№ 3889

004
Т 384

ТЕХНОЛОГИЯ РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ НА ЯЗЫКЕ C# В СРЕДЕ Visual Studio.Net

Методические указания

НОВОСИБИРСК
2010

Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

004 № 3889
Т 384

ТЕХНОЛОГИЯ РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ НА ЯЗЫКЕ C# В СРЕДЕ Visual Studio.Net

Методические указания к лабораторным работам
по дисциплине «Технология программирования и разработка
программного обеспечения» для студентов II курса АВТФ
(специальность 230101 «Вычислительные машины, комплексы,
системы и сети» по направлению 230100 «Информатика
и вычислительная техника»

НОВОСИБИРСК
2010

УДК 004.434 (076.5)
Т 384

Составитель *Васюткина И.А.*, канд. техн. наук, доцент

Рецензент *Юн С.Г.*, канд. техн. наук, доцент

Работа подготовлена на кафедре вычислительной техники

Оглавление

.

Предисловие

В настоящих методических указаниях дано описание четырех лабораторных работ, посвященных узучению языка C# и технологии разработки объектно-ориентированных программ в среде Visual Studio.Net: особенности программирования на языке C#; разработка иерархии классов и интерфейсов, обработка исключительных ситуаций; проектирование графического интерфейса пользователя и разработка SDI и MDI приложений; обработка событий в приложениях; сериализация/десериализация объектов классов; потоковый ввод/вывод данных.

В каждой лабораторной работе изложение теоретического материала сопровождается примерами, поясняющими суть вопроса, также разработан набор заданий для самостоятельного выполнения.

ЛАБОРАТОРНАЯ РАБОТА 1

ВСТРОЕННЫЕ ТИПЫ ДАННЫХ В C#. МАССИВЫ. СТРОКИ. РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Цель работы:

– изучить классификацию типов данных и отличительные особенности синтаксических конструкций языка C# от C++;

– изучить базовые типы: Array, String, StringBuilder, а также средства стандартного ввода/вывода и возможности форматирования вывода;

– получить понятие о регулярных выражениях и их применении для поиска, замены и разбиения текста на синтаксические лексемы.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Проработать примеры программ 1–8, данные в теоретических сведениях. Создать на их основе программы. Получить результаты работы программ и уметь их объяснить. Внести их в отчет по работе с комментариями.

2. Выполнить задания на двумерный массив по указанию преподавателя.

2.1. Точки на плоскости заданы координатами x и y, которые хранятся в двумерном массиве. Найти пару самых удаленных друг от друга точек.

2.2. Найти суммы элементов двухмерного массива целых чисел, расположенных на линиях, параллельных главной диагонали, и ниже ее.

2.3. Найти номер столбца двухмерного массива целых чисел, для которого среднеарифметическое значение его элементов максимально.

2.4. В двухмерном массиве вещественных чисел поменять местами строки и столбцы с одинаковыми номерами.

2.5. В двухмерном массиве целых чисел поменять местами столбцы, симметричные относительно середины массива (вертикальной линии).

2.6. В двухмерном массиве целых чисел поменять местами строки, симметричные относительно середины массива (горизонтальной линии).

2.7. Поменять местами значения элементов двухмерного массива вещественных чисел, симметричных относительно побочной диагонали.

2.8. Найти максимальный элемент среди минимальных элементов строк двухмерного массива целых чисел. Определить номер строки и столбца для такого элемента.

2.9. Найти максимальный среди минимальных элементов столбцов двухмерного массива целых чисел. Определить номер строки и столбца для такого элемента.

2.10. Удалить столбец двухмерного массива вещественных чисел, в котором находится максимальный элемент этого массива.

2.11. Найти все неповторяющиеся элементы двухмерного массива целых чисел.

2.12. Заполнить двухмерный массив целыми числами от 1 до 100 по спирали.

2.13. Определить:

а) сумму элементов главной диагонали массива;

б) сумму элементов побочной диагонали массива;

в) среднее арифметическое элементов главной диагонали массива;

г) среднее арифметическое элементов побочной диагонали массива.

3. Выполнить задания на строки по указанию преподавателя. Использовать в задачах два класса строк: String и StringBuilder.

3.1. Текстовые сообщения часто печатаются строчными буквами, но многие сотовые телефоны имеют встроенные средства преобразования строчной буквы в прописную после символа пунктуации, как точка или знак вопроса. Составить программу, которая будет вводить сообщение в переменную String (на одной строке), а затем обрабатывать его с получением новой строки с прописными буквами в соответствующих местах.

3.2. Составить программу, которая будет вводить строку в переменную String. Подсчитать, сколько различных символов встречается в ней. Вывести их на экран.

3.3. Составить программу, которая будет вводить строку в переменную String. Найти в ней те слова, которые начинаются и оканчиваются одной и той же буквой.

3.4. Составить программу, которая будет вводить строку в переменную String. Определить, сколько раз в строке встречается заданное слово.

3.5. Строка, содержащая произвольный русский текст, состоит не более чем из 200 символов. Написать, какие буквы и сколько раз встречаются в этом тексте. Ответ должен приводиться в грамматически правильной форме: например: а – 25 раз, к – 3 раза и т. д.

3.6. Двумерный массив *n* x *m* содержит некоторые буквы русского алфавита, расположенные в произвольном порядке. Написать программу, проверяющую, можно ли из этих букв составить данное слово *S*. Каждая буква массива используется не более одного раза.

3.7. Составить программу, которая будет вводить строку в переменную String. Удалить из нее все лишние пробелы, оставив между словами не более одного. Результат поместить в новую строку.

3.8. Составить программу, которая будет вводить строку в переменную String. Напечатать в алфавитном порядке все слова из данной строки, имеющие заданную длину *n*.

3.9. Составить программу, которая будет вводить строку в переменную String. Найти слово, встречающееся в каждом предложении, или сообщить, что такого слова нет.

3.10. Дана строка, содержащая текст на русском языке. В предложениях некоторые из слов записаны подряд несколько раз (предложение заканчивается точкой или знаком восклицания). Получить в новой строке отредактированный текст, в котором удалены подряд идущие вхождения слов в предложениях.

3.11. Даны две строки *A* и *B*. Составьте программу, проверяющую, можно ли из букв, входящих в *A*, составить *B* (буквы можно использовать не более одного раза и можно переставлять). Например, *A*: ИНТЕГРАЛ; *B*: АГЕНТ – составить можно; *B*: ГРАФ – нельзя.

3.12. Дана строка, содержащая текст на русском языке. Заменить все вхождения заданного слова на другое.

3.13. С клавиатуры вводится предложение, слова в котором разделены символом ‘_’. Напечатать все предложения, которые получаются путем перестановки слов исходного текста.

4. Выполнить задание на применение регулярных выражений.

4.1. Задан текст. Определить, входит ли в него заданное слово и сколько раз.

4.2. Задан текст. Определить, является ли он кодом HTML : содержит теги <html>, <form>, <h1>.

4.3. Задан текст. Определить, является ли он текстом на английском языке.

4.4. Задан текст. После каждой буквы «о» вставить сочетание «Ок».

4.5. Задан текст. Определить, является ли он текстом на русском языке.

4.6. Задан текст. Определить, содержит ли он цифры.

4.7. Задан текст. Определить, сколько предложений начинается со слова “Информатика”.

4.8. Задан текст. Выбрать из него все семизначные номера телефонов.

4.9. Задан текст. Определить, содержит ли он цифры.

4.10. Задан текст. Выбрать из него все e-mail адреса.

4.11. Задан текст, содержащий буквы и цифры. Найти произведение всех чисел в тексте.

4.12. Задано предложение. Распечатать все слова в столбик.

4.13. Задан текст. Определить количество согласных букв в нем и распечатать их.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Для создания нового проекта C# заходим в меню File и далее выбираем New, Project. В появившемся окне New Project слева выбираем Visual C#, а справа тип приложения – Console Application. Дайте имя проекту – ConsoleHello, укажите, где будет храниться проект.

Главное окно редактора кода, в котором отображается программный код, хранимый в файле ConcoleHello.cs, изображен на рис. 1.1. Ниже главного окна расположены окна вывода (Error List, Output), в которых выводится вся служебная информация.

В правой части окна находится Solution Explorer, где показывается список файлов, содержащийся в "решении", которое, может состоять из нескольких "проектов". Вкладки вверху главного окна позволяют легко перемещаться от одного открытого файла к другому.

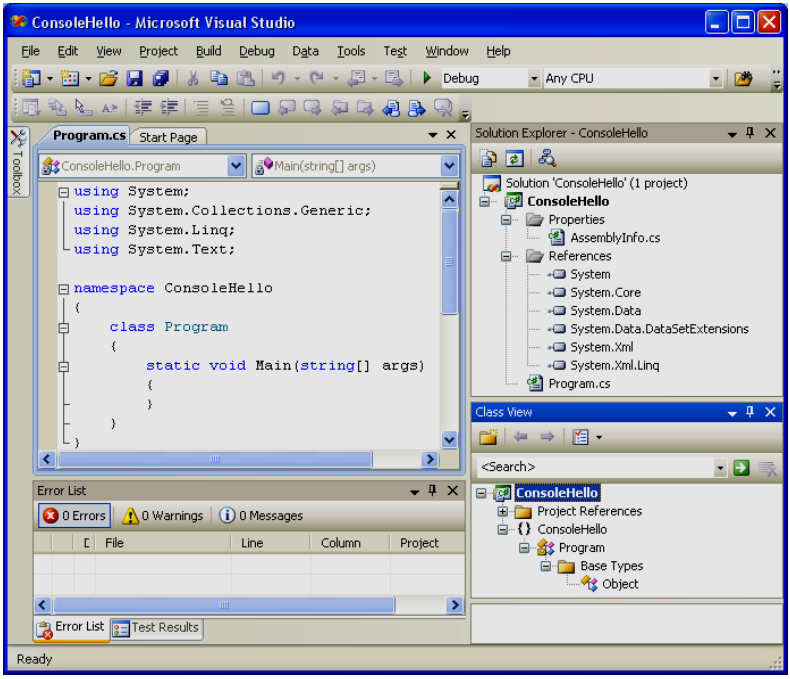


Рис. 1.1. Консольный проект в Visual Studio NET

Добавим в проект код:

```
namespace ConsoleHello  {       // создаваемое пространство имен
   class Program    {           // имя класса по умолчанию
      static void Main(string[] args)      {
         // вывод строки на экран
Console.WriteLine("Введите Ваше имя");
         string name;
name = Console.ReadLine();      // ввод строки с клавиатуры
         if (name == "")  Console.WriteLine("Здравствуй, мир!");
         else   Console.WriteLine("Здравствуй, " + name + "!");
      }
   } }
```

ТИПЫ ДАННЫХ C#

C# является типизированным языком. Необходимо всегда объявлять тип каждого объекта, который создаете.

C# подразделяет типы на два вида: ***встроенные типы***, которые определены в языке, и ***определяемые пользователем типы***, которые создает программист. Также все типы разделяются на две основные разновидности: ***размерные (структурные) типы (value-based)*** и ***ссылочные типы (reference-based***). К структурным типам относятся все числовые типы данных (int, float и др.), а также перечисления и структуры. К ссылочным типам относятся массивы, строки и классы.

Встроенные типы

Для каждого встроенного типа существует соответствующий тип в CRL (Common Language Runtime). Это означает, что каждый тип имеет два названия – **полный** (из CLR) и **сокращенный**, используемый в C# (см. табл. 1.1).

Таблица 1.1

Имена встроенных типов

Логический тип			
Имя типа	**Системный тип**	**Значения**	**Размер**
bool	System.Boolean	true, false	8 бит
Арифметические целочисленные типы			
Имя типа	**Системный тип**	**Диапазон**	**Размер**
sbyte	System.SByte	–128 … – 128	Знаковое, 8 бит
byte	System.Byte	0 – 255	Беззнаковое, 8 бит
short	System.Short	–32768 … – 32767	Знаковое, 16 бит
ushort	System.UShort	0 – 65535	Беззнаковое, 16 бит
int	System.Int32	$\approx(-2 \cdot 10^9 \ldots -2 \cdot 10^9)$	Знаковое, 32 бит

Окончание табл. 1.1

Имя типа	**Системный тип**	**Диапазон**	**Размер**
uint	System.UInt32	≈(0 … $-4 \cdot 10^9$)	Беззнаковое, 32 бит
long	System.Int64	≈($-9 \cdot 10^{18}$ … $9 \cdot 10^{18}$)	Знаковое, 64 бит
ulong	System.UInt64	≈(0 … $18 \cdot 10^{18}$)	Беззнаковое, 64 бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
float	System.Single	$-1.5 \cdot 10^{-45}$ … $+3.4 \cdot 10^{38}$	7 цифр
double	System.Double	$-5.0 \cdot 10^{-324}$ … $+1.7 \cdot \cdot 10^{308}$	15–16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
decimal	System.Decimal	$-1.0 \cdot 10^{-28}$ … $+7.9 \cdot 10^{28}$	28–29 значащих цифр
Символьные типы			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
char	System.Char	U+0000 – U+ffff	Unicode символ
string	System.String	Строка из символов Unicode	
Объектный тип			
Имя типа	**Системный тип**	**Примечание**	
Object	System.Object	Прародитель всех типов	

В языке C# сглажено различие между типом и классом. Все типы одновременно являются классами, связанными отношением наследования. Родительским, базовым, классом является класс **Object**. Все остальные типы являются его потомками, наследуя методы этого класса. У класса Object есть четыре наследуемых метода:

– bool Equals(object obj) – проверяет эквивалентность текущего объекта и объекта, переданного в качестве аргумента;

– System.Type GetType() – возвращает системный тип текущего объекта;

– string ToString() – возвращает строку, связанную с объектом. Для арифметических типов возвращается значение, преобразованное в строку;

– int GetHashCode() – служит как хэш-функция в соответствующих алгоритмах поиска по ключу при хранении данных в хэш-таблицах.

Естественно, что все встроенные типы нужным образом переопределяют методы родителя и добавляют собственные методы и свойства.

```
//определение целой переменной встроенного типа
int x=11, v = new Int32();
v = 007;
//определение строковой переменной
string s1 = "Agent";
```

Преобразования типов. Преобразования в строковый тип всегда определены, поскольку все типы наследуют метод ToString(). Метод можно вызывать явно или он вызывается неявно, когда по контексту требуется преобразование к строковому типу:

```
public void ToStringTest(){
  string s ="Владимир Петров ", s1 =" Возраст: ";
  int ux = 27;
  s = s + s1 + ux.ToString();    Console.WriteLine (s);
  s1 =" Зарплата: ";
  float dy = (float)2700.50;
  s = s + s1 + dy;          Console.WriteLine (s);  }
```

Преобразования строк в число. Класс Convert пространства имен System обеспечивает необходимые преобразования между различными типами. Класс содержит 15 статических методов вида

To<Type> (ToBoolean(), ...ToUInt64()). Все методы многократно перегружены.

```
public void FromStringTest() {
    s ="Введите возраст ";
    Console.WriteLine(s); s1 = Console.ReadLine();
    ux = Convert.ToUInt32(s1); Console.WriteLine("Возраст: "+ ux);
    Console.WriteLine("Введите зарплату ");
    dy = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Зарплата: "+ dy);
}
```

Данные, читаемые с консоли методом ReadLine или Read, всегда представляют собой строку, которую затем необходимо преобразовать в нужный тип.

Ссылочные типы

Массивы в C#

Массивом называют упорядоченную совокупность элементов одного типа. Число индексов характеризует размерность массива. При объявлении массива границы задаются выражениями. Если все границы заданы константами, то такие массивы называются **статическими**. Если же выражения, задающие границы, зависят от переменных, то такие массивы называются **динамическими**, память им отводится в процессе выполнения программы.

В языке C# имеются одномерные массивы, массивы массивов и многомерные ступенчатые массивы.

Определение одномерных массивов:

```
int[] k;                    //k – одномерный массив
k=new int [3];              //Определяем массив из трех целых
k[0]=-5; k[1]=4; k[2]=55;   //Задаем элементы массива
```

Элементы массива можно задавать сразу при объявлении:

```
int[] k = {-5, 4, 55};
```

Создание динамического массива:

```
Console.WriteLine("Введите число элементов массива ");
int size = Int32.Parse(Console.ReadLine());
int[] A1 = new int[size];      //создание динамического массива
```

Определение многомерных массивов:

```
int[,] k = new int [2,3];
```

Обратите внимание, что пара квадратных скобок только одна.

Аналогично можно задавать многомерные массивы. Вот пример трехмерного массива:

```
int[,,] k = new int [10,10,10];
```

Многомерные массивы можно сразу инициализировать:

```
int[,] k = {{2,-2},{3,-22},{0,4}};
```

Определение ступенчатых массивов:

```
int[][] k = new int [2][]; //Объявляем второй ступенчатый массив
k[0]=new int[3];     //Определяем нулевой элемент
k[1]=new int[4];     //Определяем первый элемент
k[1][3]=22;          //записываем 22 в последний элемент
```

Обратите внимание, что у ступенчатых массивов задается несколько пар квадратных скобок (столько, сколько размерностей у массива), табл. 1.2.

Таблица 1.2

Создание ступенчатых массивов

Объявление и инициализация значениями	Объявление и инициализация нулевыми значениями	Объявление и инициализация нулевыми значениями

int[][] jagger = new int[3][] {
new int[] {5,7,9,11},
new int[] {2,8},
new int[] {6,12,4}
};
	int[][] jagger1 = new int[3][] {
	new int[4],
	new int[2],
	new int[3]
	};
		int[][] jagger2 =
		{
		new int[4],
		new int[2],
		new int[3]
		};

Массив имеет два уровня. Можно считать, что у него три элемента, каждый из которых является массивом. Для каждого внутреннего массива необходимо вызвать конструктор new.

Базовый класс System.Array

Все классы-массивы являются потомками класса Array из библиотеки FCL. Класс имеет большое число методов и свойств (табл. 1.3, 1,4). Благодаря такому родителю над массивами определены самые разнообразные операции – копирование, поиск, обращение, сортировка, получение различных характеристик. Массивы можно рассматривать как коллекции и устраивать циклы **foreach** для перебора всех элементов.

Таблица 1.3

Свойства класса Array

Свойство	Родитель	Описание
IsFixedSize	Интерфейс IList	True, если массив статический
Length		Число элементов массива
Rank		Размерность массива

Таблица 1.4

Статические методы класса Array

Метод	Описание
BinarySearch()	Двоичный поиск
Clear()	Выполняет начальную инициализацию элементов в зависимости от типа: 0 – для арифметического типа, *false* – для логического типа, *null* – для ссылок, "" – для строк
CopyTo()	Копирование части или всего массива в другой массив. Описание и примеры даны в тексте
GetLength()	Используется для определения количества элементов в указанном измерении массива
IndexOf()	Индекс первого вхождения образца в массив. Описание и примеры даны в тексте
LastIndexOf()	Индекс последнего вхождения образца в массив. Описание и примеры даны в тексте
Reverse()	Обращение одномерного массива

Окончание табл. 1.4

Метод	Описание
Sort()	Сортировка одномерного массива встроенных типов данных
GetValue() **SetValue()**	Возвращает или устанавливает значение указанного индекса для массива

Программа 1. Применение методов класса Array

```
public static int Main(string[] args) {
      string[]firstNames={"Саша", "Маша", "Олег", "Света", "Игорь"};
            Console.WriteLine("Here is the array:");
            for(int i=0; i< firstNames.Length; i++)
                  Console.WriteLine(firstNames[i]+"\t");
            Console.WriteLine("\n");
            Array.Reverse(firstNames);
            for(int i=0; i< firstNames.Length; i++)
                  Console.WriteLine(firstNames[i]+"\t");
            Console.WriteLine("\n");
            Console.WriteLine("Cleared out all but one…");
            Array.Clear(firstNames,1,4);
            for(int i=0; i< firstNames.Length; i++)
                  Console.WriteLine(firstNames[i]+"\t\n");
            return 0;
}
```

В процедуре PrintAr формальный аргумент класса Array, следовательно, можно передавать массив любого класса в качестве фактического аргумента

Программа 2. Применение методов класса Array

```
public static void PrintAr(string name, Array A) {
  Console.WriteLine(name);
  switch (A.Rank)   {
    case 1:
      for(int i = 0; i<A.GetLength(0);i++)
        Console.Write("\t" + name + "[{0}]={1}", i, A.GetValue(i));
      Console.WriteLine();
      break;
```

```
    case 2:
      for(int i = 0; i<A.GetLength(0);i++)        {
        for(int j = 0; j<A.GetLength(1);j++)
          Console.Write("\t" + name + "[{0},{1}]={2}", i,j,
A.GetValue(i,j));
        Console.WriteLine();
      }
      break;
    default: break;
  }
}
```

Строки в C#

Класс Char. Использует двухбайтную кодировку символов Unicode. Константу можно задавать:

- символом, заключенным в одинарные кавычки;
- escape-последовательностью, задающей код символа;
- Unicode-последовательностью.

```
char ch1='A', ch2 ='\x5A', ch3='\u0058';   char ch = new Char();
int code;        string s;
ch = ch1;
//преобразование символьного типа в тип int
code = ch;  ch1=(char) (code +1) ;
//преобразование символьного типа в строку
s = ch1.ToString()+ch2.ToString()+ch3.ToString() ;
```

Класс Char имеет большое число методов (см. табл.1.5).

Таблица 1.5

Статические методы и свойства класса Char

Метод	Описание
GetNumericValue	Возвращает численное значение символа, если он является цифрой, и (–1) в противном случае
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой

Окончание табл. 1.5

Метод	Описание
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичной или шестнадцатеричной цифрой)
IsUpper	Возвращает true, если символ задан в верхнем регистре
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру

Класс Char[] – массив символов. Можно использовать для представления строк постоянной длины. Массив char[] – это обычный массив. Он не задает строку, заканчивающуюся нулем. В C# не определены взаимные преобразования между классами String и Char[]. Для этого можно применить метод ToCharArray() класса String или посимвольно передать содержимое переменной string в массив символов:

Программа 3. Массивы символов Char[]

```
string CharArrayToString(char[] ar) {
  string result="";
  for(int i = 0; i< ar.Length; i++) result += ar[i];
  return(result);
}
void PrintCharAr(string name,char[] ar) {
  Console.WriteLine(name);
  for(int i=0; i < ar.Length; i++) Console.Write(ar[i]);
  Console.WriteLine();
}
public void TestCharArAndString() {
  string hello = "Здравствуй, Мир!";
  char[] strM1 = hello.ToCharArray();
  PrintCharAr("strM1",strM1);
  char[] World = new char[3];
  Array.Copy(strM1,12,World,0,3);  //копирование подстроки
  PrintCharAr("World",World);
  Console.WriteLine(CharArrayToString(World)); }
```

Класс Char[] является наследником классов Object и класса Array и обладает всеми методами родительских классов.

Класс String. Является основным типом при работе со строками. Задает строки переменной длины. Над объектами этого класса определен широкий набор операций, соответствующий современному представлению о том, как должен быть устроен строковый тип. Объекты класса String объявляются как все прочие объекты простых типов – с явной или отложенной инициализацией, с явным или неявным вызовом конструктора класса. Чаще всего при объявлении конструктор явно не вызывается, а инициализация задается строковой константой. Но у класса Sring достаточно много конструкторов. Они позволяют сконструировать строку:

- из символа, повторенного заданное число раз;
- массива символов char[];
- части массива символов.

```
string world = "Мир";
string sssss = new string('s',5);
char[] yes = "Yes".ToCharArray();
string stryes = new string(yes);
string strye = new string(yes,0,2);
Console.WriteLine("world = {0}; sssss={1}; stryes={2};"+ " strye= {3}", world, sssss, stryes, strye);
```

Над строками определены следующие операции:

- присваивание (=);
- две операции проверки эквивалентности (= =) и (!=);
- конкатенация или сцепление строк (+);
- взятие индекса ([]).

В результате присваивания создается ссылка на константную строку, хранимую в "куче". Операции, проверяющие эквивалентность, сравнивают значения строк, а не ссылки. Бинарная операция "+" сцепляет две строки, приписывая вторую строку к хвосту первой. Взятие индекса при работе со строками отражает тот факт, что строку можно рассматривать как массив и получать каждый ее символ. Символ строки имеет тип char, доступный только для чтения, но не для записи.

Класс String относится к неизменяемым классам **(**immutable**)**. Ни один из его методов не меняет значения существующих объектов. Методы создают новые значения и возвращают в качестве результата новые строки. Методы класса String описаны в табл. 1.6 и 1.7.

Таблица 1.6

Статические методы и свойства класса String

Метод	Описание
Empty	Возвращается пустая строка. Свойство со статусом read only
Compare	Сравнение двух строк. Реализации метода позволяют сравнивать строки или подстроки, учитывать или не учитывать регистр, особенности национального форматирования дат, чисел и т.д.
CompareOrdinal	Сравнение двух строк. Реализации метода позволяют сравнивать строки и подстроки. Сравниваются коды символов
Concat	Конкатенация строк. Допускает сцепление произвольного числа строк
Copy	Создается копия строки
Format	Выполняет форматирование в соответствии с заданными спецификациями формата
Join	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители

Таблица 1.7

Динамические методы и свойства класса String

Метод	Описание
Insert	Вставляет подстроку в заданную позицию
Remove	Удаляет подстроку в заданной позиции

Окончание табл. 1.7

Метод	Описание
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
ToCharArray	Преобразование строки в массив символов

Строковые константы. В C# существуют два вида строковых констант:

- обычные константы, которые представляют строку символов, заключенную в кавычки;
- ***@-константы***, заданные обычной константой с предшествующим знаком @.

В обычных константах некоторые символы интерпретируются особым образом. Например, управляющие символы, начинающиеся символом "\". В @-константах все символы трактуются в полном соответствии с их изображением. Пример задания констант:

```
s1 = "c:\\c#book\\ch5\\chapter5.doc";
s2 = @"c:\c#book\ch5\chapter5.doc".
```

Класс StringBuilder – построитель строк. Компенсирует недостаток класса String. Класс принадлежит к изменяемым классам и находится в пространстве имен System.Text. Объекты класса объявляются с явным вызовом конструктора класса. Конструктор без параметров создает пустую строку.

public StringBuilder (string str, int cap). Параметр str задает строку инициализации, cap – емкость объекта;

public StringBuilder (int curcap, int maxcap). Параметры curcap и maxcap задают начальную и максимальную емкость объекта;

public StringBuilder (string str, int start, int len, int cap). Параметры str, start, len задают строку инициализации, cap – емкость объекта.

Над строками класса определены операции:

– присваивание (=);

– две операции проверки эквивалентности (= =) и (!=);

– взятие индекса ([]).

Операция конкатенации (+) не определена, ее роль играет метод Append.

Со строкой этого класса можно работать как с массивом, допускается не только чтение отдельного символа, но и его изменение.

Программа 4. Строки класса StringBuilder

```
public void TestStringBuilder(){
  StringBuilder s1 =new StringBuilder("ABC"),
  s2 =new StringBuilder("CDE"),  s3 = new StringBuilder();
  s3= s1.Append(s2);
  bool b1 = (s1==s3);
  char ch1 = s1[0], ch2=s2[0];
  Console.WriteLine("s1={0}, s2={1}, b1={2}," + "ch1={3},
   ch2={4}", s1,s2,b1,ch1,ch2);
  StringBuilder s = new StringBuilder("Zenon");
  s[0]='L';    Console.WriteLine(s);
}
```

Основные методы класса:

public StringBuilder Append (<объект>). К строке, вызвавшей метод, присоединяется строка, полученная в качестве параметра. Метод перегружен и может принимать на входе объекты всех простых типов. В качестве результата возвращается ссылка на объект, вызвавший метод.

public StringBuilder Insert (int location,<объект>). Метод вставляет строку в позицию, указанную параметром location.

public StringBuilder Remove (int start, int len). Метод удаляет подстроку длины len, начинающуюся с позиции start.

public StringBuilder Replace (string str1,string str2). Все вхождения подстроки str1 заменяются на строку str2.

public StringBuilder AppendFormat (<строка форматов>, <объекты>). Метод является комбинацией метода Format класса String и метода Append. Строка форматов, переданная методу, содержит только спецификации форматов. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода Remove, все рассмотренные методы являются перегруженными.

Пространство имен *RegularExpression*

Регулярные выражения – это один из способов поиска подстрок (соответствий) в строках. Осуществляется с помощью просмотра строки в поисках некоторого шаблона (табл. 1.8). Очень эффективны библиотеки, интерпретирующие регулярные выражения, обычно пишутся на низкоуровневых высокопроизводительных языках (C, C++, Assembler). С помощью регулярных выражений выполняются три действия:

• проверка наличия соответствующей шаблону подстроки;

• поиск и выдача пользователю соответствующих шаблону подстрок;

• замена соответствующих шаблону подстрок.

Синтаксис регулярных выражений. Регулярное выражение на C# задается строковой константой. Обычно используется @-конс-танта. В C# работа с регулярными выражениями выглядит следующим образом:

```
Regex re = new Regex(«образец», «опции»);
MatchCollection me = re.Matches(“строка для поиска”);
iCountMatchs = me.Count,
```

где re – это объект типа Regex. В конструкторе ему передается образец поиска и опции.

Таблица 1.8

Символы описания шаблона

Символ	Интерпретация
	Категория: подмножества (классы) символов
.	Соответствует любому символу, за исключением символа конца строки
[aeiou]	Соответствует любому символу из множества, заданного в квадратных скобках
[^aeiou]	Отрицание. Соответствует любому символу, за исключением символов, заданных в квадратных скобках
[0-9a-fA-F]	Задание диапазона символов, упорядоченных по коду. Так, 0-9 задает любую цифру
\w	Множество символов, используемых при задании идентификаторов – большие и малые символы латиницы, цифры и знак подчеркивания
\s	Соответствует символам белого пробела
\d	Соответствует любому символу из множества цифр
	Категория: Операции (модификаторы)
*****	Итерация. Задает ноль или более соответствий; например, \w* или (abc)*. Аналогично {0,}
+	Положительная итерация. Задает одно или более соответствий; например, \w+ или (abc)+. Аналогично {1,}
?	Задает ноль или одно соответствие; например, \w? Или (abc)? Аналогично {0,1}
{n}	Задает в точности *n* соответствий; например, \w{2}
{n,}	Задает, по меньшей мере *n* соответствий; например, (abc){2,}
	Категория: Группирование
(?<Name>)	При обнаружении соответствия выражению, заданному в круглых скобках, создается именованная группа, которой дается имя *Name*
()	Круглые скобки разбивают регулярное выражение на группы. Для каждого подвыражения, заключенного в круглые скобки, создается группа, автоматически получающая номер

Класс Regex. Это основной класс, объекты которого определяют регулярные выражения. В конструктор класса передается в качестве

параметра строка, задающая регулярное выражение. Основные методы класса Regex:

- метод **Match** запускает поиск первого соответствия. Параметром передается строка поиска. Метод возвращает объект класса Match, описывающий результат поиска.

Программа 5. Поиск первого соответствия шаблону

```
string FindMatch(string str, string strpat){
  Regex pat = new Regex(strpat);
  Match match =pat.Match(str);
  string found = "";
  if (match.Success)   {
    found =match.Value;
    Console.WriteLine("Строка ={0}\tОбразец={1}\t
      Найдено={2}", str,strpat,found);
  }
  return(found);
}
public void TestSinglePat(){
  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); //art
  str ="fab77cd efg";
  found = FindMatch(str,strpat); //ab77cd
 //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); //adddbdf
}
```

- метод **Matches** позволяет разыскать все непересекающиеся вхождения подстрок, удовлетворяющие образцу. В качестве результата возвращается объект MatchCollection, представляющий коллекцию объектов Match.

Программа 6. Поиск всех соответствий шаблону

```
void FindMatches(string str, string strpat) {
        Regex pat = new Regex(strpat);
        MatchCollection match =pat.Matches(str);
        Console.WriteLine("Строка ={0}\tОбразец={1}\t
        Найдено={2}", str,strpat,match.Count);
 }
Console.WriteLine("око и рококо");
strpat="око"; str = "рококо";
FindMatches(str, strpat);          //найдено одно соответствие
```

- метод **NextMatch** запускает новый поиск.
- метод **Split** является обобщением метода Split класса String. Он позволяет, используя образец, разделить искомую строку на элементы.

```
 static void Main() {
    string si = "Один, Два, Три, Строка для разбора";
      Regex theRegex = new Regex(" |, |,");
      int id = 1;
      foreach (string substring in theRegex.Split(si))
            Console.WriteLine("{0}: {1}", id++, substring);
}
```

- метод **Replace –** позволяет делать замену найденного образца. Метод перегружен. При вызове метода передаются две строки: первая задает строку, в которой необходимо произвести замену, а вторая – на что нужно заменить найденную подстроку.

```
Regex r = new Regex(@"(a+)");
    string s="bacghghaaab";
    s=r.Replace(s,"_$1_");        // $1 – соответствует группе (a+)
    Console.WriteLine("{0}",s);
```

Третий параметр указывает, сколько замен нужно произвести:

```
Regex r = new Regex(@"(dotsite)");
string s="dotsitedotsitedotsiterulez";
s=r.Replace(s,"f",1);   Console.WriteLine("{0}",s);
```

Четвертый параметр указывает, с какого вхождения производить замены:

```
Regex r = new Regex(@"(dotsite)");
string s="dotsitedotsitedotsiterulez";
s=r.Replace(s,"f",2,1); Console.WriteLine("{0}",s);
```

Классы Match и MatchCollection. Коллекция MatchCollection, позволяет получить доступ к каждому ее элементу – объекту Match. Для этого можно использовать цикл foreach.

При работе с объектами класса Match наибольший интерес представляют свойства класса. Рассмотрим основные свойства:

- свойства Index, Length и Value наследованы от прародителя Capture. Они описывают найденную подстроку – индекс начала подстроки в искомой строке, длину подстроки и ее значение;
- свойство Groups класса Match возвращает коллекцию групп – объект GroupCollection, который позволяет работать с группами, созданными в процессе поиска соответствия;
- свойство Captures, наследованное от объекта Group, возвращает коллекцию CaptureCollection.

Программа 7. Поиск всех образцов, соответствующих регулярному выражению

```
  public static void Main( ) {
    string si = "Это строка для поиска";
// найти любой пробельный символ следующий за непробельным
    Regex theReg = new Regex(@"(\S+)\s");
    // получить коллекцию результата поиска
    MatchCollection theMatches = theReg.Matches (si);
     // перебор всей коллекции
    foreach (Match theMatch in theMatches) {
        Console.WriteLine( "theMatch.Length: {0}",
theMatch.Length);
        if (theMatch.Length != 0)
          Console.WriteLine("theMatch: {0}", theMatch.ToString( ));
    } }
```

Классы Group и GroupCollection. Коллекция GroupCollection возвращается при вызове свойства Group объекта Match. Имея эту коллекцию, можно добраться до каждого объекта Group.

Свойства создаваемых групп:

- при обнаружении одной подстроки, удовлетворяющей условию поиска, создается не одна группа, а коллекция групп;
- группа с индексом 0 содержит информацию о найденном соответствии;
- число групп в коллекции зависит от числа круглых скобок в записи регулярного выражения. Каждая пара круглых скобок создает дополнительную группу;
- группы могут быть индексированы, но могут быть и именованными, в круглых скобках разрешается указывать имя группы.

Создание именованных групп крайне полезно при разборе строк, содержащих разнородную информацию. Например:

Программа 8. Создание именованных групп

```
public static void Main( ) {
    string stringl = "04:03:27 127.0.0.0 GotDotNet.ru";
    Regex theReg = new Regex( @"(?<время>(\d|\:)+)\s" +
          @"(?<ip>(\d|\ .)+)\s" + @"(?<url>\S+)");
    // группа time – одна и более цифр или двоеточий, за которыми следует пробельный символ
    // группа ip адрес – одна и более цифр или точек, за которыми следует пробельный символ
    // группа url – один и более непробельных символов
MatchCollection theMatches = theReg.Matches (stringl);
foreach (Match theMatch in theMatches) {
   if (theMatch.Length != 0) {
     // выводим найденную подстроку
      Console.WriteLine("\ntheMatch: {0}", theMatch.ToString ());
     // выводим группу "time"
      Console.WriteLine ("время: {0}", theMatch.Groups["время"]);
     // выводим группу "ip"
     Console.WriteLine("ip: {0}", theMatch.Groups["ip"]);
     // выводим группу "url"
     Console.WriteLine("url: {0}", theMatch.Groups["url"]);
}}}
```

ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ

1. На какие группы и разновидности разделяются все типы данных в C#? Примеры.
2. Основное отличие структурных типов от ссылочных?
3. Перечислить все целочисленные арифметические типы данных в C# и их названия в CLR. Указать их диапазон и размер занимаемой памяти.
4. Перечислить все вещественные арифметические типы данных в C# и их названия в CLR. Указать их диапазон и размер занимаемой памяти.
5. Перечислить все символьные типы данных в C# и их названия в CLR. Указать их диапазон и размер занимаемой памяти.
6. Оператор цикла foreach и его применение в программах.
7. Определение одномерного массива в C#. Инициализация одномерного массива.
8. Определение многомерного массива в C#. Инициализация многомерного массива.
9. Определение ступенчатых массивов и их инициализация.
10. Базовый класс Array, его методы и свойства.
11. Тип char и принимаемые значения переменными типа char. Методы и свойства класса Char.
12. Тип char[] и его отличительные особенности от C/C++.
13. Тип string и способы его конструирования.
14. Операции над строками типа string.
15. Способы задания строковых констант.
16. Методы и свойства класса String.
17. Класс StringBuilder. Конструкторы.
18. Операции над строками StringBuilder.
19. Основные методы класса StringBuilder.
20. Что такое регулярные выражения? Для чего применяются регулярные выражения?
21. Задание регулярного выражения. Поиск подстрок с помощью регулярного выражения.
22. Класс Regex и его методы.
23. Поиск первого вхождения образца с текст.
24. Поиск всех вхождений образца в текст.
25. Замена образца в тексте.
26. Разбор текста на лесемы.

27. Свойства класса Match.
28. Классы Match, MatchCollection и их свойства.
29. Классы Group, GroupCollection и их свойства.

ЛАБОРАТОРНАЯ РАБОТА 2

ТИПЫ ДАННЫХ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ. НАСЛЕДОВАНИЕ. ОБРАБОТКА ИСКЛЮЧЕНИЙ В C#

Цель работы:

– познакомиться с пользовательскими типами данных в языке C#: структура и перечисление;

– ознакомиться со структурой класса, его созданием и использованием, описанием членов класса: полей, свойств, инициализации объектов класса с помощью конструкторов;

– изучить механизм создания иерархий классов в C# и применение интерфейсов при наследовании;

– изучить механизм генерации и обработки исключений.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Проработать примеры программ 1–6, данные в теоретических сведениях. Создать на их основе программы. Получить результаты работы программ и уметь их объяснить. Внести в отчет с комментариями.

2. Для заданной структуры данных разработать абстрактный класс и класс-наследник. В классе реализовать несколько конструкторов. Создать методы, работающие с полями класса. Часть из них должны быть виртуальными. Добавить методы-свойства и индексаторы.

3. Разработать интерфейсные классы, добавляющие некоторые методы в класс-потомок. Изучить причины возникновения коллизий имен при наследовании и способы их устранения.

4. Разработать классы исключительных ситуаций и применить их для обработки, возникающих исключений.

5. Написать демонстрационную программу.

Описания данных пользовательских типов

1. СЛУЖАЩИЙ: имя, возраст, рабочий стаж, должности.
2. ИЗДЕЛИЕ: название, шифр, количество, комплектация.

3.ПЕЧАТНОЕ ИЗДАНИЕ: название, ФИО автора, стоимость, оглавление.

4. ЭКЗАМЕН: ФИО студента, дата, оценка, перечень вопросов.

5. ТОВАР: название, артикул, стоимость, даты (изготовление, срок реализации)

6. ЦЕХ: название, начальник, количество рабочих, перечень номенклатуры выпускаемых изделий.

7. АВТОМОБИЛЬ: марка, мощность, стоимость, даты ремонта.

8. СТРАНА: название, форма правления, площадь, список областей.

9. ЖИВОТНОЕ: вид, класс, средний вес, места обитания.

10. КОРАБЛЬ: название, водоизмещение, тип, список категорий кают.

11. КАРТИНА: ФИО автора, название, жанр, список владельцев.

12. МУЗЕЙ: Название, адрес, ФИО директора, количество и названия залов.

13. КНИГА: Название, жанр, количество страниц, список авторов.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Перечисление – это частный случай класса. Перечисление задает конечное множество возможных значений, которые могут получать объекты класса перечисление. Синтаксис объявления этого класса: включает заголовок и тело класса, содержащее список возможных значений:

[атрибуты][модификаторы]enum имя_перечисления
[: базовый класс] {список_возможных_значений}

Особенности объявления перечислений:

- перечисления могут быть объявлены непосредственно в пространстве имен проекта или могут быть вложены в описание класса;
- константы разных перечислений могут совпадать. Имя константы всегда уточняется именем перечисления;
- константы могут задаваться словами русского языка;
- разрешается задавать базовый класс перечисления (любой целочисленный тип кроме long).

Например, для создания системы расчета заработной платы сотрудников можно использовать имена VP, Manager, Grunt и Contractor.

Значения по умолчанию:

```
enum EmpType {
 Manager,    // = 0
 Grunt,      // = 1
 Contractor, // = 2
 VP          // = 3
}
```

Можно задать значения:

```
enum EmpType {
Manager = 102,
Grunt,      // = 103
Contractor, // = 104
VP          // = 105
}
```

Структуры C# можно рассматривать как разновидность классов. Для структур можно определять конструкторы (только с параметрами), структуры могут реализовывать интерфейсы. Синтаксис объявления структуры:

[атрибуты][модификаторы]struct имя_структуры [:список_интерфейсов] {тело_структуры}

Программа 1. Использование перечислений и структур в программах на C#

```
enum EmpType : byte {
        Manager = 10,           Grunt = 1,
        Contractor = 100,       VP = 9
}
struct Employee {
        public EmpType title;   // Поле – перечисление
        public string name;
        public short deptID;
}
class StructTester {
        public static int Main(string[] args) {
        Employee fred;          //Создание структурной переменной
        fred.deptID = 40;
        fred.name = "Fred";
        fred.title = EmpType.Grunt;
        return 0;
}}
```

Для структур можно определить конструкторы с параметрами:

```
struct Employee {
        . . .
        // Конструктор
        public Employee (EmpType et, string n, short d) {
                title = et;    name = n;   deptID = d;
}}
```

Теперь можно создавать объекты сотрудников следующим образом:

```
class StructTester {
public static int Main(string[] args) {
    Employee mary =new Employee (EmpType.VP, "Mary", 10);
    return 0;
}}
```

Классы являются расширением структурного типа. Синтаксис описания класса:

[атрибуты][модификаторы]class имя_класса
[:список_родителей] {тело_класса}

Модификаторы доступа к членам классов:

public – член доступен вне определения класса и иерархии производных классов.

protected – член не видим за пределами класса, к нему могут обращаться только производные классы.

private – член не доступен за пределами области видимости определяющего его класса.

internal – член видим только в пределах текущей единицы компиляции.

Обычно класс имеет модификатор доступа public, являющийся значением по умолчанию. Модификаторы private и protected используются для вложенных классов.

В теле класса могут быть объявлены: константы, поля, конструкторы и деструкторы, методы, события, делегаты, вложенные классы (структуры, интерфейсы, перечисления).

```
class Employee {
   private string fullName;
   private int empID;
   private float currPay;
   public Employee() {}
   public Employee(string fullName, int empID, float currPay) {
          this. fullName = fullName;   this.empID = empID;
          this. currPay = currPay;
  }
// Метод для увеличения зарплаты сотрудника
  public void GiveBonus(float amount)
  { currPay += amount; }
// Метод для вывода сведений о текущем состоянии объекта
  public virtual void DisplayStats() {
          Console. WriteLine("Name: {0}", fullName);
          Console.WriteLine(“Pay: {0}", currPay);
          Console.WriteLine(“ID: {0}", empID);
   }
}
public static void Main() {
  Employee e = new Employee(“Joe", 80, 30000);
  e.GiveBonus(200);
  Employee e2;
  e2 = new Employee("Beth", 81, 50000);
  e2.GiveBonus(1000);
  e2.DisplayStats();
}
```

Методы-свойства. В языке C# принято поля объекта объявлять закрытыми, а нужную стратегию доступа организовывать через методы. Для эффективности этого процесса введены специальные **методы-свойства**. Они позволяют реализовать различные фомы доступа к полям объекта.

Перечислим пять наиболее употребительных стратегий:

- чтение, запись (Read, Write);
- чтение, запись при первом обращении (Read, Write-once);
- только чтение (Read-only);
- только запись (Write-only);
- ни чтения, ни записи (Not Read, Not Write).

Рассмотрим класс Person, у которого пять полей: fam, status, salary, age, health, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны. Для каждого из этих полей может быть своя стратегия доступа.

Программа 2. Свойства в класса на C#

```
public class Person {
  string fam="", status="", health="";
  int age=0, salary=0;
  public string Fam   {              //стратегия: Read,Write-once
    set {if (fam == "") fam = value;}
    get {return(fam);}
  }
  public string Status   {           //стратегия: Read-only
    get {return(status);}
  }
  public int Age   {                 //стратегия: Read,Write
    set     {
      age = value;
      if(age < 7) status ="ребенок";
      else if(age <17) status ="школьник";
      else if (age < 22) status = "студент";
      else status = "служащий";
    }
    get {return(age);}
  }
  public int Salary   {              //стратегия: Write-only
    set {salary = value;}
  }
}
public void TestPersonProps(){
  Person pers1 = new Person();
  pers1.Fam = "Петров";   pers1.Age = 21;   pers1.Salary = 1000;
  Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,
pers1.Age, pers1.Status);
  pers1.Fam = "Иванов"; pers1.Age += 1;
  Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,
pers1.Age, pers1.Status);
}
```

Индексатор. Обеспечивает доступ к закрытому полю, представляющему массив. Объекты класса индексируются по этому полю. У класса может быть только один индексатор со стандартным именем **this**.

Добавим в класс Person массив children, задающий детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит индексатор:

Программа 3. Индексаторы в классе на C#

```
…
const int Child_Max = 10;          //максимальное число детей
Person[] children = new Person[Child_Max];
int count_children=0;              //текущее число детей объекта
public Person this[int i]  {       //индексатор
  get {
       if (i>=0 && i< count_children) return(children[i]);
       else return(children[0]);
  }
  set   {
    if (i==count_children && i< Child_Max)  {
       children[i] = value; count_children++;
    }
 }}
public void TestPersonChildren(){
  Person pers1 = new Person(), pers2 = new Person();
  pers1.Fam = "Петров";  pers1.Age = 42;
  pers1.Salary = 10000;    pers1[pers1.Count_children] = pers2;
  pers2.Fam ="Петров"; pers2.Age = 21; pers2.Salary = 1000;
 Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,
pers1.Age, pers1.Status);
  Console.WriteLine ("Сын={0}, возраст={1}, статус={2}",
pers1[0].Fam, pers1[0].Age, pers1[0].Status);
}
```

Статические поля и методы класса. У класса могут быть поля, связанные не с объектами, а с классом. Эти поля объявляются как статические с модификатором **static**. Статические поля доступны всем методам класса. Независимо от объекта используется одно и то же статическое поле, позволяя использовать информацию, созданную другими объектами класса.

Аналогично полям у класса могут быть и статические методы, объявленные с модификатором **static**. Такие методы обрабатывают общую для класса информацию, хранящуюся в его статических полях.

Наследование. Повторное использование кода – это одна из главных целей ООП. Класс-**потомок** наследует все возможности родительского класса – все поля и все методы, открытую и закрытую часть класса. Единственное, что не наследует потомок – это **конструкторы родительского класса**. Конструкторы потомок должен создавать сам. В C# разрешено только **одиночное наследование**, т. е. у класса-потомка может быть только один родительский класс.

Рассмотрим класс Found, играющий роль родительского класса:

Программа 4. Родительский класс Found

```
public class Found{
  protected string name;
  protected int credit;
  public Found()  {  }
  public Found(string name, int sum)  {
    this.name = name; credit = sum;
  }
  public virtual void VirtMethod()  {     //виртуальный метод
    Console.WriteLine ("Отец: " + this.ToString() );
  }
  //переопределенный метод базового класса Object
public override string ToString()  {
    return(String.Format("поля: name = {0}, credit = {1}", name, credit));
  }
  public void NonVirtMethod()  {
    Console.WriteLine ("Мать: " + this.ToString() );
  }
  public void Analysis()  {
    Console.WriteLine ("Простой анализ");
  }
  public void Work()  {
    VirtMethod();
    NonVirtMethod();
    Analysis();
  }
}
```

Класс Found переопределил родительский метод класса Object ToString(), задав собственную реализацию возвращаемой методом строки. На переопределение родительского метода указывает модификатор **override**.

Создадим класс Derived – потомка класса Found. Потомок может добавить новые свойства – поля класса. Заметьте, потомок **не может ни отменить, ни изменить модификаторы или типы полей, наследованных от родителя.**

Класс Derived добавляет новое поле **protected int debet.**

Класс должен позаботиться о **создании собственных конструкторов**, поскольку, как правило, добавляет собственные поля, о которых родитель ничего не может знать. Если не задать конструкторы класса, то будет добавлен конструктор по умолчанию, инициализирующий все поля значениями по умолчанию. Но это редкая ситуация. Чаще всего класс создает собственные конструкторы и, как правило, не один, задавая разные варианты инициализации полей.

```
public Derived(String name, int cred, int deb): Found (name,cred)
{}
```

Вызов конструктора родителя происходит не в теле конструктора, а в заголовке, пока еще не создан объект класса.

Добавление методов и изменение методов родителя

Потомок может добавлять новые собственные методы.

Класс-потомок может изменять наследуемые им методы. Если потомок создает метод с именем, совпадающим с именем метода предков, то возможны три ситуации:

- **перегрузка метода**. Она возникает, когда сигнатура создаваемого метода отличается от сигнатуры наследуемых методов предков;
- **переопределение метода**. Метод родителя в этом случае должен иметь модификатор **virtual** или **abstract**. При переопределении сохраняется сигнатура и модификаторы доступа наследуемого метода;
- **сокрытие метода**. Если родительский метод не является виртуальным или абстрактным, то потомок может создать новый метод с тем же именем и той же сигнатурой, скрыв родительский метод в данном контексте. При вызове метода предпочтение будет отдаваться методу потомка. Скрытый родительский метод всегда может быть вызван, если его уточнить ключевым словом **base**. Метод потомка следует сопровождать модификатором **new**, указывающим на новый метод.

Программа 4. Класс наследник с переопределенными методами родителя

```
public class Derived:Found {
   protected  int debet;
   public Derived() {}
  public Derived(String name, int cred, int deb): Found (name,cred){
            debet = deb;
  }
  public void DerivedMethod(){ //новый метод потомка
           Console.WriteLine("Это метод класса Derived");
  }
  new public void Analysis(){     //сокрытие метода родителя
             base.Analysis();
            Console.WriteLine("Сложный анализ");
  }
  public void Analysis(int level){            // перегрузка метода
             base.Analysis();
             Console.WriteLine("Анализ глубины {0}", level);
   }
   public override String ToString(){       //переопределение метода
            return(String.Format("поля:  name  =  {0},  credit  =  {1},debet
={2}",name, credit, debet));
   }
  // переопределение метода  родителя
    public override void VirtMethod()   {
            Console.WriteLine ("Сын: " + this.ToString() );
   }
}
```

Интерфейсы. Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны. Методы интерфейса объявляются без указания модификатора доступа (по умолчанию public). Класс, наследующий интерфейс, обязан реализовать все методы интерфейса.

В языке C# полного множественного наследования классов нет. Чтобы частично сгладить этот пробел, допускается множественное наследование интерфейсов.

Две стратегии реализации интерфейса. Опишем некоторый интерфейс, задающий дополнительные свойства объектов класса:

```
public interface IProps{
  void Prop1(string s);
  void Prop2 (string name, int val);
}
```

- Класс, наследующий интерфейс и реализующий его методы, может реализовать их явно, объявляя соответствующие методы класса открытыми (public).
- Другая стратегия реализации состоит в том, чтобы некоторые методы интерфейса сделать закрытыми (private), уточнив имя метода именем интерфейса:

```
public class ClainP:IProps{
  public ClainP(){ }
  void IProps.Prop1(string s)   {
    Console.WriteLine(s);
  }
  void IProps.Prop2(string name, int val)   {
    Console.WriteLine("name = {0}, val ={1}", name, val);
  }
}
```

Есть два способа получения доступа к закрытым методам.

- ***Обертывание***. Создается открытый метод, являющийся оберткой закрытого метода.
- ***Кастинг***. Создается объект интерфейсного класса IProps, полученный преобразованием (кастингом) объекта исходного класса ClainP. Этому объекту доступны закрытые методы интерфейса.

Вот пример обертывания закрытых методов в классе ClainP:

```
public void MyProp1(string s){
  ((IProps)this).Prop1(s);
}
public void MyProp2(string s, int x){
  ((IProps)this).Prop2(s, x);
}
```

Методы переименованы и они получили другие имена, под которыми и будут известны клиентам класса. В обертке для вызова закрытого метода пришлось использовать кастинг, приведя объект this к интерфейсному классу IProps.

Преобразование к классу интерфейса. Создать объект класса интерфейса обычным путем с использованием операции **new** нельзя, но можно объявить объект интерфейсного класса и связать его с настоящим объектом путем приведения (кастинга) объекта наследника к классу интерфейса.

```
public void TestClainIProps(){
  Console.WriteLine("Объект класса ClainP вызывает открытые
методы!");
  ClainP  clain = new Clain();
  clain.Prop1(" свойство 1 объекта");
  clain.Prop2("Владимир", 44);
  Console.WriteLine("Объект класса IProps вызывает открытые
методы!");
  IProps ip = (IProps)clain;
  ip.Prop1("интерфейс: свойство");
  ip.Prop2 ("интерфейс: свойство",77);
}
```

Проблемы множественного наследования. При множественном наследовании интерфейсов также возникают проблемы, но решение их становится проще. Рассмотрим две основные проблемы – коллизию имен и наследование от общего предка.

1. ***Коллизия имен***

Проблема **коллизии имен** возникает, когда два или более интерфейса имеют методы с одинаковыми именами и сигнатурой. Здесь возможны **две стратегии – склеивание методов и переименование.**

- Стратегия склеивания применяется тогда, когда класс – наследник интерфейсов полагает, что разные интерфейсы задают один и тот же метод, единая реализация которого и должна быть обеспечена наследником. В этом случае наследник строит единственную общедоступную реализацию.
- Другая стратегия исходит из того, что методы разных интерфейсов должны быть реализованы по-разному. В этом случае необходимо переименовать конфликтующие методы. Для этого достаточно реализовать методы разных интерфейсов как закрытые, а затем открыть их с переименованием.

Пример двух интерфейсов, имеющих методы с одинаковой сигнатурой, и класса – наследника этих интерфейсов, применяющего разные стратегии реализации для конфликтующих методов:

Программа 5. Коллизии имен в интерфейсах

```
public interface IProps{
  void Prop1(string s);
  void Prop2 (string name, int val);
  void Prop3();
}
public interface IPropsOne{
  void Prop1(string s);
  void Prop2 (int val);
  void Prop3();
}
public class ClainTwo:IProps,IPropsOne {
  // склеивание методов двух интерфейсов
  public void Prop1 (string s)  {  Console.WriteLine(s);  }
  // перегрузка методов двух интерфейсов
  public void Prop2(string s, int x)  {  Console.WriteLine(s + x);  }
  public void Prop2 (int x)  {   Console.WriteLine(x);  }
// private реализация и переименование методов  2-х интерфейсов
  void IProps.Prop3()  {
    Console.WriteLine("Метод 3 интерфейса 1");
  }
  void IPropsOne.Prop3()  {
    Console.WriteLine("Метод 3 интерфейса 2");
  }
  public void Prop3FromInterface1()  { ((IProps)this).Prop3();  }
  public void Prop3FromInterface2()  { ((IPropsOne)this).Prop3(); }
}
public void TestTwoInterfaces(){
  ClainTwo claintwo = new ClainTwo();
  claintwo.Prop1("Склейка свойства двух интерфейсов");
  claintwo.Prop2("перегрузка .: ",99);
  claintwo.Prop2(9999);
  claintwo.Prop3FromInterface1();
  claintwo.Prop3FromInterface2();
  Console.WriteLine("Интерфейсный объект вызывает методы 1-го интерфейса!");
```

```
  IProps ip1 = (IProps)claintwo;
  ip1.Prop1("интерфейс IProps: свойство 1");
  ip1.Prop3();
  Console.WriteLine("Интерфейсный объект вызывает методы 2-го интерфейса!");
  IPropsOne ip2 = (IPropsOne)claintwo;
  ip2.Prop1("интерфейс IPropsOne: свойство1");
  ip2.Prop3();
}
```

2. ***Наследование от общего предка***

Для интерфейсов ситуация дублирующего наследования маловероятна, но возможна, поскольку интерфейс, как и любой класс, может быть наследником другого интерфейса. Поскольку у интерфейсов наследуются только сигнатуры, а не реализации, то **проблема дублирующего наследования сводится к проблеме коллизии имен**.

Обработка исключительных ситуаций. Язык C# наследовал схему исключений языка C++, внеся в нее свои коррективы. Рассмотрим схему подробнее:

```
try {...}
catch (T1 e1) {...}
...
catch(Tk ek) {...}
finally {...}
```

Программные строки модуля, где возможно возникновение исключительной ситуации, необходимо сделать охраняемыми, заключив их в блок с ключевым словом **try**. Вслед за try-блоком могут следовать **catch**-блоки, называемые блоками-обработчиками исключительных ситуаций, их может быть несколько, но могут и отсутствовать. Завершает эту последовательность **finally-блок** – блок финализации, который также может отсутствовать. Вся эта конструкция может быть вложенной – в состав try-блока может входить другая конструкция try-catch-finally.

throw [выражение] – генерирует исключение и созает объект класса, являющегося наследником класса **Exception**. Обычно это выражение new, создающее объект класса исключения Exeption или его наследника.

Блок finally. В блоке try могли быть заняты ресурсы – открыты файлы, захвачены некоторые устройства. Освобождение ресурсов, занятых try-блоком, выполняет finally-блок. Если он присутствует, то выполняется всегда, сразу же после завершения работы try-блока, как бы тот ни завершился.

ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ

1. К какой группе типов переменных относятся перечисления и структуры?
2. Описание перечисления и его назначение?
3. Числовые значения констант перечисления по умолчанию?
4. Использование перечислений в программах.
5. Описание структуры. Конструкторы.
6. Обращение к элементам структуры.
7. Что такое класс? Для чего создаются классы?
8. Чем отличается класс от структуры?
9. Модификаторы доступа к полям и методам класса.
10. Модификаторы доступа к классам.
11. Что такое экземпляр класса? Как он создается в C#?
12. Для чего в классе определяется конструктор? Сколько может быть конструкторов в классе? Когда вызывается конструктор?
13. Как можно обратиться к полям и методам класса?
14. Методы-свойства класса. Назначение и описание.
15. Статические поля и методы класса. Назначение, описание и вызов статических методов.
16. Индексаторы. Назначение и описание.
17. Какое наследование применяется в C#? Что наследует потомок от класса-родителя?
18. Изменение методов родителя в классе наследника.
19. Конструкторы при наследовании.
20. Описание абстрактных методов и классов.
21. Вложенные классы.
22. Интерфейсы. Назначение и описание.
23. Наследование в интерфейсах.
24. Реализация методов интерфейсов в классах.
25. Коллизия имен в интерфейсах.
26. Исключения. Обработка исключений в C#.
27. Создание классов исключений и генерация исключения.

ЛАБОРАТОРНАЯ РАБОТА 3

РАЗРАБОТКА GUI. СОЗДАНИЕ SDI-ПРИЛОЖЕНИЙ ОБРАБОТКА СОБЫТИЙ

Цель работы:

– изучить принципы разработки графического интерфейса приложений для ОС Windows в Visual Studio .Net;

– освоить использование элементов графического интерфейса для управления работой приложения.

– освоить принципы построения иерархических меню, создания диалоговых окон;

– изучить модель обработки событий в языке C#.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Создать учебные примеры (программы 2–5) и разобрать принцип их работы. Поместите в отчет примеры работы программ, их код с комментариями.

2. Создать SDI-приложение (Single Document Interface, однодокументный интерфейс) с элементами ввода и отображения полей класса из задания к лабораторной работе 2. Для этого используйте различные элементы управления: текстовые поля, списки, независимые и радиокнопки, а также панели и менеджеры компоновки.

3. Ввод новых данных осуществлять через дополнительную диалоговую форму.

4. При изменении данных запрашивать подтверждение через окно диалога. В случае неполных данных сообщать об ошибке.

5. Объекты сохранять в коллекции.

6. Реализовать просмотр всей коллекции объектов через список. Для редактирования выбранного объекта создать дополнительную форму модального диалога.

7. Добавить на форму меню, позволяющее работать с пунктами: добавить, просмотреть, удалить, редактировать, справка.

8. Дублировать основные операции панелью инструментов.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Средством взаимодействия пользователя с программой является графический пользовательский интерфейс (Graphical User Interface, GUI). На практике программирование Windows-приложений с GUI

предполагает экстенсивное использование различных инструментальных средств и мастеров, которые намного упрощают этот процесс.

Windows Forms – это часть каркаса .NET Framework, которая поддерживает создание приложений со стандартным GUI на платформе Windows.

Форма – это экранный объект, обеспечивающий функциональность программы. Как правило, приложение содержит главное окно, которое реализовано с помощью некоторого класса MyForm, производного от класса Form.

Запускаем Visual Studio .NET, создаем новый проект, для которого выбираем тип **Windows Forms Application**, задаем имя проекта – **FirstForm** и сохраняем его в папку, определяемую полем Location. На экране появилась пустая Windows-форма (рис. 3.1).

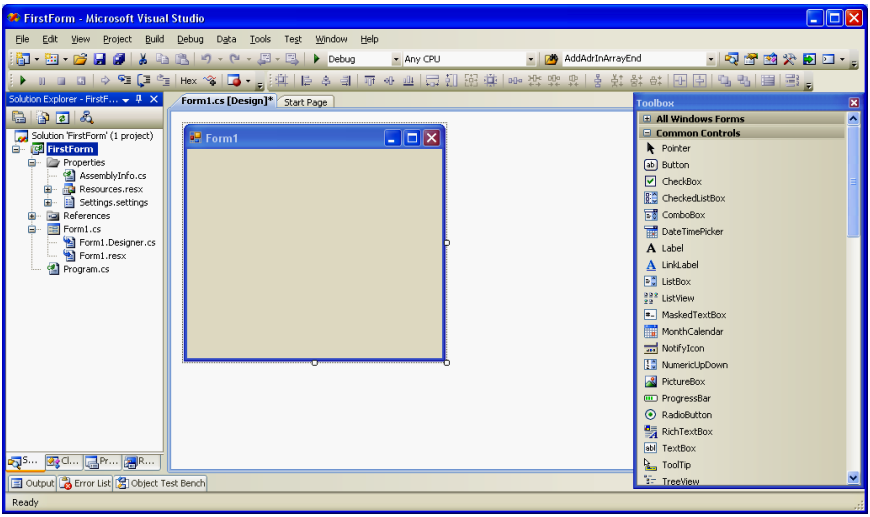


Рис. 3.1. Главное окно программы в режиме разработки приложения с GUI

Окно **Object Browser** (проводник объектов, View —> Object Browser) является средством получения информации о свойствах объектов. Можно получать краткое описание любого метода, класса или свойства, просто щелкнув на нем, – на информационной панели немедленно отобразится краткая справка.

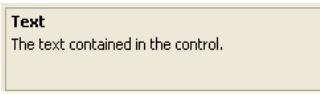
Окно **Class View** (обзор классов, View —> Class View) позволяет перемещаться в коде по выбранному объекту; содержит методы,

классы, данные всего листинга проекта. Для перехода, например, в class Form1, щелкаем на соответствующем названии в окне **Class View**.

Окно свойств **Properties** – основной инструмент настройки формы и ее компонент. Содержимое этого окна представляет собой весь список свойств выбранного в данный момент компонента или формы. Вызывается окно несколькими способами, чаще на выбранном объекте щелкаем правой кнопкой мыши и в контекстном меню пункт **Properties**. Когда вы только создали проект, в окне **Properties** отображаются свойства самой формы (табл. 3.1, 3.2).

Таблица 3.1

Описание интерфейса окна Properties

Элемент	Изображение	Описание
Object name		Название выбранного объекта, который является экземпляром какого-либо класса
Categorized		Сортировка свойств выбранного объекта по категориям
Alphabetic		Сортировка свойств и событий объекта в алфавитном порядке
Properties		Перечисление свойств объекта
Events		Перечисление событий объекта
Description Pane		Информация о выбранном свойстве

В табл. 3.2 приводится описание некоторых свойств формы, обычно определяемых в режиме дизайна.

Таблица 3.2

Некоторые свойства формы

Свойство	Описание	Значение по умолчанию
Name	Название формы в проекте	Form1 и т. д.
AcceptButton	Значение кнопки, которая будет срабатывать при нажатии клавиши Enter	None
BackColor	Цвет формы	Control
Background-Image	Изображение на заднем фоне	None
CancelButton	Значение кнопки, которая будет срабатывать при нажатии клавиши Esc	None
ControlBox	Наличие трех стандартных кнопок в верхнем правом углу формы: "Свернуть", "Развернуть" и "Закрыть"	
Cursor	Вид курсора при его положении на форме	Default
DrawGrid	Включение сетки из точек, которая помогает форматировать элементы управления	True
Font	Выбор шрифта, используемого для отображения текста на форме в элементах управления	Microsoft Sans Serif; 8,25pt
Icon	Изображение иконки, располагаемой в заголовке формы. Формат .ico	 (Icon)
Maximize-Box	Определяется активность стандартной кнопки "Развернуть" в верхнем правом углу формы	True
Maximum-Size	Максимальный размер ширины и высоты формы при нажатии на стандартную кнопку "Развернуть"	(Во весь экран)
Minimize-Box	Определяется активность стандартной кнопки "Свернуть" в верхнем правом углу формы	True
Minimum-Size	Минимальный размер ширины и высоты формы, задаваемый в пикселях	0;0
Size	Ширина и высота формы	300; 300

Окончание табл. 3.2

Свойство	Описание	Значение по умолчанию
FormBorder-Style	Определение вида границ формы: **None** – форма без границ и строки заголовка **FixedSingle** – тонкие границы без возможности изменения размера пользователем **Fixed3D** – границы без возможности изменения размера с трехмерным эффектом **FixedDialog** – границы без возможности изменения, без иконки приложения **Sizable** – обычные границы: пользователь может изменять размер границ **FixedToolWindow** – фиксированные границы, имеется только кнопка закрытия формы **SizableToolWindow** – границы с возможностью изменения размеров, имеется только кнопка закрытия формы	Sizable
StartPosition	Расположения формы при запуске приложения: **Manual** – форма появляется в верхнем левом углу экрана **CenterScreen** – в центре экрана **WindowsDefaultLocation** – расположение формы по умолчанию **WindowsDefaultBounds** – границы формы принимают фиксированный размер **CenterParent** – в центре родительской формы	Windows-Default-Location
Text	Заголовок формы	Form1 и т. д.
Window-State	Определение положения формы при запуске: **Normal** – форма запускается с размерами, указанными в свойстве Size **Minimized** – форма запускается с минимальными размерами, указанными в свойстве MinimumSize **Maximized** – форма разворачивается на весь экран	Normal

Кнопка Events  переключает окно Properties в режим управления обработчиками различных событий (например, мыши, клавиатуры)

и одновременно выводит список всех событий компонента. Двойной щелчок мыши в поле значения события генерирует обработчик для него и переключает в режим кода.

Окно **Toolbox** (панель инструментов, View —> Toolbox) содержит компоненты Windows-форм, называемые элементами управления, которые размещаются на форме. Оно состоит из нескольких закладок. Наиболее часто употребляемой закладкой является **All Windows Forms**. Для размещения нужного элемента управления достаточно просто щелкнуть на нем в окне **Toolbox** или, ухватив, перетащить его на форму.

Режимы дизайна и кода

При создании нового проекта запускается режим дизайна – форма представляет собой основу для расположения элементов управления. Для работы с программой следует перейти в режим кода. Это можно сделать нескольким способами: щелкнуть правой кнопкой мыши в любой части формы и выбрать **View Code**, в окне **Solution Explorer** сделать то же самое на компоненте Form1. После перехода в режим кода в этом проекте появится вкладка Form1.cs*, нажимая на которую, тоже можно переходить в режим кода.

Рассмотрим некоторые программные блоки.

1. Первый блок определяет, какие пространства имен используются в этом проекте:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Windows.Forms;
```

Далее определяется собственное пространство имен, имя которого совпадает с названием проекта: **namespace FirstForm.** При необходимости это название можно менять.

2. Класс формы Form1, наследуемый от **System.Windows.Forms.Form**, содержит в себе почти весь код программы:

```
namespace FirstForm{
   public partial class Form1 : Form    {
      public Form1()   { InitializeComponent(); }
   }
}
```

3. Метод Main в файле Program.cs реализует главную точку входа в программу – место, откуда начинается выполнение написанного нами кода:

```
namespace FirstForm{
   static class Program    {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()       {
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(new Form1());
      }
   }}
```

При отладке больших программ удобно использовать нумерацию строк, которую можно включить в пункте меню **Tools/Options…/Text Editor/C#** – на форме **General** – галочка **Line Numbers**.

При запуске приложения в папке **bin\Debug** внутри папки проекта возникает файл FirstForm.exe и файлы, необходимые для отладки. Файл FirstForm.exe и представляет собой готовое приложение. Для распространения приложения его необходимо скомпилировать в режиме **Release** (Build.Butch Build) – тогда появится папка **bin\Release**, которая будет содержать только FirstForm.exe. Вы можете просто скопировать его на другой компьютер, и если там имеется .NET Framework, все будет работать.

Элементы управления

Элементы управления – это компоненты, обеспечивающие взаимодействие пользователя с программой. Среда Visual Studio.NET предоставляет большое количество элементов, которые можно сгруппировать по нескольким функциональным группам.

Таблица 3.3

Основные группы элементов управления

Категория	Интерфейсные элементы
Редактирование текста	**TextBox, RichTextBox**
Отображение текста	**Label, LinkLabel, StatusBar**
Выбор из списка	**CheckedListBox, ComboBox, DomainUpDown, ListBox, ListView, NumericUpDown, TreeView**
Отображение графики	**PictureBox**
Хранение графики	**ImageList**
Ввод значений	**CheckBox, CheckedListBox, RadioButton, TrackBar**
Ввод даты	**DateTimePicker, MonthCalendar**
Диалоговые панели	**ColorDialog, FontDialog, OpenFileDialog, PrintDialog, PrintPreviewDialog, SaveFileDialog**
Создание меню	**MenuStrip, ContextMenuStrip**
Команды	**Button, LinkLabel, NotifyIcon, ToolBar**
Объединение компонентов	**Panel, GroupBox, TabControl**

Базовым классом всех интерфейсных элементов является класс **Control**, содержащийся в пространстве имен System.Windows.Forms. В этом классе определены общие для всех интерфейсных элементов свойства, события и методы. Наиболее важные из них:

- **Cursor, Font, BackColor, ForeColor** – свойства, значения которых элемент управления наследует от содержащего его контейнера, если значение этого свойства в явном виде не установлено и не определено в родительском классе;
- **Top, Left, Width, Height, Size, Location** – свойства, отвечающие за размер и местоположение элемента относительно контейнера (для формы контейнером является экран);

- **Anchor** и **Dock** – свойства, определяющие, согласно каким принципам перемещается и меняет размеры интерфейсный элемент при изменении размеров контейнера;
- **Text, ImeMode, RightToLeft** – свойства, определяющие надпись или текст в элементе управления, а также направление текста и способ его редактирования;
- **Enabled**, **Visible** – свойства, определяющие, доступен ли пользователю интерфейсный элемент и отображается ли он;
- **Parent** – свойство, указывающее, какой из интерфейсных элементов является контейнером для данного элемента.

Обработка событий в Windows Forms

GUI управляется событиями. Приложение выполняет действия в ответ на события, вызванные пользователем, например, на щелчок кнопкой мыши или выбор пункта меню. В Windows Forms применяется модель обработки событий .NET, в которой **делегаты** используются для того, чтобы связать события с обрабатывающими их методами. В классах Windows Forms используются групповые делегаты. **Групповой делегат содержит список связанных с ним методов**. Когда в приложении происходит событие, управляющий элемент возбуждает событие, вызвав делегат для этого события, который вызывает связанные с ним методы. Для того чтобы добавить делегат к событию, используется перегруженный оператор +=. Например:

```
this.MouseClick += new
MouseEventHandler(this.Form1_MouseClick);
```

Объявление обработчика для этого события:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
 { }
```

В качестве параметра обработчик событий получает объект класса MouseEventArgs (производный от класса EventArgs). Свойства этого объекта содержат информацию, связанную с данным событием.

- Button (Кнопка) определяет, какая кнопка была нажата;
- Clicks (Щелчки) определяет, сколько раз была нажата и отпущена кнопка;
- свойство Delta (Дельта) является счетчиком оборотов колесика мыши;

- *X* и *Y* – координаты точки, в которой находился указатель в момент нажатия кнопки мыши.

Внесем изменения в FirstForm, чтобы при щелчке кнопкой мыши строка с приветствием перемещалась на место щелчка.

Программа 1. Отображает перемещение приветствия по щелчку мыши.

```
public partial class Form1 : Form     {
      float x, y;                        // координаты
      Brush pStdBrush;                   // Кисть
      Graphics poGraphics;
      public Form1()       {
         InitializeComponent();
         x=10;      y=20;
         pStdBrush = new SolidBrush(Color.Black);
         poGraphics = this.CreateGraphics();
         this.Text = "Программа 1";
         this.Show();
         poGraphics.DrawString("Hello,     Window     Forms",     this.Font,
pStdBrush, x, y);
      }
      private void Form1_MouseClick(object sender,
MouseEventArgs e)       {
         // координаты точки щелчка мыши
   x = (float)e.X;    y = (float)e.Y;
         poGraphics.DrawString("Hello, Window Forms", this.Font,
pStdBrush, x, y);
      }
}
```

Параметры метода DrawString:

- выводимая строка;
- шрифт (Font— свойство класса Form, которое определяет шрифт, по умолчанию применяемый для вывода текста в форму);
- используемая кисть;
- координаты в пикселях.

Метод Form1_MouseClick устанавливает координаты текста *x* и *y*, равными координатам точки, в которой находился указатель в момент щелчка.

Несколько обработчиков для события. Реализуем два обработчика события MouseClick. Второй обработчик по щелчку кнопкой мыши просто отображает окно сообщения. Метод ShowClick подключается к событию вручную аналогично методу Form1_MouseClick в файле Form1.Designer.cs.

Программа 2. Два обработчика событий для MouseClick

```
public partial class Form1 : Form {
   . . .
      private void Form1_MouseClick(object sender,
MouseEventArgs e)       {
         x = (float)e.X; y = (float)e.Y;
         poGraphics.DrawString("Hello, Window Forms", this.Font,
pStdBrush, x, y);
      }
      void ShowClick (object pSender, MouseEventArgs e)
          MessageBox.Show("Mouse clicked!!'");
}
```

Введем в пример обработку события KeyPress, а также покажем, как в событии MouseDown различать, какая кнопка была нажата: левая или правая (**программа 3**).

Обработка событий от правой и левой кнопки мыши. Для того чтобы определить, какая кнопка мыши была нажата, используем свойство Button параметра MouseEventArgs. Перепишем методы обработчика событий:

```
private void Form1_MouseClick(object sender, MouseEventArgs e)  {
      // если левая кнопка
    if (e.Button == System.Windows.Forms.MouseButtons.Left) {
                x = (float)e.X; y = (float)e.Y;
                poGraphics.DrawString("Hello, Window Forms",
this.Font, pStdBrush, x, y);
    }
}
void ShowClick (object pSender, MouseEventArgs e)     {
     // если правая кнопка
     if (e.Button == System.Windows.Forms.MouseButtons.Right)
                 MessageBox.Show("Mouse clicked!!'");
  }
```

Событие Keypress. При нажатии пользователем на клавишу клавиатуры в конец строки приветствия будет добавлен соответствующий символ. Вместо класса String используется класс StringBuilder, который более эффективен в этой ситуации.

```
public partial class Form1 : Form     {
      StringBuilder pStr;
      String s;
      public Form1()        {
. . .
         pStr = new StringBuilder("Hello, Window Forms");
         s = pStr.ToString();
         poGraphics.DrawString(s, this.Font, pStdBrush, x, y);
      }
   private void Form1_KeyPress(object sender, KeyPressEventArgs e)
  {         pStr.Append(e.KeyChar);        // Добавляем в конец
            s = pStr.ToString();
            poGraphics.DrawString(s, this.Font, pStdBrush, x, y);
   }
}
```

Программа 4. Создадим шуточную программу, представляющую собой диалоговое окно с двумя кнопками. Назовем его SocOpros. Из окна Toolbox перетаскиваем на форму две кнопки Button и надпись Label и устанавливаем следующие свойства элементов управления и формы (табл. 3.4).

Таблица 3.4

Описание формы приложения

Форма, свойства	Значение
FormBorderStyle	Fixed3D
Icon	Путь C:\Program Files\Microsoft Visual Studio 8\ Common7\ VS2008ImageLibrary\icons\....
Text	Социологический опрос
Label1, свойство	**Значение**
Bold	True
Text	Вы довольны своей зарплатой?
Button1, свойство	**Значение**
Name	Btnyes
Text	Да

Окончание табл. 3.4

Форма, свойства	Значение
Button2, свойство	**Значение**
Name	Btnno
Text	Нет

Щелкаем дважды по кнопке "Да". В обработчике этой кнопки вставляем следующий код:

```
void btnyes_Click(object sender, EventArgs e){
  MessageBox.Show("Мы и не сомневались, что Вы так думаете!");
}
```

Выделяем кнопку "Нет". Открываем окно Properties. Переключаемся в окно событий и дважды щелкаем в поле MouseMove.

В обработчике связываем движение мыши с координатами кнопки и устанавливаем координаты, куда она будет возвращаться, если во время своего движения выйдет за указанную область:

```
private void Btnno_MouseMove(object sender, MouseEventArgs e)
{       Btnno.Top -= e.Y;     Btnno.Left += e.X;
        if (Btnno.Top < -10 || Btnno.Top > 100)   Btnno.Top = 60;
        if (Btnno.Left < -80 || Btnno.Left > 250)   Btnno.Left = 120;
 }
```

Создание меню

Добавим в приложение SocOpros простое меню для выхода из программы, пункт меню File / Exit.

1. Откройте панель инструментов Toolbox и перетащите управляющий элемент MenuStrip на форму приложения.

2. Для создания выпадающего меню File с пунктом Exit, введите File и Exit, как на рис. 3.2. В окне Properties измените названия этих пунктов меню на MenuFile и MenuExit.

3. Добавьте код в обработчик события File / Exit.

```
private void MenuExit_Click(object sender, EventArgs e)       {
        Application.Exit();
    }
```

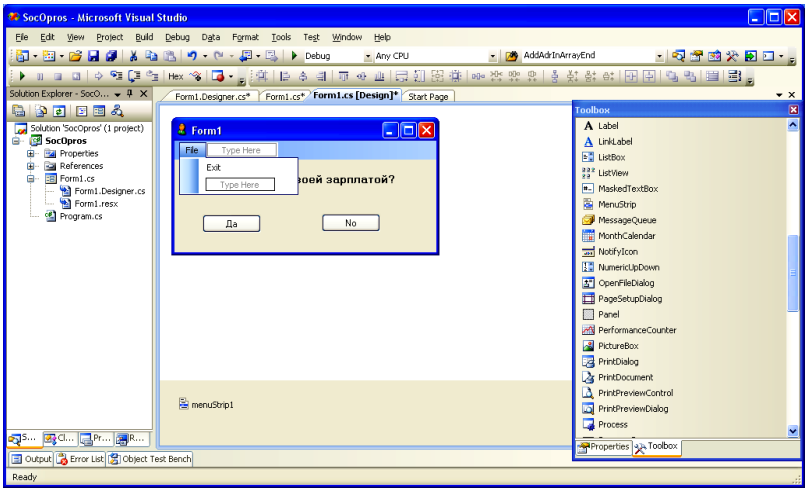


Рис. 3.2. MenuStrip для добавления в форму меню

Выбор пункта меню сочетанием клавиш. В свойстве **ShortcutKeys** в окне Properties для пункта меню выбрать из появившегося окна нужное сочетание клавиш, для отображения рядом с названием пункта меню.

Дополнительные возможности меню. В классе MenuStrip определены свойства, при помощи которых можно устанавливать флажок напротив пункта меню, прятать пункты меню, делать некоторые пункты меню недоступными и т. п. (табл. 3.5).

Таблица 3.5

Свойства MenuItem, обеспечивающие дополнительные возможности меню

Свойство	Назначение
Checked	Определяет, установлен ли флажок рядом с текстом пункта меню
DefaultItem	Определяет, какой пункт меню выбран по умолчанию
Enabled	Определяет, доступен ли тот или иной пункт меню
Index	Определяет позицию пункта меню
Shortcut	Устанавливает клавиши, используемые для активизации элемента меню в приложении
Text	Устанавливает название пункта меню

Закрытие формы

Существует несколько способов закрыть окно:

- щелкнуть на кнопке "X" (Закрыть) в правом верхнем углу окна;
- закрыть окно из системного меню в левом верхнем углу окна;
- закрыть окно с помощью комбинации клавиш Alt+F4;
- выйти из приложения с помощью меню File / Exit.

Когда закрывается форма, можно остановить процедуру завершения:

```
private void Form1_FormClosing(object sender,
FormClosingEventArgs e)       {
        if (MessageBox.Show("Do you want to close", "SocOpros",
        MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
        System.Windows.Forms.DialogResult.Yes)
                Application.Exit();
}
```

Для того чтобы закрыть главное окно и не выходить из приложения, обработчик меню File / Exit должен вызвать метод Close.

```
private void MenuExit_Click(object sender, EventArgs e)       {
        Close();
}
```

Диалоговые окна

Диалоговое окно – это набор управляющих элементов, с помощью которых упрощается процесс ввода данных. Для создания простых диалоговых окон используется класс **MessageBox**. Более сложные диалоговые окна создаются на основе форм.

Создание модального диалога

Программа 5

1. Создайте новое приложение AdminForm. На форме расположите кнопки **Add** и **Exit** и 4 элемента **Label** для отображения информации (рис. 3.3). По щелчку на кнопке Add отображается пустая форма AddHotelDialog.cs. Это обычная форма.

2. Откройте файл AddHotelDialog.cs в режиме конструктора. В окне Properties установите значение свойства FormBorderStyle равным FixedDialog.

3. Установите значение свойств ControlBox, MinimizeBox и MaximizeBox равным false.

4. Добавьте на форму текстовые поля TextBox и Label, содержащие информацию о гостинице. Кроме того, необходимо добавить кнопки OK и Cancel (рис. 3.4).

5. Установите значения и имена элементов в соответствии с табл. 3.6.

6. Установите значение свойства DialogResult кнопки OK равным OK. Точно так же установите значение этого свойства кнопки Cancel равным Cancel.

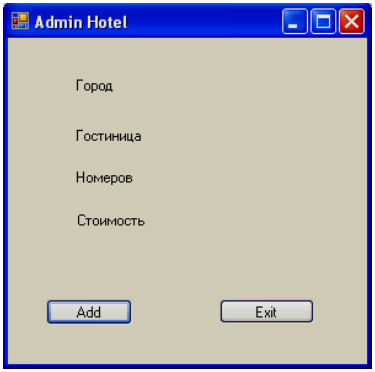


Рис. 3.3. Окно приложения AdminForm

Рис. 3.4. Диалоговое окно ввода информации о гостинице

Таблица 3.6

Значения свойств текстовых полей и кнопок диалога AddHotelDialog.cs

Имя (Name)	**Текст**
txtCity	(не заполнено)
txtHotelName	(не заполнено)
txtNumberRooms	(не заполнено)
txtRate	(не заполнено)
cmdOk	OK

cmdCancel	Cancel

7. В файле AdminForm.cs временно добавьте к обработчику cmdAdd_Click код, который отвечает за отображение окна диалога. Диалоговое окно отображается с помощью метода ShowDialog, а не метода Show, который используется для обычных форм.

```
private void cmdAdd_Click (object sender, EventArgs e) {
    AddHotelDialog dlg = new AddHotelDialog();
    dlg.ShowDialog();
}
```

8. Скомпонуйте и запустите пример. Теперь диалоговое окно уже можно открыть с помощью кнопки Add, а закрыть – с помощью любой из кнопок OK или Cancel. Можно проверить, что диалоговое окно является модальным, пощелкав мышью где-нибудь еще в приложении.

9. Закрыть приложение можно кнопкой Exit, добавив обработчик события:

```
private void cmdExit_Click(object  sender, EventArgs  e) {
  if(MessageBox.Show("Do you want to close", "Warning",
  MessageBoxButtons.YesNo, MessageBoxIcon.Question) ==
  System.Windows.Forms.DialogResult.Yes)   {
                Application.Exit();
}
```

Передача данных между диалогом и родительской формой

Для этих целей в классах .NET Framework нет встроенного механизма. В классе диалога для каждого сообщения, которое будет передаваться между родительской формой и диалогом, нужно определить свойство. Добавим в класс AddHotelDialog свойства City, HotelName, Rate (Стоимость) и NumberRooms.

```
public String City        {
        get { return txtCity.Text; }
        set { txtCity.Text = value; }     }
public String HotelName        {
        get { return (txtHotelName.Text); }
        set { txtHotelName.Text = value; }     }
public int Rooms        {
```

```
        get { return Convert.ToInt32(txtNumberRooms.Text); }
        set { txtNumberRooms.Text = value.ToString(); }      }
public double Rate       {
        get { return Convert.ToDouble(txtRate.Text); }
        set { txtRate.Text = value.ToString(); }      }
 private void cmdOk_Click(object sender, EventArgs e)
  {    Close();  }
```

Теперь можно использовать эти свойства при закрытии диалогового окна с помощью кнопки OK.

```
private void cmdAdd_Click(object sender, EventArgs e)       {
        AddHotelDialog dlg = new AddHotelDialog();
        dlg.ShowDialog();
        if (dlg.HotelName != "")          {
           label1.Text = dlg.City;
           label2.Text = dlg.HotelName;
           label3.Text = dlg.Rooms.ToString();
           label4.Text = dlg.Rate.ToString();
        } else {
      MessageBox.Show("Введите данные", "Hotel Broker
      Administration",  MessageBoxButtons.OK,
      MessageBoxIcon.Exclamation);
           return;
      } }
```

Создание панели инструментов

Кнопки панели инструментов обеспечивают пользователям более легкий доступ к возможностям, которые определены в меню. Для этого используется класс **ToolStrip**.

Для добавления объекта панель инструментов:

- перетащить объект ToolStrip с панели Toolbox;
- растянуть на форме, расположив в нужной части окна;
- по умолчанию будет сформирована метка на панели для выбора типа объектов панели инструментов;
- выбрать объект Button. Появится заготовка для кнопки с изображением. Для добавления изображения на кнопку нужно щелкнуть по

кнопке правой клавишей мыши и выбрать пункт **Set Image**. В появившемся окне нажать кнопку **Import** и выбрать путь к рисунку.

Список элементов ListBox

Добавьте в проект класс, описывающий объект гостиница.

```
class Hotel    {
        public Hotel(String city, String name, int r, double m)  {
           City = city; HotelName = name;
           Rooms = r; Rate = m;
        }
        public String City, HotelName;
        public int Rooms;
        public double Rate;
}
```

При запуске программы AdminForm в процессе инициализации метод Form1_Load осуществляет начальную загрузку списка элементов hotellist, в него загружается список гостиниц.

```
public partial class Form1 : Form    {
   ArrayList list = new ArrayList();       // список гостиниц
   public Form1()    {  InitializeComponent();  }
…
    private void Form1_Load(object sender, EventArgs e)       {
        Hotel ob1=new  Hotel("Москва","Россия",200,1500);
        list.Add(ob1);
        Hotel ob2=new Hotel("Москва","Прага",200,3000);
        list.Add(ob2);
        Hotel ob3=new Hotel("Новосибирск","Объ",150,1500);
        list.Add(ob3);
        Hotel ob4=new Hotel("Новосибирск","Тратата",300,1200);
        list.Add(ob4);
        hotellist.Items.Clear();
        if (list == null) { return; }
        foreach(Hotel hotel in list) {
          // строка для записи в элемент ListBox  hotellist
              String city = hotel.City.Trim();
              String name = hotel.HotelName.Trim();
              String rooms = hotel.Rooms.ToString();
              String rate = hotel.Rate.ToString();
              String str = city + "," + name + "," + rooms + "," + rate;
```

```
                hotellist.Items.Add(str);
        }
    }
}
```

ListBox содержит свойство Items, которое поддерживает коллекцию объектных ссылок. Сначала вызываем метод Items.Clear() для очистки списка. Потом с помощью цикла перебираем гостиницы в списке и создаем строку из полей структуры гостиницы, разделенных запятыми. Добавляем эту строку в список элементов методом Items .Add().

Для выбора элемента из списка нужно щелкнуть на нем. Выбор вызовет событие **SelectedIndexChanged**. Доступ к выбранному элементу можно получить с помощью свойств **SelectedIndex** и **SelectedItem**. Если никакой элемент не выбран, значение SelectedItem будет равно –1. Ниже приведен код обработчика события SelectedIndexChanged.

```
    private void hotellist_SelectedIndexChanged(object  sender,
EventArgs  e) {
        if (hotellist.SelectedIndex != -1) {
                String selected = hotellist.SelectedItem.ToString();
                String[] fields;
                fields = selected.Split(','); // поля разбить;
                label1.Text = fields[0];
                label2.Text = fields[1];
                label3.Text = fields[2];
                label4.Text = fields[3];
        } else { label1.Text = ""; }
    }
```

Добавление объектов, вводимых на дополнительной форме AddHotelDialog в коллекцию и элемент hotellist, осуществляется в обработчике события при нажатии на кнопку Add:

```
private void cmdAdd_Click(object  sender, EventArgs  e) {
        . . .
        String s=dlg.City+"," +dlg.HotelName+","
                +dlg.Rooms.ToString()+"," +dlg.Rate.ToString();
        hotellist.Items.Add(s);
        Hotel ob=new
                  Hotel(dlg.City,dlg.HotelName,dlg.Rooms,dlg.Rate);
```

```
            list.Add(ob);
            . . .
}
```

ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ

1. Что такое форма и ее назначение?
2. Что такое элементы управления? На какие группы они делятся?
3. Как установить элемент управления на форму и задать его свойства?
4. Что такое событие? Как в .Net реализуются события?
5. Что такое обработчик события?
6. Как выбрать событие для элемента управления?
7. Что такое делегат?
8. Как задаются обработчики событий для элементов управления?
9. Как происходит подключение к прослушиванию события?
10. Как создать собственные события и их обработчики?
11. Как создать верхнее меню?
12. Как добавить выпадающее меню в верхнее меню?
13. Как установить определенному пункту меню сочетание клавиш?
14. Как создать панель инструментов?
15. Как добавить несколько кнопок на панель инструментов?
16. Как разместить на кнопке изображение?
17. Как добавить новую форму в приложение?
18. Как организовать переход к добавленной форме?
19. Что такое модальная форма и немодальная? Как они вызываются?
20. Как организовать передачу данных между формами?
21. Как добавить новый класс в проект?

ЛАБОРАТОРНАЯ РАБОТА 4

СОЗДАНИЕ MDI-ПРИЛОЖЕНИЙ. СЕРИАЛИЗАЦИЯ ОБЪЕКТОВ. СТАНДАРТНЫЕ ДИАЛОГИ

Цель работы:

– изучить особенности разработки MDI-приложений в Visual Studio .Net;

– изучить способы сохранения данных в файл и загрузки из файла;

– освоить механизм сериализации и десериализации объектов.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Создать учебные примеры (программы 1–4) и разобрать принцип их работы. Поместить примеры работы программ и их коды с комментариями в отчет.

2. Создать текстовый редактор **NotepadC#**, добавив недостающие пункты меню и функции.

3. На основании лабораторной работы 3 создать MDI-приложение. Информация в окне должна отображаться в виде таблицы. Иметь возможность делать выборку данных по различным критериям. Переносить данные из одной формы в другую.

4. Добавить формы для ввода дополнительной информации об объекте и фото объекта.

5. Добавить пункты меню для сохранения объектов в файл и загрузки. При сохранении использовать стандартные диалоговые окна и механизм сериализации. В класс добавить поле «Дата создания объекта». Поле не сериализовать, а при десериализации заново устанавливать по системной дате.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Операции ввода/вывода данных в .NET

Поток – это перенос данных между программой и любым устройством ввода/вывода. Данные могут рассматриваться как **поток байтов, символов или объектов**. В пространстве имен **System.IO** есть несколько классов, позволяющих использовать различные устройства хранения, если только данные можно трактовать как байты или символы.

Потоковые классы. Stream — абстрактный класс, который является базовым для чтения и записи байтов в некоторое хранилище данных. Этот класс поддерживает синхронные и асинхронные чтение и запись. Класс Stream содержит типичные методы: **Read** , **Write**, **Seek**, **Flush** и **Close.**

Класс **FileStream**, который является производным от класса Stream, предоставляет операции чтения и записи последовательности байтов в файл. Конструктор FileStream создает экземпляр потока. Перегруженные методы класса Stream осуществляют чтение и запись в файл. У класса Stream есть и другие производные классы: **MemoryStream**, **BufferedStream** и **NetworkStream** (в System.Net.Sockets).

Программа 1. Запись и чтение 10 байтов в/из файл(а). Пример выполнить дважды. Первый раз программа создаст файл и запишет в него числа, а во второй раз прочитает и распечатает часть файла

```
using System.IO;
void Main() {
          byte [] data = new byte [10];
          FileStream fs = new FileStream( "FileStreamTest.txt",
FileMode.OpenOrCreate);
  if (fs.Length == 0)  {
     Console.WriteLine("Writing Data...");
     for (short i = 0; i < 10; i++)   data[i] = (byte)i;
         fs.Write(data, 0, 10);  // Запись данных
  } else  {
     fs.Seek(-5, SeekOrigin.End);  // Ищем конец
     int count = fs.Read(data, 0, 10);  // Чтение данных
     for (int i = 0; i < count; i++)   Console.WriteLine(data[i]);
  }
  fs.Close();
}
```

Встроенные типы данных и потоки. Если необходимо прочитать в поток или записать из потока простой тип, такой как Boolean, String или Int32, следует использовать классы **BinaryReader** и **BinaryWriter**. Нужно создать соответствующий поток (напимер, FileStream) и передать его в качестве параметра в конструктор BinaryReader или BinaryWriter. Потом можно использовать один из перегруженных методов Read или Write для чтения данных из потока или записи данных в поток.

Программа 2. Запись и чтение 10 чисел в/из файл(а). Пример выполнить дважды. Сначала файл создается, и в него записываются данные. Во второй раз данные читаются из файла.

```
void Main() {
```

```
    FileStream fs = new FileStream( "BinaryTest.bin",
FileMode.OpenOrCreate);
    if (fs.Length == 0)  {
          Console.WriteLine("Writing Data...");
          BinaryWriter w = new BinaryWriter(fs);
          for (short i = 0; i < 10; i++)   w.Write(i);  // Запись
          w.Close () ;
    } else {
          BinaryReader r = new BinaryReader(fs);
          for (int i = 0; i < 10; i++)  Console.WriteLine(r.ReadInt16());
          r.Close();
    }
    fs.Close();
}
```

Классы TextReader и TextWriter. В абстрактных классах TextReader и TextWriter данные рассматриваются как последовательный поток символов (текст). TextReader имеет следующие методы: Close, Peek (Считывание элемента данных), Read, ReadBlock, ReadLine и ReadToEnd. TextWriter содержит методы типа Close, Flush, Write и WriteLine. Перегруженные методы Read читают символы из потока.

Классы StringReader и **StringWriter** являются производными классами от классов TextReader и TextWriter соответственно. StringReader и StringWriter читают и записывают данные в символьную строку, которая сохраняется в базовом объекте StringBuilder. Конструктор StringWriter может принимать объект StringBuilder.

Классы StreamReader и **StreamWriter** также являются производными классами от классов TextReader и TextWriter. Они читают текст из объекта и записывают текст в объект Stream. Можно создать объект Stream и передать его в конструктор StreamReader или StreamWriter.

Программа 3. Запись и чтение символьных строк в/из файл(а). Программу выполнить дважды: первый раз – чтобы создать файл, а затем второй раз – чтобы прочитать его.

```
void Main() {
    FileStream fs = new FileStream( "TextTest.txt",
FileMode.OpenOrCreate);
    if (fs.Length == 0)    {
```

```
        Console.WriteLine("Writing Data..."); // Запись данных
        StreamWriter sw = new StreamWriter(fs);
        sw.Write(100); // Запись
        sw.WriteLine(" One Hundred"); // Сто
        sw.WriteLine("End of File"); // Конец Файла
        sw.Close();
    } else {
        String text; // Строка
        StreamReader sr = new StreamReader(fs) ;
        text = sr.ReadLine(); // текст
        while (text != null)  {
            Console.WriteLine(text);
            text = sr.ReadLine();
        }
        sr.Close ();
    }
    fs.Close ();
}
```

Классы File и FileInfo. Класс File содержит методы для создания и открытия файлов, которые возвращают объекты FileStream, StreamWriter или StreamReader, производящие фактическое чтение и запись. Перегруженный метод Create возвращает объект FileStream. Метод CreateText возвращает StreamWriter. Перегруженный метод Open в зависимости от передаваемых параметров может создавать новый файл или открывать существующий для чтения или записи. Возвращаемый объект – объект FileStream. Метод OpenText возвращает StreamReader. Метод OpenRead возвращает объект FileStream. Метод OpenWrite возвращает объект типа FileStream.

Классы File и FileInfo содержат также методы копирования, удаления, перемещения файлов, проверки существования файла, чтения и изменения атрибутов файла (время создания; время последнего обращения; последнее время записи; архивный, скрытый, обычный, системный или временный; сжатый, зашифрованный; только для чтения; файл или каталог).

Программа 4. Пример использования методов класса File и FileInfo.

```
void Main() {
```

```
    File.Delete("file2.txt");          // Удалить файл "file2.txt"
    StreamWriter sw = System.IO.File.CreateText("file.txt");
    sw.Write ("Пусть каждый день твой будет светлым, ");
    sw.WriteLine("приятным, радостным и щедрым!");
    sw.Close();
    File.Move("file.txt", "file2.txt");        // Переименование
    FileInfo fileInfo = new FileInfo("file2.txt");
    Console.WriteLine( "File {0} is {1} bytes in length, created on
{2}",  fileInfo.FullName,fileInfo.Length, fileInfo.CreationTime);
    Console.WriteLine("");
    StreamReader sr = fileInfo.OpenText();
    String s = sr.ReadLine();
    while (s != null)  {
         Console.WriteLine(s);
         s = sr.ReadLine();
    }
    sr.Close();
}
```

Сериализация объектов. Сериализация преобразовывает объекты, такие как классы, структуры и массивы в поток байтов. При преобразовании из последовательной формы в параллельную поток байтов преобразовывается обратно в объекты.

Сериализация может осуществляться в двух форматах: **бинарном и XML-формате**. Пространство имен **System.Runtime. Serialization** используется для сериализации в общей системе типов. Пространство имен **System.Xml.Serialization** используется для сериализации XML-документов.

Чтобы информировать каркас, что класс может быть преобразован в последовательную форму, нужно пометить класс атрибутом **[Serializable]**. Любое поле или свойство, которые не должны быть преобразованы в последовательную форму, могут быть отмечены атрибутом [**NonSerialized**].

```
   [Serializable]    // Объявление сериализации объектов класса
public class Personage{
      public Personage(string name, int age)   {
        this.name = name; this.age = age;
      }
      static int wishes;
```

```
    public string name, status, wealth;
    int age;
    public Personage couple;
  void SaveState(){     //сериализация в бинарном формате
    BinaryFormatter bf = new BinaryFormatter();
    FileStream fs = new FileStream ("State.bin",FileMode.Create,
FileAccess.Write);
    bf.Serialize(fs,this);
    fs.Close();
  }
  void BackState(ref Personage fisher){          //десериализация
    BinaryFormatter bf = new BinaryFormatter();
    FileStream fs = new FileStream ("State.bin",FileMode.Open, FileAc-
cess.Read);
    fisher = (Personage)bf.Deserialize(fs);
    fs.Close();
  }
}
```

РАЗРАБОТКА МНОГООКОННОГО ПРИЛОЖЕНИЯ

Разберем создание MDI (Multiple Document Interface, многодокументный интерфейс) приложений на примере разработки программы «Блокнот».

1. Создайте новое приложение (**программа 5**) и назовите его NotepadC#. Установите следующие свойства формы (табл. 4.1)

Таблица 4.1

Свойства формы NotepadC#

Форма, свойства	Значение
Name	Frmmain
Icon	Путь C:\Program Files\Microsoft Visual Studio 9\Common7\VS2008ImageLibrary....
Text	NotepadC#
WindowState	Maximized

2. Создайте меню приложения со следующими пунктами (рис. 4.1). Для этого вызовите контекстное меню, установив курсор мыши на компоненте menuStrip1, расположенной на панели невидимых компо-

нент, и выберите пункт Edit Items. Каждый пункт главного меню имеет свое окно свойств, в котором задаются значения свойств Name и Text (рис. 4.2). В поле Text перед словом New стоит знак & – так называемый амперсанд, указывающий, что N должно быть подчеркнуто и будет частью встроенного клавиатурного интерфейса Windows. Когда пользователь на клавиатуре нажимает клавишу Alt и затем N, выводится подменю New.

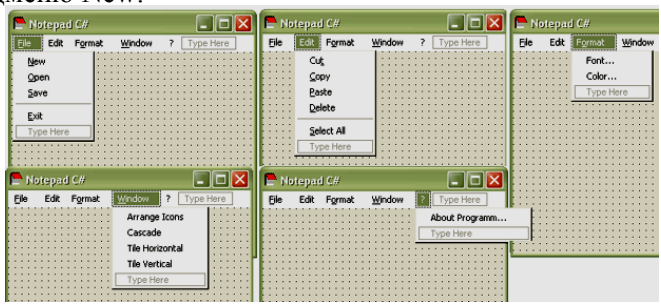


Рис. 4.1. Пункты главного меню приложения NotepadC#

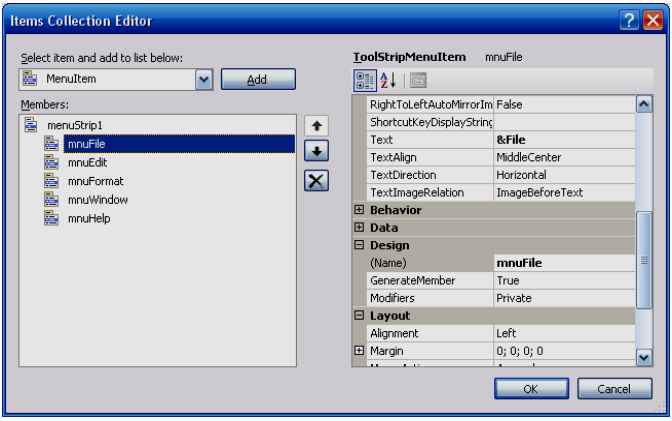


Рис. 4.2. Свойства пункта меню New

Свойства пунктов меню в приложении NotepadC# приводятся в табл. 4.2.

Таблица 4.2

Пункты главного меню приложения NotepadC#

Name	Text	Shortcut
mnuFile	&File	
mnuNew	&New	Ctrl+N
mnuOpen	&Open	Ctrl+O
mnuSave	&Save	Ctrl+S
menuItem5	-	

Окончание табл. 4.2

Name	Text	Shortcut
mnuExit	&Exit	Alt+F4
mnuEdit	&Edit	
mnuCut	Cu&t	Ctrl+X
mnuCopy	&Copy	Ctrl+C
mnuPaste	&Paste	Ctrl+V
mnuDelete	&Delete	Del
mnuSelectAll	&SelectAll	Ctrl+A
mnuFormat	F&ormat	
mnuFont	Font…	
mnuColor	Color…	
mnuWindow	&Window	
mnuArrangeIcons	Arrange Icons	
mnuCascade	Cascade	
mnuTileHorizontal	Tile Horizontal	
mnuTileVertical	Tile Vertical	
mnuHelp	?	
mnuAbout	About Programm...	

3. В MDI-приложениях главная форма содержит в себе несколько документов, каждый из которых является холстом в графических программах или полем для текста в редакторах. В окне Solution Explorer щелкаем правой кнопкой на имени проекта и в появившемся контекстном меню выбираем Add/New Item/ Windows Form. В появившемся окне называем форму – blank.h. В нашем проекте появилась новая форма – будем называть ее **дочерней**. В режиме дизайна перетаскиваем на нее элемент управления RichTextBox, размер содержимого текста в нем не ограничен 64 Кб; кроме того, RichTextBox позволяет редактировать цвет текста, добавлять изображения. Свойству Dock этого элемента устанавливаем значение Fill.

Переходим в режим дизайна формы frmmain и устанавливаем свойству **IsMdiContainer** значение **true**. Цвет формы при этом становится темно-серым. Новые документы будут у нас появляться при нажатии пункта меню New, поэтому дважды щелкаем в этом пункте и переходим в код.

При создании нескольких документов будем называть их ДокументN, где N – номер документа. Переключаемся в код формы blank, и в классе blank объявляем поле

public String DocName;

Переключаемся в код формы frmmain и в классе frmmain объявляем переменную **private int openDoc;**

Присваиваем переменной DocName часть названия по шаблону, в который включен счетчик числа открываемых документов, затем это значение передаем свойству Text создаваемой формы frm:

```
private void mnunew_Click(object  sender, EventArgs  e) {
    frm = new blank();
    frm.DocName = "Документ " + ++openDoc;
    frm.Text = frm.DocName;
    frm.MdiParent = this;
    frm.Show();
}
```

4. Для каждого пункта меню пишем обработчики событий, выполняющие соответсвтвующие пункту действия.

Перечисление MdiLayout. При работе с несколькими документами в MDI-приложениях удобно упорядочивать их на экране. В пункте меню Window реализуем процедуру выравнивания окон. Создаем обработчики:

```
private void mnuArrangeIcons_Click(object sender, EventArgs e){
  this.LayoutMdi(MdiLayout.ArrangeIcons);
}
private void mnuCascade_Click(Object sender, EventArgs e)   {
    this.LayoutMdi(MdiLayout.Cascade);
}
private void mnuTileHorizontal_Click(object sender, EventArgs e) {
    this.LayoutMdi(MdiLayout.TileHorizontal);
}
private void mnuTileVertical_Click(object sender, EventArgs e)   {
```

```
        this.LayoutMdi(MdiLayout.TileVertical);
    }
```

Метод LayoutMdi содержит перечисление MdiLayout, содержащее четыре члена. ArrangeIcons переключает фокус на выбранную форму, в свойстве MdiList пункта меню ArrangeIcons устанавливаем также значение true. При открытии нескольких новых документов окна располагаются каскадом, их можно расположить горизонтально – значение TileHorizontal или вертикально – значение TileVertical.

Вырезание, копирование и вставка текстовых фрагментов. При создании нового документа он сразу будет занимать всю область главной формы. Для этого установим свойство WindowState формы blank Maximized. Создадим обработчики для стандартных операций вырезания, копирования и вставки. Элемент управления RichTextBox имеет свойство SelectedText, которое содержит выделенный фрагмент текста. Оно будет использоваться при работе с текстом. В коде формы blank объявляем переменную, в которой будет храниться буферизованный фрагмент текста: **private String BufferText**;

Далее создаем соответствующие методы:

```
    public void Cut() {              // Вырезание текста
        this.BufferText = richTextBox1.SelectedText;
        richTextBox1.SelectedText = "";
    }
    public void Copy() {                    // Копирование текста
        this.BufferText = richTextBox1.SelectedText;
    }
    public void Paste() {                   // Вставка
        richTextBox1.SelectedText = this.BufferText;
    }
    // Выделение всего текста – используем свойство SelectAll элемента управления RichTextBox
    public void SelectAll() {
        richTextBox1.SelectAll();
    }
    public void Delete() {          // Удаление
        richTextBox1.SelectedText = "";
        this.BufferText = "";
```

```
}
```

Переключаемся в режим дизайна формы и создаем обработчики для пунктов меню:

```
private void mnucut_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Cut();
}
private void mnucopy_Click(object sender, EventArgs e) {
    blank frm = (blank)(this.ActiveMdiChild);
    frm.Copy();
}
private void mnuselectAll_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.SelectAll();
}
private void mnupaste_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Paste();
}
private void mnudelete_Click(object sender, EventArgs e) {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Delete();
}
```

Свойство **ActiveMdiChild** переключает фокус на текущую форму, если их открыто несколько, и вызывает один из методов, определенных в дочерней форме. Теперь можно выполнять все стандартные операции с текстом.

Контекстное меню. Дублирует некоторые действия основного меню. Добавим элемент управления contextMenuStrip из окна ToolBox на форму blank. Добавляем пункты контекстного меню точно так же, как мы это делали для главного.

Свойства Text и Shortcut пунктов меню оставляем прежними. Если мы установим затем для свойства ShowShortcut значение false, то сочетания клавиш будут работать, но в самом меню отображаться не будут.

Свойства Name будут следующими: для пункта Cut – cmnuCut, для Copy – cmnuCopy и т. д.

В обработчике пунктов просто вызываем соответствующие методы:

```
private void cmnuCut_Click(object sender, EventArgs e) {
    Cut();
}
private void cmnuCopy_Click(object sender, EventArgs e) {
    Copy();
}
private void cmnuPaste_Click(object sender, EventArgs e)    {
    Paste();
}
private void cmnuDelete_Click(object sender, EventArgs e) {
    Delete();
}
private void cmnuSelectAll_Click(object sender, EventArgs e) {
    SelectAll();
}
```

Необходимо определить, где будет появляться контекстное меню. Элемент RichTextBox, так же как и формы frmmain и blank, имеет свойство ContextMenuStrip, где мы и указываем contextMenuStrip1, поскольку нам нужно отображать меню именно в текстовом поле. Запускаем приложение – теперь в любой точке текста доступно меню.

5. Для пункта Format создать обработчики выбора цвета и шрифта самостоятельно.

ДИАЛОГОВЫЕ ОКНА

Выполняя различные операции с документом – открытие, сохранение, печать, предварительный просмотр, мы сталкиваемся с соответствующими диалоговыми окнами. Разработчикам .NET не приходится заниматься созданием окон стандартных процедур: элементы **OpenFileDialog, SaveFile Dialog, ColorDialog, PrintDialog** содержат уже готовые операции.

OpenFileDialog. Добавьте на форму frmmain элемент управления OpenFileDialog из окна панели инструментов ToolBox.

Свойство FileName задает название файла, которое будет находиться в поле "Имя файла:" при появлении диалога. Свойство Filter задает ограничение файлов, которые могут быть выбраны для открытия. Через вертикальную разделительную линию можно задать смену типа расширения, отображаемого в выпадающем списке "Тип файлов". Зададим Text Files (*.txt)|*.txt|All Files(*.*)|*.*, что означает обзор либо текстовых файлов, либо всех. Свойство InitialDirectory позволяет задать директорию, откуда будет начинаться обзор. Если свойство не установлено, исходной директорией будет рабочий стол.

Для работы с файловыми потоками в коде формы blank подключаем пространство имен System.IO. Создаем метод Open:

```
public void Open(String OpenFileName)  {
    if (OpenFileName == null) {    return;    }
    else  {
        StreamReader sr = new StreamReader(OpenFileName);
            richTextBox1.Text = sr.ReadToEnd();
            sr.Close();
            DocName = OpenFileName;
} }
```

Добавим обработчик пункта меню Open формы frmmain:

```
private void mnuopen_Click(object  sender, EventArgs  e) {
// задание доступных расширений файлов программно
openFileDialog1.Filter = "Text Files (*.txt)|*.txt|All Files(*.*)|*.*";
if (openFileDialog1.ShowDialog() == .DialogResult.OK) {
    blank frm = new blank();
    frm.Open(openFileDialog1.FileName);
    frm.MdiParent = this;      //Указываем родительскую форму
//Присваиваем переменной DocName имя открываемого файла
    frm.DocName = openFileDialog1.FileName;
//Свойству Text формы присваиваем переменную DocName
    frm.Text = frm.DocName;
    frm.Show();
} }
```

Для корректного чтения кириллицы текст в блокноте должен быть сохранен в кодировке Unicode.

SaveFileDialog. Для сохранения файлов добавляем на форму frmmain элемент управления saveFileDialog1. Свойства этого диалога такие же, как у OpenFileDialog. Переходим в код формы blank и создаем метод Save:

```
public void Save(String SaveFileName)  {
        if (SaveFileName == null) {     return;   }
        else  {
           StreamWriter sw = new StreamWriter(SaveFileName);
                sw.WriteLine(richTextBox1.Text);
                sw.Close();
        //Устанавливаем имя документа
                DocName = SaveFileName;
     }}
```

Добавляем обработчик пункта меню Save формы frmmain:

```
private void mnuSave_Click(object  sender, EventArgs  e)   {
    saveFileDialog1.Filter = "Text Files (*.txt)|*.txt|All
Files(*.*)|*.*";
      if (saveFileDialog1.ShowDialog() == .DialogResult.OK)  {
        blank frm = (blank)(this.ActiveMdiChild);
        frm.Save(saveFileDialog1.FileName);
        frm.MdiParent = this;
        frm.DocName = saveFileDialog1.FileName;
        frm.Text = frm.DocName;
}}
```

При сохранении внесенных изменений в уже сохраненном файле вместо его перезаписи вновь появляется окно SaveFileDialog. Изменим программу так, чтобы можно было сохранять и перезаписывать файл. В конструкторе формы frmmain после InitializeComponent отключим доступность пункта меню Save: **mnuSave.Enabled = false;**

Переключаемся в режим дизайна формы frmmain и добавляем пункт меню **Save As** после пункта Save. Устанавливаем следующие свойства этого пункта: Name – mnuSaveAs, Shortcut – Ctrl+Shift+S, Text Save &As. В обработчике Save As вставляем вырезанный обработчик

пункта Save и добавляем включение доступности Save: **mnuSave.Enabled = true;**

Сохранять изменения требуется как в только что сохраненных документах, так и в документах, созданных ранее и открытых для редактирования. Поэтому добавим в метод Open включение доступности пункта меню Save:

```
private void mnuOpen_Click(object sender, EventArgs e) {
    mnuSave.Enabled = true;
}
```

В обработчике пункта Save добавим простую перезапись файла – вызов метода Save формы blank:

```
private void mnuSave_Click(object  sender, EventArgs  e)  {
    blank frm = (blank) (this.ActiveMdiChild);
    frm.Save(frm.DocName);
}
```

Теперь, если мы работаем с несохраненным документом, пункт Save неактивен, после сохранения он становится активным и, кроме того, работает сочетание клавиш Ctrl+S. Можно сохранить копию текущего документа, вновь воспользовавшись пунктом меню Save As.

Сохранение файла при закрытии формы. Всякий раз, когда мы закрываем документ Microsoft Word, в который внесли изменения, появляется окно предупреждения, предлагающее сохранить документ. Добавим аналогичную функцию в наше приложение. В классе blank создаем переменную, которая будет фиксировать сохранение документа:

public bool IsSaved = false;

В обработчик методов Save и Save As формы frmmain добавляем изменение значения этой переменной:

```
private void mnuSave_Click(object sender, EventArgs e) {
    …
    frm.IsSaved = true;
}
private void mnuSaveAs_Click(object sender, EventArgs e) {
    …
    frm.IsSaved = true;
}
```

Переходим в режим дизайна формы blank, и в окне свойств переключаемся на события формы, щелкнув на значок с молнией. В поле события FormClosed дважды щелкаем и переходим в код:

```
private void blank_FormClosed(object sender,
FormClosedEventArgs e) {
        if(IsSaved ==true)
          if(MessageBox.Show("Do you want save changes in " +
this.DocName + "?", "Message", MessageBoxButtons.YesNo,
                MessageBoxIcon.Question) == .DialogResult.Yes) {
                this.Save(this.DocName);
} }
```

OpenFileDialog и SaveFileDialog для SDI-приложений. При создании MDI-приложений приходится разделять код для открытия и сохранения файлов, как мы делали для приложения NotepadC#. В случае SDI-приложений весь код будет находиться в одном обработчике. Создаем новое приложение, называем его TextEditor. На форме размещаем элемент управления TextBox и устанавливаем следующие свойства (табл. 4.3).

Таблица 4.3

Свойства элемента управления TextBox

TextBox, свойство	Значение
Name	txtBox
Dock	Fill
Multiline	true
Text	Да

Добавляем на форму элемент menuStrip1, в котором будет всего три пункта – File, Open и Save (свойства пунктов см. в табл. 4.2). Из окна ToolBox перетаскиваем элементы OpenFileDialog и SaveFileDialog – свойства этих элементов в точности такие же, как и у диалогов приложения NotepadC#. Переходим в код формы.

Добавляем обработчик для пункта меню Open:

```
private void mnuOpen_Click(object sender, EventArgs e){
    openFileDialog1.ShowDialog();
    String fileName = openFileDialog1.FileName;
```

```
        FileStream filestream= File.Open(fileName, FileMode.Open, FileAccess.Read);
        if(filestream != null) {
         StreamReader streamreader = new StreamReader(filestream);
                txtBox.Text = streamreader.ReadToEnd();
                filestream.Close();
    } }

    Добавляем обработчик для пункта меню Save:
    private void mnuSave_Click(object sender, EventArgs e) {
        saveFileDialog1.ShowDialog();
        String fileName=saveFileDialog1.FileName;
        FileStream filestream = File.Open(fileName, FileMode.Create, FileAccess.Write);
        if(filestream != null) {
         StreamWriter streamwriter = new StreamWriter(filestream);
                streamwriter.Write(txtBox.Text);
                streamwriter.Flush();
                filestream.Close();
    } }
```

ВОПРОСЫ К ЗАЩИТЕ ЛАБОРАТОРНОЙ РАБОТЫ

1. Что такое поток? Какой класс является родоначальником всех потоков?
2. Какие бывают потоки?
3. В каких форматах можно сохранять файловые потоки?
4. Режимы работы с файлом.
5. Основные методы работы с файлом.
6. Какие возможности имеют классы **File**, **FileInfo?**
7. Что такое сериализация? Для чего она применяется?
8. Что такое десериализация? Для чего она применяется?
9. Как задать сериализацию объектов класса?
10. В каких форматах можно сериализовать данные?
11. Как исключить некоторые свойства объекта при сериализации?
12. Как десериализовать объект?
13. Что такое MDI-приложение? Как создать такое приложение?
14. Что такое контекстно-зависимое меню? Как создать контекстно-зависимое меню?

15. Какая компонента позволяет отображать на форме рисунок?

16. Какая компонента служит для ввода текста и многострочного текста на форме?

17. Как создать и вызвать стандартные диалоговые окна: подтверждение действия, сохранение в файл, загрузка из файла?

СПИСОК ЛИТЕРАТУРЫ

1. *Эндрю Троелсен.* C# и платфо ЕТ. – СПб.: Питер, 2005. – 796 с.

2. *Том Арчер.* Основы C#. – М. горговый дом «Русская редакция», 2001.

3. *Лабор В.В.* Си Шарп: Создание приложений для Windows/ В.В. Лабор. – Минск: Харвест, 2003. – 384 с.

4. *Петцольд Чарльз.* Программирование с использованием Microsoft Windows Forms : пер. с англ. – М. : Русская редакция ; СПб. : Питер, 2006. – 410 с. ил.

5. Курс лекций «Создание Windows-приложений на основе Visual C# http://www.intuit.ru/department/pl/visualcsharp.

ТЕХНОЛОГИЯ РАЗРАБОТКИ
ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ НА ЯЗЫКЕ C#
В СРЕДЕ Visual Studio.Net

Методические указания

Редактор *Л.Н. Ветчакова*
Выпускающий редактор *И.П. Брованова*
Компьютерная верстка *С.И. Ткачева*

Подписано в печать 27.10.2010. Формат 60 × 84 1/16. Бумага офсетная. Тираж 100 экз.
Уч.-изд. л. 4,88. Печ. л. 5,25. Изд. № 224. Заказ № . Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630092, г. Новосибирск, пр. К. Маркса, 20