставляется символ доллара ($). Этот символ, а также заданное по умолчанию количество позиций можно изменить при помощи объекта NumberFormatlnfo. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части D или d Вывод целых значений. Непосредственно после спецификатора можно задать целое число, определяющее ширину поля вывода. Недостающие места заполняются нулями, например, вывод числа 12 по формату D3 выглядит как 012 Е или е Вывод значений в экспоненциальном формате, то есть в виде d.ddd..E+ddd или d.ddd...e+ddd. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части. Минимальная длина порядка — 3 символа F или f Вывод значений с фиксированной точностью. Непосредственно после спецификатора можно задать целое число, определяющее длину дробной части, например, число 0,12, выведенное по формату F3, выглядит как 0,120 G или g Формат общего вида. Применяется для вывода значений с фиксированной точностью или в экспоненциальном формате в зависимости от того, какой формат требует меньшего количества позиций. Для различных типов величин по умолчанию используется разная ширина вывода, например; для single — 7 позиций, для byte и sbyte — 3, для decimal — 29 продолжение & 424 Приложение. Спецификаторы формата для строк С# [продолжение) Спецификатор Описание N или п Вывод значений в формате d,ddd,ddd.ddd..., то есть группы разрядов разделяются разделителями, соответствующими региональным настройкам. Непосредственно после спецификации можно задать целое число, определяющее длину дробной части Р или р Вывод числа в процентном формате (число, умноженное на 100, после которого выводится знак %) R или г Отмена округления числа при преобразовании в строку. Гарантирует, что при обратном преобразовании в значение того же типа получится то же самое X или х Вывод значений в шестнадцатеричном формате. Если используется прописная буква X, то буквенные символы в шестнадцатеричных символах также будут прописными Пример применения спецификаторов: using System; namespace ConsoleApplicationl { class Classl { static void MainO { int i = 1234; Console.WriteLineC i.ToStringC "C" ) ); Console.WriteLineC i.ToStringC "D5" ) ); Console.WriteLineC i.ToStringC "E" ). ); Console.WriteLineC i.ToStringC "G" ) ); Console.WriteLineC "{0,9:n2}", i ); Console.WriteLineC "{0,l:p3}", i ); Console.WriteLineC "{0.1:x}", i ); } } } Результат работы программы: 1 234.00р. 01234 1.234000Е+003 1234 1 234,00 123 400,000% 4d2 Другой пример использования спецификаторов приведен на с. 146. Список литературы 1. Биллиг В. А. Основы программирования на С#. - М.: Изд-во «Интернет-уни­ верситет информационных технологий - ИНТУИТ.ру» , 2006. - 488 с. 2. Брукс Ф. Мифический человеко-месяц, или как создаются программные ком­ плексы. - М.: Символ-Плюс, 2000. - 304 с. 3. Ватсон К. С#. - М.: Лори, 2004. - 880 с. 4. Вирт Н. Алгоритмы и структуры данных. - СПб: Невский диалект, 2001. - 352 с. 5. Гиббонз П. Платформа .NET для Java-программистов. - СПб.: Питер, 2003. - 336 с. 6. Голуб А. И.Си С++. Правила программирования. - М.: Бином, 1996. - 272 с. 7. Гуннерсон Э. Введение в С#. Библиотека программиста. - СПб.: Питер, 2001. - 304 с. 8. Кораблев В. Самоучитель Visual С+ + .NET. - СПб.: Питер; Киев: Издатель­ ская группа BHV, 2004. - 528 с. 9. Либерти Д. Программирование на С#. - СПб.: Символ-Плюс, 2003. - 688 с. 10. Майо Д. С#. Искусство программирования: Энциклопедия программиста. — Киев: ДиаСофт, 2002. - 656 с. 11. МайоДж. С # Builder. Быстрый старт. — М.: Бином, 2005. - 384 с. 12. Микелсен К. Язык программирования С#. Лекции и упражнения: Учебник. - Киев: ДиаСофт, 2002. - 656 с. 13. Оньон Ф. Основы ASP.NET с примерами на С# . - М.: Издательский дом «Вильяме», 2003. - 304 с. 14. Павловская Т. А. C/C++ . Программирование на языке высокого уровня: Учеб­ ник для вузов. - СПб.: Питер, 2001. - 464 с. 15. Павловская Т. А. Паскаль. Программирование на языке высокого уровня: Учебник для вузов. - СПб.: Питер, 2003. - 393 с. 426 Список литературы 16. Паппас К., Мюррей У. Эффективная работа: Visual С+ + .NET. — СПб.: Питер, 2002. - 816 с. 17. Петцолъд Ч. Программирование для MS Windows на С#. Т. 1. — М.: Изда- тельско-торговый дом «Русская Редакция», 2002. — 576 с. 18. Петцолъд Ч. Программирование для MS Windows на С#. Т. 2. — М.: Изда- тельско-торговый дом «Русская Редакция», 2002. — 624 с. 19. Пономарев В. А. Программирование на С++/С # в Visual Studio .NET 2003. Серия «Мастер программ». — СПб.: БХВ-Петербург, 2004. — 352 с. 20. Прайс Д., Гандэрлой М. Visual C#.NET. Полное руководство. — Киев: Век, 2004. - 960 с. 21. Робинсон С, Корнес О., ГлиннДж. и др. С# для профессионалов. В 2 т. — М.: Лори, 2003. - 512 с. 22. Робисон У. С# без лишних слов. — М.: ДМК Пресс, 2002. — 352 с. 23. Секунов Н. Разработка приложений на С+ + и С#. Библиотека программиста. — СПб.: Питер, 2003. - 608 с. 24. Секунов Н. Самоучитель С#. Серия "Самоучитель". — СПб.: БХВ-Петербург, 2001. - 576 с. 25. Смайли Д. Учимся программировать на С # вместе с Джоном Смайли. — Киев: «ДиаСофт», 2003. - 528 с. 26. Тай Т., Лэм X. К. Платформа .NET. Основы. — СПб.: Символ-Плюс, 2003. — 336 с. 27. Троелсен Э. С # и платформа .NET. Библиотека программиста. — СПб.: Питер, 2002. - 796 с. 28. Фролов А. В., Фролов Г. В. Язык С# : Самоучитель. — М.: Диалог-МИФИ, 2003. - 560 с. 29. Шилдт Г. С#: Учебный курс. - СПб.: Питер, 2002. - 512 с: ил. 30. Шилдт Г. Полный справочник по С#. — М.: Издательский дом «Вильяме», 2004. - 752 с. 31. Microsoft Corporation. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс. Сертификационный экзамен № 70-306, 70-316. — М.: Издательско-торговый дом «Русская Редак­ ция», 2003. - 512 с. 32. Microsoft Corporation. Разработка Web-приложений на Microsoft Visual Basic .NET и Microsoft Visual C # .NET. Учебный курс MCAD/MCSD . - M.: Изда­ тельско-торговый дом «Русская Редакция», 2003. — 704 с. 33. Microsoft Corporation. Разработка Web-сервисов XM L и серверных компо­ нентов на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс MCAD / MCSD . — M.: Издательско-торговый дом «Русская Редакция», 2004. - 576 с. Алфавитный указатель !, операция, 43, 49 I-, операция, 43, 54 #define, директива, 288 #elif, директива, 288 #else, директива, 288 #endif, директива, 288 #endregion, директива, 288 #еггог, директива, 288 #if, директива, 288 #line, директива, 288 #pragma, директива, 288 #region, директива, 288 #warning, директива, 288 %, операция, 43, 51 % = , операция, 43 &&, операция, 43, 56 &, операция, 43, 55 &=, операция, 44 ( ) , операция, 42 \*, операция,. 43, 50 \* = , операция, 43, 57 , операция, 42 -, операция, 43 --, операция, 42, 43, 47 -, операция, 48, 53 ., операция, 42 /, операция, 43, 50 / = , операция, 43 ?, операция, 43, 56 ??, операция, 310 @, управляющий символ, 30 Л , операция, 43, 55 А = , операция, 44 |, операция, 43, 55 |, операция, 43, 56 |=, операция, 44 -, операция, 43, 49 +, операция, 43, 52 ++, операция, 42, 43, 47 +=, операция, 43, 57 < =, операция, 43, 54 >, операция, 43, 54 », операция, 43, 54 » = , операция, 4 4 > = , операция, 43, 54 =, операция, 43, 57 -=, операция, 44, 57 —, операция, 43, 54 А - С Abort, метод, 240 Abs, метод, 65 abstract, специ фикатор, 101,181 Acos, метод, 65 ADO.NET, 367 ANSI, 22 API, 312 Append, метод, 147 AppendFormat, метод, 147 Application, класс, 342 Array, класс, 133 ArrayList, класс, 296 as, операция, 43, 194 Asin, метод, 65 ASP.NET, 367 AsyncCallback, делегат, 242, 255 Atan, метод, 65 Atan2, метод, 65 base, ключевое слово, 117, 175 Beginlnvoke, метод, 242 BeginRead, метод, 250, 253 BeginWrite, метод, 250 BigMul, метод, 65 BinaryFormatter, класс, 268 Binary Reader, класс, 261 Binary Search, метод, 134 BinaryWriter, класс, 260 break, оператор, 84 Brush, класс, 344 Button, класс, 326 Capture, класс, 363 case, ключевое слово, 74 catch, ключевое слово, 90 Ceiling, метод, 65 char, ключевое слово, 139 CheckBox; класс, 331 checked, операция, 42, 46 class, ключевое слово, 100, 305 Close, метод, 250, 255 CLR, 9 CLS, 14 Color, класс, 344 Combine, метод, 222 Compare, метод, 143 CompareOrdinal, метод, 144 CompareTo, метод, 144 Component, класс, 337 Concat, метод, 144 Console, класс, 19, 59, 262 const, ключевое слово, 41 ContainerControl, класс, 338 ContextMenu, класс, 331 continue, оператор, 86 Control, класс, 323, 338 Convert, класс, 61 Сору, метод, 134, 144 Cos, метод, 65 Cosh, метод, 65 D - F DataSet, класс, 367 decimal, тип, 34 default, ключевое слово, 74, 306 delegate, ключевое слово, 220 Deserialize, метод, 268 DialogResult, перечисление, 327 Dictionary, класс, 302 Directory, класс, 263 Directorylnfo, класс, 263 Dispose, метод, 345 DivRem, метод, 65 DLL, 312 do while, оператор, 79 Е, поле, 65 Endlnvoke, метод, 242 EndRead, метод, 250, 253 EndWrite, метод, 250 Enum, класс, 218 enum, ключевое слово, 215 Epsilon, константа, 73 Equals, метод, 184 е8саре-пЪследовательность, 24,29 event, ключевое слово, 233 EventArgs, класс, 235 EventHandler, делегат, 235 Exception, класс, 89, 94 Ехр, метод, 65 explicit, ключевое слово, 168 File, класс, 263 FileAccess, перечисление, 249 FileAttributes, перечисление, 265 Filelnfo, класс, 263 428 Алфавитный указатель FileMode, перечисление, 249 FileShare, перечисление, 249 FileStream, класс, 250 FileSystemlnfo, класс, 263 finally, ключевое слово, 90 FindMembers, метод, 280 fixed, оператор, 352 Floor, метод, 65 Flush, метод, 250, 255 Font, класс, 344 for, оператор, 81 foreach, оператор, 136 Form, класс, 337, 338 Format, метод, 144 G - J GAC, 275 get, ключевое слово, 121 GetConstructors, метод, 280 GetEnumerator, метод, 207 GetEvents, метод, 280 GetFields, метод, 280 GetHashCode, метод, 184 Getlnterfaces, метод, 280 GetMembers, метод, 280 GetMethods, метод, 280 GetName, метод, 218 GetNestedTypes, метод, 280 GetNumericValue, метод, 140 GetProperties, метод, 280 GetType, метод, 184,280 global, идентификатор, 287 goto, оператор, 83 Graphics, класс, 344 Group, класс, 363 GroupBox, класс, 332 IAsyncResult, интерфейс, 255 ICloneable, интерфейс, 205 IComparable, интерфейс, 199 IComparer, интерфейс, 200 IDisposable, интерфейс, 345 IEEERemainder, метод, 65 IEnumerable, интерфейс, 198, 207, 219 IEnumerator, интерфейс, 198, 207, 219 if, оператор, 70 IIS, 368 ILDasm, дизассемблер, 273 implicit, ключевое слово, 168 IndexOf, метод, 134 Insert, метод, 144 internal, спецификатор, 101 is, операция, 43, 194 IsControl, метод, 140 IsDigit, метод, 140 IsLetter, метод, 140 IsLetterOrDigit, метод, 140 IsLower, метод, 140 IsMatch, метод, 360 IsNumber, метод, 140 IsPunctuation, метод, 140 IsSeparator, метод, 140 IsUpper, метод, 140 IsWhiteSpace, метод, 140 JIT-компилятор, 9 Join, метод, 144 L — N Label, класс, 326 LastlndexOf, метод, 134 Length, свойство, 144 List, класс, 300 ListBox, класс, 334 lock, оператор, 242 Log, метод, 65 Log 10, метод, 66 Main, метод, 156 MainMenu, класс, 330 MarshalByRefObject, класс, 337 Match, класс, 361 Match, метод, 361 MatchCollection, класс, 362 Matches, метод, 362 MatchEvaluator, делегат, 363 Мах, метод, 66 MaxValue, метод, 34, 140 MDI, 337 MemberwiseClone, метод, 205 Menultem, класс, 330 MessageBox, класс, 322 Min, метод, 66 MinValue, метод, 34, 140 Monster, класс, 119 MSIL, 9 NaN, не число, 34 new ключевое слово, 175, 305 операция, 42, 48, 102 спецификатор, 101 Next, метод, 149 NextBytes, метод, 149 NextDouble, метод, 149 NextMatch, метод, 362 null, константа, 30 nullable, тип, 36,309 O - R object, класс, 13, 183 OLE, 312 operator, ключевое слово, 161 out, спецификатор, 113 override, ключевое слово, 179 params, ключевое слово, 154 Parse, метод, 61, 140 partial, модификатор, 308 Peek, метод, 256 Реп, класс, 344 PI, 66 Pow, метод, 66 private, спецификатор, 101 protected, спецификатор, 101 public, спецификатор, 101 RadioButton, класс, 331 Random, класс, 148 Read, метод, 62, 250 ReadBlock, метод, 256 ReadByte, метод, 250 ReadLine, метод, 62 readonly, спецификатор, 105 ReadToEnd, метод, 256 ref, ключевое слово, 112 ReferenceEquals, метод, 184 Regex, класс, 360 RegexOptions, перечисление, 360 Remove, метод, 144, 225 Replace, метод, 144, 362 return, оператор, 87 Reverse, метод, 134 Round, метод, 66 Run, метод, 316 S — И sealed, ключевое слово, 101,182 Seek, метод, 250 Serialize, метод, 268 set, ключевое слово, 121 Sign, метод, 66 Sin, метод, 66 Sinh, метод, 66 sizeof, операция, 352 Sleep, метод, 240 Sort, метод, 134 Split, метод, 144,303,362 Sqrt, метод, 66 stackalloc, операция, 354 Start, метод, 241 static одификатор, 118 спецификатор, 101 Stream, класс, 250 StreamReader, класс, 255 Stream Writer, класс, 255 string, тип, 143 StringBuilder, класс, 147, 151 strong name, 275 struct, ключевое слово, 212, 305 Алфавитный указатель 429 Substring, метод, 144 switch, оператор, 73 System. Collections, пространство имен, 295 System.Collections.Generic, пространство имен, 295, 300 System.Collections.Specialized, пространство имен, 295 System.Drawing, пространство имен, 344 System.Reflection, пространство имен, 279 System.Text. RegularExpression, пространство имен, 355 System.Windows.Forms, пространство имен, 315 Tan, метод, 66 Tanh, метод, 66 TextBox, класс, 327 TextReader, класс, 255 TextWriter, класс, 255 this, ключевое слово, 114 Thread, класс, 239 ThreadStart, делегат, 239 throw, оператор, 93 ToCharArray, метод, 144 ToLower, метод, 140 ToString, метод, 60, 184 ToUpper, метод, 140 try, оператор, 90 Туре, класс, 280 typeof, операция, 42 unchecked, операция, 42, 46 Unicode, кодировка, 22, 24 unsafe оператор, 348 спецификатор, 348 using директива, 286 оператор, 345 V-Z value, ключевое слово, 122 virtual, ключевое слово, 178 VMT, 178 void, тип, 109 volatile, спецификатор, 105 where, ключевое слово, 305 while, оператор, 76 Write, метод, 60, 250 WriteByte, метод, 250 WriteLine, метод, 19, 60 XML, 365 yield, ключевое слово, 208 А - В абстрагирование, 11 абстрактная структура, 294 абстрактный класс, 181 албанский язык, 22 алфавит, 22 анализ лексический, 24 синтаксический, 24 анонимный метод, 228 арифметический тип, 32 арифметическое отрицание, 48 асинхронный ввод-вывод, 253 асинхронный делегат, 242 ассоциативный массив, 293 атрибут, 283 базовая конструкция, 87 базовый класс среды, 10 базовый тип перечисления, 215 бесплодный класс, 182 библиотека динамическая, 312 классов, 10 бинарная операция, 164 бинарное дерево, 292 блок, 40,70 буфер, 62,246 ввод данных, 246 веб-приложение, 368 веб-сервер, 368 веб-служба, 369 версия информационная, 274 совместимости, 274 ветвление, 87 вещественный тип, 33 виртуальная DOS-машина, 312 виртуальный каталог, 368 виртуальный метод, 178 визуальное проектирование, 317 вложение, 182 вложенный класс, 101 вложенный тип, 169 встроенный тип, 32 вторичный поток, 239 вывод, 246 выражение, 23, 42, 70, 355 выходной параметр метода, ИЗ вычитание, 53 г-з главное меню, 15, 330 главное окно, 337 глобальный кэш сборок, 275 граф, 294 данные класса, 102 экземпляра, 102 двоичный файл, 248, 260, 271 двунаправленный список, 291 двусвязный список, 291 декремент, 47 делегат, 220 асинхронный, 242 использование, 221 описание, 220 передача в метод, 226 регистрация, 225 делегирование, 182 деление, 50 дерево бинарное, 292 поиска, 293 десериализация, 267 деструктор, 169 диалоговое окно, 326, 337 динамическая библиотека, 312 директива препроцессора, 23, 287 документирование в формате XML, 365 домен, 237 дополнительный код, 32 дополнительный номер версии, 274 дословный литерал, 30 доступ последовательный, 248, 271 произвольный, 248 прямой, 248 дочернее окно, 337 заголовок метода, 106 заготовка формы, 314 закрытый конструктор, 117 запись данных, 246 заплатка, 274 знак операции, 25, 42 значение по умолчанию, 48 значимый тип, 35 и - к идентификатор версии, 274 правильный, 24 программного объекта, 24 иерархия объектов, 12 именованный параметр, 284 имя, 24 класса, 100 метода, 19 сильное, 275 430 Алфавитный указатель индекс, 126 индексатор, 157, 161 инициализатор, 116 инициализация, 39 инкапсуляция, 12 инкремент, 47 инстанцирование, 305 интерфейс, 12, 124, 188 многодокументный, 337 параметризованный, 307 информационная версия, 274 информация о безопасности, 274 исключение, 46, 89 исключительная ситуация, 89 источник, 223 итератор, 207 итерация, 75 каркас, 11 каталог виртуальный, 368 физический, 368 класс, 13, 100 абстрактный, 181 бесплодный, 182 вложенный, 101, 169 потомок, 172 предок, 172 прототип, 299 ' статический, 118 клонирование поверхностное, 205 ключ, 292 ключевое слово, 25 кнопка по умолчанию, 326 код дополнительный, 32 доступа, 121, 158 небезопасный, 347 кодовая таблица, 22 коллекция, 295 параметризованная, 300 коллизия, 294 кольцевой список, 292 командная строка, 15 комментарий, 23, 359 многострочный, 30 однострочный, 30 компилятор, 8 компонент, 314, 325 сущности, 367 управляемого поставщика, 367 консоль, 59 консольное окно, 15 консольное приложение, 15 константа, 26, 42 конструктор, 114 закрытый, 117 класса, 117 наследование, 174 по умолчанию, 48,115 статический, 117 экземпляра, 117 контейнер, 295 контекстное меню, 331 корень дерева, 292 кэш, 9 л - н лексема, 23 лексический анализ, 24 линейная программа, 59 лист, 293 литерал, 26 дословный, 30 строковый, 29 логическое отрицание, 49 локальная переменная, 39 манифест, 273 мантисса, 28 массив, 126, 291 ассоциативный, 293 инициализация, 129 как параметр метода, 156 одномерный, 128 прямоугольный, 130 размерность, 127 создание, 126 ступенчатый, 132 меню главное, 330 контекстное, 331 метаданные, 9, 273 метасимвол, 355 метка, 84 метод, 18, 106 анонимный, 228 виртуальный, 178 класса, 102 обобщенный, 306 перегрузка, 152 полиморфный, 182 рекурсивный, 153 статический, 102 универсальный, 226 экземпляра, 102 многодокументный интерфейс, 337 многозадачность, 311 многомерный индексатор, 161 многострочный комментарий, 30 множество, 294 модальное окно, 337 модальность, 326 модель включения- делегирования, 182 наблюдатель, 223 наследование, 12, 172 небезопасный код, 347 небезопасный контекст, 348 неизменяемый тип данных, 143 немодальное окно, 337 неявное преобразование типа, 45 номер версии, 274 ревизии, 274 сборки, 274 нотация, 25 о-п область действия переменной, 40 параметров, 111 обнуляемый тип, 309 обобщенный метод, 306 оболочка, 8 обработка событий, 233 обработчик исключений, 91 событий, 313 обратная ссылка, 359 обратный вызов, 226 общеязыковая спецификация, 14 общеязыковая среда выполнения, 9 объект, 11 класса, 101 перечислитель, 211 объектно-ориентированное программирование, 69 объявитель операции, 161 ограничение, 305 одномерный массив, 128 однонаправленный список, 291 односвязный список, 291 однострочный комментарий, 30 окно, 311 главное, 337 диалоговое, 326, 337 дочернее, 337 модальное, 337 немодальное, 337 редактора, 17 родительское, 337 свойств, 16 сообщений, 322 управления проектом, 16 Алфавитный указатель 431 операнд, 42 оператор, 24 break, 84 continue, 86 lock, 242 безусловного перехода, 83 ветвления, 70 возврата из функции, 87 выбора, 73 перебора элементов, 136 передачи управления, 83 пустой, 70 составной, 70 цикла, 75 операция, 42 бинарная, 164 выделения памяти в стеке, 354 вычитания, 53 декремента, 47 деления, 50 доступа, 105, 353 инкремента, 47 класса, 161 логическая, 56 логического отрицания, 49 объединения, 310 остатка от деления, 51 отношения, 54 перегрузка, 161 поразрядная, 55 поразрядного отрицания, 49 преобразования типа, 49, 168 присваивания, 57 разадресации, 351 разыменования, 351 сдвига, 54 сложения, 52 создания объекта, 48 умножения, 50 унарная, 162 унарного минуса, 48 условная, 56 описание класса, 18, 100 основной номер версии, 274 остаток от деления, 51 открытая сборка, 273 отправитель, 232 отрицание арифметическое, 48 логическое, 49 поразрядное, 49 очередь, 292 панель инструментов, 15 параметр выходной, 113 делегата, 220 значение, 111 именованный, 284 массив, 156 метода, 107 позиционный, 284 ссылка, 112 цикла, 76 параметризованная коллекция, 300 параметризованный интерфейс, 307 паттерн «наблюдатель», 223 патч, 274 первичный поток, 238 перегруженный метод, 60 перегрузка метода, 152 операций, 161 передача по адресу, ПО по значению, 110 по ссылке, 110 переменная, 38 локальная, 39 перечисление, 215 % перечислитель, 211 перечисляемый тип, 215 платформа, 8 поверхностное клонирование, 205 повторитель, 357 погрешность, 86 позднее связывание, 178 позиционный параметр, 284 поле класса, 39, 104 полиморфизм, 13,153,180,188 полиморфный метод, 182 получатель, 232 поразрядная операция, 55 поразрядное отрицание, 49 порядок, 28 последовательный доступ, 248, 271 поток, 311 ввода-вывода, 246 вторичный, 239 выполнения, 238 первичный, 238 потомок, 172 правильный идентификатор, 24 предок, 172 преобразование типа, 45, 49 приложение, 11 приоритет, 44 присваивание операция, 57 простое, 57 структуры, 214 пробельный символ, 23 проверяемый контекст, 47 программа линейная, 59 программирование объектно- ориентированное, 69 структурное, 69 программный интерфейс приложения, 312 проект, 14 произвольный доступ, 248 промежуточный язык, 273 простое присваивание, 57 пространство имен, 13, 285 прототип, 299 процесс, 237 прямоугольный массив, 130 псевдоним пространства имен, 287 типа, 286 пул потоков, 239 пустой оператор, 70 Р - Т рабочий стол, 312 разадресация, 351 развертывание, 9 разделитель, 26 размерность массива, 127 разрешение перегрузки, 152 разыменование, 351 раннее связывание, 177 распаковка, 36, 214 распространение исключения, 93 регистрация делегата, 225 регулярное выражение, 355 редактор, 8 режим доступа к файлу, 249 открытия файла, 249 совместного использования файла, 249 рекурсивный метод, 153 рекурсия, 153 ресурс, 274, 345 рефлексия, 279 решение, 15 родительское окно, 337 сборка, 9, 272 открытая, 273 частная, 273 432 Алфавитный указатель сборщик мусора, 169 свойство, 120 связывание позднее, 178 раннее, 177 сдвиг, 54 секция атрибутов, 283 сервер, 367 сериализация, 267 сигнатура метода, 108 сильное имя, 275 символ перевода строки, 23 подчеркивания, 23 структуры, 17 синтаксическая ошибка, 21 синтаксический анализ, 24 сконструированный тип, 304 следование, 87 словарь, 293 слово ключевое, 25 сложение, 52 событие, 232,312 обработка, 233 создание, 233 событийное управление, 312 совесть программиста, 45, 350 сообщение, 313 сортировка выбором, 306 составной оператор, 70 спецификатор делегата, 220 доступа, 101 индексатора, 157 класса, 101 перечисления, 215 события, 233 структуры, 212 формата, 146 список двунаправленный, 291 двусвязный, 291 кольцевой, 292 однонаправленный, 291 односвязный, 291 среда выполнения, 9 разработки, 8 ссылочный тип, 35 статический класс, 118 статический конструктор, 117 статический метод, 102 статический элемент, 102 стек, 31, 292 строка командная, 15 символов, 143 создание, 143 строковый литерал, 29 структура, 212 абстрактная, 294 структурное программирование, 69 ступенчатый массив, 132 счетчик цикла, 76 таблица виртуальных методов, 178 тег, 365 тело деструктора, 169 интерфейса, 189 класса, 100 метода, 19,106 операции, 161 перечисления, 215 структуры, 212 цикла, 75, 137 тип арифметический, 32 вещественный, 33 вложенный, 169 встроенный, 32 данных, 31 делегата, 220 значение, 35 метода, 19 неизменяемый, 143 обнуляемый, 309 перечисляемый, 215 сконструированный, 304 события, 233 ссылочный, 35 целый, 32 частичный, 308, 310 элемента сборки, 284 У - Я указатель, 348 умножение, 50 унарная операция, 162 унарный минус, 48 универсальный метод, 226 упаковка, 36, 214 управление, 312 управляющая последовательность, 29 условная операция, 56 файл, 248,260,271 физический каталог, 368 функциональная параметризация, 226 функция, 19 операция, 161 расстановки, 294 член, 19 хеш-код, 294 хеш-таблица, 293 хеш-функция, 294 хип, 31 целый тип, 32 цикл, 87 обработки сообщений, 313 перебора, 82 с параметром, 81 с постусловием, 79 с предусловием, 76 частичный тип, 308, 310 частная сборка, 273 чтение данных, 246 шаблон проектирования, 224 форматирования, 146 экземпляр, 13, 101 элемент статический, 102 управления, 325 язык албанский, 22 промежуточный, 273 |рограмми на языке высокого уровня Павловская Татьяна Александровна — профессор кафедры информатики и прикладной математики Санкт-Петербургского государственного университета информационных технологий, механики и оптики (СП6ГУИТМ0), автор нескольких учебников по программированию . одного из самых перспективных современных языко в программирования . Книга содержит описание версии С# 2.0 (200 5 года) и предназначена для студентов, изучающих язы к «с нуля», но будет полезна и опытны м программистам , желающи м быстро освоить новый язык . Многочисленные достоинства язык а С# позволяю т расценивать его ка к перспективну ю замену языко в Паскаль, BASIC и С++ при обучении программировани ю ка к студентов, т ак и школьников старших классов. На кажду ю ключеву ю тему в учебнике приводится по 20 однотипных вариантов заданий для лабораторных работ. Планируется интернет-поддержка^&щ ^ по адресу http://ips.ifmo.ru. Темы, рассмотренны е в книге : • основные понятия платформы .NET; •типы данных и конструкции язык а с#||р • принципы структурного и объектно-ориентированного программирования ; •динамические структуры данных и их реализация в библиотеке .NET; • классы, интерфейсы, делегаты, события, исключения, сборки, рефлексия типов; • классы-прототипы, ите ) ы, указатели, потоки • основы программирования под " Заказ книг: 197198, Санкт-Петербург, а/я 619 тел.: (812) 703-73-74, postbook@piter.com 61093, Харьков-93, а/я 9130 тел.: (057) 712-27-05, piter@kharkov.piter.co www.piter.com — вся ия о книгах и веб-магазин~~