Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых
(ВлГУ)

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические указания
к лабораторным занятиям

Составители:
О.Н. Павлова
А.А. Касьянов

Владимир 2013

УДК 004.43
ББК 32.973.26-018.1
Я

Рецензент
Кандидат технических наук,
генеральный директор ООО "ФС Сервис"
Д.С. Квасов

Печатается по решению редакционного совета
Владимирского государственного университета

Я **Языки** программирования: метод. указания к лаб. занятиям / Сост. : О.Н. Павлова, А.А. Касьянов; Владим. гос. ун-т. – Владимир : Изд-во Владим. гос. ун-та, 2013. – 74 с.

Приведены лабораторные работы по дисциплине «Языки программирования».

Предназначены для студентов первого курса очной и заочной формы обучения по направлению подготовки бакалавров 010300 «Фундаментальные информатика и информационные технологии».

Рекомендованы для формирования профессиональных компетенций в соответствии с ФГОС 3-го поколения.

Ил. 28. Табл. 24. Библиогр.: 5 назв.

УДК 004.43
ББК 32.973.26-018.1

ВВЕДЕНИЕ

Изучение языков программирования является актуальной задачей для современного образовательного стандарта. Количество различных языков программирования неуклонно растёт. Поэтому возникает потребность в подготовке специалистов, владеющих наиболее востребованными в промышленности и производстве языками программирования. В связи с ростом объектно-ориентированных приложений и мобильных платформ необходимо сделать акцент на объектно-ориентированных языках программирования. Одним из наиболее распространённых языков программирования является .Net версия языка С++ – С#.

Исходя из вышеизложенного, С# является оптимальным при разработке различного рода графических и Windows-приложений. Кроме того, данный язык программирования позволяет оперировать с большими объемами данных.

Данный методический материал является практической частью дисциплины «Языки программирования».

В результате выполнения представленных лабораторных работ студент должен:

- Изучить основные структурные элементы языка программирования С#;
- Изучить логику построения программы;
- Освоить основы лексического, синтаксического и семантического анализа кода программы;
- Научиться создавать графические приложения с дружественным интерфейсом;
- Уметь применять полученные знания в профессиональной деятельности.

ЛАБОРАТОРНАЯ РАБОТА 1. ТИПЫ ДАННЫХ. ОПЕРАТОРЫ

Цель работы: *изучение системы базовых и составных типов данных языка C#*

Краткая теоретическая часть

Типы, переменные и значения

C# является строго типизированным языком. Каждая переменная и константа имеет тип, как и каждое выражение, результатом вычисления которого является значение. Каждая сигнатура метода задает тип для каждого входного параметра и для возвращаемого значения. Библиотека классов платформы .NET Framework определяет набор встроенных числовых типов, а также более сложных типов, представляющих широкое разнообразие логических конструкций, например, файловую систему, сетевые подключения, коллекции и массивы объектов и даты.

Компилятор использует сведения о типе, чтобы убедиться, что все операции, выполняемые в коде, являются *типобезопасными*. Например, при объявлении переменной типа ***int***, компилятор позволяет использовать в дополнение переменную и операции вычитания. При попытке выполнить эти же операции в переменной типа ***bool***, компилятор вызовет ошибку, как показано в следующем примере:

```
int a = 5;
int b = a + 2; //OK
bool test = true;
// Error. Operator '+' cannot be applied to operands
of type 'int' and 'bool'.
int c = a + test;
```

Компилятор внедряет сведения о типе в исполняемый файл.

Задание типов в объявлениях переменных

При объявлении переменной или константы в программе необходимо либо задать ее тип, либо использовать ключевое слово **var**,

чтобы дать возможность компилятору определить его. В следующем примере показаны некоторые объявления переменных, использующие встроенные числовые типы и сложные пользовательские типы:

```
// Объявление:
float temperature;
string name;
MyClass myClass;
// Объявление с инициализацией:
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
where item <= limit
select item;
```

После объявления переменной она не может быть повторно объявлена с новым типом, и ей нельзя присвоить значение, несовместимое с ее объявленным типом.

Приведение и преобразование типов

Поскольку в C# тип определяется статически тип во время компиляции, после объявления переменной, она не может быть объявлена вновь или использоваться для хранения значений другого типа, если этот тип не преобразуется в тип переменной. Например, невозможно преобразование из целого числа в произвольную строку. Поэтому после объявления переменной «i» как целочисленной, нельзя ей присвоить строку "Hello", как показано в следующем коде.

```
int i;
i = "Hello"; // Ошибка: "Cannot implicitly convert
type 'string' to 'int'"
```

Но иногда может быть необходимым скопировать значение в переменную или параметр метода другого типа. Например, может быть переменная, которую требуется передать методу, параметр которого имеет тип **double**. Операции такого вида называются *преобразованиями типов*. В C# можно выполнять следующие виды преобразований:

- **Неявные преобразования**. Не требуется никакого специального синтаксиса, поскольку преобразование безопасно для типов и данные не теряются. Примерами могут служить преобразования от меньшего к большему целому типу, и преобразования из производных классов в базовые классы.
- **Явные преобразования (приведения)**. Для явных преобразований необходим оператор приведения. Приведение требуется, когда при преобразовании может быть потеряна информация, или когда преобразование может завершиться неудачей по другим причинам. К типичным примерам относится числовое преобразование в тип, который имеет меньшую точность или меньший диапазон значений, а также преобразование экземпляра базового класса в производный класс.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        a = (int)x; //Преобразование из double в int
        System.Console.WriteLine(a);
    }
}
```

- **Пользовательские преобразования.** Пользовательские преобразования выполняются специальными методами, которые можно определить для включения явных и неявных преобразований между пользовательскими типами, не имеющих отношения базовый класс — производный класс.Дополнительные сведения см. в разделе-Операторы преобразования (Руководство по программированию в C#).
- **Преобразования с помощью вспомогательных классов.** Для преобразования между несовместимыми типами, например целые числа и объекты **System.DateTime**, или шестнадцатеричные строки и байтовые массивы, можно использовать класс **System.BitConverter**,

класс **System.Convert** и методы **Parse** встроенных числовых типов, таких как **Int32.Parse**.

Таблица встроенных типов

В приведенной ниже таблице представлены ключевые слова для встроенных типов C#, которые являются псевдонимами предопределенных типов в пространстве имен **System**.

Таблица 1. Целые типы

Type	**Диапазон**	**Размер**
sbyte	От -128 до 127	8-разрядное знаковое целое число
byte	От 0 до 255	8-разрядное целое число без знака
char	от U+0000 до U+ffff	16-разрядный символ Юникода
short	От -32 768 до 32 767	16-разрядное знаковое целое число
ushort	От 0 до 65 535	16-разрядное целое число без знака
int	От -2 147 483 648 до 2 147 483 647	32-разрядное знаковое целое число
uint	От 0 до 4 294 967 295	32-разрядное целое число без знака
long	От -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	64-разрядное целое число со знаком
ulong	От 0 до 18 446 744 073 709 551 615	64-разрядное целое число без знака

Таблица 2. Таблица типов с плавающей запятой

Тип	**Приблизительный диапазон**	**Точность**
float	От ±1,5e−45 до ±3,4e38	7 знаков
double	От ±5,0e−324 до ±1,7e308	15-16 знаков
decimal	(от -7,9 x 1028 до 7,9 x 1028) / (100–28)	28-29 значимых цифр

Ключевое слово **decimal** обозначает 128-разрядный тип данных. По сравнению с типом данных с плавающей запятой, тип **decimal** имеет более точный и узкий диапазон, благодаря чему он походит для

финансовых расчетов. Если необходимо, чтобы числовой фактический литерал рассматривался как **decimal**, используйте суффикс m или M, например:

```
decimal myMoney = 300.5m;
```

Если суффикс m отсутствует, число рассматривается как double и возникает ошибка компилятора.

Логический тип данных

Ключевое слово **bool** используется для объявления переменных для хранения логических значений: **true** и **false,** является псевдонимом свойства **System.Boolean**.

```
bool b = true;
```

Символьный тип данных

Константы типа **char** могут быть записаны в виде символьных литералов, шестнадцатеричной escape-последовательности или представления Юникода. Кроме того, можно привести коды целых символов. В следующем примере показана инициализация четырех переменных **char** с одним и тем же символом X:

```
char[] chars = new char[4];
chars[0] = 'X';         // символьный литерал
chars[1] = '\x0058';    // шестнадцатеричная константа
chars[2] = (char)88;    // преобразование из целого
числа
chars[3] = '\u0058';    // представление Юникода
```

Массивы

Массив – это структура данных, содержащая несколько переменных одного типа. Массив объявляется со следующим типом:

```
type[] arrayName;
```

В следующем примере показано создание одномерных, многомерных массивов и массивов массивов.

```
class TestArraysClass
{
static void Main()
{
     // Объявление одномерного массива
```

```
        int[] array1 = newint[5];
        // Объявление и инициализация одномерного массива
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };
        // Альтернативный синтаксис
        int[] array3 = { 1, 2, 3, 4, 5, 6 };
        // Объявление двумерного массива
        int[,] multiDimensionalArray1 = newint[2, 3];
        // Объявление и инициализация двумерного массива
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, {
        4, 5, 6 } };
        // Объявление невыровненного массива
        int[][] jaggedArray = new int[6][];
        // Заполнение нулевой строки невыровненного мас-
        сива
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
    }
```

Массив имеет следующие свойства:

- Значение по умолчанию числовых элементов массива задано равным нулю, а элементы ссылок имеют значение **NULL**.
- Индексация массивов начинается с нуля: n-элементный массив индексируется от 0 до n–1.
- Элементы массива могут быть любых типов, включая тип массива.

Доступ к элементам массивов осуществляется по индексам, которые указываются в квадратных скобках.

Строки

Строка является объектом типа **String**, значением которого является текст. По сути, текст хранится в виде последовательности символов. В конце строки на языке C# отсутствует символ, заканчивающийся на NULL; поэтому строка C# может содержать любое число внедренных символов NULL ('\0'). Свойство **Length** строки представляет число объектов **Char**, содержащихся в этой строке, а не число символов Юникода. В C# **String** и **string** эквивалентны, и пользователи могут использовать любое наиболее предпочтительное для них соглашение по наименованию.

Объявление и инициализация строк

Объявление и инициализацию строк можно выполнять различными способами, как показано в следующем примере:

```
// Объявление без инициализации
string message1;
// Инициализация пустой строкой
string message2 = null;
string message3 = System.String.Empty;
//Инициализация константной строкой
string oldPath = "c:\\Program Files\\MS VStudio 8.0";
// Инициализация строкой без специальных символов
string newPath = @"c:\Program Files\ MS VStudio 9.0";
// Использование System.String
System.String greeting = "Hello World!";
// Создание локальной строки
var temp = "I'm still a strongly-typed Sys-
tem.String!";
// Создание константной строки
const string message4 = "You can't get rid of me!";
//Создание строки из массива символов
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Строковые объекты являются *неизменяемыми*: после создания их нельзя изменить. Все методы **String** и операторы C#, которые, как можно было бы представить, изменяют строку, в действительности возвращают результаты в новый строковый объект.

Таблица 3. Основные методы работы со строками

Метод	Описание
Empty	Возвращается пустая строка. Свойство со статусом read only
Compare	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать или не учитывать регистр, особенности национального форматирования дат, чисел и т.д.
CompareOrdinal	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
Concat	Конкатенация строк. Метод перегружен, допускает сцепление произвольного числа строк
Copy	Создается копия строки

Метод	Описание
Format	Выполняет форматирование в соответствии с заданными спецификациями формата. Ниже приведено более полное описание метода
Intern, IsIntern	Отыскивается и возвращается ссылка на строку, если таковая уже хранится во внутреннем пуле данных. Если же строки нет, то первый из методов добавляет строку во внутренний пул, второй - возвращает null. Методы применяются обычно тогда, когда строка создается с использованием построителя строк - класса StringBuilder
Join	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом Join, является обратной к операции, заданной методом Split. Последний является динамическим методом и, используя разделители, осуществляет разделение строки на элементы
Insert	Вставляет подстроку в заданную позицию
Remove	Удаляет подстроку в заданной позиции
Replace	Заменяет подстроку в заданной позиции на новую подстроку
Substring	Выделяет подстроку в заданной позиции
IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
StartsWith, EndsWith	Возвращается true или false, в зависимости от того, начинается или заканчивается строка заданной подстрокой
PadLeft, PadRight	Выполняет набивку нужным числом пробелов в начале и в конце строки
Trim,TrimStart, TrimEnd	Обратные операции к методам Pad. Удаляются пробелы в начале и в конце строки, или только с одного ее конца
ToCharArray	Преобразование строки в массив символов

Операторы языка C#

Состав операторов языка C#, их синтаксис и семантика унаследованы от языка C++. Как и положено, потомок частично дополнил состав, переопределил синтаксис и семантику отдельных операторов, постарался улучшить характеристики языка во благо программиста.

Оператор присваивания

Как в языке C++, так и в C# присваивание формально считается операцией. Вместе с тем запись:

```
X = expr;
```

следует считать настоящим оператором присваивания, так же, как и одновременное присваивание со списком переменных в левой части:

```
X1 = X2 = ... = Xk = expr;
```

Блок или составной оператор

С помощью фигурных скобок несколько операторов языка (возможно, перемежаемых объявлениями) можно объединить в единую синтаксическую конструкцию, называемую *блоком* или *составным оператором*:

```
{
    оператор_1
    ...
    оператор_N
}
```

Синтаксически блок воспринимается как единичный оператор и может использоваться всюду в конструкциях, где синтаксис требует одного оператора. Тело цикла, ветви оператора **if**, как правило, представляются блоком.

Пустой оператор

Пустой оператор – это "пусто", завершаемое точкой с запятой. Иногда полезно рассматривать отсутствие операторов как существующий пустой оператор. Синтаксически допустимо ставить лишние точки с запятой, полагая, что вставляются пустые операторы.

Операторы выбора

Начнем с синтаксиса условного оператора **if**:

```
if(выражение_1) оператор_1;
else if(выражение_2) оператор_2;
...
else if(выражение_K) оператор_K;
else оператор_N;
```

Выражения **if** должны заключаться в круглые скобки и быть булевого типа (значения **true** или **false**). Может быть опущена **else**-ветвь. Ветвь **else**, если она есть, относится к ближайшему открытому **if**.

Оператор switch

Оператор выбора из нескольких вариантов является **switch**.

```
switch(выражение)
{
```

```
    case константное_выражение_1: [операторы_1 опера-
тор_перехода_1]
    ...
    case константное_выражение_K: [операторы_K опера-
тор_перехода_K]
    [default: операторы_N оператор_перехода_N]
}
```

Константные выражения в case должны иметь тот же тип, что и switch-выражение.

Вначале вычисляется значение **switch**-выражения. Затем оно поочередно в порядке следования **case** сравнивается на совпадение с константными выражениями. Как только достигнуто совпадение, выполняется соответствующая последовательность операторов case-ветви. Поскольку последний оператор этой последовательности является оператором перехода (чаще всего это оператор **break**), то обычно он завершает выполнение оператора **switch**. Если значение switch-выражения не совпадает ни с одним константным выражением, то выполняется последовательность операторов ветви **default**, если же таковой ветви нет, то оператор **switch** эквивалентен пустому оператору.

Операторы break и continue

Оператор **break** может стоять в теле цикла или завершать case-ветвь в операторе **switch**. При выполнении оператора **break** в теле цикла завершается выполнение самого внутреннего цикла.

Оператор **continue** используется только в теле цикла. В отличие от оператора **break**, завершающего внутренний цикл, **continue** осуществляет переход к следующей итерации этого цикла.

Оператор return

Еще одним оператором, относящимся к группе операторов перехода, является оператор **return**, позволяющий завершить выполнение процедуры или функции. Его синтаксис:

```
return [выражение];
```

Для функций его присутствие и аргумент обязательны, поскольку выражение в операторе **return** задает значение, возвращаемое функцией.

Операторы цикла

Оператор for

Синтаксис:

```
for(инициализаторы; условие; список_выражений) опера-
тор
```

Оператор, стоящий после закрывающей скобки, задает тело цикла. Сколько раз будет выполняться тело цикла, зависит от трех управляющих элементов, заданных в скобках. Инициализаторы задают начальное значение одной или нескольких переменных, часто называемых счетчиками или просто переменными цикла. Условие задает условие окончания цикла, соответствующее выражение при вычислении должно получать значение **true** или **false**. Список выражений, записанный через запятую, показывает, как меняются счетчики цикла на каждом шаге выполнения. Если условие цикла истинно, то выполняется тело цикла, затем изменяются значения счетчиков и снова проверяется условие. Как только условие становится ложным, цикл завершает свою работу. В нормальной ситуации тело цикла выполняется конечное число раз.

Циклы While

Тело цикла выполняется до тех пор, пока остается истинным выражение **while**. В языке C# у этого вида цикла две модификации – с проверкой условия в начале и в конце цикла. Первая модификация имеет следующий синтаксис:

```
while(выражение) оператор;
```

Эта модификация соответствует стратегии: "сначала проверь, а потом делай". В результате проверки может оказаться, что и делать ничего не нужно. Тело такого цикла может ни разу не выполняться.

Цикл, проверяющий условие завершения в конце, соответствует стратегии: "сначала делай, а потом проверь". Тело такого цикла выполняется, по меньшей мере, один раз. Вот синтаксис этой модификации:

```
do
     оператор
while(выражение);
```

Цикл foreach

Данный вид цикла удобен при работе с массивами, коллекциями и другими подобными контейнерами данных. Его синтаксис:

```
foreach(тип идентификатор in контейнер) оператор
```

Цикл работает в полном соответствии со своим названием - тело цикла выполняется для каждого элемента в контейнере. Тип идентификатора должен быть согласован с типом элементов, хранящихся в контейнере данных. Предполагается также, что элементы контейнера (массива, коллекции) упорядочены. На каждом шаге цикла идентификатор, задающий текущий элемент контейнера, получает значение очередного элемента в соответствии с порядком, установленным на элементах контейнера. С этим текущим элементом и выполняется тело цикла – выполняется столько раз, сколько элементов находится в контейнере. Цикл заканчивается, когда полностью перебраны все элементы контейнера.

Серьезным недостатком циклов **foreach** в языке C# является то, что цикл работает только на чтение, но не на запись элементов. Так что наполнять контейнер элементами приходится с помощью других операторов цикла.

```
/// Вычисление суммы, максимального и минимального
/// элементов трехмерного массива, заполненного
/// случайными числами.
public void SumMinMax()
{
     int [,,] arr3d = new int[10,10,10];
     Random rnd = new Random();
     for (int i =0; i<10; i++)
          for (int j =0; j<10; j++)
               for (int k =0; k<10; k++)
                    arr3d[i,j,k]= rnd.Next(100);
     long sum =0; int min=arr3d[0,0,0], max=arr3d[0,0,0];
     foreach(int item in arr3d)
     {
          sum +=item;
          if (item > max) max = item;
          else if (item < min) min = item;
     }
```

```
     Console.WriteLine("sum = {0}, min = {1}, max =
{2}", sum, min, max);
}
```

Структура программы

```
using System;     // подключение библиотек
namespace YourNamespace//объявление пространства имен
{
class YourClass        // описание собственных клас-
сов
{}
struct YourStruct // описание собственных структур
{}
interface IYourInterface // описание интерфейсов
{}
delegate int YourDelegate();// описание делегатов
enum YourEnum          // описание перечислений
{}
namespace YourNestedNamespace
{
     struct YourStruct
      {}
}
class YourMainClass // описание основного класса
(обычно по-умолчанию название проекта)
{
     static void Main(string[] args) // главная запус-
каемая функция приложения
    {
        // перечень допустимых операторов
    }
}
}
```

Задания к работе

1. Дан одномерный числовой массив Т(к).Вычислить сумму произведений всех троек соседних чисел.

2. Дан массив B(n), содержащий большое количество нулевых элементов. Заменить все группы подряд встречающихся нулей на один нуль.

3. Дана последовательность целых чисел. Найти количество различных чисел в этой последовательности.

4. Дан массив целых чисел. Найти в этом массиве минимальный элемент m и максимальный элемент M. Получить в порядке возрастания все целые числа из интервала (m; M), которые не входят в данный массив.

5. В данном целочисленном массиве A(n) указать индексы всех элементов, имеющих наибольшее значение.

6. Дана квадратная целочисленная матрица F(m,m).Найти суммы элементов тех строк, имеющих четные элементы на главной диагонали.

7. Для заданной матрицы размером n на n найти такие k, что k-я строка матрицы совпадает с k-м столбцом.

8. Для заданной матрицы размером n на n найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

9. 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.

10. Найти номер первой из строк прямоугольно числовой матрицы, не содержащих ни одного положительного элемента.

11. Дана строка, содержащая текст, заканчивающийся точкой. Вывести на экран слова, содержащие три буквы.

12. Дана строка. Найти в ней те слова, которые начинаются и оканчиваются одной и той же буквой.

13. Даны две строки f1 и f2. Строка f1 содержит произвольный текст. Слова в тексте разделены пробелами и знаками препинания. Строка f2 содержит не более 30 слов, которые разделены запятыми. Эти слова образуют пары: каждое второе является синонимом первого. Заменить в строке f1 те слова, которые можно, их синонимами. Результат поместить в новую строку.

Распределение по вариантам

Задания выбираются по номеру в журнале подгруппы. Если номер больше, чем количество вариантов, то отсчет ведется с учетом нормировки.

Номер варианта	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Номера задач	1, 6, 11, 13	2, 7, 12, 13	3, 8, 11, 13	4, 9, 12, 13	5, 10, 11, 13	1,10, 12, 13	2, 9, 11, 13	3, 8, 12, 13	4, 7, 11, 13	5, 6, 12, 13

Контрольные вопросы

1. Назовите базовые типы данных в языке C#: функции и назначение, характеристики.
2. Составные типы данных: функции и назначение.
3. Циклические структуры в языке C#: способы записи, виды и применение.
4. Строки: тип данных, способ записи, методы.
5. Массивы: способ записи, применение при решении задач.
6. Основные операторы языка C#.

ЛАБОРАТОРНАЯ РАБОТА 2. ЛЕКСИЧЕСКИЙ АНАЛИЗ ВЫРАЖЕНИЙ. ФОРМЫ ЗАПИСИ ВЫРАЖЕНИЙ

Цель работы: *изучить принципы работы обратной польской записи и применить ее для расчета математического выражения.*

Краткая теоретическая часть

Числовое выражение – это запись, составленная по определенным правилам из констант, имен, знаков операций и скобок. Константы и имена в выражении обозначают операнды, а знаки операций со скобками задают последовательность операций.

Обычной формой выражений является *инфиксная*, когда знак бинарной операции записывается между обозначениями операндов этой операции, например a+b.

Если знак операции стоит после операндов, такая запись назовается *постфиксной* или *обратной польской* (ОПЗ), например ab+.

ОПЗ была предложена польским логиком Яном Лукасевич.

Преобразование выражения в ОПЗ с использованием стека

Стек – это структура данных, которая сохраняет элементы по принципу: первым пришел, последним ушел. С точки зрения архитектуры компьютера, стек относится к области памяти, поддерживаемой процессором, в которой сохраняются локальные переменные. Доступ к стеку во много раз быстрее, чем к общей области памяти, поэтому использование стека для хранения данных ускоряет работу вашей программы.

В языке C# переменные размерных типов (например, целые числа) располагаются в стеке, и доступ к ним осуществляется по имени. Переменные ссылочных типов (например, объекты) располагаются в куче.

Куча – это оперативная память вашего компьютера. Доступ к ней осуществляется медленнее, чем к стеку. Когда объект располагается в куче, то переменная хранит лишь адрес объекта. Этот адрес хранится в стеке. По адресу программа имеет доступ к самому объекту, все данные которого сохраняются в общем куске памяти (куче).

Таблица 4. Правила построения ОПЗ

№ п/п	Значение	Действие
1	Идентификатор, число	Поместить символ (набор символов) идентификатора в ОПЗ
2	Оператор:	
2а	открывающаяся скобка	поместить скобку в стек
2б	арифметический оператор	выталкивать из стека операторы в ОПЗ до тех пор, пока приоритет текущего оператора будет ниже или равным приоритета оператора, находящегося в вершине стека
3	закрывающаяся скобка	вытолкнуть из стека в ОПЗ все операторы до первой открывающейся скобки. Эту скобку из стека тоже вытолкнуть, но в ОПЗ не заносить.
4	Завершение выражения	вытолкнуть из стека в ОПЗ все оставшиеся операторы

Рассмотрим алгоритм на примере простейшего выражения:

Таблица 5. Алгоритм построения ОПЗ для выражения a + (b - c) * d

Символ	Действие	Состояние выходной строки после совершенного действия	Состояние стека после совершенного действия
a	'a' – переменная. Помещаем ее в выходную строку	a	пуст
+	'+' – знак операции. Помещаем его в стек (поскольку стек пуст, приоритеты можно не проверять)	a	+
(	'(' – открывающаяся скобка. Помещаем в стек.	a	+ (
b	'b' – переменная. Помещаем ее в выходную строку	a b	+ (
-	'-' – знак операции, который имеет приоритет 2. Проверяем стек: на вершине находится символ '(', приоритет которого равен 1. Следовательно мы должны просто поместить текущий символ '-' в стек.	a b	+ (-
c	'c' – переменная. Помещаем ее в выходную строку	a b c	+ (-
)	')' – закрывающаяся скобка. Извлекаем из стека в выходную строку все символы, пока не встретим открывающую скобку. Затем уничтожаем обе скобки.	a b c -	+
*	'*' – знак операции, который имеет приоритет 3. Проверяем стек: на вершине находится символ '+', приоритет которого равен 2, т.е. меньший, чем приоритет текущего символа '*'. Следовательно, мы должны просто поместить текущий символ '*' в стек.	a b c -	+ *
d	'd' – переменная. Помещаем ее в выходную строку	a b c - d	+ *

Теперь вся входная строка разобрана, но в стеке еще остаются знаки операций, которые мы должны просто извлечь в выходную строку. Поскольку стек – это структура, организованная по принципу

LIFO, сначала извлекается символ '*', затем символ '+'. Итак, мы получили конечный результат: a b c - d * +

Алгоритм вычисления выражения, записанного в ОПЗ

В качестве входной строки рассматриваем выражение, записанное в форме ОПЗ: 1. Если очередной символ входной строки – число, то кладем его в стек. 2. Если очередной символ – знак операции, то извлекаем из стека два верхних числа, используем их в качестве операндов для этой операции, затем кладем результат обратно в стек. Когда вся входная строка будет разобрана, в стеке должно остаться одно число, которое и будет результатом данного выражения.

Таблица 6. Пример вычисления выражения 7 5 2 - 4 * + по ОПЗ

Символ	Действие	Состояние стека после совершенного действия
7	'7' – число. Помещаем его в стек.	7
5	'5' – число. Помещаем его в стек.	7 5
2	'2' – число. Помещаем его в стек.	7 5 2
-	'-' – знак операции. Извлекаем из стека два верхних числа (5 и 2) и совершаем операцию 5 - 2 = 3, результат которой помещаем в стек	7 3
4	'4' – число. Помещаем его в стек.	7 3 4
*	'*' – знак операции. Извлекаем из стека два верхних числа (3 и 4) и совершаем операцию 3 * 4 = 12, результат которой помещаем в стек	7 12
+	'+' – знак операции. Извлекаем из стека два верхних числа (7 и 12) и совершаем операцию 7 + 12 = 19, результат которой помещаем в стек	19

Задания к работе

Написать программу, представляющую математические выражения в форме обратной польской записи и выполняющую расчет значения выражения по ОПЗ. Для тестирования использовать следующие варианты выражений:

1)R=a/(b-c)*(d+e) a=8.6 b=2.4 c=5.1 d=0.3 e=7.9 R=-26.12

2)R=(a+b)*(c-d)/e a=7.4 b=3.6 c=2.8 d=9.5 e=0.9 R=-81.89

3)R=a-(b+c*d)/e a=3.1 b=5.4 c=0.2 d=9.6 e=7.8 R=2.16

4)R=a/b-((c+d)*e) a=1.2 b=0.7 c=9.3 d=6.5 e=8.4 R=-131.006
5)R=a*(b-c+d)/e a=9.7 b=8.2 c=3.6 d=4.1 e=0.5 R=168.78
6)R=(a+b)*(c-d)/e a=0.8 b=4.1 c=7.9 d=6.2 e=3.5 R=2.38
7)R=a*(b-c)/(d+e) a=1.6 b=4.9 c=5.7 d=0.8 e=2.3 R=-0.413
8)R=a/(b*(c+d))-e a=8.5 b=0.3 c=2.4 d=7.9 e=1.6 R=1.151
9)R=(a+(b/c-d))*e a=5.6 b=7.4 c=8.9 d=3.1 e=0.2 R=0.666
10)R=a*(b+c)/(d-e) a=0.4 b=2.3 c=6.7 d=5.8 e=9.1 R=-1.091
11)R=a-(b/c*(d+e)) a=5.6 b=3.2 c=0.9 d=1.7 e=4.8 R=-17.51
12)R=(a-b)/(c+d)*e a=0.3 b=6.7 c=8.4 d=9.5 e=1.2 R=-0.429
13)R=a/(b+c-d*e) a=7.6 b=4.8 c=3.5 d=9.1 e=0.2 R=1.173
14)R=a*(b-c)/(d+e) a=0.5 b=6.1 c=8.9 d=2.4 e=7.3 R=-0.144
15)R=(a+b*c)/(d-e) a=9.1 b=0.6 c=2.4 d=3.7 e=8.5 R=-2.196

Контрольные вопросы и задания

1. Дайте определение ОПЗ.
2. Что называется стеком? Что такое «куча»?
3. Поясните принцип формирования ОПЗ и вычисления значения выражения.
4. Приведите примеры применения ОПЗ в реальных программах.

ЛАБОРАТОРНАЯ РАБОТА 3. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Цель работы: *написать программу осуществляющую поиск подходящих подстрок по указанному регулярному выражению.*

Краткая теоретическая часть

Регулярные выражения – это один из способов поиска подстрок в строках с помощью некоторого шаблона.

Обычно регулярные выражения используются для:

- проверки наличия соответствующей шаблону подстроки;
- поиска и выдачи пользователю соответствующих шаблону подстрок;
- замены соответствующих шаблону подстрок.

Работа с регулярными выражениями в C#

```
Regex re = new Regex("pattern", "options");
MatchCollection mc = re.Matches("this is just one
test");
iCountMatchs = mc.Count;
```

где re – это новый объект-**Regex**, в чьем конструкторе передается образец поиска (pattern) и опции (options) (таблица 7), задающие различные варианты поиска.

Таблица 7. ***Опции Regex***

Символ	**Значение**
I	Поиск без учета регистра.
m	Многострочный режим, позволяющий находить совпадения в начале или конце строки, а не всего текста.
n	Находит только явно именованные или нумерованные группы в форме (?<name>…). Значение этого будет объяснено ниже, при обсуждении роли скобок в регулярных выражениях.
c	Компилирует. Генерирует промежуточный MSIL-код, перед исполнением превращающийся в машинный код.
s	Позволяет интерпретировать конец строки как обыкновенный символ-разделитель. Часто это значительно упрощает жизнь.
x	Исключает из образца незначащие символы (пробелы, табуляция и т.д.).
r	Ищет справа налево.

Сочетание флагов m и s дает очень удобный режим работы, учитывающий концы строк и позволяющий пропустить все незначащие символы, включая символ конца строки.

Искомые выражения

Выражением может быть один символ или последовательность символов, заключенных в круглые или квадратные скобки. Особенности использования скобок будут описаны ниже.

Классы символов (Character class)

Используя квадратные скобки, можно указать группу символов (это называют классом символов) для поиска. Например, конструкция 'б[аи]ржа' соответствует словам «баржа» и «биржа», т.е. словам, начинающимся с «б», за которым следуют «а» или «и», и заканчивающимся на «ржа».

Возможно и обратное, то есть, можно указать символы, которых не должно содержаться в найденной подстроке. Так, '[^1-6]' находит все символы, кроме цифр от 1 до 6. Следует упомянуть, что внутри класса символов '\b' обозначает символ backspace (стирания).

Таблица 8. Символы записи регулярных выражений

Символ	Значение
\w	Слово. То же, что и [a-zA-Z_0-9].
\W	Все, кроме слов. То же, что и [^a-zA-Z_0-9].
\s	Любое пустое место. То же, что и [\f\n\r\t\v].
\S	Любое непустое место. То же, что и [^ \f\n\r\t\v].
\d	Десятичная цифра. То же, что и [0-9].
\D	Не цифра. То же, что и [^0-9].

Квантификаторы (Quantifiers)

Если неизвестно, сколько именно знаков должна содержать искомая подстрока, можно использовать спецсимволы, именуемые мудреным словом квантификаторы. Например, можно написать "hel+o", что будет означать слово, начинающееся с "he", со следующими за ним одно или несколько "l", и заканчивающееся на "o".

Таблица 9. Список квантификаторов

Символ	Описание
*	Соответствует 0 или более вхождений предшествующего выражения. Например, 'zo*' соответствует "z" и "zoo".
+	Соответствует 1 или более предшествующих выражений. Например, "zo+" соответствует "zo" and "zoo", но не "z".
?	Соответствует 0 или 1 предшествующих выражений. Например, 'do(es)?' соответствует "do" в "do" or "does".
{n}	n – неотрицательное целое. Соответствует точному количеству вхождений. Например, 'o{2}' не найдет "o" в "Bob",но найдет два "o"' в "food".
{n,}	n – неотрицательное целое. Соответствует вхождению, повторенному не менее n раз. Например, 'o{2,}' не находит "o" в "Bob", зато находит все "o" в "foooood". 'o{1,}' эквивалентно 'o+'. 'o{0,}' эквивалентно 'o*'.
{n,m}	m и n – неотрицательные целые числа, где n <= m. Соответствует минимум n и максимум m вхождений. Например, 'o{1,3} находит три первые "o" в "fooooood". 'o{0,1}' эквивалентно 'o?'. Пробел между запятой и цифрами недопустим.
*?	Предыдущий элемент не повторяется вообще или повторяется, но как можно меньшее число раз.
+?	Предыдущий элемент повторяется один или несколько раз, но как можно меньшее число раз.
??	Предыдущий элемент не повторяется или повторяется один раз, но как можно меньшее число раз.

Символ	Описание
{ n }?	Предыдущий элемент повторяется ровно n раз.
{ n ,}?	Предыдущий элемент повторяется по крайней мере n раз, но как можно меньшее число раз.
{ n , m }?	Предыдущий элемент повторяется не меньше n и не больше m раз, но как можно меньшее число раз.

Концы и начала строк

Проверка начала или конца строки производится с помощью метасимволов ^ и $. Например, "^thing" соответствует строке, начинающейся с "thing". "thing$" соответствует строке, заканчивающейся на "thing". Эти символы работают только при включенной опции 's'. При выключенной опции 's' находятся только конец и начало текста. Но и в этом случае можно найти конец и начало строки, используя escape-последовательности \A и \Z. В .Net имеется еще и символ \z, точный конец строки.

Граница слова

Привязки, или атомарные утверждения нулевой ширины, приводят к успеху или сбою сопоставления, в зависимости от текущей позиции в строке, но не предписывают обработчику перемещаться по строке или обрабатывать символы.

Таблица 10. Позиционирующие символы

Символ	Значение
^	Начало строки
$	Конец строки, или перед \n в конце строки (см. опцию m).
\A	Началостроки (ignores the m option).
\Z	Конец строки, или перед \n в конце строки (игнорирует опцию m).
\z	Точно конец строки (игнорирует опцию m).
\G	Начало текущего поиска (Часто это в одном символе за концом последнего поиска).
\b	На границе между \w (алфавитно-цифровыми) и \W (не алфавитно-цифровыми) символами. Возвращает true на первых и последних символах слов, разделенных пробелами.
\B	Не на \b-границе.

Вариации и группировка

Символ '|' можно использовать для перебора нескольких вариантов. Использование этого символа совместно со скобками: '(...|...|...)' – позволяет создать группы вариантов.

Работа с регулярными выражениями в .NET. Пространство RegularExpressions

В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.

1. Класс Regex

Объекты этого класса определяют регулярные выражения. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая регулярное выражение. В других вариантах конструктора ему может быть еще передан объект, принадлежащий перечислению **RegexOptions** и задающий опции, которые действуют при работе с данным объектом.

Основные методы класса **Regex**:

Метод **Match** запускает поиск соответствия. В качестве параметра методу передается строка поиска, где разыскивается первая подстрока, которая удовлетворяет образцу, заданному регулярным выражением. В качестве результата метод возвращает объект класса **Match**, описывающий результат поиска. При успешном поиске свойства объекта будут содержать информацию о найденной подстроке.

Метод **Matches** позволяет разыскать все вхождения, то есть все подстроки, удовлетворяющие образцу. У алгоритма поиска есть важная особенность – разыскиваются непересекающиеся вхождения подстрок. Можно считать, что метод **Matches** многократно запускает метод **Match**, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск. В качестве результата возвращается объект **MatchCollection**, представляющий коллекцию объектов **Match**.

Метод **NextMatch** запускает новый поиск, начиная с того места, на котором остановился предыдущий поиск.

Метод **Split** является обобщением метода **Split** класса **String**. Он позволяет, используя образец, разделить искомую строку на элементы.

2. Классы Match и MatchCollection

Объекты этих классов создаются автоматически при вызове методов **Match** и **Matches**. Коллекция **MatchCollection** позволяет получить доступ к каждому ее элементу – объекту **Match**. Класс **Match**

является непосредственным наследником класса **Group**, который, в свою очередь, является наследником класса **Capture**. При работе с объектами класса **Match** наибольший интерес представляют не столько методы класса, сколько его свойства, большая часть которых унаследована от родительских классов. Рассмотрим основные свойства:

1) свойства **Index**, **Length** и **Value** наследованы от прародителя **Capture**. Они описывают найденную подстроку – индекс начала подстроки в искомой строке, длину подстроки и ее значение;

2) свойство **Groups** класса **Match** возвращает коллекцию групп – объект **GroupCollection**, который позволяет работать с группами, созданными в процессе поиска соответствия;

3) свойство **Captures**, наследованное от объекта **Group**, возвращает коллекцию **CaptureCollection**.

При работе с регулярными выражениями реально приходится создавать один объект класса **Regex**, объекты других классов автоматически появляются в процессе работы с объектами **Regex**.

3. Классы Group и GroupCollection

Коллекция **GroupCollection** возвращается при вызове свойства **Group** объекта **Match**. Имея эту коллекцию, можно добраться до каждого объекта **Group**, в нее входящего. Класс **Group** является наследником класса **Capture** и, одновременно, родителем класса **Match**. От своего родителя он наследует свойства **Index**, **Length** и **Value**, которые и передает своему потомку.

Особенности создания групп:

1) при обнаружении одной подстроки, удовлетворяющей условию поиска, создается не одна группа, а коллекция групп;

2) группа с индексом 0 содержит информацию о найденном соответствии;

3) число групп в коллекции зависит от числа круглых скобок в записи регулярного выражения. Каждая пара круглых скобок создает дополнительную группу, которая описывает ту часть подстроки, которая соответствует шаблону, заданному в круглых скобках;

4) группы могут быть индексированы, но могут быть и именованными, поскольку в круглых скобках разрешается указывать имя группы.

4. Классы Capture и CaptureCollection

Коллекция **CaptureCollection** возвращается при вызове свойства **Captures** объектов класса **Group** и **Match**. Класс **Match** наследует это свойство у своего родителя – класса **Group**. Каждый объект **Capture**, входящий в коллекцию, характеризует соответствие, захваченное в процессе поиска, – соответствующую подстроку.

5. Перечисление RegexOptions

Объекты этого перечисления описывают опции, влияющие на то, как устанавливается соответствие (таблица 11). Обычно такой объект создается первым и передается конструктору объекта класса **Regex**.

Таблица 11. Элементы перечисления RegexOptions

Имя члена	Описание
None	Указывает на отсутствие заданных параметров.
IgnoreCase	Указывает соответствие, не учитывающее регистр.
Multiline	Многострочный режим. Изменяет значение символов "^" и "$" так, что они совпадают, соответственно, в начале и конце любой строки, а не только в начале и конце целой строки.
ExplicitCapture	Указывает, что единственные допустимые записи являются явно поименованными или пронумерованными группами в форме (?<name>…).
Compiled	Указывает, что регулярное выражение скомпилировано в сборку. Это порождает более быстрое исполнение, но увеличивает время запуска.
Singleline	Указывает однострочный режим. Изменяет значение точки (.) так, что она соответствует любому символу (вместо любого символа, кроме "\n").
IgnorePatternWhitespace	Устраняет из шаблона неизбежные пробелы и включает комментарии, помеченные "#".
RightToLeft	Указывает, что поиск будет выполнен в направлении справа налево, а не слева направо.
CultureInvariant	Указывает игнорирование региональных языковых различий.

6. Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска регулярные выражения. В этом случае необходимо будет создать объект класса **RegexCompilationInfo** и передать ему информацию о регулярных выражениях, подлежащих компиляции, и о том, куда поместить оттранслированную программу. Дополнительно в таких ситуациях следует включить опцию **Compiled**.

Компиляция и повторное использование регулярных выражений в .Net

По умолчанию Regex компилирует регулярные выражения в последовательность внутренних байт-кодов регулярных выражений. При исполнении регулярных выражений байт-код интерпретируется.

Если же конструировать объект Regex с опцией 'c', он компилирует регулярные выражения в MSIL-код вместо упомянутого байт-кода. Это позволяет JIT-компилятору Microsoft .NET Framework преобразовать выражение в родные машинные коды для повышения производительности.

Пример работы с регулярными выражениями

Функция FindMatch производит поиск первого вхождения подстроки, соответствующей шаблону поиска **strpat**, в строку **str**, соответствующей образцу:

```
string FindMatch(string str, string strpat)
{
//создание регулярного выражения
Regex pat = new Regex(strpat);
//получение подстроки, соответствующей шаблону
Match match =pat.Match(str);
string found = "";
if (match.Success)//анализ успешности поиска
{
    //запись найденной подстроки в строку found
    found =match.Value;
    Console.WriteLine("Строка ={0}\tОбразец={1}\
    tНайдено={2}", str,strpat,found);
```

```
}
return(found);
}//FindMatch
```

Для работы с классами регулярных выражений нужно добавить в начало проекта предложение: using System.Text.RegularExpressions.

Тестирование работы с регулярными выражениями:

```
public void TestSinglePat()
{
//поиск по образцу первого вхождения
string str,strpat,found;
Console.WriteLine("Поиск по образцу");
//образец задает подстроку, начинающуюся с символа a,
//далее идут буквы или цифры.
str ="start"; strpat =@"a\w+";
found = FindMatch(str,strpat);
str ="fab77cd efg";
found = FindMatch(str,strpat);
//образец задает подстроку, начинающуюся с символа a,
//заканчивающуюся f с возможными символами b и d в
//середине
strpat = "a(b|d)*f"; str = "fabadddbdf";
found = FindMatch(str,strpat);
//диапазоны и escape-символы
strpat = "[X-Z]+"; str = "aXYb";
found = FindMatch(str,strpat);
strpat = @"\u0058Y\x5A"; str = "aXYZb";
found = FindMatch(str,strpat);
}//TestSinglePat
```

Результаты, полученные при работе этой процедуры (рис. 1).

```
E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
Поиск по образцу
Строка =start   Образец=a\w+    Найдено=art
Строка =fab77cd efg     Образец=a\w+    Найдено=ab77cd
Строка =fabadddbdf      Образец=a(b|d)*f        Найдено=adddbdf
Строка =aXYb    Образец=[X-Z]+  Найдено=XY
Строка =aXYZb   Образец=\u0058Y\x5A     Найдено=XYZ
Press any key to continue_
```

Рис. 1. Регулярные выражения. Поиск по образцу

Задания к работе

1. Дана строка вида «2541312аеполр№»;4523673ьбтва932#@#467-0@». Написать программу с использованием регулярных выражений, которая выводит на экран цифры в виде номеров телефона. Например, 452-36-73.
2. Дана строка вида «dafd#$245243564@#243$yet12&». Написать программу, которая выводит на экран строку из цифр и латинских букв.
3. Дано предложение «Одним из свойств алгоритма является его доступность. Доступность алгоритма позволяет максимально эффективно реализовывать программу». Написать программу с использованием регулярных выражений, которая будет выводить предложение на экран, и подчеркивать или выделять цветом слово ***доступность***.
4. Написать программу, которая будет заменять цифры от 0-9 пустым местом, в строке вида «21415@#3w42y4thgbxcnnxdfb@542531».
5. Даны несколько видов IP-адресов 1231.45.12, 122.12.23, 118.26.0, 178.115.42, 56.108.12, 78.45.255, 117.45.0, 96.25.100, 78.25.0. Напишите программу, которая выведет на экран последовательность адресов с правильной формой записи.
6. Дана строка вида «445@”;”’;/\\utrjkhd%^576ty34t-erwqrt». Составьте программу, которая будет выводить на экран только буквы латинского алфавита.
7. Дана строка вида «734нироаыти5644121 _еррку%:6753674». Составьте программу, которая будет выводить на экран только русские буквы.
8. Даны несколько видов IP-адресов 1231.45.12, 122.12.23, 118.26.0, 178.115.42, 56.108.12, 78.45.255, 117.45.0, 96.25.100, 78.25.0., 54.4402.785, 78.426.45, 0.255.36, 45.259.47. Напишите программу, которая выведет на экран последовательность адресов с неправильной формой записи.
9. Дано предложение вида «Свойство алгоритма массовость позволяет реализовывать программу в различных условиях и средах.». Напишите программу, которая будет выводить на экран предложение, таким образом, чтобы часть предложения «в различных условиях и

средах» заменялась на фразу «независимо от вида программного и аппаратного обеспечения».

10. Напишите программу, которая проверяет правильность ввода пользователем адреса электронной почты. Учесть, что адрес содержит один символ '@', в домене не менее двух символов, в правой части адреса допустима только одна точка.

Вариант задания для выполнения выбирается по таблице.

№ варианта	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
№ задания	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5

Контрольные вопросы и задания

1. Регулярные выражения: определение, способ записи, применение при решении задач
2. Класс Regex и его основные методы реализации в программе.
3. Классы пространства RegularExpressions.
4. Синтаксис записи регулярных выражений.
5. Перечислите квантификаторы и их назначение.

ЛАБОРАТОРНАЯ РАБОТА 4. БАЗОВЫЕ ГРАФИЧЕСКИЕ КОМПОНЕНТЫ, СВОЙСТВА И ОБРАБОТКА СОБЫТИЙ

Цель работы: *изучить способы работы с базовыми графическими компонентами и разработать программу с графическим интерфейсом.*

Краткая теоретическая часть

Рассмотрим создание простейшего приложения с графическим пользовательским интерфейсом. Сначала необходимо создать проект оконного приложения (Windows Application). В среде VS в главном меню выбираем **File -> New -> Project** (рис. 2).

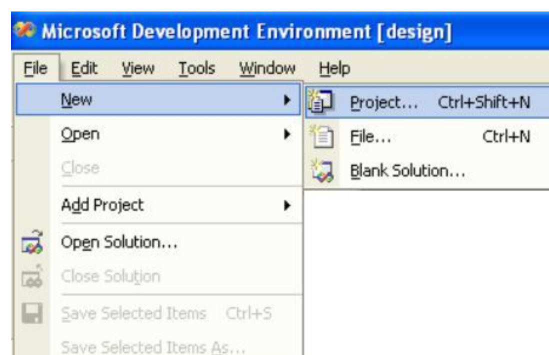


Рис. 2. Создание проекта, шаг 1

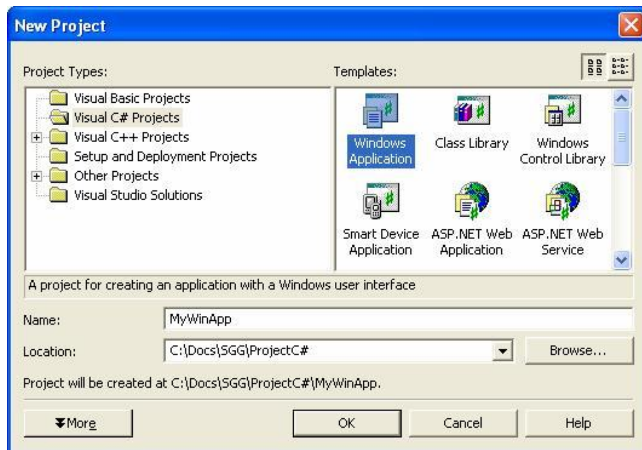


Рис. 3. Создание проекта в среде Visual Studio .NET, шаг 2.

В появившемся окне нужно выбрать тип приложения: **Visual C# Projects -> Windows Application**, путь для размещения проекта и название проекта (рис. 3).

После нажатия на кнопку "OK" появится окно редактора форм, в котором отобразится автоматически созданная форма (**Ошибка! Источник ссылки не найден.**). Окно "Solution Explorer" содержит дерево файлов созданного решения (solution) и список ссылок на библиотеки типов. Среди прочих в этом списке содержится ссылка на библиотеку System.Windows.Forms, в которой и определено одноименное пространство имен.

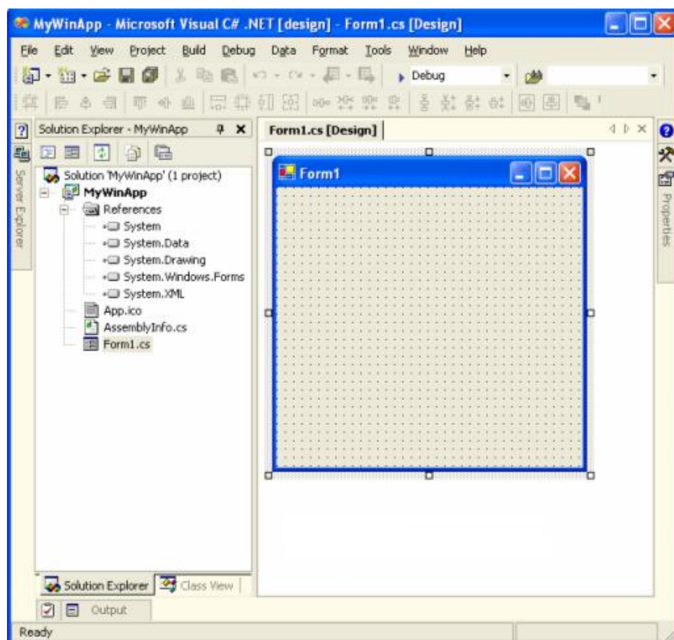


Рис. 4. Окно редактора формы

Для просмотра автоматически сформированного шаблона программы нужно нажать на кнопку "View Code" на панели инструментов окна "Solution Explorer". Этот шаблон должен быть заменен кодом соответствующей программы.

Для создания простейшего оконного приложения понадобится минимум два класса из пространства имен **System.Windows.Forms**:

1. Класс **Application**, предназначенный для управления приложением;
2. Класс **Form**, описывающий форму.

Код создания оконного приложения:

```
namespace MyWinApp
{    // подключаем пространства имен
     using System;
     using System.Windows.Forms;

     public class MainForm : Form
     {// запускаем приложение
          public static int Main(string[] args)
          {  //создаем экземпляр MainForm и запускаем
          Application.Run(new MainForm());//приложение
          return 0;
          }
     }
}
```

Для вывода сообщений в приложениях с графическим интерфейсом используют класс **System.Windows.Forms.MessageBox**. В этом классе реализован статический метод **Show**:

```
public static DialogResult Show(string);
public static DialogResult Show(string, string);
```

где **enum DialogResult** – перечисление, элементы которого характеризуют результат работы диалогового окна (формы). Элементыданногоперечисления: **Abort**, **Cancel**, **Ignore**, **No**, **None**, **OK**, **Retry**, **Yes**.

Пример:

```
MessageBox.Show("Сообщение", "Заголовок");
```

При вызове этого метода отобразится диалоговое окно, представленное на рис. 5.

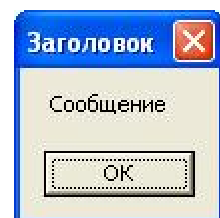


Рис. 5. Окно вывода сообщения

Событием *(event)* называется некоторое действие, вызванное либо действиями пользователя, например, нажатие клавиши мыши или клавиатуры, либо некоторой программой, например, завершение копирования файла. Работа с событиями в C# соответствует модели "издатель – подписчик", согласно которой некоторый класс определяет событие, которое он может инициировать, а другие классы могут "подписаться" на это событие. Класс-подписчик должен реализовать метод, который будет вызван при возникновении события. Такой метод называется *обработ-*

чиком события (event handler). Событие является свойством класса, опубликовавшего его. При создании оконного приложения достаточно:

1. Подписаться на событие. Синтаксис:

```
имя-объекта-издателя.событие+= new EventHandler(имя-обработчика);
```

2. Реализовать обработчик события:

```
private void имя-обработчика (object sender, EventArgs e) { … }
```

Обработчики события не должны возвращать значение и должны принимать два параметра:

1. Источник события – объект класса **System.Object**;

2. Объект, содержащий информацию о событии – экземпляр класса **EventArgs** или любого другого класса, порожденного данным.

Например, класс **System.Windows.Forms.Form** содержит событие **Click**, которое инициируется при нажатии кнопки мыши на форме. Чтобы форма реагировала на нажатие кнопки мыши на ней, нужно создать обработчик данного события, для этого необходимо в некотором методе класса формы (определенного пользователем) подписаться на событие и реализовать обработчик события:

```
public class MainForm : System.Windows.Forms.Form
{    …
     public MainForm()
     {
     // форма подписывается на собственное событие
     this.Click += new EventHandler(MainForm_Click);
     }

     private void MainForm_Click(object sender,
EventArgs e)
     {
          MessageBox.Show("Щелчок на форме");
     }
}
```

Базовым в иерархии классов элементов управления является класс System.Windows.Forms.Control. Далее рассмотрены некоторые члены этого класса (таблица 12–14).

Таблица 12. Свойства класса System.Windows.Forms.Control

Свойство	Описание
public Control.ControlCollection **Controls** { get; }	Возвращает коллекцию элементов управления.
public bool **Enabled** {get; set;}	Состояние доступности.
public Point **Location** {get; set;}	Позиция элемента управления относительно левого верхнего угла его родительского элемента управления.
public Size **Size** {get; set;}	Ширина и высота.
public virtual string **Text** {get; set;}	Надпись, ассоциированная с элементом управления.
public bool **Visible** {get; set;}	Состояние видимости.

Таблица 13. Методы класса System.Windows.Forms.Control

Метод	Описание
public void Hide();	Устанавливает состояние видимости в "невидимый".
public void Show();	Устанавливает состояние видимости в "видимый".

Таблица 14. События класса System.Windows.Forms.Control

Событие	Описание
Click	Нажатие клавиши мыши на элементе управления.
DoubleClick	Двойное нажатие клавиши мыши на элементе управления.
KeyPress	Нажатие клавиши клавиатуры, когда элемент управления находится в фокусе.
Move	Изменение места расположения.
Resize	Изменение размера.

В классе **Control** определен вложенный класс **ControlCollection**. Класс представляет коллекцию для хранения списка элементов управления, ассоциированных с данным элементом управления. Для работы со списком этот класс содержит следующие члены (таблица 15).

Таблица 15. Члены класса Control.ControlCollection

Название	Описание
public virtual void **Add**(Control control);	Добавляет элемент управления control в коллекцию.
public virtual void **AddRange**(Control[]	Добавляет один или несколько элементов

Название	Описание
controls);	управления массива controls в коллекцию.
public virtual void **Remove**(Control control);	Удаляет элемент управления control из коллекции.
public virtual void **Clear**();	Удаляет все элементы управления из коллекции.
public virtual int **Count** {get; }	Возвращает количество элементов управления в коллекции.

Класс **System.Windows.Forms.Form** описывает форму, на которой можно размещать другие элементы управления.

Поскольку класс Form является потомком класса Control, то он содержит поле класса ControlCollection.

Итак, для того чтобы добавить элемент управления на форму, необходимо:

1. Создать элемент управления;
2. Добавить созданный элемент управления в коллекцию **ControlCollection**. Если этого не сделать, элемент управления на форме отображен не будет.

Пример:

```
public class MainForm : Form
{   …
     private Control ctrl = new Control();
     public MainForm()
     {// Добавление в коллекцию
          Controls.Add(ctrl);
     }
}
```

В рассмотренном примере создан экземпляр класса **Control**. Если заменить его на конкретный элемент управления, то этот элемент будет отображен на форме.

В таблицах 16–17 приведены основные свойства и методы форм.

Таблица 16. Методы класса Form

Метод	Описание
public void **Close**();	Закрывает форму.
public DialogResult **ShowDialog**();	Модально отображает форму.

Таблица 17. События класса Form

Событие	Описание
Closed	Форма закрыта.
Closing	Форма закрывается.
MouseMove	Движение курсора мыши по форме.

Форма отображается *модально*, если родительская форма становится недоступной до тех пор, пока отображенное окно не будет закрыто. Пример модального отображения формы – вызов метода **MessageBox.Show**.

В пространстве имен **System.Windows.Forms** определено четыре класса для работы с меню.

Абстрактный класс **Menu** предоставляет базовую функциональность для контекстного и главного меню. Он содержит вложенный класс **MenuItemCollection**. Этот класс представляет коллекцию для хранения списка подменю данного меню. Коллекция содержит элементы класса **MenuItem**.

Класс **MenuItem** описывает строку меню, которая может содержать один или несколько подпунктов меню.

Класс **MainMenu** и класс **ContextMenu** описывают главное и контекстное меню соответственно. Они могут содержать один или несколько подпунктов меню типа **MenuItem**.

Класс Menu содержит поле класса MenuItemCollection. Для обращения к этому полю используется свойство

```
public Menu.MenuItemCollection MenuItems {get; }
```

Для работы с коллекцией элементов **MenuItem** в классе **MenuItemCollection** определены следующие члены (таблица 18).

Таблица 18. Члены класса Menu.MenuItemCollection

Название	Описание
public virtual MenuItem **Add** (string itemName);	Добавляет подпункт меню с именем itemName в коллекцию и возвращает созданный пункт меню.
public virtual void **AddRange** (MenuItem[] items);	Добавляет один или несколько пунктов меню массива items в коллекцию.
public virtual void **Remove**(MenuItem item);	Удаляет пункт меню item из коллекции.

Название	Описание
public virtual void **Clear**();	Удаляет все пункты меню из коллекции.
public virtual int **Count** {get; }	Возвращает количество пунктов меню в коллекции.

Для обработки события нажатия на пункт меню используется событие **Click** класса **MenuItem**.

Пример:

Создадим главное меню, содержащее пункт подменю "File", который в свою очередь содержит подпункты "New" и "Exit". Используя свойство формы Menu, ассоциируем созданное меню с формой.

```
public class MainForm : Form
{
// создание главного меню
private MainMenu mainMenu = new MainMenu();
// создание подпунктов меню
private MenuItem mnuFile = new MenuItem("File");
private MenuItem mnuFileNew = new MenuItem("New");
private MenuItem mnuFileExit = new MenuItem("Exit");

public MainForm()
{
// организация иерархии пунктов меню
mnuFile.MenuItems.AddRange(mnuFileNew, mnuFileExit);
mainMenu.MenuItems.Add(mnuFile);
// связь с главным меню формы
Menu = mainMenu;
Text = "Main menu test";
}
…
}
```

При запуске программы появится форма (рис. 6).

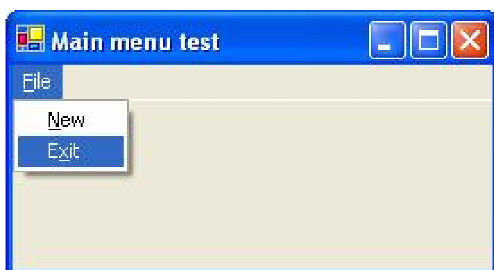


Рис. 6. Создание главного меню

Контекстное меню в пространстве имен **System.Windows.Forms** представляется классом **ContextMenu**.

Пример: Создадим контекстное меню, содержащее подпункты меню "Copy", "Cut" и "Paste".

```
public class MainForm : Form
```

```
{
     private ContextMenu cntxMenu = new ContextMenu();

     private MenuItem mnuCopy = new MenuItem("Copy");
     private MenuItem mnuCut = new MenuItem("Cut");
     private MenuItem mnuPaste= new MenuItem("Paste");

     public MainForm()
     {
          cntxMenu.MenuItems.Add(mnuCopy);
          cntxMenu.MenuItems.Add(mnuCut);
          cntxMenu.MenuItems.Add(mnuPaste);

          ContextMenu = cntxMenu;
          Text = "Contextmenutest";
     }
     …
}
```

При запуске программы появится форма (рис. 7).

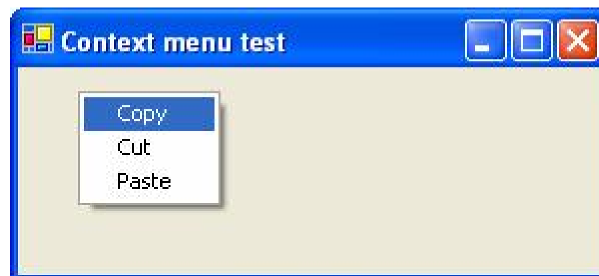


Рис. 7. Создание контекстного меню

В пространстве имен System.Windows.Forms определено четыре класса для работы с кнопками.

Класс Button описывает кнопку. Чаще всего взаимодействие пользователя с кнопкой ограничивается нажатием. Класс **Button** реализует свойство

```
public virtual DialogResult DialogResult {get; set;}
```

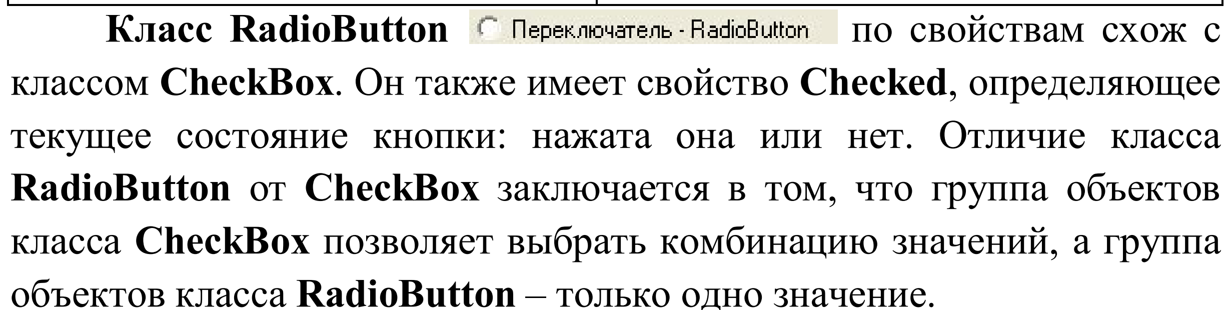
Это свойство определяет, какое значение вернет форма, содержащая кнопку, при закрытии.

Класс CheckBox описывает кнопку с флажком
.

Кнопка с флажком может находиться в одном из трех либо в одном из двух состояний, в зависимости от значения свойства **ThreeStates**. Свойства кнопок с флажками приведены в таблице 19.

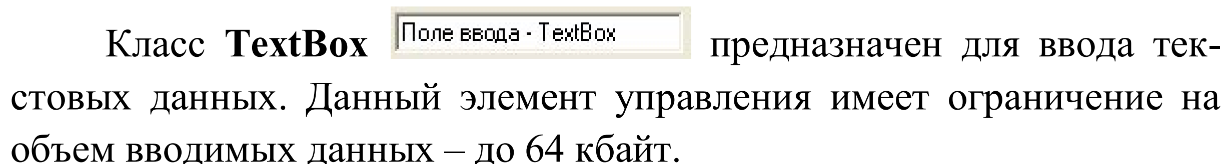
Таблица 19. Свойства класса CheckBox

Свойство	Описание
public Appearance **Appearance** {get; set;}	Вид флажка.
public bool **Checked** {get; set;}	Текущее состояние кнопки-флажка (помечен / непомечен).
public CheckState **CheckState** {get; set;}	Текущее состояние кнопки-флажка при использовании "трех состояний".
public bool **ThreeStates** {get; set;}	Режим "три состояния".

Класс RadioButton 

по свойствам схож с классом **CheckBox**. Он также имеет свойство **Checked**, определяющее текущее состояние кнопки: нажата она или нет. Отличие класса **RadioButton** от **CheckBox** заключается в том, что группа объектов класса **CheckBox** позволяет выбрать комбинацию значений, а группа объектов класса **RadioButton** – только одно значение.

Класс Label предназначен для создания подписей к другим элементам управления или для вывода сообщений на поверхности формы.

Класс **TextBox** 

предназначен для ввода текстовых данных. Данный элемент управления имеет ограничение на объем вводимых данных – до 64 кбайт.

Таблица 20. Свойства класса TextBox

Свойство	Описание
public bool AutoSize {get; set;}	Автоматическое изменение размера поля ввода при изменении размера шрифта.
public int MaxLength {get; set;}	Максимальное количество символов в поле ввода.
public bool Multiline {get; set;}	Однострочный / многострочный режимы.
public bool ReadOnly {get; set;}	Доступ только для чтения.
public string SelectedText {get; set;}	Выделенный текст.
public int SelectionLength {get; set;}	Длина выделенного текста.
public CharacterCasing Character-Casing {get; set;}	Регистр вводимых символов: CharacterCasing.Lower CharacterCasing.Normal

Свойство	Описание
	CharacterCasing.Upper
public char PasswordChar {get; set;}	Символ, используемый для отображения вводимых данных. Используется для ввода паролей.

Задания к работе

1. Написать программу, решающую квадратное уравнение. Интерфейс программы должен выглядеть, как показано на рис. 8.

Пользователь вводит коэффициенты уравнения и нажимает на кнопку "Найти корни", после чего в отдельном окне выводится решение: два значения корня, одно значение корня, либо корней нет.

При изменении значений в полях ввода коэффициентов текст метки "Уравнение:…" меняется в соответствии с новым значением. Коэффициенты "a", "b" и "c" квадратного уравнения – целые числа.

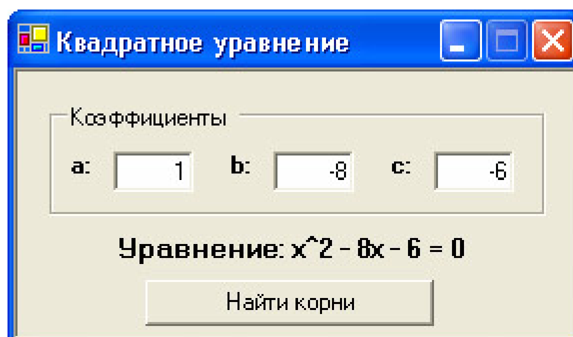


Рис. 8. Интерфейс программы "Квадратное уравнение"

2. Написать программу тестирования. Пользователю предлагается ответить на 10 вопросов. На каждый вопрос предусматривается четыре варианта ответов. Интерфейс программы должен состоять из трех окон: окно приветствия (рис. 10), окно тестирования (рис. 9), окно результатов.

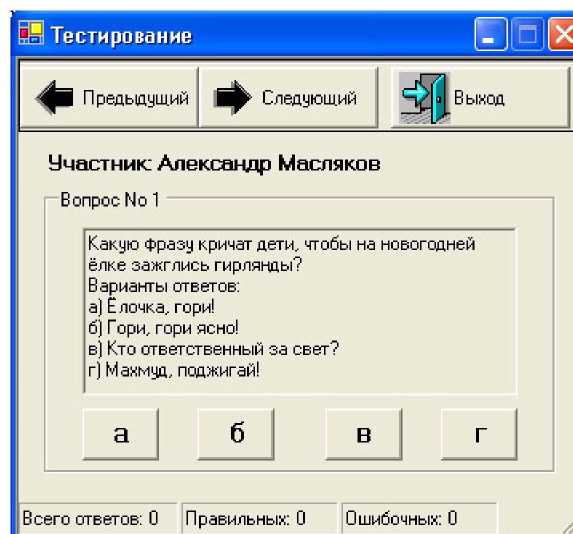


Рис. 9. Окно тестирования

При запуске программы появляется окно приветствия. Пользователь вводит своё имя и нажимает кнопку "Готово". После этого появляется окно тестирования. Текст метки "Участник:" содержит

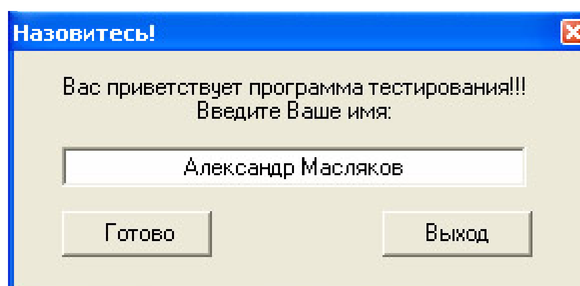


Рис. 10. Окно приветствия

имя тестируемого. Текст группы содержит номер вопроса. В поле ввода отображается текст вопроса и варианты ответов, содержащиеся в файле "Questions.dat".При нажатии на одну из кнопок с буквами "а", "б", "в" или "г" в отдельном окне выводится сообщение о правильности ответа. Строка состояния содержит три панели:

1) Левая панель – "Всего ответов";
2) Средняя панель – "Правильных", отображает число верных ответов;
3) Правая панель – "Ошибочных:", отображает количество неправильных ответов.

Панель инструментов содержит кнопки переключения вопросов и кнопку "Выход". Если пользователь ответил на все вопросы, выводится окно результатов.

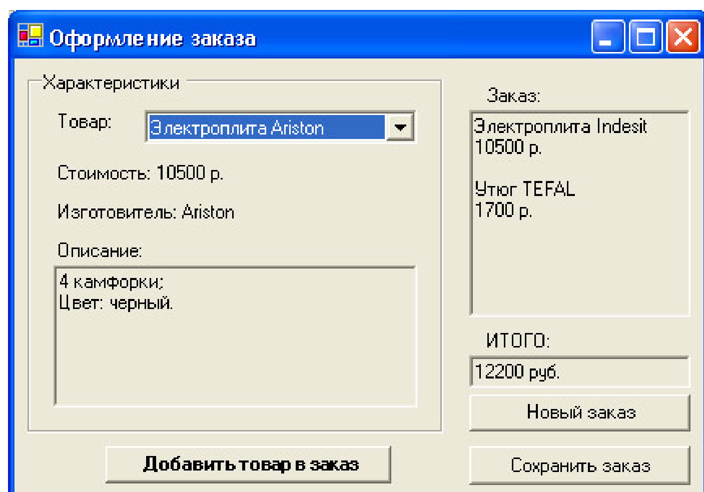


Рис. 11. Интерфейс для программы "Оформление заказа"

3. Написать программу, предоставляющую возможность составить заказ на приобретение некоторого товара. Интерфейс программы должен иметь следующий вид (рис. 11):

В выпадающем списке "Товар:" содержатся наименования товаров, доступных для покупки. При выборе товара текст метки "Стоимость:…", текст метки "Изготовитель:" и текст поля ввода "Описание:" меняют свои значения на значения, соответствующие выбранному товару. Комбинированный список "Товар:" заполняется при запуске программы. Значения для списка хранятся в исходном файле "PriceList.dat" в произвольном формате. В этом же файле хранятся стоимость, изготовитель и описание товара.

При нажатии на кнопку "Добавить товар в заказ" в поле ввода "Заказ:" добавляется запись о выбранном товаре и пересчитывается общая стоимость в поле ввода "ИТОГО:". При нажатии на кнопку

"Новый заказ" очищаются поля ввода "Заказ:" и "ИТОГО:". При нажатии на кнопку "Сохранить заказ" – заказ и общая стоимость сохраняются в файле.

Пользователь выбирает товар из списка, нажимает на кнопку "Добавить товар в заказ", затем выбирает другой товар, снова нажимает на кнопку "Добавить товар в заказ" и т.д. После того, как добавлены все товары для приобретения, пользователь нажимает на кнопку "Сохранить заказ" и выбирает файл для сохранения. Для открытия и сохранения файлов использовать диалоговые окна OpenFileDialog и SaveFileDialog.

4. Написать программу "Словарь". Программа предоставляет возможность перевода слов с русского на английский язык, и с английского – на русский. Интерфейс программы должен выглядеть, как показано на рис. 12.

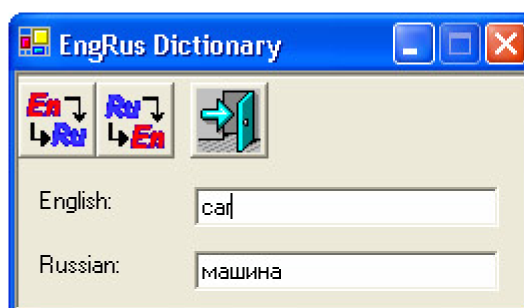


Рис. 12. Интерфейс программы "Словарь"

Словарь хранить в файлах EnRuDict.dat и RuEnDict.dat.

Пример:

EnRuDict.dat

car машина

dog собака

RuEnDict.dat

машина car

собака dog

5. Написать программу "Расширение словаря", расширяющую словарный запас программы *"Словарь"*. Интерфейс программы должен иметь следующий вид (рис. 13). Русско-английский словарь находится в файле RuEnDict.dat, англо-русский – в EnRuDict.dat. При

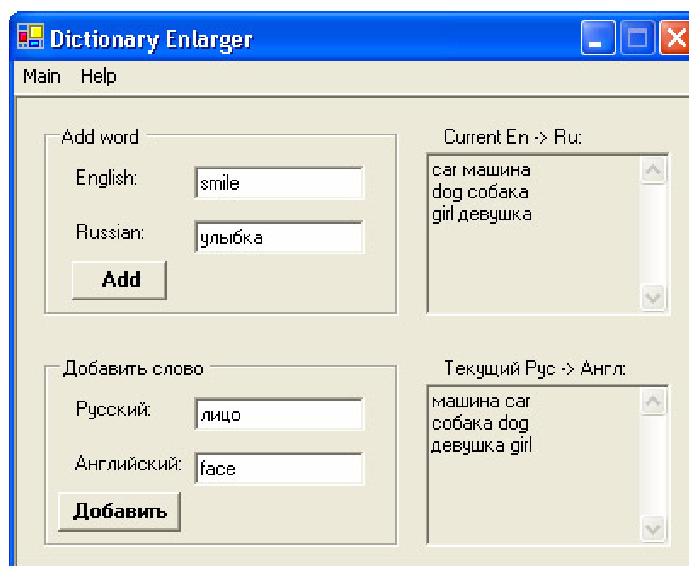


Рис. 13. Интерфейс для программы "Расширение словаря".

нажатии на кнопку "Add" словарь пополняется парой слов из полей ввода "English" и "Russian:". При нажатии на кнопку "Добавить" словарь пополняется парой слов из полей "Русский:" и "Английский:". В многострочных полях ввода "Current En->Ru" и "Текущий Рус -> Англ.", доступных только для чтения, отображаются содержания файлов EnRuDict.dat и RuEnDict.dat соответственно. Меню "Main" содержит только подменю "Exit", при нажатии на которое программа завершает свою работу. Меню "Help" содержит подменю "About", при нажатии на которое в отдельном окне отображается информация о назначении приложения и об её разработчике.

6. Написать программу, позволяющую создавать текстовый файл со списком факультетов. Каждая запись о факультете имеет три параметра:

1. Название факультета;
2. Ф.И.О. декана факультета;
3. Аббревиатура факультета.

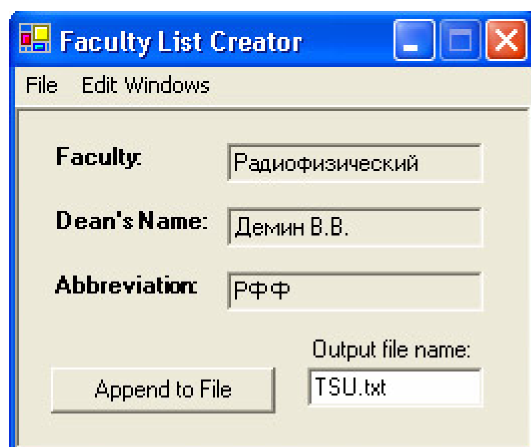


Рис. 14. Главное окно программы "Создание списка факультетов"

Интерфейс программы должен состоять из двух окон:

1. главное окно (рис. 14);
2. окно заполнения полей ввода (рис. 15).

Поля ввода доступны только для чтения. Чтобы изменить значение в поле ввода, необходимо воспользоваться меню "Edit Windows" (рис. 16). При выборе подменю "Faculty" появляется окно, показанное на рис. 15. При выборе подменю "Dean's Name" появляется то же окно, только текст метки "New faculty name:" изменяется на "New Dean's name:". При выборе "Abbreviation" изменяется на "New abbreviation:". В поле ввода "Output file name:" вводится имя выходного файла. При нажатии на кнопку "Append to File" в упомянутый файл

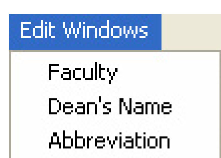


Рис. 16. Меню "Edit Windows"

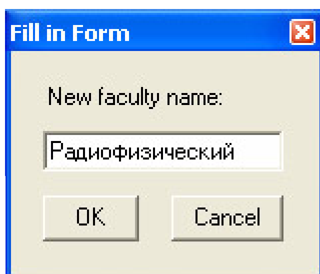


Рис. 15. Окно заполнения полей ввода

добавляется информация в следующем формате: Факультет:<Текст в поле ввода "Faculty"> Декан: <Текст в поле ввода "Dean's Name"> Аббревиатура: <Текст в поле ввода "Abbreviation">. Если выходного файла не существует, его следует создать. Меню "File" содержит только подменю "Exit", при выборе которого приложение завершает свою работу.

Пример выходного файла TSU.txt:

Факультет: Радиофизический Декан: Демин В.В. Аббревиатура: РФФ

Факультет: Физический Декан: Кузнецов В.М. Аббревиатура: ФФ.

Примечание: чтобы символы кириллицы выводились в файл, необходимо при открытии файла указать кодировку Unicode.

7. Написать программу "Учет пользователей". Программа хранит информацию о пользователях: имя, фамилию и адрес электронной почты. Изменять информацию пользователи могут только о себе. Интерфейс программы должен состоять из трёх окон:

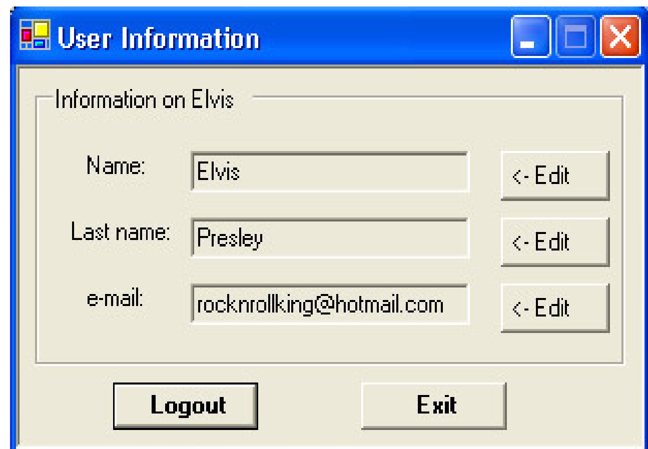


Рис. 17. Главное окно программы

1. Главное окно программы (рис. 17).
2. Окно авторизации (рис. 18).
3. Окно создания новой учетной записи (рис. 19).

Рис. 19. Окно авторизации

Рис. 18. Окно создания новой учетной записи

При запуске программы появляется *окно авторизации*. Если пользователя нет в списке авторизированных пользователей, то при нажатии на кнопку "OK" появляется сообщение об ошибке. При нажатии на кнопку "Create User" появляется *окно создания новой учетной записи*. Информацию об учетных записях хранить в файле "Accounts.txt" в произвольном формате. После нажатия на кнопку "Create" в окне создания учетной записи появляется *главное окно программы*. Если же пользователь есть в списке авторизированных пользователей, при нажатии на кнопку "OK" появляется *главное окно программы*. Поля ввода в окне доступны только для чтения. Чтобы изменить значение в поле ввода, введены три кнопки "<- Edit". Это элементы управления типа CheckBox со значением свойства Apperance = Button. Эти кнопки изменяют состояния полей ввода с "Только для чтения" на "Чтение / запись" и обратно. Информацию о пользователях (Name, Last name, e-mail) требуется хранить в файле (файлах) в произвольном формате.

Нумерация вариантов 1,2,3,4,5,6,7 соответствуют заданию, варианты 8,9,10,11,12,13,14 соответствуют заданию 1,2,3,4,5,6,7 соответственно. 15 вариант выполняет задание под номером 1.

Контрольные вопросы и задания

1. Опишите поэтапно процесс создания WinApi–приложения.
2. Опишите функции диалогового окна и наиболее частые примеры использования данного графического элемента.
3. Перечислите основные базовые графические компоненты.
4. Опишите принцип наследования графического элемента формы.
5. Опишите процесс обработки события при добавлении графического элемента.

ЛАБОРАТОРНАЯ РАБОТА 5. РАБОТА С ГРАФИЧЕСКИМИ КОМПОНЕНТАМИ: TREEVIEW, DATAGRIDVIEW, LISTBOX

Цель работы: *изучение особенностей работы, основных свойств и методов составных элементов управления.*

Краткая теоретическая часть

1. ListBox

Данный графический элемент предназначен для отображения коллекции значений или объектов, в простейшем случае **ListBox** выводит список строк.

Особенность **ListBox** заключается в том, что каждый элемент это отдельный объект со своим значением и возможностью доступа к нему.

Основным свойством **ListBox** является **Items -** коллекция элементов, работа с которой осуществляется следующими методами (таблица 21).

Таблица 21. Методы работы с элементами списка значений listBox

Метод	Действие
Items[]	Предоставляет доступ к массиву элементов по индексу
Items.Add()	Добавляет элемент
Items.Clear()	Очищает массив элементов
Items.Indexof()	Получает индекс элемента в массиве
Items.Remove()	Удаляет элемент по значению
Items.Removeat()	Удаляет элемент по индексу
Items.Count	Возвращает количество элементов
Items.Insert()	Вставляет элемент в массив в указанный индекс
Items.Contains()	Проверяет, является ли параметр элементом данного списка

Элемент управления **ListBox** (рис. 20) в основном применяется для предоставления пользователю выбора одного или нескольких элементов из списка. С помощью методов **SelectedIndex**, **SelectedItem** мы можем получить доступ к выбранному элементу, а используя **SelectedItems**, **SelectedIndices** к массиву выбранных элементов. Для отслеживания изменения состояния выделения используются события **SelectedIndexChanged**, **SelectedValueChanged**.

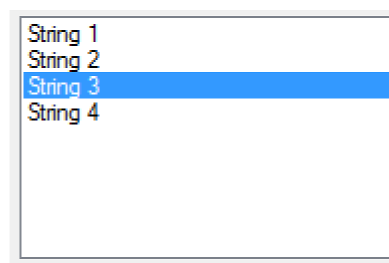


Рис. 20

2. TreeView

Графический элемент **TreeView** (рис. 21) предназначен для отображения иерархической коллекции обозначенных элементов, каждый из которых представлен как **TreeNode**.

TreeView имеет схожий набор методов и классов с **Listbox**. Основное отличие заключается в том, что список элементов называется **Nodes** и каждый элемент имеет свое свойство **Nodes**, что позволяет создавать иерархические списки. Для доступа к элементам иерархического списка используется **TreeView1.Nodes[index]**.

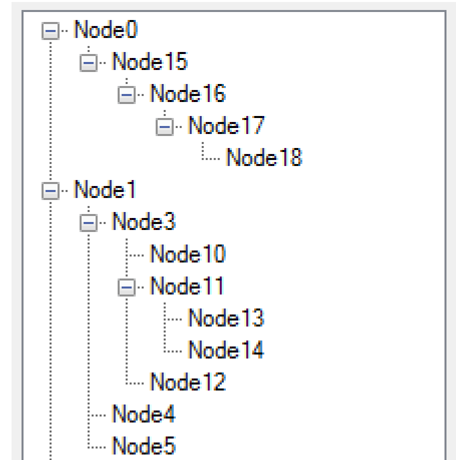


Рис. 21. Элемент управления TreeView

Событие **AfterSelect** возникает при выделении элемента списка, также существуют специфические события, отвечающие за отображение изменения состояния узлов дерева **AfterExpand**, **AfterCollapse,** срабатывающие при разворачивании или сворачивании ветви с элементами иерархического дерева элементов.

3. DataGridView

Графический элемент **DataGridView** предназначен для отображения и редактирования данных в виде таблиц. **DataGridView** имеет три основных свойства: **Columns** содержащий список колонок, **Rows** содержащий список строк и **DataGridView[x,y]** обеспечивающий доступ к двухмерному списку ячеек и их значений. В отличии об предыдущих элементов ячейка является не объектом данных, а элементом управления, содержащим данные, доступные по свойству **Value**. Так как ячейка является управляющим элементом, она может быть нескольких видов, обеспечивающих функционал сходный с элементами TextBox, ComboBox, CheckBox. Для изменения вида ячейки создается объект нужного типа (**DataGridViewComboBoxCell**, **DataGridViewCheckBoxCell**, **DataGridViewTextBoxCell**) и приравнивается к нужной ячейке.

Доступ к ячейкам можно получить через свойства Rows[].Cells[], задав в квадратных скобках номер строки и номер столбца или их на-вания.

Общая структура представлена на рис. 22.

Заголовки столбцов (ColumnHeaders)

Текущая(выбранная) ячейка

.ColumnIndex

.RowIndex

Заголовки строк (RowHeaders)

	ProductName	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
0	Chai	10 boxes x 20 bags	18.0000	39	0
	Chang	24 - 12 oz bottles	19.0000	17	40
	Aniseed Syrup	12 - 550 ml bottles	10.0000	13	70
	Chef Anton's Cajun Seasoning	48 - 6 oz jars	22.0000	53	0
	Chef Anton's Gumbo Mix	36 boxes	21.3500	0	0
5	Grandma's Boysenberry Spread	12 - 8 oz jars	25.0000	120	0

Рис. 22. Графическое представление DataGridView

Доступ к выделенным строкам, колонкам и ячейкам обеспечивают свойства **SelectedCells**, **SelectedRows**, **SelectedColumns**. Они представляют массивы соответствующих выделенных объектов. Основным событием, срабатывающим при выделении, является **SelectionChanged**. В виду того, что **DataGridView** поддерживает редактирование данных пользователем, существует ряд событий описывающих состояние ячейки при начале редактирования **CellBeginEdit**, при окончании **CellEndEdit**, входа в ячейку **CellEnter** и покидание ячейки **CellLeave.**

Задание к работе

1) Написать программу с графическим интерфейсом, которая в поле для ввода получает строку или массив строк, по нажатию кнопки строка должна разбиваться на слова и список слов должен выводиться в ListBox.

2) В элемент DataGridView с тремя столбцами вводятся два числа и слово. По нажатию кнопки заполнить элемент TreeView древовидным списком, в котором имя ветви – это первое число в строке, второе число – принадлежность к родителю, строка – информация. По событию NodeMouseDoubleClick() на ветви или листе должна выделяться строка, содержащая эти данные в DataGridView.

Контрольные вопросы

1. Назначение свойства Nodes элемента списка Treeview.Nodes?
2. Назовите способы доступа к значению ячейки DataGridView?
3. Как получить значение выделенных пользователем значений в каждом из элементов управления?
4. Как получить количество строк и столбцов в таблице DataGridView?
5. Можно ли вставить новый элемент ListBox в указанный пользователем индекс?

ЛАБОРАТОРНАЯ РАБОТА 6. ОБРАБОТКА ИСКЛЮЧЕНИЙ, ИСПОЛЬЗОВАНИЕ МНОГОПОТОЧНОСТИ ДЛЯ ОБЕСПЕЧЕНИЯ РАБОТЫ ИНТЕРФЕЙСА

Цель работы: *изучить понятие «исключение», способы обработки исключений и принципы создания многопоточных приложений.*

Краткая теоретическая часть

Обработка исключительных ситуаций

В общем случае исключительная ситуация – это класс, содержащий набор свойств и методов, который конструируется в момент возникновения ошибки и "запускается" с помощью специальной конструкции языка. В C# запуск исключительной ситуации выполняется с помощью **throw**, а все классы исключительных ситуаций должны быть порождены от **System.Exception**, например:

```
class MyException : System.Exception {}
// ...
throw new MyException();
```

Конструкция **throw** приводит к тому, что все последующие инструкции программы пропускаются до тех пор, пока не будет встречен обработчик исключительной ситуации в защищённом блоке кода. Если необходимый обработчик не будет найден, то исключительная ситуация будет перехвачена обработчиком по умолчанию для приложения.

Для формирования защищённого блока кода в C# используются конструкци:

```
try-catch
try-finally
try-catch-finally
```

Блок **try** содержит код, который может сгенерировать исключительную ситуацию:

```
String s1 = "abc";
try
{
     Double d1 = Convert.ToDouble(s1);
// ERROR: FormatException
}
```

Блок **catch** без аргументов перехватывает исключительные ситуации всех типов. Блок может принимать аргумент типа **System.Exception**, в результате чего будут перехватываться только исключительные ситуации заданного типа.

```
catch (InvalidCastException e)
{...}
```

Допускается использование нескольких блоков **catch**. В этом случае важен порядок перехвата. Необходимо указывать более общие исключения позднее. Если необходимо, то после необходимой обработки исключительная ситуация в блоке **catch** может быть запущена далее с использованием **throw**:

```
catch (InvalidCastException e)
{
     throw (e); // Rethrowing exception e
}
```

или

```
catch{
     throw;
}
```

Блок **finally** может быть использован для освобождения любых ресурсов, распределённых в **try**. Независимо от того, возникла ли ис-

ключительная ситуация или нет, после обработки защищённого блока или блоков **catch** управление всегда передаётся в блок **finally**.

```
finally
{
// этот код выполняется всегда
}
```

Следующий пример демонстрирует использование **try-catch-finally** и **throw** для генерации и обработки ошибок в приложении.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApp
{
    class Program
    {
    static void Main(string[] args)
    {
    try
    {// Защищённый блок кода
    Console.Write("Введите x: ");
    Double x = Convert.ToDouble(Console.ReadLine());
    Console.Write("Введите y: ");
    Double y = Convert.ToDouble(Console.ReadLine());
    // Генерация ошибки деления на ноль
    if (y == 0)
    throw new DivideByZeroException();
    Console.Write("Результат деления: ");
    Console.WriteLine(x / y);
    }
    catch (FormatException e)
    {
    // Вывод сообщения из поля Message исключения
    Console.WriteLine(e.Message);
    }
    catch (DivideByZeroException)
    {// Вывод собственного сообщения
    Console.WriteLine("Ошибка деления на ноль.");
    }
    finally
```

```
            {// Этот код выполняется всегда
            Console.WriteLine("Этот код выполняется все-
        гда!");
            }
            Console.ReadKey();
            }
        }
}
```

Иерархия исключений среды выполнения

Среда выполнения имеет базовый набор исключений, производных от класса **SystemException**, которые порождаются при выполнении отдельных инструкций. В следующей таблице приведен список иерархии стандартных исключений, предоставляемых средой выполнения.

Таблица 22. Список иерархии стандартных исключений

Тип исключения	Базовый тип	Описание
Exception	Object	Базовый класс для всех исключений.
SystemException	Exception	Базовый класс для всех ошибок, созданных средой выполнения.
IndexOutOfRangeException	SystemException	Генерируется только при неправильной индексации массива.
NullReferenceException	SystemException	Создается только при ссылке на пустой объект.
AccessViolationException	SystemException	Создается только при доступе к недопустимому участку памяти.
InvalidOperationException	SystemException	Генерируется методами в случае недопустимого состояния.
ArgumentException	SystemException	Базовый класс для всех исключений аргументов.
ArgumentNullException	ArgumentException	Генерируется методами, для которых не допускается пустое значение аргументов.
ArgumentOutOfRangeException	ArgumentException	Генерируется методами, проверяющими принадлежность аргументов к заданному диапазону.

Многопоточные приложения

С помощью C# можно создавать приложения, которые выполняют несколько задач одновременно. Задачи, которые потенциально могут задержать выполнение других задач, выполняются в отдельных потоках; такой способ организации работы приложения называется *многопоточностью* или *свободным созданием потоков*.

Приложения, использующие многопоточность, более оперативно реагируют на действия пользователя, поскольку пользовательский интерфейс остается активным, в то время как задачи, требующие интенсивной работы процессора, выполняются в других потоках.

Использование компонента BackgroundWorker

Этот класс управляет отдельными потоками указанного метода обработки. Чтобы запустить операцию в фоновом режиме, необходимо создать **BackgroundWorker** и отслеживать события, сообщающие о ходе выполнения операции и сигнализирующие о ее завершении. Можно создать объект **BackgroundWorker** программными средствами или перетащить его в форму из вкладки *Компоненты Панели элементов*. При создании **BackgroundWorker** в Forms Designer, оно появляется в Области компонента, и его свойства отображаются в окне Свойства.

Настройка выполнения операции в фоновом режиме

Чтобы настроить выполнение операции в фоновом режиме, необходимо добавить обработчик события для события **DoWork** и вызвать в нем операцию, которая занимает много времени.

Чтобы начать операцию, вызовите **RunWorkerAsync**. Чтобы получать уведомления о ходе выполнения, необходимо обработать событие **ProgressChanged**. Если необходимо получать уведомление после завершения операции, обработайте событие **RunWorkerCompleted**. Методы, обрабатывающие события **ProgressChanged** и **RunWorkerCompleted** имеют доступ к пользовательскому интерфейсу приложения, так как эти события вызываются в потоке, который вызвал метод **RunWorkerAsync**. Однако обработчик событий **DoWork** не может работать с объектами пользовательского интерфейса, поскольку он запускается в фоновом потоке.

Создание и использование потоков

Если требуется больший контроль над поведением потоков приложения, можно управлять потоками самостоятельно. Однако необходимо иметь в виду, что написание правильных многопоточных приложений может быть сложной задачей. Приложение может перестать отвечать на запросы или могут возникать временные ошибки, вызванные конфликтами.

Новый поток создается путем объявления переменной типа **Thread** и вызова конструктора, которому предоставляется имя процедуры или метода, которые требуется выполнить в новом потоке.

```
System.Threading.Thread newThread =
    new System.Threading.Thread(AMethod);
```

Запуск и остановка потоков

Чтобы начать выполнение нового потока, следует использовать метод **Start**.

```
newThread.Start();
```

Чтобы остановить выполнение потока, следует использовать метод Abort.

```
newThread.Abort();
```

Помимо запуска и остановки потоки можно приостанавливать, вызывая метод **Sleep** или **Suspend**, возобновлять приостановленный поток методом **Resume** и уничтожать поток методом **Abort**.

*Методы управления потоками (*таблица 23*)*

Таблица 23. ***Методы управления отдельными потоками***

Метод	Действие
Start	Запускает поток.
Sleep	Приостанавливает поток на определенное время.
Suspend	Приостанавливает поток, когда он достигает безопасной точки.
Abort	Останавливает поток, когда он достигает безопасной точки.
Resume	Возобновляет работу приостановленного потока
Join	Приостанавливает текущий поток до тех пор, пока не будет завершен другой поток. При заданном времени ожидания этот метод возвращает значение **True** при условии, что другой поток закончится за это время.

Безопасные точки

Безопасные точки располагаются в тех местах кода, в которых среда CLR может безопасно выполнить автоматическую сборку мусора – процесс уничтожения неиспользуемых переменных и освобождения памяти. При вызове методов потока **Abort** или **Suspend** среда CLR анализирует код и определяет подходящее место для остановки потока.

Свойства потока (таблица 24).

Таблица 24. Свойства потоков

Свойство	Значение
IsAlive	Содержит значение True, если поток активен.
IsBackground	Возвращает или задает логическое значение, которое указывает, является ли поток (должен ли являться) фоновым потоком. Фоновые потоки отличаются от основного потока лишь тем, что они не влияют на завершение процесса. Когда обработка всех основных потоков закончена, общеязыковая среда выполнения завершает процесс, применяя метод Abort к тем фоновым потокам, которые еще продолжают существовать.
Name	Возвращает или задает имя потока. Наиболее часто используется для обнаружения отдельных потоков при отладке.
Priority	Возвращает или задает значение, используемое операционной системой для установки приоритетов потоков.
ApartmentState	Возвращает или задает потоковую модель для конкретного потока. Потоковые модели важны, когда поток вызывает неуправляемый код.
ThreadState	Содержит значение, описывающее состояние или состояния потока.

Приоритеты потоков

Каждый поток имеет приоритетное свойство, которое определяет, какую часть процессорного времени он должен занять при выполнении. Операционная система выделяет более длинные отрезки времени на потоки с высоким приоритетом и более короткие на потоки с низким приоритетом. Новые потоки создаются со значением **Normal**, но можно изменить свойство **Priority** на любое значение в перечислении **ThreadPriority**.

Основные и фоновые потоки

Основной поток выполняется бесконечно, тогда, как фоновый поток останавливается сразу после остановки последнего основного

потока. Для определения или изменения фонового статуса потока можно использовать свойство **IsBackground**.

Использование многопоточности для форм и элементов управления

Многопоточность наиболее эффективна при выполнении процедур и методов класса, однако ее можно использовать и при работе с формами и элементами управления. При этом нужно принять во внимание следующее:

1. По возможности методы элемента управления должны выполняться в том же потоке, в котором был создан этот элемент. Если необходимо вызвать метод элемента управления из другого потока, необходимо использовать для вызова метода метод **Invoke**.

2. Не используйте оператор **lock** для блокировки потоков, работающих с элементами управления или формами. Поскольку методы элементов управления и форм иногда выполняют обратный вызов вызывающей процедуры, можно непреднамеренно вызвать взаимоблокировку – ситуацию, когда два потока ожидают друг от друга выполнения действий, необходимых для дальнейшей работы приложения.

Задание к работе

Напишите программу, которая создает форму, которая вычисляет числа Фибоначи. Вычисление выполняется в потоке, отдельном от потока пользовательского интерфейса, так что последний продолжает выполняться без задержек в ходе вычисления. В программе предусмотреть обработку возможных исключительных ситуаций.

Контрольные вопросы и задания

1. Что такое исключение?
2. Как обработать исключительную ситуацию?
3. Какие типы исключений бывают?
4. Назначение многопоточности.
5. Принципы создания и работы с потоками.
6. Свойства и методы компонента BackgroundWorker.

ЛАБОРАТОРНАЯ РАБОТА 7. ФАЙЛЫ

Цель работы: *изучить методы работы с текстовыми и бинарными файлами в среде .NET.*

Краткая теоретическая часть

Пространство имен **System.IO** содержит классы для работы с каталогами и дисками, такие как: **DriveInfo**, **Directory**, **DirectoryInfo**. Классы для работы с файлами: **File**, **FileInfo**. Класс для работы с путями: **Path**. Помимо этого **System.IO** содержит основные классы для работы с потоками: **Stream**, **FileStream**, **StreamReader**, **StreamWriter**, **StringReader**, **StringWriter**, **TextReader**, **TextWriter**, **BinaryReader**, **BinaryWriter**и**MemoryStream**. Для использования вышеперечисленных объектов требуется добавить в код следующую строку кода:

```
using System.IO;
```

Создание файла

Для этого воспользуемся классом **FileInfo** из пространства имен **System.IO**. В конструктор объекта передаем имя будущего файла. Создание файла производится вызовом метода **Create**().

```
FileInfo fi =new FileInfo("new.txt");
fi.Create();
```

Удаление файла

Все аналогично предыдущему примеру. Для удаления файла используем метод **Delete**().

```
FileInfo fi =newFileInfo("new.txt");
fi.Delete();
```

Запись в текстовый файл

Для записи в файл мы будем использовать символьные потоки позволяющие оперировать непосредственно с символами Unicode.

```
StreamWriter sw =new StreamWriter("name.txt");
sw.WriteLine("некоторый текст");
sw.Close();
```

После всех действий над файлом не забываем закрывать его используя метод **Close**(). Приведенный выше пример записи файла полностью перезаписывает его, если же надо дописать в конец файла:

```
StreamWriter sw;
FileInfo fi =new FileInfo("name.txt");
sw= fi.AppendText();
sw.WriteLine("некоторый текст");
sw.Close();
```

Чтение из текстового файла

Для того чтобы файл был корректно считан (без всяких непонятных символов), он должен быть сохранен в юникоде.

```
StreamReader streamReader =new StreamReader("n.txt");
string str ="";
while(!streamReader.EndOfStream)
{
    str+= streamReader.ReadLine();
}
```

Для корректной работы с потоками ввода/вывода рекомендуется использовать конструкцию **using**:

```
using( StreamWriter streamWriter=new StreamWriter(
"n.txt"))
{
// ...
}
```

По выходу из блока **using**, поток, с которым работали, автоматически закроется.

Пути

Класс **Path** позволяет формировать пути к каталогам и файлам, имеет методы для анализа имен файлов. Класс **Path** работает исключительно с путями (со строками) и не производит ни каких реальных манипуляций с файлами и папками, он лишь позволяет получить все необходимое для осуществления этих действий.

Функцию **Combine** позволяет объединить фрагменты путей в один путь. Например, у вас есть путь к каталогу и отдельно имя файла (или другого каталога), и вам необходимо объединить их в один путь:

```
String a = @"C:\Папка";
String b = "файл.exe";
String p = Path.Combine(a,
b);
MessageBox.Show(
String.Format("Фрагмент 1:
{0}\r\nФрагмент 2:
{1}\r\nРезультат
объединения фрагментов:
{2}", a, b, p));
```

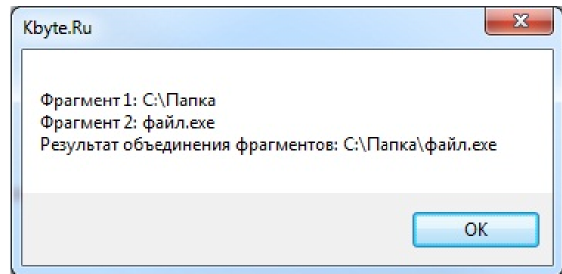


Рис. 23. Результат объединения фрагментов пути в один при помощи функции Combine класса Path

Таблица 25. Методы класса Path

Имя	Описание
ChangeExtension	Изменяет расширение строки пути.
Combine(String[])	Объединяет массив строк в путь.
Combine(String, String)	Объединяет две строки в путь.
Combine(String, String, String)	Объединяет три строки в путь.
Combine(String, String, String, String)	Объединяет четыре строки в путь.
GetDirectoryName	Возвращает для указанной строки пути сведения о каталоге.
GetExtension	Возвращает расширение указанной строки пути.
GetFileName	Возвращает имя файла и расширение указанной строки пути.
GetFileNameWithoutExtension	Возвращает имя файла указанной строки пути без расширения.
GetFullPath	Возвращает для указанной строки пути абсолютный путь.
GetInvalidFileNameChars	Получает массив, содержащий символы, которые не разрешены в именах файлов.
GetInvalidPathChars	Получает массив, содержащий символы, которые не разрешены в именах путей.
GetPathRoot	Получает сведения о корневом каталоге для указанного пути.
GetRandomFileName	Возвращает произвольное имя каталога или файла.
GetTempFileName	Создает на диске временный пустой файл с уникальным именем и возвращает полный путь этого файла.
GetTempPath	Возвращает путь к временной папке текущего пользователя.

Имя	Описание
HasExtension	Определяет, включает ли путь расширение имени файла.
IsPathRooted	Получает значение, показывающее, содержит ли указанная строка пути корень.

Диски

Для работы с дисками предназначен класс **DriveInfo**. Этот класс имеет всего один статичный метод – **GetDrives**, который возвращает массив дисков компьютера. Каждый элемент массива является экземпляром класса **DriveInfo**.

```
StringBuilder driveList = new StringBuilder();
foreach(DriveInfo d inDriveInfo.GetDrives())
{
if(d.IsReady) driveList.AppendLine( String.Format(
"Диск: {0}; метка тома: {1}; файловая система: {2};
тип: {3}; объем: {4} байт; свободно: {5} байт",
d.Name, d.VolumeLabel, d.DriveFormat, d.DriveType,
d.TotalSize, d.AvailableFreeSpace));
}
MessageBox.Show(driveList.ToString());
```

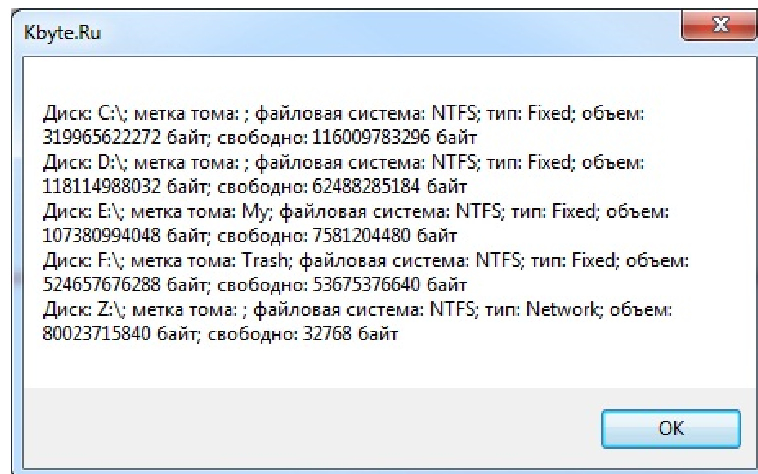


Рис. 24. Вывод списка дисков

Каждый экземпляр устройства **DriveInfo** позволяет получить дополнительную информацию о нем. Можно создать экземпляр класса для любого доступного диска в системе.

```
DriveInfo d = new DriveInfo("C:");
```

Прежде чем начинать работать с каким-либо диском, необходимо проверить его доступность. Для этого предназначено свойство **IsReady**.

```
DriveInfo d = new DriveInfo("C:");
if(d.IsReady){
     MessageBox.Show("Диск готов к работе!");
}
else{
     MessageBox.Show("Диск недоступен!");
}
```

Если у вас в системе есть сетевые диски, получение информации о них может занять значительное время, на протяжении которого ваша программа может быть недоступна для пользователя (подвиснет).

Если устройство не готово к работе (например, у вас есть **CD-ROM**, но в нем нет диска), при попытке получить детальную информацию о диске произойдет исключение.

Класс **DriveInfo** очень простой. Букву диска можно получить в свойстве **Name**. Свойство **VolumeLabel** содержит метку тома. Фактический объем диска, и оставшееся свободное пространство находятся в свойствах **TotalSize** и **AvailableFreeSpace**. Тип устройства можно узнать в свойстве **DriveType**. Получить информацию о файловой системе можно в свойстве **DriveFormat**.

Букву диска (**Name**) и его тип (**DriveType**) можно получить, даже если диск не готов к работе. Например:

```
StringBuilder driveList = new String-
Builder();
foreach(DriveInfo d inDriveIn-
fo.GetDrives())
{
driveList.AppendLine(
String.Format("Диск: {0}; тип: {1}; ",
d.Name, d.DriveType));
}
MessageBox.Show(driveList.ToString());
```

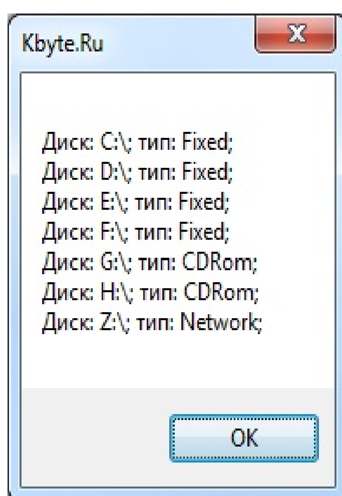


Рис. 25. Список типов устройств

Результат работы программы приведен на рис. 25.

Отформатировать диск при помощи класса **DriveInfo** не получится. Этот класс предоставляет только информацию о конкретном устройстве.

Каталоги

Работа с каталогами в **.NET Framework** осуществляется при помощи двух классов. Основной класс – **DirectoryInfo**, он не имеет статичных методов и предназначен для работы с конкретными каталогами. Второй класс – **Directory**, является вспомогательным и содержит часто используемые статичные методы, функционал которых, по сути, реализован через класс **DirectoryInfo**.

Класс **DirectoryInfo** при инициализации принимает путь к каталогу, который он реализует. Причем можно указывать путь как к реально существующему каталогу, так и к каталогу, который еще не создан.

```
DirectoryInfo d = newDirectoryInfo("C:\\Новаяпапка");
```

Файлы

Для работы с файлами предназначены классы **FileInfo** и **File**. Класс **FileInfo** – основной, у него нет статичных методов, в отличие от вспомогательного класса **File**, который кроме таковых ничего больше не имеет.

Класс **FileInfo** очень похож на класс **DirectoryInfo**, более того у обоих классов общий родитель – **FileSystemInfo**. Он также имеет свойство **Exists**, для проверки существования файла. У **FileInfo** есть метод **MoveTo**, для перемещения файла, и метод **Delete** для удаления файла, и даже метод **Create** для создания файла, хотя последний в реальных условиях используется редко. Установка прав доступа к файлу осуществляется методом **SetAccessControl**. Получить список разрешений можно методом **GetAccessControl**.

Но помимо этого, у класса **FileInfo** есть свойство, содержащее размер файла в байтах – **Length**, а также свойство **IsReadOnly**, которое указывает на возможность записи данных в файл. Из других особенностей, **FileInfo** имеет ссылку на директорию, в которой находится файл – **Directory**, ссылка представляет из себя экземпляр класса

DirectoryInfo. Для удобства получения имени директории, существует вспомогательное свойство **DirectoryName**.

Для записи и чтения данных, класс **FileInfo** содержит функции: **Open**, **OpenRead**, **OpenText** и **OpenWrite**. Большинство функций возвращают **FileStream** – поток записи или чтения. **FileStream** позволяет записывать и считывать данные из/в файла по байтам.

```
FileStream fs = f.Open(FileMode.OpenOrCreate, FileAc-
cess.Write, FileShare.Read);
byte[] data = Encoding.UTF8.GetBytes("Kbyte.Ru");
fs.Write(data, 0, data.Length);
fs.Close();
```

Работа с бинарными файлами

Работа с бинарными файлами практически не отличается от работы с текстовыми, за тем лишь исключением, что доступ к бинарным файлам осуществляется побайтно, ну, и естественно, используются для этого отдельные классы.

Для записи бинарных данных предназначен класс **BinaryWriter**. Этот класс содержит всего один основной метод – **Write**, который позволяет записать в файл указанные данные. В качестве данных может выступать все что угодно, в самом файле данные будут записаны в виде массива байт. Этот метод принципиально такой же, как и его собрат в классе **FileStream**, хотя и немного проще в использовании, не зря же он был сделан.

```
FileInfo f = new FileInfo(@"C:\Новаяпапка\test.dat");
using(FileStream fs = f.Open(FileMode.OpenOrCreate,
FileAccess.Write, FileShare.Read))
{
    using(BinaryWriter bw = new BinaryWriter(fs))
    {
        bw.Write(newbyte[] { 1, 2, 3, 4, 5, 6, 7, 8,
    9, 10 });
    }
}
```

В вышеуказанном примере в файл записываются десять байт от 1 до 10. Не путать с числами. Это именно байты. Если открыть полу-

ченный файл в текстовом редакторе, то вы скорей всего в нем ничего не увидите (в лучшем случае - абракадабру), т.к. байты от 1 до 10 не являются печатными символами.

Для считывания бинарных данных существует класс **BinaryReader**. У этого класса есть множество методов **Read**, предназначенных для считывания разных типов данных: **ReadBoolean**, **ReadByte**, **ReadBytes**, **ReadChar**, **ReadChars**, **ReadDecimal**, **ReadDouble**, **ReadInt16**, **ReadInt32**, **ReadInt64**, **ReadSByte**, **ReadSingle**, **ReadString**, **ReadUInt16**, **ReadUInt32** и **ReadUInt64**. Но основной метод – это **Read**. Он похож на аналогичный метод класса **StreamReader**, за тем лишь исключением, что в качестве буфера используются не массив символов (**char**), а массив байт (**byte**).

```
FileInfo f = new FileInfo(@"C:\Новаяпапка\test.dat");
using(FileStream fs = f.Open(FileMode.Open, FileAc-
cess.Read, FileShare.ReadWrite))
{
using(BinaryReader br = new BinaryReader(fs))
{
int bytesRead=0;
byte[] buffer = newbyte[255];
StringBuilder result = new StringBuilder();
while((bytesRead=br.Read(buffer, 0, buffer.Length))
!=0)
{
for(int i = 0; i <= bytesRead - 1; i++)
{
result.AppendFormat("{0:x2} ", buffer[i]);
}
Array.Clear(buffer, 0, buffer.Length);
}
MessageBox.Show(result.ToString());
}
}
```

Задание к работе

Все задания выполняются каждым студентом подгруппы.

1) По нажатию кнопки запустить диалог выбора файла. Из выбранного текстового файла извлекается массив строк, сделать разде-

ление построчно и каждую строку на слова. Вывести данный массив в DataGridView. По нажатию второй кнопки сохранить измененные данные из таблицы в текстовый файл в таком же формате, как и исходный.

2) По нажатию кнопки запустить диалог выбора файла. В выбранный файл сохраняется структура или объект с данными введенными пользователем. По второй кнопке данные вывести в те же поля.

3) Написать программу-блокнот, позволяющую открывать, редактировать и сохранять текстовые документы. Интерфейс программы должен иметь следующий вид (рис. 26).

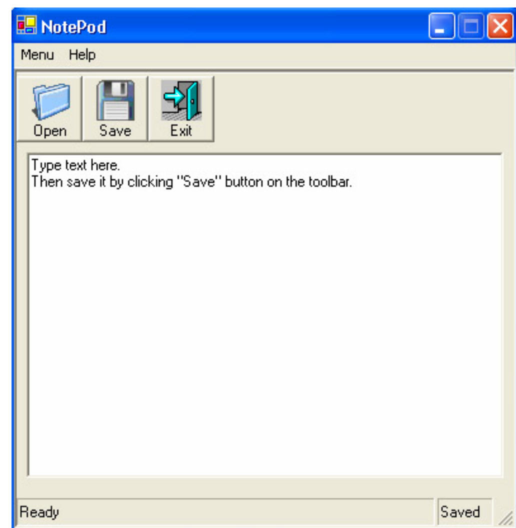


Рис. 26. Интерфейс для программы "NotePod"

Меню "File" предполагает возможность извлечения текста для редактирования из файла, сохранения результатов в файл, выход из программы (рис. 28). Меню "Help" содержит подменю "About" (рис. 27), при нажатии на которое в отдельном окне отображается информация о разработчике приложения.

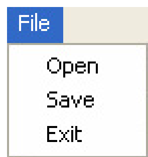


Рис. 28. Меню "File"

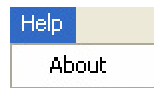


Рис. 27. Меню "Help"

Строка состояния содержит две панели:

1. Правая панель – состояние документа "Сохранен / Изменен" (текст на панели = "Saved", если документ только что открыли или сохранили, либо "Modified", если содержимое документа изменено).

2. Левая панель – состояние блокнота (если пользователь сохраняет документ, текст панели = "Saving…", открывает = "Opening…", программа находится в режиме ожидания = "Ready").

Для открытия и сохранения файлов использовать диалоговые окна OpenFileDialog и SaveFileDialog.

Контрольные вопросы и задания

1. Какое пространство имен требуется подключить для работы с файлами.
2. Можно ли получить информацию о дисках в компьютере? Каким образом?
3. Классы пространства System.IO.
4. В чем отличия работы с бинарными и текстовыми файлами?
5. Как получить путь до файла, с которым осуществляется работа?

ЛАБОРАТОРНАЯ РАБОТА 8. РАБОТА С СЕТЬЮ

Цель работы: *изучить принципы создания клиент-серверных приложений.*

Краткая теоретическая часть

Понятие сокета

Если требуется получить доступ к сетевым операциям низкого уровня, в программе следует использовать класс **Socket**. Протокол TCP/IP основывается на соединениях, устанавливаемых между двумя компьютерами, обычно называемых клиентом и сервером.

Сокеты (socket) – это описатель сетевого соединения с другим приложением. Сокет TCP использует протокол TCP, наследуя все свойства этого протокола. Для создания сокета TCP требуется:

- ✓ IP-адрес локальной машины;
- ✓ Номер порта TCP, который использует приложение на локальной машине;
- ✓ IP-адрес машины, с которой устанавливается связь;
- ✓ Номер порта TCP, на который отзывается приложение, ожидающее установления связи.
- ✓ Сокеты в C#

Язык C# поддерживает два типа сетевых соединений:

- ✓ серверные, реализуемые объектом класса **TcpListener**;
- ✓ клиентские, реализуемые с помощью объектов класса **TcpClient**.

Программа – сервер выполняет следующие шаги:

- ✓ Ожидание подключения клиента;
- ✓ Установка соединения с клиентом;
- ✓ Отправка/получение от клиента сообщения;
- ✓ Разрыв соединения и завершение программы.

Необходимо понимать, что серверные приложения запускаются на локальном компьютере. В качестве параметра конструктор класса TcpListener принимается номер порта, который впоследствии и будет прослушиваться программой.

Однако объект класса **TcpListener** позволяет только прослушивать определенный порт компьютера. Любые процессы передачи данных через этот сокет осуществляются с использованием объекта **TcpClient**. Объект **TcpClient** возвращается методом **AcceptTcpClient** класса **TcpListener**, что обеспечивает сам процесс прослушивания порта.

При создании серверного приложения следует подключить модули:

```
using System.Net;
using System.Net.Sockets;
```

Затем использовать класс **TcpListener**. Объекты этого класса связываются с заданным портом.

```
Int32 port = 13000;
IPAddress localAddr= IPAddress.Parse("127.0.0.1");
TcpListener server= new TcpListener(localAddr, port);
server.Start();
```

После связывания программы с портом с помощью вызова метода **AcceptTcpClient()** можно подключиться к этому порту и начать прослушивание входящих соединений:

```
TcpClient client = server.AcceptTcpClient();
```

После подключения создается поток входных/выходных сообщений:

```
NetworkStream stream = client.GetStream();
```

Чтение сообщений

Через сокет передается массив байт, который после получения нужно преобразовать в строку.

```
Byte[] bytes = new Byte[256];
String data = null;
int i = stream.Read(bytes, 0, bytes.Length);
data=System.Text.Encoding.UTF8.
GetString(bytes, 0, i);
```

Отправка сообщений

Перед отправкой через сокет строку необходимо преобразовать в массив байт.

```
Byte[] bytes = new Byte[256];
String data = "text";
bytes =System.Text.Encoding.UTF.
GetBytes(data);
stream.Write(bytes, 0, bytes.Length);
```

Фрагмент программы-сервера

```
try
{
Int32 port = 13200;
IPAddress localAddr= IPAddress.Parse("127.0.0.1");
//создание и запуск сервера
TcpListener server= new TcpListener(localAddr, port);
    server.Start();
    Byte[] bytes = new Byte[256];
    String data = null;
while(true)
    {
//прослушивание порта и ожидание подключения
    TcpClient client = server.AcceptTcpClient();
    data = null;
//произошло подключение, создается поток сообщений
    NetworkStream stream = client.GetStream();

// чтение сообщения
    int i = stream.Read(bytes, 0, bytes.Length);
    data = System.Text.Encoding.UTF8.GetString(bytes,
0, i);
```

```
//преобразование данных
     data = data.ToUpper();
//отправка сообщения
     byte[] msg = System.Text.Encoding.UTF8.GetBytes(
data);
     stream.Write(msg, 0, msg.Length);

//завершение работы с потоком сообщений
client.Close();
     }//end while
}// end try
catch(SocketException ex)
{
     // ошибки соединения
}
```

Программа-клиент выполняет следующие действия:

- ✓ Подключение к серверу.
- ✓ Отправка/получение сообщения от сервера.
- ✓ Отображение полученного сообщения пользователю.
- ✓ Разрыв соединения с сервером.

Для подключения к серверу используется конструктор, параметрами которого служат IP-адрес серверной программы и порт, который сервер прослушивает:

```
TcpClient client = new TcpClient(server, port);
```

Затем, после подключения к серверу, аналогично серверной программе, создается поток входных/выходных сообщений:

```
NetworkStream stream = client.GetStream();
```

Можно отправить сообщение серверу, предварительно преобразовав строку в массив байт:

```
Byte[] bytes = System.Text.Encoding.UTF8.GetBytes(
message);
stream.Write(bytes, 0, bytes.Length);
```

Или получить сообщение от сервера, преобразовав массив байт в строку:

```
bytes = new Byte[256];
String responseData = String.Empty;
```

```
Int32 i = stream.Read(bytes, 0, bytes.Length);
responseData = System.Text.Encoding.ASCII.GetString(
bytes, 0, i);
```

Фрагмент программы-клиента

```
void Connect(String server, String message, Int32
port)
{
try
     {
// Подключение к серверу
     TcpClient client = new TcpClient(server, port);
// Создание потока сообщений
     NetworkStream stream = client.GetStream();
// Отправка сообщения
Byte[] bytes = System.Text.Encoding.ASCII.GetBytes(
message);
     stream.Write(bytes, 0, bytes.Length);
bytes = new Byte[256];
String responseData = String.Empty;
// получение сообщения
Int32 i = stream.Read(bytes, 0, bytes.Length);
responseData = System.Text.Encoding.UTF8.GetString(
bytes, 0, i);
client.Close();
     }
     catch (Exception e)
     {
// ошибка соединения
     }
}// end connection
```

Пример соединения с программой-сервером.

```
Connect("localhost","text",13000);
Connect("192.168.112.1","text",13200);
```

Задания к работе

Разработать программу интерактивной переписки клиента с сервером (чат). Сервер ожидает запрос клиента на соединение. После соединения клиентского приложения с серверным последнее отправляет

клиенту массив байтов, извещающий его об успешном соединении. Затем программа-клиент выдает пользователю соответствующее сообщение. Клиентское и серверное приложения содержат текстовые поля, в которых пользователи вводят сообщения и отправляют их в другое приложение. Когда клиент или сервер отправляют сообщение «Terminate», соединение между ними прерывается. После этого сервер ожидает подключения другого клиента.

Контрольные вопросы и задания

1. Какие пространства имен нужны для создания клиент-серверного приложения?
2. Поясните принцип работы клиент-серверного приложения.
3. Какие функции выполняет сервер?
4. Какие функции выполняет клиент?
5. С помощью каких объектов можно реализовать сервера и клиента?
6. Как можно разорвать соединение между клиентом и сервером?

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. *Кауфман, В. Ш.* Языки программирования. Концепции и принципы / В.Ш. Кауфман. – М. : РиС, 1993. – 432 с.

2. *Трей Нэш*. C# 2010. Ускоренный курс для профессионалов. – СПб. : Вильямс, 2010. – 592 с. – ISBN 978-5-8459-1638-9, 978-1-43-022537-9.

3. *Герберт Шилдт*. C# 3.0. Полное руководство / Г. Шилдт. - СПб. : Вильямс, 2011. – 1056 с. – ISBN: 978-5-8459-1684-6, 0-07-174116-X

4. *Нейгел*, К. C# 4.0 и платформа .NET 4 для профессионалов / К. Нейгел, Б. Ивьен и др. – Изд. Диалектика, 2011. – 1140 с. – ISBN 978-5-8459-1656-3, 978-0-470-50225-9.

5. *Троелсен*, Э. Язык программирования C# 2008 и платформа .NET 3.5 / Э. Троелсен. – СПб. : Вильямс, 2010. – 1344 с. – ISBN 978-5-8459-1589-4, 978-1-59-059884-9.

ОГЛАВЛЕНИЕ

Введение 3
Лабораторная работа 1. Типы данных. Операторы 4
Лабораторная работа 2. Лексический анализ выражений. Формы записи выражений 18
Лабораторная работа 3. Регулярные выражения 22
Лабораторная работа 4. Базовые графические компоненты, свойства и обработка событий 32
Лабораторная работа 5. Работа с графическими компонентами: treeview, datagridview, listbox 47
Лабораторная работа 6. Обработка исключений, использование многопоточности для обеспечения работы интерфейса 51
Лабораторная работа 7. Файлы 59
Лабораторная работа 8. Работа с сетью 68
Список рекомендуемой литературы 73

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

Составители:
ПАВЛОВА Ольга Николаевна
КАСЬЯНОВ Аркадий Александрович

Ответственный за выпуск – зав. кафедрой профессор С.М. Аракелян

Подписано в печать __.__.13
Формат 60x84/16. Усл. печ. л._,_. Тираж 100 экз.
Заказ
Издательство
Владимирского государственного университета.
600000, Владимир, ул. Горького, 87.