int i = new int();// 0

**Инициализация многомерных массивов**

Многомерный массив можно инициализировать, заключив список инициализаторов

каждой размерности в собственный набор фигурных скобок. Например, вот каков формат

инициализации двумерного массива:

*тип*[,] *имя\_массива* = {

{*val*, *val*, *val*, ..., *val*}

{*val*, *val*, *val*, ..., *val*}

.

.

.

{*val*, *val*, *val*, ..., *val*)

};

Jagged

int[][] jagged = new int[3][];

jagged[0] = new int[4];

jagged[1] = new int[3];

jagged[2] = new int[5];

После выполнения этого фрагмента кода массив jagged выглядит так:

Можно также создать string-объект из массива типа char. Вот пример:

char[] charray = ('t', 'e', 's', 't');

string str = new string(charray);

static string Copy(

string *str*) Возвращает копию строки *str*

int CompareTo(string *str*) Возвращает отрицательное значение, если вызывающая

строка меньше строки *str*, положительное значение, если

вызывающая строка больше строки *str*, и нуль, если

сравниваемые строки равны

int IndexOf(string *str*) Выполняет в вызывающей строке поиск подстроки,

заданной параметром *str*. возвращает индекс первого

вхождения искомой подстроки или -1, если она не будет

обнаружена

int LastIndexOf(string *str*) Выполняет в вызывающей строке поиск подстроки,

заданной параметром *str*. Возвращает индекс последнего

вхождения искомой подстроки или -1, если она не будет

обнаружена

string ToLower() Возвращает строчную версию вызывающей строки

string ToUpper() Возвращает прописную версию вызывающей строки

class SubStr {

public static void Main() {

string orgstr = "C# упрощает работу со строками.";

// Создание подстроки.

string substr = orgstr.Substring(4, 14);

Console.WriteLine("orgstr: " + orgstr);

Console.WriteLine("substr: " + substr);

}

}

Управление доступом к членам класса достигается за счет использования четырех

*спецификаторов доступа:* public, private, protected и internal.

for (int i = 0; i < 10; i++)

{

Console.WriteLine((char)('A'+i));

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication3

{

class Stack1

{

int[] arr;

int index;

public Stack1(int size)

{

arr = new int[size];

index = 0;

}

public void Push(int num)

{

if (index >= arr.Length)

{

Console.WriteLine("Stack is full");

return;

}

arr[index] = num;

index++;

}

public int Pop()

{

if (index == 0)

{

Console.WriteLine("Стек пуст!");

return 0;

}

index--;

return arr[index];

}

public bool isFull()

{

return index == arr.Length;

}

public bool isEmpty()

{

return index == 0;

}

}

class Program

{

static void Main(string[] args)

{

Stack1 numeric = new Stack1(10);

for (int i = 0; !numeric.isFull(); i++)

{

numeric.Push(33 + i);

}

while (!numeric.isEmpty())

{

Console.WriteLine(numeric.Pop());

}

Console.ReadKey();

}

}

}

Используя ключевые слова ref и out, можно передать значение любого

нессылочного типа по ссылке. Тем самым мы позволим методу изменить аргумент,

используемый при вызове.

static void Main(string[] args)

{

int a = 20;

Azaza(ref a);

Console.WriteLine(a);

Console.ReadKey();

}

public static void Azaza(ref int i)

{

i = ++i;

}

Значение передавалось без предварительной инициализации

class Decompose {

/\* Метод разбивает число с плавающей точкой на

целую и дробную части. \*/

public int parts(double n, out double frac) {

int whole;

whole = (int) n;

frac = n - whole; // Передаем дробную часть

// посредством параметра frac.

return whole; // Возвращаем целую часть числа.

}

}

class Program

{

static void Main(string[] args)

{

Decompose ob = new Decompose();

int i; double f;

i = ob.parts(10.45, out f);

Console.WriteLine("Целая часть числа равна " + i);

Console.WriteLine("Дробная часть числа равна " + f);

Console.ReadKey();

}

Params

class Min {

public int minVal(params int[] nums) {

int m;

if(nums.Length == 0) {

Console.WriteLine("Ошибка: нет аргументов.");

return 0;

}

m = nums[0];

for(int i=1; i < nums.Length; i++)

if(nums[i] < m) m = nums[i];

return m;

}

}

class ParamsDemo {

public static void Main() {

Min ob = new Min();

int min;

int a = 10, b = 20;

// Вызываем метод с двумя значениями.

min = ob.minVal(a, b);

Console.WriteLine("Минимум равен " + min);

// call with 3 values

min = ob.minVal(a, b, -1);

Console.WriteLine("Минимум равен " + min);

// Вызываем метод с пятью значениями.

min = ob.minVal(18, 23, 3, 14, 25);

Console.WriteLine("Минимум равен " + min);

При перегрузки методов тип параметра которого int,а перегруженого double. Если мы туда передаем типы byte, short или float, то исполняется переобразование типов к заданым в параметрах.

Таким образом, при выполнении инструкции

ob.f(i);

вызывается метод f(int x), но при выполнении инструкции

ob.f(ref i);

вызывается метод f(ref int x).

Демонстрация вызова конструктора с помощью ссылки this.

using System;

class XYCoord {

public int x, y;

public XYCoord() : this(0, 0) {

Console.WriteLine("Внутри конструктора XYCoord()");

}

public XYCoord(XYCoord obj) : this(obj.x, obj.y) {

Console.WriteLine("Внутри конструктора XYCoord(obj)");

}

public XYCoord(int i, int j) {

Console.WriteLine("Внутри конструктора XYCoord(int, int)");

x = i;

y = j;

}

}

class OverloadConsDemo {

public static void Main() {

XYCoord t1 = new XYCoord();

XYCoord t2 = new XYCoord(8, 9);

XYCoord t3 = new XYCoord(t2);

Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);

Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);

Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);

}

Рекурсия

static int Main(string[] args)

{

string s = "System of a Down";

displayRev(s);

Console.ReadKey();

return 0;

}

public static void displayRev(string str)

{

if (str.Length > 0)

{

displayRev(str.Substring(1, str.Length - 1));

}

else return;

Console.Write(str[0]);

}

Внутри static-метода нельзя вызывать не статичиские методы, а также обращатся к не статическим переменным. Можна обращятся к статическим методам только через ссылку на обьект.

using System;

class CountInst {

static int count = 0;

// Инкрементируем счетчик при создании объекта.

public CountInst() {

count++;

}

// Декрементируем счетчик при разрушении объекта.

~CountInst() {

count--;

}

public static int getcount() {

return count;

}

}

class CountDemo {

public static void Main() {

CountInst ob;

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

ob = new CountInst();

Console.WriteLine("Текущее содержимое счетчика: " +

CountInst.getcount());

}

}

}

Результаты выполнения этой программы выглядят так:

Текущее содержимое счетчика: 1

Текущее содержимое счетчика: 2

Текущее содержимое счетчика: 3

Текущее содержимое счетчика: 4

Текущее содержимое счетчика: 5

Текущее содержимое счетчика: 6

Текущее содержимое счетчика: 7

Текущее содержимое счетчика: 8

Текущее содержимое счетчика: 9

Текущее содержимое счетчика: 10

Каждый раз, когда создается объект типа CountInst, static-поле count

инкрементируется. И каждый раз, когда объект типа CountInst разрушается, static-

поле count декрементируется. Таким образом, статическая переменная count всегда

содержит количество объектов, существующих в данный момент. Это возможно только

благодаря использованию статического поля. Переменная экземпляра не в состоянии

справиться с такой задачей, поскольку подсчет экземпляров класса связан с классом в

целом, а не с конкретным его экземпляром.

Перегрузка операторов

public static ThreeD operator +(ThreeD op1,

ThreeD op2)

{

ThreeD result = new ThreeD();

/\* Суммирование координат двух точек

и возврат результата. \*/

result.x = op1.x + op2.x; // Эти операторы выполняют

result.y = op1.y + op2.y; // целочисленное сложение.

result.z = op1.z + op2.z;

return result;

}

Implicit

// Пример использования implicit-оператора преобразования.

using System;

// Класс трехмерных координат.

class ThreeD {

int x, y, z; // 3-х-мерные координаты.

public ThreeD() {

x = y = z = 0;

}

public ThreeD(int i, int j, int k) {

x = i;

y = j;

z = k;

}

// Перегружаем бинарный оператор "+".

public static ThreeD operator +(ThreeD op1, ThreeD op2)

{

ThreeD result = new ThreeD();

result.x = op1.x + op2.x;

result.y = op1.y + op2.y;

result.z = op1.z + op2.z;

return result;

}

// Неявное преобразование из типа ThreeD в тип int.

public static implicit operator int(ThreeD op1) {

return op1.x \* op1.y \* op1.z;

}

// Отображаем координаты X, Y, Z.

public void show()

{

Console.WriteLine(x + ", " + y + ", " + z);

}

}

class ThreeDDemo {

public static void Main() {

ThreeD a = new ThreeD(1, 2, 3);

ThreeD b = new ThreeD(10, 10, 10);

ThreeD c = new ThreeD();

int i;

Console.Write("Координаты точки a: ");

a.show();

***248*** Часть I. Язык C#

Console.WriteLine();

Console.Write("Координаты точки b: ");

b.show();

Console.WriteLine();

c = a + b; // Суммируем координаты точек а и b.

Console.Write("Результат сложения а + b: ");

c.show();

Console.WriteLine();

i = a; // Преобразуем в значение типа int.

Console.WriteLine(

"Результат присваивания i = a: " + i);

Console.WriteLine();

i = a \* 2 - b; // Преобразуем в значение типа int.

Тож самое, только explicit – явное пеобразование i = (int) a; // Преобразуем объект в значение

Индексаторы

Главное назначение индексаторов — поддержать создание специализированных массивов, на которые налагается одно или несколько ограничений.

Создание одномерных индексаторов

Одномерный индексатор имеет следующий формат.

1. Индексаторы можна перегружать

тип\_элемента this[int индекс] {

// Аксессор считывания аннях.

Get {

// Возврат значения, заданного элементом индекс.

}

// Аксессор установки аннях.

Set {

// Установка значения, заданного

// элементом индекс.

}

}

public int this[int index] { // Это – get-аксессор.

Get {

if(ok(index)) {

errflag = false;

return a[index];

}

else {

errflag = true;

return 0;

}

}

// Это – set-аксессор.

Set {

if(ok(index)) {

a[index] = value;

errflag = false;

}

else

errflag = true;

}

}}

\*\*\*

Многомерный индексатор

public int this[int index1, int index2] {

// Это — get-аксессор.

get {

if(ok(index1, index2)) {

errflag = false;

return a[index1, index2];

} else {

errflag = true;

return 0;

}

}

Глава 10. Индексаторы и свойства 265

// Это — set-аксессор.

set {

if(ok(index1, index2)) {

a[index1, index2] = value;

errflag = false;

} else

errflag = true;

}

}

Свойство

1.свойство нельзя перегружать

2.нельзя передать свойство в качестве ref или out параметра

Свойство нужно чтобы ограничевать доступ к полям и давать пользователю той доступ который дал программист.

Важно понимать, что свойства не определяют область памяти. Следовательно,

свойство управляет доступом к полю, но самого поля не обеспечивает. Это поле должно

быть задано независимо от свойства.

Наследование

Если есть конструктор в дочернем классе и он инициализирует поля базового класса , то можно не создавать конструктор в базовом классе.

// Класс треугольников, производный от TwoDShape.

class Triangle : TwoDShape {

string style; // Закрытый член.

/\* Конструктор по умолчанию. Он автоматически вызывает

конструктор по умолчанию класса TwoDShape. \*/

public Triangle() {

style = "null";

}

// Конструктор, который принимает три аргумента.

public Triangle(string s, double w, double h) : base(w, h) {

style = s;

}

\*\*\*

**Наследование и сокрытие имен**

class A

{

public int i = 0;

}

// Создаем производный класс.

class B : A

{

new int i; // Этот член i скрывает член i класса A.

public B(int b)

{

i = b; // Член i в классе В.

}

public void show()

{

Console.WriteLine(

"Член i в производном классе: " + i);

}

}

\*\*\*

base.i = a; // Так можно обратиться к i класса А.

\*\*\*

public Triangle(string s, double w, double h) : base(w, h) { style = s;}

Сокрытие имен

class A {

public int i = 0;

}

// Создаем производный класс.

class В : A {

new int i; // Этот член i скрывает член i класса A.

\*\*\*

class A {

public int i = 0;

}

// Создаем производный класс.

class B : A {

new int i; // Эта переменная i скрывает i класса А.

public B(int a, int b) {

base.i = a; // Так можно обратиться к i класса А.

Нельзя присвоить разные типы, но если есть наследования то ОК.

class X {

int a;

public X(int i) {

a = i;

}

}

class Y {

int a;

public Y(int i) {

a = i;

}

}

class IncompatibleRef {

public static void Main() {

X x = new X(10);

X x2;

298 Часть I. Язык C#

Y y = new Y(5);

x2 = x; // OK, обе переменные имеют одинаковый тип.

x2 = y; // Ошибка, здесь переменные разного типа.

Checked и unchecked

byte a, b;

byte result;

a = 127;

b = 127;

try {

result = unchecked((byte) (a \* b) );

Console.WriteLine("Unchecked-результат: " + result);

result = checked((byte)(a \* b)); // Эта инструкция

// вызывает исключение.

Console.WriteLine("Checked-реэультат: " +

result); // Инструкция не будет

// выполнена.

}

catch(OverflowException exc) {

// Перехватываем исключение.

Console.WriteLine(exc);

}

FileMode.Append Добавляет выходные данные в конец файла

FileMode.Create Создает новый выходной файл. Существующий файл с таким

же именем будет разрушен

FileMode.CreateNew Создает новый выходной файл. Файл с таким же именем не

должен существовать

FileMode.Open Открывает существующий файл

FileMode.OpenOrCreate Открывает файл, если он существует, В противном случае

создает новый

FileMode.Truncate Открывает существующий файл, но усекает его длину до нуля.

public static void Main(string[] args) {

int i;

FileStream fin;

try {

fin = new FileStream(args[0], FileMode.Open);

}

catch(FileNotFoundException exc) {

Console.WriteLine(exc.Message);

return;

}

catch(IndexOutOfRangeException exc) {

Console.WriteLine(exc.Message +

"\nПрименение: ShowFile Файл");

return;

}

// Считываем байты до тех пор, пока не встретится EOF.

do {

try {

i = fin.ReadByte();

}

catch(Exception exc) {

Console.WriteLine(exc.Message);

return;

}

if(i != -1) Console.Write((char) i);

} while(i != -1);

fin.Close();

}

Вероятно, вы уже знаете, что при выполнении операции вывода в файл выводимые

данные зачастую не записываются немедленно на реальное физическое устройство, а

буферизируются операционной системой до тех пор, пока не накопится порция данных

достаточного размера, чтобы ее можно было всю сразу переписать на диск. Такой способ

выполнения записи данных на диск повышает эффективность системы. Например,

дисковые файлы организованы по секторам, которые могут иметь размер от 128 байт.

Данные, предназначенные для вывода, обычно буферизируются до тех пор, пока не

накопится такой их объем, который позволяет заполнить сразу весь сектор. Но если вы

хотите записать данные на физическое устройство вне зависимости от того, полон буфер

или нет, вызовите следующий метод Flush():

\*\*\*

str = str + "\r\n"; // Добавляем символ

// новой строки.

\*\*\*

BinaryWriter

dataOut = new

BinaryWriter(new FileStream("testdata",

FileMode.Create));

\*\*\*

FileStream f;

char ch;

try {

f = new FileStream("random.dat", FileMode.Create);

}

catch(IOException exc) {

Console.WriteLine(exc.Message);

return;

}

// Записываем в файл алфавит.

for(int i=0; i < 26; i++) {

try {

f.WriteByte( (byte) ('A'+i) );

}

catch(IOException exc) {

Console.WriteLine(exc.Message);

return;

}

}

try {

// Теперь считываем отдельные значения.

f.Seek(0, SeekOrigin.Begin); // Поиск первого байта.

ch = (char) f.ReadByte();

Console.WriteLine("Первое значение равно " + ch);

f.Seek(1, SeekOrigin.Begin); // Поиск второго байта.

\*\*\*

Класс MemoryStream — это реализация класса Stream, в которой для операций ввода-

вывода используются массивы байтов. Вот как выглядит конструктор этого класса:

MemoryStream(byte[] *buf*)

byte[] storage = new byte[255];

// Создаем поток с ориентацией на память.

MemoryStream memstrm = new MemoryStream(storage);

// Помещаем объект memstrm в оболочки StreamWriter

//и StreamReader.

StreamWriter memwtr = new StreamWriter(memstrm);

StreamReader memrdr = new StreamReader(memstrm);

// Записываем данные в память с помощью

// объекта memwtr.

for(int i=0; i < 10; i++)

Глава 14. Использование средств ввода-вывода 403

memwtr.WriteLine("'byte[" + i + "]: " + i);

// Ставим в конце точку.

memwtr.Write('.');

memwtr.Flush();

Console.WriteLine(

"Считываем данные прямо из массива storage: ");

// Отображаем напрямую содержимое памяти.

foreach(char ch in storage) {

if(ch == '.') break;

Console.Write(ch);

}

Console.WriteLine(

"\nСчитываем данные посредством объекта memrdr: ");

// Считываем данные из объекта memstrm, используя

// средство чтения потоков.

memstrm.Seek(0, SeekOrigin.Begin); // Установка

// указателя позиции в начало потока.

string str = memrdr.ReadLine();

while(str != null) {

str = memrdr.ReadLine();

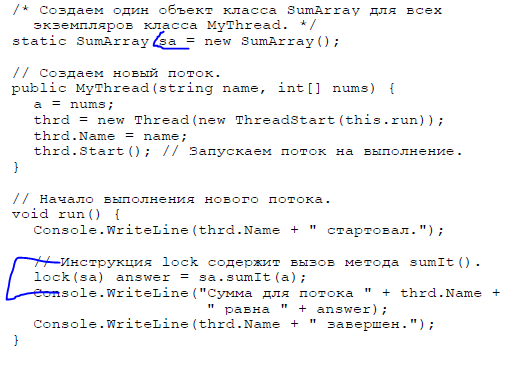
if(str.CompareTo(".") == 0) break;

Console.WriteLine(str);

}

\*\*\*

Блокировка потока в чужих классах



На static-методы накладывается ряд ограничений.

1. static-метод не имеет ссылки this,

2. static-метод может напрямую вызывать только другие static-методы. Он не может напрямую вызывать метод экземпляра своего класса. Дело в том, что методы экземпляров работают с конкретными экземплярами класса, чего не скажешь о static-методах.

3. static-метод должен получать прямой доступ только к static-данным. Он не может напрямую использовать переменные экземпляров, поскольку не работает с экземплярами класса.

\*\*\*

Опасный код — это код, который не плохо написан, а код, который не выполняется

под полным управлением системы Common Language Runtime (CLR). Как разъяснялось в

главе 1, язык C# обычно используется для создания управляемого кода. Однако можно

написать и "неуправляемый" код, который не подчиняется тем же средствам управления и

ограничениям, налагаемым на управляемый код.

\*\*\*

Unsafe

// Отмечаем метод Main() как "опасный".

unsafe public static void Main() {

\*\*\*

Fixed

При работе с указателями зачастую используется модификатор fixed. Он

предотвращает удаление управляемых переменных системой сбора мусора.

fixed(type\* *p* = &*var*) {

\*\*\*

Stackalloc

С помощью ключевого слова stackalloc можно использовать память стека. Это

можно делать только при инициализации локальных переменных. Распределение памяти

стека имеет следующий формат:*тип* \**p* - stackalloc *тип*[*размер*]

Здесь *p —* указатель, который получает адрес области памяти достаточно большого

размера, чтобы хранить *размер* объектов типа *тип*. Ключевое слово stackalloc можно

использовать только в контексте опасного (unsafe) кода.

Обычно память для объектов выделяется из кучи,

\*\*\*

Volatile

Модификатор volatile сообщает компилятору о том, что значение

соответствующего поля может быть изменено в программе неявным образом. Например,

поле, содержащее текущее значение системного времени, может обновляться операционной

системой автоматически. В этой ситуации содержимое такого поля изменяется без явного

выполнения инструкции присваивания. Причина внешнего изменения поля может быть

очень важной.

\*\*\*

Enum

*Перечисление* (enumeration) — это множество именованных целочисленных констант.

Ключевое слово enum объявляет перечислимый тип. Формат записи перечисления таков:

enum *имя* {*список\_перечисления*};

Здесь с помощью элемента *имя* указывается имя типа перечисления. Элемент

*список\_перечисления* представляет собой список идентификаторов, разделенных

запятыми.

enum apple { Jonathan, GoldenDel, RedDel, Winsap,

Cortland, McIntosh };

public static void Main() {

string[] color = {

"красный",

"желтый",

"красный",

"красный",

"красный",

"красно-зеленый",

"azaza"

};

apple i; // Объявляем переменную перечислимого типа.

// Используем переменную i для обхода всех

// членов перечисления.

for(i = apple.Jonathan; i <= apple.McIntosh; i++)

Console.WriteLine(i + " имеет значение " + (int)i);

Console.WriteLine();

// Используем перечисление для индексации массива.

for(i = apple.Jonathan; i <= apple.McIntosh; i++)

Console.WriteLine("Цвет сорта " + i + " - " +

color[(int)i]);

Console.ReadKey();

\*\*\*

По умолчанию перечисления используют тип int, но можно также создать

перечисление любого другого целочисленного типа, за исключением типа char. Чтобы

задать тип, отличный от int, укажите этот базовый тип после имени перечисления и

двоеточия. Например, следующая инструкция создает перечисление apple с базовым

типом byte.

enum apple : byte { Jonathan, GoldenDel, RedDel=10, Winsap,

Cortland, McIntosh };

\*\*\*

public enum action { старт, стоп, вперед, назад };

public void conveyor(action com) {

switch(com) {

case action.старт:

Console.WriteLine("Запуск конвейера.");

\*\*\*

*Пространство имен* определяет декларативную область, которая позволяет отдельно

хранить множества имен. По существу, имена, объявленные в одном пространстве имен, не

будут конфликтовать с такими же именами, объявленными в другом. Библиотека .NET

Framework (которая является C#-библиотекой) использует пространство имен System.

Поэтому в начало каждой программы мы включали следующую инструкцию:

Лямбда выражения

delegate int Incr(int v);

Incr incr = count => count + 2;

int x = -10;

while (x <= 0)

{

Console.Write(x + " ");

x = incr(x); // увеличить значение x на 2

}

Console.WriteLine("\n");

IsEven isEven = n => n % 2 == 0;

for (int i = 0; i <= 10; i++)

if (isEven(i)) Console.WriteLine(i + " четное.");

\*\*\*

delegate bool InRange(int lower, int upper, int v);

Следовательно, экземпляр делегата InRange может быть создан следующим образом.

InRange rangeOK = (low, high, val) => val >= low && val <= high;

\*\*\*

delegate int IntOp(int end);

class StatementLambdaDemo

{

static void Main()

{

// Блочное лямбда-выражение возвращает факториал

// передаваемого ему значения.

IntOp fact = n =>

{

int r = 1;

for (int i = 1; i <= n; i++)

r = i \* r;

return r;

};

\*\*\*

**Вторая форма директивы using**

Вторая форма директивы using позволяет определить еще одно имя (так называемый

*псевдоним*) типа данных или пространства имен. Эта форма приведена ниже:

using *псевдоним = имя;*

\*\*\*

**Интерфейсы**

public interface ISeries

{

int GetNext(); // возвратить следующее по порядку число

void Reset(); // перезапустить

void SetStart(int х); // задать начальное значение

}

**интерфейсные ссылки**

Primes primeOb = new Primes();

ISeries ob;

for(int i=0; i < 5; i++) {

ob = twoOb;

**Интерфейсные свойства**

public interface ISeries

{

// Интерфейсное свойство.

int Next

{

get; // возвратить следующее по порядку число

set; // установить следующее число

}

}

//реализация

public int Next {

get {

val += 2;

return val;

}

set {

val = value;

}

}

}

**Интерфейсные индексаторы**

public interface ISeries

{

// Интерфейсный индексатор.

int this[int index]

{

get; // возвратить указанное в ряду число

}

}

// Получить значение по индексу.

public int this[int index] {

get {

val = 0;

for(int i=0; i < index; i++)

val += 2;

return val;

}

}

**Наследование интерфейсов**

public interface IA {

void Meth1();

void Meth2();

}

// В базовый интерфейс включены методы Meth1() и Meth2().

// а в производный интерфейс добавлен еще один метод — Meth3().

public interface IB : IA {

void Meth3();

}

// В этом классе должны быть реализованы все методы интерфейсов IA и IB.

class MyClass : IB {

**Явные реализации**

interface IMyIF {

int MyMeth(int x);

}

class MyClass : IMyIF {

int IMyIF.MyMeth(int x) {

return x / 3;

}

**Выбор между интерфейсом и абстрактным классом**

Если какое-то понятие

можно описать с точки зрения функционального назначения, не уточняя конкретные

детали реализации, то следует использовать интерфейс. А если требуются некоторые

детали реализации, то данное понятие следует представить абстрактным классом

**Структуры**

Одни структуры не могут наследовать другие структуры и классы или служить

в качестве базовых для других структур и классов. (Разумеется, структуры, как и все

остальные типы данных в С#, наследуют класс object.) Тем не менее в структуре можно

реализовать один или несколько интерфейсов, которые указываются после имени

структуры списком через запятую. Как и у классов, у каждой структуры имеются свои

члены: методы, поля, индексаторы, свойства, операторные методы и события. В структурах

допускается также определять конструкторы, но не деструкторы. В то же время

для структуры нельзя определить конструктор, используемый по умолчанию (т.е. конструктор

без параметров). Дело в том, что конструктор, вызываемый по умолчанию,

определяется для всех структур автоматически и не подлежит изменению. Такой конструктор

инициализирует поля структуры значениями, задаваемыми по умолчанию.

А поскольку структуры не поддерживают наследование, то их члены нельзя указывать

как abstract, virtual или protected.

Объект структуры может быть создан с помощью оператора new таким же образом,

как и объект класса, но в этом нет особой необходимости. Ведь когда используется

оператор new, то вызывается конструктор, используемый по умолчанию.

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается

одно из главных отличий структуры от класса

// Определить структуру.

struct Book

{

public string Author;

public string Title;

public int Copyright;

public Book(string a, string t, int c)

{

Author = a;

Title = t;

Copyright = c;

}

}

Book book1 = new Book("Герберт Шилдт",

"Полный справочник пo C# 4.0",

2010) ; // вызов явно заданного конструктора

Book book2 = new Book(); // вызов конструктора по умолчанию

Book bоок3; // конструктор не вызывается

**О назначении структур**

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена

структура, если она обладает более скромными возможностями, чем класс? Ответ на

этот вопрос заключается в повышении эффективности и производительности программ.

Структуры относятся к типам значений, и поэтому ими можно оперировать

непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не

требуется переменная ссылочного типа, а это означает в ряде случаев существенную

экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению

производительности, столь характерному для обращения к объекту класса.

Обработка исключительных

Ситуаций

ArrayTypeMismatchException Тип сохраняемого значения несовместим с типом массива

DivideByZeroException Попытка деления на нуль

IndexOutOfRangeException Индекс оказался за границами массива

InvalidCastException Неверно выполнено динамическое приведение типов

OutOfMemoryException Недостаточно свободной памяти для дальнейшего выполнения

программы. Это исключение может быть,

например, сгенерировано, если для создания объекта

с помощью оператора new не хватает памяти

OverflowException Произошло арифметическое переполнение

NullReferenceException Попытка использовать пустую ссылку, т.е. ссылку, котораяне указывает ни на один из объектов

**Применение нескольких операторов catch**

catch (DivideByZeroException) {

Console.WriteLine("Делить на нуль нельзя!");

}

catch (IndexOutOfRangeException) {

Console.WriteLine("Подходящий элемент не найден.");

}

**Перехват всех исключений**

catch { // "Универсальный" перехват.

Console.WriteLine("Возникла некоторая исключительная ситуация.");

}

**Генерирование исключений вручную**

try {

Console.WriteLine("До генерирования исключения.");

throw new DivideByZeroException();

}

catch (DivideByZeroException) {

Console.WriteLine("Исключение перехвачено.");

**Подробное рассмотрение класса Exception**

Свойство Message содержит символьную строку, описывающую

характер ошибки; свойство StackTrace — строку с вызовами стека, приведшими к исключительной

ситуации, а свойство TargetSite получает объект, обозначающий метод,

сгенерировавший исключение.

\*\*\*

**Получить гипертекст страницы (HtppWebRequest, HttpWebResponse)**

int ch;

// Сначала создать объект запроса типа WebRequest по указанному URI.

HttpWebRequest req = (HttpWebRequest)

WebRequest.Create("https://vk.com/reisereisereise");

// Затем отправить сформированный запрос и получить на него ответ.

HttpWebResponse resp = (HttpWebResponse)

req.GetResponse();

// Получить из ответа поток ввода.

Stream istrm = resp.GetResponseStream();

/\* А теперь прочитать и отобразить гипертекстовое содержимое,

полученное по указанному URI. Это содержимое выводится на экран

отдельными порциями по 400 символов. После каждой такой порции

следует нажать клавишу <ENTER>, чтобы вывести на экран

следующую порцию из 400 символов. \*/

for(int i=1; ; i++) {

ch = istrm.ReadByte();

if(ch == -1) break;

Console.Write((char)ch);

if((i%400)==0) {

Console.Write("\nНажмите клавишу <Enter>.");

Console.ReadLine();

}

}

// Закрыть ответный поток. При этом закрывается также поток ввода istrm.

resp.Close();

\*\*\*

**Обработка исключений**

static void Main()

{

int ch;

try

{

//Код

resp.Close();

}

catch (WebException exc)

{

Console.WriteLine("Сетевая ошибка: " + exc.Message +

"\nКод состояния: " + exc.Status);

}

catch (ProtocolViolationException exc)

{

Console.WriteLine("Протокольная ошибка: " + exc.Message);

}

catch (UriFormatException exc)

{

Console.WriteLine("Ошибка формата URI: " + exc.Message);

}

catch (NotSupportedException exc)

{

Console.WriteLine("Неизвестный протокол: " + exc.Message);

}

catch (IOException exc)

{

Console.WriteLine("Ошибка ввода-вывода: " + exc.Message);

}

catch (System.Security.SecurityException exc)

{

Console.WriteLine("Исключение в связи с нарушением безопасности: " +

exc.Message);

}

catch (InvalidOperationException exc)

{

Console.WriteLine("Недопустимая операция: " + exc.Message);

}

}

\*\*\*

**URI**

Uri sample = new

Uri("https://vk.com/pikabu");

Console.WriteLine("Хост: " + sample.Host);

Console.WriteLine("Порт: " + sample.Port);

Console.WriteLine("Протокол: " + sample.Scheme);

Console.WriteLine("Локальный путь: " + sample.LocalPath);

Console.WriteLine("Запрос: " + sample.Query);

Console.WriteLine("Путь и запрос: " + sample.PathAndQuery);

\*\*\*

**Headers**

// Создать объект запроса типа WebRequest по указанному URI.

HttpWebRequest req = (HttpWebRequest)

WebRequest.Create("https://vk.com/pikabu");

// Отправить сформированный запрос и получить на него ответ.

HttpWebResponse resp = (HttpWebResponse)

req.GetResponse();

// Получить список имен.

string[] names = resp.Headers.AllKeys;

// Отобразить пары "имя-значение" из заголовка.

Console.WriteLine("{0,-20}{1}", "Имя", "Значение");

foreach (string n in names)

{

Console.Write("{0,-20}", n);

foreach (string v in resp.Headers.GetValues(n))

Console.WriteLine(v);

}

\*\*\*

**WebClient**

WebClient user = new WebClient();

string uri = "http://www.McGraw-Hill.com";

string fname = "data.txt";

try

{

Console.WriteLine("Загрузка данных по адресу " +

uri + " в файл " + fname);

user.DownloadFile(uri, fname);

}

catch (WebException exc)

{

Console.WriteLine(exc);

}

Console.WriteLine("Загрузка завершена.");

**\*\*\***

**Hastable**

// Создать хеш-таблицу.

Hashtable ht = new Hashtable();

ICollection с = ht.Keys;

// Использовать ключи для получения значений.

foreach(string str in с)

Console.WriteLine(str + ": " + ht[str]);

}