МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Ю.А. ПЛОТОНЕНКО

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические рекомендации по выполнению лабораторных работ

ТЕМА 1. ВВЕДЕНИЕ В C#. СИСТЕМА ТИПОВ ЯЗЫКА C#. ВЫРАЖЕНИЯ И ОПЕРАТОРЫ. УПРАВЛЕНИЕ ДЕЙСТВИЯМИ С ДАННЫМИ. МАССИВЫ.

Лабораторная работа № 1 Основные конструкции языка C#.

Цель работы – ознакомиться с основными конструкциями языка C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Правила синтаксиса.

При написании программы придерживаются синтаксических правил таких как:

{ } операторные скобки объединяют несколько операторов в один блок,
; конец оператора,
, разделитель при перечислении констант, переменных,
( ) содержат параметры функций или операторов.

2.2 Комментарии.

C# представляет несколько механизмов комментирования кода:

- построчное //
- многострочное /* */
- комментарии, которые позволяют автоматически генерировать документацию в XML – формате. Эти комментарии начинаются с символов ///, за которыми следуют специальные тэги.

2.3 Класс Console. Консольный ввод–вывод.

При обсуждении процедур ввода–вывода следует иметь в виду одно важное обстоятельство. Независимо от типа выводимого значения, в конечном результате выводится, СИМВОЛЬНОЕ ПРЕДСТАВЛЕНИЕ значения. Это становится очевидным при выводе информации в окно приложения, что и обеспечивают по умолчанию методы Console.Write и Console.WriteLine.

Консольный вывод.

Методы WriteLine и Write позволяют отображать информацию на консоль.

```
// Вывод строки.
Console.WriteLine("The Captain is on the board!");
```

```
//Символьное представление целочисленного значения.
Console.WriteLine(314);
//Символьное представление значения типа float.
Console.WriteLine(3.14);
```

Могут принимать различное число параметров. Однако:

```
Console.WriteLine("qwerty",314,3.14);
```

Результатом выполнения выражения вызова функции будет строка:

qwerty

Значения второго и третьего параметров не выводятся. Первый строковый параметр выражений вызова функций Write и WriteLine используется как управляющий шаблон для представления выводимой информации. Значения следующих за строковым параметром выражений будут выводиться в окно представления лишь в том случае, если первый параметр–строка будет явно указывать места расположения выводимых значений, соответствующих этим параметрам, маркерами, которые в самом простом случае представляют собой заключённые в фигурные скобки целочисленные литералы.

При этом способ указания места состоит в следующем:

- CLR индексирует все параметры метода WriteLine, следующие за первым параметром–строкой. При этом второй по порядку параметр получает индекс 0, следующий за ним – 1, и т.д. до конца списка параметров.
- В произвольных местах параметра–шаблона размещаются маркеры выводимых значений.
- Значение маркера должно соответствовать индексу параметра, значение которого должно выводиться на экран.
- Значение целочисленного литерала маркера не должно превышать максимального значения индекса параметра.

Таким образом, оператор

```
Console.WriteLine("The sum of {0} and {1} is {2}",314,3.14,314+3.14);
```

обеспечивает вывод следующей строки:

The sum of 314 and 3.14 is 317.3

В последнем операнде выражения вызова WriteLine при вычислении значения выражения используется неявное приведение типа. При вычислении значения суммы операнд типа int без потери значения приводится к типу float. Безопасные преобразования типов проводятся автоматически.

Несмотря на явную абсурдность выводимых утверждений, операторы

```
Console.WriteLine("The sum of {0} and {1} is {0}",314,3.14,314+3.14);
Console.WriteLine("The sum of {2} and {1} is {0}",314,3.14,314+3.14);
```

также отработают вполне корректно.

Форматирование строки вывода.

Помимо индекса параметра маркер выводимого значения может содержать дополнительную информацию относительно формата представления выводимой информации.

Выводимые значения преобразуются к символьному представлению, которое, в свою очередь, при выводе в окно приложения может быть дополнительно преобразовано в соответствии с предопределённым "сценарием преобразования".

Вся необходимая для дополнительного форматирования информация размещается непосредственно в маркерах и отделяется запятой от индекса маркера.

Кроме того, в маркерах вывода могут также размещаться дополнительные строки форматирования (FormatString). При этом маркер приобретает достаточно сложную структуру, внешний вид которой в общем случае можно представить следующим образом (M – значение индекса, N – область позиционирования):

{M,N:FormatString}

Сама же строка форматирования аналогична ранее рассмотренной строке–параметру метода ToString и является комбинацией предопределённых символов форматирования и дополнительных целочисленных значений (см. табл.1).

Таблица 1. Предопределенные символы форматирования.

Символ форматирования	Описание
C	Отображение значения как валюты с использованием принятого по соглашению символа
D	Отображение значения как decimal integer
E	Отображение значения в соответствии с научной нотацией
F	Отображение значения как fixed Point
G	Display the number as a fixed–Point or integer, depending on which is the most compact
N	Применение запятой для разделения порядков
X	Отображение значения в шестнадцатеричной нотации

Непосредственно за символом форматирования может быть расположена целочисленная ограничительная константа, которая в зависимости от типа выводимого значения может определять количество выводимых знаков после точки, либо общее количество выводимых символов. При этом дробная часть действительных значений округляется, либо дополняется нулями справа. При выводе целочисленных значений ограничительная константа игнорируется, если количество выводимых

символов превышает её значение. В противном случае выводимое значение слева дополняется нулями.

Следующие примеры иллюстрируют варианты применения маркеров со строками форматирования:

```
Console.WriteLine("Integer fotmating – {0:D3},{1:D5}",12345, 12);
Console.WriteLine("Currency formatting – {0:C},{1:C5}", 99.9, 999.9);
Console.WriteLine("Exponential formatting – {0:E}", 1234.5);
Console.WriteLine("Fixed Point formatting – {0:F3}", 1234.56789);
Console.WriteLine("General formatting – {0:G}", 1234.56789);
Console.WriteLine("Number formatting – {0:N}", 1234567.89);
//Integers only!
Console.WriteLine("Hexadecimal formatting – {0:X7}",12345);
```

В результате выполнения этих операторов в окно консольного приложения будут выведены следующие строки

Integer fotmating – 12345,00012
Currency formatting – $99.90,$999.90000
Exponential formatting – 1.234500E+003
Fixed Point formatting – 1234.568
General formatting – 1234.56789
Number formatting – 1,234,567.89
Hexadecimal formatting – 0003039

Консольный ввод.

Метод Read читает по одному символу до тех пор, пока все символы в потоке не закончатся, и возвращает код символа либо -1, если символов больше нет в потоке.

Метод Readln читает строку символов.

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

Результатом считывания всегда будет строка. Поэтому, для получения нужного типа данных необходимо использовать преобразование.

2.4 Преобразования и класс Convert.

Класс Convert, определенный в пространстве имен System, играет важную роль, обеспечивая необходимые преобразования между различными типами. Внутри арифметического типа можно использовать явный (скобочный) способ приведения к нужному типу. Но таким способом нельзя привести, например, переменную типа string к типу int, Оператор присваивания: ux = (int)s1; приведет к ошибке периода компиляции. Здесь необходим вызов метода ToInt32 класса Convert.

Методы класса Convert поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит 15 статических методов вида To<Type> (ToBoolean(),…ToUInt64()), где Type может принимать значения от Boolean до UInt64 для всех встроенных типов. Единственным исключением является тип object, – метода ToObject нет по понятным причинам, поскольку для всех типов существует неявное преобразование к типу object.

Также есть возможность преобразования к системному типу DateTime, который хотя и не является встроенным типом языка C#, но допустим в программах, как и любой другой системный тип.

```
//System type: DateTime
System.DateTime dat = Convert.ToDateTime("15.03.2003");
Console.WriteLine("Date = {0}", dat);
```

Результатом вывода будет строка:

Date = 15.03.2003 0:00:00

Все методы To<Type> класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#.

Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования значений объектов и другие методы.

Преобразование типа с использованием класса Convert создает значение нового типа, эквивалентное значению старого типа, однако при этом не обязательно сохраняется идентичность (или точные значения) двух объектов.

Различаются преобразования:

- расширяющие;
- сужающие.

Расширяющее преобразование.

Значение одного типа преобразуется к значению другого типа, которое имеет такой же или больший размер. Например, значение, представленное в виде 32–разрядного целого числа со знаком, может быть преобразовано в 64–разрядное целое число со знаком.

Расширяющее преобразование считается безопасным, поскольку исходная информация при таком преобразовании не искажается (см. табл. 2.1).

Таблица 2.1. Перечень безопасных преобразований к типу.

Тип	Возможно безопасное преобразование к типу…
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

Некоторые расширяющие преобразования типа могут привести к потере точности. Таблица 2.2 описывает варианты преобразований, которые иногда приводят к потере информации.

Таблица 2.2. Перечень преобразований, при которых возможна потеря точности.

Тип	Возможна потеря точности при преобразовании к типу…
Int32	Single
UInt32	Single
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

Сужающее преобразование

Значение одного типа преобразуется к значению другого типа, которое имеет меньший размер (из 64–разрядного в 32–разрядное).

Такое преобразование потенциально опасно потерей значения.

Сужающие преобразования могут приводить к потере информации. Если тип, к которому осуществляется преобразование, не может правильно передать значение источника, то результат преобразования оказывается равен константе PositiveInfinity или NegativeInfinity.

При этом значение PositiveInfinity интерпретируется как результат деления положительного числа на ноль, а значение NegativeInfinity интерпретируется как результат деления отрицательного числа на ноль.

Если сужающее преобразование обеспечивается методами класса System.Convert, то потеря информации сопровождается генерацией исключения, которые рассмотрим позже (см. табл. 3).

Таблица 3. Варианты сужающих преобразований.

Тип	Возможна потеря значения и генерация исключения при преобразовании в:
Byte	Sbyte
SByte	Byte, UInt16, UInt32, UInt64
Int16	Byte, SByte, UInt16
UInt16	Byte, SByte, Int16
Int32	Byte, SByte, Int16, UInt16, UInt32
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32, UInt32, UInt64
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

Метод Parse().

Этот метод поддерживают все предопределенные типы значений. Используется для извлечения из строки значения нужного типа.

Например: `int i = Int32.Parse(Console.ReadLine());`

2.5 Класс System.Environment. Окружение.

Класс System.Environment позволяет узнать информацию о текущем окружении и платформе. Основные его свойства и методы сведены в таблицу 4.

Таблица 4. Свойства и методы класса System.Environment.

Название	Описание
Свойства	
CommandLine	возвращает командную строку, с помощью которой была запущена программа
CurrentDirectory	содержит текущую директорию, с которой был запущен процесс
ExitCode	не зависимо от типа возврата Main (int или void), процесс будет возвращать значение
MashineName	позволяет узнать имя машины, на которой запущен процесс
NewLine	возвращает строку, которая содержит символ перехода на новую строку в текущей системе
OSVersion	возвращает специальный объект, который содержит описание текущей платформы и версию
SystemDirectory	возвращает полный путь к системной директории

UserDomainName	возвращает имя домена, с которым ассоциируется текущий пользователь системы
UserName	возвращает имя пользователя системы
Методы	
Exit()	завершает текущий процесс, возвращая системе код выхода, который определен свойством ExitCode
GetCommandLineArgs()	позволяет вернуть массив строк, которые содержат параметры командной строки, передаваемые программе
GetEnvironmentVariale()	позволяет вернуть значение переменной окружения
GetEnvironmentVariales()	позволяет вернуть список всех переменных окружения и их значения
GetLogicalDrives()	позволяет вернуть список всех логических дисков, которые имеются на машине

2.6 Система типов

C# является языком со строгой типизацией данных. Таким образом, IL не допускает никаких действий, которые дают в результате неопределенные типы данных.

.NET Framework предоставляет общую систему типов CTS (Common Type System), использование которой позволяет разрабатывать приложение на любом из языков, поддерживающих эту среду, и при этом не заботиться о несовместимости типов при повторном использовании разработанных компонентов.

Система типов поддерживает две категории типов, каждая из которых разделена на подкатегории:

- типы значений (типы–значения),
- ссылочные типы (типы–ссылки).

Схема типов представлена на рисунке 1.

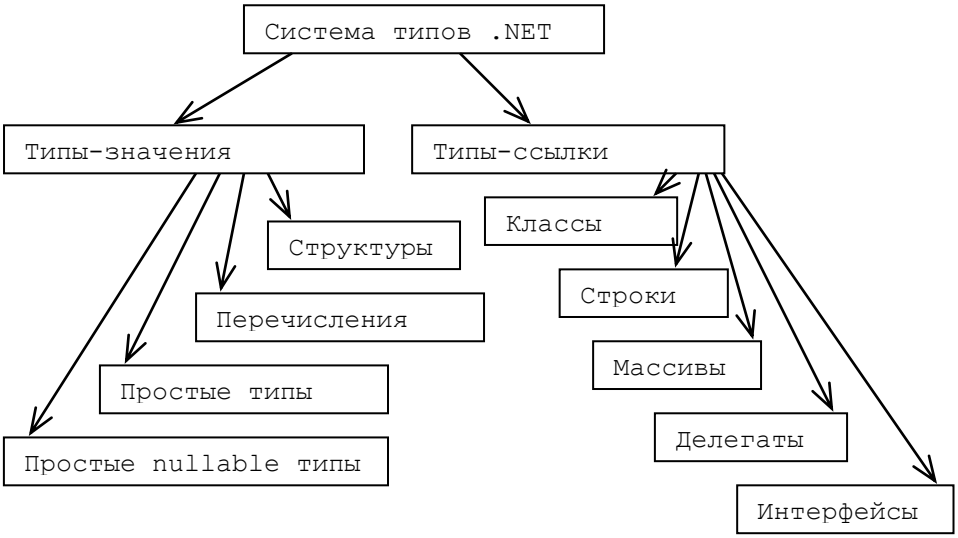


Рис.1. Система типов .Net

Ниже представлены основные отличия ссылочных типов и типов–значений.

Таблица 5. Отличия типов-значения от типов-ссылок.

	Типы–значения value-types	Типы–ссылки reference-types
Объект представлен	непосредственно значением	ссылкой в куче
Объект располагается	в стеке	в куче
Значение по умолчанию	0, false, '\0', null	ссылка имеет значение null
При выполнении операции присваивания копируется	значение	ссылка

Для типов-ссылок необходимо явно выделять место в памяти, используя метод New().

В C# объявление любой структуры и класса основывается на объявлении предопределённого класса Object (наследует класс Object). Следствием этого является возможность вызова от имени объектов–представителей любой структуры или класса унаследованных от класса object методов. В частности, метода ToString. Этот метод возвращает строковое (значение типа string) представление объекта.

Все типы (типы-значения и типы-ссылки), за исключением простых типов-значений и пары предопределённых ссылочных типов (string и object), должны определяться (если уже не были ранее специально определены) программистами в рамках объявлений. Подлежащие объявлению типы называются производными типами.

Простые (элементарные) типы

Это типы, имя и основные свойства которых известны компилятору. Относительно элементарных типов компилятору не требуется никакой дополнительной информации. Свойства и функциональность этих типов известны.

Используемые в .NET языки программирования основываются на общей системе типов. Между именами простых типов в C# и именами типов, объявленных в Framework Class Library, существует взаимно однозначное соответствие (см. табл. 6).

Таблица 6. Соответствие простых типов C# FCL-типу.

Тип в C#	Соответствует FCL–типу	Описание
sbyte	*System.SByte*	Целый. 8–разрядное со знаком. Диапазон значений: 128 … 127
byte	*System.Byte*	Целый. 8–разрядное без знака. Диапазон значений: 0 … 255
short	*System.Int16*	Целый. 16–разрядное со знаком. Диапазон значений: –32768 … 32767
ushort	*System.UInt16*	Целый. 16–разрядное без знака. Диапазон значений: 0 … 65535
int	*System.Int32*	Целый. 32–разрядное со знаком. Диапазон значений: –2147483648 … 2147483647
uint	*System.UInt32*	Целый. 32–разрядное без знака. Диапазон значений: –0 … 4294967295
long	*System.Int64*	Целый. 64–разрядное со знаком. Диапазон значений: –9223372036854775808 … 9223372036854775807
ulong	*System.UInt64*	Целый. 64–разрядное без знака. Диапазон значений: 0 … 18446744073709551615
char	*System.Char*	16 (!) разрядный символ UNICODE.
float	*System.Single*	Плавающий. 32 разряда. Стандарт IEEE.
double	*System.Double*	Плавающий. 64 разряда. Стандарт IEEE.
decimal	*System.Decimal*	128–разрядное значение повышенной точности с плавающей точкой.
bool	*System.Boolean*	Значение true или false.

Перечисления

Это тип-значение, состоящий из набора поименованных констант.

Формат определения перечисления:

```
enum Имя [:базовый класс]
{
   Имя0 =  Значение0,…,ИмяN = ЗначениеN
}
```

По умолчанию нумерация начинается с нуля и увеличивается на 1.

В качестве примера опишем перечисление, указывающее день недели:

```
enum DayOfWeek { Monday = 1, Tuesday = 2, Wednesday = 3,
                 Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7 };
//Переменная xVal инициализируется значением перечисления.
int xVal = DayOfWeek.Friday;
```

Перечисление является классом, а это означает, что в распоряжении программиста оказываются методы сравнения значений перечисления, методы преобразования значений перечисления в строковое представление, методы перевода строкового представления значения в перечисление, а также (судя по документации) средства для создания объектов–представителей класса перечисления.

Список членов класса перечисления сведен в таблицу 7.

Таблица 7. Соответствие простых типов C# FCL-типу.

Название метода	Описание
Открытые методы	
CompareTo()	Сравнивает этот экземпляр с заданным объектом и возвращает сведения об их относительных значениях.
Equals()	Переопределен. Возвращает значение, показывающее, равен ли данный экземпляр заданному объекту.
Format()	Статический. Преобразует указанное значение заданного перечисляемого типа в эквивалентное строчное представление в соответствии с заданным форматом.
GetHashCode()	Переопределен. Возвращает хеш–код для этого экземпляра.
GetName()	Статический. Выводит имя константы в указанном перечислении, имеющем заданное значение.
GetNames()	Статический. Выводит массив имен

	константа в указанном перечислении.
GetType() (унаследовано от Object)	Возвращает Type текущего экземпляра.
GetTypeCode()	Возвращает базовый тип TypeCode для этого экземпляра.
GetUnderlyingType()	Статический. Возвращает базовый тип указанного перечисления.
GetValues()	Статический. Выводит массив значений констант в указанном перечислении.
IsDefined()	Статический. Возвращает признак наличия константы с указанным значением в заданном перечислении.
Parse()	Статический. Перегружен. Преобразует строковое представление имени или числового значения одной или нескольких перечисляемых констант в эквивалентный перечисляемый объект.
ToObject()	Статический. Перегружен. Возвращает экземпляр указанного типа перечисления, равный заданному значению.
ToString()	Перегружен. Переопределен. Преобразует значение этого экземпляра в эквивалентное ему строковое представление.
Защищенные методы	
Finalize() (унаследовано от Object)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone() (унаследовано от Object)	Создает неполную копию текущего Object.

Структуры

Во многом схожи с классами. Основное их отличие в том, что структуры не являются ссылочным типом. Дополнительно, на структуры накладываются следующие ограничения:

- структура не может иметь деструктор;

- структура не может иметь конструктор без параметров;
- структура не может использоваться как базовый тип.

В качестве примера укажем структуру, описывающую точку.

```
struct Point
{
   public int x, y;
   public Point(int p1, int p2)
   { x = p1; y = p2; }
}
```

2.7 Литералы. Представление значений.

В программах на языках высокого уровня (C# в том числе) литералами называют последовательности входящих в алфавит языка программирования символов, обеспечивающих явное представление значений, которые используются для обозначения начальных значений в объявлении членов классов, переменных и констант в методах класса.

Арифметические литералы

Арифметические литералы кодируют значения различных (арифметических) типов. Тип арифметического литерала определяется следующими интуитивно понятными внешними признаками:

- *стандартным внешним видом.* Значение целочисленного типа обычно кодируется интуитивно понятной последовательностью символов '1',…, '9', '0'. Значение плавающего типа также предполагает стандартный вид (точка–разделитель между целой и дробной частью, либо научная или экспоненциальная нотация – 1.2500E+052). Шестнадцатеричное представление целочисленного значения кодируется шестнадцатеричным литералом, состоящим из символов '0',…,'9', а также 'a',…, 'f', либо 'A',…,'F' с суффиксом '0x',
- *значением – соответствующем типу.* 32768 никак не может быть значением типа short,
- дополнительным суффиксом. Суффиксы l, L соответствуют типу long; ul, UL – unsigned long; f, F – float; d, D – decimal. Значения типа double кодируются без префикса.

Логические литералы

К логическим литералам относятся следующие последовательности символов: true и false. Больше логических литералов в C# нет.

Символьные литералы

Представляют собой заключённые в одинарные кавычки вводимые с клавиатуры одиночные символы: 'X', 'p', 'Q', '7',

В C# char — это 16-разрядный тип без знака, который позволяет представлять значения в диапазоне 0—65 535. Стандартный 8-разрядный набор символов ASCII составляет лишь подмножество Unicode с диапазоном 0—127. Таким образом, ASCII-символы — это действительные C# - символы.

Например, чтобы присвоить значение буквы X переменной ch, нужно выполнить следующие инструкции:

```
char ch;
ch = 'X';
```

Чтобы вывести char-значение, хранимое в переменной ch:

```
Console.WriteLine("Это ch: " + ch) ;
```

Хотя тип char определяется в C# как целочисленный, его нельзя свободно смешивать с целыми числами, т.к. автоматического преобразования целочисленных значений в значения типа char не существует.

Например, следующий фрагмент программы содержит ошибку.

```
//ошибка, это работать не будет.
char ch;
ch = 10;
```

Поскольку 10 — целое число, оно не может быть автоматически преобразовано в значение типа char. При попытке скомпилировать этот код вы получите сообщение об ошибке. Ниже в этой главе мы рассмотрим "обходной путь", позволяющий обойти это ограничение.

Символьные управляющие последовательности.

Эта категория литералов (см. табл. 8) используется для создания дополнительных эффектов (звонок), простого форматирования выводимой информации и кодирования символов при выводе и сравнении (в выражениях сравнения). Заключаются в одинарные кавычки.

Таблица 8. Перечень основных символов управляющих последовательностей.

Управляющая последовательность	Описание
\a	Предупреждение (звонок)
\b	Возврат на одну позицию
\f	Переход на новую страницу
\n	Переход на новую строку
\r	Возврат каретки

\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Ноль
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Строковые литералы

Это последовательность символов и символьных управляющих последовательностей, заключённых в двойные кавычки.

…"c:\\My Documents\\sample.txt"…

Строковый литерал, интерпретируемый компилятором в таком виде, котором он записан. Управляющие последовательности воспринимаются строго как последовательности символов.

Символы в строковом литерале будут трактоваться так, как они есть, если строковый литерал предварить символом @. Другими словами, не будут интерпретироваться как управляющие последовательности:

…@"c:\My Documents\sample.txt"…

Оба примера имеют одно и то же значение:

c:\My Documents\sample.txt

Строковые литералы являются литералами типа string.

2.8 Переменные.

CTS позволяет работать со значениями двух видов:

- переменные, которые непосредственно хранят данные,
- переменные, которые содержат только ссылку на данные (память в этом случае выделяется с использованием оператора new).

Объявление и инициализация.

Выполнение оператора объявления переменной типа-значения в методе класса приводит к созданию в памяти объекта соответствующего типа, возможно проинициализированного определённым значением. Это значение может быть задано в виде литерала соответствующего типа

Например,

```
//определения переменных типов-значений:
int a;
```

```
System.Int32 a;

//определение и инициализация переменных типа значения:
int a = 0;
int a = new int();
System.Int32 a = 0;
System.Int32 a = new System.Int32();
```

Следует учитывать: CLR не допускает использования в выражениях неинициализированных локальных переменных, таких, которые объявлены в теле метода.

```
int a;          // Объявление a.
int b;          // Объявление b.
b = 10;         // Инициализация b.
a=b+b;          // Инициализация a.
```

Область видимости.

При определении классов и методов используются фигурные скобки. Содержимое между этими скобками называется блоком. Блоки могут быть вложены друг в друга. Любая переменная, объявленная внутри конкретного блока, называется локальной переменной для этого блока, и она не существует вне, то есть ее нельзя использовать в других блоках, если они не являются вложенными.

```
{
    {
        double a=3;
        . . .
        {
            Console.Writeln(a);        //правильно
        }
    }
    Console.Writeln(a);                //ошибка компиляции
}
```

Не допускается объявление переменной во вложенном блоке с тем же именем, что и переменная, объявленная в основном блоке:

```
    {
        double a=3;
        . . .
        {
            double a;                  //ошибка компиляции
        }
```

}

2.9 Константы

Объявляются с дополнительным спецификатором const. Требуют непосредственной инициализации

```
//константа инициализируется литералом 3.14.
const float Pi = 3.14;
```

2.10 Операции и выражения

Для каждого определённого в C# типа существует собственный набор операций, определённых на множестве значений этого типа.

Эти операции определяют диапазон возможных преобразований, которые могут быть осуществлены над элементами множеств значений типа. Несмотря на специфику разных типов, в C# существует общее основание для классификации соответствующих множеств операций. Каждая операция является членом определённого подмножества операций и имеет собственное графическое представление (см. табл. 9).

Общие характеристики используемых в C# операций представлены ниже.

Таблица 9. Общие характеристики операций, используемых в C#.

Категории операций	**Операции**
Arithmetic	+ – * / %
Логические (boolean и побитовые)	& | ^ ! ~ && ||
Строковые (конкатенаторы)	+
Increment, decrement	++ —
Сдвига	>> <<
Сравнения	== != < > <= >=
Присвоения	= += –= *= /= %= &= |= ^= <<= >>=
Member access	.
Индексации	[]
Cast (приведение типа)	()
Conditional (трёхоперандная)	?:
Delegate concatenation and removal	+ –
Создания объекта	new()
Type information	is sizeof typeof
Overflow exception control (управление исключениями)	checked unchecked
Indirection and Address (неуправляемый код)	* –> [] &

При создании программы на стадии выполнения учитываются следующие обстоятельства:

- приоритет операций (см. табл. 10),
- типы операндов и приведение типов.

Таблица 10. Приоритет операций.

Приоритет	Операции
1	() [] . (постфикс)++ (постфикс)— new sizeof typeof checked unchecked
2	! ~ (имя типа) +(унарный) –(унарный) ++(префикс) —(префикс)
3	* / %
4	+ –
5	<< >>
6	< > <= => is as
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= += –= *= /= %= &= |= ^= <<= >>=

Контроль за переполнением. Checked и unchecked.

Причиной некорректных результатов выполнения арифметических операций является особенность представления значений арифметических типов.

Арифметические типы имеют ограниченные размеры. Поэтому любая арифметическая операция может привести к переполнению. По умолчанию в C# переполнение, возникающее при выполнении операций, никак не контролируется. Возможный неверный результат вычисления остаётся всего лишь результатом выполнения операции.

Механизм контроля за переполнением, возникающим при выполнении арифметических операций, обеспечивается ключевыми словами checked (включить контроль за переполнением) и unchecked (отключить контроль за переполнением), которые используются в составе выражений. CLR выполняет контроль переполнения и, в случае обнаружения такового, генерирует исключение OverflowException. Конструкции управления контролем за переполнением имеют две формы:

- операторную, которая обеспечивает контроль над выполнением одного выражения:

```
:::::
short x = 32767;
short y = 32767;
short z = 0;

try
{
 z = checked(x + unchecked(x+y));
}
catch (System.OverflowException e)
{
 Console.Writeline("Переполнение при выполнении сложения");
}

return z;
:::::
```

При этом контролируемое выражение может быть произвольной формы и сложности и может содержать другие вхождения как контролируемых, так и неконтролируемых выражений,

- блочную, которая обеспечивает контроль над выполнением операций в блоке операторов:

```
:::::
short x = 32767;
short y = 32767;
short z = 0, w = 0;

try
{
unchecked
{
 w = x+y;
}

checked
{
 z = x+w;
}

}
catch (System.OverflowException e)
{
```

```
 Console.Writeline("Переполнение при выполнении сложения");
}

return z;
```

Операция is

Позволяет проверить, совместим ли (относится к данному типу либо к типу, унаследованному от данного) объект с определенным типом. Результат использования логического типа.

```
        int i = 10;
if (i is object)
{
   Console.WriteLine(" i is object");
      }
```

Операция as

Применяется для явного преобразования типа ссылочных переменных. Если конвертируемый тип совместим с указанным, преобразование осуществляется успешно. Иначе операция возвращает значение NULL.

```
     object o1 = "my str";
object o2 = 5;
string s1 = o1 as string; // s1="my str"
string s2 = o2 as string; // s2= NULL
```

Операция as позволяет выполнить безопасное преобразование типа за один шаг, без необходимости первоначальной проверки его на совместимость с помощью операции is до собственно преобразования.

Особенности выполнения арифметических операций

Особенности выполнения операций над целочисленными операндами и операндами с плавающей точкой связаны с особенностями выполнения арифметических операций и с ограниченной точностью переменных типа float и double.

Представление величин:

float – 7 значащих цифр.

double – 16 значащих цифр.

1000000*100000==1000000000000, но максимально допустимое положительное значение для типа System.Int32 составляет 2147483647. В результате переполнения получается неверный результат –727379968.

Ограниченная точность значений типа System.Single проявляется при присвоении значений переменной типа System.Double. Приводимый ниже простой программный код иллюстрирует некоторые особенности арифметики .NET.

```
using System;

class Class1
{
const double epsilon = 0.00001D;
static void Main(string[] args)
{
int valI = 1000000, resI;
resI = (valI*valI)/valI;

// –727379968/1000000 == –727
Console.WriteLine
("The result of action (1000000*1000000/1000000) is {0}", resI);

float valF00 = 0.2F, resF;
double valD00 = 0.2D, resD;

// Тест на количество значащих цифр для значений типа double и float.
resD = 12345678901234567890; Console.WriteLine(">>>>>
{0:F10}",resD);
resF = (float)resD; Console.WriteLine(">>>>> {0:F10}",resF);
resD = (double)(valF00 + valF00); // 0.400000005960464
if (resD == 0.4D) Console.WriteLine("Yes! {0}",resD);
else Console.WriteLine("No! {0}",resD);

resF = valF00*5.0F;
resD = valD00*5.0D;

resF = (float)valD00*5.0F;
resD = valF00*5.0D; //1.0000000149011612

if (resD == 1.0D) Console.WriteLine("Yes! {0}",resD);
else          Console.WriteLine("No! {0}",resD);

resF = valF00*5.0F;
resD = valF00*5.0F; //1.0000000149011612
```

```
if (resD.Equals(1.0D))  Console.WriteLine("Yes! {0}",resD);
else                    Console.WriteLine("No! {0}",resD);

if (Math.Abs(resD – 1.0D) < epsilon)
Console.WriteLine("Yes! {0:F7}, {1:F7}",resD – 1.0D, epsilon);
else
Console.WriteLine("No! {0:F7}, {1:F7}",resD – 1.0D, epsilon);
}
}
```

В результате выполнения программы выводится такой результат:

```
The result of action (1000000*1000000/1000000) is -727
>>>>> 123456789012346000000,0000000000
>>>>> 123456800000000000000,0000000000
No! 0,400000005960464
No! 1,00000001490116
No! 1,00000001490116
Yes! 0,0000000, 0,0000100
```

Особенности арифметики с плавающей точкой

- Если переменной типа float присвоить величину x из интервала
- –1.5E–45 < x < 1.5E–45 (x != 0),
- результатом операции окажется положительный (x > 0) или отрицательный (x < 0) нуль (+0, –0),
- Если переменной типа double присвоить величину x из интервала
- –5E–324 < x < 5E–324 (x != 0), результатом операции окажется положительный
- (x > 0) или отрицательный (x < 0) нуль (+0, –0),
- Если переменной типа float присвоить величину x, которая
- –3.4E+38 > x или x < 3.4E+38, результатом операции окажется положительная
- (x > 0) или отрицательная (x < 0) бесконечность (+Infinity, – Infinity),
- Если переменной типа double присвоить величину x, для которой

- –1.7E+308 > x или x < 1.7E+308, результатом операции окажется положительная (x > 0) или отрицательная (x < 0) бесконечность (+Infinity, – Infinity),
- Выполнение операции деления над значениями типами с плавающей точкой (0.0/0.0) даёт NaN (Not a Number).

2.11 Управляющие операторы

Управляющие операторы применяются в рамках методов. Они определяют последовательность выполнения операторов в программе и являются основным средством реализации алгоритмов.

Различаются следующие категории управляющих операторов:

- операторы выбора. Вводятся ключевыми словами if, if … else …, switch,
- итеративные операторы. Вводятся ключевыми словами while, do … while, for, foreach,
- операторы перехода (в рамках методов). Вводятся ключевыми словами goto, break, continue.

if, if … else …

После ключевого слова if располагается взятое в круглые скобки условное выражение (логического типа), следом за которым располагается оператор (блок операторов) произвольной сложности.

Далее в операторе if … else … после ключевого слова else размещается ещё один оператор.

В силу того, что в C# отсутствуют предопределённые алгоритмы преобразования значений к булевскому типу, в условное выражение должно быть выражением типа bool – переменной, константой, либо выражением на основе операций сравнения и логических операций.

```
        if (…)
         {

         }
        else
         {
}
```

Оператор if часто сам в свою очередь является условным оператором произвольной сложности.

```
if (…) if (…)
 {

 }
else
 {

 }
else
 {

 }
```

Первый Else всегда относится к ближайшему if.

```
if (true) { int XXX = 125; }
if (true) { int XXX = 125; } else { int ZZZ = 10; }
```

switch

Представляет собой оператор, организующий множественный выбор.

Описание:

```
switch (выражение)
{
case значение1: оператор1; break;
::::::::::
case значениеN: операторN; break;
default: операторN+1; break;
}
```

Каждое значение Case в обязательном порядке ЗАВЕРШАЕТСЯ оператором break.

```
char val;
:::::
switch (val)
{
case ‘A’: Console.WriteLine(" A");break;
case ‘B’: Console.WriteLine(" B");break;

default: Console.WriteLine(" unknown");break;
}
```

while

```
while (выражение)
```

{
операторы
}

Правило выполнения этого итеративного опера состоит в следующем: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Невозможно построить оператор WHILE на основе одиночного оператора объявления. Оператор

```
while (true) int XXX = 0;
```

С самого первого момента своего существования (ещё до начала трансляции!) сопровождается предупреждением:

Embedded statement cannot be a declaration or labeled statement.

do … while

```
do
{
операторы
} while (выражение)
```

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения оператора.

for

```
for (Инициализация;УсловиеПродолжения;изменение
переменных)
Оператор
```

Инициализация, УсловиеПродолжения, изменение переменных в заголовке оператора цикла for могут быть пустыми. Однако наличие пары символов ';' в заголовке цикла for обязательно.

Список выражений представляет собой разделённую запятыми последовательность выражений.

Следует иметь в виду, что оператор объявления также строится на основе списка выражений (выражений объявления), состоящих из спецификаторов типа, имён и, возможно, инициализаторов. Этот список завершается точкой с запятой, что позволяет рассматривать список выражений инициализации как самостоятельный оператор в составе оператора цикла for. При этом область видимости имён переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла. Это значит, что переменные,

объявленные в операторе инициализации данного оператора цикла НЕ МОГУТ быть использованы непосредственно после оператора до конца блока, содержащего этот оператор. А следующие друг за другом в рамках общего блока операторы МОГУТ содержать в заголовках одни и те же выражения инициализации.

foreach

foreach (тип имя__переменной in коллекция) инструкци

имя__переменной обозначает переменную которая представляет элемент коллекции.

коллекция объект, представляющий массив или коллекцию.

Этим оператором обеспечивается повторение множества операторов, составляющих тело цикла, для каждого элемента массива или коллекции. После перебора ВСЕХ элементов массива или коллекции и применения множества операторов для каждого элемента массива или коллекции, управление передаётся следующему за ОператорFOREACH оператору (разумеется, если таковые имеются).

Область видимости имён переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла.

```
// Объявили и определили массив
int[] array = new int[10];
// Для каждого элемента массива надо сделать…
foreach (int i in array) {/*:::::*/};
```

Специализированный оператор, приспособленный для работы с массивами и коллекциями. Обеспечивает повторение множества (единичного оператора или блока операторов) операторов для КАЖДОГО элемента массива или коллекции.

Конструкция экзотическая и негибкая.

Предполагает выполнение примитивных последовательностей действий над массивами и коллекциями (начальная инициализация или просмотр ФИКСИРОВАННОГО количества элементов). Действия, связанные с изменениями размеров и содержимого коллекций в рамках этого оператора могут привести к непредсказуемым результатам.

goto, break, continue

Объявляется метка (правильнее, оператор с меткой). Оператор может быть пустым. Метка – идентификатор отделяется от оператора двоеточием. В качестве дополнительного разделителя могут быть использованы пробелы,

символы табуляции и перехода на новую строку. Метка, как и любое другое имя, подчиняется правилам областей видимости. Она видна в теле метода только в одном направлении: из внутренних (вложенных) блоков. Поэтому оператор перехода

goto ИмяПомеченногоОператора;

позволяет ВЫХОДИТЬ из блоков, но не входить в них.

Обычно об этом операторе обычно говорится много нехороших слов как о главном разрушителе структурированного программного кода и его описание сопровождается рекомендациями к его НЕИСПОЛЬЗОВАНИЮ.

Операторы

break;

и

continue;

используются как вспомогательные средства управления в операторах цикла.

2.12 Массив.

Массив – множество однотипных элементов. Любой массив является производным от класса System.Array.

В отличие от других языков программирования, при объявлении массива в C# нельзя указать его размер, т.к. при объявлении не создается сам массив, а только ссылка на будущий массив. Поэтому после объявления необходима инициализация массива.

Объявление массивов

Одномерные массивы

Объявление одномерного массива выглядит следующим образом:

<тип>[] < имя массива >;

Заметьте, в отличие от языка C++ квадратные скобки приписаны не к имени переменной, а к типу. Они являются неотъемлемой частью определения класса, так что запись T[] следует понимать как класс **одномерный массив с элементами типа** T.

В данном случае речь идет об отложенной инициализации. **При объявлении с отложенной инициализацией сам массив не создается, а создается только ссылка на массив,** имеющая неопределенное значение Null. Поэтому **пока массив не будет реально создан и его элементы инициализированы, использовать его в вычислениях нельзя**.

int[] a, b, c; // пример объявления трех массивов с отложенной

инициализацией

Чаще всего, при объявлении массива используется имя с инициализацией. И опять таки, как и в случае простых переменных, могут быть два варианта инициализации:

а) инициализация является явной и задается константным массивом:

double[] x= {5.5, 6.6, 7.7};

Синтаксически, элементы константного массива следует заключать в фигурные скобки.

б) массив создается и инициализируется (выделяется место в памяти с указанным числом элементов массива) массив из 5 элементов типа int:

int[] d= new int[5];

При создании массива можно указывать число элементов, которое хранит переменная, инициализируемая в результате работы программы:

```
string s = Console.ReadLine();
int size = int.Parse(s);
double[] rr = new double[size];
```

Доступ к отдельному элементу массива осуществляется посредством индекса. Ин-
декс описывает позицию элемента внутри массива. Первый элемент имеет нулевой индекс.

Выход за границы массивов в C# расценивается как динамическая ошибка. Будет сгенерирована исключительная ситуация типа IndexOutOfRangeException, и программа прекратит выполнение.

Многомерные массивы

Объявление *многомерного массива* в общем случае:

<тип>[, … ,] <имя массива>;

Число запятых, увеличенное на единицу, и задает размерность массива. Можно лишь отметить, что хотя явная инициализация с использованием многомерных константных массивов возможна, но применяется редко из-за громоздкости такой структуры.

Простейший многомерный массив — **двумерный**. В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй — столбец.

Чтобы объявить двумерный массив целочисленных значений размером

10x20 с именем table, достаточно записать следующее:

int [,] table = new int[10, 20];

Обратите особое внимание на то, что значения размерностей отделяются запятой. Синтаксис первой части этого объявления означает, что создается ссылочная переменная двумерного массива. Для реального выделения памяти для этого массива с помощью оператора new используется более конкретный синтаксис:

int[10, 20]

Чтобы получить доступ к элементу двумерного массива, необходимо указать оба индекса, разделив их запятой. Например, чтобы присвоить число 10 элементу массива table, позиция которого определяется координатами 3 и 5, можно использовать следующую инструкцию:

table [3 , 5] = 10;

Пример использования массивов.

Рассмотрим классическую задачу умножения прямоугольных матриц. Нам понадобится три динамических массива для представления матриц и три процедуры, одна из которых будет заполнять исходные матрицы случайными числами, другая выполнять умножение матриц, третья – печатать сами матрицы. Вот тестовый пример:

```
public void TestMultiMatr()
    {
        int n1, m1, n2, m2,n3, m3;
            Arrs.GetSizes("MatrA",out n1,out m1);
            Arrs.GetSizes("MatrB",out n2,out m2);
            Arrs.GetSizes("MatrC",out n3,out m3);
            int[,]MatrA = new int[n1,m1], MatrB = new int[n2,m2];
            int[,]MatrC = new int[n3,m3];
            Arrs.CreateTwoDimAr(MatrA);
Arrs.CreateTwoDimAr(MatrB);
            Arrs.MultMatr(MatrA, MatrB, MatrC);
            Arrs.PrintAr2("MatrA",MatrA);
Arrs.PrintAr2("MatrB",MatrB);
            Arrs.PrintAr2("MatrC",MatrC);
      }//TestMultiMatr
```

Три матрицы MatrA, MatrB и MatrC имеют произвольные размеры, выясняемые в диалоге с пользователем, и использование для их описания динамических массивов представляется совершенно естественным. Метод CreateTwoDimAr заполняет случайными числами элементы матрицы, переданной ему в качестве аргумента, метод PrintAr2 – выводит матрицу на

печать. Я не буду приводить их код, похожий на код их одномерных аналогов.

Метод MultMatr выполняет умножение прямоугольных матриц. Текст этого метода:

```
public void MultMatr(int[,]A, int[,]B, int[,]C)
    {
      if (A.GetLength(1) != B.GetLength(0))
                Console.WriteLine("MultMatr: ошибка размерности!");
      else
        for(int i = 0; i < A.GetLength(0); i++)
          for(int j = 0; j < B.GetLength(1); j++)
          {
            int s=0;
            for(int k = 0; k < A.GetLength(1); k++)
              s+= A[i,k]*B[k,j];
                        C[i,j] = s;
          }
    }//MultMatr
```

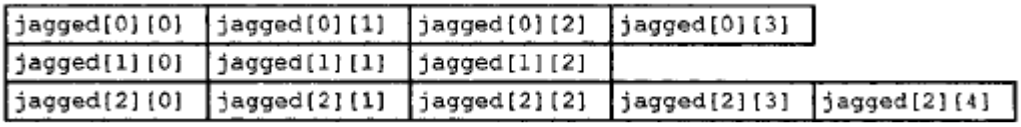
Массивы массивов

Еще одним видом массивов C# являются массивы массивов, называемые также ступенчатыми массивами (jagged arrays). Такой массив массивов можно рассматривать как одномерный массив, элементы которого являются массивами, элементы которых, в свою очередь снова могут быть массивами и так может продолжаться до некоторого уровня вложенности.

Следующий фрагмент программы при объявлении массива jagged выделяет память для его первой размерности, а память для его второй размерности выделяется "вручную".

```
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:



В каких ситуациях может возникать необходимость в таких структурах данных? Такие массивы могут применяться для представления деревьев, у которых узлы могут иметь произвольное число потомков. Это может быть, например, генеалогическое дерево. Вершины первого уровня – Fathers, представляющие отцов, могут задаваться одномерным массивом, так что Fathers[i] – это i-й отец. Вершины второго уровня представляются массивом массивов – Children, так что Children[i] – это массив детей i-го отца, а Children[i][j] – это j-й ребенок i-го отца. Для представления внуков понадобится третий уровень, так что GrandChildren [i][j][k] будет представлять к-го внука j-го ребенка i-го отца.

Есть некоторые особенности в объявлении и инициализации таких массивов. Если при объявлении типа многомерных массивов для указания размерности использовались запятые, то для изрезанных массивов используется более ясная символика – совокупности пар квадратных скобок, например int[][] задает массив, элементы которого одномерные массивы элементов типа int.

Сложнее с созданием самих массивов и их инициализацией. Здесь нельзя вызвать конструктор new int[3][5], поскольку он не задает изрезанный массив. Фактически нужно вызывать конструктор для каждого массива на самом нижнем уровне. В этом и состоит сложность объявления таких массивов. Начну с формального примера:

```
//массив массивов - формальный пример
        //объявление и инициализация
        int[][] jagger = new int[3][]
                {
                   new int[] {5,7,9,11},
                   new int[] {2,8},
                   new int[] {6,12,4}
                };
```

Массив jagger имеет всего два уровня. Можно считать, что у него три элемента, каждый из которых является массивом. Для каждого такого массива необходимо вызвать конструктор new, чтобы создать внутренний массив. В данном примере элементы внутренних массивов получают значение, будучи явно инициализированы константными массивами.

Встроенный сервис по обслуживанию массивов

Массивам соответствует свой класс **System.Array**, который позволяет с ними работать. Этот класс характеризуется специальным набором свойств и методов для создания, управления, поиска, и сортировки, элементов массива представленных в таблице 4.1.

Для использования свойств операнд принимает вид:

Имя_массива. Свойство;

Для использования методов оператор:

Array. Метод(параметры);

либо **Имя_массива. Метод(параметры);**

Таблица 11. Набор свойств и методов для работы с массивами.

Название	Описание
Свойство Length	Возвращает целое число представляющее общее количество элементов во всех измерениях массива.
Свойство Rank	Возвращает целое число представляющее количество измерений (размерность) массива.
Метод Array CreateInstance()	Статический метод (один из вариантов), создаёт массив элементов заданного типа и определённой размерности.
Метод GetLength ()	Возвращает количество элементов в указной размерности массива.
Метод SetValue()	Присваивает элементу массива значение, представленное первым параметром (один из вариантов).
Метод GetValue()	Извлекает значение из двумерного массива по индексам (один из вариантов).
Метод Sort()	Позволяет сортировать одномерный массив, **массив передается как параметр**
Метод Clear()	Устанавливает элементы массива в 0 для массивов, содержащих значения, null – содержащих ссылки, false – логического типа
Метод Clone()	Для создания копии массива, при этом копируются лишь ссылки, сами объекты копироваться не будут

2.13 Символы и строки.

Обработка текстовой информации является, вероятно, одной из самых распространённых задач в программировании, и C# предоставляет для ее решения широкий набор средств: отдельные символы, массивы символов, изменяемые и неизменяемые строки и регулярные выражения.

Символы

Символьный тип char предназначен для хранения символов в кодировке Unicode. Символьный тип относится к встроенным типам данных C# и соответствует стандартному классу Char библиотеки .NET из пространства имен System. В этом классе определены статические методы, позволяющие задать вид и категорию символа, а также преобразовать символ в верхний или нижний регистр и в число. Основные методы приведены в табл. 12.

Таблица 12. Основные методы класса System.Char.

Название	Описание
GetNumericValue	Возвращает числовое значение символа, если он является цифрой, и -1 в противном случае
GetUnicodeCategory	Возвращает категорию Unicode-символа. Все Unicode-символы разделены на категории, например, десятичные цифры (Decimal-DigitNumber), римские цифры (LetterNumber), разделители строк (LineSeparator), буквы в нижнем регистре (LowercaseLetter) и т. д.
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичным или шестнадцатеричным)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsUpper	Возвращает true, если символ записан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является пробельным (пробел, перевод строки и возврат каретки)
Parse	Преобразует строку в символ (строка должна состоять из одного символа)
ToLower	Преобразует символ в нижний регистр
ToUpper	Преобразует символ в верхний регистр
MaxValue, MinValue	Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления)

В примере продемонстрировано использование этих методов.

```
using System;
```

```
namespace ConsoleApplication
{
 class Classl
 {
     static void Main()
  {
     try
     {
        char b = 'B', c = '\x63', d = '\u0032';              //1
        Console.WriteLine("{0} {1} {2}", b, c, d);
        Console.WriteLine("{0} {1} {2}",
            char.ToLower(b), char.ToUpper(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} ",
                  a, (int)a);
           if (char.IsLetter(a)) Console.WriteLine("Буква");
           if (char.IsUpper(a)) Console.WriteLine("Верхий рег.");
           if (char.IsLower(a)) Console.WriteLine("Нижний рег.");
           if (char.IsControl(a)) Console.WriteLine("Управляющий");
           if (char.IsNumber(a)) Console.WriteLine("Число");
           if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");
        } 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 выполняются вывод и преобразование символов.

В цикле 2 анализируется вводимый с клавиатуры символ. Можно вводить и управляющие символы, используя сочетание клавиши Ctrl с латинскими буквами. При вводе использован метод Parse, преобразующий строку, которая должна содержать единственный символ, в символ типа char. Поскольку вводится строка, ввод каждого символа следует завершать нажатием клавиши Enter. Цикл выполняется, пока пользователь не введет

символ q.

Вывод символа сопровождается его кодом в десятичном виде. Для вывода кода используется явное преобразование к целому типу. Явное преобразование из символов в строки и обратно в C# не существует, неявным же образом любой объект, в том числе и символ, может быть преобразован в строку, например:

```
string s = 'к' + 'о' + 'т'; // результат - строка "кот"
```

При вводе и преобразовании могут возникать исключительные ситуации, например, если пользователь введет пустую строку. Для «мягкого» завершения программы предусмотрена обработка исключений.

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. 11. Применение этих методов позволяет эффективно решать некоторые задачи. Пример использования массива символов:

```
using System;
namespace ConsoleApplication
{
 class Classl
 {
    static void Main()
     {
      char[] a = { 'm', 'a', 's', 's', 'i', 'v' }; // 1
      char[] b = "а роза упала на лапу азора".ToCharArray(); // 2
      PrintArray("Исходный массив a:", a);
      int pos = Array.IndexOf(a, 'm');
      a[pos] = 'M';
      PrintArray("Измененный массив a:", a);
      PrintArray("Исходный массив b:", b);
      Array.Reverse(b);
      PrintArray("Измененный массив b:", b);
     }
    public static void PrintArray(string header, Array a )
     {
      Console.WriteLine(header);
      foreach ( object x in a ) Console.Write(x);
      Console.WriteLine("\n");
     }
  }
}
```

Результат работы программы:

```
Исходный массив a:
massiv
```

Измененный массив a:
Massiv
Исходный массив b
а роза упала на лапу азора
Измененный массив b:
ароза упал ан алапу азор а

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 C#. 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 string(' ',20); // конструктор создает строку из 20 пробелов
char[] a = { 'O' , 'O' , 'O' }; // массив для инициализации строки
string v = new string( a ); // создание из массива символов
```

Для 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 имеют одинаковое количество символов и совпадают посимвольно.

Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к так называемым неизменяемым типам данных Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые ≪старые≫ копии автоматически удаляются сборщиком мусора.

В классе System.String предусмотрено множество методов, полей и свойств, позволяющих выполнять со строками практически любые действия. Основные элементы класса приведены в табл. 13.

Таблица 13. Основные элементы класса System.String

Название	Описание
Compare	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.
CompareOrdinal	Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки
CompareTo	Сравнение текущего экземпляра строки с другой строкой
Concat	Конкатенация строк. Метод допускает сцепление произвольного числа строк
Copy	Создание копии строки
Empty	Пустая строка (только для чтения)
Format	Форматирование в соответствии с заданными спецификаторами формата
IndexOf, IndexOfAny, LastlndexOf, LastIndexOfAny	Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора
Insert	Вставка подстроки в заданную позицию
Intern, Islnterned	Возвращает ссылку на строку, если такая уже существует. Если строки нет, Intern добавляет строку во внутренний пул, IsInterned возвращает null
Join	Слияние массива строк в единую строку. Между элементами массива вставляются разделители
Length	Длина строки (количество символов)
PadLeft, PadRight	Выравнивание строки по левому или правому краю путем вставки нужного числа пробелов в начале или в конце строки
Remove	Удаление подстроки из заданной позиции
Replace	Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом
Split	Разделяет строку на элементы, используя заданные разделители. Результаты помещаются в массив строк
StartsWith, EndsWith	Возвращает true или false в зависимости оттого, начинается или заканчивается строка заданной подстрокой
Substring	Выделение подстроки, начиная с заданной позиции
ToCharArray	Преобразование строки в массив символов
ToLower, ToUlpper	Преобразование символов строки к нижнему или верхнему регистру
Tim, TrimStart, TrimEnd	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 для работы со строкой типа string:

```
using System;
namespace ConsoleApplication
{
 class Classl
 {
    static void Main()
     {
        string s = "прекрасная королева Изольда";
      Console.WriteLine(s);
      string sub = s.Substring(3).Remove(12, 2); //1
      Console.WriteLine(sub);
      string [ ] mas = s . Split(' ');          //2
      string joined = string.Join("! ", mas);
      Console.WriteLine(joined);
      Console.WriteLine("Введите строку");
      string x = Console.ReadLine();           // 3
      Console.WriteLine("Вы ввели строку " + x);
      double a = 12.234;
      int b = 29;
     Console.WriteLine("a = {0,6:C} b = {1,2:X}", a, b); //4
     Console.WriteLine("a = {0,6:0.##} b={1,5:0.#' руб.'}", a, b );//5
     Console.WriteLine("a = {0:F3} b={1:D3}", a, b );//6
    }
  }
}
```

```
Результат работы программы:
прекрасная королева Изольда
красная корова Изольда
прекрасная! королева! Изольда
Введите строку
не хочу!
Вы ввели строку не хочу!
a = 12,23р. b = 1D
a =  12,23 b = 29 руб.
a = 12,234 b = 029
```

В операторе 1 выполняются два последовательных вызова методов: метод Substring возвращает подстроку строки s, которая содержит символы исходной строки, начиная с четвертого символа. Для этой подстроки вызывается метод Remove, удаляющий из нее два символа, начиная с 12-го. Результат работы метода присваивается переменной sub.

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) объединяет элементы массива mas в одну строку, вставляя между каждой парой слов строку "! ".

В операторе 5 используются так называемые пользовательские шаблоны форматирования. 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.

Пользовательский шаблон может также содержать текст, который в общем случае заключается в апострофы.

3 Контрольные вопросы

1. Что понимается под термином «.NET Framework»?
2. Зависят ли приложения, разрабатываемые в .NET, от платформы?
3. Возможно ли создание гетерогенных приложений в среде .NET?
4. Что означает аббревиатура «CLR»?
5. Является ли среда CLR многоязычной?
6. Приведите обобщенный синтаксис объявления переменной на языке C#.
7. Приведите обобщенный синтаксис инициализации переменной на языке C#.
8. Какая дисциплина (вариант контроля) типов принята в языке C#?
9. Каковы основные категории типов в языке C#? Перечислите пять простых типов языка C#.
10. Что понимается под областью видимости переменной в языке C#?
11. Как обозначается область видимости переменной в языке C#?
12. Как соотносится время жизни переменной и область видимости?
13. Приведите синтаксис условного оператора в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
14. Приведите синтаксис оператора выбора в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
15. Что понимается под термином «пространство имен»?
16. В чем состоит назначение пространств имен в языке C#?
17. Благодаря какому механизму удается избежать коллизий имен в языке C#?
18. Какое пространство имен использует системная библиотека .NET Framework?
19. Какое пространство имен использует системная библиотека C#?
20. В чем состоит назначение директивы using?

21. Какой символ используется для указания полного имени объекта в языке C#?
22. Приведите синтаксис директивы using в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
23. Приведите синтаксис описания пространства имен в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
24. Опишите структуру программы на языке C#.
25. Какие группы символов входят в алфавит языка C#.
26. Что такое управляющие последовательности, и каким образом они задаются?
27. Как задаются идентификаторы?
28. Перечислите ключевые слова языка C#.
29. Как определить константу?
30. Опишите возможности ввода-вывода данных.
31. Опишите известные вам манипуляторы ввода-вывода.
32. Опишите оператор выбора case.
33. Опишите условный оператор if.
34. Какие операции отношения вы знаете.
35. Каков приоритет логических выражений.
36. Какие виды операторов цикла существуют.
37. Опишите оператор цикла с предусловием.
38. Опишите оператор цикла с постусловием.
39. Опишите оператор цикла с параметром.
40. Перечислите операторы передачи управления.
41. Какое назначение оператора break.
42. Какое назначение оператора continue.
43. Какое назначение оператора return.
44. Какое назначение оператора goto.
45. Как определить массив?
46. Как проинициализировать массив?
47. Какие варианты объявления с инициализацией вы знаете?
48. Как обратиться к элементу массива?
49. Как объявить многомерный массив?
50. Как проинициализировать многомерный массив?
51. Какие виды строк существуют в C#?
52. Как объявить C-строку?
53. Как осуществляется ввод-вывод строк?
54. Какие операции над строками вы знаете?
55. Перечислите операции над символами?

4 Задание

1. Создать новый проект.

2. Написать программы в соответствии с вариантом задания из пункта 5.1 и 5.2.
3. Отладить и протестировать программу.
4. Вариант определяется согласно номеру студента в списке группы.

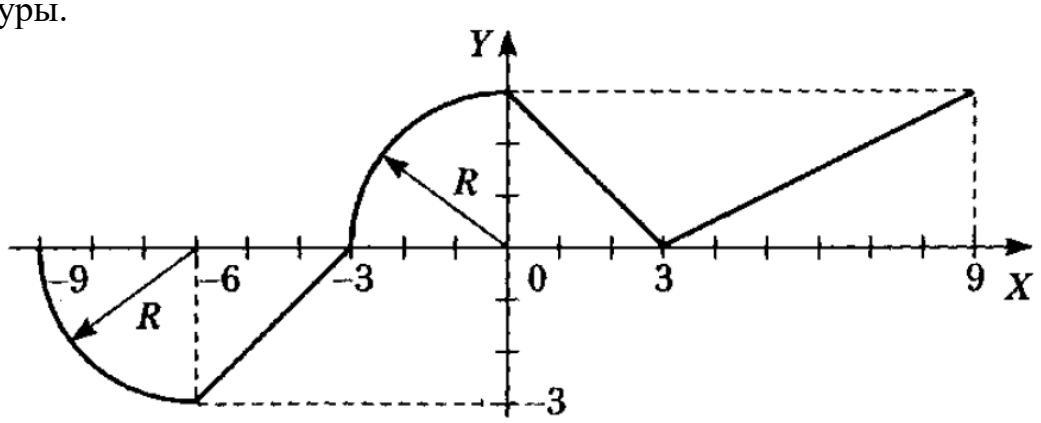
5 Варианты заданий

5.1 Управляющие операторы и операторы цикла

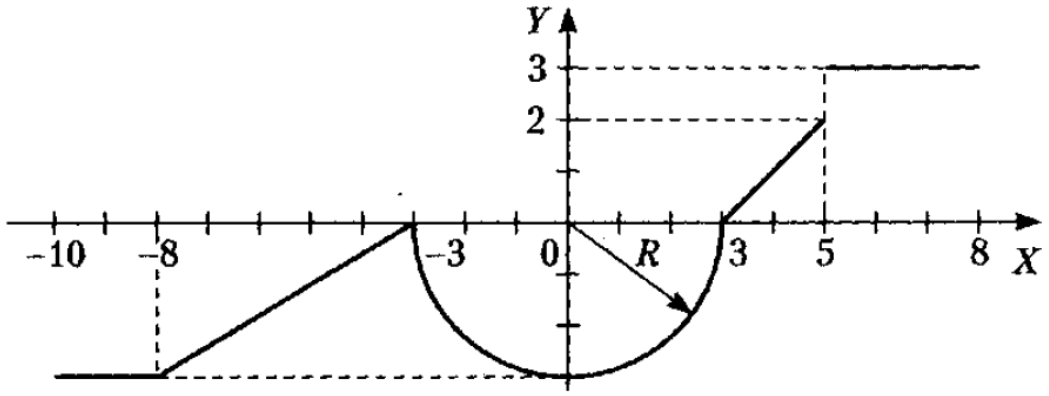
Управляющие операторы

Написать программу, которая по введенному значению аргумента вычисляет значение функции, заданной в виде графика. Параметр R вводится с клавиатуры.

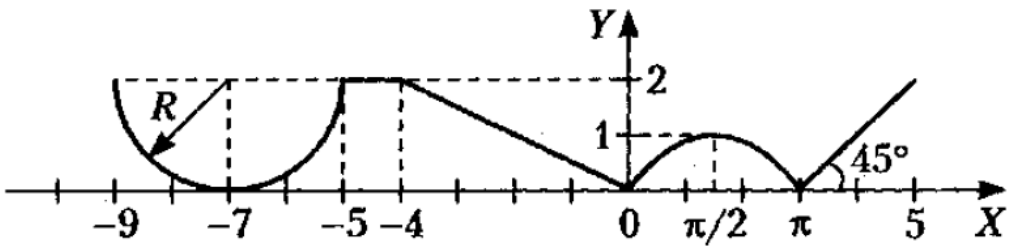
1



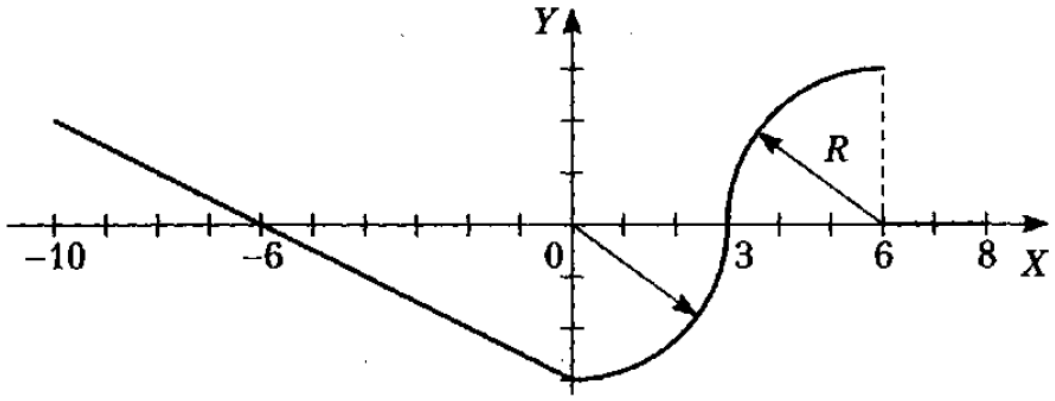
2



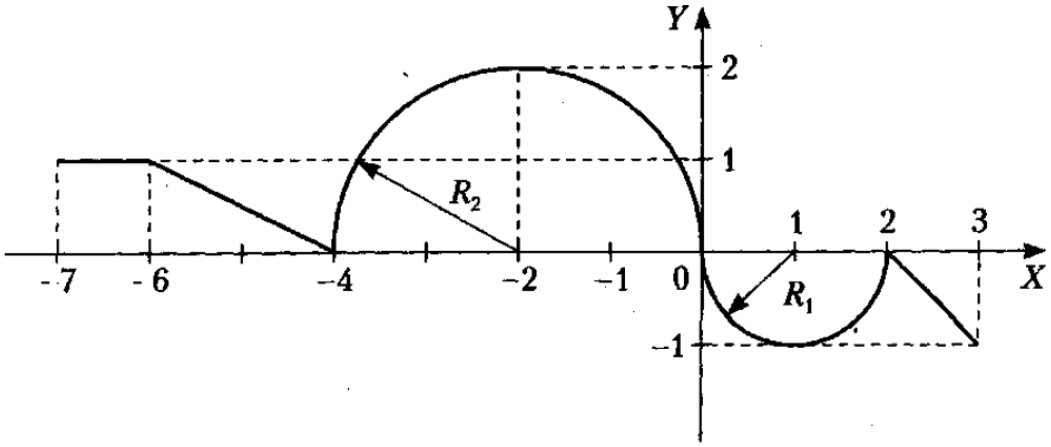
3



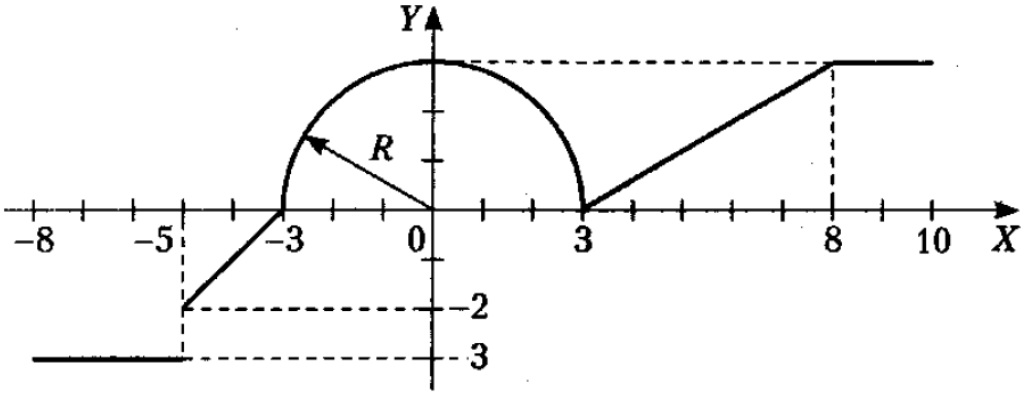
4


Y
R
−10
−6
0
3
6
8
X

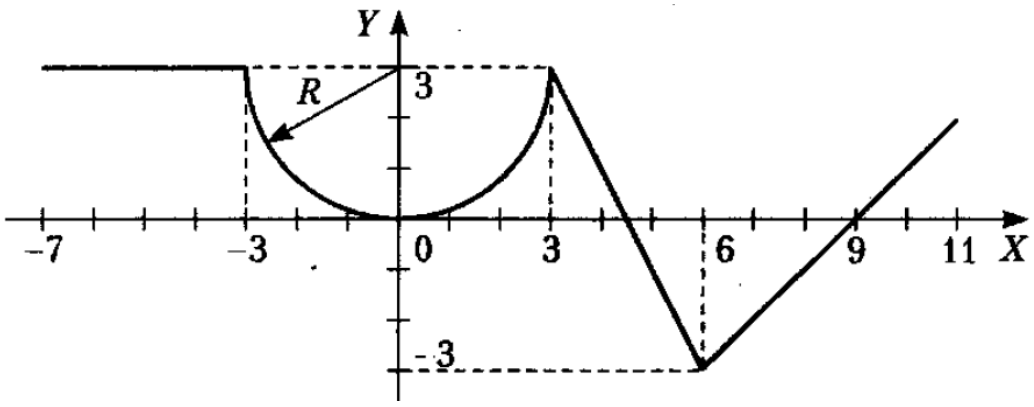
5


Y
2
1
R_2
1
2
3
-7
-6
−4
−2
−1
0
R_1
X
−1

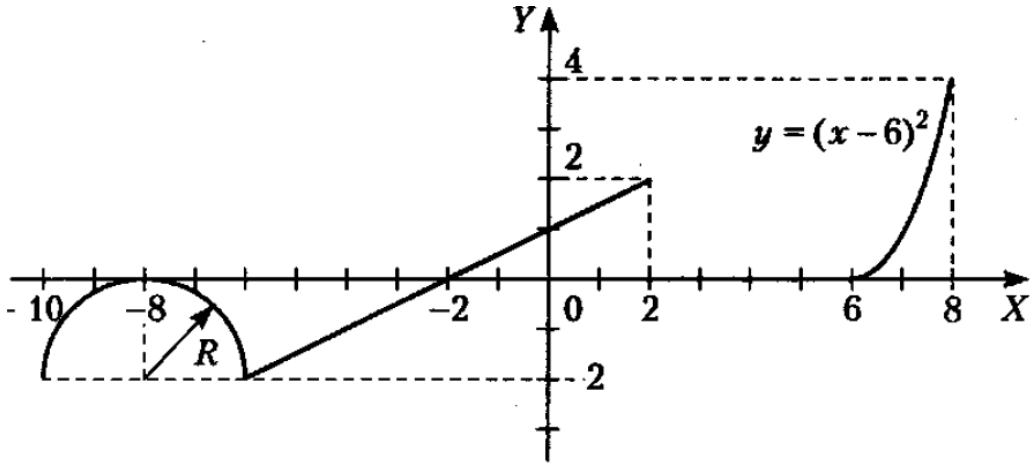
6


Y
R
−8
−5
−3
0
3
8
10
X
−2
3

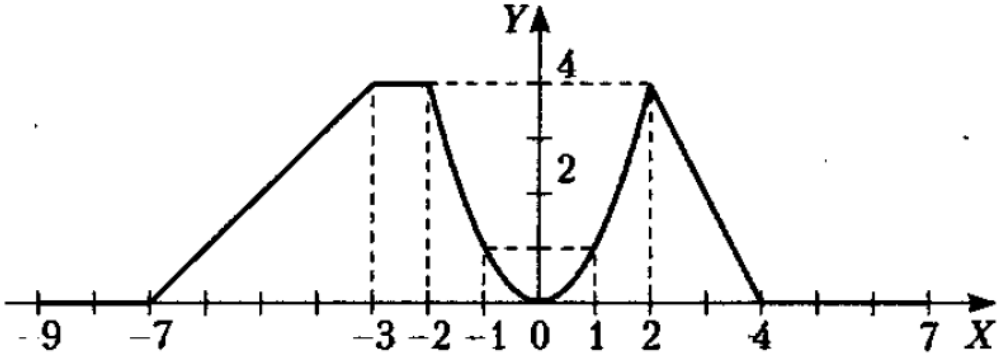
7


Y
R
3
−7
−3
0
3
6
9
11
X
-3

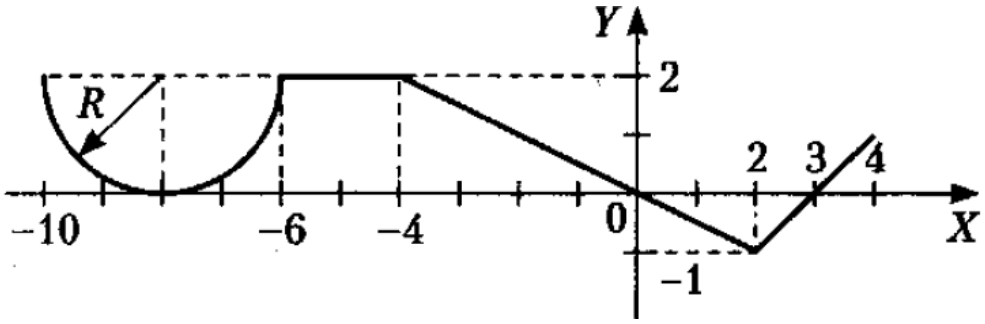
8


Y
4
2
y = (x − 6)²
-10
−8
R
−2
0
2
6
8
X
−2

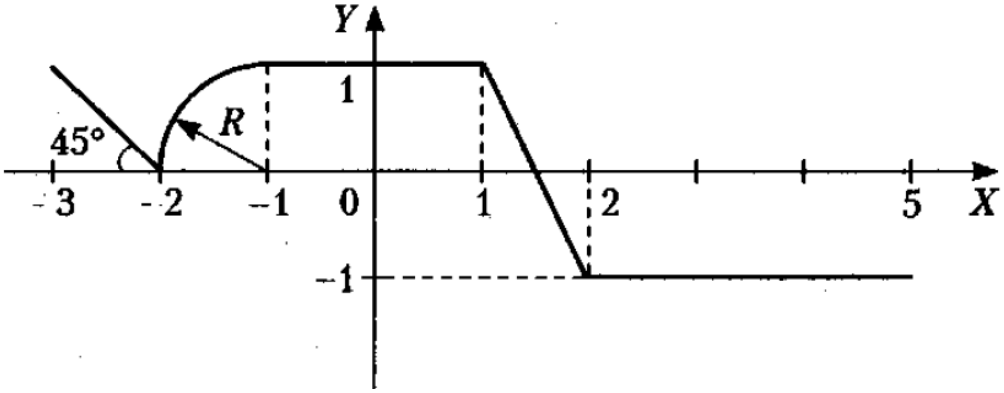
9


Y
4
2
-9
−7
−3
−2
−1
0
1
2
4
7
X

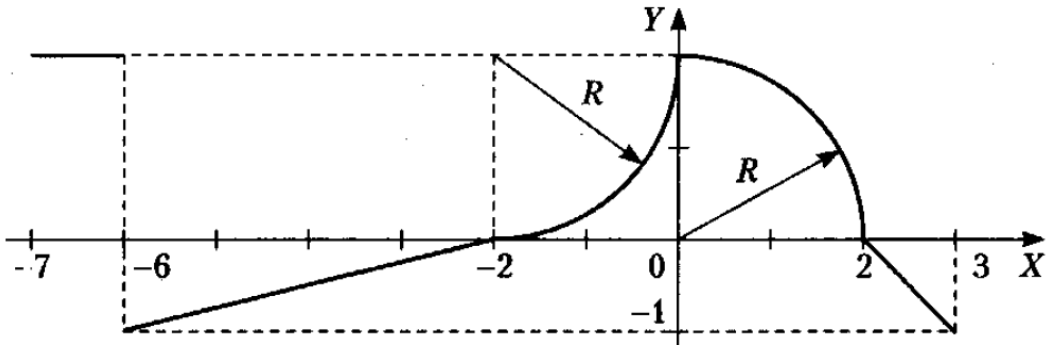
10


Y
2
R
2
3
4
−10
−6
−4
0
X
−1

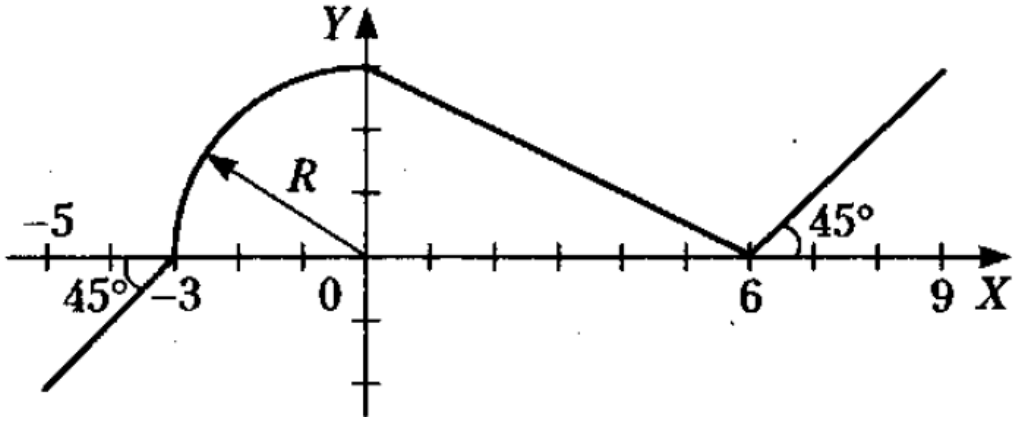
11


Y
1
45°
R
-3
-2
−1
0
1
2
5
X
−1

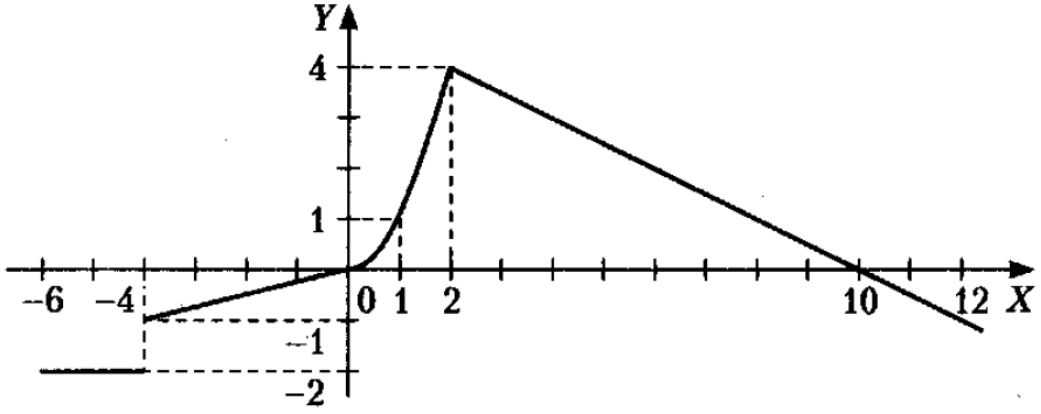
12


Y
R
R
-7
-6
-2
0
2
3
X
-1

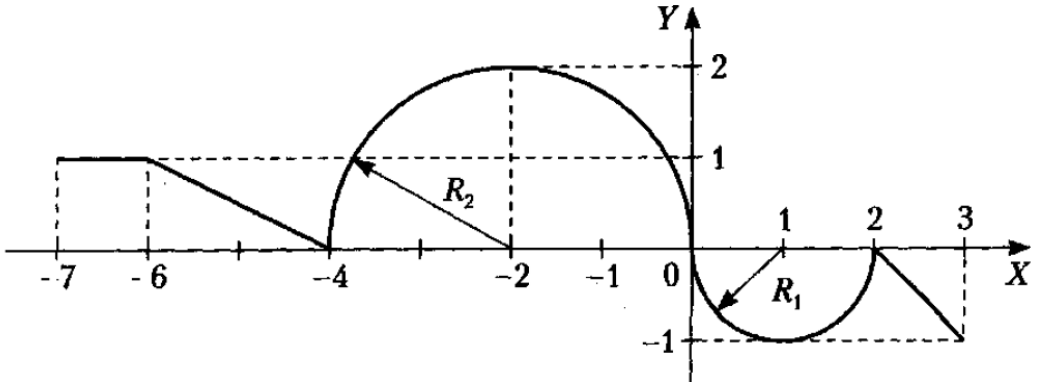
13


Y
R
-5
45°
45°
-3
0
6
9
X

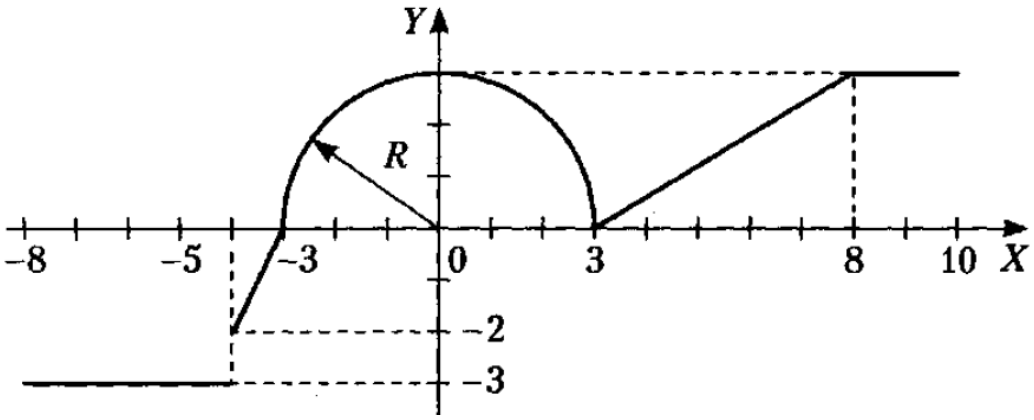
14


Y
4
1
-6
-4
0
1
2
10
12
X
-1
-2

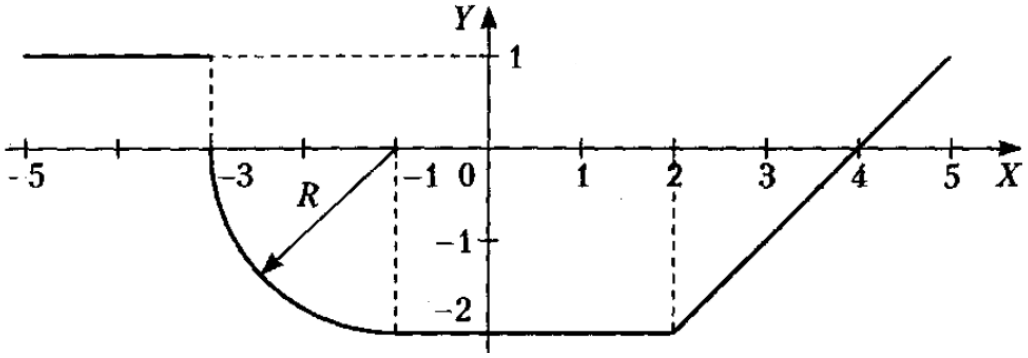
15


Y
2
1
R2
1
2
3
-7
-6
-4
-2
-1
0
X
R1
-1

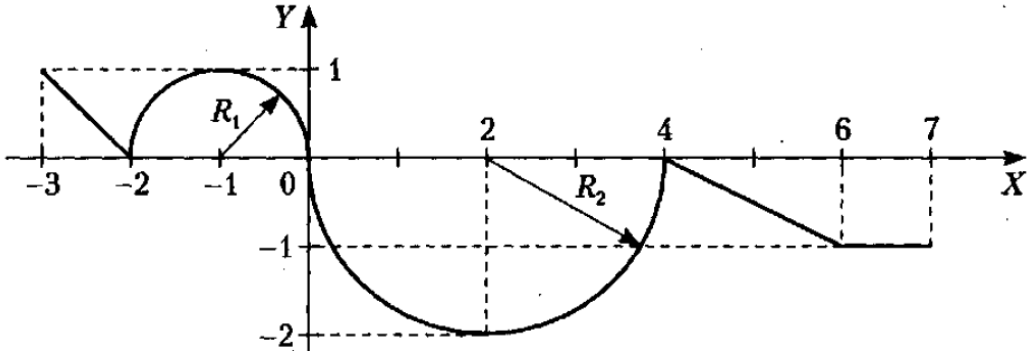
16


Y
R
-8
-5
-3
0
3
8
10
X
-2
-3

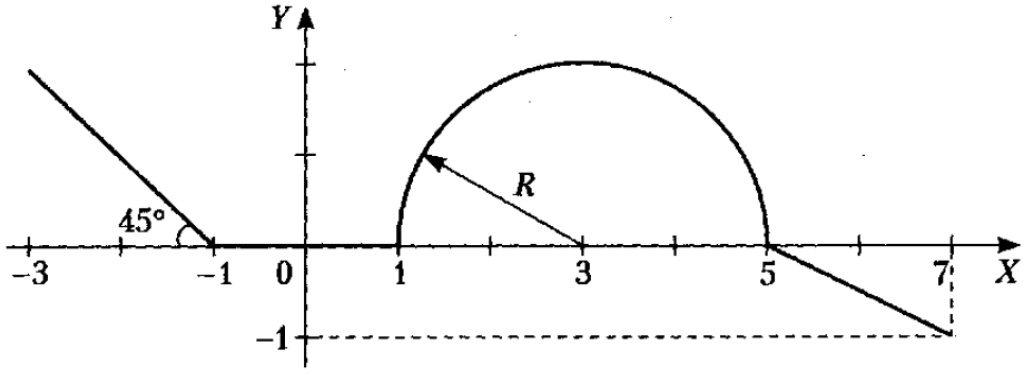
17


Y
1
-5
-3
R
-1
0
1
2
3
4
5
X
-1
-2

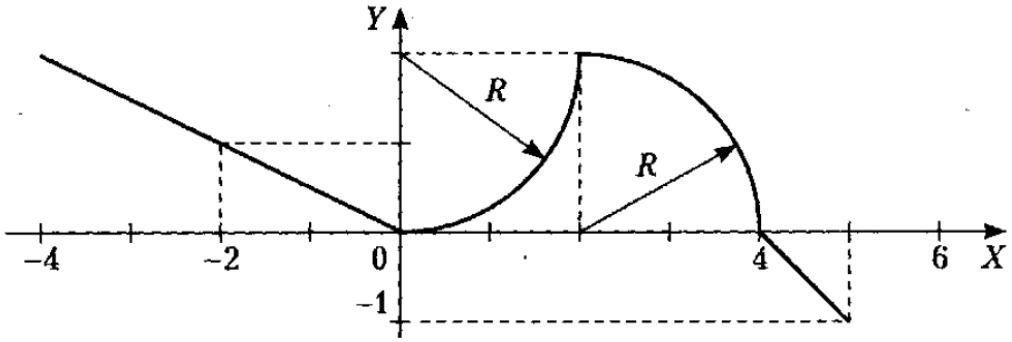
18


Y
1
R1
2
4
6
7
-3
-2
-1
0
R2
X
-1
-2

19


Y
R
45°
-3
-1
0
1
3
5
7
X
-1

20



Операторы цикла:

1. Имеется серия измерений элементов треугольника. Группы элементов пронумерованы. В серии в произвольном порядке могут встречаться такие группы элементов треугольника:
   a. основание и высота;
   b. две стороны и угол между ними (угол задан в радианах);
   c. три стороны.

   Разработать программу, которая запрашивает номер группы элементов, вводит соответствующие элементы и вычисляет площадь треугольника. Вычисления прекратить, если в качестве номера группы введен 0.
2. Начав тренировки, спортсмен в первый день пробежал 10 км. Каждый день он увеличивал дневную норму на 10% нормы предыдущего дня. Какой суммарный путь пробежит спортсмен за 7 дней?
3. Одноклеточная амеба каждые 3 часа делится на 2 клетки. Определить, сколько амеб будет через 3, 6, 9, 12,…, 24 часа.
4. Около стены наклонно стоит палка длиной x м. Один ее конец находится на расстоянии y м от стены. Определить значение угла a между палкой и полом для значений x = k м и y, изменяющегося от 2 до 3 м с шагом h м.
5. У гусей и кроликов вместе 64 лапы. Сколько может быть кроликов и гусей (указать все сочетания)?
6. Составить алгоритм решения задачи: сколько можно купить быков, коров и телят, платя за быка 10 руб., за корову — 5 руб., а за теленка — 0,5 руб., если на 100 руб. надо купить 100 голов скота?
7. Составить программу для проверки утверждения: «Результатами вычислений по формуле $x^2 + x + 17$ при $0 \geq x \geq 15$ являются простые числа». Все результаты вывести на экран.
8. Покупатель должен заплатить в кассу 5253 руб. У него имеются купюры по 1, 5, 10, 50, 100, 500 и 1000 руб. Сколько купюр разного достоинства отдаст покупатель, если он начинает платить с самых крупных купюр?
9. Ежемесячная стипендия студента составляет A руб., а расходы на проживание превышают стипендию и составляют B руб. в месяц. Рост цен ежемесячно увеличивает расходы на 3%. Составьте программу

расчета суммы денег, которую необходимо единовременно попросить у родителей, чтобы можно было прожить учебный год (10 месяцев), используя только эти деньги и стипендию.

10. Составить программу, которая печатает таблицу умножения и сложения натуральных чисел в двоичной системе счисления.
11. Составить программу, которая печатает таблицу умножения и сложения натуральных чисел в четырехричной системе счисления.
12. Составить программу, которая печатает таблицу умножения и сложения натуральных чисел в восьмеричной системе счисления.
13. Составить программу, которая печатает таблицу умножения и сложения натуральных чисел в шестнадцатеричной системе счисления.
14. Найти сумму всех n-значных чисел, кратных k ($1 \le n \le 4$).
15. Показать, что для всех n = 1, 2, 3, …, N
    (15 + 25 + ... + n5) + (17 + 27 + ... + n7) = 2(1 + 2 + ... + n)4.
16. Заменить буквы цифрами так, чтобы соотношение оказалось верным (одинаковым буквам соответствуют одинаковые цифры, разным — разные):
    ХРУСТ · ГРОХОТ = РРРРРРРРРРР.
17. Составить программу, которая находит наибольшее значение отношения трехзначного числа к сумме его цифр.
18. Вычислить сумму кодов всех символов, которые в цикле вводятся с клавиатуры до нажатия на клавишу Esc.
19. Вычислить количество точек с целочисленными координатами, находящихся в круге радиуса R (R > 0).
20. Напечатать в возрастающем порядке все трехзначные числа, в десятичной записи которых нет одинаковых цифр (операции деления и нахождения остатка от деления не использовать).

5.2 Массивы и строки

Одномерные массивы:

1. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • сумму отрицательных элементов массива;
   • произведение элементов массива, расположенных между максимальным и минимальным элементами.
   Упорядочить элементы массива по возрастанию.
2. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • сумму положительных элементов массива;
   • произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

3. В одномерном массиве, состоящем из n целочисленных элементов, вычислить:
   • произведение элементов массива с четными номерами;
   • сумму элементов массива, расположенных между первым и последним нулевыми элементами.
   Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные нулю, считать положительными).
4. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • сумму элементов массива с нечетными номерами;
   • сумму элементов массива, расположенных между первым и последним отрицательными элементами.
   Сжать массив, удалив из него все элементы, модуль которых не превышает единицу. Освободившиеся в конце массива элементы заполнить нулями.
5. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • максимальный элемент массива;
   • сумму элементов массива, расположенных до последнего положительного элемента.
   Сжать массив, удалив из него все элементы, модуль которых находится в интервале [a, b]. Освободившиеся в конце массива элементы заполнить нулями.
6. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • минимальный элемент массива;
   • сумму элементов массива, расположенных между первым и последним положительными элементами.
   Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.
7. В одномерном массиве, состоящем из n целочисленных элементов, вычислить:
   • номер максимального элемента массива;
   • произведение элементов массива, расположенных между первым и вторым нулевыми элементами.
   Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

8. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • номер минимального элемента массива;
   • сумму элементов массива, расположенных между первым и вторым отрицательными элементами.
   Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает единицу, а потом — все остальные.
9. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
   • максимальный по модулю элемент массива;
   • сумму элементов массива, расположенных между первым и вторым положительными элементами.
   Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.
10. В одномерном массиве, состоящем из n целочисленных элементов, вычислить:
    • минимальный по модулю элемент массива;
    • сумму модулей элементов массива, расположенных после первого элемента, равного нулю.
    Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине — элементы, стоявшие в нечетных позициях.
11. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
    • номер минимального по модулю элемента массива;
    • сумму модулей элементов массива, расположенных после первого отрицательного элемента.
    Сжать массив, удалив из него все элементы, величина которых находится в интервале [a, b]. Освободившиеся в конце массива элементы заполнить нулями.
12. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
    • номер максимального по модулю элемента массива;
    • сумму элементов массива, расположенных после первого положительного элемента.
    Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале [a, b], а потом — все остальные.
13. В одномерном массиве, состоящем из n вещественных элементов, вычислить:
    • количество элементов массива, лежащих в диапазоне от A до B;

• сумму элементов массива, расположенных после максимального элемента.

Упорядочить элементы массива по убыванию модулей.

14. В одномерном массиве, состоящем из n вещественных элементов, вычислить:

• количество элементов массива, равных нулю;

• сумму элементов массива, расположенных после минимального элемента.

Упорядочить элементы массива по возрастанию модулей.

15. В одномерном массиве, состоящем из n вещественных элементов, вычислить:

• количество элементов массива, больших C;

• произведение элементов массива, расположенных после максимального по модулю элемента.

Преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом — все положительные (элементы, равные нулю, считать положительными).

16. В одномерном массиве, состоящем из n вещественных элементов, вычислить:

• количество отрицательных элементов массива;

• сумму модулей элементов массива, расположенных после минимального по модулю элемента.

Заменить все отрицательные элементы массива их квадратами и упорядочить элементы массива по возрастанию.

17. В одномерном массиве, состоящем из n целочисленных элементов, вычислить:

• количество положительных элементов массива;

• сумму элементов массива, расположенных после последнего элемента, равного нулю.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых не превышает единицу, а потом — все остальные.

18. В одномерном массиве, состоящем из и вещественных элементов, вычислить:

• количество элементов массива, меньших C;

• сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, отличающиеся от максимального не более чем на 20%, а потом — все остальные.

19. В одномерном массиве, состоящем из n вещественных элементов, вычислить:

• произведение отрицательных элементов массива;

• сумму положительных элементов массива, расположенных до максимального элемента.

Изменить порядок следования элементов в массиве на обратный.

20. В одномерном массиве, состоящем из n вещественных элементов, вычислить:

• произведение положительных элементов массива;

• сумму элементов массива, расположенных до минимального элемента.

Упорядочить по возрастанию отдельно элементы, стоящие на четных местах, и элементы, стоящие на нечетных местах.

Многомерные массивы:

1. Дана целочисленная прямоугольная матрица. Определить:

   • количество строк, не содержащих ни одного нулевого элемента;

   • максимальное из чисел, встречающихся в заданной матрице более одного раза.
2. Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

   Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.
3. Дана целочисленная прямоугольная матрица. Определить:

   • количество столбцов, содержащих хотя бы один нулевой элемент;

   • номер строки, в которой находится самая длинная серия одинаковых элементов.
4. Дана целочисленная квадратная матрица. Определить:

   • произведение элементов в тех строках, которые не содержат отрицательных элементов;

   • максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.
5. Дана целочисленная квадратная матрица. Определить:

   • сумму элементов в тех столбцах, которые не содержат отрицательных элементов;

   • минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.
6. Дана целочисленная прямоугольная матрица. Определить:

   • сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;

   • номера строк и столбцов всех седловых точек матрицы.

Матрица A имеет седловую точку A_{ij}, если A_{ij} является минимальным элементом в i-й строке и максимальным — в j-м столбце.

7. Для заданной матрицы размером 8x8 найти такие k, при которых k-я строка матрицы совпадает с k-м столбцом.
   Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.
8. Характеристикой столбца целочисленной матрицы назовем сумму модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик.
   Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.
9. Соседями элемента A_{ij} в матрице назовем элементы A_{kl}, где $i-1 \le k \le i+1, j-1 \le l \le j+1, (k,l) \ne (i,j)$. Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания заданной вещественной матрицы размером 10x10.
   В сглаженной матрице найти сумму модулей элементов, расположенных ниже главной диагонали.
10. Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей (определение соседних элементов см. в варианте 9). Подсчитать количество локальных минимумов заданной матрицы размером 10 x 10.
    Найти сумму модулей элементов, расположенных выше главной диагонали.
11. Коэффициенты системы линейных уравнений заданы в виде прямоугольной матрицы. С помощью допустимых преобразований привести систему к треугольному виду.
    Найти количество строк, среднее арифметическое элементов которых меньше заданной величины.
12. Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями.
    Найти номер первой из строк, содержащих хотя бы один положительный элемент.
13. Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (в зависимости от введенного режима), n может быть больше количества элементов в строке или столбце.
14. Осуществить циклический сдвиг элементов квадратной матрицы размером MxN вправо на k элементов таким образом: элементы первой строки сдвигаются в последний столбец сверху вниз, из

него — в последнюю строку справа налево, из нее — в первый столбец снизу-вверх, из него — в первую строку; для остальных элементов — аналогично.

15. Дана целочисленная прямоугольная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент. Характеристикой строки целочисленной матрицы назовем сумму ее отрицательных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с убыванием характеристик.
16. Упорядочить строки целочисленной прямоугольной матрицы по возрастанию количества одинаковых элементов в каждой строке. Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.
17. Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2, 2), следующий по величине — в позиции (3, 3) и т. д., заполнив таким образом всю главную диагональ.
    Найти номер первой из строк, не содержащих ни одного положительного элемента.
18. Дана целочисленная прямоугольная матрица. Определить:
    • количество строк, содержащих хотя бы один нулевой элемент;
    • номер столбца, в котором находится самая длинная серия одинаковых элементов.
19. Дана целочисленная квадратная матрица. Определить:
    • сумму элементов в тех строках, которые не содержат отрицательных элементов;
    • минимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.
20. Дана целочисленная прямоугольная матрица. Определить:
    • количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент;
    • номера строк и столбцов всех седловых точек матрицы.
    Матрица A имеет седловую точку A_{ij}, если A_{ij} является минимальным элементом в i-й строке и максимальным — в j-м столбце.

Строковый тип

1. Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.
2. Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие введенное с клавиатуры слово.

3. Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.
4. Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.
5. Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.
6. Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.
7. Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих не более чем из четырех букв.
8. Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.
9. Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.
10. Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся на гласные буквы.
11. Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел.
12. Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы.
13. Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив прописной каждую первую букву слов, начинающихся с гласной буквы.
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.
15. Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.
16. Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.
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.
18. Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.

19. Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.
20. Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации.

ТЕМА 2. ОСНОВНЫЕ ПРИНЦИПЫ И ЭТАПЫ ООП. КЛАССЫ И ОБЪЕКТЫ. ЭЛЕМЕНТЫ КЛАССА. ПОЛЯ И МЕТОДЫ. СВОЙСТВА ОБЪЕКТОВ.

Лабораторная работа № 2

1 Цель и порядок работы

Цель работы – ознакомиться с понятием класса и объекта в языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Введение в объектно-ориентированное программирование

Как мы видим окружающий мир? Ответ зависит от нашего прошлого. Ученый может видеть мир как множество молекулярных структур. Художник видит мир как набор форм и красок. А кто-то может сказать, что мир — это множество вещей. Вероятно, первая мысль, которая пришла вам в голову, когда вы прочитали этот вопрос, была: какое значение имеет то, как кто-то видит мир?

Это имеет большое значение для программиста, которому необходимо написать программу, эмулирующую реальный мир.

Мы видим мир как составляющие его вещи, и, возможно, это правильно. Дом — вещь. Имущество, которое находится в доме, — вещи. То, что вы выкидываете, — вещи, покупаете вы вещи. Вещи, которые мы имеем, например, дом, сделаны из других вещей, таких как окна или двери.

Технически говоря, вещь (stuff) — это объект. То есть дом — объект. Вещи, которые находятся в доме, — объекты. Вещи, которые вы выбрасываете, — объекты, и вещи, которые вы хотите купить, — тоже объекты. Все мы, независимо от нашего прошлого, видим окружающий мир как множество объектов. Объект — это: человек, место, вещь, понятие и, возможно, событие.

Лучший способ изучить объекты состоит в том, чтобы исследовать наиболее важный из них — нас самих. В мире объектно-ориентированного программирования каждый из нас рассматривается как объект. Мы называем этот объект — человек. Человек, так же как дом, машина и любой другой объект реального мира, описывается с помощью двух наборов свойств: 1) атрибутов и 2) поведения. Атрибут — это характеристика объекта. Например, у человека есть имя, фамилия, рост и вес. Имя, фамилия, рост и вес — это атрибуты всех людей. Человека характеризуют и сотни других атрибутов, но

мы остановимся на этих четырех. Поведение — это действие, которое объект может осуществить. Человек может: сидеть, стоять, идти или бежать, и это не считая тысяч других вариантов поведения.

Кому: Иван Петров
345247, Тюмень
Ямская, 11, кв. 45

Заказ

Номер заказа	Дата заказа	Заказано через	Дата поставки	Срок
AT345	12.02.2013	UPS	14.02.2013	15 дней

Количество	Номер товара	Наименование	Цена	Сумма
4	42	Полотенце	100.00 руб.	400.00 руб.
3	58	Подушка	250.00 руб.	750.00 руб.
		Стоимость товара без налога		1150.00 руб.
		Налог		0.00 руб.
		Доставка		44.00 руб.
		Итого		1194.00 руб.

Рис. 2.1. Форма заказа — это объект, характеризуемый атрибутами и вариантами поведения

Каждый из объектов автомобиль, самолет, документ, форма заказа, характеризуется своими атрибутами и поведением. Атрибуты и варианты поведения автомобиля и самолета довольно очевидны. Оба они имеют ширину, высоту, вес, колеса и двигатель, а также множество других атрибутов. Автомобиль и самолет могут двигаться в некотором направлении, останавливаться и начинать двигаться в другом направлении, также они могут осуществлять сотни других действий (вариантов поведения). Однако сложнее определить атрибуты и варианты поведения документа или формы заказа (рис. 1.1). Атрибутами объекта – форма заказа являются: имя клиента, адрес клиента, заказанный товар, его количество, а также другая информация, которую можно найти в форме заказа. Варианты поведения формы заказа включают сбор информации, модификацию информации и обработку заказа.

Объекты рассматривают двумя способами: 1) как абстрактные объекты и 2) как реальные объекты. Абстрактный объект — это описание реального объекта. Например, абстрактный человек — это описание человека, которое содержит атрибуты и варианты поведения. Вот четыре атрибута, которые могут быть найдены у абстрактного человека (эти атрибуты только идентифицируют тип характеристики, например, имя или вес, но не определяют само имя или значение):

имя,

фамилия,

рост,
вес.

Абстрактный объект используется как модель для реального объекта. Реальный человек имеет все атрибуты и варианты поведения, определенные в абстрактном объекте, а также подробности, упущенные в абстрактном объекте. Например, абстрактный человек — это модель реального человека. Абстрактный человек определяет, что реальный человек должен иметь имя, фамилию, рост и вес. Реальный человек определяет значения, связанные с этими атрибутами, например, такие:

Иван,
Петров,
180 см,
80 кг.

Программисты создают абстрактный объект, а затем используют его для создания реального объекта. Реальный объект называется экземпляром (instance) абстрактного объекта. Можно сказать, что реальный человек — это экземпляр абстрактного человека.

Почему же целесообразнее смотреть на мир как на множество объектов? Фокусирование на объектах упрощает для нас понимание сложных вещей. Объекты позволяют уделять внимание важным для нас подробностям и игнорировать те подробности, которые нам не интересны. Например, преподаватель — это человек, и он имеет множество атрибутов и вариантов поведения, которые вам интересны. Тем не менее, вы, возможно, игнорируете множество атрибутов и вариантов поведения преподавателя и фокусируетесь только на тех, которые имеют отношение к вашему образованию.

Аналогично, преподаватель фокусируется на ваших атрибутах и вариантах поведения, которые показывают, как хорошо вы изучаете материал на занятиях. Другие атрибуты, такие как ваша работа на других занятиях или ваш рост и вес, игнорируются преподавателем.

И вы, и преподаватель упрощаете ваши отношения, решая, какие атрибуты и варианты поведения являются важными для ваших целей, и используя в ваших отношениях только их.

Несмотря на то, что в различных источниках делается акцент на те или иные особенности внедрения и применения объектно-ориентированного программирования (ООП), 3 основных (базовых) понятия ООП остаются неизменными. К ним относятся:

- наследование (Inheritance),
- инкапсуляция (Encapsulation),
- полиморфизм (Polymorphism).

ООП позволяет разложить проблему на связанные между собой задачи. Каждая проблема становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом случае исходная задача в целом упрощается, и программист получает возможность оперировать с большими по объему программами.

В этом определении ООП отражается известный подход к решению сложных задач, когда мы разбиваем задачу на подзадачи и решаем эти подзадачи по отдельности. С точки зрения программирования подобный подход значительно упрощает разработку и отладку программ.

Центром внимания ООП является объект. Объект - это осязаемая сущность, которая четко проявляет свое поведение.

Объект состоит из следующих трех частей:

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

Объект ООП - это совокупность переменных состояния и связанных с ними методов (операций). Эти методы определяют, как объект взаимодействует с окружающим миром.

Возможность управлять состояниями объекта посредством вызова методов в итоге и определять поведение объекта. Эту совокупность методов часто называют интерфейсом объекта.

Класс (class) - это сложный тип данных, в котором объединены элементы данных (поля) и методы, обрабатывающие эти данные и выполняющие операции по взаимодействию с окружающей средой.

Объект (object) - это конкретная реализация, экземпляр класса. В программировании отношения объекта и класса можно сравнить с описанием переменной, где сама переменная (объект) является экземпляром какого-либо типа данных (класса).

Обычно, если объекты соответствуют конкретным сущностям реального мира, то классы являются некими абстракциями, выступающими в роли понятий. Для формирования какого-либо реального объекта необходимо иметь шаблон, на основании которого и строится создаваемый объект. При рассмотрении основ ООП часто смешивают понятие объекта и класса. Дело в том, что класс - это некоторое абстрактное понятие, а объект - это физическая реализация класса (шаблона).

Методы (methods) - это функции (процедуры), принадлежащие классу. Метод класса вызывается конкретным экземпляром класса и привязан к описанию и структуре класса.

Сообщение (message) - это практически то же самое, что и вызов функций в обычном программировании. В ООП обычно употребляется выражение "послать сообщение" какому-либо объекту. Понятие "сообщение" в ООП можно объяснить с точки зрения основ ООП: мы не можем напрямую изменить состояние объекта и должны как бы послать сообщение объекту, что мы хотим так и так изменить его состояние. Объект сам меняет свое состояние, а мы только его просим об этом, посылая сообщения.

Инкапсуляция - это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования. Когда методы и данные объединяются таким способом, создается объект.

Применяя инкапсуляцию, как бы, возводим крепость, которая защищает данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы. Инкапсуляция подразумевает под собой скрытие данных (data hiding), что позволяет защитить эти данные.

Примером применения принципа инкапсуляции являются команды доступа к файлам. Обычно доступ к данным на диске можно осуществить только через специальные функции. Вы не имеете прямой доступ к данным, размещенным на диске. Таким образом, данные, размещенные на диске, можно рассматривать скрытыми от прямого вмешательства. Доступ к ним можно получить с помощью специальных функций, которые по своей роли схожи с методами объектов.

Наследование - это процесс, посредством которого, один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него.

Исследователи во многих областях естествознания тратят львиную долю времени на классификацию объектов в соответствии с определенными особенностями. Для животных, растений существуют специальные принципы классификации и разбиения на классы, подклассы, виды и подвиды и т.д. В результате образуется своего рода иерархия или дерево с одной общей категорией в корне и подкатегориями, разветвляющимися на подкатегории и т.д.

Пытаясь провести классификацию некоторых новых животных или объектов, мы задаем следующие вопросы: В чем сходство этого объекта с другими объектами общего класса? В чем различия?

Каждый класс имеет набор поведений и характеристик, которые его определяют. Более высокие уровни являются более общими. Каждый уровень является более специфическим, чем предыдущий уровень и менее общим. Когда характеристика определена, все категории ниже этого определения включают эту характеристику. Поэтому, когда мы говорим про того или иного конкретного представителя класса (отряда, вида и т.д.), то нам не надо говорить про его общие особенности, характерные для этого класса, а говорим только про его специфические особенности в рамках этого класса.

Смысл и универсальность наследования заключается в том, что не надо каждый раз заново (с нуля) описывать новый объект, а можно указать родителя (базовый класс) и описать отличительные особенности нового класса. В результате, новый объект будет обладать всеми свойствами родительского класса плюс своими собственными отличительными особенностями.

Пример: Создадим базовый класс "транспортное средство", который универсален для всех средств передвижения на 4-х колесах. Этот класс "знает" как двигаются колеса, как они поворачивают, тормозят и т.д. А затем на основе этого класса создадим класс "легковой автомобиль", который

унаследуем из класса "транспортное средство". Поскольку новый класс является наследником класса "транспортное средство", то класс "легковой автомобиль" унаследовал все особенности класса "транспортное средство" и в этом случае не надо в очередной раз описывать как двигаются колеса и т.д. После построения класса "легковой автомобиль" можно добавить особенности поведения, которые характерны для легковых автомобилей. В то же время можем взять за основу этот же класс "транспортное средство" и построить класса "грузовые автомобили". Описав отличительные особенности грузовых автомобилей, получим уже новый класс "грузовые автомобили". А, к примеру, на основании класса "грузовой автомобиль" уже можно описать определенный подкласс грузовиков и т.д. Таким образом, не надо каждый раз описывать все "с нуля". В этом и заключается главное преимущество использования механизма наследования. Сначала формируем простой шаблон, а затем все усложняя и конкретизируя, поэтапно создаем все более сложный шаблон.

В данном случае был приведен пример простого наследования, когда наследование производится только из одного класса. В некоторых объектно-ориентированных языках программирования определены механизмы наследования, позволяющие наследовать из одного и более класса. Однако реализация подобных механизмов зависят от самого применяемого языка ООП. Кроме того, следует отметить, что особенности реализации даже простого наследования могут различаться от языка ООП к языку.

В описаниях языков ООП принято класс, из которого наследуют называть родительским классом (parent class) или основой класса (base class). Класс, который получаем в результате наследования, называется порожденным классом (derived or child class). Родительский класс всегда считается более общим и развернутым. Порожденный же класс всегда более строгий и конкретный, что делает его более удобным в применении при конкретной реализации.

ООП - это процесс построения иерархии классов. А одним из наиболее важных свойств ООП является механизм, по которому типы классов могут наследовать характеристики из более простых, общих типов. Этот механизм называется наследованием. Наследование обеспечивает общность функций, в то же время допуская столько особенностей, сколько необходимо.

Полиморфизм – это свойство, которое позволяет одно и тоже имя использовать для решения нескольких технически разных задач.

В общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия, в зависимости от ситуации, возлагается на компилятор.

Применительно к ООП, целью полиморфизма, является использование

одного имени для задания общих для класса действий. На практике это означает способность объектов выбирать внутреннюю процедуру (метод) исходя из типа данных, принятых в сообщении.

Представьте, что нужно открыть замок и у нас есть связка ключей. Для того, чтобы открыть дверь мы перебираем один ключ за другим пока не найдем подходящий. Т.е. когда шаблон замка совпадает с шаблоном параметров ключа, замок открывается. Аналогично работает компилятор при наличие нескольких функций. Он последовательно проверяет шаблоны функций с одним и тем же именем пока не найдет подходящий.

Механизм работы ООП можно описать примерно так: при вызове того или иного метода класса сначала ищется метод у самого класса. Если метод найден, то он выполняется и поиск этого метода на этом завершается. Если же метод не найден, то обращаемся к родительскому классу и ищем вызванный метод у него. Если найден – поступаем как при нахождении метода в самом классе. А если нет - продолжаем дальнейший поиск вверх по иерархическому дереву. Вплоть до корня (верхнего класса) иерархии.

2.2 Синтаксис объявления класса

Классы C# – это шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные и какую функциональность может иметь каждый конкретный объект (экземпляр) этого класса. Например, класс, представляющий студента, может определять такие поля, как studentID, firstName, lastName, group, и т.д. которые нужны для хранения информации о конкретном студенте.

Класс также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного студента, устанавливаете значения полей экземпляра и используете его функциональность. При создании классов, как и всех ссылочных типов – используется ключевое слово new для выделения памяти под экземпляр. В результате объект создастся и инициализируется (числовые поля инициализируются нулями, логические – false, ссылочные – в null и т.д.).

Синтаксис объявления и инициализации класса:

```
[спецификатор][модификатор] Class <имя класса>
{
        [спецификатор][модификатор]     тип <имя поля1>;
        [спецификатор][модификатор]     тип <имя поля2>;
        …
        [спецификатор][модификатор]     тип <Метод1()>
        {...}
        [спецификатор][модификатор]     тип <Метод2()>
        {...}
}
```

Пример объявления класса описывающего студента:

```
class Student
{
        public int studentID;
        public string firstName;
        public string lastName;
        public string group;
}

class Program
{
        static void Main(string[] args)
        {
                Student st1;
                st1 = new Student();
                Student st2 = new Student();
        }
}
```

Доступ к полям и методам класса осуществляется через «.» из- под объекта класса. Доступ к содержимому класса вне его границ, осуществляется только к общедоступным данным. Остальные остаются инкапсулированными.

2.3 Спецификаторы доступа

С помощью спецификаторов доступа можно регулировать доступность некоторого типа или данных внутри типа.

При определении класса с видимостью в рамках файла, а не другого класса, его можно сделать открытым (**public**) или внутренним (**internal**). Открытый тип доступен любому коду любой сборки. Внутренний класс доступен только из сборки, где он определен. По умолчанию компилятор C# делает класс внутренним.

При определении члена класса (в том числе вложенного) можно указать спецификатор доступа к члену. Спецификаторы определяют, на какие члены можно ссылаться из кода. В общеязыковой среде выполнения (Common Language Runtime, CLR) определен свой набор возможных спецификаторов доступа, но в каждом языке программирования существует свой синтаксис и термины. Рассмотрим спецификаторы определяющие уровень ограничения – от максимального (**private**) до минимального (**public**):

private – данные доступны только методам внутри класса и вложенных в него классам,

protected – данные доступны только методам внутри класса (и вложенным в него классам) или одном из его производных классов,

internal – данные доступны только методам в сборке

protected internal – данные доступны только методам вложенного или производного типа класса и любым методам сборки,

public – данные доступны всем методам во всех сборках.

Доступ к члену класса можно получить, только если он определен в видимом классе. То есть, если в сборке А определен внутренний класс, имеющий открытый метод, то код сборки Б не сможет вызвать открытый метод, поскольку внутренний класс сборки А не доступен из Б.

В процессе компиляции кода компилятор проверяет корректность обращения кода к классам и членам. Обнаружив некорректную ссылку на какие-либо классы или члены, выдается ошибка компиляции.

Если не указать явно спецификатор доступа, компилятор C# выберет по умолчанию закрытый – наиболее строгий из всех.

Если в производном классе переопределяется член базового – компилятор C# потребует, чтобы у членов базового и производного классов был одинаковый спецификатор доступа. При наследовании базовому классу CLR позволяет понижать, но не повышать уровень доступа к члену.

2.4 Поля класса

Поле – это переменная, которая хранит значение любого стандартного типа или ссылку на ссылочный тип. При объявлении полей могут указываться следующие модификаторы:

- Если модификатор не указывать, то это означает, что поле связано с экземпляром класса, а не самим классом.
- Модификатор **static** – означает, что поле является частью класса, а не объекта.
- Модификатор **readonly** – означает, что поле будет использоваться только для чтения и запись в поле разрешается только из кода метода конструктора либо сразу при объявлении.

CLR поддерживает изменяемые (**read/write**) и неизменяемые (**readonly**) поля. Большинство полей – изменяемые. Это означает, что значение таких полей может многократно меняться во время исполнения кода. Неизменяемые поля сродни константам, но являются более гибкими, так как значение константам можно задать только при объявлении, а у неизменяемых полей это еще можно сделать и в конструкторах. Важно понимать, что неизменность поля ссылочного типа означает неизменность ссылки, которую оно содержит, но только не объекта, на которую эта ссылка указывает. То есть перенаправить ссылку на другое место в памяти мы не можем, но изменить значение объекта, на который указывает ссылка – можем. Значения неизменяемых полей значимых типов – изменять не можем.

Пример:

```
class MyClass
{
        public readonly int var1 = 10;
```

```
        public readonly int[] myArr = { 1, 2, 3 };
        public readonly int var2; // инициализация readonly
                                  // поля при объявлении
        public MyClass(int i)
        {
            var2 = i;     // инициализация readonly
        }                 // поля в конструкторе
}

class Program
{
        static void Main(string[] args)
        {
            MyClass obj = new MyClass();
            obj.var = 100;                  // Ошибка
            obj.myArr = new int[10];        // Ошибка
            obj.myArr[0] = 11;              //Ошибки нет
        }
}
```

Если объявляется статическое поле, то оно принадлежит классу в целом, а не конкретному объекту. И соответственно получить доступ к такому объекту можно только из-под класса, используя следующий синтаксис:

<Имя класса>.<имя поля>

Например, есть класс Bank. В этом классе будет статическое поле balance. Сымитируем ситуацию, когда в любом филиале банка можно будет положить деньги на депозит или взять кредит. Пусть все филиалы работают с общим счетом.

```
class Bank
{
        public static float balance = 1000000;
}
class Program
{
        static void Main(string[] args)
        {
            Bank filial1 = new Bank();
            Bank filial2 = new Bank();
            Console.WriteLine("1-ому филиалу доступно {0:C}",
                    Bank.balance);
            Console.WriteLine("2-ому филиалу доступно {0:C}",
                    Bank.balance);
            Console.WriteLine("В 1-ом филиале взяли кредит на
            100000" + ", осталось {0:C}", Bank.balance-=100000);
```

```
                Console.WriteLine("2-ому филиалу доступно {0:C}",
                        Bank.balance);
                Console.WriteLine("В 2-ом филиале взяли кредит на
                200000" +", осталось {0:C}", Bank.balance -= 200000);
                Console.WriteLine("1-ому филиалу доступно {0:C}",
                        Bank.balance);
                Console.WriteLine("В 1-ом филиале открыли депозит на "
                + "200000, осталось {0:C}", Bank.balance += 200000);
        }
}
```

Результат выполнения программы изображен на рисунке 2.2.

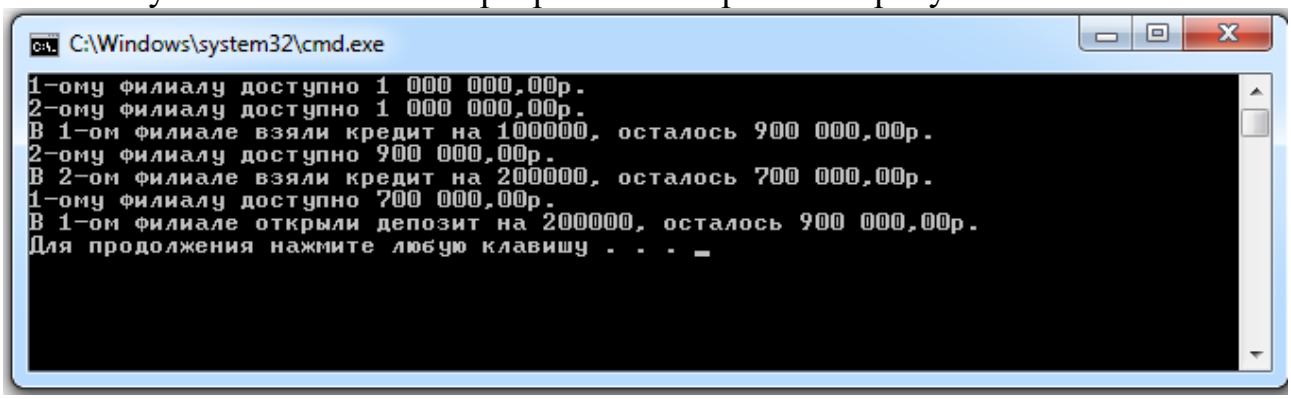


Рис. 2.2 Результат выполнения программы.

2.5 Методы класса

Описание метода

Все функции в языке C# обязательно должны определяться внутри классов или структур. Каждый метод должен быть объявлен отдельно как общедоступный или приватный. То есть блоки public:, private: для группировки нескольких методов использовать нельзя. Все методы C# объявляются и определяются внутри определения класса, таким образом нельзя отделить реализацию метода от объявления.

Определение метода состоит из: спецификаторов и модификаторов, типа возвращаемого значения, за которым следует имя метода, списка аргументов (если они есть), заключенных в скобки, и тела метода, заключенного в фигурные скобки:

[спецификаторы][модификаторы] тип_возврата
<Имя Метода>([параметры])

{
// Тело метода
}

Добавим в класс Student (описанный выше) метод. При этом сделаем

все поля класса закрытыми. Используя практику сокрытия данных класса и предоставления открытых методов по работе с этими данными, тем самым поддерживается принцип инкапсуляции данных и уменьшается вероятность некорректной работы программы.

```
class Student
{
        private string firstName = "Петя";
        public void ShowName()
        {
                Console.WriteLine(firstName);
        }
}
class Program
{
        static void Main(string[] args)
        {
                Student st = new Student();
                st.ShowName();
        }
}
```

Результат выполнения программы изображен на рисунке 2.3.

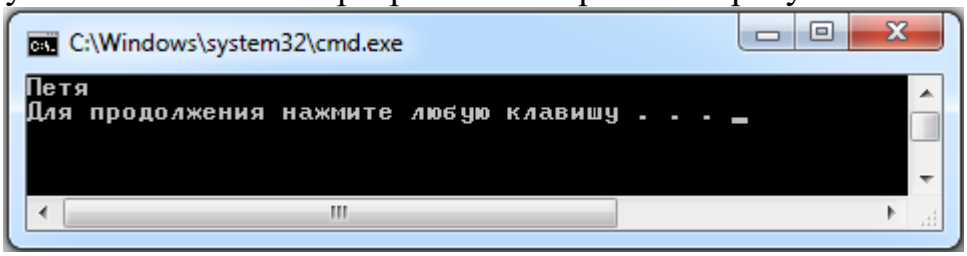


Рис. 2.3 Результат выполнения программы.

Для работы со статическими полями – были введены статические методы. Эти методы, как и статические поля, принадлежат классу, а не объекту. Они исключают возможность вызова из-под объекта и соответственно не работают с нестатическими полями.

Предположим, что все создаваемые нами студенты будут студентами ТюмГУ и добавим в наш класс Student статическое поле, которое будет содержать имя учебного заведения. Чтоб поле нельзя было изменить – сделаем его закрытым и позволим статическому методу ShowUniversity работать с нашим полем в режиме чтения.

```
class Student
   {
      private static string UniversityName = "ТюмГУ";
      // старые поля и методы остаются без изменения
      public static void ShowUniversity()
      { Console.WriteLine(UniversityName); }
```

```
}
class Program
{
   static void Main(string[] args)
   { Student.ShowUniversity(); }
}
```

Результат выполнения программы изображен на рисунке 2.4.

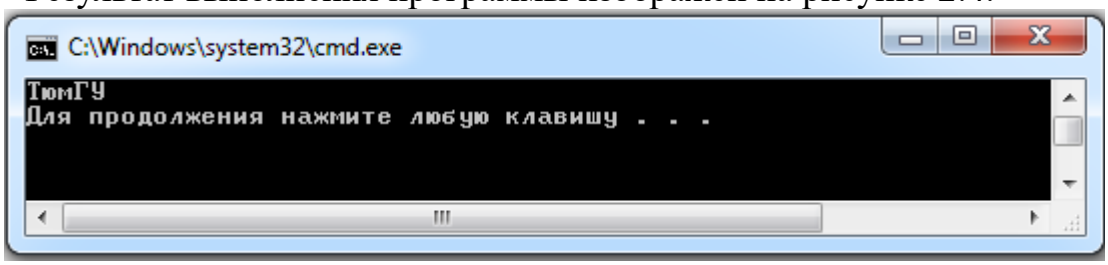


Рис. 2.4 Результат выполнения программы.

Передача параметров

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.
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 необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.

Параметры, указываемые в заголовке метода при его описании, называются формальными параметрами.

Параметры, указываемые при вызове метода, называются фактическими параметрами.

Корректность передачи параметров гарантируется соблюдением порядка перечисления в заголовке метода и совместимостью по присваиванию между соответствующими фактическими и формальными параметрами.

Существуют два способа передачи параметров: по значению и по ссылке. Когда переменная передастся по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым производятся над переменной внутри метода, останутся в силе после его завершения. С другой стороны, если переменная передается по значению, то вызываемый метод получает копию этой переменной, и соответственно все изменения в копии по завершении метода будут утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

При передаче параметров нужно быть осторожным в отношении ссылочных типов. Поскольку переменная ссылочного типа содержит лишь ссылку на объект, то именно ссылка будет скопирована при передаче параметра, а не сам объект. То есть изменения, произведенные в самом объекте, сохранятся. В отличие от этого переменные типа значений действительно содержат данные, поэтому в метод передается копия самих данных. Например, переменная a передастся в метод по значению, и любые изменения, которые сделает метод с соответствующим переменной a формальным параметром, не изменят значения переменной a. В противоположность этому, если в метод передастся массив или любой другой ссылочный тип, такой как класс, то изменения, сделанные при выполнении тела метода, будут отражены в исходном объекте массива.

Пример:

```
class Program
{
        static void MyFunctionByVal(int[] myArr, int i)
        { myArr[0] = 100;  i = 100; }
        static void Main(string[] args)
        {
                int i = 0;
                int[] myArr = { 0, 1, 2, 4 };
                Console.WriteLine("i = {0}", i);
                Console.WriteLine("MyArr[0] = {0}", myArr[0]);
                Console.WriteLine("Вызов MyFunction");
                MyFunctionByVal (myArr, i);
                Console.WriteLine("i = {0}", i);
                Console.WriteLine("MyArr[0] = {0}", myArr[0]);
        }
}
```

Результат выполнения программы изображен на рисунке 2.5.

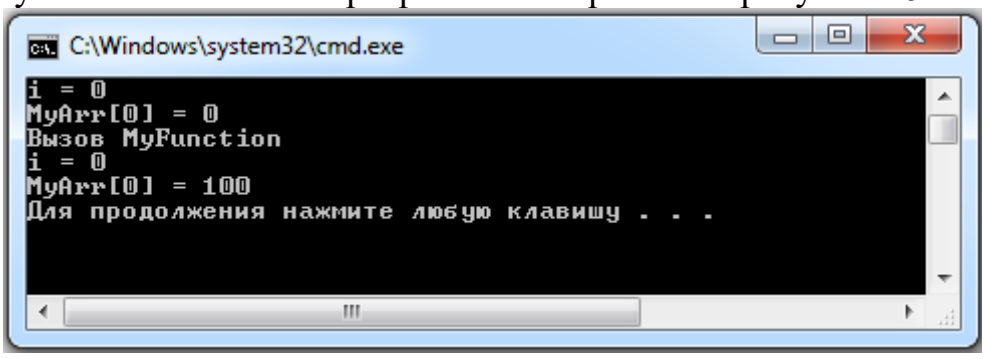


Рис. 2.5 Результат выполнения программы.

Передача параметров по значению.

При вызове метода:

а) выполняется выделение памяти под формальные параметры и локальные данные (в стеке или в специальной области памяти для локальных

данных);

б) выполняется копирование значений фактических параметров в память, выделенную для формальных параметров.

Во время выполнения метода:

а) изменение значений формальных параметров не оказывает никакого влияния на содержимое ячеек памяти фактических параметров.

При окончании выполнения метода:

а) память, выделенная под формальные параметры и локальные данные, очищается;

б) новые значения формальных параметров, полученные в процессе выполнения тела метода, теряются вместе с очисткой памяти.

Параметр-значение описывается в заголовке метода следующим образом:

тип имя

Пример:

```
class Program
   {
     private static void MyFunction(int i)
      {
         Console.WriteLine("Внутри функции MyFunction до
                  изменения i = {0}", i);
       i = 100;
       Console.WriteLine("Внутри функции MyFunction после
                изменения i = {0}", i);
      }
     static void Main(string[] args)
      {
       int i = 10;
       Console.WriteLine("Внутри метода Main до передачи в метод
                  MyFunction i = {0}", i);
       MyFunction(i);
       Console.WriteLine("Внутри метода Main после передачи в метод
MyFunction i = {0}", i);
      }
   }
```

Результат выполнения программы изображен на рисунке 2.6.

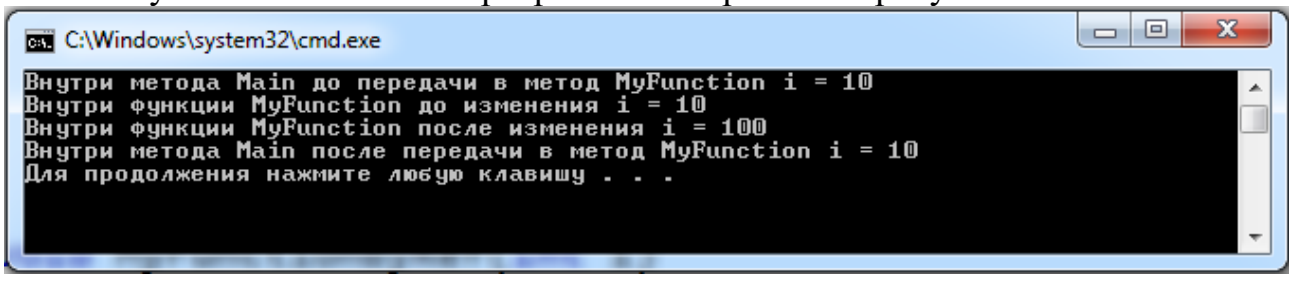


Рис. 2.6 Результат выполнения программы.

Передача параметров по ссылке (адресу).

При вызове метода:

а) выполняется выделение памяти только для локальных данных и для сохранения адресов фактических параметров (в стеке или в специальной области памяти для локальных данных);

б) выполняется копирование адресов (но не значений!) фактических параметров в выделенную память для локальных параметров;

в) использовать в качестве фактических параметров константы запрещено.

Во время выполнения метода:

а) никаких ограничений на использование параметров данного вида не накладывается;

б) изменение значений формальных параметров, используя скопированные адреса, выполняется непосредственно на ячейках памяти соответствующих фактических параметров.

При окончании выполнения метода:

а) специального копирования результата не требуется, поскольку все действия с формальными параметрами выполнялись непосредственно над ячейками памяти фактических параметров;

б) память, выделенная для работы метода, очищается.

Использование модификатора ref.

Ключевым словом ref помечаются те параметры, которые должны передаваться в метод по ссылке. Таким образом, мы будем внутри метода манипулировать данными, объявленными в вызывающем методе. Аргументы, которые передаются в метод с ключевым словом ref, обязательно должны быть проинициализированы, иначе компилятор выдаст сообщение об ошибке.

Описание параметра-ссылки в заголовке метода следующее:

ref тип имя

Пример:

```
class Program
   {
     private static void MyFunctionByRef(ref int i)
      {
         Console.WriteLine("Внутри функции MyFunction до
                изменения i = {0}", i);
       i = 100;
       Console.WriteLine("Внутри функции MyFunction после
              изменения i = {0}", i);
      }
     static void Main(string[] args)
      {
       int i = 10;
       Console.WriteLine("Внутри метода Main до передачи в метод
```

```
                MyFunction i = {0}", i);
        MyFunctionByRef(ref i);
        Console.WriteLine("Внутри метода Main после передачи в
                метод MyFunction i = {0}", i);
      }
   }
```

Результат выполнения программы изображен на рисунке 2.7.

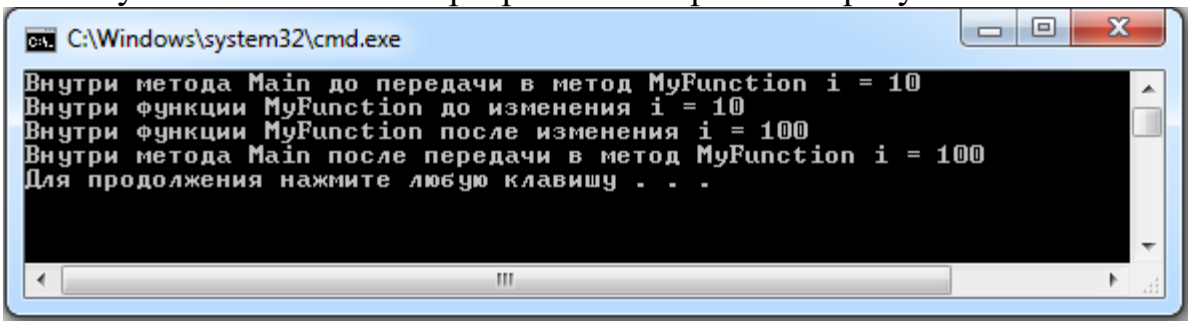


Рис. 2.7 Результат выполнения программы.

Использование модификатора out

Параметры, обозначенные ключевым словом out, также используются для передачи по ссылке. Отличие от ref состоит в том, что параметр считается выходным и соответственно компилятор разрешит не инициализировать его до передачи в метод и проследит, чтоб метод занес в этот параметр значение (иначе будет выдано сообщение об ошибке).

Описание параметра-ссылки в заголовке метода следующее:

out тип имя

Пример:

```
class Program
{
        static void Main(string[] args)
        {
                int i;
                GetDigit(out i);
                Console.WriteLine("i = " + i);
        }
        private static void GetDigit(out int digit)
        {
                digit = new Random().Next(10);
        }
}
```

Результат выполнения программы изображен на рисунке 2.8.

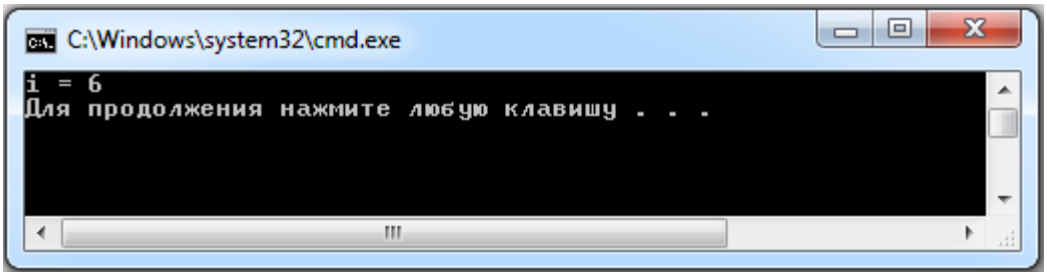


Рис. 2.8 Результат выполнения программы.

Создание методов с переменным количеством параметров

Иногда бывает удобно создать метод, в который можно передавать разное количество аргументов. Язык C# предоставляет такую возможность с помощью ключевого слова **params**. Параметр, помеченный этим ключевым словом, размещается в списке параметров последним и обозначает массив заданного типа неопределенной длины, например:

public int Sum(int a, out int b. params int [ ] c) ...

В этот метод можно передать три и более параметров. Внутри метода к параметрам, начиная с третьего, обращаются как к обычным элементам массива. Количество элементов массива получают с помощью его свойства **Length**. При использовании ключевого слова необходимо учитывать, что параметр, помечаемый ключевым словом **params** должен стоять последним в списке параметров.

Пример:

```
class Program
{
        static void Main(string[] args)
        {
                Console.WriteLine("Сумма = " + Sum( 1, 2, 3, 4, 5 ));
        }
        private static int Sum(params int[] arr)
        {
                int res = 0;
                foreach (int i in arr)
                res += i;
                return res;
        }
}
```

Результат выполнения программы изображен на рисунке 2.9.

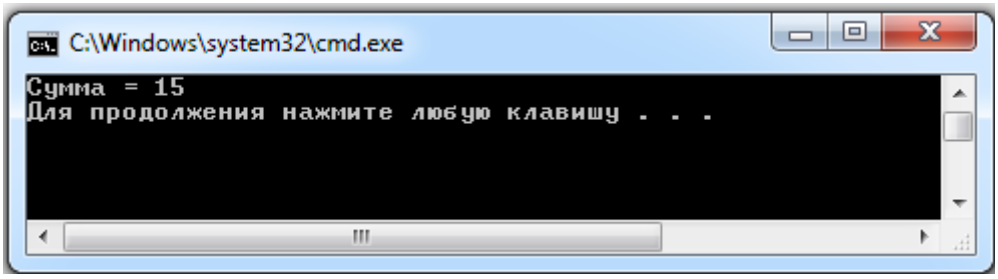


Рис. 2.9 Результат выполнения программы.

Ключевое слово return

Метод может завершить свое выполнение тремя способами:

- Когда управление дойдет до завершающей фигурной скобки (при этом метод ничего не возвращает).
- Когда управление дойдет до ключевого слова **return** (без возвращаемого значения и тип возвращаемого значения метода – **void**).
- Когда управление дойдет до ключевого слова **return** (после которого стоит возвращаемое значение, метод что-либо возвращает).

Метод может содержать несколько операторов **return**, если это необходимо для реализации алгоритма. Если метод описан как **void**, выражение не указывается. Выражение, указанное после **return**, неявно преобразуется к типу возвращаемого методом значения и передается в точку вызова метода.

private static int f1() { return 1; } // правильно

private static void f2() { return 1; } // неправильно, f2 не должен возвращать значение

private static double f3() { return 1; } // правильно, 1 неявно преобразуется к типу double

Метод, возвращающий значение, должен делать это с помощью оператора **return**. Если поток управления достигнет конца метода, возвращающего значение, не встретив на своем пути оператор **return**, в вызывающий модуль вернется "мусор".

Если метод не объявлен со спецификатором **void**, его можно использовать в качестве операнда в любом выражении.

Пример: Программа возвращает позицию числа в неотсортированном массиве, а если число не найдено, возвращает число –1.

```
class Program
   {
      static void Main(string[] args)
      {
         int num;
         Console.WriteLine("Введите искомое число: ");
         string buf = Console.ReadLine();
```

```
            int x = Convert.ToInt32(buf);
            if ((num = find_number(x)) == -1)
                Console.WriteLine("Такое число не найдено");
            else Console.WriteLine("Позиция числа в массиве {0}", num);

        }
        private static int find_number(int number)
         {
            int N = 7, i;
            int[] array = { 21, 2, -4, 6, 1, 11, -3 };
            for (i = 0; i < N; i++)
              if (number == array[i]) return i;
            return -1;
         }
    }
```

Результат выполнения программы изображен на рисунке 2.10.

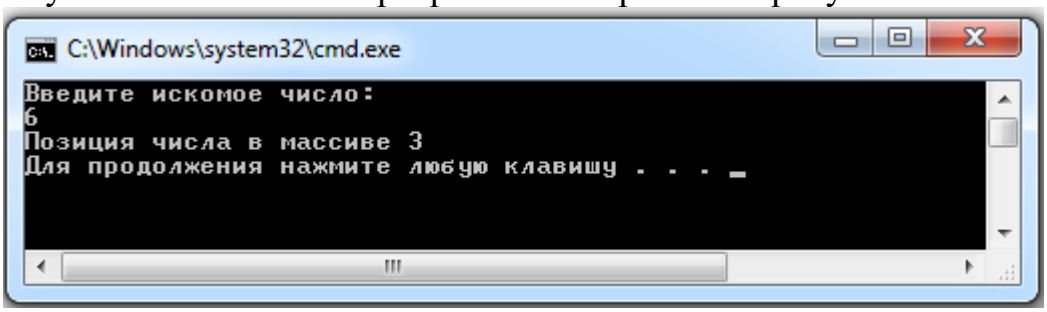


Рис. 2.10 Результат выполнения программы.

Методы можно разделить на три категории:

К первой категории относятся вычислительные методы. Они выполняют некие операции над своими аргументами и возвращают результат вычислений.

Методы второго типа выполняют обработку информации. Их возвращаемое значение просто означает, успешно ли завершены операции.

Методы третьего вида не имеют явно возвращаемого значения. По существу, эти методы являются процедурами и не вычисляют никакого результата.

Пример:

```
class Program
   {
      static void Main(string[] args)
      {
         int x, y, z;
         x = 10; y = 20;
         z = mul(x, y);                    /* 1 */
         Console.WriteLine(mul(x,y));   /* 2 */
         mul(x, y);                        /* 3 */
       }
```

```
    private static int mul(int a, int b)
     {
       return a * b;
     }
  }
```

В первой строке значение, возвращаемое методом **mul()**, присваивается переменной z. Во второй строке возвращаемое значение ничему не присваивается, но используется внутри метода **Console.WriteLine().** В третьей строке значение, возвращаемое с помощью оператора **return**, отбрасывается.

Рекурсивные методы

Рекурсивным называется метод, который вызывает сам себя. Такая рекурсия называется прямой. Существует косвенная рекурсия, когда два или более метода вызывают друг друга. Если метод вызывает себя, в стеке создается копия значений его параметров, после чего управление передается первому исполняемому оператору метода. При повторном вызове этот процесс повторяется. Для завершения вычислений каждый рекурсивный метод должен содержать хотя бы одну нерекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении метода соответствующая часть стека освобождается, и управление передается вызывающему методу, выполнение которого продолжается с точки, следующей за рекурсивным вызовом.

Пример: Вычислить значение факториала.

Функцию факториала n! определяют как произведение первых n целых чисел:

n! = 1 * 2 * 3 * … * n

Такое произведение легко вычислить с помощью итеративных конструкций (например for).

Другое определение факториала, в котором используется рекуррентная формула, имеет вид:

(1) 0!=1

(2) для ∀ n>0 n!=n*(n-1)!

```
private static int fact ( int n ) // описание рекурсивного метода
       {
        if ( n <= 1 ) return 1 ; // нерекурсивная ветвь
        return ( n*fact (n - 1)) ; // метод вызывает сам себя
       }
```

Или

```
private static int fact ( int n ) // описание рекурсивного метода
      {
       return ( n > 1 ) ? n*fact ( n - 1 ) : 1 ;
      }
```

Пример: Для заданного числа вычислить число Фибоначчи.

Для чисел Фибоначчи рекурсивное определение выглядит следующим образом:

(1) F(1)=1,

(2) F(2)=1,

(3) для ∀ n>2 F(n)=F(n-1)+F(n-2)

```
class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Введите искомое число: ");
         string buf = Console.ReadLine();
         int n = Convert.ToInt32(buf);
         Console.WriteLine("Число Фибоначчи = {0}", Fib(n));
       }
      private static int Fib(int i)
      {
         if ((i == 1) || (i == 2)) return 1;
         else return Fib(i - 1) + Fib(i - 2);
      }
    }
}
```

Перегрузка методов

В C# несколько методов могут иметь одно и то же имя. В этом случае метод, идентифицируемый этим именем, называется перегруженным. Перегрузить можно только методы, которые отличаются либо типом, либо числом своих аргументов. Перегрузить методы, которые отличаются только типом возвращаемого значения, нельзя. Перегружаемые методы дают возможность упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Чтобы перегрузить некоторый метод, нужно объявить, а затем определить все требуемые варианты его вызова. Компилятор автоматически выберет правильный вариант вызова на основании числа и типа используемых аргументов.

На перегруженные методы накладываются несколько ограничений:

- любые два перегруженные методы должны иметь различные списки параметров;
- перегрузка методов с совпадающими списками аргументов на основе лишь типа возвращаемых ими значений недопустима;
- методы не могут быть перегружены исключительно на основе того, что один из них является статическим, а другой – нет;

- типы "массив" и "указатель" рассматриваются как идентичные с точки зрения перегрузки.

Пример:

```
class Program
   {
      static void Main(string[] args)
      {
         int mN, N = -255; float mF, F = -25.0f; double mD, D = -2.55;
         mN = abs(N); // вызов перегруженной функции abs ( int )
         mF = abs(F); // вызов перегруженной функции abs ( float )
         mD = abs(D); // вызов перегруженной функции abs ( double )
         Console.WriteLine("abs(int) \t| {0}\t| = {1}", N, mN);
         Console.WriteLine("abs(float) \t| {0}\t| = {1}", F, mF);
         Console.WriteLine("abs(double) \t| {0}\t| = {1}", D, mD);
      }
      public static int abs(int a)
        { return a < 0 ? -a : a; }
      public static float abs(float a)
        { return a < 0 ? -a : a; }
      public static double abs(double a)
        { return a < 0 ? -a : a; }
   }
```

Результат выполнения программы изображен на рисунке 2.11.

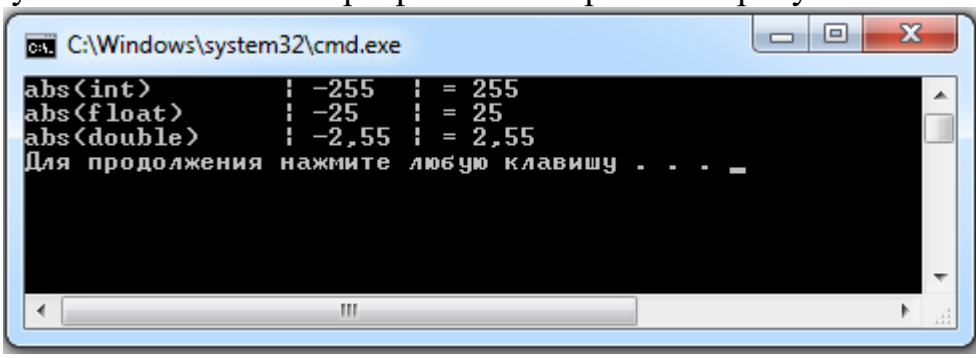


Рис. 2.11 Результат выполнения программы.

2.6Ключевое слово this

Каждый объект содержит свой экземпляр полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями 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 y;
    public Demo T() // метод возвращает ссылку на экземпляр
      { return this; }
    public void Sety(double y)
      { this.y = y; } // полю y присваивается значение параметра y
  }
```

2.7Конструкторы

Понятие конструктора

Конструкторы – это методы класса, которые вызываются при создании объекта.

Конструктор предназначен для инициализации объекта. Он вызывается автоматически при создании объекта класса с помощью операции **new**. Имя конструктора совпадает с именем класса. Конструкторы обладают следующими свойствами:

- Конструктор не возвращает значение, даже типа **void**.
- Класс может иметь несколько конструкторов с разными параметрами для разных 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**.
- Конструктор, вызываемый без параметров, называется ***конструктором по умолчанию***.

В C# существует 3 вида конструкторов:

- Конструктор по умолчанию.
- Конструктор с параметрами – конструктор, который может принимать необходимое количество параметров для инициализации полей класса или каких-либо других действий.
- Статический конструктор – конструктор, относящийся к классу, а не к объекту. Существует для инициализации статических полей класса. Определяется без какого-либо спецификатора доступа с ключевым словом **static**.

При создании конструкторов нужно помнить, что все конструкторы (кроме статического) имеют спецификатор доступа **public**, все конструкторы, кроме конструктора по умолчанию и статического конструктора, могут иметь необходимое количество параметров. Конструктор по умолчанию может быть только один. Если конструктор описан в классе, то конструктор по умолчанию, который вам предоставлялся компилятором, работать не будет.

Пример определения своего конструктора по умолчанию. Для работы с

конструкторами создан новый класс, описывающий машину.

```
class Car
  {
    private string driverName; // Имя водителя
    private int currSpeed; // Текущая скорость
    public Car() // Конструктор по умолчанию
    {
      driverName = "Михаель Шумахер";
      currSpeed = 10;
    }
    public void PrintState() // Распечатка текущих данных
    {
      Console.WriteLine("{0} едет со скоростью {1} км/ч.",
      driverName, currSpeed);
    }
    public void SpeedUp(int delta) // Увеличение скорости
    { currSpeed += delta; }
  }
  class Program
  {
    static void Main(string[] args)
    {
      Car myCar = new Car();
      for (int i = 0; i <= 10; i++)
      {
        myCar.SpeedUp(5);
        myCar.PrintState();
      }
    }
  }
```

Результат выполнения программы изображен на рисунке 2.12.

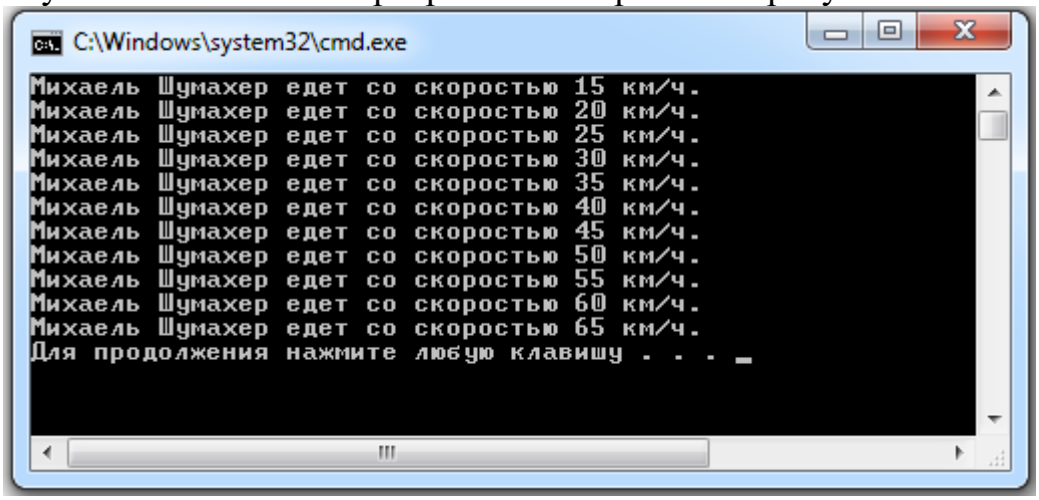


Рис. 2.12 Результат выполнения программы.

Конструктор с параметрами

Конструктор с параметрами отличается от конструктора по умолчанию наличием параметров. Количество параметров определяет количество полей, которые необходимо проинициализировать при создании объекта.

Пример: Добавить в класс Car конструктор позволяющий инициализировать поле driverName.

```
class Car
{
	// Старые поля и методы...
	public Car(string name)
	{
		driverName = name;
		currSpeed = 10;
	}
}
class Program
{
	static void Main(string[] args)
	{
		Car myCar = new Car("Рубенс Барикелло");
		for (int i = 0; i <= 10; i++)
		{
			myCar.SpeedUp(5);
			myCar.PrintState();
		}
	}
}
```

Перегруженные конструкторы

Конструкторы, как и другие методы, можно перегружать: создавать в классе несколько конструкторов с различными списками параметров, чтобы обеспечить возможность инициализации объектов разными способами.

Пример: определим конструкторы с параметрами для класса Car.

```
class Car
{
	// Старые поля и методы...
	public Car(string name)
	{ driverName = name; currSpeed = 10; }
	public Car(string name, int speed)
	{ driverName = name; currSpeed = speed; }
}
class Program
{
```

```
        static void Main(string[] args)
        {
                Car myCar = new Car("Ральф Шумахер", 10);
                for (int i = 0; i <= 10; i++)
                {
                myCar.SpeedUp(5);
                myCar.PrintState();
                }
        }
}
```

Статические конструкторы

Статический конструктор связан с классом, а не с конкретным объектом. Сам конструктор нужен для инициализации статических данных. Когда вызывается этот конструктор – неизвестно, но гарантируется, что вызов произойдет до первого создания объекта класса. Для примера со статическим конструктором создадим класс описывающий банковские филиалы, но на этот раз статическое поле будет содержать бонус в процентах для оформления депозитов. А текущий баланс у каждого филиала будет свой.

```
class Bank
   {
      private double currBalance;
      private static double bonus;
      public Bank(double balance)
      { currBalance = balance; }
      static Bank()
      { bonus = 1.04; }
      public static void SetBonus(double newRate)
      { bonus = newRate; }
      public static double GetBonus()
      {  return bonus; }
      public double GetPercents(double summa)
      {
         if ((currBalance - summa) > 0)
         {
            double percent = summa * bonus;
            currBalance -= percent;
            return percent;
         }
         return -1;
      }
   }
   class Program
   {
```

```
        static void Main(string[] args)
        {
            Bank b1 = new Bank(1000000);
            Console.WriteLine("Текущий бонусный процент: " +
            Bank.GetBonus());
            Console.WriteLine("Ваш депозит на {0:C}, в кассе забрать:" +
            "{1:C}", 10000, b1.GetPercents(10000));
        }
    }
```

Результат выполнения программы изображен на рисунке 2.13.

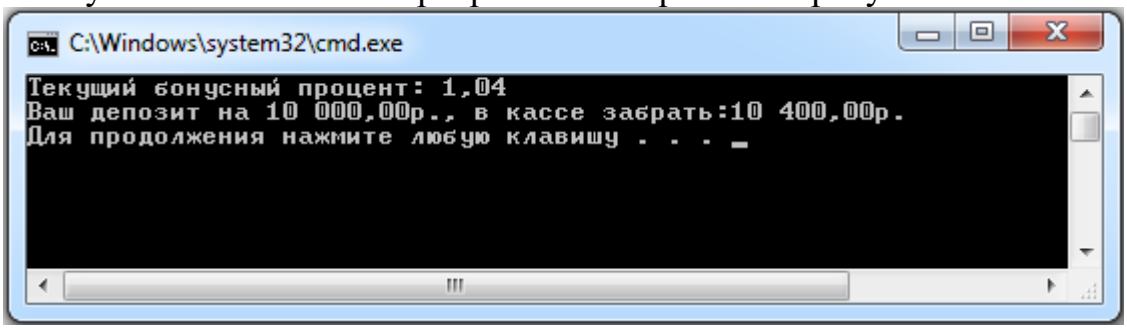


Рис. 2.13 Результат выполнения программы.

3 Контрольные вопросы

1. Что такое объект?
2. Что такое атрибут объекта?
3. Что такое поведение объекта?
4. В чем заключается принцип объектно-ориентированного программирования?
5. Почему важно использовать объектно-ориентированное программирование при разработке компьютерных систем?
6. Каким образом объектно-ориентированное программирование помогает поддерживать сложные компьютерные системы?
7. Определите атрибуты объекта Форма заказа на рис. 2.1.
8. Определите варианты поведения объекта Форма заказа на рис. 2.1.
9. Что такое метод?
10. Объясните роль наследования в объектно-ориентированном программировании.
11. Объясните роль инкапсуляции в объектно-ориентированном программировании.
12. Объясните понятие инкапсуляции в объектно-ориентированном программировании.
13. Что понимается под термином «класс»?
14. Какие элементы определяются в составе класса?
15. Каково соотношение понятий «класс» и «объект»?
16. Что понимается под термином «члены класса»?

17. Какие члены класса Вам известны?
18. Какие члены класса содержат код?
19. Какие члены класса содержат данные?
20. Перечислите пять разновидностей членов класса специфичных для языка C#.
21. Что понимается под термином «конструктор»?
22. Сколько конструкторов может содержать класс?
23. Приведите синтаксис описания класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
24. Какие модификаторы типа доступа Вам известны?
25. В чем заключаются особенности доступа членов класса с модификатором public?
26. В чем заключаются особенности доступа членов класса с модификатором private?
27. В чем заключаются особенности доступа членов класса с модификатором protected?
28. В чем заключаются особенности доступа членов класса с модификатором internal?
29. Какое ключевое слово языка C# используется при создании объекта?
30. Приведите синтаксис создания объекта в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
31. В чем состоит назначение конструктора?
32. Каждый ли класс языка C# имеет конструктор?
33. Какие умолчания для конструкторов приняты в языке C#?
34. Каким значением инициализируются по умолчанию значения ссылочного типа?
35. В каком случае конструктор по умолчанию не используется?
36. Приведите синтаксис конструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
37. Что понимается под термином «деструктор»?
38. В чем состоит назначение деструктора?
39. Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
40. Что такое рекурсия?
41. Что такое статистический метод?

4 Задание

5. Написать программу в соответствии с вариантом задания. Для класса предусмотреть конструктор по умолчанию, несколько конструкторов по с параметрами, деструктор, методы: изменения, отображения полей класса и методы согласно задания. Для

хранения объектов класса использовать динамический массив или стандартный список List. Описание класса и методов класса должны находиться в отдельном модуле.
6. Отладить и протестировать программу.
7. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Вариант 1.
Описать класс с именем WORKER, содержащий поля:
- фамилия и инициалы работника;
- название занимаемой должности;
- зарплату;
- год поступления на работу.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа WORKER;
- вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
- если таких работников нет, вывести соответствующее сообщение.

Вариант 2.
Описать класс с именем TRAIN, содержащий поля:
- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа TRAIN;
- вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
- если таких поездов нет, вывести соответствующее сообщение.

Вариант 3.
Описать класс с именем TRAIN, содержащий поля:
- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа TRAIN;
- вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры;
- если таких поездов нет, вывести соответствующее сообщение.

Вариант 4.
Описать класс с именем MARSH, содержащий поля:

– название начального пункта маршрута;
– название конечного пункта маршрута;
– номер маршрута.

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа MARSH;
– вывод на экран информации о маршруте, номер которого введен с клавиатуры;
– если таких маршрутов нет, вывести соответствующее сообщение.

Вариант 5.

Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 6.

Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на год, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 7.

Описать класс с именем NOTE, содержащий поля:
– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения приходятся на день, значение которого введено с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 8.

Описать класс с именем NOTE, содержащий поля:

– фамилия и имя;
– номер телефона;
– дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа NOTE;
– вывод на экран информации о людях, чьи дни рождения совпадают с введенными с клавиатуры;
– если таких людей нет, вывести соответствующее сообщение.

Вариант 9.

Описать класс с именем ORDER, содержащий поля:
– расчетный счет плательщика;
– расчетный счет получателя;
– перечисляемая сумма в руб.

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа ORDER;
– вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
– если таких счетов нет, вывести соответствующее сообщение.

Вариант 10.

Описать класс с именем STUDENT, содержащий поля:
– фамилия и инициалы;
– номер группы;
– успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа STUDENT;
– вывод на дисплей фамилий и номеров групп для всех студентов, если средний балл студента больше 4.0;
– если таких студентов нет, вывести соответствующее сообщение.

Вариант 11.

Описать класс с именем STUDENT, содержащий поля:
– фамилия и инициалы;
– номер группы;
– успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:
– ввод с клавиатуры данных класса типа STUDENT;
– вывод на дисплей фамилий и номеров групп для всех студентов, если они имеют оценки 4 и 5;
– если таких студентов нет, вывести соответствующее сообщение.

Вариант 12.

Описать класс с именем STUDENT, содержащий поля:
- фамилия и инициалы;
- номер группы;
- успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа STUDENT;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 13.

Описать класс с именем AEROFLOT, содержащий поля:
- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа AEROFLOT;
- вывод на экран информации о рейсе, номер которого введен с клавиатуры;
- если таких рейсов нет, вывести соответствующее сообщение.

Вариант 14.

Описать класс с именем AEROFLOT, содержащий поля:
- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа AEROFLOT;
- вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
- если таких рейсов нет, вывести соответствующее сообщение.

Вариант 15.

Описать класс с именем NOTE, содержащий поля:
- фамилия и имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа NOTE;
- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
- если таких людей нет, вывести соответствующее сообщение.

Вариант 16.

Описать класс с именем WORKER, содержащий поля:
- фамилия и инициалы работника;
- название занимаемой должности;
- зарплату;
- год поступления на работу.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа WORKER;
- вывод на дисплей фамилий работников, чья зарплата выше средней по организации и стаж работы больше трех лет;
- если таких работников нет, вывести соответствующее сообщение.

Вариант 17.

Описать класс с именем TRAIN, содержащий поля:
- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа TRAIN;
- вывод на экран информации о поездах, отправляющихся в пункт назначения введенного с клавиатуры;
- если таких поездов нет, вывести соответствующее сообщение.

Вариант 18.

Описать класс с именем TRAIN, содержащий поля:
- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа TRAIN;
- вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры;
- если таких поездов нет, вывести соответствующее сообщение.

Вариант 19.

Описать класс с именем MARSH, содержащий поля:
- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Написать программу, выполняющую следующие действия:
- ввод с клавиатуры данных класса типа MARSH;
- вывод на экран информации о маршруте, номер которого введен с клавиатуры;
- если таких маршрутов нет, вывести соответствующее сообщение.

Вариант 20.

Описать класс с именем NOTE, содержащий поля:

- фамилия и имя;
- номер телефона;
- дата рождения (массив из трех чисел).

Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных класса типа NOTE;
- вывод на экран информации о людях, чьи дни рождения приходятся на год, значение которого введено с клавиатуры;
- если таких людей нет, вывести соответствующее сообщение.

Лабораторная работа № 3

1 Цель и порядок работы

Цель работы – ознакомиться с понятием перегрузки операторов, понятием свойств и индексаторов в языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Перезагрузка операторов

Введение в перегрузку операторов

Перегрузка операторов позволяет указать, как стандартные операторы будут использоваться с объектами класса. Перегрузка

Требования к перегрузке операторов:

- перегрузка операторов должна выполняться открытыми статическими методами класса (спецификаторы public static);
- у метода - оператора тип возвращаемого значения или одного из параметров должен совпадать с типом, в котором выполняется перегрузка оператора;
- параметры метода - оператора не должны включать модификатор out и ref.

Таким образом, невозможно изменить значение стандартных операций для стандартных типов данных.

Таблица 1. Операторы, допускающие перегрузку.

Операторы	**Категория операторов**
-	Изменение знака переменной
!	Операция логического отрицания
~	Операция побитового дополнения, которая приводит к инверсии каждого бита
++, --	Инкремент и декремент
true, false	Критерий истинности объекта, определяется разработчиком класса
+, –, *, /, %	Арифметические операторы
&, |, ^, <<, >>	Битовые операции
==, !=, <, >, <=, >=	Операторы сравнения
&&, ||	Логические операторы
[]	Операции доступа к элементам массивов моделируются за счет индексаторов
()	Операции преобразования

Таблица 2. Операторы, не допускающие перегрузку.

Операторы	Категория операторов
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Перегружаются автоматически при перегрузке соответствующих бинарных операций
=	Присвоение
.	Доступ к членам типа
?:	Оператора условия
new	Создание объекта
as, is, typeof	Используются для получения информации о типе
–>, sizeof,*,&	Доступны только в небезопасном коде

Перегрузка операторов имеет некоторые ограничения:

- перегрузка не может изменить приоритет операторов;

- при перегрузке невозможно изменить число операндов, с которыми работает оператор;
- не все операторы можно перегружать (таблицы 1 и 2).

Перегрузку операторов можно использовать как в классах, так и в структурах.

Поскольку перегруженные операторы являются статическими методами, они не получают указателя this, поэтому унарные операторы должны получать 1 параметр, бинарные 2.

Синтаксис перегрузки:

public static <тип результата>

operator <символ операции> (параметры)

Перегрузка унарных операторов

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 передать его в качестве результата.

Пример операторов инкремента, декремента и изменения знака -.

Класс Point описывает точку на плоскости, точка имеет координаты x и y. Оператор ++ увеличивает обе координаты на 1, оператор -- уменьшает, оператор – изменяет знак координат на противоположный.

```
namespace UnaryOperator
  {
   //класс точки на плоскости – пример для перегрузки операторорв
   class CPoint
    {
     int x, y;
     public CPoint(int x, int y)
       { this.x = x; this.y = y; }
     //перегрузка инкремента
     public static CPoint operator ++(CPoint s)
       { s.x++; s.y++; return s;}
     //перегрузка декремента
     public static CPoint operator --(CPoint s)
       { s.x--; s.y--; return s; }
     //перегрузка оператора -
     public static CPoint operator -(CPoint s)
       {
        CPoint p = new CPoint(s.x, s.y);
        p.x = -p.x; p.y = -p.y; return p;
       }
     public override string ToString()
       {
        return string.Format("X = {0} Y = {1}", x, y);
```

```
    }
  }
 class Test
  {
   static void Main()
    {
     CPoint p = new CPoint(10, 10);
     //префиксная и постфиксная формы выполняются одинаково
     Console.WriteLine(++p); //x=11, y=11
     CPoint p1 = new CPoint(10, 10);
     Console.WriteLine(p1++); //x=11, y=11
     Console.WriteLine(--p); //x=10, y=10
     Console.WriteLine(-p); //x=-10, y=-10
     //после выполнения оператора –
     //состояние исходного объекта не изменилось
     Console.WriteLine(p); //x=10, y=10
    }
  }
```

Результат выполнения программы изображен на рисунке 2.1.

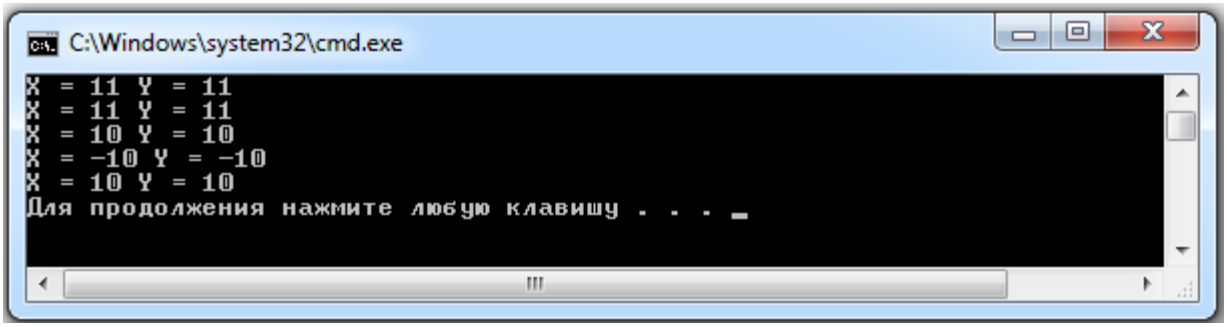


Рис. 2.1 Результат выполнения программы.

В данном примере CPoint является ссылочным типом, поэтому изменения значений x и y, которые выполняются в перегруженных операторах инкремента и декремента, изменяют переданный в них объект. Оператор – (изменение знака) не должен изменять состояние переданного

объекта, а должен возвращать новый объект с измененным знаком. Для этого в реализации этого метода создается новый объект CPoint, изменяется знак его координат и этот объект возвращается из метода.

В C# нет возможности выполнить отдельно перегрузку постфиксной и префиксной форм операторов инкремента и декремента. Поэтому при вызове постфиксная и префиксная форма работают одинаково.

При перегрузке операторов true и false разработчик задает критерий истинности для своего типа данных. После этого объекты типа напрямую можно использовать в структуре операторов if, do, while, for в качестве условных выражений.

Перегрузка выполняется по следующим правилам:

- оператор true должен возвращать значение true, если состояние объекта истинно и false в противном случае;
- оператор false должен возвращать значение true, если состояние объекта ложно и false в противном случае;
- операторы true и false надо перегружать в паре.

При этом возможна ситуация, когда состояние не является не истинным ни ложным, т.е. оба оператора могут вернуть результат false.

При перегрузке операторов true и false используется следующая таблица истинности (табл. 3).

Таблица 3. Таблица истинности при перегрузке операторов true и false.

Значение	Оператор True	Оператор False
1	true	False
-1	false	True
0	false	false

```
public struct DBBool
   {
```

```
    //три возможных значения
    // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    sbyte value;
    DBBool(int value)
    {  this.value = (sbyte)value; }
    public DBBool(DBBool b)
    { this.value = (sbyte)b.value; }
    // Возвращает true, если в операнде содержится True,
    // иначе возвращает false
    public static bool operator true(DBBool x)
    { return x.value > 0; }
    // Возвращает true, если в операнде содержится False,
    // иначе возвращает false
    public static bool operator false(DBBool x)
    { return x.value < 0; }
  }
  class Test
  {
    static void Main()
    {
      DBBool b1 = new DBBool(DBBool.True);
      if (b1) Console.WriteLine("b1 is true");
      else Console.WriteLine("b1 is not true");
    }
  }
```

Как видно из реализации, если в объекте класса содержится значение

null, то оба оператора возвращают значение false.

Перегрузка бинарных операторов

Можно определять в классе следующие бинарные операции:

+ - * / % & ≪ ≫ == != > < >= <=

Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, 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 возвращают 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.

Пример перегрузки бинарных операций:

```
namespace BinaryOperator
 {
  class CVector
  {
    public int x;
    public int y;
    public CVector(int x, int y)
     { this.x = x; this.y = y; }
    public override string ToString()
     { return string.Format("Vector: X = {0} Y = {1}", x, y); }
  }
```

```
class CPoint
 {
   private int x;
   private int y;
   public CPoint(int x, int y)
    { this.x = x; this.y = y; }
   //перегрузка бинарного оператора +
   public static CPoint operator +(CPoint p, CVector v)
    { return new CPoint(p.x + v.x, p.y + v.y); }
   //перегрузка бинарного оператора *
   public static CPoint operator *(CPoint p, int a)
    { return new CPoint(p.x * a, p.y * a); }
   //перегрузка бинарного оператора -
   public static CVector operator -(CPoint p1, CPoint p2)
    { return new CVector(p1.x - p2.x, p1.y - p2.y); }
   public override string ToString()
    { return string.Format("Point: X = {0} Y = {1}", x, y); }
 }
class Program
 {
  static void Main(string[] args)
  {
     CPoint p1 = new CPoint(10, 10);
     CPoint p2 = new CPoint(12, 20);
     CVector v = new CVector(10, 20);
     Console.WriteLine("Точка p1: {0}", p1);
     Console.WriteLine("Сдвиг: {0}", p1 + v);
     Console.WriteLine("Масштабирование: {0}", p1 * 10);
     Console.WriteLine("Точка p2: {0}", p2);
```

```
        Console.WriteLine("Расстояние: {0}", p2 - p1);
    }
   }
  }
```

Результат выполнения программы изображен на рисунке 2.2.

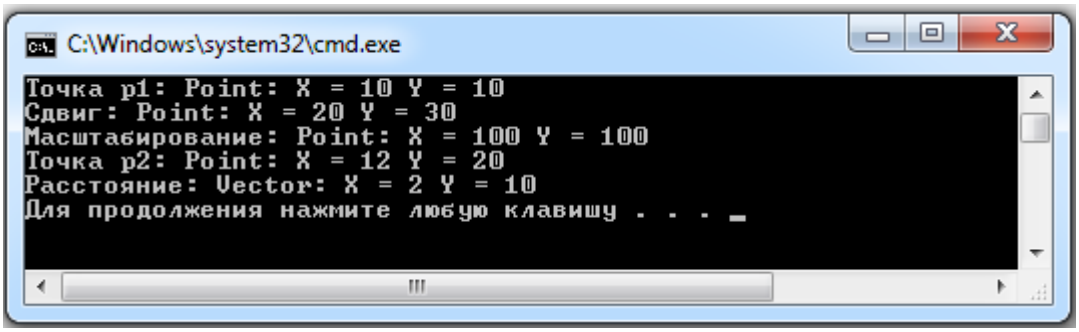


Рис. 2.2 Результат выполнения программы.

Выполненная перегрузка автоматически перегружает операторы +=, *=, -=. Например, можно записать: p1+= v;

Для реализации этого действия будет использован перегруженный оператор +.

Однако перегруженные в примере операторы будут использоваться компилятором только если переменная типа CPoint находится слева от знака операнда. Т.е. выражение p * 10 откомпилируется нормально, а при перестановке сомножителей, т.е. в выражении 10 * p произойдет ошибка компиляции. Для исправления этой ошибки следует перегрузить оператор * с другим порядком операндов:

```
public static CPoint operator *(int a, CPoint p)
{ return p * a; }
```

При перегрузке операторов отношения надо учитывать, что есть два способа проверки равенства:

- равенство ссылок (тождество);
- равенство значений.

В классе Object определены следующие методы сравнения объектов:

```
public static bool ReferenceEquals(Object obj1, Object obj2)
```

public bool virtual Equals(Object obj)

Есть отличия в работе этих методов со значимыми и ссылочными типами.

Метод ReferenceEquals() проверяет, указывают ли две ссылки на один и тот же экземпляр класса; точнее - содержат ли две ссылки один и тот адрес памяти. Этот метод невозможно переопределить. Со значимыми типами ReferenceEquals() всегда возвращает false, т.к. при сравнении выполняется приведение к Object и упаковка, упакованные объекты располагаются по разным адресам.

Метод Equals() является виртуальным. Его реализация в Object выполняется проверку равенства ссылок, т.е. работает так же как и ReferenceEquals. Для значимых типов в базовом типе System.ValueType выполнена перегрузка метода Equals(), которая выполняет сравнение объектов путем сравнения всех полей (побитовое сравнение).

Пример использования операторов ReferenceEquals() и Equals() со ссылочными и значимыми типами:

```
namespace Equals_and_ReferenceEquals
  {
    class CPoint
    {   private int x, y;
      public CPoint(int x, int y)
      { this.x = x; this.y = y; }
    }
    struct SPoint
    {
      private int x, y;
      public SPoint(int x, int y)
      { this.x = x; this.y = y; }
    }
```

```
class Program
{
   static void Main()
   {
//Работа метода ReferenceEquals с ссылочным и значимым типами
      //ссылочный тип
      CPoint p = new CPoint(0, 0);
      CPoint p1 = new CPoint(0, 0);
      CPoint p2 = p1;
      Console.WriteLine("ReferenceEquals(p, p1)= {0}",
      ReferenceEquals(p, p1)); //false
      //хотя p,p1содержат одинаковые значения,
      //они указывают на разные адреса памяти
      Console.WriteLine("ReferenceEquals(p1, p2)= {0}",
      ReferenceEquals(p1, p2));//true
      //p1 и p2 указывают на один и тот же адрес памяти
      //значимый тип
      SPoint p3 = new SPoint(0, 0);
  //при передаче в метод ReferenceEquals выполняется упаковка,
  //упакованные объекты располагаются по разным адресам
      Console.WriteLine("ReferenceEquals(p3, p3) = {0}",
      ReferenceEquals(p3, p3)); //false
      //Работа метода Equals с ссылочным и значимым типами
      //ссылочный тип
      CPoint cp = new CPoint(0, 0);
      CPoint cp1 = new CPoint(0, 0);
   Console.WriteLine("Equals(cp, cp1) = {0}", Equals(cp, cp1)); //false
      //выполняется сравнение адресов значимый тип
      SPoint sp = new SPoint(0, 0);
```

```
            SPoint sp1 = new SPoint(0, 0);
        Console.WriteLine("Equals(sp, sp1) = {0}", Equals(sp, sp1)); //true
            //выполняется сравнение значений полей
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.3.

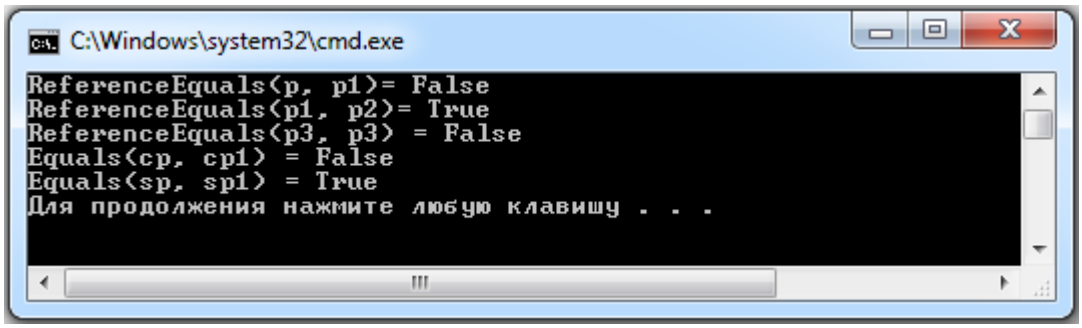


Рис. 2.3 Результат выполнения программы.

При создании собственного типа оператор Equals() можно перегрузить. Для ссылочных типов перегрузку следует выполнять, только если тип представляет собой неизменяемый объект. Например, для типа String, который содержит в себе неизменную строку, имеется перегруженный метод Equals() и оператор ==.

Поскольку в System.ValueType перегруженный метод Equals() выполняет побитовое сравнение, то в собственных значимых типах его можно не перегружать. Однако, в System.ValueType получение значений полей для сравнения в методе Equals() выполняется с помощью рефлексии, что приводит к снижению производительности. Поэтому при разработке значимого типа для увеличения быстродействия рекомендуется выполнить перегрузку метода Equals().

При перегрузке метода Equals() следует также перегружать метод GetHashCode(). Этот метод предназначен для получения целочисленного значения хеш - кода объекта. Причем, различным (т.е. не равным между собой) объектам должны соответствовать различные хеш – коды. Если

перегрузку метода GetHashCode() не выполнить возникнет предупреждение компилятора.

Перегрузка оператора == обычно выполняется путем вызова метода Equals().

Если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется унаследовать этот тип от интерфейсов System.IComparable и System.IComparable<T> и реализовать метод CompareTo(). В дальнейшем этот метод можно вызывать из реализации Equals() и возвращать true, если CompareTo() возвращает 0.

Пример перегрузки операторов == и != для класса CPoint:

```
namespace ComparisonOperator
  {
    using System;
    class CPoint
    {
      int x, y;
      public CPoint(int x, int y)
      { this.x = x; this.y = y; }
      //Перегрузка метода Equals
      public override bool Equals(object obj)
      {
        //если obj == null,
  //значит он != объекту, от имени которого вызывается этот метод
          if (obj == null) return false;
          CPoint p = obj as CPoint;
          //переданный объект не является ссылкой на CPoint
          if (p == null) return false;
          //проверяется равенство содержимого
          return ((x == p.x) && (y == p.y));
```

```
    }
//При перегрузке Equals надо также перегрузить GetHashCode()
    public override int GetHashCode()
    {
        return x ^ y;//использование XOR для получения хеш кода
    }
    public static bool operator ==(CPoint p1, CPoint p2)
    {
//проверка, что переменные ссылаются на один и тот же адрес
        //сравнение p1 == p2 приведет к бесконечной рекурсии
        if (ReferenceEquals(p1, p2)) return true;
        //приведение к object необходимо,
//т.к. сравнение p1 == null приведет к бесконечной рекурсии
        if ((object)p1 == null) return false;
        return p1.Equals(p2);
    }
    public static bool operator !=(CPoint p1, CPoint p2)
    { return (!(p1 == p2)); }
}
class Program
{
    static void Main(string[] args)
    {
        CPoint cp = new CPoint(0, 0);
        CPoint cp1 = new CPoint(0, 0);
        CPoint cp2 = new CPoint(1, 1);
        Console.WriteLine("cp == cp1: {0}", cp == cp1); //true
        Console.WriteLine("cp == cp1: {0}", cp == cp2); //false
    }
```

```
        }
    }
```

Условные логические операторы && и || нельзя перегрузить, но они вычисляются с помощью & и |, допускающих перегрузку.

Пример класса DBBool использующего перегрузку логических операторов:

```
public struct DBBool
    {
        //три возможных значения
        // Значение параметра может быть: - 1(false), 1 - true и 0 - null.
        public static readonly DBBool Null = new DBBool(0);
        public static readonly DBBool False = new DBBool(-1);
        public static readonly DBBool True = new DBBool(1);
        sbyte value;
        DBBool(int value)
        { this.value = (sbyte)value; }
        public DBBool(DBBool b)
        { this.value = (sbyte)b.value; }
        // Возвращает true, если в операнде содержится True,
        // иначе возвращает false
        public static bool operator true(DBBool x)
        { return x.value > 0; }
        // Возвращает true, если в операнде содержится False,
        // иначе возвращает false
        public static bool operator false(DBBool x)
        { return x.value < 0; }
        // Оператор Логическое И. Возвращает:
        // False, если один из операндов False независимо
        // от 2-ого операнда
```

```
    // Null, если один из операндов Null, а другой Null или True
    // True, если оба операнда True
    public static DBBool operator &(DBBool x, DBBool y)
    { return new DBBool(x.value < y.value ? x.value : y.value); }
    // Оператор Логическое ИЛИ. Возвращает:
    // True, если один из операндов True независимо
   // от 2-ого операнда
    // Null, если один из операндов Null, а другой False или Null
    // False, если оба операнда False
    public static DBBool operator |(DBBool x, DBBool y)
    { return new DBBool(x.value > y.value ? x.value : y.value); }
    // Оператор Логической инверсии. Возвращает:
    // False, если операнд содержит True
    // True, если операнд содержит False
    // Null, если операнд содержит Null
    public static DBBool operator !(DBBool x)
    { return new DBBool(-x.value); }
    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
class Test
{
    static void Main()
    {
        DBBool bTrue = new DBBool(DBBool.True);
```

```
            DBBool bNull = new DBBool(DBBool.Null);
            DBBool bFalse = new DBBool(DBBool.False);
            Console.WriteLine("bTrue && bNull is {0}", bTrue && bNull);
            Console.WriteLine("bTrue && bFalse is {0}", bTrue && bFalse);
            Console.WriteLine("bTrue && bTrue is {0}", bTrue && bTrue);
            Console.WriteLine();
            Console.WriteLine("bTrue || bNull is {0}", bTrue || bNull);
            Console.WriteLine("bFalse || bFalse is {0}", bFalse || bFalse);
            Console.WriteLine("bTrue || bFalse is {0}", bTrue || bFalse);
            Console.WriteLine();
            Console.WriteLine("!bTrue is {0}", !bTrue);
            Console.WriteLine("!bFalse is {0}", !bFalse);
            Console.WriteLine("!bNull is {0}", !bNull);
        }
    }
```

Результат выполнения программы изображен на рисунке 2.4.

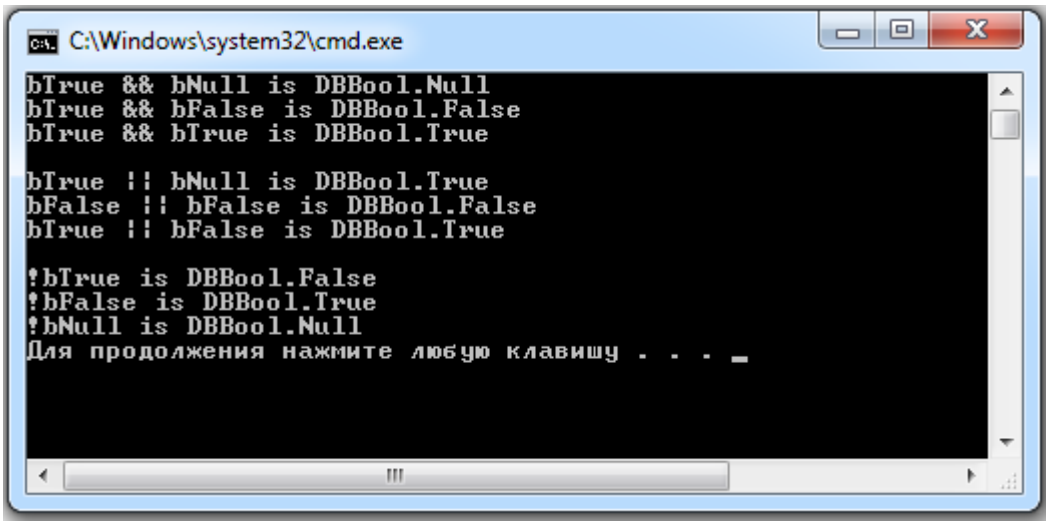


Рис. 2.4 Результат выполнения программы.

Перегрузка операторов преобразования типа

Операции преобразования типа обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных.

Синтаксис объявителя операции преобразования типа:

implicit operator тип (параметр) // неявное преобразование

explicit operator тип (параметр) // явное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого определяется операция. Таким образом, операции выполняют преобразования либо типа класса к другому типу, либо наоборот. Преобразуемые типы не должны быть связаны отношениями наследования.

Конструктор с 1-им параметром не используется для преобразования произвольного типа в собственный тип. Для ссылочных и значимых типов приведение выполняется одинаково.

Приведение может выполняться явным и неявным образом. Явное приведение типов требуется, если возможна потеря данных в результате приведения. Например:

- при преобразовании int в short, потому что размер short недостаточен для сохранения значения int;
- при преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение;
- при конвертировании типов с плавающей точкой в целые дробная часть теряется;
- при конвертировании типа, допускающего null-значения, в тип, не допускающий null, если исходная переменная содержит null, генерируется исключение.

Если потери данных в результате приведения не происходит приведение можно выполнять как неявное.

Пример:

Приведение CPoint к типу int с потерей точности, к типу double без потери точности. В качестве результата возвращается расстояние от точки до

начала координат.

Приведение типа int к CPoint без потери точности, типа double к CPoint с потерей точности. В качестве результата возвращается точка, содержащее заданное значение в качестве значений координат.

```
namespace CastOperator
  {
    class CPoint
    {
      int x, y;
      public CPoint(int x, int y)
      { this.x = x; this.y = y; }
//может быть потеря точности, преобразование должно быть явным
      public static explicit operator int(CPoint p)
      {
        return (int)Math.Sqrt(p.x * p.x + p.y * p.y);
        //можно и так: return (int)(double)p;
      }
      //преобразование без потери точности, может быть неявным
      public static implicit operator double(CPoint p)
      { return Math.Sqrt(p.x * p.x + p.y * p.y); }
      //переданное значение сохраняется в x и y координате,
      //преобразование без потери точности, может быть неявным
      public static implicit operator CPoint(int a)
      { return new CPoint(a, a); }
      //преобразование с потерей точности, должно быть явным
      public static explicit operator CPoint(double a)
      {
        return new CPoint((int)a, (int)a);
       }
```

```
        public override string ToString()
        {
          return string.Format("X = {0} Y = {1}", x, y);
         }
    }
    class Test
    {
        static void Main()
        {
          CPoint p = new CPoint(2, 2);
           //выполнение явного преобразования CPoint в int
           int a = (int)p;
           //выполнение неявного преобразования CPoint в double
           double d = p;
           Console.WriteLine("p as int: {0}", a); //2
           Console.WriteLine("p as double: {0:0.0000}", d);      //2.8284
           p = 5; //выполнение неявного преобразования int в CPoint
           Console.WriteLine("p: {0}", p); //x = 5 y = 5
          //выполнение явного преобразования double в CPoint
           p = (CPoint)2.5;
           Console.WriteLine("p: {0}", p); //x = 2 y = 2
        }
    }
}
```

Результат выполнения программы изображен на рисунке 2.5.

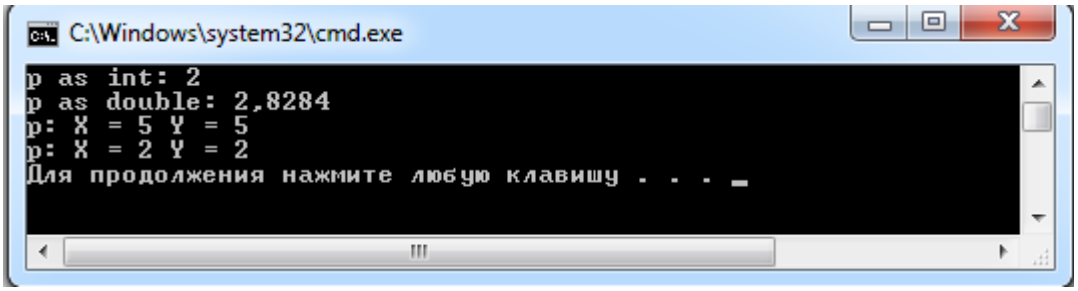


Рис. 2.5 Результат выполнения программы.

Имеется возможность выполнять приведение между экземплярами разных собственных структур или классов. Однако при этом существуют следующие ограничения:

- нельзя определить приведение между классами, если один из них является наследником другого;
- приведение может быть определено только в одном из типов: либо в исходном типе, либо в типе назначения.

Например, имеется следующая иерархия классов (рис. 2.6).

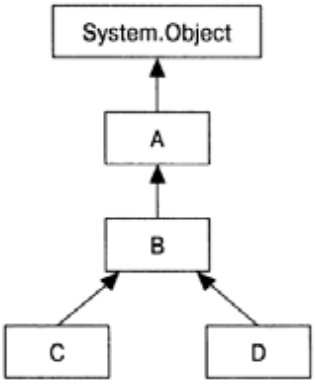


Рис. 2.6 Пример иерархии классов.

Единственное допустимое приведение типов – это приведения между классами C и D, потому что эти классы не наследуют друг друга. Код таких приведений может выглядеть следующим образом:

```
public static explicit operator D(C value) {…}
public static explicit operator C(D value) {…}
```

Эти операторы могут быть внутри определения класса C или же внутрь определения класса D. Если приведение определено внутри одного класса, то нельзя определить такое же приведение внутри другого.

2.2 Свойства

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя_свойства

{

[get код_доступа]

[set код_доступа]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public), поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении (**get**) или установке (**set**) свойства. Может отсутствовать либо часть **get**, либо **set**, но не обе одновременно.

Если отсутствует часть **set**, свойство доступно только для чтения (read-only), если отсутствует часть **get**, свойство доступно только для записи (write-only).

В C# введена удобная возможность задавать разные уровни доступа для частей **get** и **set**. Например, во многих классах возникает потребность обеспечить неограниченный доступ для чтения и ограниченный — для записи.

Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как public, его части могут иметь любой спецификатор доступа, а если свойство имеет доступ protected internal, его части могут объявляться как internal, protected или private. Синтаксис свойства имеет вид

[атрибуты] [спецификаторы] тип имя_свойства

{

[[атрибуты] [спецификаторы] get код_доступа]

[[атрибуты] [спецификаторы] set код_доступа]

}

Пример описания свойств:

```
public class Button : Control
   {
      // закрытое поле, с которым связано свойство
      private string caption;
      public string Caption
      { // свойство
         get
         { // способ получения свойства
            return caption;
         }
         set
         { // способ установки свойства
            if (caption != value)
            {
               caption = value;
            }
         }
      }
…..
   }
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод **get** должен содержать оператор **return**, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе **set** используется параметр со стандартным именем **value**, который содержит устанавливаемое значение.

Вообще говоря, свойство может и не связываться с полем. Фактически, оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство. В отличие от открытых полей, свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом и, таким образом, упрощают внесение изменений в класс.

С помощью свойств можно отложить инициализацию поля до того момента, когда оно фактически потребуется, например:

```
class A
   {
      private static ComplexObject x;// закрытое поле
      public static ComplexObject X  // свойство
      {
         get
         {
            if (x == null)
            {
               // создание объекта при 1-м обращении
                x = new ComplexObject();
            }
            return x;
         }
      }
…….
   }
```

Пример: Класс Employee реализован с четырьмя приватными полями, обозначающими имя, фамилию, возраст и зарплату сотрудника. Класс также включает два перегруженных конструктора (с параметрам и без параметров). Для каждого из полей класса предусмотрено открытое свойство с двумя

методами get и set. В свойствах осуществляется дополнительная проверка задаваемых значений. Имя и фамилия приводятся к верхнему регистру, возраст проверяется на принадлежность интервалу допустимых значений, зарплат не может быть отрицательной величиной. Перегруженный метод ToString() позволяет распечатать состояние объекта.

```
class Emoloyee
   {
      private string firstName;
      private string lastName;
      private int age;
      private float wage;
      public Emoloyee()
      {
      }
      public Emoloyee(string first, string last, int age, float wage)
      {
         this.FirstName = first;
         this.LastName = last;
         this.Age = age;
         this.Wage = wage;
      }
      public string FirstName
      {
         get { return firstName != null ? firstName : "Not set"; }
         set { firstName = value.ToUpper(); }
      }
      public string LastName
      {
```

```
        get { return lastName != null ? lastName : "Not set"; ; }
        set { lastName = value.ToUpper(); }
    }
    public int Age
    {
        get { return age; }
        set { age = (value > 100 || value < 1) ? 0 : value; }
    }
    public float Wage
    {
        get { return wage; }
        set { wage = value < 0 ? 0 : value; }
    }
    public override string ToString()
    {
        return string.Format("First name: {0}\nLast name:
                                        {1}\nAge:{2}\nWage: {3}\n",
        this.FirstName, this.LastName, this.Age, this.Wage);
    }
}
class Tester
{
    static void Main(string[] args)
    {
        Emoloyee emp1 = new Emoloyee("Oleg", "Sikorsky", 29,
                                                    4800F);
        Emoloyee emp2 = new Emoloyee();
        emp2.FirstName = "Daniel";
        //Last name не установлено
```

```
            //попытка присвоить невозможный возраст
            emp2.Age = 120;
            //попытка задать зарплату со знаком минус
            emp2.Wage = -1000;
            Emoloyee emp3 = new Emoloyee("Natali", "Borisova", 29,
                                                              2500F);
            Console.WriteLine(emp1.ToString());
            Console.WriteLine(emp2.ToString());
            Console.WriteLine(emp3.ToString());
        }
    }
```

Результат выполнения программы изображен на рисунке 2.7.

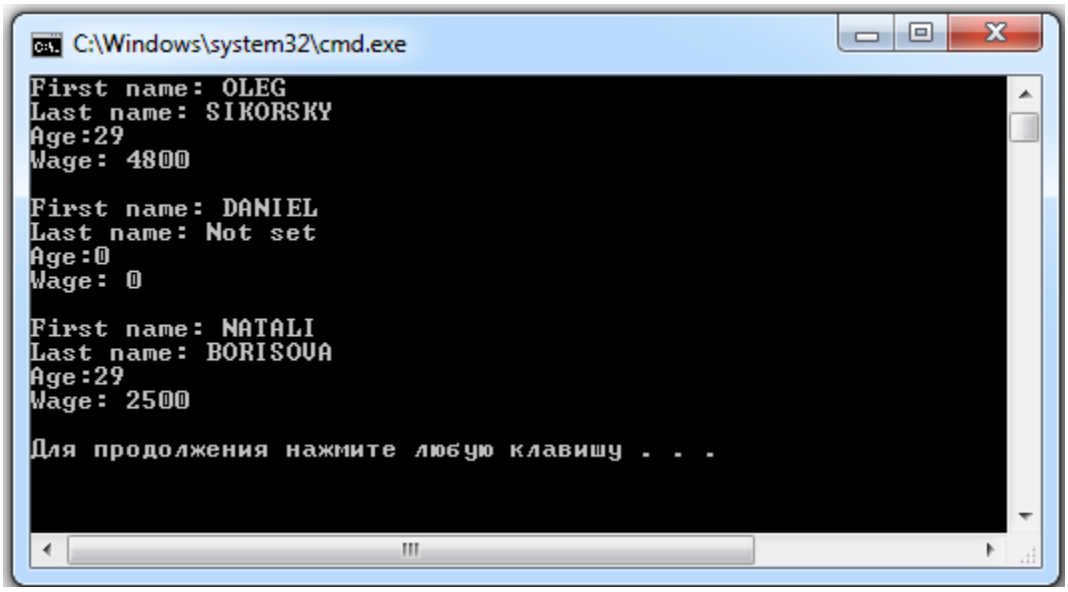


Рис. 2.7 Результат выполнения программы.

2.3 Индексаторы

Понятие индексатора

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.

Объявление индексатора подобно свойству, но с той разницей, что индексаторы безымянные (вместо имени используется ссылка this) и что индексаторы включают параметры индексирования.

Синтаксис объявления индексатора следующий:

тип this [тип аргумента] {get; set;}

Тип – это тип объектов коллекции. This - это ссылка на объект, в котором появляется индексатор. Тип аргумента представляет индекс объекта в коллекции, причём этот индекс необязательно целочисленный, как мы привыкли, он может быть любого типа.

У каждого индексатора должен быть минимум один параметр, но их может быть и больше (напоминает многомерные массивы).

То, что для индексаторов используется синтаксис со ссылкой this, подчёркивает, что нельзя использовать их иначе, как на экземплярном уровне.

Создание одномерного индексатора

Пример создания и применения индексатора. Предположим, есть некий магазин, занимающийся реализацией ноутбуков. Эта ситуация отображается при помощи двух классов: класса Shop, изображающего магазин, и класса Laptop, изображающего его продукцию. Дабы не перегружать пример лишней информацией, снабдим класс Laptop только двумя полями: vendor – для отображения имени фирмы-производителя, а также price – для отображения цены ноутбука. Класс будет включать соответствующие открытые свойства Vendor и Price, конструктор с двумя параметрами, а также переопределённый метод ToString() для отображения информации по конкретной единице товара. В качестве единственного поля класса Shop выступает ссылка на массив объектов Laptop. В конструкторе с одним параметром задаётся количество элементов массива и выделяется память для их хранения. Далее нам нужно сделать возможным обращение к элементам этого массива через экземпляр класса Shop, пользуясь синтаксисом массива так, словно класс Shop и есть массив элементов типа Laptop. Для этого добавлен в класс Shop

индексатор:

```
public Laptop this[int pos]
        {
           get
           {
              if (pos >= LaptopArr.Length || pos < 0)
                 throw new IndexOutOfRangeException();
              else
                 return (Laptop)LaptopArr[pos];
           }
           set
           { LaptopArr[pos] = (Laptop)value; }
        }
```

Здесь в аксессоре get мы предусматриваем выход за пределы массива, и при этом генерируется исключение IndexOutOfRangeException.

Для проверки работы классов создаётся отдельный класс Tester, содержащий точку входа. В нём создаётся экземпляр класса Shop, причём в конструкторе задается количество элементов, которые в нём можно разместить.

```
Shop sh = new Shop(3);
```

Далее мы заполняем этот массив объектами Laptop.

```
sh[0] = new Laptop("Samsung", 5200);
sh[1] = new Laptop("Asus", 4700);
sh[2] = new Laptop("LG", 4300);
```

И, наконец, вывод на экран данных по каждому объекту Laptop, пользуясь синтаксисом массива.

```
for (int i = 0; i < 3; i++)
Console.WriteLine(sh[i].ToString());
```

В цикле ограничивающим значением в условии является явно заданное

число 3. Дело в том, что индексатор позволяет нам пользоваться лишь синтаксисом индексирования массива, но других функциональных возможностей массива не предоставляет. Если это был бы стандартный массив, то это условие описывается иначе:

```
for (int i = 0; i < sh.Length; i++)
. . .
```

Для того, чтобы подобная функциональная возможность появилась и в примере, необходимо добавить в класс Shop дополнительное свойство Length.

```
public int Length
{
get { return LaptopArr.Length; }
}
```

Общий вид программы:

```
namespace NS
   {
      public class Laptop
      {
         private string vendor;
         private double price;
         public string Vendor
         {
            get { return vendor; }
            set { vendor = value; }
         }
         public double Price
         {
            get { return price; }
            set { price = value; }
```

```
    }
    public Laptop(string v, double p)
    {
        vendor = v;
        price = p;
    }
    public override string ToString()
    {
        return vendor + " " + price.ToString();
    }
}
public class Shop
{
    private Laptop[] LaptopArr;
    public Shop(int size)
    {
        LaptopArr = new Laptop[size];
    }
    public int Length
    {
        get { return LaptopArr.Length; }
    }
    public Laptop this[int pos]
    {
        get
        {
            if (pos >= LaptopArr.Length || pos < 0)
                throw new IndexOutOfRangeException();
            else
```

```
            return (Laptop)LaptopArr[pos];
        }
        set
        {
            LaptopArr[pos] = (Laptop)value;
        }
    }
}
public class Tester
{
    public static void Main()
    {
        Shop sh = new Shop(3);
        sh[0] = new Laptop("Samsung", 5200);
        sh[1] = new Laptop("Asus", 4700);
        sh[2] = new Laptop("LG", 4300);
        try
        {
            for (int i = 0; i < sh.Length; i++)
                Console.WriteLine(sh[i].ToString());
            Console.WriteLine();
        }
        catch (System.NullReferenceException)
        {  }
    }
}
```

Результат выполнения программы изображен на рисунке 2.8.

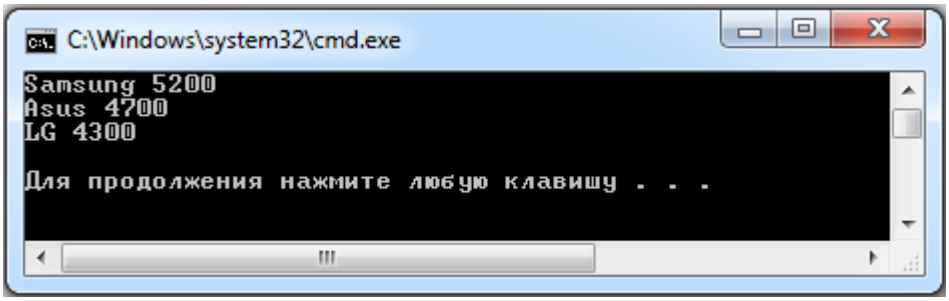


Рис. 2.8 Результат выполнения программы.

Создание многомерных индексаторов

В C# есть возможность создавать не только одномерные, но и многомерные индексаторы. Это возможно, если класс-контейнер содержит в качестве поля массив с более чем одним измерением.

Пример использования многомерного индексатора:

```
namespace NS
{
   public class A
   {
      private int[,] arr;
      private int rows, cols;
      public int Rows
      {
         get { return rows; }
      }
      public int Cols
      {
         get { return cols; }
      }
      public A(int rows, int cols)
      {
         this.rows = rows;
         this.cols = cols;
```

```
            arr = new int[rows, cols];
        }
        public int this[int r, int c]
        {
            get { return arr[r, c]; }
            set { arr[r, c] = value; }
        }
    }
    public class Tester
    {
        static void Main()
        {
            A obj = new A(2, 3);
            for (int i = 0; i < obj.Rows; i++)
            {
                for (int j = 0; j < obj.Cols; j++)
                {
                    obj[i, j] = i + j;
                    Console.Write(obj[i, j].ToString());
                }
                Console.WriteLine();
            }
        }
    }
```

2.4 Деструкторы

В C# существует специальный вид метода, называемый деструктором. Он вызывается сборщиком мусора непосредственно перед удалением объекта из памяти. В деструкторе описываются действия, гарантирующие

корректность последующего удаления объекта, например, проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т.п.).

Синтаксис деструктора:

[атрибуты] [extern] ~имя_класса()

тело

Как видно из определения, деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению 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 соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени. Поскольку деструктор вызывается сборщиком мусора, невозможно 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.

2.5 Вложенные типы

В классе можно определять типы данных, внутренние по отношению к классу. Так определяются вспомогательные типы, которые используются только содержащим их классом. Механизм вложенных типов позволяет скрыть ненужные детали и более полно реализовать принцип инкапсуляции. Непосредственный доступ извне к такому классу невозможен (имеется в виду доступ по имени без уточнения). Для вложенных типов можно использовать те же спецификаторы, что и для полей класса.

Например, класс Monster содержит вспомогательный класс Gun.

Объекты этого класса без «хозяина» бесполезны, поэтому его можно определить как внутренний:

```
using System;
namespace ConsoleApplicationi
{ class Monster
   {
      class Gun
        { ….. }
…..
   }
}
```

Помимо классов вложенными могут быть и другие типы данных: интерфейсы, структуры и перечисления.

2.6 Исключения

Иерархия исключений

Возникновение ошибок при выполнении приложения - не всегда следствие ошибок в коде приложения. Ошибки могут быть вызваны неправильными действиями пользователя или внешними причинами такими как аппаратные сбои, недоступность некоторых ресурсов (например, сетевого диска или сервера базы данных).

Таким образом, профессионально разработанное приложение должно быть готово к возникновению ошибок и должно обеспечивать их обработку.

Для обработки ошибок можно использовать различные подходы. В WinAPI многие функции при неуспешном выполнении возвращают признак ошибки (например, FALSE, INVALID_HANDLE_VALUE, NULL). Далее с помощью функции GetLastError() можно получить код ошибки и найти по этому коду ее описание. Такой способ обработки ошибок является трудоемким, т.к. после вызова функции надо проверять результат возврата, и

не вполне надежным, т.к. можно продолжить работу без проверки результата, так как будто функция завершилась успешно.

В .Net Framework способ обработки ошибок значительно улучшен. Все методы при неуспешном выполнении генерируют исключения. Этот подход имеет следующие преимущества:

- Обработку ошибок можно отделить от основной логики работы программы: всю обработку ошибок можно сосредоточить в одном блоке, а не выполнять проверку после каждого вызова метода.

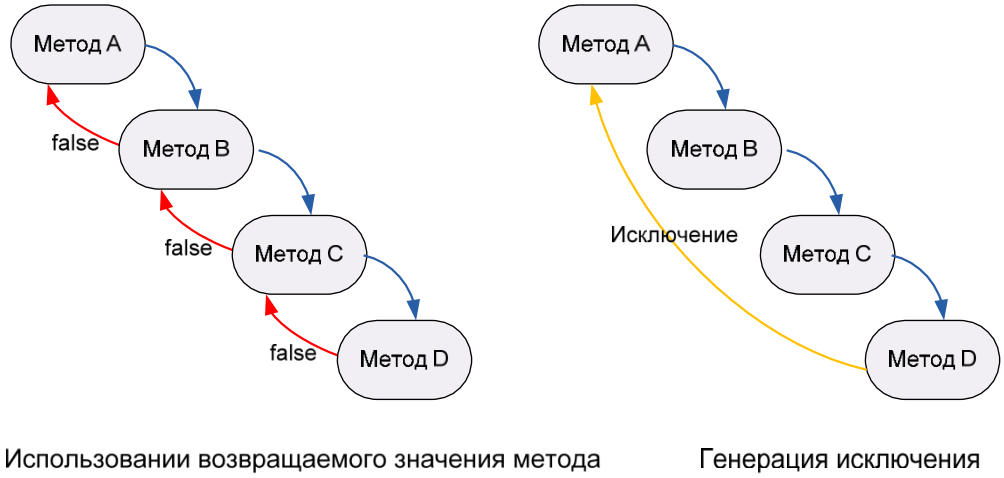


Рис. 2.9. Передача информации о возникновении ошибки по цепочке вызовов

- Возможна ситуация, когда имеется цепочка вызовов (рис. 2.9.), и в последнем методе цепочки возникает ошибка. Без обработки исключений каждый из методов должен вернуть вызывающему методу код ошибки, в случае использования исключений при возникновении ошибки управление будет сразу передано методу, содержащему обработчик исключения.
- Исключение нельзя проигнорировать, т.к. необработанное исключение приведет к аварийному завершению программы.

Базовый класс System.Exception.

Таблица 4. Свойства класса System.Exception

Название свойства	Описание
string Message	Содержит текст сообщения с указанием причины возникновения исключения.
IDictionary Data	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, добавляет записи в этот набор. Код, перехвативший исключение, может использовать эти данные для получения дополнительной информации о причине возникновения исключения.
string Source	Содержит имя сборки, сгенерировавшей исключение.
string StackTrace	Содержит имена и сигнатуры методов, вызов которых привел к возникновению исключения.
MethodBase TargetSite	Содержит метод, сгенерировавший исключение.
string HelpLink	Содержит URL документа с описанием исключения.
Exception InnerException	Указывает предыдущее исключение, если текущее было сгенерировано при обработке предыдущего исключения.

В C# исключение является объектом, который создается и «выбрасывается» (throw) в случае возникновения ошибки. CLR позволяет генерировать исключения любого типа, например Int32, String и др. CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от базового класса SystemException (таб. 4). Эти исключения называются CLS-совместимыми. Такое исключение несет дополнительную информацию об ошибке, которая облегчает отладку программы.

Существует множество классов исключений (наследников System.Exception), разработчик может использовать эти классы и также создавать собственные классы исключений. Все исключения делятся на 2 группы: SystemException и ApplicationException.

SystemException – это класс исключений, которые обычно генерируются CLR или являются исключениями общей природы и могут быть сгенерированы любым приложением. Например, исключение StackOverflowException генерируется CLR при переполнении стека, исключение ArgumentException (и производные от него) могут быть сгенерированы любым приложением, если метод получает недопустимые значения аргументов.

ApplicationExcption – от этого класса должны наследоваться пользовательские исключения, специфичные для приложения.

Иерархия классов исключений является в некоторой степени необычной, т.к. производные классы в основном не добавляют новую функциональность к возможностям базового класса. Эти классы используются для указания более специфических причин возникновения ошибки. Например, от класса ArgumentException наследуются классы ArgumentNullException (генерируется при передаче null в качестве параметра метода) и ArgumentOutOfRange (генерируется при выходе переменной за допустимый диапазон значений).

Обработка исключений

Синтаксис блока обработки исключений:

```
try
  {
    // код, в котором может возникнуть исключение
  }
catch(тип исключения)
  {
```

```
    // обработка исключения
  }
finally
  {
    // освобождение ресурсов
  }
```

Блок try содержит код, требующий общей очистки ресурсов или восстановления после исключения.

Блок catch содержит код, который должен выполняться при возникновении исключения. При объявлении блока catch указывается тип исключения, для обработки которого он предназначен. Если блок try завершился без генерации исключения, блоки catch не выполняются. Если в блоке try не предполагается возникновение исключения, блок catch может отсутствовать, но тогда обязательно должен быть блок finally.

Блок finally обычно содержит очистку ресурсов, а также другие действия, которые необходимо гарантировано выполнить после завершения блока try и catch. Например, в этом блоке можно выполнить закрытие файла или закрытие соединения с БД. Блок finally выполняется всегда, независимо от возникновения исключения в блоке try. Если нет необходимости выполнять очистку ресурсов, блок finally может отсутствовать, но тогда обязательно должен быть блок catch. Блок try сам по себе не имеет смысла.

Для генерации исключения используется ключевое слово throw. Синтаксис генерации исключения:

throw new Конструктор класса исключения()

В качестве типа исключения надо использовать производный класс иерархии исключений, который наиболее полно описывает возникшую проблему. Не рекомендуется использовать базовые классы, т.к. в этом случае при обработке исключения трудно будет точно определить причину его возникновения.

При возникновении исключения в блоке try выполнение этого блока прекращается и CLR выполняет поиск блока catch, предназначенного для обработки исключений такого типа. Если этот блок найден, он выполняется, затем выполняется блок finally (если он есть). Если подходящий блок catch не найден, исключение считается необработанным и приводит к аварийному завершению программы.

Пример. Последовательность выполнения кода при возникновении исключения.

```
namespace ExceptionExample1
  {
    class Program
    {
      static void Main(string[] args)
      {
        try
        {
          Console.WriteLine("Before Exception"); //1
          throw new Exception("Test Exception"); //2
       Console.WriteLine("After Exception.This line will never appear");
        }
        catch (Exception e)
        { Console.WriteLine("Exception: {0}", e.Message); //3 }
        finally
        { Console.WriteLine("In finally block"); //4 }
      }
    }
  }
```

Результат выполнения программы изображен на рисунке 2.10.

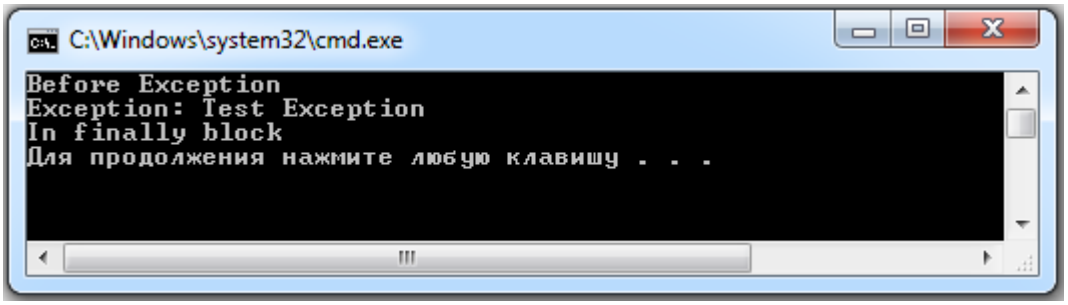


Рис. 2.10 Результат выполнения программы.

Текст «After Exception» не будет выведен, т.к. после генерации исключения выполнение блока try прекращается.

У одного блока try может быть несколько блоков catch для обработки исключений различных типов. При возникновении исключения поиск обработчика начинается с 1-ого блока catch, поэтому следует, сначала размещать обработчики для исключений производных классов, затем базовых.

Для обработки всех типов CLS совместимых исключений можно использовать блок catch, который перехватывает System.Exception, т.к. этот класс является базовым для всех классов исключений:

catch(Exception e)

{ … }

Пример. Использование нескольких блоков catch: вычисляется выражение d = 100/ln(n), n вводится пользователем. Перехватываются следующие типы исключений:

- FormatException – возникает, если введенную пользователем строку невозможно преобразовать в число.
- DivisionByZeroException – возникает, если Ln(n) = 0, т.е. n = 1
- Exception – все остальные исключения, наследуемые от Exception, например Overfloaw.

```
namespace MultipleCatchBlocks
  {
    class Program
```

```
{
   static void Main(string[] args)
   {
      do
      {
         try
         {
      Console.Write("{0}Input int number: ", Environment.NewLine);
            //чтение ввода пользователя
            string s = Console.ReadLine();
            //условие выхода из цикла
            if (s == string.Empty) return;
            //преобразование строки в число
            int n = Convert.ToInt32(s);
            //проверка, что полученное число принадлежит
            //области определения функции ln()
     if (n <= 0) throw new ArgumentOutOfRangeException("n <= 0");
            double f = Math.Log(n);
            int d = 100 / (int)f;
            Console.WriteLine("d = {0} f = {1}", d, f);
         }
         catch (FormatException)
         {
          //происходит, если введенное пользователем значение
          //невозможно преобразовать в целое число
            Console.WriteLine("FormatException");
         }
         catch (DivideByZeroException)
         {
```

```
                //происходит, елси Log(n) = 0 (т.е. n = 1)
                Console.WriteLine("DivideByZeroException");
            }
            catch (Exception e)
            {
         //прехват всех остальных исключений
        //например, исключения ArgumentOutOfRangeException,
        //которое генерируется, если Log(n) не определен (т.е. n <= 0)
                Console.WriteLine("Exception: {0}", e.Message);
            }
        }
        while (true);
    }
  }
}
```

Результат выполнения программы изображен на рисунке 2.11.

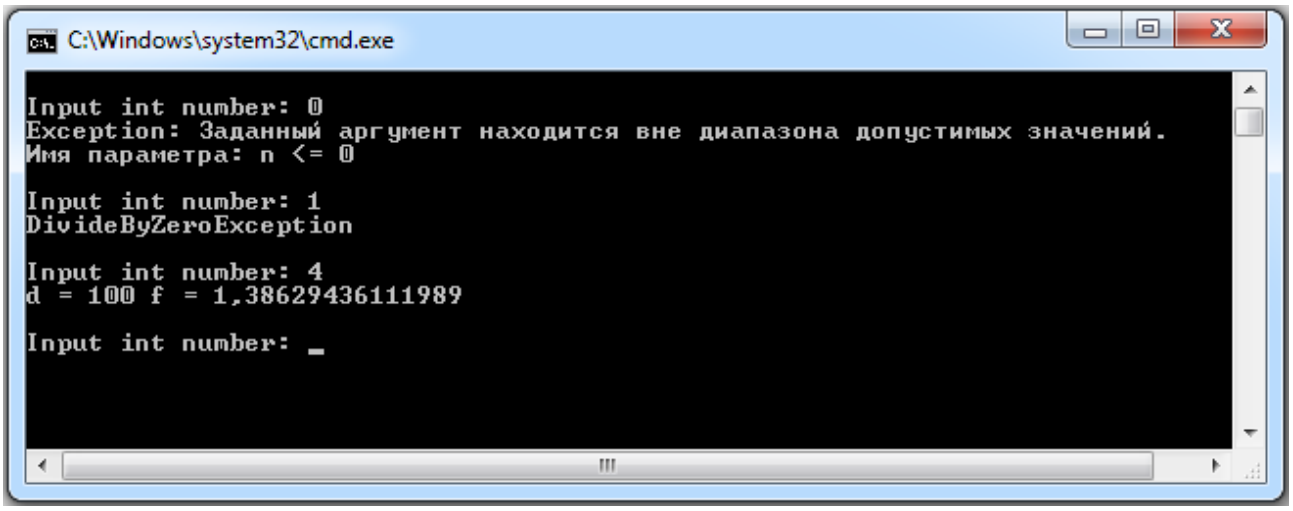


Рис. 2.11 Результат выполнения программы.

Блоки try могут быть вложенными.

Если исключение возникает во вложенном блоке, выполняется блок finally вложенного блока try, затем выполняется поиск подходящего обработчика исключения. Если исключение может быть обработано

внутренним блоков catch, оно перехватывается и обрабатывается, после чего продолжается выполнение кода внешнего блока. Если не возникало нового исключения, блок catch внешнего блока игнорируется, блок finally внешнего блока выполняется в любом случае. Если исключение не может быть перехвачено внутренним блоком catch, выполняется проверка внешнего блока catch. Если этот блок не в состоянии обработать исключение, поиск подходящего обработчика выполняется выше по стеку вызовов.

Пример. Вложенные блоки try. Во внутреннем блоке происходит 2 вида исключений:

- деление на 0;
- обращение к массиву по недопустимому индексу.

1-ое исключение перехватывается внутренним блоком catch, 2-ое – внешним.

```
namespace NestedTryBlocks
  {
    class Program
    {
      static void Main()
      {
        int[] a = new int[5];
        int cnt = 0;
        try //внешний блок try
        {
          for (int i = -3; i <= 3; i++)
          {
            //при делении на 0 не происходит выход из цикла:
            //это исключение перехватывается
            //и обрабатывается вложенным блоком try
            try //вложенный блок try
```

```
                {
                    a[cnt] = 100 / i;
                    Console.WriteLine(a[cnt]);
                    cnt++;
                }
                catch (DivideByZeroException e)
                {
                    Console.WriteLine("In inner catch");
                    Console.WriteLine(e.Message);
                }
            }
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine("In outer catch");
            Console.WriteLine(e.Message);
        }
    }
  }
}
```

Результат выполнения программы изображен на рисунке 2.12.

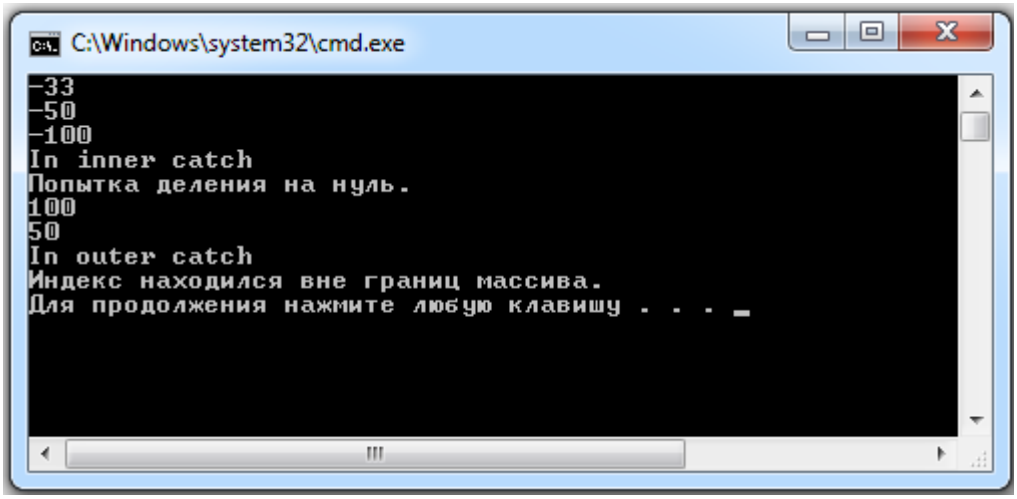


Рис. 2.12 Результат выполнения программы.

Применение конструкций checked и unchecked

При выполнении арифметических операций с целочисленными типами данных может возникать переполнение, т.е. ситуация при которой количество двоичных разрядов полученного результата превышает разрядность переменной, в которую этот результат записывается.

Переполнение может возникнуть в следующих ситуациях:

- в выражениях, которые используют арифметические операторы ++ -- - (унарный) + - * /;
- при выполнении явного преобразования целочисленных типов.

При возникновении переполнения CLR может использовать один из двух вариантов:

- проигнорировать переполнение и отбросить старшие разряды;
- сгенерировать исключение OverflowException.

По умолчанию при возникновении переполнения старшие разряды отбрасываются.

Можно явно задать режим контроля переполнения с помощью ключевых слов checked и unchecked. Ключевое слово checked задает режим контроля переполнения с генерацией исключения. Ключевое слово unchecked задает игнорирование возникновения переполнения.

Пример: Оператор ++ выполняется для переменной b с начальным значением 255:

- в блоке checked – возникает исключение, которое перехватывается в блоке catch;
- в блоке unchecked – исключение не возникает, в переменную b записывается значение 0.

```
namespace CheckOverflowException
  { class Program
```

```
{  static void Main(string[] args)
   {  byte b = 255;
      try
      {  checked
         { b++; //генерация OverflowException thrown }
           Console.WriteLine(b.ToString()); }
      catch (OverflowException e)
      { Console.WriteLine(e.Message);}
      try
      { unchecked
         { b++; //переполнение игнорируется }
         Console.WriteLine(b.ToString()); //0
      }
      catch (OverflowException e)
      { Console.WriteLine(e.Message);}
   }
}
```

Результат выполнения программы изображен на рисунке 2.13.

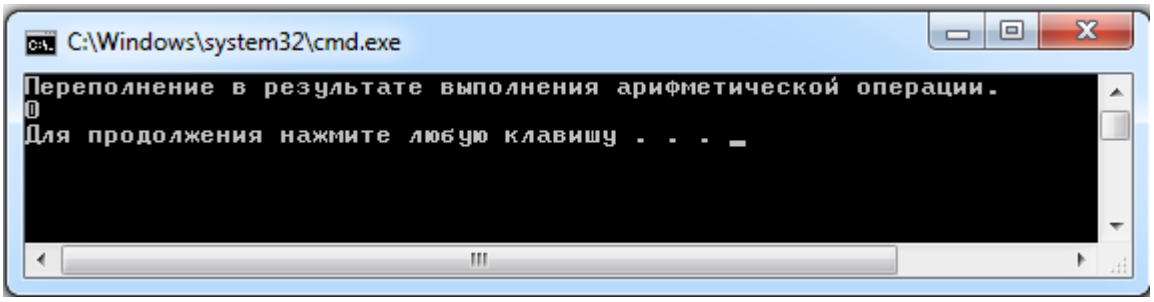


Рис. 2.13 Результат выполнения программы.

3 Контрольные вопросы

1Что понимается под термином «деструктор»?
2В чем состоит назначение деструктора?

3Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

4Что такое рекурсия?

5Что такое статистический метод?

6Что такое перегрузка операторов?

7Какие операторы можно перегружать?

8Что такое индексаторы?

9Для чего нужны одномерные и многомерные индексаторы?

10 Для чего нужны вложенные типы?

11 Для чего нужна обработка исключений?

12 Какие виды исключений бывают? Приведите примеры использования исключений в программах.

13 Для чего используются конструкции checked и unchecked для обработки исключений?

4 Задание

8. Каждый разрабатываемый класс должен, как правило, содержать следующие элементы: скрытые поля, конструкторы с параметрами и без параметров, методы; свойства, индексаторы; перегруженные операции. Функциональные элементы класса должны обеспечивать непротиворечивый, полный, минимальный и удобный интерфейс класса. При возникновении ошибок должны выбрасываться исключения.
9. Отладить и протестировать программу.
10. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

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 одинаковыми границами индексов;
- выполнение операций умножения и деления всех элементов массива на скаляр;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 2.

Описать класс для работы с одномерным массивом строк фиксированной длины. Обеспечить следующие возможности:

- 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 образованием нового массива;
- выполнение операций слияния двух массивов с исключением повторяющихся элементов;
- вывод на экран элемента массива по заданному индексу и всего массива.

Вариант 3.

Описать класс многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Обеспечить следующие возможности:

- вычисление значения многочлена для заданного аргумента;
- операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена;
- получение коэффициента, заданного по индексу;
- вывод на экран описания многочлена.

Вариант 4.

Описать класс, обеспечивающий представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывода на экран подматрицы любого размера и всей матрицы, доступа по индексам к элементу матрицы, сложение и умножение элементов матрицы с заданным числом.

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.

Вариант 6.

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку

(по автору, по году издания или категории), добавления книг в библиотеку, удаления книг из нее, доступа к книге по номеру, сортировки данных (по автору, по году издания или категории).

Вариант 7.

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по фамилии и доступа к записи по номеру.

Вариант 8.

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, имени, дате рождения), добавления и удаления записей, сортировки по разным полям, доступа к записи по номеру.

Вариант 9.

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- сложение, вычитание (как с другой матрицей, так и с числом);
- комбинированные операции присваивания (+=, -=);
- операции сравнения на равенство/неравенство;
- операции вычисления обратной и транспонированной матрицы;
- доступ к элементу по индексам.

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 определителя и нормы;
- доступ к элементу по индексам.

Вариант 11.

Описать класс для работы с двоичным 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.

Вариант 12.

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, нулевая, единичная, симметричная, верхняя треугольная, нижняя треугольная);
- операции сравнения на равенство/неравенство;
- доступ к элементу по индексам.

Вариант 13.

Описать класс «множество», позволяющий выполнять основные операции: добавление и удаление элемента, пересечение, объединение и разность множеств.

Вариант 14.

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 возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя, сортировки по словам или по номеру страниц.

Вариант 15.

Описать класс «автостоянка» для хранения сведений об автомобилях. Для каждого 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 автомобилей, доступа к имеющимся сведениям по номеру места, сортировки данных по различным полям.

Вариант 16.

Описать класс «колода карт», включающий закрытый массив элементов класса «карта». В карте хранятся масть и номер. Обеспечить возможность вывода карты по номеру, вывода всех карт, перемешивания колоды и выдачи всех карт из колоды поодиночке и по 6 штук в случайном порядке.

Вариант 17.

Описать класс для работы с четырехричным 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.

Вариант 18.

Описать класс «поезд», содержащий следующие закрытые поля:

- название пункта назначения;
- номер поезда (может содержать буквы и цифры);
- время отправления.

Предусмотреть свойства для получения состояния объекта.

Описать класс «вокзал», содержащий закрытый массив поездов. Обеспечить следующие возможности:

вывод информации о поезде по номеру с помощью индекса;

- вывод информации о поездах, отправляющихся после введенного с клавиатуры времени;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух поездов;
- вывод информации о поездах, отправляющихся в заданный пункт назначения.

Информация должна быть отсортирована по времени отправления.

Вариант 19.

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 следующие возможности:

- • вывод информации о товаре по номеру с помощью индекса;
- • вывод на экран информации о товаре, название которого введено с клавиатуры;
- 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.

Вариант 20.

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;
- вывод информации о самолетах, отправляющихся в течение часа после введенного с клавиатуры времени;
- вывод информации о самолетах, отправляющихся в заданный пункт назначения;
- перегруженную операцию сравнения, выполняющую сравнение времени отправления двух самолетов.

Информация должна быть отсортирована по времени отправления.

ТЕМА 3. НАСЛЕДОВАНИЕ В C#.

Лабораторная работа № 4

1 Цель и порядок работы

Цель работы – познакомиться с основой объектного подхода в языке C#, созданием объектов, классов и механизмом наследования.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Наследование в C#

Наследование позволяет повторно использовать уже имеющиеся классы, но при этом расширять их возможности. Существует два вида наследования – наследование типа, при котором новый тип получает все свойства и методы родителя и наследование интерфейса, при котором новый тип получает от родителя сигнатуру методов, без их реализации. Все классы, для которых не указан базовый класс, наследуются от класса System.Object (краткая форма названия object). C# не поддерживает множественного наследования. Это означает, что класс в C# может быть наследником только одного класса, но при этом может реализовывать несколько интерфейсов. Другими словами, класс может наследоваться не более чем от одного базового класса и нескольких интерфейсов.

Спецификаторы доступа при наследовании

В C# существуют следующие спецификаторы доступа: private, protected, public, internal, protected internal.

private означает, что никакие другие классы не могут получить доступ к методу. Если в базовом классе объявлены private-методы, то наследник не может ими воспользоваться.

protected предоставляет доступ только для классов, которые

наследуются от данного. Для остальных классов protected-методы недоступны.

public делает метод полностью открытым, то есть он является доступным для всех классов.

Ключевое слово internal является модификатором доступа для типов и членов типов. Внутренние типы или члены доступны только внутри файлов в одной и той же сборке.

При использовании ключевого слова protected internal методы и свойства доступны только в пределах одной сборки, а также для всех производных элементов.

Классам, как и их элементам, может быть назначен любой из этих уровней доступа. Если для элемента класса указан иной модификатор прав доступа, чем для класса,- приоритет у более строгого модификатора.

```
public class Human
{
  private String fName;
  public String FirstName
   {
    get { return fName; }
   set { fName = value; }
   }
}
```

Никакой другой класс не сможет получить доступ к полю fName (так как оно имеет модификатор доступа private), к свойству FirstName получат доступ все классы (потому что оно имеет модификатор доступа public).

Особенности использования конструкторов при наследовании

Конструктор наследования в C# имеет следующий вид:

```
class Наследуемый_класс : Базовый_класс
{
```

// поля, свойства, события и методы класса

}

Если класс наследуется от базового класса и нескольких интерфейсов, то они перечисляются через запятую:

class Наследуемый_класс : Базовый_класс,

Интерфейс1, Интерфейс2

{

// поля, свойства, события и методы класса

}

При создании класса – наследника на самом деле вызывается не один конструктор, а целая цепочка конструкторов. Сначала выбирается конструктор класса, экземпляр которого создается. Этот конструктор пытается обратиться к конструктору своего непосредственного базового класса. Этот конструктор в свою очередь пытается вызвать конструктор своего базового класса. Так происходит, пока не доходим до класса System.Object, который не имеет базового класса. В результате имеем последовательный вызов конструкторов всех классов иерархии, начиная с System.Object заканчивая классом, экземпляр которого хотим создать. В этом процессе каждый конструктор инициализирует поля собственного класса.

Для каждого класса можно определить несколько конструкторов. Если мы для класса-наследника хотим вызвать конструктор базового класса, то необходимо использовать ключевое слово base().

public Наследуемый_класс() : base()

{

// поля, свойства, события и методы класса

}

Усовершенствуем класс Human добавив конструктор не принимающий параметров; конструктор принимающий в качестве параметров имя, отчество и фамилию; конструктор принимающий в качестве параметров имя, отчество,

фамилию и дату рождения:

```
public class Human
  {
    protected String fName;
    public String FirstName
    {
      get { return fName; }
      set { fName = value; }
    }
    protected String mName;
    public String MiddleName
    {
      get { return mName; }
      set { mName = value; }
    }
    protected String lName;
    public String LastName
    {
      get { return lName; }
      set { lName = value; }
    }
    protected DateTime birthday;
    public DateTime Birthday
    {
      get { return birthday; }
      set { birthday = value; }
    }
    public Human() { }
    public Human(String FirstName, String MiddleName,
```

```
                                        String LastName)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
    }
    public Human(String FirstName, String MiddleName,
                                                String LastName,
    DateTime Birthday)
    {
        this.fName = FirstName;
        this.mName = MiddleName;
        this.lName = LastName;
        this.birthday = Birthday;
    }
    public void Work()
    {  // Do something   }
}
```

Тогда класс-наследник может быть реализован следующим образом:

```
public class Employee : Human
{
    public Employee()
        : base()
    {
        // Create Employee object
    }
    public Employee(String FirstName, String MiddleName,
                                            String LastName)
        : base(FirstName, MiddleName, LastName) { }
```

```
    public Employee(String FirstName, String MiddleName,
    String LastName, DateTime Birthday)
      : base(FirstName, MiddleName, LastName, Birthday) { }
  }
```

Сокрытие имен при наследовании

Сокрытие имен происходит, когда в базовом классе и в классе-наследнике объявлены методы с одинаковым именем. В такой ситуации метод базового класса скрывается, и программа, может работать не так, как предусматривал программист. В таких случаях необходимо воспользоваться модификатором new, который скажет компилятору о явном намерении скрыть метод базового класса и использовать метод, объявленный в классе наследнике. Например, пусть у нас есть класс Human, который имеет метод Work().

```
class Human
{
 public void Work()
  { }
}
```

Объявим еще один класс Employee, который наследуется от класса Human и объявим в нем метод, который тоже назовем Work(). Чтобы в дальнейшем избежать путаницы, воспользуемся модификатором new:

```
public class Employee : Human
{
public new void Work()
  { }
}
```

Ключевое слово base

Ключевое слово base используется для доступа к членам базового класса из производного, для вызова метода базового класса при его

переопределении в классе наследнике и при создании конструктора класса наследника, который должен вызвать конструктор класса родителя. Доступ к базовому классу разрешен только в конструкторе, методе экземпляра или методе доступа экземпляра.

Предположим, что необходимо вызвать в классе Employee метод Work() определенный в базовом классе Human:

```
public class Employee : Human
{
  public Employee() : base()
   {
     // Create Employee object
   }
  public override void Work()
    {
      base.Work();
     // Do something great
      }
}
```

Использование ключевого слова sealed (бесплодные классы)

Иногда возникают ситуации, когда необходимо запретить наследовать некоторый класс или переопределять некоторый метод. Для этого класс или метод нужно объявить как терминальный. Для этого используют ключевое слово sealed. Объявим терминальный класс:

```
public sealed class Tutor : Human { }
```

Никакие другие классы не могут наследовать класс Tutor. При попытке это сделать компилятор выдаст ошибку. При этом сам класс Tutor может быть наследован от другого класса.

Аналогично можно запретить переопределять свойства и методы базового класса. Если член базового класса имеет модификатор наследования

virtual, тогда наследник может закрыть дальнейшее переопределение члена. Разрешим переопределение наследуемыми классами метода Work() класса Human.

```
class Human
{
  public virtual void Work()
   {
     // Do something
    }
}
```

Тогда класс Employee может переопределить метод Work() и закрыть возможность его переопределения для собственных наследников.

```
public class Employee : Human
{
    public sealed override void Work()
     {
      // Do something great
      }
}
```

При попытке откомпилировать следующий код, компилятор выдаст ошибку

```
public class Manager : Employee
{
   public override void Work()
    {
       // Try to do something unbelievable
     }
}
```

2.2 Виртуальные методы

Иногда необходимо изменить методы, которые наследуются от базового класса. Для того чтобы была возможность его изменить используют виртуальные методы. Объявив метод или свойство класса как виртуальные, вы тем самым позволяете классам наследникам переопределять данный метод или свойство. Для этого используется ключевое слово **virtual**. Оно записывается в заголовке метода базового класса, например:

virtual public void Passport() . . .

Особенность использования виртуальных методов состоит в следующем:

- Поля-члены и статические методы не могут быть объявлены как виртуальные.
- Применение виртуальных методов позволяет реализовывать механизм позднего связывания.
- На этапе компиляции строится только таблица виртуальных методов, а конкретный адрес метода, который будет вызван, определяется на этапе выполнения.

При вызове метода - члена класса действуют следующие правила:

- для виртуального метода вызывается метод, соответствующий типу объекта, на который имеется ссылка;
- для не виртуального метода вызывается метод, соответствующий типу самой ссылки.

При позднем связывании определение вызываемого метода происходит на этапе выполнения (а не при компиляции) в зависимости от типа объекта, для которого вызывается виртуальный метод.

Виртуальные методы необходимы, когда классу-наследнику нужно изменить некоторые методы, определенные в базовом классе. Виртуальные методы позволяют определить базовому классу методы, реализация которых есть общей для всех производных классов, и методы, которые можно

переопределять. Это позволяет поддерживать динамический полиморфизм. Определения классов-наследников собственных методов становится более гибким, по-прежнему оставляя в силе требование согласующегося интерфейса.

Если в производном классе требуется переопределить виртуальный метод, используется ключевое слово **override**, например:

override public void Passport() . . .

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Пример использования виртуальных методов:

```
class Monster
   {   string name; // закрытые поля
      int health, ammo;
      public Monster()
      {
         this.name = "Noname";
         this.health = 100; this.ammo = 100;
      }
      public Monster(string name) : this()
      { this.name = name; }
      public Monster(int health, int ammo, string name)
      {
         this.name = name;
         this.health = health;
         this.ammo = ammo;
      }
```

```
    public string GetName()
    { return name; }
    public int GetHealth()
    { return health; }
    public int GetAmmo()
    { return ammo; }
    virtual public void Passport()
    {
       Console.WriteLine("Monster {0} \t health = {1} ammo = {2} ",
                                    name, health, ammo);
    }
    public int Health // свойство Health связано с полем health
    {
       get { return health; }
       set { if (value > 0) health = value; else health = 0; }
    }
    public int Ammo // свойство Ammo связано с полем ammo
    {
       get { return ammo; }
       set { if (value > 0) ammo = value; else ammo = 0; }
    }
    public string Name // свойство Name связано с полем name
    { get { return name; } }
}
class Daemon : Monster
{   int brain; // закрытое поле
    public Daemon ()
    { 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; }
    override public void Passport()
    {
        Console.WriteLine("Daemon {0} \t health = {1} ammo = {2}
                    brain = {3} ", Name, Health, Ammo, brain);
    }
}
class Class1
{
    static void Main()
    {
        const int n = 3;
        Monster[] stado = new Monster[n];
        stado[0] = new Monster("Monia");
        stado[1] = new Monster("Monk");
        stado[2] = new Daemon("Dimon", 3);
        foreach (Monster elem in stado) elem.Passport();
        for (int i = 0; i < n; ++i) stado[i].Ammo = 0;
        Console.WriteLine();
        foreach (Monster elem in stado) elem.Passport();
    }
}
```

Результат выполнения программы изображен на рисунке 2.1.

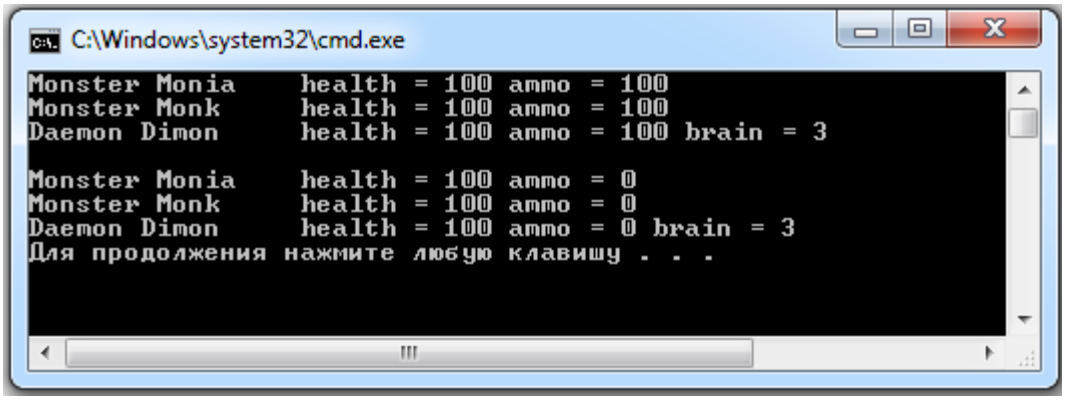


Рис. 2.1 Результат выполнения программы.

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 виртуальных методов. Переопределять виртуальный метод в каждом из потомков необязательно: если он выполняет устраивающие потомка действия, метод наследуется.

Вызов виртуального метода выполняется так: из объекта берется адрес его таблицы (Virtual Method Table, VMT), из VMT 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.

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.

2.3 Абстрактные классы

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие 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**.

Если в классе есть хотя бы один абстрактный метод, весь класс также должен быть описан как абстрактный, например:

```
abstract class Spirit
   {
     public abstract void Passport();
   }
  class Monster : Spirit
  { …
    override public void Passport()
    {
      Console.WriteLine( "Monster {0} \t health = {1} ammo = {2} " ,
                          name, health, ammo );
    }
   … }
class Daemon : Monster
   { …
override public void Passport()
       {
          Console.WriteLine("Daemon {0} \t health = {1} ammo = {2}
```

```
            brain = {3} ", Name, Health, Ammo, brain);
        }
    … }
```

Абстрактные классы используются при работе со структурами данных, предназначенными для хранения объектов одной иерархии, и в качестве параметров методов. Если класс, производный от абстрактного, не переопределяет все абстрактные 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, работающие с объектом любого типа в пределах одной иерархии. Полиморфизм в различных формах является мощным и широко применяемым инструментом ООП.

2.4 Базовый класс Object

Корневой класс System.Object всей иерархии объектов .NET, называемый в C# object, обеспечивает всех наследников несколькими важными методами. Производные классы могут использовать эти методы непосредственно или переопределять их.

Класс object часто используется и непосредственно при описании типа параметров методов для придания им общности, а также для хранения ссылок на объекты различного типа — таким образом реализуется полиморфизм.

Открытые методы класса System.Object:

Метод Equals с одним параметром возвращает значение true, если параметр и вызывающий объект ссылаются на одну и ту же область памяти. Синтаксис:

public virtual bool Equals(object obj);

Метод Equals с двумя параметрами возвращает значение true, если оба

параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool Equals(object ob1, object ob2);

Метод GetHashCode формирует хеш-код объекта и возвращает число, однозначно идентифицирующее объект. Это число используется в различных структурах и алгоритмах библиотеки. Если переопределяется метод Equals, необходимо перегрузить и метод GetHashCode. Синтаксис:

public virtual int GetHashCode ();

Метод GetType возвращает текущий полиморфный тип объекта, то есть не тип ссылки, а тип объекта, на который она в данный момент указывает. Возвращаемое значение имеет тип Type. Это абстрактный базовый класс иерархии, использующийся для получения информации о типах во время выполнения. Синтаксис:

public Type GetType ();

Метод ReferenceEquals возвращает значение true, если оба параметра ссылаются на одну и ту же область памяти. Синтаксис:

public static bool ReferenceEquals (object ob1, 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 ToString ()

В производных объектах эти методы часто переопределяют. Например, можно переопределить метод Equals для того, чтобы задать собственные критерии сравнения объектов, потому что часто бывает удобнее использовать для сравнения не ссылочную семантику (равенство ссылок), а значимую (равенство значений).

Пример применения и переопределения методов класса object.

```
class Monster
   {
```

```
    string name;
    int health, ammo;
    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 || GetType() != obj.GetType()) 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 ToString()
    {
        return string.Format("Monster {0} \t health = {1} ammo = { 2 } ",
                                                    name, health, ammo);
    }
}
class Class1
{
    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 ");
            if (X == Z) Console.WriteLine(" X == Z ");
            else Console.WriteLine(" X != Z ");
            if (X.Equals(Y)) Console.WriteLine(" X Equals Y ");
            else Console.WriteLine(" X not Equals Y ");
            Console.WriteLine(X.GetType());
        }
    }
```

Результат выполнения программы изображен на рисунке 2.2.

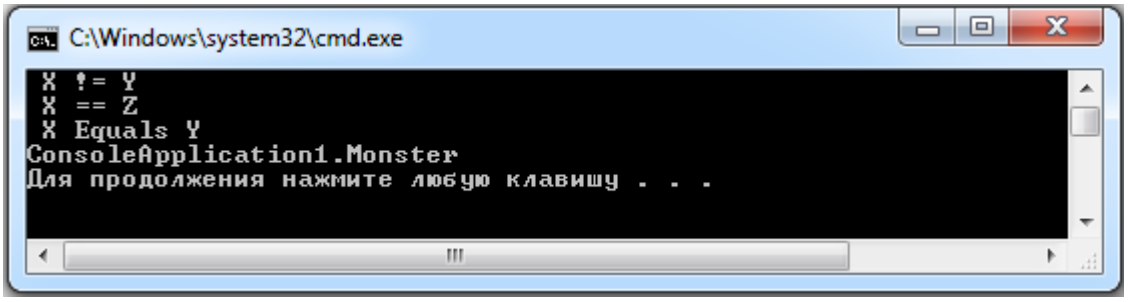


Рис. 2.2 Результат выполнения программы.

В методе Equals сначала проверяется переданный в него аргумент. Если он равен null 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, содержащую значения полей объекта.

Анализируя результат работы программы, можно увидеть, что в операции сравнения на равенство сравниваются ссылки, а в перегруженном методе Equals - значения.

2.5 Наследование исключений

В C# исключения представлены в виде класса. Программист имеет возможность воспользоваться встроенными исключениями, или создавать свои. Все исключения в C# наследуются от класса System.Exception. Из этого класса выведены два класса исключений: SystemException (исключения, которые генерируются общеязыковой средой выполнения CLR) и ApplicationException (генерируются прикладными программами).

При создании исключений в своих программах программист наследует их от ApplicationException. Классы-наследники должны иметь как минимум четыре конструктора: один по умолчанию, второй, задающий свойство сообщению, третий, задающий свойства Message и InnerException, четвертый – для сериализации исключения, поскольку новые классы исключений должны быть сериализируемые.

3 Контрольные вопросы

1. Что понимается под термином «наследование»?
2. Какая классификация объектов соответствует наследованию?
3. Что общего имеет дочерний класс с родительским?
4. В чем состоит различие между дочерним и родительским классами?
5. Приведите синтаксис описания наследования классов в общем виде.
6. Проиллюстрируйте его фрагментом программы на языке C#.
7. Какому отношению соответствует иерархия классов?
8. Какому отношению соответствует иерархия объектов?
9. Что понимается под термином «виртуальный метод»?
10. Какое ключевое слово языка C# используется для определения виртуального метода?
11. В чем состоит особенность виртуальных методов в производных (дочерних) классах?
12. В какой момент трансляции программы осуществляется выбор версии виртуального метода?
13. Какие условия определяют выбор версии виртуального метода?

14. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе?
15. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в производном (дочернем) классе?
16. Какие модификаторы недопустимы для определения виртуальных методов?
17. Что означает термин «переопределенный метод»?
18. В какой момент трансляции программы осуществляется выбор вызываемого переопределенного метода?
19. Приведите синтаксис виртуального метода в общем виде.
20. Что понимается под термином «абстрактный класс»?
21. В чем заключаются особенности абстрактных классов?
22. Какой модификатор языка C# используется при объявлении абстрактных методов?
23. Являются ли абстрактные методы виртуальными?
24. Используется ли модификатор virtual языка C# при объявлении абстрактных методов?
25. Возможно ли создание иерархии классов посредством абстрактного класса?
26. Возможно ли создание объектов абстрактного класса?

4 Задание

11. Выбрать задание согласно варианта.
12. Порядок выполнения работы:
    - разработать поля, методы и свойства для каждого из определяемых классов;
    - все поля классов должны быть описаны с ключевым словом private;
    - реализовать для каждого класса конструкторы по умолчанию и конструкторы с параметрами;
    - методы по изменению значения полей, поиска информации из массива данных объектов по определенным критериям.
13. Реализовать программу на C# в соответствии с вариантом исполнения.
14. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Построить иерархию классов в соответствии с вариантом задания:

1. Студент, преподаватель, персона, заведующий кафедрой.
2. Служащий, персона, рабочий, инженер.

3. Рабочий, кадры, инженер, администрация.
4. Деталь, механизм, изделие, узел.
5. Организация, страховая компания, нефтегазовая компания, завод.
6. Журнал, книга, печатное издание, учебник.
7. Тест, экзамен, выпускной экзамен, испытание.
8. Место, область, город, мегаполис.
9. Игрушка, продукт, товар, молочный продукт.
10. Квитанция, накладная, документ, счет.
11. Автомобиль, поезд, транспортное средство, экспресс.
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
13. Республика, монархия, королевство, государство.
14. Млекопитающее, парнокопытное, птица, животное.
15. Корабль, пароход, парусник, корвет.
16. Самолет, автомобиль, корабль, транспортное средство.
17. Точка, линия, фигура плоская, фигура объемная.
18. Картина, рисунок, репродукция, пейзаж.
19. Статья, раздел, журнал, издательство.
20. Квартира, дом, улица, населенный пункт.

ТЕМА 4. ВИРТУАЛЬНЫЕ И ДИНАМИЧЕСКИЕ МЕТОДЫ. ПОЛИМОРФИЗМ.

Лабораторная работа № 5

1 Цель и порядок работы

Цель работы – Познакомиться с программированием полиморфных методов при объектно-ориентированном подходе при использовании языка C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Полиморфизм

Полиморфизм – одна из основных составляющих объектно-ориентированного программирования, позволяющая определять в наследуемом классе методы, которые будут общими для всех наследующих классов, при этом наследующий класс может определять специфическую реализацию некоторых или всех этих методов. Главный принцип полиморфизма: «один интерфейс, несколько методов». Благодаря ему, можно пользоваться методами, не обладая точными знаниями о типе объектов.

Основным инструментом для реализации принципа полиморфизма является использование виртуальных методы и абстрактных классов.

2.2 Виртуальные методы

Метод, при определении которого в наследуемом классе было указано ключевое слово **virtual**, и который был переопределен в одном или более наследующих классах, называется виртуальным методом. Следовательно, каждый наследующий класс может иметь собственную версию виртуального метода.

Выбор версии виртуального метода, которую требуется вызвать, осуществляется в соответствии с типом объекта, на который ссылается

ссылочная переменная, во время выполнения программы. Другими словами, именно тип объекта, на который указывает ссылка (а не тип ссылочной переменной), определяет вызываемую версию виртуального метода. Таким образом, если класс содержит виртуальный метод и от этого класса были наследованы другие классы, в которых определены свои версии метода, при ссылке переменной типа наследуемого класса на различные типы объектов вызываются различные версии виртуального метода.

При определении виртуального метода в составе наследуемого класса перед типом возвращаемого значения указывается ключевое слово **virtual**, а при переопределении виртуального метода в наследующем классе используется модификатор **override**. Виртуальный метод не может быть определен с модификатором **static** или **abstract**.

Переопределять виртуальный метод не обязательно. Если наследующий класс не предоставляет собственную версию виртуального метода, то используется метод наследуемого класса.

Переопределение метода положено в основу концепции динамического выбора вызываемого метода - выбора вызываемого переопределенного метода осуществляется во время выполнения программы, а не во время компиляции.

Синтаксис:

virtual *тип имя* (*список_параметров*){*тело_метода*};

2.3 Абстрактные классы

В абстрактном классе определяются лишь общие предназначения методов, которые должны быть реализованы в наследующих классах, но сам по себе этот класс не реализует один, или несколько подобных методов, называемых абстрактными (для них определены только некоторые характеристики, такие как тип возвращаемого значения, имя и список параметров).

При объявлении абстрактного метода используется модификатор

abstract. Абстрактный метод автоматически становится виртуальным, так что модификатор **virtual** при объявлении метода не используется.

Абстрактный класс предназначен только для создания иерархии классов, нельзя создать объект абстрактного класса.

Пример:

```
abstract class Animal
{
        public string Name;
        protected int Weight;
        private int Type;
        abstract void Feed();
        public int Animal(int W, int T, string N)
        {
                Weight=W;
                Type=T;
                Name=N;
        }
        public int GetWeight()
        {
                return Weight;
        }
}
class Predator:Animal
{
        private int Speed;
        override void Feed(int Food)
        {
                Weight += Food;
        }
}
```

3 Контрольные вопросы

1) Что понимается под термином «полиморфизм»?
2) В чем состоит основной принцип полиморфизма?
3) В чем состоит значение основного принципа полиморфизма?
4) Какие механизмы используются в языке C# для реализации концепции полиморфизма?
5) Что понимается под термином «виртуальный метод»?
6) Какое ключевое слово языка C# используется для определения виртуального метода?

7) В чем состоит особенность виртуальных методов в производных (дочерних) классах?
8) В какой момент трансляции программы осуществляется выбор версии виртуального метода?
9) Какие условия определяют выбор версии виртуального метода?
10) Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе?
11) Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в производном (дочернем) классе?
12) Какие модификаторы недопустимы для определения виртуальных методов?
13) Что означает термин «переопределенный метод»?
14) В какой момент трансляции программы осуществляется выбор вызываемого переопределенного метода?
15) Приведите синтаксис виртуального метода в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
16) Что понимается под термином «абстрактный класс»?
17) В чем заключаются особенности абстрактных классов?
18) Какой модификатор языка C# используется при объявлении абстрактных методов?
19) Являются ли абстрактные методы виртуальными?
20) Используется ли модификатор virtual языка C# при объявлении абстрактных методов?
21) Возможно ли создание иерархии классов посредством абстрактного класса?
22) Возможно ли создание объектов абстрактного класса?
23) Приведите синтаксис абстрактного класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

4 Задание

15. Реализовать иерархию классов с использованием абстрактного класса в качестве основы иерархии. Разработать поля, методы и свойства для каждого из определяемых классов. Все поля классов должны быть описаны с ключевым словом private.
16. Показать на примере одного из методов, присутствующих в каждом классе, свойство полиморфизма.
17. Отладить и протестировать программу.
18. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Построить иерархию классов в соответствии с вариантом задания:

21. Журнал, книга, печатное издание, учебник.
22. Тест, экзамен, выпускной экзамен, испытание.
23. Место, область, город, мегаполис.
24. Игрушка, продукт, товар, молочный продукт.
25. Студент, преподаватель, персона, заведующий кафедрой.
26. Служащий, персона, рабочий, инженер.
27. Рабочий, кадры, инженер, администрация.
28. Статья, раздел, журнал, издательство.
29. Квартира, дом, улица, населенный пункт.
30. Деталь, механизм, изделие, узел.
31. Организация, страховая компания, нефтегазовая компания, завод.
32. Квитанция, накладная, документ, счет.
33. Корабль, пароход, парусник, корвет.
34. Самолет, автомобиль, корабль, транспортное средство.
35. Точка, линия, фигура плоская, фигура объемная.
36. Картина, рисунок, репродукция, пейзаж.
37. Автомобиль, поезд, транспортное средство, экспресс.
38. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
39. Республика, монархия, королевство, государство.
40. Млекопитающее, парнокопытное, птица, животное.

ТЕМА 5. АБСТРАКТНЫЕ КЛАССЫ. ИНТЕРФЕЙСЫ. ИСКЛЮЧЕНИЯ. ДЕЛЕГАТЫ И СОБЫТИЯ.

Лабораторная работа № 6

1 Цель и порядок работы

Познакомиться с расширенными возможностями языка программирования C#, такими, как интерфейсы и делегаты.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Интерфейсы

В C# для полного отделения структуры класса от его реализации используется механизм интерфейсов.

Интерфейс является расширением идеи абстрактных классов и методов. Синтаксис интерфейсов подобен синтаксису абстрактных классов. Объявление интерфейсов выполняется с помощью ключевого слова interface. При этом методы в интерфейсе не поддерживают реализацию. Членами интерфейса могут быть методы, свойства, индексаторы и события.

Интерфейс может реализовываться произвольным количеством классов. Один класс, в свою очередь, может реализовывать любое число интерфейсов. Каждый класс, включающий интерфейс, должен реализовывать его методы. В интерфейсе для методов неявным образом задается тип public. В этом случае также не допускается явный спецификатор доступа.

Синтаксис:
[атрибуты] [*модификаторы*] **interface**
Имя_интерфейса[**:***список_родительских_интерфейсов*]
{
обьявление_свойств_и_методов
}
Пример:

```
interface Species
{
```

```
        string Species();
        void Feed();
}
class Cheetah:Animal,Species
{
        private string ScientificName;
        public string Species()
                {return ScientificName;}
        public void Feed()
                {Weight++;}
}
```

Можно объявлять ссылочную переменную, имеющую интерфейсный тип. Подобная переменная может ссылаться на любой объект, который реализует ее интерфейс. При вызове метода объекта с помощью интерфейсной ссылки вызывается версия метода, реализуемого данным объектом.

Возможно наследование интерфейсов. В этом случае используется синтаксис, аналогичный наследованию классов. Если класс реализует интерфейс, который наследует другой интерфейс, должна обеспечиваться реализация для всех членов, определенных в составе цепи наследования интерфейсов.

2.2 Делегаты

Делегат — это объект, имеющий ссылку на метод. Делегат позволяет выбрать вызываемый метод во время выполнения программы. Фактически значение делегата – это адрес области памяти, где находится точка входа метода.

Важным свойством делегата является то, что он позволяет указать в коде программы вызов метода, но фактически вызываемый метод определяется во время работы программы, а не во время компилирования.

Делегат объявляется с помощью ключевого слова delegate, за которым указывается тип возвращаемого значения, имя делегата и список параметров вызываемых методов.

Синтаксис:
delegate *тип_возвращаемого_значения имя_делегата* (*список_параметров*);

Характерной особенностью делегата является возможность его использования для вызова любого метода, который соответствует подписи делегата. Это дает возможность определить во время выполнения программы, какой из методов должен быть вызван. Вызываемый метод может быть методом экземпляра, ассоциированным с объектом, либо статическим методом, ассоциированным с классом. Метод можно вызвать только тогда, когда его подпись соответствует подписи делегата.

Многоадресность делегатов

Многоадресность — это способность делегата хранить несколько ссылок на различные методы, что позволяет при вызове делегата инициировать эту цепочку методов.

Для создания цепочки методов необходимо создать экземпляр делегата, и пользуясь операторами + или += добавлять методы к цепочке. Для удаления метода из цепочки используется оператор - или -=. Делегаты, хранящие несколько ссылок, должны иметь тип возвращаемого значения void.

3 Контрольные вопросы

1) Что понимается под термином «интерфейс»?
2) Чем отличается синтаксис интерфейса от синтаксиса абстрактного класса?
3) Какое ключевое слово языка C# используется для описания интерфейса?
4) Поддерживают ли реализацию методы интерфейса?
5) Какие объекты языка C# могут быть членами интерфейсов?
6) Каким количеством классов может быть реализован интерфейс?
7) Может ли класс реализовывать множественные интерфейсы?
8) Необходима ли реализация методов интерфейса в классе, включающем этот интерфейс?
9) Какой модификатор доступа соответствует интерфейсу?
10) Допустимо ли явное указание модификатора доступа для интерфейса?
11) Приведите синтаксис интерфейса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
12) Возможно ли создание ссылочной переменной интерфейсного

типа?
13) Возможно ли наследование интерфейсов?
14) Насколько синтаксис наследования интерфейсов отличается от синтаксиса наследования классов?
15) Необходимо ли обеспечение реализации в иерархии наследуемых интерфейсов?
16) Что понимается под термином «делегат»?
17) В чем состоят преимущества использования делегатов?
18) В какой момент осуществляется выбор вызываемого метода в случае использования делегатов?
19) Что является значением делегата?
20) Какое ключевое слово языка C# используется для описания делегатов?
21) Приведите синтаксис делегата в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
22) Возможно ли использование делегата для вызова метода, соответствующего подписи делегата?
23) Возможен ли вызов метода в том случае, если его подпись не соответствует подписи делегата?
24) Что понимается под термином «многоадресность»?
25) В чем состоит практическое значение многоадресности?
26) Каким образом осуществляется создание цепочки методов для многоадресных делегатов?
27) Какие операторы языка C# используются для создания цепочки методов для многоадресных делегатов?
28) Каким образом осуществляется удаление цепочки методов для многоадресных делегатов?
29) Какие операторы языка C# используются для удаления цепочки методов для многоадресных делегатов?
30) Каким должен быть тип возвращаемого значения для многоадресных делегатов?

4 Задание

19. Реализовать для иерархии из лабораторной работы №5 механизм интерфейсов, при этом один из классов должен реализовывать как минимум 2 интерфейса.
20. Использовать для проверки всех методов данного класса многоадресный делегат.
21. Отладить и протестировать программу.
22. Варианты заданий определяются согласно списка студентов в группе.

Лабораторная работа № 7

1 Цель и порядок работы

Познакомиться с механизмами событийно-ориентированного программирования на языке C#, такими как механизм обработки событий и исключительные ситуации.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя, согласно своему варианту;
- написать программу и отладить ее на ЭВМ.

2 Краткая теория

2.1 Обработка событий

Событие — это автоматическое извещение о каком-либо произошедшем действии. События являются членами класса и объявляются с использованием ключевого слова event. Механизм событий основан на использовании делегатов.

Синтаксис:

event *имя_делегата имя_обьекта*;

Широковещательные события

События могут активизировать несколько обработчиков, в том числе те, что определены в других объектах. Такие события называются широковещательными. Широковещательные события создаются на основе многоадресных делегатов.

Пример:

```
// Объявление делегата, на основе которого будет
// определено событие.
delegate void MyEventHandler () ;
// Объявление класса, в котором инициируется событие.
class MyEvent
{
        public event MyEventHandler activate;
        // В этом методе инициируется событие.
        public void fire()
                { if (activate != null) activate(); }
}
```

```
class X
{
        public void Xhandler()
        {
                Console.WriteLine("Событие получено объектом класса
        X.");
        }
}
class Y
{
        public void Yhandler()
        {
                Console.WriteLine("Событие получено объектом класса Y.");
        }
}
class EventDemo
{
        static void handler()
        {
                Console.WriteLine("Событие получено объектом класса
        EventDemo.")
        }
        public static void Main()
        {
                MyEvent evt = new MyEvent();
                X xOb = new X();
                Y yOb = new Y();
                // Добавление методов handler (), Xhandler()
                // и Yhandler() в цепочку обработчиков события.
                evt.activate += new MyEventHandler(handler);
                evt.activate += new MyEventHandler(xOb.Xhandler);
                evt.activate += new MyEventHandler(yOb.Yhandler);
                evt.fire();
                Console.WriteLine();
                evt.activate -= new MyEventHandler(xOb.Xhandler);
                evt.fire();
        }
}
```

2.2 Исключительные ситуации

Исключение представляет собой ошибку, происходящую во время выполнения программы. С помощью подсистемы обработки исключений для

C# можно обрабатывать такие ошибки, не вызывая краха программы.

Обработка исключений в C# выполняется с применением четырех ключевых слов: try, catch, throw и finally. Эти ключевые слова образуют взаимосвязанную подсистему, в которой использование одного из ключевых слов влечет за собой использование других.

Основа обработки исключений основана на использовании блоков try и catch.

Синтаксис:
try
{*Блок_кода_для_которого_выполняется_мониторинг_ошибок*}
catch (*ExcepType1 exOB1*
{*Обработчик_исключений_ExcepType1*}
catch (*ExcepType2 exOB2*)
{*Обработчик_исключений_ExcepType2*}
Основные системные исключения приведены в следующей таблице:

Исключение	**Значение**
ArrayTypeMismatchException	Тип сохраненного значения несовместим с типом массива
DivideByZeroException	Предпринята попытка деления на ноль.
IndexOutOfRangeException	Индекс массива выходит за пределы диапазона.
InvalidCastException	Некорректное преобразование в процессе выполнения.
OutOfMemoryException	Вызов new был неудачным из-за недостатка памяти.
OverflowException	Переполнение при выполнении арифметической операции.
StackOverflowException	Переполнение стека.

Тип исключения в операторе catch должен соответствовать типу перехватываемого исключения. Неперехваченное исключение непременно приводит к досрочному прекращению выполнения программы.

Для выполнения перехвата исключений вне зависимости от их типа (перехват всех исключений) возможно использование catch без параметров.

Возврат из исключения

Так как оператор catch не вызывается из программы, то после выполнения блока catch управление не передается обратно оператору программы, при выполнении которого возникло исключение. Выполнение программы продолжается с операторов, находящихся после блока catch.

С целью предотвращения этой ситуации возможно указание блока кода, который вызывается после выхода из блока try/catch, с помощью блока finally в конце последовательности try/catch. Общая форма конструкции try/catch, включающей блок finally, показана ниже:

try
    {*Блок кода, выполняющий мониторинг ошибок*}
catch (*ExcepType1 exOB1*)
    {*Обработка исключения ExcepType1*}
catch (*ExcepType2 exOB2*)
    {*Обработка исключения ExcepType2*}
finally
    {*Код блока finally*}

Блок finally будет вызываться независимо от того, появится исключение или нет, и независимо от причин возникновения такового.

Генерация исключений

Исключения автоматически генерируются системой. Однако исключение может быть сгенерировано и посредством оператора throw.

Синтаксис:
    throw exceptOb;

Исключение, перехваченное одним оператором catch, может генерироваться повторно, благодаря чему оно может перехватываться внешним оператором catch. Для этого указывается ключевое слово throw без имени исключения.

Наследование классов исключений

Можно создавать заказные исключения, выполняющие обработку

ошибок в пользовательском коде. Генерирование исключений не представляет особых сложностей. Просто определите класс, наследуемый из класса Exception. В качестве общего правила руководствуйтесь тем, что определенные пользователем исключения наследуются из класса ApplicationException, так как они представляют собой иерархию зарезервированных исключений, связанных с приложениями. Наследуемые классы не нуждаются в фактической реализации в каком-либо виде, поскольку само их существование в системе типов данных позволяет воспользоваться ими в качестве исключений.

Создаваемые пользователем классы исключений автоматически получают доступные для них свойства и методы, определенные в классе Exception.

3 Контрольные вопросы

1) Что понимается под термином «событие»?

2) Являются ли события членами классов?

3) Какое ключевое слово языка C# используется для описания событий?

4) На каком механизме языка C# основана поддержка событий?

5) Приведите синтаксис описания события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

6) Что понимается под термином «широковещательное событие»?

7) На основе какого механизма языка C# строятся широковещательные события?

8) Приведите синтаксис описания широковещательного события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

9) Что понимается под термином «исключительная ситуация (исключение)»?

10) В чем состоит значение механизма исключений в языке C#?

11) Какие операторы языка C# используются для обработки исключений?

12) Какие операторы языка C# являются важнейшими для обработки исключений?

13) Приведите синтаксис блока try…catch в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

14) Приведите пять видов основных системных исключений.

15) Необходимо ли обеспечивать соответствие типов исключения в операторе catch типу перехватываемого исключения?

16) Что происходит в случае неудачного перехвата исключения?

17) В каком случае возможно использование оператора языка C# catch без параметров?

18) Каким образом осуществляется возврат в программу после обработки исключительной ситуации?

19) Какой оператор языка C# используется для обеспечения возврата в программу после обработки исключения?

20) Приведите синтаксис блока finally (в составе оператора try…catch) в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

21) Зависит ли вызов блока finally от наличия исключения?

22) Какие способы генерации исключений Вам известны?

23) Что является источником автоматически генерируемых (неявных) исключений?

24) Каким образом возможно осуществить явную генерацию исключений?

25) Какой оператор языка C# используется для явной генерации исключений?

26) Приведите синтаксис оператора throw в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

27) Каким образом осуществляется повторный перехват исключений в языке C#?

28) Возможно ли создавать специализированные исключения для

обработки ошибок в коде пользователя?

29) Какой системный класс является базовым для создания исключений?

30) На основе какого системного класса осуществляется генерация пользовательских исключений?

31) Необходима ли явная реализация классов, наследуемых от системных исключений?

32) Каким образом обеспечивается обращение к свойствам и методам системных исключений?

4 Задание

Реализовать класс геометрическая фигура на плоскости (геометрическая фигура выбирается согласно варианта). В классе предусмотреть конструкторы; свойства; метод вычисления площади фигуры; события: площадь фигуры равна 1, одна из координат вершины стала нулевой. Обработчики событий должны выводить соответствующие сообщения на экран. Предусмотреть возможность обработки исключений при вводе данных.

0) треугольник;

1) квадрат;

2) ромб;

3) трапеция;

4) прямоугольник;

5) окружность (вместо координат вершины – координаты центра курга);

6) ломанная (вместо площади – длина ломанной).

Вариант определяется следующим образом: номер студента по списку mod 7.

ТЕМА 6. ОСНОВЫ ВИЗУАЛЬНОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C#.

Лабораторная работа № 8

Цель и порядок работы

Познакомиться с основами визуального программирования на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

1. Объект класса Form.

1.1. Создайте новый проект. Дайте форме заголовок "Лабораторная работа №6, Фамилия Имя". Сохраните проект в отдельном каталоге.

1.2. Установите размер формы 500 на 400 пикселей и позицию в центре экрана.

1.3. Установите для формы Вашу фирменную иконку. Эту же иконку присвойте приложению.

1.4. Включите для формы подсказку (ToolTip) "Моя формочка".

1.5. При перемещении указателя мыши по форме она должна плавно менять свой цвет от значений в углах: лев.верхн. - черный, лев.нижн. - голубой, прав.верхн. - красный, прав.нижн. - белый. Нажатием Ctrl-Alt-C этот режим можно включать-выключать.

1.6. Двойной щелчок мыши или Alt-X или F10 закрывают форму (с запросом на разрешение).

1.7. Нажатие клавиш со стрелками сдвигает форму (в границах экрана)

1.8. Щелчок правой кнопки мыши сжимает форму на 5 пикселей со всех сторон. Щелчок правой кнопки мыши с нажатой клавишей Shift увеличивает форму на 5 пикселей во все стороны.

1.9. При изменении размеров формы раздается щелчок, а сами размеры выводятся в заголовке формы.

1.10. При щелчке центральной клавиши мыши или обоих крайних клавишей форма центрируется по указателю курсора.

2. Полное исследование квадратного уравнения ax2 + bx + c = 0 и биквадратного уравнения ax4 + bx2 + c = 0 в одном проекте (рассмотреть случаи a=0, b=0, c=0).

2.1. Реализовать класс квадратное уравнение. В классе предусмотреть:

- поля;
- свойства (коэффициенты квадратного уравнения (чтение, запись) и дискриминант (чтение);
- конструктор по умолчанию;
- конструктор с параметрами;
- методы:
  - решение квадратного уравнения, который возвращает список корней квадратного уравнения;
  - вывод корней квадратного уравнения (переопределить метод ToString() класса Object) который вызывает метод решения квадратного уравнения;
- событие на изменение коэффициентов квадратного уравнения. В качестве обработчика создать и зарегистрировать метод класса формы, на которой расположены компоненты ввода коэффициентов.

Ввод коэффициентов осуществлять в три textBox. В textBox организовать ограничение на ввод данных (можно вводить только действительные числа, все остальные символы должны игнорироваться, в т.ч. вторая и последующие запятые минус на втором и последующих местах). Результаты выводить в компонент Label с

помощью обработчика события.

2.2. Реализовать класс биквадратное уравнение – как наследника квадратного уравнения. В классе предусмотреть:

- конструктор по умолчанию;
- конструктор с параметрами;
- методы:
  - переопределенный метод решения квадратного уравнения (класса родителя) для решения биквадратного уравнения (метод должен использовать метод решения квадратного уравнения базового класса). Переопределенный метод должен возвращать список корней биквадратного уравнения;
  - вывод корней биквадратного уравнения (переопределить метод ToString() класса Object) который вызывает метод решения биквадратного уравнения;
- событие на изменение коэффициентов биквадратного уравнения, в качестве обработчика создать и зарегистрировать метод класса формы, на которой расположены компоненты ввода коэффициентов.

Ввод коэффициентов осуществлять в три textBox. В textBox организовать ограничение на ввод данных (можно вводить только действительные числа, все остальные символы должны игнорироваться, в т.ч. вторая и последующие запятые минус на втором и последующих местах). Результаты выводить в компонент Label с помощью обработчика события.

ТЕМА 7. ИСПОЛЬЗОВАНИЕ СТАНДАРТНЫХ КОМПОНЕНТ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА.

Лабораторная работа № 9

Цель и порядок работы

Познакомиться с элементом управления Button на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

Создать проект, содержащий 4 кнопки со следующими функциями:

1-я кнопка

a. прячет / показывает 2-ю;

b. блокирует / включает 2-ю;

c. переключает видимость 2-й и 3-й (видна только одна из кнопок по очереди);

d. переключает блокировку 2-й и 3-й (активна только одна из кнопок по очереди);

e. меняет подсказки 2-й и 3-й.

2-я кнопка (учесть реальные ограничения, связанные с размером формы)

a. сдвигает первую кнопку на 10 пикселей вверх;

b. сдвигает первую кнопку на 10 пикселей вниз;

c. сдвигает первую кнопку на 10 пикселей вправо;

d. сдвигает первую кнопку на 10 пикселей влево;

e. сжимает форму на 5 пикселей со всех сторон;

f. раздвигает форму на 5 пикселей во все стороны.

3-я кнопка (1-сама по себе, 2 - с Shift, 3 - с Ctrl)

a. 1-увеличивает, 2-уменьшает, 3-переключает по кругу (из 3-х) шрифты на форме;

b. вкл/выкл 1-полужирн., 2-накл., 3-подчерк. стили шрифтов на форме

c. переключает цвет формы по кругу (из 5-и) 1 – в одну сторону, 2 – в др., 3 – включает белый цвет.

4-я кнопка

a. вкл./выкл. системную кнопку;

b. вкл./выкл. кнопку “развернуть”;

c. вкл./выкл. кнопку “свернуть”;

d. перебирает тип курсора (по кругу из 5-ти);

e. перебирает тип рамки (по кругу из 5-ти).

Вариант определяется согласно списка студентов в группе.

Вариант	1 кнопка	2 кнопка	3 кнопка	4 кнопка
1	a	a	a	a
2	b	b	a	b
3	c	c	a	c
4	d	d	a	d
5	e	e	b	e
6	a	f	b	b
7	b	a	b	c
8	c	b	b	d
9	d	c	c	e
10	e	d	c	a
11	a	e	c	c
12	b	f	c	d
13	c	a	a	e
14	d	b	b	a
15	e	c	b	b

16	a	d	c	d
17	b	e	c	e
18	c	f	c	a
19	d	a	c	b
20	e	b	b	c

Общее для все вариантов:

Щелчок на форме и <ALT-I> восстанавливают начальное состояние кнопок.

Двойной щелчок и <ALT-X>- закрывают форму.

Для каждой кнопки определить подсказку (ToolTip), описывающую функцию кнопки, причем подсказка появляется не стандартным образом, а вместе с именем и заголовком кнопки в специальной области формы (StatusStrip) при попадании указателя мыши на кнопку и гаснет при уходе указателя с кнопки.

Лабораторная работа № 10

Цель и порядок работы

Познакомиться с основными компонентами управления на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

Создать проект, содержащий следующие компоненты:

1 - **ComboBox**, позволяющий выбрать из списка в 8 элементов (* - с возможностью дополнения через внутреннее окно **TextBox**):

a. тип курсора;

b. цвет формы;

c. стиль шрифта на форме;

d. *размер формы;

e. *заголовок формы.

2 - **RadioButton** – переключатель (список переключателей организовать используя компонент **GroupBox):**

a. типа рамки формы (не менее 6);

b. положения формы на экране: 2 группы по 3 положения в каждой группе;

c. размер шрифта формы (не менее 6);

d. цвет формы (не менее 6).

3 - выключатели **CheckBox** (в пунктах e-g не использовать свойства типа Visible и Enable). Переключатели реализовать на уровне программного кода с инициализацией в конструкторе формы. Для каждого компонента назначить необходимые обработчики событий.

a. кнопки системного меню;

b. подсказок;

c. очистки/восстановления заголовка формы

d. запрещения закрытия приложения;

e. запрещение изменения выключателей;

f. запрещения изменений в ComboBox;

g. запрещения изменений переключателей **RadioButton.**

4 - Меню формы **MenuStrip.** На форме расположен компонент ListBox1 в котором заданы числа. Меню формы должно содержать следующие команды:

1. Возможность добавления чисел в список и удаления выделенного элемента из списка.

2. Возможность выполнения следующих операций:

Определить:

a. количество и сумму четных чисел;

b. произведение первых 3 положительных чисел;

c. все положительные числа;

d. среднее геометрическое положительных чисел;

e. есть ли среди чисел хоть одно нулевое;

f. есть ли среди первых 5 чисел хотя бы одно нечетное, если есть, то эти 5 чисел внести в отдельный список ListBox2;

g. есть ли среди последних 5 чисел хотя бы одно четное, если есть, то эти 5 чисел внести в отдельный список ListBox2;

h. есть ли среди элементов списка с нечетными номерами хоть один нулевой, если есть, то эти элементы внести в отдельный список ListBox2.

5. Контекстное меню формы **ContextMenuStrip.** На форме расположен компонент ListBox1 в котором записаны строки. Контекстное меню компонента ListBox1 должно содержать следующие операции:

Найти:

a. количество строк, в которых нет цифр;

b. строки, оканчивающиеся на русскую букву;

c. текст, составленный из строк максимальной и минимальной длины;

d. текст, полученный после удаления пробелов из нечетных строк и удвоения пробелов в четных строчках;

e. строки в которых содержится максимальное количество различных символов;

f. строки в которых есть слова «перевертыши» (например: абвггвба);

g. строки с максимальным количеством слов начинающихся с первого символа строки.

Для каждого компонента определить подсказку.

Варианты заданий определяются согласно списка студентов в группе.

Варианты

№	**1**	**2**	**3**	**4**	**5**
1	a	a	ad	af	bd
2	b	b	be	dg	cf
3	c	c	cf	ch	ad

4	d	d	dg	ad	be
5	e	a	ae	be	cd
6	a	b	bf	cg	ag
7	b	c	cg	ae	bd
8	c	d	ad	bd	ce
9	d	a	be	cf	cg
10	e	b	cf	ad	ad
11	a	c	af	bh	be
12	b	d	bg	cd	cg
13	c	a	ac	ag	bd
14	d	b	bd	bd	cf
15	e	c	ce	ce	ad
16	a	d	af	bf	ad
17	b	a	dg	gh	be
18	c	b	ce	ad	cf
19	d	c	ad	be	dg
20	e	d	be	cf	be

ТЕМА 8. РАЗРАБОТКА МНОГООКОННЫХ ПРИЛОЖЕНИЙ. СТАНДАРТНЫЕ ОКНА ДИАЛОГА. ФАЙЛОВЫЕ ТИПЫ ДАННЫХ.

Лабораторная работа № 11

Цель и порядок работы

Познакомиться с файловами типами и стандартными диалогами для работы с файлами на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

1. С помощью стандартного диалога выбрать папку и загрузить из нее в **ListBox** все названия текстовых файлов.

Переписать из текстовых файлов, названия которых выделены в **ListBox** в другой текстовый файл, заданный с помощью **SaveFileDialog:**

a. нечетные строки;
b. строки, в которых нет цифр;
c. все строки, кроме пустых;

Определить в текстовых файлах, названия которых выделены в **ListBox**:

a. количество строк в файле;
b. количество символов в файле;
c. количество символов в последней строке.

Загрузить текстовый файл в компонент **ListBox.** Переписать в другой текстовый файл:

a. строки списка в обратном порядке (последняя, предпоследняя...);
b. выделенные строки;
c. строки отсортированного списка.

Для выбора файлов использовать стандартные диалоги **OpenFileDialog** и **SaveFileDialog**.

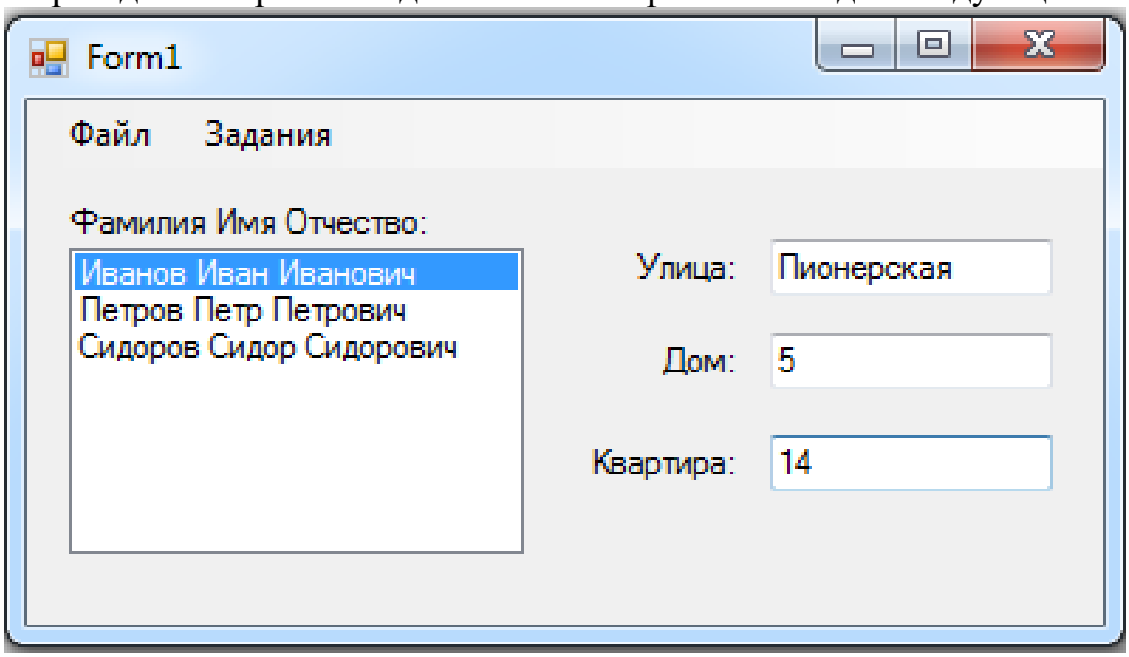
2. Задан текстовый файл, в строках которого через пробел содержится информация согласно задания (вариант задания определяет согласно списка студентов в группе).

Необходимо предусмотреть следующие операции:

- возможность сохранения данных из текстового файла в xml файл;
- возможность загружать данные из xml файла следующим образом:

Пример: Текстовый файл содержит в строках (через пробел) следующую информацию:

Фамилия Имя Отчество Улица Дом Квартира
Форма для отображения данных из xml файла выглядит следующим образом:



- при выделении соответствующего элемента из ListBox должны отображаться данные соответствующей записи из xml файла;
- предусмотреть возможность добавления, удаления и изменения данных в xml файле;
- формирование карточки выделенной записи в word, excel, pdf файлы;
- все операции должны реализовываться через меню и контекстное меню.

Варианты для второго задания:

Задание 1.

В текстовом файле храниться следующая информация:

–название груза,
– номер вагона,
– стоимость перевозки,
– дата отгрузки,
– дата возврата вагона.

Определить общую стоимость перевозок. Информацию выдавать в виде сообщения.

Задание 2.

В текстовом файле храниться следующая информация:

– номер посылки,
– вес посылки,
– цена,
– дата отправки,
– пункт назначения.

Определить средний вес посылок. Информацию выдавать в виде сообщения.

Задание 3.

В текстовом файле храниться следующая информация:
- название товара,
- название фирмы-производителя,
- цена,
- количество,
- дата поступления на склад.

Определить общую стоимость товаров. Информацию выдавать в виде сообщения.

Задание 4.

В текстовом файле храниться следующая информация:
- наименование товара,
- место покупки,
- цена,
- дата покупки.

Определить общую стоимость покупок. Информацию выдавать в виде сообщения.

Задание 5.

В текстовом файле храниться следующая информация:
- название заказа,
- дата заказа,
- стоимость,
- код исполнителя,
- дата выполнения.

Определить общую стоимость заказов. Информацию выдавать в виде сообщения.

Задание 6.

В текстовом файле храниться следующая информация:
- Ф.И.О.,
- должность,
- оклад,
- дата поступления на работу,

Определить средний оклад. Информацию выдавать в виде сообщения.

Задание 7.

В текстовом файле храниться следующая информация:
- Фамилия владельца,
- номер автомобиля,
- марка автомобиля,

– дата выпуска,
– дата регистрации.

Вывести всех владельцев автомобилей указанной марки. Информацию выдавать в виде сообщения.

Задание 8.

В текстовом файле храниться следующая информация:
– Ф.И.О.,
– вес,
– рост,
– дата рождения,
– пол,
– место рождения.

Определить количество человек, чей вес больше заданного. Информацию выдавать в виде сообщения.

Задание 9.

В текстовом файле храниться следующая информация:
– шифр книги,
– название,
– автор,
– дата последней выдачи,
– год издания.

Определить количество книг указанного автора. Информацию выдавать в виде сообщения.

Задание 10.

В текстовом файле храниться следующая информация:
– номер билета,
– номер рейса,
– цена,
– дата продажи,
– фамилия кассира.

Определить сумму и количество проданных билетов. Информацию выдавать в виде сообщения.

Задание 11.

В текстовом файле храниться следующая информация:
– пункт назначения,
– номер рейса,
– стоимость билета,
– название авиакомпании.

Определить общее количество рейсов, выполняемых указанной авиакомпанией. Информацию выдавать в виде сообщения.

Задание 12.

В текстовом файле храниться следующая информация:

– наименование оборудования,
– дата покупки,
– срок гарантии (в месяцах),
– стоимость,
– фирма-производитель.

Определить средний срок (в месяцах) гарантии. Информацию выдавать в виде сообщения.

Задание 13.

В текстовом файле храниться следующая информация:

– Ф.И.О. студента,
– факультет,
– курс,
– дата рождения,
– место рождения.

Определить общее количество студентов указанного курса. Информацию выдавать в виде сообщения.

Задание 14.

В текстовом файле храниться следующая информация:

– Ф.И.О. пациента,
– дата рождения,
– дата посещения врача,
– диагноз,
– пол.

Определить количество пациентов, которым был поставлен указанный диагноз. Информацию выдавать в виде сообщения.

Задание 15.

В текстовом файле храниться следующая информация:

– название валюты,
– цена покупки,
– цена продажи,
– дата,
– название банка.

Определить среднюю разницу между ценой продажи и ценой покупки для указанной валюты. Информацию выдавать в виде сообщения.

Задание 16.

В текстовом файле храниться следующая информация:

- порода собаки,
- год рождения,

- кличка,
- дата регистрации,
- Ф.И.О. владельца.

Найти средний возраст собак для указанной породы. Информацию выдавать в виде сообщения.

Задание 17.

В текстовом файле храниться следующая информация:
- код задания,
- Ф.И.О. исполнителя,
- контрольный срок выполнения (в днях),
- дата выдачи задания,
- дата выполнения.

Найти общую продолжительность выполнения заданий для указанного кода задания. Информацию выдавать в виде сообщения.

Задание 18.

В текстовом файле храниться следующая информация:
- название продукции,
- стоимость за единицу,
- количество,
- дата выпуска,
- изготовитель.

Найти общую стоимость продукции для указанного названия продукции. Информацию выдавать в виде сообщения.

Задание 19.

В текстовом файле храниться следующая информация:
- Ф.И.О.,
- дата заключения контракта,
- срок действия контракта,
- должность,
- отдел,
- оклад.

Найти средний размер оклада для указанного отдела. Информацию выдавать в виде сообщения.

Задание 20.

В текстовом файле храниться следующая информация:
- название фирмы,
- количество акций,
- стартовая цена акции,
- цена продажи,
- дата продажи.

Найти общее количество акций, проданных по цене, превышающей стартовую цену в 1,5 раза для указанной фирмы. Информацию выдавать в виде сообщения.

Лабораторная работа № 12

Цель и порядок работы

Познакомиться с разработкой многооконных приложений на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

Вариант задания определяется следующим образом: остаток от деления на три номера студента в группе.

Общие требования к пользовательскому интерфейсу:

1. На экране появляется на 20 секунд форма-заставка с произвольным графическим изображением и текстовой информацией о данном приложении. Далее запускается приложение с возможностью создания, открытия, изменения и сохранения файла по типу электронной таблицы (многодокументный интерфейс).

2. Приложение должно содержать:

Меню приложения:

Работа с файлами:

создание файла собственного формата,
открытие файла собственного формата,
закрытие файла собственного формата,
сохранение файла собственного формата,
сохранение таблицы в файл формата Excel,
сохранение таблицы в файл формата Word.

Окно:

расположение окон каскадом,
расположение окон вертикально,
расположение окон горизонтально.

Задание:

(согласно варианта).

Панель инструментов с быстрыми кнопками, каждая кнопка должна содержать соответствующую подсказку согласно производимой операции. Кнопки должны дублировать системное меню.

Контекстное меню для работы ячейками таблицы.

3. Предусмотреть механизм ввода, изменения и удаления информации из ячеек, возможность распознавания текстовой и числовой информации.

4. Создание диаграммы по выделенному диапазону.

Задание №0

Для числовой информации: возможность подсчета суммы значений в выделенных ячейках или из указанного диапазона, при невозможности выполнения выдать соответствующее сообщение.

Для текстовой информации: занесение в ячейку строки получаемой в результате выбора из выделенных ячеек или с указанного диапазона всех слов начинающихся на букву "А", в случае невозможности найти такие слова выдать соответствующее сообщение.

Задание №1

Для числовой информации: возможность подсчета среднего значения в выделенных ячейках или из указанного диапазона, при невозможности выполнения выдать соответствующее сообщение.

Для текстовой информации: занесение в ячейку строки получаемой в результате выбора из выделенных ячеек или с указанного диапазона всех слов, длина которых меньше 4 символов, при невозможности найти такие слова выдать соответствующее сообщение.

Задание №2

Для числовой информации: возможность подсчета количества четных чисел в выделенных ячейках или из указанного диапазона, при невозможности выполнения выдать соответствующее сообщение.

Для текстовой информации: занесение в ячейку строки получаемой в результате выбора из выделенных ячеек или с указанного диапазона всех слов, в которых есть словосочетание "Del", в случае невозможности найти такие слова выдать соответствующее сообщение.

ТЕМА 9. ОРГАНИЗАЦИЯ МЕХАНИЗМА DRAG&DROP.

Лабораторная работа № 13

Цель и порядок работы

Познакомиться с организацией и реализацией механизма DragDrop на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

Общее для всех вариантов:

При нажатии клавиши **Esc** при переносе или копировании происходит отмена операции.

Варианты заданий:

Вариант задания определяется следующим образом: остаток от деления на три номера студента в группе.

Задание №0

Создайте проект, содержащий 2 списка **ListBox** и две кнопки **Button** и обеспечивающий решение следующих задач:

1. 1 кнопка сохраняет данные в файл из первого списка.
2. 2 кнопка выводит содержимое из этого файла во второй список.
3. Перенос выделенных строк из первого списка и обратно методом **DragDrop**:

   - с клавишами **Alt + C** происходит копирование строк;

   - при переносе строк из одного списка в другой источник меняет цвет на красный, по окончании переноса на стандартный;

   - при копировании цвет источника меняется на зелёный, по окончании копирования на стандартный.
4. При нажатии на клавишу **Del** выделенные строки удаляются из списка.

Задание №1

Создайте проект, содержащий компоненты **TextBox**, **Button**, **ListBox**, **label** и

обеспечивающий решение следующих задач:

1. Кнопка позволяет сохранять/выводить данные в/из файла, меняя при нажатии соответствующее название для последующей операции.

2. Метка отражает произведенную операцию сохранения или вывода данных.

3. Перенос или копирование выделенной части строки из **TextBox** в список методом **DragDrop:**

   - с клавишами **Ctrl + D** происходит копирование строки;

   - при переносе из **TextBox** в список источник меняет цвет на синий, по окончании переноса на стандартный;

   - при копировании источник меняет свой цвет на красный, по окончании копирования на стандартный;

4. при нажатии клавиши **Shift** список очищается, а при нажатии **Shift+ Del** очищается **TextBox**.

Задание №2

Создайте проект, содержащий компоненты три **CheckBox** и **ComboBox** обеспечивающий решения следующих задач:

1. В **ComboBox** содержатся названия **CheckBox**.

2. Выбранное название в **ComboBox** переносится или копируется с помощью механизма **DragDrop** и становится названием **CheckBox**:

   - в зависимости от флажка, если флажок установлен, то название изменяется, в противном случае выдается сообщение о невозможности переноса;

   - с клавишами **Alt + K** происходит копирование названия;

   - при переносе названия приемник изменяет свой цвет на зеленый по окончании переноса на стандартный;

   - при копировании цвет источника меняется на синий, по окончании копирования на стандартный.

3. Нажатие кнопок **Ctrl+Shift+P** возвращает все в исходное состояние.

4. При нажатии клавиш **F10** или **Alt +X** форма минимизируется в низу экрана.

ТЕМА 10. ПОСТРОЕНИЕ ГРАФИЧЕСКИХ ИЗОБРАЖЕНИЙ.

Лабораторная работа № 14

Цель и порядок работы

Познакомиться с основными графическими инструментами на языке C#.

Порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить задание у преподавателя;
- написать программу и отладить ее на ЭВМ.

Задания:

I. Построить объемный чертеж, согласно варианта, и обеспечить изменение его размеров и перемещение:

a. объект чертежа описать через собственный класс;
b. после установки новых размеров (в компонентах **NumericUpDown**) должна происходить автоматическая перерисовка чертежа;
c. при каждом нажатии правой кнопки мыши на объект должен происходить сдвиг на определенный шаг (учесть реальные размеры формы);
d. непрерывное перемещение и остановка при выборе соответствующего пункта контекстного меню объекта (учесть реальные размеры формы).

1. В треугольной пирамиде построить сечение, параллельное основанию.

2. В треугольной пирамиде построить сечение, проходящее через боковое ребро и медиану основания.

3. В треугольной пирамиде построить сечение, проходящее через одну из сторон основания и середину противоположного ребра.

4. В треугольной пирамиде построить сечение, проходящее через среднюю линию боковой грани и противоположную вершину основания.

5. В треугольной пирамиде провести сечение, проходящее через сторону основания и наклоненное к основанию под углом 30°.

6. В правильной четырехугольной пирамиде провести сечение, проходящее через диагональ основания и вершину пирамиды.

7. В правильной четырехугольной пирамиде провести сечение, проходящее через диагональ основания и середину бокового ребра.

8. В правильной четырехугольной пирамиде провести сечение,

проходящее через диагональ основания и наклоненное к плоскости основания под углом 30°.

9. В правильной четырехугольной пирамиде провести сечение, параллельное основанию и проходящее через середину бокового ребра.

10. В правильной четырехугольной пирамиде провести сечение, проходящее через вершину пирамиды и перпендикулярное плоскости основания.

11. В правильной четырехугольной пирамиде провести сечение, проходящее через одну из сторон основания и середину высоты.

12. Основание четырехугольной пирамиды — ромб. Вершина пирамиды проектируется в центр симметрии ромба. Провести сечение, проходящее через высоту основания, опущенную из тупого угла ромба, и боковое ребро, которое проходит через эту же вершину.

13. Основание пирамиды — ромб. Вершина пирамиды проектируется в вершину острого угла ромба. Провести сечение, проходящее через вершину пирамиды и высоту ромба, опущенную из тупого угла.

14. В прямоугольном параллелепипеде провести диагональное сечение.

15. В прямоугольном параллелепипеде провести сечение, проходящее через сторону нижнего основания и противоположную сторону верхнего основания

16. В прямой четырехугольной призме провести сечение, проходящее через диагональ нижнего основания и одну из вершин верхнего основания.

17. В прямой четырехугольной призме провести сечение, проходящее через сторону нижнего основания под углом 30° к основанию.

18. В правильной шестиугольной призме провести сечение, проходящее через одну из сторон нижнего основания и противоположную ей сторону верхнего основания.

19. В прямоугольном параллелепипеде построить сечение, проходящее через одну из сторон нижнего основания и одну из вершин верхнего.

20. В прямоугольном параллелепипеде построить сечение, проходящее через одно из его ребер и точку пересечения диагоналей противолежащей этому ребру грани.

21. В правильной шестиугольной пирамиде построить сечение, проходящее через вершину и большую диагональ основания.

22. В прямом цилиндре построить осевое сечение.

23. В правильной шестиугольной призме построить сечение, проходящее через большую диагональ нижнего основания и одну из сторон верхнего.

II. Обеспечить постоянную закраску области (круга, эллипса, сектора, треугольника, прямоугольника, многоугольника) с возможностью выбора из **ComboBox**:

- вида фигуры;
- цвета заливки;
- вида кисти.

III. Загрузить из файла или буфера обмена рисунок, обеспечить его дорисовку разными цветами и возможности сохранения рисунка в файле или буфере обмена.

IV. Обеспечить перемещение на форме нескольких рисунков (каждый перемещается по определенному закону) и реакцию на их возможное пересечение.

Объект рисунок реализовать через собственный класс. Предусмотреть возможность добавление объекта рисунок на форму.

ТЕМА 11. ОРГАНИЗАЦИЯ МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ.

Лабораторная работа № 15

Цель и порядок работы

Цель работы – изучение организации многопоточных приложений.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по организации многопоточных приложений.

Выполнить задание.

Задание

1. Реализуйте последовательную обработку элементов вектора, например, умножение элементов вектора на число. Число элементов вектора задается параметром N.
2. Реализуйте многопоточную обработку элементов вектора, используя разделение вектора на равное число элементов. Число потоков задается параметром M.
3. Выполните анализ эффективности многопоточной обработки при разных параметрах N (10, 100, 1000, 100000) и M (2, 3, 4, 5, 10). Результаты представьте в табличной форме.
4. Выполните анализ эффективности при усложнении обработки каждого элемента вектора.
5. Исследуйте эффективность разделения по диапазону при неравномерной вычислительной сложности обработки элементов вектора.
0. Исследуйте эффективность параллелизма при круговом разделении элементов вектора. Сравните с эффективностью разделения по диапазону.

Методические указания

В работе исследуется эффективность распараллеливания независимой обработки элементов вектора. В первом задании в качестве обработки можно выбрать то или иное математическое преобразование элементов

вектора:

```
for(int i=0; i<a.Length; i++)
        b[i] = Math.Pow(a[i], 1.789);
```

Многопоточная обработка реализуется с помощью объектов Thread. На многоядерной системе многопоточная обработка приводит к параллельности выполнения. Классы для работы с потоками расположены в пространстве имен System.Threading.

Для создания потока необходимо указать имя рабочего метода потока, который может быть реализован в отдельном классе, в главном классе приложения как статический метод или в виде лямбда-выражения. Метод потока либо не принимает никаких аргументов, либо принимает аргумент типа object. Запуск потока осуществляется вызовом метода Start.

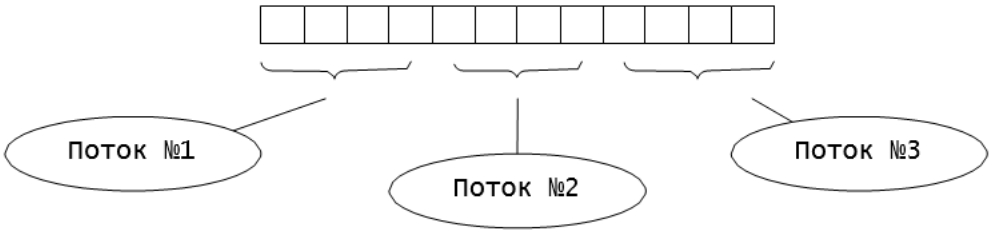
```
class Program
{
        static void Run(object some_data)
        {
                int m = (int) some_data;
                ..
        }
        static void Main()
        {
                ..
                Thread thr = new Thread(Run);
                thr.Start(some_data);
        }
}
```

Дождаться завершения работы потоков можно с помощью метода Join:

```
thr1.Join(); thr2.Join();
```

В функции потока необходимо предусмотреть возможность разбиения диапазона
0.. (N-1)на число потоков nThr. При запуске потока в качестве аргумента передается либо «индекс потока», определяющий область массива, который обрабатывается в данном

потоке, либо начальный и конечный индексы массива.



Многопоточное выполнение будет параллельным при наличии в вычислительной системе нескольких процессоров (ядер процессора). Число процессоров можно узнать с помощью свойства:

System.Environment.ProccessorCount;

Параллельное выполнение вычислений также можно реализовать с помощью классов библиотеки TPL (Task Parallel Library). Классы библиотеки располагаются в пространстве имен System.Threading.Tasks. Параллельное вычисление операций над элементами цикла выполняется с помощью метода Parallel.For:

```
Parallel.For(0, a.Length, i =>
    { b[i] = Math.Pow(a[i], 1.789); });
```

Для анализа производительности последовательного и параллельного выполнения можно использовать переменные типа DateTime.Например,

```
DateTime dt1, dt2;
dt1 = DateTime.Now;
// Вызов_вычислительной_процедуры;
dt2 = DateTime.Now;
TimeSpan ts = dt2 – dt1;
Console.WriteLine(“Total time: {0}“, ts.TotalMilliseconds);
```

Также можно использовать объект Stopwatch пространства System.Diagnostics:

```
Stopwatch sw = new Stopwatch();
sw.Start();
// Вызов_вычислительной_процедуры;
sw.Stop();
TimeSpan ts = sw.Elapsed;
Console.WriteLine(“Total time: {0}“, ts.TotalMilliseconds);
```

При оценке производительности необходимо учесть, что время выполнения алгоритма зависит от множества параметров. Поэтому желательно оценивать среднее время выполнения при нескольких прогонах алгоритма, исключая первый разогревающий прогон.

Эффективность параллельного алгоритма существенно зависит от элементов массива, числа потоков, сложности математической функции и т.д. Следует учитывать, что при малом объеме элементов массива, накладные расходы, связанные с организацией многопоточной обработки, превышают выигрыш от параллельности обработки. При последовательном выполнении примитивной циклической обработки быстродействие достигается за счет оптимального использования кэш-памяти.

Выполняя анализ зависимости быстродействия от числа потоков, следует учитывать число ядер процессора. Увеличение числа потоков сверх возможностей вычислительной системы приводит к конкуренции потоков и ухудшению быстродействия.

Усложнение обработки элементов массива предлагается реализовать с помощью внутреннего цикла. Например,

```
for(int i=0; i<a.Length; i++)
{
        // Обработка i-элемента
        for(int j=0; j < K; j++)
                b[i] += Math.Pow(a[i], 1.789);
}
```

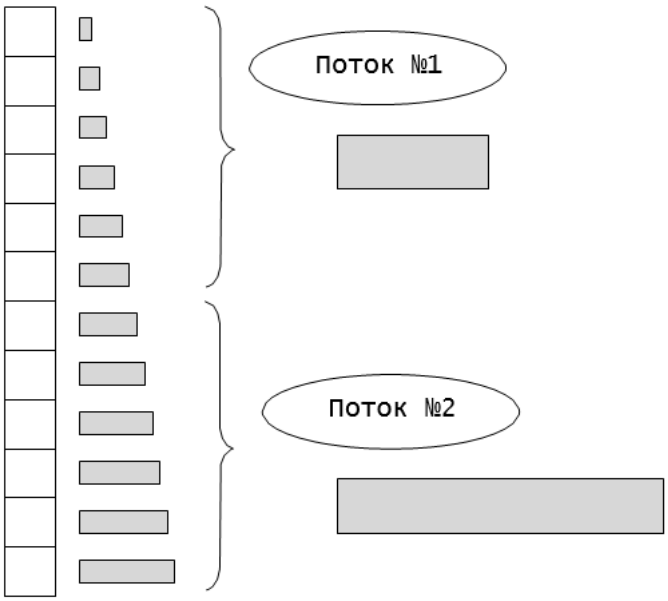
K – параметр «сложности». Увеличивая параметр K, наблюдаем повышение эффективности параллельной обработки при меньшем объеме массива чисел.

В рассмотренных вариантах обработки вычислительная нагрузка на каждой итерации относительно одинакова. В ситуациях, когда вычислительная нагрузка зависит от индекса элемента, разделение массива по равным диапазонам может быть не эффективно. Рассмотрим следующий вариант обработки:

```
for(int i=0; i<a.Length; i++)
{
        // Обработка i-элемента
        for(int j=0; j < i; j++)
                b[i] += Math.Pow(a[i], 1.789);
```

}

Вычислительная нагрузка при обработке i-элемента зависит от индекса i. Обработка начальных элементов массива занимает меньшее время по сравнению с обработкой последних элементов. Разделение данных по диапазону приводит к несбалансированной
загрузке потоков и снижению эффективности распараллеливания.



Одним из подходов к выравниванию загрузки потоков является применение круговой декомпозиции. В случае двух потоков получаем такую схему: первый поток обрабатывает все четные элементы, второй поток обрабатывает все нечетные элементы. Реализуйте круговую декомпозицию для нескольких потоков (больше двух).

ТЕМА 12. ОСНОВЫ ЯЗЫКА PYTHON.

Лабораторная работа № 16

1. Цель и порядок работы

Цель работы – начальное изучение средств работы со списками и строками в языке Питон.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по языку программирования Питон, выполняя указанные в тексте примеры.

Выполнить индивидуальное задание.

Выполнение заданий можно осуществлять в среде интерпретатора, позволявшей пошаговое выполнение команд.

2. Краткая теория

2.1 Списки

Списки в Питоне являются одним из основных типов данных. Их используют для группирования вместе нескольких значений и обозначают, как список элементов, разделенных запятыми, заключенный в квадратные скобки.

2.1.1 Общее понятие о работе со списками

Создадим однородный список из целых чисел (тип integer):

```
>>> b = [100, 1, 3, 4, 1234]
>>>
```

Убедимся, что мы действительно его создали:

```
>>> b
[100, 1, 3, 4, 1234]
```

Аналогичным образом создадим однородный список из чисел с плавающей запятой (тип float):

```
>>> c = [1.0, 3.14, 2.71]
>>> c
[1.0, 3.1400000000000001, 2.71]
```

Далее создадим однородный список элементами которого будут являться строковые переменные

(тип string):

```
>>> rgb = ['red', 'green', 'blue']
>>> rgb
['red', 'green', 'blue']
```

Не обязательно, чтобы элементы списка были одного типа.

```
>>> a = ['cat', 'dog', 'fish', 1, 2, 33, 8.0]
>>> a
['cat', 'dog', 'fish', 1, 2, 33, 8.0]
```

2.1.2 Нумерация элементов списка. Срезы. Многомерные списки.

Нумерация элементов

Все элементы списка пронумерованы. В общепринятой терминологии номер элемента списка называется индексом. Условимся называть начало списка (самый левый элемент) "головой", а конец (самый правый элемент) - "хвостом".

Нумерация элементов списка начинается с нуля. Например, для списка a из предыдущего раздела:

```
>>> a
['cat', 'dog', 'fish', 1, 2, 33, 8.0]
>>> a[0]
'cat'
>>> a[2]
'fish'
>>> a[3]
1
```

Номер элемента списка может быть и отрицательным. В этом случае - элемент "минус один" - это хвостовой, последний элемент списка, "минус два" - это предпоследний, "второй с хвоста" элемент:

```
>>> a
['cat', 'dog', 'fish', 1, 2, 33, 8.0]
>>> a[-1]
8.0
>>> a[-2]
33
```

К каждому элементу списка можно обратится по его индексу:

```
>>> b
[100, 1, 3, 4, 1234]
>>> b[0] = 1
>>> b
[1, 1, 3, 4, 1234]
>>> b[3] = b[4]
>>> b
[1, 1, 3, 1234, 1234]
```

Срезы.

Список может быть разделен на несколько частей. Это делается с помощью так называемых "срезов". Для того, чтобы осуществить срез следует указать название списка и в квадратных скобках после него указать начальный и конечный элемент среза:

```
>>> a
['cat', 'dog', 'fish', 1, 2, 33, 8.0]
>>> a[1:4]
['dog', 'fish', 1]
```

При срезе списка удобно мысленно нумеровать не элементы, а промежутки (интервалы) между ними. Это очень удобно для указания произвольных срезов. Перед нулевым (по индексу) элементом списка промежуток имеет номер 0, после него - 1 и так далее. Отрицательные значения отсчитывают промежутки с конца строки.

```
>>> a
['cat', 'dog', 'fish', 1, 2, 33, 8.0]
>>> a[2:3] ['fish']
>>> a[2:2]
[]
>>> a[2:5] ['fish', 1, 2]
```

Обратите внимание на то, что для a[2:2] был получен пустой список (это логично - если срез списка начинается и заканчивается в одном и том же интервале - это пустой список).

Если до или после двоеточия не указывается индекс, интерпретатор Питона истолковывает это как указание "до конца списка". Пример:

```
>>> a[:5]
['cat', 'dog', 'fish', 1, 2]
>>> a[3:]
[1, 2, 33, 8.0]
```

Вложенные (многомерные) списки

Список допускает, что его элементом может быть другой (вложенный) список:

```
>>> m = [1, 2, ['a', 'b', 'c'], 3, 8, 14]
>>> m[0]
1
>>> m[1]
2
>>> m[2]
['a', 'b', 'c']
>>> m[3]
3
```

Обратите внимание, что m[2] - это список ['a', 'b', 'c'] целиком.

Как обратится к одному из элементов этого вложенного списка? Нужно указать еще один, дополнительный индекс:

```
>>> m
[1, 2, ['a', 'b', 'c'], 3, 8, 14]
>>> m[2]
['a', 'b', 'c']
>>> m[2][0]
'a'
>>> m[2][1]
'b'
>>> m[2][:2] ['a', 'b']
```

2.1.3 Действия со списками

Списки допускают два типа действий - конкатенацию и размножение (мультипликацию). Конкатенация обозначает склейку списков и осуществляется следующим образом:

```
>>> s1 = [1, 2, 3, 4]
>>> s2 = [10, 20, 30, 40]
>>> s3 = ['a', 'b', 'c']
>>> sk = s1 + s2 + s3 + [2, 2]
>>> sk
```

```
[1, 2, 3, 4, 10, 20, 30, 40, 'a', 'b', 'c', 2, 2]
```

Размножение списков осуществляется с помощью знака умножения:

```
>>> s1
[1, 2, 3, 4]
>>> sm = s1 * 4
>>> sm
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

2.1.4 Методы работы со списками

Длина списка

Функция len(список) - определяет длину списка, указанного в качестве аргумента:

```
>>> sp = [1, 3, 3, 4, 5, 67, 7, 1, 1]
>>> sp
[1, 3, 3, 4, 5, 67, 7, 1, 1]
>>> len(sp)
9
>>> sp[1:]
[3, 3, 4, 5, 67, 7, 1, 1]
>>> len(sp[1:])
8
>>> sp[:4]

[1, 3, 3, 4]
>>> len(sp[:4])
4
>>> sp[1:4] [3, 3, 4]
>>> len(sp[1:4])
3
```

Генерация списка с последовательными номерами

Функция range(N) отвечает за генерацию списка из N элементов арифметической прогрессии. Напоминаем: счет элементов начинается с нуля:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(40)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39]
```

Также можно указать начальный элемент прогрессии:

```
>>> range(5,10) [5, 6, 7, 8, 9]
>>> range(5,20)
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> range(15,40)
[15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39]
```

Также можно указывать начальный элемент и шаг прогрессии:

```
>>> range(5,10,2) [5, 7, 9]
>>> range(15,40,5)
[15, 20, 25, 30, 35]
```

Обратите внимание: часто функция range() используется в связке с функцией len() - чтобы создать нумерованную последовательность списка. Это может значительно упрощать работу с обработкой списков:

```
>>> sk
[1, 2, 3, 4, 10, 20, 30, 40, 'a', 'b', 'c', 2, 2]
>>> range(len(sk))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

В дальнейшем будут приведены примеры использования такой связки в циклах обработки списков.

Добавление элемента к списку

Присоединение элемента к списку осуществляется с помощью функции append(). Обратите внимание на то, как эта функция используется в приведенных ниже примерах:

```
>>> k = [1, 2]
>>> k.append(3)
>>> k
[1, 2, 3]
```

```
>>> k.append(3)
>>> k
[1, 2, 3, 3]
>>> k.append(9)
>>> k

[1, 2, 3, 3, 9]
```

Сортировка списка

Сортировка списка осуществляется с помощью функции sort() и reverse(), которые отсортировывают список по порядку:

```
>>> sk
[1, 2, 3, 4, 10, 20, 30, 40, 'a', 'b', 'c', 2, 2]
>>> sk.sort()
>>> sk
[1, 2, 2, 2, 3, 4, 10, 20, 30, 40, 'a', 'b', 'c']
>>> sk.reverse()
>>> sk
['c', 'b', 'a', 40, 30, 20, 10, 4, 3, 2, 2, 2, 1]
```

Добавление элементов в список

Существует специальная функция insert(i,n), которая добавляет элемент n в интервал i (если вы забыли что такое интервал - перечитайте раздел "Срезы"):

```
>>> a = [1, 1, 2, 2, 6, 13, 1, 23, 2, 7]
>>> a.insert(1,7)
>>> a
[1, 7, 1, 2, 2, 6, 13, 1, 23, 2, 7]
>>> a.insert(3,7)
>>> a
[1, 7, 1, 7, 2, 2, 6, 13, 1, 23, 2, 7]
>>> a.insert(3,'a')
>>> a
[1, 7, 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]
```

Также можно обойтись и без этой функции - используя срезы. Вставим во второй интервал списка a строковую переменную 'b':

```
>>> a
[1, 7, 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]
```

```
>>> a = a[:2] + ['b'] + a[2:]
>>> a
[1, 7, 'b', 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]
```

Обратите внимание: 'b' вставляется в список не само по себе, а виде отдельного элемента- списка ['b']. В противном случае можно получить ошибку:

```
>>> a = a[:2] + 'b' + a[2:]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Удаление элементов из списка

За удаление элементов из списка отвечает команда del. Она может использоваться как для удаления одного из элементов, так и для удаления целых фрагментов списка, которые задаются с помощью срезов:

```
>>> a = [7, 'b', 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]
>>> a
[7, 'b', 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]
>>> del a[0]
>>> a
['b', 1, 'a', 7, 2, 2, 6, 13, 1, 23, 2, 7]

>>> len(a)
12
>>> del a[10:]
>>> a
['b', 1, 'a', 7, 2, 2, 6, 13, 1, 23]
>>> del a[4:6]
>>> a
['b', 1, 'a', 7, 6, 13, 1, 23]
```

Подсчет элементов в списке

Функция count() производит подсчет элементов в списке:

```
b = [1, 6, 3, 3, 4, 9, 2, 3, 4]
b.count(3) 3
b.count(1) 1
b.count(4) 2
b.count(-1) 0
```

В последнем случае был задан элемент, который отсутствует в списке - поэтому в качестве результата был получен ноль.

Поиск вхождения в список первого элемента

За поиск вхождения в список первого элемента отвечает функция index():

```
>>> b = [1, 6, 3, 3, 4, 9, 2, 3, 4]
>>> b.index(3)
2
>>> b.index(9)
5
```

Если элемент отсутствует - будет выдана ошибка:

```
>>> b.index(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Удаление из списка элементов с заданным значением

Функция remove() удаляет из списка первое вхождение элемента с заданным значением:

```
>>> b = [1, 6, 3, 3, 4, 9, 2, 3, 4]
>>> b
[1, 6, 3, 3, 4, 9, 2, 3, 4]
>>> b.remove(3)
>>> b
[1, 6, 3, 4, 9, 2, 3, 4]
>>> b.remove(3)
>>> b
[1, 6, 4, 9, 2, 3, 4]
>>> b.remove(9)
>>> b
[1, 6, 4, 2, 3, 4]
```

2.2 Строки

2.2.1 Общее понятие о работе со строками

Строковые переменные в языке Питон обозначаются одинарной либо двойной кавычками. Можно использовать любое из этих обозначений:

```
>>> s = 'string 1'
>>> s
'string 1'
>>> s = "string 1"
>>> s
'string 1'
>>> s = "string 2"
>>> s
'string 2'
```

Если переменная занимает больше одной строки - можно использовать для ее обозначения тройные кавычки. В приведенном ниже примере троеточия не вводятся:

```
>>> s = ' ' '
... first line
... second line
... third line
... ' ' '
>>> s
'\nfirst line\nsecond line\nthird line\n'
```

Обратите внимание, что введенная переменная вытянута в одну линию, разбитую на абзацы с помощью спецсимвола \n, обозначающего конец строки. Чтобы посмотреть переменную "с абзацами" следует использовать команду print:

```
>>> print s

first line
second line
third line
```

Когда нужно набрать очень длинную одиночную строку - можно использовать символ "бэкслэш" -
\ - "обратный слэш":

```
>>> x = 'this is \
... multilong\
... string \
... but it \
```

```
... may  view  \
... as  one line'
>>>  x
'this  is multilongstring  but  it may  view  as  one line'
```

Обратите внимание на местонахождение пробела перед обратным слэшем.

Спецсимволы

С одним из спецсимволов мы уже сталкивались в предыдущем разделе - это \n - символ перевода строки. Так же очень распространенными символами являются \t - символ табуляции, \r - символ возврата каретки, \a - bell (звонок). Обратите внимание на то, что чтобы увидеть символы "в действии", нужно распечатать переменную с помощью команды print, а в обычном режиме обращения спецсимволы показываются как они есть - чтобы ими было удобно манипулировать:

```
>>>  z =  'new  line\ndemo'
>>>  z
'new  line\ndemo'
>>>  print  z
new  line
demo
>>>  z =  'tabulation\tdemo'
>>>  z
'tabulation\tdemo'
>>>  print  z

tabulation    demo
```

Экранирование спецсимволов

Если необходимо оперировать со строками в которых есть символ "бэкслэш" - это может вызвать проблемы. Например, такая ситуация может возникать, при указании путей в файловых системах fat32 или ntfs (используемых в семействе ОС MS Windows).

```
>>>  path  =  "D:\temp\new_file.txt"
>>>  path
'D:\temp\new_file.txt'
>>>  print  path
D:      emp
ew_file.txt
```

Здесь интерпретатор Питона посчитал кусочки строк \t и \n - символами табуляции и новой строки соответственно. Что привело к ошибкам.

Чтобы такого не происходило - бэкслэш необходимо экранировать. Это делается с помощью второго бэкслэша, который экранирует использование первого. Таким образом, если для питона \n - это символ new line ("новая строка"), то \\n - это просто символ бэкслэша, за которым следует символ n:

```
>>> back = 'new line\ndemo'
>>> back
'new line\ndemo'
>>> print back
new line
demo
>>> back = 'new line\\ndemo'
>>> back
'new line\\ndemo'
>>> print back
new line\ndemo
```

Аналогично это действует для других спецсимволов:

```
>>> back = 'tabulation\tdemo'
>>> back
'tabulation\tdemo'
>>> print back
tabulation    demo
>>> back = 'tabulation\\tdemo'
>>> back
'tabulation\\tdemo'
>>> print back
tabulation\tdemo
```

Поэтому при указании путей в ОС MS Windows следует использовать двойной слэш:

```
>>> path = "D:\\temp\\new_file.txt"
>>> path
'D:\\temp\\new_file.txt'
>>> print path
D:\temp\new_file.txt
```

Здесь все в порядке.

Есть еще один способ. Можно поставить перед кавычками строки строчную букву r - например

так: r"D:\temp\new_file.txt". Это указание для Питона воспринимать содержимое кавычек в формате raw - как есть, без спецсимоволов:

```
>>> path = r"D:\temp\new_file.txt"

>>> path
'D:\\temp\\new_file.txt'
>>> print path
D:\temp\new_file.txt
```

Эти нюансы работы со строками будут важны в дальнейшем, когда мы перейдем к операциям с файлами и работой в файловой системе.

Юникод

Строчная u перед кавычками, обозначающими строку, указывает на то, что эта строка задана в

Юникоде (подробнее этот вопрос будет рассмотрен в дальнейшем).

2.2.2 Нумерация симоволов строки. Срезы

Нумерация строк в целом аналогична нумерации элементов списка. Только вместо элементов списка в строке нумеруются символы.

```
>>> br = 'lazy brown fox'
>>> br[0]
'l'
>>> br[2]
'z'
>>> br[5]
'b'
>>> br[4]
' '
>>> br[-1]
'x'
>>> br[-2]
'o'
>>> br[-3]
'f'
```

Обратите внимание на то, что спецсимволы \n, \t и так далее - с точки

зрения интерпретатора являются одиночными символами:

```
>>> br = 'lazy\nbrown\tfox'
>>> br
'lazy\nbrown\tfox'
>>> print br
lazy
brown	fox
>>> br[3]
'y'
>>> br[4]
'\n'
>>> br[5]
'b'
>>> br[9]
'n'
>>> br[10]
'\t'
>>> br[11]
'f'
```

Работа со срезами, так же аналогична тому как это организовано у списков, за тем исключением, что нумеруются не интервалы между списками, а интервалы между символами строки:

```
>>> br = 'lazy brown fox'
>>> br[5:]
'brown fox'
>>> br[:4]

'lazy'
>>> br[5:10]
'brown'
>>> br[-3:]
'fox'
```

2.2.3 Работа со строками. Склейка и размножение

Работа со строками в целом аналогична работет со списками, за одним исключением - строковые переменные нельзя менять посимвольно. Вместо этого можно использовать срезы и операции склейки.

Строки допускают те же операции, что и списки. С ними можно производить операции склейки:

```
>>> z = "help"
>>> y = "me"
>>> x = z + y
>>> x
'helpme'
>>> x = z + " " + y
>>> x
'help me'
```

Аналогично происходить операция размножения:

```
>>> z * 3
'helphelphelp'
>>> y * 10
'memememememememememe'
>>> x = z + " "*7 + y
>>> x
'help me'
>>> x = (z + " ") * 7 + y
>>> x
'help help help help help help help me'
```

Хотя мы не можем изменить отдельный элемент строки:

```
>>> br = 'lazy brown fox'
>>> br[1] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Мы можем работать со склейкой и срезами так, чтобы добиться нужного эффекта:

```
>>> br
'lazy brown fox'
>>> br = br[:1]+'z'+br[2:]
>>> br
'lzzy brown fox'
```

2.2.4 Функции работы со строками

Длина строки

Функция len() работает аналогично функции len() для списков:

```
>>> br = 'lazy brown fox'
>>> len(br)
14
```

Переход от числовой переменной к строке и обратно

Функции str() (переход к символьному виду), float() (переход к числу с плавающей запятой)
и int() (переход к целому числу) обеспечивают операции переходов между типами:

```
>>> a = '2010'
>>> a
'2010'
>>> int(a)
2010
>>> float(a)
2010.0
>>> pi = 3.1415926
>>> pi
3.1415926000000001
>>> str(pi)
'3.1415926'
>>> int(pi)
3
>>> year = 2010
>>> str(year)
'2010'
>>> float(year)
2010.0
```

Получение ascii-кодов символов

Функция ord() обеспечивает получение ascii-кода символа, chr() - получение символа по его ascii-коду:

```
>>> print chr(34)
"
>>> print chr(36)
$
>>> print chr(37)
%
```

```
>>> ord('$')
36
>>> ord('@')
64
>>> ord('f')
102
>>> ord('z')
122
```

Ввод с клавиатуры

Функция raw_input('приветствие') останавливает выполнение программы, печатает приветствие и ждет ввода с клавиатуры, завершающегося нажатием <Enter>. При этом функция возвращает набранное пользователем в виде строки:

```
>>> a = raw_input('Input please: ')
Input please: Please
>>> a
'Please'
```

Как правило используется в сочетании с функцией int() или float(), чтобы сразу преобразовать введенное число в нужный тип:

```
>>> a = int(raw_input('Input integer number: '))
Input integer number: 888
>>> a

888
>>> f = float(raw_input('Input real number: '))
Input real number: 3.14
>>> f
3.1400000000000001
```

3. Варианты заданий

3.1 Списки

Вариант студента определяется по последней цифре в списке студентов в группе.

Задайте переменную spisok, в которую занесите любые семь элементов из следующего набора данных (выбирается согласно варианту):

1. Список дней недели.
2. Название любых семи фильмов.
3. Список из семи любых названий месяцев.
4. Название любых семи стран.
5. Список из семи любых названий предметов в расписании.
6. Список из семи любых предметов в вашем рюкзаке.
7. Название любых семи улиц города.
8. Название любых семи городов.
9. Название любых семи марок автомобилей.
0. Название любых семи деталей компьютера.

Общее задание для всех вариантов:

• Распечатайте первый, третий, четвертый, последний и предпоследний элементы списка.

• Разбейте с помощью срезов список на три части - в первой части три элемента, во второй - один, в третьей - тоже три.

• Создайте новый список mnogo_spisok, в который бы в виде элементов входили три части из предыдущего задания (то есть первый элемент mnogo_spisok - это под-список из первых трех элементов списка spisok, второй элемент - это один, средний элемент списка spisok и так далее)

• Поменяйте местами первый и последний элементы mnogo_spisok

• Добавьте к mnogo_spisok еще один, любой элемент

• Добавьте еще один, любой элемент к любому из подсписков mnogo_spisok

• Удалите из списка mnogo_spisok все элементы кроме первого

3.2 Строки

Варианты заданий определяться по номеру в списке студентов:

1 вариант 1,21 задание.
2 вариант 2,22 задание.
3 вариант 3,23 задание.
4 вариант 4,24 задание.
5 вариант 5,25 задание.
6 вариант 6,26 задание.
7 вариант 7,27 задание.
8 вариант 8,28 задание.
9 вариант 9,29 задание.
10 вариант 10,30 задание.
11 вариант 11,31 задание.
12 вариант 12,32 задание.
13 вариант 13,33 задание.
14 вариант 14,34 задание.

15 вариант 15,35 задание.
16 вариант 16,36 задание.
17 вариант 17,37 задание.
18 вариант 18,38 задание.
19 вариант 19,39 задание.
20 вариант 20,40 задание.

1) Написать программу, которая преобразует строку таким образом, что цифры, которые находятся в слове, переносятся в конец строки без изменения порядка следования остальных символов.
2) Написать программу, которая, если строка начинается и оканчивается одним и тем же знаком, во всей строке заменяет этот знак четвертым символом строки.
3) Дана строка состоящая из слов, разделенных пробелами (одним ил несколькими). Вывести строку содержащую эти же слова (разделенные одним пробелом), но расположенные в обратном порядке.
4) Дана строка символов. Для сохранения ее в сжатом виде найти максимальную последовательность символов произвольной длины, которая повторяется, и заменить ее своим кодом.
5) В строку S добавить необходимое количество пробелов так, чтобы ее длина стала равна n. Причем: перед первым словом пробелы не добавлять, после последнего слова все пробелы удалить, добавленные пробелы равномерно распределить между словами. Если длина S превосходит n, удалить S из все слова, которые не укладываются в первые n символов, а оставшуюся часть преобразовать по вышеуказанным правилам.
6) Палиндромом называют последовательность символов, которая читается как слева направо, так и справа налево. Найти во введённой строке подстроку-палиндром максимальной длины.
7) Вывести сообщение "МОЖНО", если из букв введённой строки X можно составить введенную строку Y, при условии, что каждую букву строки X можно использовать один раз; и сообщение "НЕЛЬЗЯ" в противном случае.
8) По введённому числовому значению N (0 < N < 4000) вывести его запись в римской системе счисления. Римская система счисления использует 7 цифр (I=1 V=5 X=10 L=50 C=100 D=500 M=1000).
9) Дана строка символов. Найти слово, в котором число различных символов минимально. Если таких слов несколько, найти первое из них.
10) Дана строка символов. Найти количество слов, содержащих только символы латинского алфавита, а среди них – количество слов с равным числом гласных и согласных букв.

11) Дана строка символов. Найти слово, символы в котором идут в строгом порядке возрастания их кодов. Если таких слов несколько, найти первое из них.
12) Дана строка символов. Найти слово, состоящее только из различных символов. Если таких слов несколько, найти первое из них.
13) Заменить в данной строке все вхождения подстроки s на порядковый номер вхождения. Подстрока s вводится с клавиатуры.
14) Ьтасипан уммаргорп адовереп йоннадаз икортс оп умещюуделс упицнирп.
15) В строке переставить местами рядом стоящие слова.
16) Из строки вырезать слова, стоящие на четном месте.
17) В строке удалить все пробелы, а затем после каждой пятой буквы вставить знак вопроса.
18) Переставить местами слова в строке (первая с последней и т.д.).
19) Из строки удалить все встречающиеся символы.
20) Все слова в строке расположить в алфавитном порядке.
21) Подсчитать сколько раз в данной строке встречается некоторая последовательность букв, введенная с клавиатуры.
22) В строке после каждого слова вставить запятую.
23) Каждое слово в строке распечатать с новой строчки экрана.
24) В строке удалить последнюю букву у слов.
25) В строке все запятые заменить точкой, и перед первым словом вставить слово STRING.
26) Подсчитать количество слов и букв в этих словах в строке.
27) В строке удалить все знаки препинания.
28) С клавиатуры считывается строка состоящая из цифр от 0 до 9. Разбить ее на две части, полученные строки преобразовать к целочисленному типу.
29) В строке каждый символ заменить на соответствующий ему код.
30) В строке удалить каждый заданный символ, а остальные продублировать.
31) В строке посчитать наибольшее количество идущих подряд пробелов.
32) В строке удалить лишние пробелы.
33) В строке подсчитать количество слов начинающихся с заданной буквы.
34) В строке удалить все символы, не являющиеся буквами или цифрами.
35) В строке найти самое длинное симметричное слово.
36) В строке оставить только те символы, которые встречаются один раз.
37) В строке указать слово, в котором количество гласных букв минимально.

38) В строке указать слово, в котором количество согласных букв максимально.
39) В строке удалить все заданные группы букв.
40) В строке подсчитать количество слов

ТЕМА 13. ОРГАНИЗАЦИЯ РАБОТЫ С ФАЙЛАМИ В PYTHON.

Лабораторная работа № 17

1. Цель и порядок работы

Цель работы – изучить возможности работы с файлами в Питоне.

Порядок выполнения работы:
Самостоятельно проработать теоретические основы по языку программирования Питон, выполняя указанные в тексте примеры.
Выполнить индивидуальное задание.

2. Краткая теория

2.1 Общие замечания

2.1.1 Вопрос именования адресов

Пути задаются либо с удвоением симола бэкслэш (например 'D:\\txt.txt'), либо с указанием r ('raw') перед кавычками r'D:\\txt.txt'. Это делается для того, чтобы избежать случаев, когда символы \n или \t принимаются за спецсимволы.

2.1.2 Порядок работы с файлами

Общий порядок работы с файлами выглядит следующим образом - с произвольной переменной с помощью функции open ('путь к файлу', 'режим работы с файлом') связывается определенный файл, после чего все операции с данным файлом выполняются с помощью этой переменной. В конце операций файл отключается от переменной с помощью функции имя_переменной.close().

Для следующего примера нам понадобится текстовый файл. Создадим его и назовем 1.txt
(используйте любой текстовый редактор) со следующим текстом:

1, 2, 3
one two three
this is end of the file

Сохраните файл на диск C. Тогда путь к нему будет выглядеть как C:\ 1.txt.

Для начала посмотрим как файл подключается к произвольной

переменной. В командной строке наберите:

```
>>> a = open(' C:\\ 1.txt', 'rb')
```

Этим вы свяжете файл 1.txt (файл-как-объект) с переменной a. Эта переменная является "отражением" файла в пространстве Питона. Проверим это (вывод вашей среды может отличаться):

```
>>> a
<open file ' C:\\ 1.txt', mode 'rb' at 0xb780b430>
```

Это указывает на то, что переменная a связана с файлом 1.txt. Причем файл 1.txt открыт в режиме чтения.

Тип переменной a можно будет сделать при помощи функции type()

```
>>> type(a)
<type 'file'>
```

2.2 Режимы работы с файлом

Режим работы задается с помощью текстовой строки, которая может принимать следующие значения:

• r - (read) Файл открывается для чтения. Файл должен уже существовать.

• w - (write) Файл открывается для записи. Если файл не существует - он будет создан. При этом он будет перезаписан, если там что-то уже находилось.

• a - (append) Если файл не существует - он будет создан. Файл открывается для записи, при этом новая запись будет добавлена к содержимому файла.

• r+ - (read+) Позволяет сразу чтение и запись. Если файл не существует - он будет создан.

При этом он будет перезаписан, если там что-то уже находилось.

• a+ - (append+) Позволяет сразу чтение и запись. Если файл не существует - он будет создан.

При этом новая запись будет добавлена к содержимому файла.

Кроме того, к режиму дописывается постфикс обозначающий - как именно следует читать файл. Если там указано b - (binary) Питон будет работать с файлом, как с двоичным файлом любого формата, в противном

случае - он будет открыт как текстовый файл.

Обратите внимание на то, что мы открыли файл, как двоичный (режим 'rb') - это основная практика при работе с платформой Windows.

Также обратите внимание на то, что режимы и путь к файлу - это обычные текстовые строки, которые могут быть заданы как явно, так и неявно (в виде переменной, которая формируется в зависимости от результатов вычислений).

2.3 Методы работы с файлом

Все методы работы с файлом-объектом a можно посмотреть с помощью директивы dir():

```
>>> dir(a)
['__class__', '__delattr__', '__doc__', '__enter__', '__exit__', '__format__',
'__getattribute__', '__hash__', '__init__', '__iter__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush',
'isatty', 'mode', 'name', 'newlines', 'next', 'read', 'readinto', 'readline',
'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

С частью из этих функций мы познакомимся ниже.

2.4 Чтение файла: read()

Функция read() считывает файл целиком, одной длинной строкой с разделителями \n в тех местах, где стоят абзацы. Если указывается количество байтов (например read(15)) - то считывается именно это количество, если нет - то считывается весь файл целиком.

Прочитаем файл:

```
>>>  a.read()
'1,  2,  3\none  two  three\nthis  is end of  the  file\n'
```

Перемещение "указателя", с которого начинается чтение, вдоль байтовой структуры файла осуществляется с помощью двух функций - tell() и seek(). Первая из них возвращает число байтов от начала файла до текущего положения, вторая - перемещает указатель на нужное количество байтов. Функция seek() задается в форме seek('на сколько байтов переместиться', 'точка отсчета'), при этом точка отсчета может принимать следующие

значения:

- 0 - начало файла
- 1 - текущее положение указателя
- 2 - конец файла

Переместим указатель к началу файла:

```
>>>  a.seek(0,0)
>>>  a.tell()
0L
```

Теперь считаем из файла четыре байта:

```
>>>  a.read(4)
'1,  2'
>>>  a.tell()
4L
```

Мы считали четыре байта и указатель сместился на четыре байта к концу файла. Обратите внимание, что в данной кодировке один байт равен одному символу. Это далеко не всегда так, (например, это неверно если файл записан в кодировке Unicode), но в данном случае позволяет легко ориентироваться в работе файловых функций).

```
>>>  a.seek(2,1)
>>>  a.tell()
6L
>>>  s =  a.read(7)
>>>  s
'3\none  t'
```

И так далее.

```
>>>  a.seek(-12,2)
>>>  a.read()
'of  the  file\n'
>>>  a.seek(14,0)
>>>  a.tell()
14L
>>>  a.read()
'o  three\nthis  is end of  the  file\n'
```

```
>>> a.tell()
46L
```

После работы с файлом, его нужно закрыть "отвязав" от переменной a:

```
>>> a.close()
>>> a
<closed file '1.txt', mode 'rb' at 0xb780b430>
```

2.5 Чтение файла: readlines()

Функция read(), является базовой, но не всегда удобной для работы с файлом функцией. Наиболее удобная в Питоне функция называется readlines() - она считывает весь файл сразу в виде списка из строк:

```
>>> a = open('1.txt', 'rb')
>>> a.readlines()
['1, 2, 3\n', 'one two three\n', 'this is end of the file\n']
```

Здесь мы считываем сразу весь файл и дальше можем работать со списком всеми традиционными средствами питона. Например, создадим скрипт, распечатывающий первые буквы каждой строки файла:

```
a = open('1.txt','rb')
s = a.readlines()
for i in s:
        print i
```

Результат работы:

```
1
o
t
```

Обратите внимание на то, что строки записываются в список вместе со знаком перевода строки \n. Если вам требуется строка "сама по себе" ее можно, обрезать - например с помощью среза:

```
a = open('1.txt','rb')
s = a.readlines()

print 'all lines:' print
for i in s:
```

```
        print  i,

print  'all  lines  in  one line:' print
for  i in  s:
        print  i[:-1],
```

Результат работы:

```
all  lines:

1,  2,  3
one two  three
this  is end of  the  file all  lines  in  one line:

1,  2,  3  one two  three  this  is end of  the  file
```

2.6 Запись в файл: write()

Запись в файл производится с помощью функции write(). Вызов функции вида имя_файла.write('строка') записывает в файл (дописывает или записывает - определяется режимом открытия файла) строку 'строка'. Режим открытия файла определяет - как именно будет туда записана строка.

Обратите внимание - при записи в файл нужно самому заботиться о сохранении знаков перевода строки.

Также обратите внимание, что чтобы записать в файл что-то "не-строчное" его нужно преобразовать в строку с помощью функции str()

Введем скрипт следующего примера:

```
b  =  open('2.txt','wb')
for  i in  range(29):
        b.write(str(i))
```

Результатом работы будет файл 2.txt со следующим содержанием:

```
0123456789101112131415161718192021222324252627 28
```

Если мы изменим его:

```
b  =  open('2.txt','wb')
for  i in  range(29):
        b.write(str(i)  +  '  ')
```

То цифры в файле будут разделены пробелами:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
```

А если мы позаботимся о абзацах:

```
b = open('2.txt','wb')
for i in range(29):
        b.write(str(i) + '\n')
```

Получим:

```
0
1
2
3
4
5

...

23
24
25
26
27
28
```

2.7 Полный цикл работы чтения-записи файла

Пример ниже иллюстрирует работу с чтением и записью файла. Он меняет местами первую строку файла с последней. В данном случае мы читаем файл 1.txt, считываем из него информацию, меняем местами первую и последнюю строки файла и записываем получившийся список в файл 3.txt.

```
a = open('1.txt','rb')
old = a.readlines()
a.close()

first = old[0]
```

```
last  =  old[-1]
new  =  old
new[0]  =  last
new[-1]  =  first

b  =  open('3.txt','wb')
for  i in  new:
        b.write(i)
```

Файл 1.txt (исходный файл):

```
1,  2,  3
one two  three
this  is end of  the  file
```

Файл 3.txt (файл с результатом работы программы):

```
this  is end of  the  file one two  three
1,  2,  3
```

3. Вопросы для самопроверки

• Где Питон будет искать файл, если указано только имя файла?

• Каких правил следует придерживаться, задавая путь к файлу в системе MS Windows?

• Опишите последовательность действий с файлом при его открытии. В каком режиме возможно открытие файла? Как указывается путь к нему, какие есть варианты указания такого пути? Приведите пример.

• Перечислите и опишите режимы работы с файлом и их отличия друг от друга.

• Опишите последовательность действий с файлом при его закрытии. Как указывается путь к нему, какие есть варианты указания такого пути? Приведите пример.

• Как в общем случае организуется последовательность чтения-записи из-в файл? Приведите пример.

• Опишите работу команды read() Приведите пример ее использования.

• Опишите работу команды seek() Приведите пример ее использования.

• Опишите работу команды tell() Приведите пример ее использования.

• Опишите работу команды readline() Приведите пример ее использования.

• Опишите работу команды readlines() Приведите пример ее использования.

• Опишите работу команды write() Приведите пример ее использования.

• Какие преимущества предоставляет команда readlines() по сравнению с командой read()?

4. Варианты заданий

Вариант определяется по последней цифре в списке студентов группы.

Вариант 1.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- фамилия и инициалы;
- номер группы;
- успеваемость (пять оценок).

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на дисплей фамилий и номеров групп для всех студентов, если средний балл студента больше 4.0, если таких студентов нет, вывести соответствующее сообщение.

Вариант 2.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- фамилия и инициалы;
- номер группы;
- успеваемость (пять оценок).

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2, если таких студентов нет, вывести соответствующее сообщение.

Вариант 3.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о рейсе, номер которого введен с клавиатуры, если таких рейсов нет, вывести соответствующее сообщение.

Вариант 4.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- название пункта назначения рейса;
- номер рейса;
- тип самолета.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры, если таких рейсов нет, вывести соответствующее сообщение.

Вариант 5.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- фамилия и инициалы работника;
- название занимаемой должности;
- зарплату;
- год поступления на работу.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры, если таких работников нет, вывести соответствующее сообщение.

Вариант 6.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени, если таких поездов нет, вывести соответствующее сообщение.

Вариант 7.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- название пункта назначения;
- номер поезда;
- время отправления.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о пункте назначения, в который отправляется поезд, номер которого введен с клавиатуры, если таких поездов нет, вывести соответствующее сообщение.

Вариант 8.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о маршруте, номер которого введен с клавиатуры, если таких маршрутов нет, вывести соответствующее сообщение.

Вариант 9.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- фамилия и имя;
- номер телефона;
- дата рождения.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о человеке, номер телефона которого введен с клавиатуры, если таких людей нет, вывести соответствующее сообщение.

Вариант 0.

Текстовый файл может содержать данные следующего типа записанные через соответствующий разделить:

- расчетный счет плательщика;
- расчетный счет получателя;
- перечисляемая сумма в руб.

Написать программу, выполняющую следующие действия:

1. ввод с клавиатуры данных и запись их в файл;
2. вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры, если таких счетов нет, вывести соответствующее сообщение.

ТЕМА 14. ФУНКЦИИ В PYTHON.

Лабораторная работа № 18

1. Цель и порядок работы

Цель работы – изучить возможности работы с функциями в Питоне.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по языку программирования Питон, выполняя указанные в тексте примеры.

Выполнить индивидуальное задание.

2. Краткая теория

Функции в Python

Функция - это блок организованного, многократно используемого кода, который используется для выполнения конкретного задания. Функции обеспечивают лучшую модульность приложения и значительно повышают уровень повторного использования кода.

Существуют некоторые правила для создания функций в Python.

Блок функции начинается с ключевого слова def, после которого следуют название функции и круглые скобки ().

Любые аргументы, которые принимает функция должны находиться внутри этих скобок. После скобок идет двоеточие «:» и с новой строки с отступом начинается тело функции.

Пример функции в Python:

```
def my_function(argument):
    print argument
```

Вызов функции

После создания функции, ее можно исполнять, вызывая из другой функции или напрямую из оболочки Python. Для вызова функции следует ввести ее имя и добавить скобки.

Например:

```
my_function("abracadabra")
```

Аргументы функции в Python

Вызывая функцию, мы можем передавать ей следующие типы аргументов:

Обязательные аргументы (Required arguments)

Аргументы-ключевые слова (Keyword argument)

Аргументы по умолчанию (Default argument)

Аргументы произвольной длины (Variable-length argumens)

Если при создании функции мы указали количество передаваемых ей

аргументов и их порядок, то и вызывать ее мы должны с тем же количеством аргументов, заданных в нужном порядке.

Например:

```
def bigger(a,b):
    if a > b:
        print a
    else:
        print b
#Корректное использование функции
bigger(5,6)
#Некорректное использование функции
bigger(3)
bigger(12,7,3)
```

Аргументы - ключевые слова используются при вызове функции. Благодаря ключевым аргументам, вы можете задавать произвольный (то есть не такой каким он описан при создании функции) порядок аргументов.

Например:

```
def person(name, age):
    print name, "is", age, "years old"
#Хотя в описании функции первым аргументом идет имя, мы можем
#вызвать функцию вот так
person(age=23, name="John")
```

Аргумент по умолчанию, это аргумент, значение для которого задано изначально, при создании функции.

Например:

```
def space(planet_name, center="Star"):
    print planet_name, "is orbiting a", center
#Можно вызвать функцию space так:
space("Mars")
#В результате получим: Mars is orbiting a Star
#Можно вызвать функцию space иначе:
space("Mars", "Black Hole")
#В результате получим: Mars is orbiting a Black Hole
```

Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. В этом случае следует использовать аргументы произвольной длины. Они задаются произвольным именем переменной, перед которой ставится звездочка (*).

Например:

```
def unknown(*args):
        for argument in
                args: print
                argument
unknown("hello","world") #напечатает оба слова, каждое с новой строки
unknown(1,2,3,4,5) # напечатает все числа, каждое с новой строки
unknown() # ничего не выведет
```

Ключевое слово return

Выражение return прекращает выполнение функции и возвращает указанное после выражения значение. Выражение return без аргументов это то же самое, что и выражение return None. Соответственно, теперь становится возможным, например, присваивать результат выполнения функции какой либо переменной.

Например:

```
def bigger(a,b):
        if a > b:
                return a # Если а больше чем b, то возвращаем b
                        # и прекращаем выполнение функции
        return b  # Незачем использовать else. Если мы дошли
                  # до этой строки, то b, точно не меньше чем a
# присваиваем результат функции bigger переменной
num  num = bigger(23,42)
```

Область видимости

Некоторые переменные скрипта могут быть недоступны некоторым областям программы. Все зависит от того, где вы объявили эти переменные.

В Python две базовых области видимости переменных:

- Глобальные переменные.
- Локальные переменные.

Переменные объявленные внутри тела функции имеют локальную область видимости, те что объявлены вне какой-либо функции имеют глобальную область видимости.

Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

```
# глобальная переменная
age age = 44
```

```
def info():
    print age # Печатаем глобальную переменную age
def local_info():
    age = 22 # создаем локальную переменную
    age print age
info() # напечатает 44
local_info() # напечатает 22
```

Важно помнить, что для того чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. Однако, если перед нами стоит задача изменить глобальную переменную внутри функции - необходимо использовать ключевое слово global.

Например:

```
#глобальная переменная
age age = 13
#функция изменяющая глобальную переменную
def get_older():
    global age
    age += 1
print age # напечатает 13
get_older() # увеличиваем age на 1
print age # напечатает 14
```

Рекурсия

Рекурсией в программировании называется ситуация, в которой функция вызывает саму себя. Классическим примером рекурсии может послужить функция вычисления факториала числа.

Напомним, что факториалом числа, например, 5 является произведение всех натуральных (целых) чисел от 1 до 5. То есть, 1 * 2 * 3 * 4 * 5

Рекурсивная функция вычисления факториала на языке Python будет выглядеть так:

```
def fact(num):
    if num == 0:
        return 1 # Факториал нуля равен единице
    else:
        return num * fact(num - 1)
        # возвращаем результат произведения num
        #и результата возвращенного функцией fact(num - 1)
```

Однако следует помнить, что использование рекурсии часто может быть неоправданным. Дело в том, что в момент вызова функции в оперативной памяти компьютера резервируется определенное количество памяти, соответственно чем больше функций одновременно мы запускаем -

тем больше памяти потребуется, что может привести к переполнению стека (stack overflow) и программа завершится аварийно, не так как предполагалось. Учитывая это, там где это возможно, вместо рекурсии лучше применять циклы.

Анонимные функции, инструкция lambda

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции lambda. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

```
>>>func = lambda x, y: x + y
>>>func(1, 2)
3
>>>func('a', 'b')
'ab'
>>>(lambda x, y: x + y)(1, 2)
3
>>>(lambda x, y: x + y)('a', 'b')
'ab'
```

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

```
>>>func = lambda *args: args
>>>func(1, 2, 3, 4)
(1, 2, 3, 4)
```

3 Контрольные вопросы

1. Определение функции?
2. Объявление функции?
3. Какие типы аргументов существуют у функции?
4. Как осуществляется вызов функции?
5. Что такое анонимные функция?
6. Что такое рекурсия?
7. Какие виды рекурсии вы знаете?

4 Задание

1. Написать программу в соответствии с вариантом задания из пункта 5.1.
2. Отладить и протестировать программу.
3. Написать программу в соответствии с вариантом задания из пункта 5.2.
4. Отладить и протестировать программу.
5. Написать программу в соответствии с вариантом задания из пункта 5.3.
6. Отладить и протестировать программу.
7. Написать программу в соответствии с вариантом задания из пункта 5.4.
8. Отладить и протестировать программу.

5 Варианты заданий

5.1 Функции, параметры функций

Варианты заданий выбирать согласно номера в списке группы.

Определить функции, выполняющие действия в соответствии с вариантом задания, по одной на каждый способ передачи параметров. Написать программу, осуществляющую вызов этих функций несколько раз с различными типами параметров.

1. Вычислить с использованием подпрограммы – функции Z=НОД(a,b)+НОК(a,b), где a, b – целые положительные числа, НОД – наибольший общий делитель, НОК – наименьшее общее кратное.
2. Определить функцию нахождения расстояния между точками. Во множестве точек на плоскости найти пару точек с максимальным расстоянием между ними.
3. Найти наибольшую из высот треугольника. Известны две стороны треугольника и угол между ними.
4. Найти: y=среднее(a,b,c)/min(a,b,c).
5. Даны действительные числа s,t. Получить g(1.2,s) + g(t,s) - g(2s-1,st), где $g(a,b)=\frac{a^2-b^2}{2\cdot a\cdot b-a-b}+(a+b)\cdot\sqrt{\frac{|a+b|}{2}}$

6. Вычислить сумму значений функций

$$Z = f(\sin(x)+\cos(y), x+y, 2) + f(x\cdot y, \sin(x), \cos(y)) + f(\sin^2(x), x-y^2, y-x^2)$$

где

$$f(u,v,t) = \begin{cases} v\cdot u\cdot t, & \text{если} \quad u > 1 \\ u+t+v, & \text{если} \quad 0 \le u \le 1 \\ v-t-u, & \text{если} \quad 0 < u \end{cases}$$

7. Вычислить площадь треугольника, если известны его стороны.
8. Даны действительные числа s,t. Получить g(1.2,s) + g(t,s) - g(2s-1,st), где $g(a,b) = \dfrac{a^2+b^2-4\cdot a\cdot b}{a^2+5\cdot a\cdot b+3\cdot b^2+4\cdot a-b}$
9. Составить программу вычисления суммы квадратов простых чисел, лежащих в интервале [M, N].
10. Даны отрезки a, в, c и d. Для каждой тройки этих отрезков, из которых можно построить треугольник, напечатать площадь данного треугольника. (Определить функцию, вычисляющую площадь треугольника, если она существует)
11. Определить функцию нахождения расстояния между точками. Во множестве точек на плоскости найти пару точек с минимальным расстоянием между ними.
12. Найти: y=min(a,b,c)/max(a,b,c).
13. Вычислить сумму значений функций

$$Z = f(\sin(x)+\cos(y), x+y) + f(\sin(x), \cos(y)) + f(\sin^2(x)-2, a+b^2)$$

Где

$$f(u,t) = \begin{cases} u+t, & \text{если} \quad u > 1 \\ u-t, & \text{если} \quad 0 \le u \le 1 \\ t-u, & \text{если} \quad 0 < u \end{cases}$$

14. Даны значения a и b, найти их среднее арифметическое, среднегеометрическое.
15. Найти: y=max(a,b,c)+min(a,b,c).
16. Вычислить с использованием подпрограммы – функции Z=НОК(a+b,a*b)+НОК(a, b), где a, b – целые положительные числа, НОД – наибольший общий делитель, НОК – наименьшее общее кратное.
17. Вычислить среднее геометрическое шести вводимых чисел.
18. Вычислить сумму значений функции Z=F(a,b)+F(a2,b2)+F(a2-1,b)+F(a-b,b)

$$F(u,t) = \begin{cases} u^2+t^2, & \text{если} \quad u > 0, \quad t > 0 \\ u+t^2, & \text{если} \quad u \le 0, \quad t \le 0 \\ u-t, & \text{если} \quad u > 0, \quad t \le 0 \\ u+t, & \text{если} \quad u \le 0, \quad t > 0 \end{cases}$$

19. Даны действительные числа s, t. Получить f(t, -2s, 1.17) + f(2.2,t,s-t), где $f(a,b,c)=\frac{2\cdot a-b-\sin(c)+a\cdot b}{1+|c+a|}$
20. Найти: y=max(a,b,c,d)*min(a,b,c,d).
21. Вычислить с использованием подпрограммы $Z=\frac{\sum_{i=1}^{40}\sin(x_i)+\sum_{i=1}^{40}\cos(y_i)}{\sum_{i=1}^{40}|x_i|}$. Каждую сумму вычислять с использованием одной подпрограммы.
22. Задано множество точек на плоскости. Найти сумму длин отрезков между ними.
23. Вычислить с использованием функции Z=НОД(a,b)+НОД(a*b,a+b), где a, b – целые положительные числа, НОД – наибольший общий делитель.
24. Вычислить сумму значений функции

$$Z=f(\sqrt{|x|},y)+f(a,b)+f(\sqrt{|x|}+1,-y)+f((|x|-|y|),x),$$

$$\text{где}\quad f(u,t)=\begin{cases} u+2t, & \text{если} \quad u\ge 0 \\ u+t, & \text{если} \quad u\le -1 \\ u^2-2t+1, & \text{если} \quad -1<u<0 \end{cases}$$

25. Вычислить среднее арифметическое четырех вводимых чисел.

5.2 Передача массивов в функцию

Варианты заданий выбирать согласно номера в списке группы.

Определить функции, выполняющие действия в соответствии с вариантом задания.

1. Дан одномерный массив, состоящий из N целочисленных элементов.
   1.1. Найти максимальный положительный элемент.
   1.2. Вычислить сумму элементов массива.
2. Дан одномерный массив, состоящий из N вещественных элементов.
   2.1. Найти максимальный элемент.
   2.2. Вычислить среднеарифметическое отрицательных элементов массива.
3. Дан одномерный массив, состоящий из N вещественных элементов.
   3.1. Найти минимальный элемент.
   3.2. Вычислить произведение не нулевых элементов массива.
4. Дан одномерный массив, состоящий из N целочисленных элементов.
   4.1. Найти минимальный положительный элемент.

4.2. Вычислить сумму положительных элементов массива, кратных 3.

5. Дан одномерный массив, состоящий из N целочисленных элементов.
   5.1. Найти максимальный положительный элемент.
   5.2. Вычислить произведение элементов массива.
6. Дан одномерный массив, состоящий из N целочисленных элементов.
   6.1. Найти максимальный элемент.
   6.2. Вычислить сумму четных элементов массива.
7. Дан одномерный массив, состоящий из N вещественных элементов.
   7.1. Найти минимальный отрицательный элемент.
   7.2. Вычислить среднеарифметическое положительных элементов массива.
8. Дан одномерный массив, состоящий из N целочисленных элементов.
   8.1. Найти максимальный элемент.
   8.2. Вычислить среднеарифметическое элементов массива.
9. Дан одномерный массив, состоящий из N целочисленных элементов.
   9.1. Найти минимальный элемент.
   9.2. Вычислить сумму элементов массива.
10. Дан одномерный массив, состоящий из N целочисленных элементов.
   10.1. Найти максимальный отрицательный элемент.
   10.2. Вычислить произведение отрицательных элементов массива.
11. Дан одномерный массив, состоящий из N целочисленных элементов.
   11.1. Найти максимальный элемент.
   11.2. Вычислить среднеарифметическое нечетных элементов массива.
12. Дан одномерный массив, состоящий из N целочисленных элементов.
   12.1. Найти минимальный положительный элемент.
   12.2. Вычислить сумму четных элементов массива.
13. Дан одномерный массив, состоящий из N целочисленных элементов.
   13.1. Найти минимальный отрицательный элемент.
   13.2. Вычислить произведение ненулевых элементов массива, кратных 3.
14. Дан одномерный массив, состоящий из N целочисленных элементов.
   14.1. Найти максимальный отрицательный элемент.

14.2. Вычислить среднеарифметическое четных элементов массива.

15. Дан одномерный массив, состоящий из N вещественных элементов.

15.1. Найти максимальный элемент.

15.2. Вычислить среднеарифметическое положительных элементов массива.

16. Дан одномерный массив, состоящий из N вещественных элементов.

16.1. Найти минимальный положительный элемент.

16.2. Вычислить произведение не нулевых элементов массива.

17. Дан одномерный массив, состоящий из N целочисленных элементов.

17.1. Найти максимальный отрицательный элемент.

17.2. Вычислить сумму отрицательных элементов массива.

18. Дан одномерный массив, состоящий из N целочисленных элементов.

18.1. Найти минимальный элемент.

18.2. Вычислить сумму положительных нечетных элементов массива.

19. Дан одномерный массив, состоящий из N целочисленных элементов.

19.1. Найти минимальный положительный элемент.

19.2. Вычислить произведение нечетных элементов массива.

20. Дан одномерный массив, состоящий из N вещественных элементов.

20.1. Найти максимальный элемент.

20.2. Вычислить среднеарифметическое отрицательных элементов массива.

21. Дан одномерный массив, состоящий из N целочисленных элементов.

21.1. Найти максимальный положительный элемент.

21.2. Вычислить сумму положительных четных элементов массива.

22. Дан одномерный массив, состоящий из N целочисленных элементов.

22.1. Найти минимальный элемент.

22.2. Вычислить произведение ненулевых нечетных элементов массива.

23. Дан одномерный массив, состоящий из N вещественных элементов.

23.1. Найти минимальный положительный элемент.

23.2. Вычислить среднеарифметическое отрицательных элементов массива.

24. Дан одномерный массив, состоящий из N целочисленных элементов.
    24.1. Найти максимальный отрицательный элемент.
    24.2. Вычислить среднеарифметическое нечетных элементов массива.
25. Дан одномерный массив, состоящий из N целочисленных элементов.
    25.1. Найти минимальный отрицательный элемент.
    25.2. Вычислить сумму несчетных отрицательных элементов массива.

5.3 Передача массивов в функцию

Написать программу, выполняющую действия в соответствии с вариантом задания и передающую массив в функцию. Ввод и вывод массивов выполнить в отдельных функциях.

Вариант определяется по последней цифре в списке студентов группы.

1. Вычислить с использованием функции наименьшие элементы в строке и сумму номеров строк и столбцов, в которых они расположены, для матрицы A(10,15). Результаты формировать в одномерных массивах M(10) и S(10).
2. Дан массив a(8,5). С использованием функции найти среднеквадратичное значение положительных элементов каждой строки массива и сформировать из них одномерный массив b(8).
3. Вычислить с использованием функции max элементы каждой строки матрицы A(10,20). Результаты формировать в одномерных массивах C(10) и D(10).
4. Даны массивы a(3,4), b(2,5). Найти Z = (Ma+Mb)/(da+db), где Ma ,Mb - среднеарифметические значения массивов A, B. da, db - максимальные отклонения от среднеарифметических значений.
5. Дана матрица A(5,5). Сформировать одномерный массив C(5) из среднегеометрических значений положительных элементов каждого столбца матрицы.
6. Вычислить $Z = \frac{x_{\max} - y_{\min}}{x_{\min} - y_{\max}}$ с использованием функции, где xmax ,xmin ,ymax ,ymin – максимальные и минимальные элементы соответственно массива x(5,2) и массива y(3,4).
7. Дана матрица A(4,5), B(5,6). Вычислить $Z = p_a + p_b, \quad где \quad p = \sum_{i=1}^{N} \max\{x_{ij}\}_{j=1,N}$ сумма максимальных элементов каждой строки матрицы.

8. Вычислить с использованием функции min элементы каждой строки матрицы A(10,20). Результаты формировать в одномерных массивах C(10) и D(10).
9. Преобразовать массив x(3,3) в y, оставив в нем только положительные элементы. Вместо остальных элементов записать 0.
10. Определить количество положительных, отрицательных и нулевых элементов матрицы A(10,15). (Создать три функции для нахождения этих значений).

5.4 Использование рекурсии

Написать программу, рекурсивно вычисляющую сумму.

Вариант определяется по последней цифре в списке студентов группы.

1. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{n!}{5^n}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
2. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{1}{2^n} + \frac{1}{3^n}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
3. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{(-1)^{n-1}}{n^n}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
4. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{1}{(3n-2)(3n+1)}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
5. Найти сумму ряда с точностью ε, общий член которого равен $a_n = n^2 e^{-\sqrt{n}}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
6. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{n!}{(2n)!}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
7. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{3 \cdot n!}{(2n)!}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
8. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{\ln(n!)}{n^2}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.

9. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{(2n-1)}{2^n}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.
10. Найти сумму ряда с точностью ε, общий член которого равен $a_n = \frac{10^n}{(2n)!}$. Точность считается достигнутой, если следующий член последовательности меньше заданного ε.

ТЕМА 15. ОСНОВЫ ООП В PYTHON.

Лабораторная работа № 19

1. Цель и порядок работы

Цель работы – изучить возможности работы с классами в Питоне.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по языку программирования Питон, выполняя указанные в тексте примеры.

Выполнить индивидуальное задание.

2. Краткая теория

2.1 Классы и объекты

Класс является шаблоном или формальным описанием объекта, а объект представляет экземпляр этого класса, его реальное воплощение. С точки зрения кода класс объединяет набор функций и переменных, которые выполняют определенную задачу. Функции класса называют методами. Они определяют поведение класса. А переменные класса называют атрибутами- они хранят состояние класса.

Класс определяется с помощью ключевого слова *class*:

```
class название_класса:
    методы_класса
```

Для создания объекта класса используется следующий синтаксис:

```
название_объекта = название_класса([параметры])
```

Пример, определим простейший класс *Person*:

```
class Person:
    name = "Tom"

    def display_info(self):
        print("Привет, меня зовут", self.name)

person1 = Person()
person1.display_info()        # Привет, меня зовут Tom

person2 = Person()
person2.name = "Sam"
person2.display_info()        # Привет, меня зовут Sam
```

Класс *Person* определяет атрибут *name*, который хранит имя человека, и метод *display_info*, с помощью которого выводится информация о человеке.

При определении методов любого класса следует учитывать, что все

они должны принимать в качестве первого параметра ссылку на текущий объект, который называется *self*. Через эту ссылку внутри класса можем обратиться к методам или атрибутам этого же класса. В частности, через выражение *self.name* можно получить имя пользователя.

После определения класс *Person* создаем пару его объектов - *person1* и *person2*. Используя имя объекта, мы можем обратиться к его методам и атрибутам. В данном случае у каждого из объектов вызываем метод *display_info()*, который выводит строку на консоль.

Конструкторы

Для создания объекта класса используется конструктор. В предыдущем примере в классе *Person*, использовался конструктор по умолчанию, который неявно имеют все классы:

```
person1 = Person()
person2 = Person()
```

Конструктор в классах можно определить с помощью специального метода, который называется *__init()*. Пример, изменим класс *Person*, добавив в него конструктор:

```
class Person:

    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def display_info(self):
        print("Привет, меня зовут", self.name)

person1 = Person("Tom")
person1.display_info()      # Привет, меня зовут Tom
person2 = Person("Sam")
person2.display_info()      # Привет, меня зовут Sam
```

В качестве первого параметра конструктор принимает ссылку на текущий объект - *self*. Нередко в конструкторах устанавливаются атрибуты класса. Так, в данном случае в качестве второго параметра в конструктор передается имя пользователя, которое устанавливается для атрибута *self.name*. Для атрибута необязательно определять в классе переменную *name*, как это было в предыдущей версии класса *Person*. Установка значения *self.name = name* уже неявно создает атрибут *name*.

Деструктор

После окончания работы с объектом можем использовать оператор *del* для удаления его из памяти:

```
person1 = Person("Tom")
del person1     # удаление из памяти
```

Удалять объект необязательно, так как после окончания работы скрипта все объекты автоматически удаляются из памяти.

Для определения деструктора необходимо реализовать встроенную функцию *__del__*, которая будет вызываться либо в результате вызова оператора *del*, либо при автоматическом удалении объекта. Например:

```
class Person:
    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def __del__(self):
        print(self.name,"удален из памяти")
    def display_info(self):
        print("Привет, меня зовут", self.name)

person1 = Person("Tom")
person1.display_info()  # Привет, меня зовут Tom
del person1     # удаление из памяти
person2 = Person("Sam")
person2.display_info()  # Привет, меня зовут Sam
```

Определение классов в модулях и подключение

Как правило, классы размещаются в отдельных модулях и затем уже импортируются в основной скрипт программы. Допустим в проекте два файла: файл *main.py* (основной скрипт программы) и *classes.py* (скрипт с определением классов).

В файле *classes.py* определим два класса:

```
class Person:

    # конструктор
    def __init__(self, name):
        self.name = name  # устанавливаем имя

    def display_info(self):
        print("Привет, меня зовут", self.name)


class Auto:
    def __init__(self, name):
        self.name = name

    def move(self, speed):
```

```
        print(self.name, "едет со скоростью", speed, "км/ч")
```

Подключим эти классы и используем их в скрипте main.py:

```
from classes import Person, Auto

tom = Person("Tom")
tom.display_info()

bmw = Auto("BMW")
bmw.move(65)
```

Подключение классов происходит точно также, как и функций из модуля. Мы можем подключить весь модуль выражением:

```
import classes
```

Либо подключить отдельные классы, как в примере выше.

2.2 Инкапсуляция

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его. Например:

```
class Person:
    def __init__(self, name, age):
        self.name = name    # устанавливаем имя
        self.age = age      # устанавливаем возраст

    def display_info(self):
        print("Имя:", self.name, "\tВозраст:", self.age)


tom = Person("Tom", 23)
tom.name = "Человек-паук"       # изменяем атрибут name
tom.age = -129                  # изменяем атрибут age
tom.display_info()              # Имя: Человек-паук     Возраст: -129
```

В данном случае можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта.

С данной проблемой тесно связано понятие инкапсуляции. Инкапсуляция является фундаментальной концепцией объектно-ориентированного программирования. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода.

Касательно инкапсуляции непосредственно в языке программирования *Python* скрыть атрибуты класса можно сделав их приватными или закрытыми

и ограничив доступ к ним через специальные методы, которые еще называются свойствами.

Изменим выше определенный класс, определив в нем свойства:

```
class Person:
    def __init__(self, name, age):
        self.__name = name    # устанавливаем имя
        self.__age = age      # устанавливаем возраст

    def set_age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)

tom = Person("Tom", 23)

tom.__age = 43          # Атрибут age не изменится
tom.display_info()      # Имя: Tom  Возраст: 23
tom.set_age(-3486)       # Недопустимый возраст
tom.set_age(25)
tom.display_info()      # Имя: Tom  Возраст: 25
```

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: *self.__name*. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст:

```
tom.__age = 43
```

А попытка получить его значение приведет к ошибке выполнения:

```
print(tom.__age)
```

Однако все же может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута:

```
def get_age(self):
    return self.__age
```

Данный метод называют геттер или аксессор.

Для изменения возраста определено другое свойство:

```
def set_age(self, value):
    if value in range(1, 100):
        self.__age = value
    else:
        print("Недопустимый возраст")
```

Здесь уже можем решить в зависимости от условий, надо ли переустанавливать возраст. Данный метод называют сеттер или мьютейтор (*mutator*).

Необязательно создавать для каждого приватного атрибута подобную пару свойств. Так, в примере выше имя человека мы можем установить только из конструктора. А для получение определен метод *get_name*.

<u>Аннотации свойств</u>

Python имеет также еще один - способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация *@property.*

Для создания свойства-сеттера над свойством устанавливается аннотация *имя_свойства_геттера.setter*.

Перепишем класс Person с использованием аннотаций:

```
class Person:
    def __init__(self, name, age):
        self.__name = name    # устанавливаем имя
        self.__age = age      # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
    def name(self):
        return self.__name

    def display_info(self):
```

```
        print("Имя:", self.__name, "\tВозраст:", self.__age)

tom = Person("Tom", 23)
tom.display_info()      # Имя: Tom  Возраст: 23
tom.age = -3486         # Недопустимый возраст
print(tom.age)          # 23
tom.age = 36
tom.display_info()      # Имя: Tom  Возраст: 36
```

Свойство-сеттер определяется после свойства-геттера. Сеттер, и геттер называются одинаково - *age*. И поскольку геттер называется *age*, то над сеттером устанавливается аннотация *@age.setter*. После этого, что к геттеру, что к сеттеру, можно обратится через выражение *tom.age*.

2.3 Наследование

Наследование позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного программирования.

Ключевыми понятиями наследования являются подкласс и суперкласс. Подкласс наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (*base class*) или родительским (*parent class*), а подкласс - производным (*derived class*) или дочерним (*child class*).

Синтаксис для наследования классов выглядит следующим образом:

```
class подкласс (суперкласс):
    методы_подкласса
```

Расмотрим пример наследования на основе классов *Person*, который представляет человека, и класс *Employee* - класс работника.

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
```

```
    def name(self):
        return self.__name

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)

class Employee(Person):

    def details(self, company):
        # print(self.__name, "работает в компании", company)
        # так нельзя, self.__name - приватный атрибут
        print(self.name, "работает в компании", company)

tom = Employee("Tom", 23)
tom.details("Google")
tom.age = 33
tom.display_info()
```

Класс *Employee* полностью перенимает функционал класса *Person* и в дополнении к нему добавляет метод *details()*.

Стоит обратить внимание, что для *Employee* доступны через ключевое слово *self* все методы и атрибуты класса *Person*, кроме закрытых атрибутов типа *__name* или *__age*.

При создании объекта *Employee* фактически используем конструктор класса *Person*. И кроме того, у этого объекта мы можем вызвать все методы класса *Person*.

2.4 Полиморфизм

Полиморфизм является еще одним базовым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса.

Например:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age
```

```
    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)

class Employee(Person):
    # определение конструктора
    def __init__(self, name, age, company):
        Person.__init__(self, name, age)
        self.company = company

    # переопределение метода display_info
    def display_info(self):
        Person.display_info(self)
        print("Компания:", self.company)

class Student(Person):
    # определение конструктора
    def __init__(self, name, age, university):
        Person.__init__(self, name, age)
        self.university = university

    # переопределение метода display_info
    def display_info(self):
        print("Студент", self.name, "учится в университете", self.university)

people = [Person("Tom", 23), Student("Bob", 19, "Harvard"), Employee("Sam", 35, "Google")]

for person in people:
    person.display_info()
    print()
```

В производном классе *Employee*, который представляет служащего, определяется свой конструктор. Так как нам надо устанавливать при создании объекта еще и компанию, где работает сотрудник. Для этого конструктор принимает четыре параметра: стандартный параметр *self*, параметры *name* и *age* и параметр *company*.

В самом конструкторе *Employee* вызывается конструктор базового

класса *Person*. Обращение к методам базового класса имеет следующий синтаксис:

суперкласс.название_метода(self [, параметры])

Поэтому в конструктор базового класса передаются имя и возраст. Сам же класс *Employee* добавляет к функционалу класса *Person* еще один атрибут - *self.company*.

Кроме того, класс *Employee* переопределяет метод *display_info()* класса *Person*, поскольку кроме имени и возраста необходимо выводить еще и компанию, в которой работает служащий. И чтобы повторно не писать код вывода имени и возраста здесь также происходит обращение к методу базового класса - методу *get_info: Person.display_info(self)*.

Похожим образом определен класс *Student*, представляющий студента. Он также переопределяет конструктор и метод *display_info* за тем исключением, что вместо в методе *display_info* не вызывается версия этого метода из базового класса.

В основной части программы создается список из трех объектов *Person*, в котором два объекта также представляют классы *Employee* и *Student*. И в цикле этот список перебирается, и для каждого объекта в списке вызывается метод *display_info*. На этапе выполнения программы *Python* учитывает иерархию наследования и выбирает нужную версию метода *display_info()* для каждого объекта.

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции isinstance() мы можем проверить тип объекта. Эта функция принимает два параметра:

isinstance(object, type)

Первый параметр представляет объект, а второй - тип, на принадлежность к которому выполняется проверка. Если объект представляет указанный тип, то функция возвращает *True*. Например, возьмем выше описанную иерархию классов:

```
for person in people:
    if isinstance(person, Student):
        print(person.university)
    elif isinstance(person, Employee):
        print(person.company)
    else:
        print(person.name)
    print()
```

2.5 Класс object. Строковое представление объекта

Начиная с 3-й версии *Python* все классы неявно имеют один общий суперкласс - *object* и все классы по умолчанию наследуют его методы.

Одним из наиболее используемых методов класса *object* является метод

__str__(). Когда необходимо получить строковое представление объекта или вывести объект в виде строки, то *Python* как раз вызывает этот метод. И при определении класса хорошей практикой считается переопределение этого метода.

К примеру, возьмем класс *Person* и выведем его строковое представление:

```
class Person:
    def __init__(self, name, age):
        self.__name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def name(self):
        return self.__name

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def display_info(self):
        print("Имя:", self.__name, "\tВозраст:", self.__age)

tom = Person("Tom", 23)
print(tom)
```

При запуске программа выведет что-то наподобие следующего:

```
<__main__.Person object at 0x0000017D2BEBDCF8>
```

Это не очень информативная информация об объекте. Теперь определим в классе *Person* метод *__str__*:

```
class Person:
    def __init__(self, name, age):
        self._name = name  # устанавливаем имя
        self.__age = age  # устанавливаем возраст

    @property
    def name(self):
        return self.__name
```

```
    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    def display_info(self):
        print(self.__str__())

    def __str__(self):
        return "Имя: {} \t Возраст: {}".format(self.__name, self.__age)

tom = Person("Tom", 23)
print(tom)
```

Метод *__str__()* должен возвращать строку. И в данном случае мы возвращаем базовую информацию о человеке. И теперь консольный вывод будет другим:

3 Контрольные вопросы

Понятие класса в языке Питон.

Поля и методы класса.

Конструкторы и деструкторы в Питоне.

Свойства и аннотации.

Что понимается под термином «наследование»? Приведите примеры на язык Питон.

Что понимается под термином «полиморфизм»? Приведите примеры на язык Питон.

Что понимается под термином «инкапсуляция»? Приведите примеры на язык Питон.

4 Задание

Выбрать задание согласно варианта.

Порядок выполнения работы:

- разработать поля, методы и свойства для каждого из определяемых классов;
- все поля классов должны быть приватными;
- реализовать для каждого класса конструктор и деструктор;

- свойства по изменению и отображению значения полей;
- метод поиска информации из списка данных объектов по определенным критериям;
- переопределенный метод *__str__()* для вывода информации об объекте;
- реализовать меню по работе со списком объектов которое должно включать: добавление, редактирование, удаление объекта; отображение данных об объекте; поиска информации по определенным критериям.

Реализовать программу на *Python* в соответствии с вариантом исполнения. Описание классов должно быть в отдельном модуле. Варианты заданий определяются согласно списка студентов в группе.

5 Варианты заданий

Построить иерархию классов в соответствии с вариантом задания:

1. Студент, преподаватель, персона, заведующий кафедрой.
2. Служащий, персона, рабочий, инженер.
3. Рабочий, кадры, инженер, администрация.
4. Деталь, механизм, изделие, узел.
5. Организация, страховая компания, нефтегазовая компания, завод.
6. Журнал, книга, печатное издание, учебник.
7. Тест, экзамен, выпускной экзамен, испытание.
8. Место, область, город, мегаполис.
9. Игрушка, продукт, товар, молочный продукт.
10. Квитанция, накладная, документ, счет.
11. Автомобиль, поезд, транспортное средство, экспресс.
12. Двигатель, двигатель внутреннего сгорания, дизель, реактивный двигатель.
13. Республика, монархия, королевство, государство.
14. Млекопитающее, парнокопытное, птица, животное.
15. Корабль, пароход, парусник, корвет.
16. Самолет, автомобиль, корабль, транспортное средство.
17. Точка, линия, фигура плоская, фигура объемная.
18. Картина, рисунок, репродукция, пейзаж.
19. Статья, раздел, журнал, издательство.
20. Квартира, дом, улица, населенный пункт.

ТЕМА 16. ТЕХНОЛОГИИ ДОСТУПА К ДАННЫМ.

Лабораторная работа № 20

Цель и порядок работы

Цель работы – изучить технологии доступа к данным.

Порядок выполнения работы:

Самостоятельно проработать теоретические основы по технологиям доступа к данным.

Выполнить индивидуальное задание.

Задание (базовый уровень)

1. Проектирование базы данных:
   - описать таблицы на уровне 1 и 2 нормальной формы, все поля таблиц должны соответствовать правилу написания идентификаторов;
   - построить логическую и физическую модель базы данных;
   - описать связи между таблицами и характеристики связи;
   - письменный отчет предоставить преподавателю.
2. Разработка 1-й версии клиентского приложения MDI Win32:
   - в клиентской части должен быть создан объект типа DataSet содержащий физическую модель базы данных;
   - в клиенте предусмотреть возможность отображения, добавления, редактирования и удаления данных из таблиц на форме;
   - в качестве первичных ключей необходимо использовать тип int с возможностью автоматического увеличения при добавлении данных;
   - . информация о схеме данных и данных в таблицах должна сохраняться в XML файл.
3. Разработка 2-й версии клиентского приложения MDI Win32:
   - в качестве СУБД необходимо использовать базу данных Access (можно использовать любую другую локальную версию СУБД);
   - в качестве первичных ключей использовать тип Счетчик (для других СУБД предусмотреть аналогичный тип данных);
   - доступ к СУБД осуществлять с использованием технологии ADO.NET;
   - предусмотреть возможность добавления, редактирования, удаления и отображения данных из таблиц на форме с возможностью быстрого поиска данных в таблицах по определенному полю.
4. . Разработка 3-й версии клиентского приложения MDI Win32:

- в качестве СУБД использовать базу данных MS SQL Server 2008 и выше;
- в качестве первичных ключей для справочных таблиц необходимо использовать тип int с возможностью автоматического увеличения при добавлении данных, для основных таблиц использовать типа GUID;
- все запросы, связанные с изменением данных в базе данных, поиска информации и выбора информации для отчетов должны осуществляться на основе хранимых процедур;
- для отображения данных таблиц на форме использовать просмотры;
- доступ к СУБД осуществлять с использованием технологии ADO.NET;
- выборка данных для формирования отчетов должно осуществляться как минимум с использованием 2 таблиц;
- один отчет должен сохраняться в формате MS Word, второй в формате MS Excel;
- запрос данных и формирование отчета производить в отдельном потоке.

Задание (повышенный уровень)

1. Проектирование базы данных:
   - описать таблицы на уровне 1 и 2 нормальной формы, все поля таблиц должны соответствовать правилу написания идентификаторов;
   - построить логическую и физическую модель базы данных;
   - описать связи между таблицами и характеристики связи;
   - письменный отчет предоставить преподавателю.
2. Разработка 1-й версии клиентского приложения MDI WPF:
   - в клиентской части возможно использование объекта типа DataSet содержащего физическую модель базы данных;
   - в клиенте предусмотреть возможность отображения, добавления, редактирования и удаления данных из таблиц на форме;
   - в качестве первичных ключей необходимо использовать тип int с возможностью автоматического увеличения при добавлении данных;
   - . информация о схеме данных и данных в таблицах должна сохраняться в фале типа XML или Json.
3. Разработка 2-й версии клиентского приложения MDI WPF:
   - в качестве СУБД необходимо использовать любую локальную СУБД;

- в качестве первичных ключей необходимо использовать тип int с возможностью автоматического увеличения при добавлении данных.
- доступ к СУБД осуществлять либо с использованием технологии ADO.NET или на основе другой технологии подключения к базе данных;
- предусмотреть возможность добавления, редактирования, удаления и отображения данных из таблиц на форме с возможностью быстрого поиска данных в таблицах по определенному полю.

4. Разработка 3-й версии клиентского приложения MDI WPF:
   - в качестве СУБД использовать любую сетевую СУБД;
   - в качестве первичных ключей для справочных таблиц необходимо использовать тип int с возможностью автоматического увеличения при добавлении данных, для основных таблиц использовать типа GUID;
   - доступ к СУБД осуществлять с использованием технологий Entity Framework Core или же ей подобной;
   - выборка данных для формирования отчетов должно осуществляться как минимум с использованием 3 таблиц;
   - один отчет должен сохраняться в формате MS Word, второй в формате MS Excel;
   - запрос данных и формирование отчета производить в отдельном потоке.

Вариант задания определяется по номеру студента в списке группы.

Вариант 1

«Учёт работ строительной компании»

Информация хранится в базе данных следующей структуры:

- подразделение (код подразделения, название подразделения);
- работник (код работника, ФИО работника, дата рождения, ИНН, № пенсионного страхового свидетельства, код подразделения, паспортные данные);
- справочник работ (код работы, название работы);
- заказчик (код заказчика, наименование, телефон, адрес, ИНН);
- заказ (код заказа, код заказчика, название объекта, содержание работ, дата начала работы, дата окончания работы);
- работа (код заказа, код работы, код работника, дата начала работы, дата окончания работы, описание работы).

Вариант 2

«Учёт материалов на складе»
Информация хранится в базе данных следующей структуры:

- материал (код материала, название материала, код категории);
- категория (код категории, название категории, единицы измерения);
- поступление материала (код поступления, код материала, количество, код накладной);
- накладная (код накладной, дата оформления);
- расход материала (код расхода, код материала, количество, код накладной).

Вариант 3
«Учёт студентов, проживающих в общежитиях»
Информация хранится в базе данных следующей структуры:

- группа (код группы, название группы, название факультета);
- общежитие (код общежития, название, адрес);
- комната (код комнаты, № комнаты, код общежития);
- студент (код студента, ФИО студента, дата рождения, пол, код группы, серия паспорта, номер паспорта, кем и когда выдан);
- информация о заселении (код комнаты, код студента, дата заселения, дата выселения).

Вариант 4
«Учёт работы поликлиники»
Информация хранится в базе данных следующей структуры:

- отделение (код отделения, название отделения);
- должность (код должности, название должности);
- врач (код врача, ФИО врача, дата рождения, код отделения, код должности, пол);
- пациент (код пациента, ФИО пациента, дата рождения, пол, № медицинского полиса, паспортные данные, пол);
- приём у врача (код приёма, код пациента, код врача, дата приёма, время приёма, отчёт о приёме).

Вариант 5
«Учёт отпусков сотрудников»
Информация хранится в базе данных следующей структуры:

- отдел (код отдела, название отдела);
- должность (код должности, название должности);
- сотрудник (табельный номер, ФИО сотрудника, дата рождения, ИНН, № пенсионного страхового свидетельства, паспортные данные);
- вид отпуска (код вида отпуска, вид отпуска);

- рабочее место (код работы, код сотрудника, код должности, код отдела, дата начала работы, дата завершения работы);
- отпуск (код отпуска, код вида отпуска, код работы, дата начала отпуска, дата окончания отпуска).

Вариант 6
«Учёт арендуемых помещений»
Информация хранится в базе данных следующей структуры:

- здание (код здания, название, адрес);
- помещение (код помещения, название помещения, площадь, код здания);
- арендатор (код арендатора, название фирмы, юридический адрес, ФИО руководителя, контактный телефон);
- аренда (код аренды, код помещения, код арендатора, № договора, дата оформления договора, дата начала аренды, дата окончания аренды).

Вариант 7
«Учёт работы автомастерской»
Информация хранится в базе данных следующей структуры:

- вид работ (код вида работ, вид работ);
- модели машин (код модели, название модели);
- автовладелец (код автовладельца, ФИО автовладельца, серия паспорта, номер паспорта, кем и когда выдан);
- автомобиль (код автомобиля, код модели, код автовладельца, номер автомобиля);
- мастер (код мастера, ФИО мастера, паспортные данные, дата рождения);
- работа (код автомобиля, код вида работ, дата начала работ, код мастера, дата окончания работ, описание проделанной работы).

Вариант 8
«Учёт работы туристического агентства»
Информация хранится в базе данных следующей структуры:

- туроператор (код туроператора, название туроператора, ФИО контактного лица, контактный телефон, факс, адрес в WWW, e-mail);
- место назначения (код места назначения, название);
- курорт (код курорта, код места назначения, название курорта);
- клиент (код клиента, ФИО клиента, паспорт, контактный телефон);
- тур (код тура, код курорта, начало тура, окончание тура, код клиента, № договора, дата оплаты, стоимость).

Вариант 9

«Учёт библиотечного фонда»

Информация хранится в базе данных следующей структуры:

- книга (код книги, наименование, год издания, цена, ISBN, код типа книги);
- тип книги (код типа книги, тип книги);
- поступления (№ экземпляра, код книги, код типа поступления, дата поступления);
- тип поступления (код типа поступления, тип поступления);
- списание (код списания, № экземпляра, дата списания, № акта, причина списания);
- абонемент (код записи в абонементе, № экземпляра, ФИО читателя, дата получения, дата возвращения).

Вариант 10

«Учёт материальных ценностей»

Информация хранится в базе данных следующей структуры:

- материально-ответственное лицо (код сотрудника, ФИО сотрудника, паспортные данные, подразделение);
- тип ценности (код типа ценности, тип ценности);
- акт приёма (код акта приёма, № акта, дата оформления);
- акт списания (код акта списания, № акта, дата оформления);
- материальная ценность (код ценности, инвентарный номер, название, стоимость, код акта приёма, код акта списания, код типа ценности, код сотрудника).

Вариант 11

«Учёт выступлений студентов на научных конференциях»

Информация хранится в базе данных следующей структуры:

- группа (код группы, название группы);
- студент (код студента, ФИО студента, № зачётки, дата рождения, пол студента, код группы);
- научный руководитель (код сотрудника, ФИО сотрудника, кафедра, должность);
- конференция (код конференции, название, место проведения, дата начала, дата окончания);
- доклад (код доклада, название доклада, дата выступления, код конференции, код студента, код сотрудника).

Вариант 12

«Учёт работы научных конференций»

Информация хранится в базе данных следующей структуры:

- конференция (код конференции, название конференции, дата начала, дата окончания);
- руководители секций (код сотрудника, ФИО сотрудника, кафедра, должность);
- секция (код секции, название секции, дата начала работы, дата завершения работы, код конференции, код сотрудника);
- участник (код участника, ФИО участника, место и должность работы, e-mail, контактный телефон, адрес);
- доклад (код доклада, название доклада, дата выступления, код секции, код участника).

Вариант 13

«Учёт успеваемости студентов»

Информация хранится в базе данных следующей структуры:

- специальность (код специальности, название специальности);
- учебная группа (код группы, название группы, код специальности);
- студент (код студента, ФИО студента, № зачётки, дата рождения, код группы);
- преподаватель (код преподавателя, табельный №, ФИО преподавателя, кафедра, должность);
- дисциплина (код дисциплины, название дисциплины, количество лекционных часов, количество часов практических занятий, количество часов лабораторных занятий, семестр);
- успеваемость (код записи, код дисциплины, код преподавателя, код студента, форма контроля, оценка, дата сдачи).

Вариант 14

«Учёт посещаемости студентов»

Информация хранится в базе данных следующей структуры:

- студент (код студента, ФИО студента, № зачётки, дата рождения, группа);
- вид занятия (код вида занятия, название вида занятия);
- преподаватель (код преподавателя, табельный №, ФИО преподавателя, кафедра, должность);
- дисциплина (код дисциплины, название дисциплины, семестр);
- занятия (код занятия, код дисциплины, код преподавателя, код вида занятия, дата, № пары, тема занятия);
- посещаемость (код занятия, код студента).

Вариант 15

«Расписания движения поездов»

Информация хранится в базе данных следующей структуры:

- поезд (код поезда, № поезда, название, код станции отправления, код станции прибытия);
- станция (код станции, название станции);
- расписание поезда (код, код поезда, время в пути, время прибытия, время стоянки, время отправления, код станции);
- состав (код состава, код поезда, номер состава, дата отправления, ФИО начальника поезда).

Вариант 16
«Учет домашних животных»
Информация хранится в базе данных следующей структуры:

- домашнее животное (код животного, код породы, кличка, дата рождения, код владельца);
- владельцы (код владельца, ФИО, адрес проживания);
- порода (код породы, название породы);
- прививки (код прививки, название прививки, комментарий);
- прививки выставленные (код, код животного, код прививки, дата).

Вариант 17
«Учет банковских операций»
Информация хранится в базе данных следующей структуры:

- счет (код счета, счет, код владельца счета, дата открытия, код банка);
- владельцы (код владельца, ФИО, адрес проживания);
- валюта (код валюты, название валюты, кратность);
- банки (код банка, название банка);
- валютные операции (код, код счета с которого производиться перечисление, код счета на который производиться перечисление, сумма операции в валюте, код валюты, сумма операции в рублях).

Вариант 18
«Учет матчей»
Информация хранится в базе данных следующей структуры:

- матчи (код, код первой команды, код второй команды, дата проведения матча, код стадиона, код турнира);
- команды (код команды, название команды, адрес дислокации);
- стадион (код стадиона, название стадиона, адрес стадиона);
- турниры (код турнира, название турнира, код спонсора, дата начало проведения, дата конца проведения);
- спонсоры (код спонсора, код турнира, название спонсора).

Вариант 19

«Поликлиника»

Информация хранится в базе данных следующей структуры:

- прием (код приема, код больного, код назначения, дата приема, диагноз, дата окончания болезни);
- врачи (код врача, ФИО врача, код специализации);
- больные (код больного, ФИО больного, адрес проживания);
- специализации (код специализации, название специализации);
- назначения (код назначения, дата назначения, код лекарства);
- лекарства (код лекарства, код назначения, название лекарства).

Вариант 20

«Регистрация автомобилей»

Информация хранится в базе данных следующей структуры:

- регистрация (код регистрации, код владельца, код автомобиля, дата регистрации, номер автомобиля);
- владельцы (код владельца, ФИО, адрес проживания);
- автомобиль (код автомобиля, код марки автомобиля, номер кузова, номер двигателя);
- марка (код марки, название марки, код производителя);
- производитель (код производителя, название производителя, адрес).

Ссылки на источники:

1. Описание СУБД https://metanit.com/sql/
2. Руководство по ADO.NET и работе с базами данных https://metanit.com/sharp/adonet/
3. Руководство по Entity Framework Core https://metanit.com/sharp/entityframeworkcore/
4. Руководство по WPF https://metanit.com/sharp/wpf/

СПИСОК ЛИТЕРАТУРЫ

Основная литература:

1. Маляров, А. Н. Объектно-ориентированное программирование : учебник для технических вузов / А. Н. Маляров. — Самара : Самарский государственный технический университет, ЭБС АСВ, 2017. — 332 с. — ISBN 978-5-7964-1952-6. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/91772.html (дата обращения: 25.05.2020).

Дополнительная литература:

1. Осипов, Н. А. Разработка Windows приложений на C# / Н. А. Осипов. — Санкт-Петербург : Университет ИТМО, 2012. — 74 с. — ISBN 2227-8397. — Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. — URL: http://www.iprbookshop.ru/68071.html (дата обращения: 25.05.2020).
2. Биллиг, В. А. Основы программирования на C# : учебное пособие / В. А. Биллиг. — 2-е изд. — Москва : ИНТУИТ, 2016. — 574 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100319 (дата обращения: 25.05.2020).
3. Сузи, Р. А. Язык программирования Python : учебное пособие / Р. А. Сузи. — 2-е изд. — Москва : ИНТУИТ, 2016. — 350 с. — ISBN 5-9556-0058-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100546 (дата обращения: 01.05.2020). — Режим доступа: для авториз. пользователей.
4. Северенс, Ч. Введение в программирование на Python : учебное пособие / Ч. Северенс. — 2-е изд. — Москва : ИНТУИТ, 2016. — 231 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100703 (дата обращения: 01.05.2020). — Режим доступа: для авториз. пользователей.
5. Хахаев, И. А. Практикум по алгоритмизации и программированию на Python : учебное пособие / И. А. Хахаев. — 2-е изд. — Москва : ИНТУИТ, 2016. — 178 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100377 (дата обращения: 01.05.2020). — Режим доступа: для авториз. пользователей.