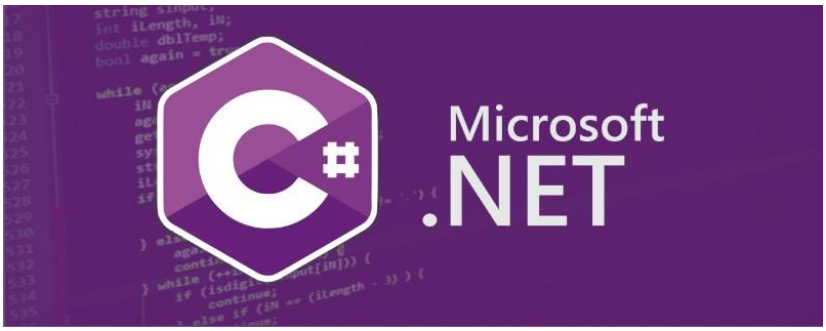
Версия документа 4.0

ФГБОУ ВО «Воронежский государственный
технический университет»

Кафедра систем автоматизированного проектирования
и информационных систем

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторным работам по дисциплине «Программирование на платформе .NET Framework» для студентов направления 09.03.02 «Информационные системы и технологии» очной формы обучения



Воронеж 2021

Составители: Д.В. Иванов
УДК 681.38+681.3

Методические указания к лабораторным работам по дисциплине «Программирование на платформе .NET Framework» для студентов направления 09.03.02 «Информационные системы и технологии» очной формы обучения / ФГБОУ ВО «Воронежский государственный технический университет»; сост. Д.В. Иванов. Воронеж, 2021. 147 с.

Методические указания содержат краткие теоретические и практические сведения об основных конструкциях языках SQL и их практического применения.

Методические указания подготовлены в электронном виде в текстовом редакторе MS Word 2003 и содержатся в файле .NET 4.0.pdf.

Табл. 11. Библиогр.: 4 назв.

Рецензент д-р техн. наук, проф. К.А. Разинкин

Ответственный за выпуск зав. кафедрой
д-р техн. наук, проф. Я.Е. Львович

Издается по решению редакционно-издательского совета Воронежского государственного технического университета

Лабораторная работа № 1-2

«Разработка пользовательских интерфейсов для программирования ввода-вывода и основных операторов языка C# на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ разработки визуальных пользовательских интерфейсов для программирования ввода-вывода и основных операторов языка C# на платформе .NET Framework.

Краткие теоретические сведения

Программирование

Программирование — процесс создания компьютерных программ.

В узком смысле (так называемое кодирование) под программированием понимается написание инструкций (программ) на конкретном языке программирования (часто по уже имеющемуся алгоритму — плану, методу решения поставленной задачи). Соответственно, люди, которые этим занимаются, называются программистами (на профессиональном жаргоне — кодерами), а те, кто разрабатывают алгоритмы — алгоритмистами, специалистами предметной области, математиками.

В более широком смысле под программированием понимают весь спектр деятельности, связанный с созданием и поддержанием в рабочем состоянии программ — программного обеспечения ЭВМ. Иначе это называется «программная инженерия» («инженерия ПО»). Сюда входят анализ и постановка задачи, проектирование программы, построение алгоритмов, разработка структур данных, написание текстов программ, отладка и тестирование программы (испытания программы), документирование, настройка (конфигурирование), доработка и сопровождение.

Программирование для ЭВМ основывается на использовании языков программирования, на которых записывается программа. Чтобы программа могла быть понята и исполнена ЭВМ, требуется специальный инструмент — транслятор.

В настоящее время активно используются интегрированные среды разработки, включающие в свой состав также редактор для ввода и редактирования текстов программ, отладчики для поиска и устранения ошибок, трансляторы с различных языков программирования, компоновщики для сборки программы из нескольких модулей и другие служебные модули.

Текстовый редактор среды программирования может иметь специфичную функциональность, такую как индексация имен, отображение документации, средства визуального создания пользовательского интерфейса. С помощью текстового редактора программист производит набор и редактирования текста создаваемой программы, который называют исходным кодом. Язык программирования определяет синтаксис и изначальную семантику исходного кода. Компилятор преобразует текст программы в машинный код, непосредственно исполняемый электронными компонентами компьютера. Интерпретатор создаёт виртуальную машину для выполнения программы, которая полностью или частично берёт на себя функции исполнения программ.

Язык C#

C# (произносится си шарп) — объектно-ориентированный язык программирования. Разработан в 1998—2001 годах группой инженеров под руководством Андерса Хейлсберга в компании Microsoft как язык разработки приложений для платформы Microsoft .NET Framework и впоследствии был стандартизирован как ECMA-334 и ISO/IEC 23270.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения,комментарии в формате XML.

Переняв многое от своих предшественников — языков C++, Pascal, Модула, Smalltalk и в особенности Java — C#, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем, например, C# в отличие от C++ не поддерживает множественное наследование классов (между тем допускается множественное наследование интерфейсов).

Тип данных

Для обработки ЭВМ данные представляются в виде величин и их совокупностей. С понятием величины связаны такая важная характеристика, как ее тип.

Тип определяет:

– возможные значения переменных, констант, функций, выражений, принадлежащих к данному типу;

– внутреннюю форму представления данных в ЭВМ;

– операции и функции, которые могут выполняться над величинами, принадлежащими к данному типу.

В языке программирования C# типы данных подразделяются на категории:

– значение;
– ссылка.

Ниже приведена таблица основных типов данных языка C#.

Название	Ключевое слово	Тип .NET	Диапазон	Описание	Размер, битов
Логический тип	bool	Boolean	true, false		
Целые типы	sbyte	SByte	-128..127	Со знаком	8
	byte	Byte	0..255	Без знака	8
	short	Int16	-32768..32767	Со знаком	16
	ushort	UInt16	0..65535	Без знака	16
	int	Int32	-2*10^9..2*10^9	Со знаком	32
	uint	UInt32	0..4*10^9	Без знака	32
	long	Int64	-9*10^18..9*10^18	Со знаком	64
	ulong	UInt64	0..18*10^18	Без знака	64
Символьный тип	char	Char	U+0000..U+ffff	Unicode-символ	16
Вещественные типы	float	Single	1.5*10^-45..3.4*10^38	7 цифр	32
	double	Double	5.0*10^-324..1.7*10^308	15-16 цифр	64
Финансовый тип	decimal	Decimal	1.0*10^-28..7.9*10^28	28-29 цифр	128
Строковый тип	string	String	Длина ограничена объемом доступной памяти	Строка из Unicode-символов	
Тип object	object	Object	Можно хранить все что угодно	Всеобщий порядок	

Типы значения

Встроенные типы

Эти типы, которые нам любезно предоставляет Visuai Studio - System.Double (double), System.Byte (byte), System.Int32 (int), System.Char (char), System.Single (float). В скобках указаны псевдонимы - сокращенное название типов данных в C sharp (си шарп). Удобнее все-таки в коде писать именно псевдонимы, чем полные названия типов.

Перечислимые

Это сборка необходимых нам символов. Лучше всего в понимании перечислимого поможет пример:

```
public enum typeTraine { skTrain, pasTrain, tTrain };
```

Т.е. в этом примере мы создаем перечислимое видов поездов. (skTrain - скоростной, pasTrain - пассажирский, tTrain - транспортный)

Обращение к элементу перечислимого происходит так же, как мы обращаемся к элементу класса - т.е. через точку:

```
typeTraine ob = typeTraine.pasTrain;
Console.WriteLine("{0}", ob);
```

На экране будет слово – pasTrain

Пользовательские типы

А вот понятие пользовательские типы вам уже знакомо с языков C/C++. Т.е. это тип который создает сам пользователь, при чем пользовательский тип является структурой. И поэтому будет начинаться с ключевого слова - struct. Вот пример:

```
    struct fleshka {
  int color;
  int volume;
  public int fleshka(int _color, int _volume)
  {
    color = _color;
    volume =_volume;
  }
}
```

В отличие от языка C++ поля структуры являются по умолчанию закрытыми.

К ссылочным типам относятся:

В отличие от типов значения, ссылочные типы в стеке хранят не сами значения типов, а ссылки на них. Значения хранятся совершенно в другой области памяти, которая называется кучей.

Ссылочные типы, так же как и значимые бывают:

Встроенные типы

Просто приведу вам примеры встроенных ссылочных типов. Самым главным является тип System.Object. Этот тип данных в C sharp (си шарп) является, чуть ли не самым важным, так как в него можно преобразовать любой тип, будь то значимый или ссылочный.

System.String - этот тип так же является весьма распространенным - все строковые значения обычно хранятся именно в этом типе данных. Ну и для хранения массивов, которые и являются сами по себе ссылочным типом, существует класс system.Array. Так же для обработки различных исключительных ситуаций в Visual Studio предусмотрен класс System.Exception.

Пользовательские типы

Тут на самом деле говорить многого не надо. Знаете, чем отличается класс от структуры в C sharp (си шарп)? Ответом на этот вопрос и является тема "ссылочные | пользовательские | значимые типы данных в C sharp (си шарп)".

Если мы пишем ключевое слово класс, то у нас по умолчанию создается ссылочный тип. Т.е. в том примере, где мы создавали структуру fleshka, достаточно поменять ключевое слово struct на class и вуаля - ссылочный пользовательский тип готов.

Иными словами, если пользователь создает значимый тип - то нужно создавать структуру. А при создании ссылочного типа просто нужно создать класс.

Наряду с переменными в программах используются и константы. Константы представляют собой неизменные значения, известные во время компиляции и неизменяемые на протяжении времени существования программы. Константы объявляются с модификатором const. Только встроенные типы C# (за исключением System::Object) могут быть объявлены как const. Список встроенных типов см. в разделе Таблица встроенных типов (Справочник по C#). Определяемые пользователем типы, включая классы, структуры и массивы, не могут быть const. Для создания класса, структуры или массива, которые инициализируются один раз во время выполнения (например, в конструкторе) и после этого не могут быть изменены, используется модификатор readonly.

Структура программы

Для упорядочения и оформления кода в языке C# используются классы. В действительности весь выполняемый код C# должен содержаться в классе, что справедливо и для короткой программы типа "Hello World!". Ниже приведен полный текст программы, отображающей в окне консоли сообщение "Hello World!".

```
using System;
namespace HelloWorld
{
   class Hello
   {
      static void Main()
      {
         System.Console.WriteLine("Hello World!");
         System.Console.WriteLine("Press any key to exit.");
         System.Console.ReadKey();
      }
   }
}
```

При создании консольного приложения с Visual C#, экспресс-выпуск, в первой линии в редакторе кода содержится директива using с перечислением нескольких пространств имен .NET Framework. Пространство имен позволяет, в некотором смысле, сгруппировать вместе классы и структуры, что ограничивает их область действия и позволяет избежать конфликта имен с другими классами и структурами. При создании программы в Visual C# Express пространство имен создается автоматически. Для использования в программе классов из других пространств имен необходимо указать их с директивой using. При создании нового приложения наиболее часто используемые пространства имен .NET Framework включены в список по умолчанию. При использовании классов из других пространств имен в библиотеке классов необходимо добавить директиву using для пространства имен к исходному файлу.

В C# классы используются для оформления кода: весь выполняемый код C# должен содержаться в классе.

В программе на C# должен присутствовать метод Main, в котором начинается и заканчивается управление. В методе Main создаются объекты и выполняются другие методы. Метод Main является статическим методом, расположенным внутри класса или структуры.

Оператор присваивания

Основное преобразование данных, выполняемое компьютером, - присвоение переменной нового значения.

Общий вид оператора присваивания:

Имя_переменной = арифметическое выражение;

При выполнении оператора присваивания рассматривается арифметические выражения, из ячеек оперативной памяти, соответствующих стоящим там именам, вносятся в процессор значения и выполняется указанные действия над данными. Полученный результат записывается в ячейку памяти, имя которой указано слева от знака присваивания.

Например:

```
x=3.14;          //Переменной x присвоить значение 3.14
a=b+c  ;
//Из ячеек b и c считываются заранее помещенные туда данные, вычисляется сумма, результат записывается в ячейку a
i=i++;  //Значение переменной увеличивается на единицу
```

Для типов переменной слева и арифметического выражения справа от знака присваивания существуют ограничения:

1) если переменная вещественного типа, то арифметическое выражение может быть как целого, так и вещественного типа, т. е. содержать либо целые переменные и допустимые для них операции, либо вещественные, либо и те, и другие (тогда выражение преобразуется к вещественному типу);

2) если переменная слева целого типа, то арифметическое выражение только целочисленное.

Это означает, что можно, например, вещественной переменной присвоить целое значение.

Организация консольного ввода/вывода

Ввод данных

Для того чтобы получить данные, вводимые вручную (то есть с консоли), применяются команды

<строковая переменная> = Console.ReadLine()

Пример:

```
int x; double y; string s;
s = Console.ReadLine();
x = Convert.ToInt32(s);
y = Convert.ToDouble(Console.ReadLine());
```

Вывод данных

Для того чтобы вывести на экран какое-либо сообщение, воспользуйтесь процедурой Console.Write или Console.WriteLine

Первая из них, напечатав на экране все, о чем ее просили, оставит курсор в конце выведенной строки, а вторая переведет его в начало следующей строчки.

```
Console.WriteLine(s);// переменная
Console.WriteLine(55.3);// константа
Console.WriteLine(y*3+7);// выражение

Console.Write(z);// переменная
Console.Write(-5.3);// константа
Console.Write(i*3+7/j);// выражение
```

Выражения

Выражение задает правило вычисления некоторого значения. Выражение состоит из констант, переменных, знаков операций и скобок.

Математические операции

В таблице приведены основные математические операции.

Символ операции	Название операции	Пример
*	Умножение	2*3 (результат: 6)
/	Деление	30/2 (результат: 1.5E+01)
+	Сложение	2+3 (результат: 5)
-	Вычитание	5-3 (результат: 2)
%	Остаток от деления	5 % 2 (результат: 1)

Логические операции

Над логическими аргументами в C# определены следующие операции:

! - логическое отрицание ("НЕ")

& - логическое умножение ("И")

|| - логическое сложение ("ИЛИ")

^ - логическое "Исключающее ИЛИ"

Операции отношения

К операциям отношения в C# относятся такие операции, как:

> - больше

< - меньше

== - равно

!= - не равно

>= - больше или равно

<= - меньше или равно

В операциях отношения могут принимать участие не только числа, но и символы, строки, множества и указатели.

Основные математические функции

Кроме переменных и констант, первичным материалом для построения выражений являются функции. Большинство их в проекте будут созданы самим программистом, но не обойтись и без встроенных функций. Умение работать в среде Visual Studio предполагает знание встроенных возможностей этой среды, знание возможностей каркаса Framework .Net, пространств имен, доступных при программировании на языке C#, а также соответствующих встроенных классов и функций этих классов. Рассмотрим еще один класс - класс Math, содержащий стандартные математические функции, без которых трудно обойтись при построении многих выражений. Этот класс содержит два статических поля, задающих константы E (число e) и PI (число пи), а также 23 статических метода. Методы задают:

- тригонометрические функции - Sin, Cos, Tan;

- обратные тригонометрические функции
- ASin, ACos, ATan, ATan2(sinx, cosx);

- гиперболические функции - Tanh, Sinh, Cosh;

- экспоненту и логарифмические функции - Exp, Log, Log10;

- модуль, корень, знак - Abs, Sqrt, Sign;

- функции округления - Ceiling, Floor, Round;

- минимум, максимум, степень, остаток - Min, Max, Pow, lEEEReminder.

Оператор условного перехода if

Для организации условного ветвления язык C# унаследовал от C и C++ конструкцию if...else. Ее синтаксис должен быть интуитивно понятен для любого, кто программировал на процедурных языках:

```
if (условие)
  оператор (операторы);
else
  оператор (операторы);
```

Если по каждому из условий нужно выполнить более одного оператора, эти операторы должны быть объединены в блок с помощью фигурных скобок {...}.

Оператор выбора switch

Вторым оператором выбора в C# является оператор switch, который обеспечивает многонаправленное ветвление программы. Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы. Несмотря на то что многонаправленная проверка может быть организована с помощью последовательного ряда вложенных операторов if, во многих случаях более эффективным оказывается применение оператора switch. Этот оператор действует следующим образом. Значение выражения последовательно сравнивается с константами выбора из заданного списка. Как только будет обнаружено совпадение с одним из условий выбора, выполняется последовательность операторов, связанных с этим условием. Ниже приведена общая форма оператора switch:

```
switch(выражение) {
    case константа1:
        последовательность операторов
    break;
    case константа2:
        последовательность операторов
    break;
    case константа3:
        последовательность операторов
    break;

...

    default:
        последовательность операторов
    break;
}
```

Структуры

Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

Типы структуры имеют семантики значений. То есть переменная типа структуры содержит экземпляр этого типа. По умолчанию значения переменных копируются при назначении, передаче аргумента в метод и возврате результата метода. В случае переменной типа структуры копируется экземпляр типа. Дополнительные сведения см. в разделе Типы значений.

Как правило, типы структуры используются для проектирования небольших ориентированных на данные типов, которые предоставляют минимум поведения или не предоставляют его вовсе. Например, платформа .NET использует типы структуры для представления числа (как целого, так и вещественного), логического значения, символа Юникода, экземпляра времени. Если вы сконцентрированы на поведении типа, рекомендуется определить класс. Типы классов имеют семантики ссылок. То есть переменная типа класса содержит ссылку на экземпляр этого типа, а не сам экземпляр.

Тип структуры представляет собой тип значения, который может инкапсулировать данные и связанные функции. Для определения типа структуры используется ключевое слово struct.

Пример:

```
struct User
{
   public string name;
   public int age;

   public void DisplayInfo()
   {
      Console.WriteLine($"Name: {name}  Age: {age}");
   }
}
```

Как и классы, структуры могут хранить состояние в виде переменных и определять поведение в виде методов. Так, в данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод DisplayInfo для вывода информации о человеке.

Используем эту структуру в программе:

```
using System;

namespace HelloApp
{
    struct User
    {
        public string name;
        public int age;

        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            User tom;
            tom.name = "Tom";
            tom.age = 34;
            tom.DisplayInfo();

            Console.ReadKey();
        }
    }
}
```

В данном случае создается объект tom. У него устанавливаются значения глобальных переменных, и затем выводится информация о нем.

Как и класс, структура может определять конструкторы. Но в отличие от класса нам не обязательно вызывать конструктор для создания объекта структуры.

Однако если мы таким образом создаем объект структуры, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры.

В отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении.

Задание на лабораторную работу

Задание 1

Вариант	№	Задания
Вариант № 1	1	Даны x, y, z . Вычислить a, b, если:
	2	Вычислить площадь поверхности и объем правильной пирамиды, в основании которой квадрат со стороной a и высота h.
	3	Из трех чисел определить минимальное, а затем расположить их в порядке убывания
	4	Написать программу, которая по номеру машины выводит фамилию студента, сидящего за ней.
Вариант № 2	1	Даны x, y, z . Вычислить a, b, если:
	2	Треугольник задан координатами (x1, y1), (x2, y2), (x3, y3) своих вершин. Вычислить радиус окружности, вписанной в треугольник.
	3	Составить программу вычисления модуля |5x - 4|.
	4	В зависимости от времени года "весна", "лето", "осень", "зима" определить погоду "тепло", "жарко", "холодно", "очень холодно".
Вариант № 3	1	Даны x, y, z . Вычислить a, b, если:
	2	Ромб задан координатами трех вершин (x1, y1), (x2, y2), (x3, y3). Вычислить площадь и периметр ромба.
	3	В какой четверти координатной плоскости находится точка с координатами x, y (xy<>0).
	4	Составить программу вычисления значения функции
Вариант № 4	1	Даны x, y, z . Вычислить a, b, если:
	2	Вычислить время падения тела с высоты H с начальной скоростью V.
	3	Определить, имеет ли квадратное уравнение с коэффициентами a, b и c решение.
	4	Составить программу вычисления площадей различных геометрических фигур
Вариант № 5	1	Даны x, y, z . Вычислить a, b, если:

	2	Треугольник задан координатами (x1, y1), (x2, y2), (x3, y3) своих вершин. Найти периметр и площадь треугольника.
	3	Из трех чисел определить максимальное, а затем расположить их в порядке убывания
	4	По номеру дня недели (1,2,3,4,5,6,7) указать название этого дня. Указать рабочие и выходные дни.
Вариант № 6	1	Даны x, y, z . Вычислить a, b, если:
	2	Найти площадь кольца, внутренний радиус которого равен 20, внешний - заданному числу R > 20.
	3	Составить программу вычисления корня $\sqrt{7-5x}$
	4	Даны три числа a, b, c, удовлетворяющие аксиоме треугольника и число p. Выполнить следующие действия: если p=1 - найти периметр треугольника; если p=2 - найти площадь треугольника; если p=3 - найти угол a. Иначе напечатать слово "треугольник".
Вариант № 7	1	Даны x, y, z . Вычислить a, b, если:
	2	Известно, что точки с координатами (x1, y1), (x2, y2, (x3, y3) являются тремя вершинами некоторого параллелограмма. Найти координаты четвертой вершины.
	3	Даны три действительных числа. Определить, что больше, сумма или произведение этих чисел.
	4	Составить программу вычисления значения функции
Вариант № 8	1	Даны x, y, z . Вычислить a, b, если:
	2	Вычислить длину окружности, площадь круга и объем шара одного радиуса.
	3	Из трех чисел определить максимальное, а затем расположить их в порядке возрастания
	4	Составить программу вычисления площадей различных геометрических фигур
Вариант № 9	1	Даны x, y, z . Вычислить a, b, если:
	2	По длинам двух сторон треугольника и углу между ними найти длину третьей стороны и площадь треугольника.
	3	Дана точка M(x, y). Проверить, принадлежит ли точка окружности единичного радиуса.

	4	В зависимости от введённого символа L, S, V программа должна вычислять длину окружности; площадь круга; объём цилиндра.
Вариант № 10	1	Даны x, y, z . Вычислить a, b, если:
	2	Треугольник задан координатами (x1, y1, (x2, y2, (x3, y3) своих вершин. Вычислить радиус окружности, описанной около треугольника.
	3	Даны два числа. Если они не равны, то найти их сумму и произведение.
	4	Напишите программу, которая по введенному числу из промежутка 0..24, определяет время суток.
Вариант № 11	1	Даны x, y, z . Вычислить a, b, если:
	2	Вычислить площадь треугольника по всем известным формулам. Длины сторон заданы тремя числами a, b и c.
	3	Вывести сообщение о количестве корней квадратного уравнения и найти эти корни.
	4	Написать программу преобразования цифр в слова.

Формулы для задания № 1 (по вариантам)

1.

$$a = \frac{\sqrt{|x-1|} - \sqrt{|y|}}{1 + \frac{x^2}{2} + \frac{y^2}{4}}$$

$$b = x(arctg(z) + e)$$

2.

$$a = \frac{3 + e^2}{1 + x^2|y - tg(z)|}$$

$$b = 1 + |y - x| + \frac{(y-x)^2}{2} + \frac{(x-y)^2}{3}$$

3.

$$a = (1+y)\frac{x + \frac{y}{(x^2+4)}}{e^2 + \frac{1}{(x^2+4)}}$$

$$b = \frac{1 + \cos^3(y-x)}{\frac{x^2}{2} + \sin^2(z)}$$

4.

$$a = y + \frac{x}{y^2 + \left|\frac{x^2}{y + x^2}\right|}$$

$$b = (1 + tg^2\frac{z}{2})^2$$

5.

$$a = \frac{2\cos^4(x - \frac{\pi}{6})}{\frac{1}{2} + \sin^2 y}$$

$$b = 1 + \frac{z^2}{3 + \frac{z^2}{5}}$$

6.

$$a = \frac{\sqrt{|x-1|} - \sqrt{|y|}}{1 + \frac{x^2}{2} + \frac{y^2}{4}}$$

$$b = x^3(arctg^3 z + e)$$

7.

$$a = \frac{1 + \sin^2(x+y)}{2 + \left|x - \frac{2x}{(1 + x^2y^2)}\right|} + x$$

$$b = \cos^2(arctg\frac{1}{z})$$

8.

$$a = \ln\left|(y - \sqrt{|x|})(x - \frac{y}{z + \frac{x^2}{4}})\right|$$

$$b = x - \frac{x^2}{3} + \frac{x+y}{x}$$

9.

$$a = \frac{x^2}{8 + \frac{x^2}{3} + \frac{y^2}{6}}$$

$$b = x(\cos^3(x+z) + 1)$$

10.

$$a = \frac{|5 - 2e|}{1 + x^2(y - tg(z))}$$

$$b = |y - 4| + \frac{(y-x)^2}{6} + \frac{(x-y)^2}{7}$$

11.

$$a = (2 + x)\frac{1 + \dfrac{y}{(x^2 + 3)}}{y^2 + \dfrac{1}{(z^2 + 4)}} \qquad b = (1 + tg^2\frac{x}{2})^2$$

Формулы для задания № 4 (для вариантов 3, 4, 7, 8)

3

$$y = \begin{cases} ax^2 + bx + c, & при \quad n = 1 \\ ax^3 + 3bx + b^3, & при \quad n = 2 \\ 4\sin^2 x + \cos^2 x, & при \quad n = 3 \\ \sqrt{|x - y|} \cdot \cos^2 x, & при \quad n = 4 \\ ctg^2 x, & при \quad n = 5 \end{cases}$$

7

$$y = \begin{cases} e^2 \sin x \cdot tg^2 x, & при \quad k = 1 \\ \pi \cdot R^2, & при \quad k = 2 \\ \dfrac{4}{3\pi R} + 2.1, & при \quad k = 3 \\ (a \cdot \cos(bx))^2 & при \quad k = 4 \end{cases}$$

4

$$S = \begin{cases} a \cdot b, & если \quad n = 1 \\ a \cdot \dfrac{h}{2}, & если \quad n = 2 \\ (a + b) \cdot \dfrac{h}{2}, & если \quad n = 3 \\ \pi \cdot R^2, & если \quad n = 4 \\ \pi \cdot R^2 \cdot \dfrac{a}{360}, & если \quad n = 5 \end{cases}$$

8

$$S = \begin{cases} p \cdot l, & если \quad k = 1 \\ p \cdot \dfrac{h}{2}, & если \quad k = 2 \\ 2\pi \cdot R \cdot h, & если \quad k = 3 \\ 2\pi \cdot R \cdot l, & если \quad k = 4 \\ \pi \cdot R^2, & если \quad k = 5 \\ \pi \cdot R \cdot (2h + a), & если \quad k = 6 \end{cases}$$

Задание 2

1. Создать структуру согласно предметной области варианта:

Вариант	**Класс**
1	Человек
2	Студент
3	Преподаватель
4	Мотоцикл
5	Легковой автомобиль
6	Грузовой автомобиль
7	Вертолет
8	Самолет
9	Жилой дом
10	Административное здание
11	Завод

2. Добавить метод отображения данных структуры.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Лабораторная работа № 1 – выполнить полученное задание в консольном приложении (.NET Framework).

5. Лабораторная работа № 2 – выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 3-4

«Разработка пользовательских интерфейсов для программирования конструкций циклов на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ программирования циклов на языке C# и получение навыков работы с циклами в визуальном приложении на платформе .NET Framework.

Краткие теоретические сведения

Понятие цикла

В большинстве задач, встречающихся на практике, необходимо производить многократное выполнение некоторого действия. Такой многократно повторяющийся участок вычислительного процесса называется циклом.

Если заранее известно количество необходимых повторений, то цикл называется арифметическим. Если же количество повторений заранее неизвестно, то говорят об итерационном цикле.

В итерационных циклах производится проверка некоторого условия, и в зависимости от результата этой проверки происходит либо выход из цикла, либо повторение выполнения тела цикла. Если проверка условия производится перед выполнением блока операторов, то такой итерационный цикл называется циклом с предусловием (цикл "пока"), а если проверка производится после выполнения тела цикла, то это цикл с постусловием (цикл "до").

Особенность этих циклов заключается в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, а тело цикла с предусловием может ни разу не выполниться. В зависимости от решаемой задачи необходимо использовать тот или иной вид итерационных циклов.

Арифметический цикл for

Цикл for

Цикл for в C# предоставляет механизм итерации, в котором определенное условие проверяется перед выполнением каждой итерации. Синтаксис этого оператора показан ниже:

for (инициализатор; условие; итератор)

оператор (операторы)

Здесь:

– инициализатор

это выражение, вычисляемое перед первым выполнением тела цикла (обычно инициализация локальной переменной в качестве счетчика цикла). Инициализация, как правило, представлена оператором присваивания, задающим первоначальное значение переменной, которая выполняет роль счетчика и управляет циклом;

– условие

это выражение, проверяемое перед каждой новой итерацией цикла (должно возвращать true, чтобы была выполнена следующая итерация);

– итератор

выражение, вычисляемое после каждой итерации (обычно приращение значения счетчика цикла).

Цикл с предусловием while

Подобно for, while также является циклом с предварительной проверкой. Синтаксис его аналогичен, но циклы while включают только одно выражение:

while(условие)

оператор (операторы);

где оператор — это единственный оператор или же блок операторов, а условие означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле оператор выполняется до тех пор, пока условие истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла.

Как и в цикле for, в цикле while проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может вообще не выполняться, а также избавляет от необходимости выполнять отдельную проверку перед самим циклом.

Цикл do. . . while

Цикл do...while в C# — это версия while с постпроверкой условия. Это значит, что условие цикла проверяется после выполнения тела цикла. Следовательно, циклы do...while удобны в тех ситуациях, когда блок операторов должен быть выполнен как минимум однажды. Ниже приведена общая форма оператора цикла do-while:

do {

операторы;

} while (условие);

При наличии лишь одного оператора фигурные скобки в данной форме записи необязательны. Тем не менее они зачастую используются для того, чтобы сделать конструкцию do-while более удобочитаемой и не путать ее с конструкцией цикла while. Цикл do-while выполняется до тех пор, пока условное выражение истинно.

Цикл foreach

Цикл foreach служит для циклического обращения к элементам коллекции, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла foreach:

foreach (тип имя_переменной_цикла in коллекция)

оператор;

Здесь тип имя_переменной_цикла обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла foreach. А коллекция обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, тип переменной цикла должен соответствовать типу элемента массива. Кроме того, тип может обозначаться ключевым словом var. В этом случае компилятор определяет тип переменной цикла, исходя из типа элемента массива. Это может оказаться полезным для работы с определенного рода запросами. Но, как правило, тип указывается явным образом.

Оператор цикла foreach действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными.

Цикл foreach позволяет проходить по каждому элементу коллекции (объект, представляющий список других объектов). Формально для того, чтобы нечто можно было рассматривать как коллекцию, это нечто должно поддерживать интер-

фейс IEnumerable. Примерами коллекций могут служить массивы C#, классы коллекций из пространства имен System.Collection, а также пользовательские классы коллекций.

Операторы завершения цикла

Для всех операторов цикла выход из цикла осуществляется как вследствие естественного окончания оператора цикла, так и с помощью операторов перехода и выхода.

В языке C# определены стандартные процедуры:

– *Break*

– *Continue*

Процедура Break выполняет безусловный выход из цикла. Процедура Continue обеспечивает переход к началу новой итерации цикла.

Заметим, что хотя и существует возможность выхода из цикла с помощью оператора безусловного перехода goto, делать этого не желательно. Во всех случаях можно воспользоваться специально предназначенными для этого процедурами Break и Continue.

Задание на лабораторную работу

Задание 1

Найти сумму ряда при заданном n:

$$S = \frac{1}{1^2} + \frac{1}{2^2} + \ldots + \frac{1}{n^2} + \ldots$$

Задание 2

Найти сумму бесконечного ряда:

$$S = \frac{1}{\sqrt{1}} + \frac{1}{\sqrt{1 \cdot 2}} + \ldots + \frac{1}{\sqrt{1 \cdot 2 \ldots n}} + \ldots$$

Суммировать до тех про, пока члены ряда не станут меньше заданного eps>0.

Задание 3

Дано натуральное число. Определить сколько раз в нем встречается цифра a.

Задание 4

Написать программу, реализующую вычисление значений заданной функции в каждой точке отрезка, а затем найти сумму полученных значений. Решение задачи реализовать через каждый вид цикла.

Вариант	Функция	Отрезок
1	$\sqrt{2(x-2)^2(8-x)-1}$	[0;6]
2	$4-x-\frac{4}{x^2}$	[1;5]
3	$x^2+\frac{16}{x}-16$	[1;6]
4	$\frac{2(x^2+3)}{x^2-2x+5}-1{,}5$	[-3;3]
5	$2\sqrt{x}-x-0.5$	[0;4]
6	$1+\sqrt{2(x-1)^2(x-7)}$	[7;12]
7	x^2-7x+1	[1;8]
8	$\frac{10x}{x^2+1}-3$	[0;3]
9	$-2+\sqrt{2(x+1)^2(5-x)}$	[-3;3]
10	$2x^2+\frac{108}{x^2}-59$	[2;7]
11	$\sqrt{4(x-1)^2(8-1/x)}$	[1;3]

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Лабораторная работа № 3 – выполнить полученное задание в консольном приложении (.NET Framework).

5. Лабораторная работа № 4 – выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 5

«Разработка пользовательских интерфейсов для программирования массивов на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ программирования массивов на платформе .NET Framework.

Краткие теоретические сведения

Массивы

Массивом называют упорядоченную совокупность элементов одного типа. Каждый элемент массива имеет индексы, определяющие порядок элементов. Индексы задаются целочисленным типом. Число индексов характеризует размерность массива. Если конечное значение задано константным выражением, то число элементов массива известно в момент его объявления и ему может быть выделена память ещё на этапе трансляции. Такие массивы называются статическими. Если же конечное значение зависят от переменной, то такой массив называют динамическим, поскольку память ему может быть отведена только динамически в процессе выполнения программы, когда становятся известными значения соответствующих переменных. Массиву, как правило, выделяется непрерывная область памяти.

Массивы характеризуются они следующим:

– каждый компонент массива может быть явно обозначен и к нему имеется прямой доступ;

– число компонент массива определяется при его описании и в дальнейшем не меняется.

Для обозначения компонент массива используется имя переменной-массива и так называемые индексы, которые обычно указывают желаемый элемент.

Для того, чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор new.

В языке C# каждый индекс изменяется в диапазоне от 0 до некоторого конечного значения. Массивы в языке C# являются настоящими динамическими массивами. Как следствие этого, массивы относятся к ссылочным типам, память им отводится динамически в "куче". Массивы могут быть одномерными и многомерными.

Одномерные массивы

Объявление одномерного массива выглядит следующим образом:

<тип>[] <объявители>;

Как и в случае объявления простых переменных, каждый объявитель может быть именем или именем с инициализацией. В первом случае речь идёт об отложенной инициализации. Нужно понимать, что при объявлении с отложенной инициализацией сам массив не формируется, а создаётся только ссылка на массив, имеющая неопределённое значение Null. Поэтому пока массив не будет реально создан и его элементы инициализированы, использовать его в вычислениях нельзя. Вот пример объявления трёх массивов с отложенной инициализацией:

```
int[] a, b, c;
```

Чаще всего при объявлении массива используется имя с инициализацией. И опять-таки, как и в случае простых переменных, могут быть два варианта инициализации. В первом случае инициализация является явной и задаётся константным массивом. Вот пример:

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

Следуя синтаксису, элементы константного массива следует заключать в фигурные скобки. Если массив инициализируется константным массивом, то в динамической памяти создаётся константный массив с заданными значениями, с которым и связывается ссылка.

Во втором случае создание массива выполняется с помощью операции new. Вот пример:

```
int[] d = new int[5];
```

Здесь объявлен динамический целочисленный массив, в котором будут храниться 5 целых чисел. Массив создаётся в динамической памяти, его элементы получают начальные нулевые значения, и ссылка связывается с этим массивом.

Как обычно задаются элементы массива, если они не заданы при инициализации? Они либо вычисляются, либо вводятся пользователем. Рассмотрим пример работы с массивами:

```
// объявляются три одномерных массива A,B,C
int[] A = new int[5], B= new int[5], C= new int[5];
// заполняется данными с клавиатуры массив A
for(int i = 0; i < 5; i++) A[i] = int.Parse(Console.ReadLine());
// вычисляются элементы массива C
for(int i = 0; i < 5; i++) C[i] = A[i] + B[i];
// объявление массива с явной инициализацией
```

```
int[] x ={5.5, 6.6, 7.7};
// объявление массивов с отложенной инициализацией
int[] u, v;
u = new int[3];
for(int i = 0; i < 3; i++) u[i] = i + 1;
// v = {1,2,3}; // присваивание константного массива недопустимо!
v = new int[4];
v = u; // допустимое присваивание – массивы одного класса
int [,] w = new int[3,5];
// v = w; // недопустимое присваивание: объекты разных классов
```

На что следует обратить внимание, анализируя этот текст:

Показаны разные способы объявления массивов. В начале объявляются одномерные массивы A, B и C. Значения элементов этих трёх массивов имеют один и тот же тип int. То, что они имеют одинаковое число элементов, произошло по воле программиста, а не диктовалось требованиями языка. Значения в массив A вводились, а в массив B - нет, но сложение элементов корректно, потому что при объявлении элементы массива B получили нулевые значения.

Массив x объявлен с явной инициализацией. Число и значения его элементов определяется константным массивом.

Массивы u и v объявлены с отложенной инициализацией. В последующих операторах массив u инициализируется в объектном стиле – его элементы получают значения в цикле.

Обратите внимание на закомментированный оператор присваивания! В отличие от инициализации, использовать константный массив в правой части оператора присваивания недопустимо. Эта попытка приводит к ошибке, поскольку v - это ссылка, и ей нельзя присвоить константный массив. А вот ссылку присвоить можно. Что происходит в операторе присваивания v = u? Это корректное ссылочное присваивание: хотя u и v имеют разное число элементов, но они являются объектами одного класса – оба массива целочисленные. В результате присваивания память, отведённая массиву v, освободится, ею займется теперь сборщик мусора. Обе ссылки u и v будут теперь указывать на один и тот же массив, так что изменение элемента одного массива немедленно отразится на другом массиве. Имена u и v становятся синонимами (или псевдонимами друг друга…).

Далее определяется двумерный массив w и делается попытка выполнить оператор присваивания v = w. Это ссылочное присваивание некорректно, поскольку объекты w и v - разных классов (разноразмерные массивы) и для них не выполняется требуемое для присваивания согласование по типу.

Многомерные массивы

Перейдем к многомерным массивам. Многомерные массивы задаются следующей формой:

<тип>[, ... ,] <объявители>;

Число запятых, увеличенное на единицу, и задаёт размерность массива. Что касается объявителей, то всё, что сказано для одномерных массивов, справедливо и для многомерных. Можно лишь отметить, что хотя явная инициализация с использованием многомерных константных массивов возможна, но применяется редко из-за громоздкости такой структуры. Проще инициализацию реализовать программно, но иногда она всё же применяется. Вот пример:

int[,] matrix = {{1,2},{3,4}};

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

В приведённом ниже примере показана работа с трёхмерным массивом. Массив создаётся с использованием циклов типа for, а при нахождении суммы его элементов, минимального и максимального значения используется цикл foreach:

```
int [,,] arr3d = new int[10, 10, 10];
for(int i = 0; i < 10; i++)
  for(int j = 0; j < 10; j++)
    for(int k = 0; k < 10; k++)
      arr3d[i, j, k] = int.Parse(Console.ReadLine());
long sum = 0;
int min = arr3d[0, 0, 0], max = arr3d[0, 0, 0];
foreach(int item in arr3d)
{
  sum += item;
  if(item > max) max = item;
  else if (item < min) min = item;
}
Console.WriteLine("sum = {0}, min = {1}, max = {2}",
                  sum, min, max);
```

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

Массив массивов

Массив массивов — это массив, элементы которого сами являются массивами. Элементы массива массивов могут иметь различные размеры и измерения. Массивы массивов иногда также называются "невыровненными массивами". В следующих примерах показано, как выполняется объявление, инициализация и доступ к массивам массивов.

Ниже показано объявление одномерного массива, включающего три элемента, каждый из которых является одномерным массивом целых чисел.

```
int[][] jaggedArray = new int[3][];
```

Перед использованием jaggedArray его элементы нужно инициализировать. Сделать это можно следующим образом.

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Каждый элемент представляет собой одномерный массив целых чисел. Первый элемент массива состоит из пяти целых чисел, второй — из четырех и третий — из двух.

Задание на лабораторную работу

Задание 1

Вариант	№	Задание
1	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $\sqrt{\frac{n^k}{(n+1)^k}}$
	2	Заданы два массива A и B. Первым на печать вывести массив, сумма значений которого окажется наименьшей.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных выше главной диагонали.
	4	Сортировка элементов массива методом простого выбора
2	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $\frac{2^k\sqrt{n}}{3^k}$
	2	Заданы два массива A и B. Первым на печать вывести массив, произведение значений которого окажется наименьшим.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных ниже главной диагонали.
	4	Сортировка элементов массива методом простого обмена
3	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $\frac{2^k\sqrt{n}}{n^k}$
	2	Заданы два массива A и B. В каждом из массивов найти наименьшее значение и прибавить его ко всем элементам массивов. На печать вывести исходные и преобразованные массивы.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных выше побочной диагонали.

	4	Сортировка элементов массива методом прямого включения
4	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $$\frac{\sqrt{n}3^k}{5^k}$$
	2	Заданы два массива A и B. В каждом из массивов найти наибольшее значение и вычесть его из всех элементов массивов. На печать вывести исходные и преобразованные массивы.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных ниже побочной диагонали.
	4	Сортировка элементов массива слияниями
5	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $$\frac{5^k}{\sqrt{n(n+1)}}$$
	2	Заданы два массива A и B. В каждом из массивов найти среднее арифметическое всех элементов массивов. На печать вывести исходные массивы и найденные значения.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных на главной диагонали.
	4	Быстрая сортировка элементов массива
6	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $$\frac{(-1)^k}{\sqrt{n}3^k}$$
	2	Заданы два массива A и B. Первым на печать вывести массив, содержащий наибольшее значение. Напечатать также это значение и его порядковый номер.
	3	Дан массив A(n,n). Найти сумму всех его элементов, расположенных на побочной диагонали.
	4	Пирамидальная сортировка элементов массива
7	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число:

		$\frac{1}{\sqrt{(2n-1)^k}}$
	2	Заданы два массива A и B. Подсчитать в них количество отрицательных элементов и первым на печать вывести массив, имеющий наименьшее их количество.
	3	Задан массив действительных чисел A(n,n). Необходимо каждый элемент массива разделить на среднее арифметическое этих элементов. На печать вывести исх. и преобразов. массивы.
	4	Сортировка элементов массива методом простого выбора
8	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $(-1)^k \frac{1}{n\sqrt{3^k}}$
	2	Заданы два массива A и B. Подсчитать в них количество элементов, больших значения t и первым на печать вывести массив, имеющий наименьшее их количество.
	3	Дан массив A(n,n). Вычислить сумму всех неотрицательных элементов, а также их количество.
	4	Сортировка элементов массива методом простого обмена
9	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $\frac{2^k}{\left(\sqrt{n}+3\right)^2}$
	2	Заданы два массива A и B. В каждом из массивов найти наименьшее значение и умножить на него все элементы массивов. На печать вывести исходные и преобразованные массивы.
	3	Дан массив A(n,n). Найти число элементов массива a(i,j) > t и просуммировать все эти элементы.
	4	Сортировка элементов массива методом прямого включения
10	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число:

		$\frac{(-1)^k\sqrt{n}}{2+n^2}$
	2	Заданы два массива A и B. Подсчитать в них количество элементов, кратных двум и первым на печать вывести массив, имеющий наибольшее их количество.
	3	Задан двухмерный массив целых чисел A размером N на M. Найти произведение элементов, расположенных на главной диагонали.
	4	Сортировка элементов массива слияниями
11	1	Сформировать одномерный (длиной n) и двумерный (длиной n на m) массивы по заданному правилу, где k – случайное число: $\left(\frac{2\sqrt{n}+5}{2n}\right)^k$
	2	Задан массив A. Получить из него массив B, состоящий из элементов массива A, которые кратны двум.
	3	Задана матрица A(n,n), состоящая из нулей и единиц. Подсчитать количество нулей и единиц в этой матрице.
	4	Пирамидальная сортировка элементов массива

Задание 2

Написать программу для ввода/вывода в массив данных с различной длинной записей, то есть создать массив массивов, перечисляющий различные свойства одного объекта.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Выполнить полученное задание в консольном приложении (.NET Framework).

5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 6

«Разработка пользовательских интерфейсов для программирования строк на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ программирования строк на языке C# и получение навыков работы с строками в визуальном приложении на платформе .NET Framework.

Краткие теоретические сведения

Строки

Общее описание строк

Тип string представляет последовательность из нуля или более символов в кодировке Юникод. Тип string — это псевдоним для типа String платформы .NET Framework.

Несмотря на то, что тип string является ссылочным типом, операторы равенства (== и !=) определены для сравнения значений объектов типа string, а не ссылок. Это упрощает проверку равенства строк. Например:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine((object)a == (object)b);
```

В этом примере отображается "True", а затем "False", поскольку содержимое строк одинаково, но a и b не ссылаются на один и тот же экземпляр строки.

Оператор «+» служит для объединения строк:

```
string a = "good " + "morning";
```

В данном примере создается строковый объект, содержащий текст "good morning".

Строки являются неизменяемыми: содержимое строкового объекта невозможно изменить после создания объекта, хотя из-за синтаксиса изменения кажутся возможными. Например, при написании этого кода компилятор на самом деле создает новый строковый объект для новой последовательности знаков, и этот новый объект получает значение "b". Затем строку "h" можно применять для сборки мусора.

```
string b = "h";
b += "ello";
```

Оператор [] служит для доступа только для чтения к отдельным знакам объекта string.

```
string str = "test";
char x = str[2];  // x = 's';
```

Методы для работы со строками

Класс Strings включает множество методов сравнения, поиска и изменения строковых значений. Здесь я приведу список часто необходимых (лично мне) возможностей этого класса:

Empty — свойство определяющее пустая ли строка;

Compare() — сравнивает две строки;

CompareOrdinal() — позволяет сравнивать строки в независимости от региональных настроек;

Concat() — объединяет две и более строки в новую строку;

Copy() — Копирует исходную строку;

Equals() — проверяет, содержат ли две строки одинаковые значения;

Format() — форматирует строку, используя строго заданный формат;

Intern() — возвращает ссылку на существующий экземпляр строки;

Join() — добавляет новую строку в заданное место уже существующей строки;

Chars — индексатор символов строки;

Length — количество символов в строке;

Clone() — возвращает ссылку на существующую строку;

CompareTo() — сравнивает одну строку с другой;

CopyTo() — копирует заданное число символов строки в массив Unicode символов;

EndsWith() — проверяет, заканчивается ли строка определённой последовательностью символов;

Equals() — определяет, имеют ли две строки одинаковые значения;

Insert() — вставляет новую строку в уже существующую;

LastIndexOf() — возвращает индекс последнего вхождения элемента в строку;

PadLeft() — выравнивает строку по правому краю, пропуская все проблеы или другие специально заданные символы;

PadRight() — выравнивает строку по левому краю, пропуская все проблеы или другие специально заданные символы;

Remove() — удаляет заданное число символов из строки;

Split() — возвращает подстроку, отделённую от основного массива определённым символом;

StartsWith() — определяет, начинается ли строка с определённой последовательности символов;

Substring() — возвращает подстроку из общего массива символов;

ToCharArray() — копирует символы из строки в массив символов;

ToLower() — преобразует символы в строке к нижнему регистру;

ToUpper() — преобразует символы в строке к верхнему регистру;

Trim() — удаляет все вхождения определённых символов в начале и в конце строки;

TrimEnd() — удаляет все вхождения определённых символов в конце строки;

TrimStart() — удаляет все вхождения определённых символов в начале строки.

Коллекции

Основные понятия

Во многих приложениях требуется создавать группы связанных объектов и управлять ими. Существует два способа группировки объектов: создать массив объектов и создать коллекцию.

Массивы удобнее всего использовать для создания фиксированного числа строго типизированных объектов и работы с ними.

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, коллекция, с которой вы работаете, может расти или уменьшаться динамически при необходимости. Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции.

Коллекция является классом, поэтому необходимо объявить экземпляр класса перед добавлением в коллекцию элементов.

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен System.Collections.Generic. Универсальная коллекция обеспечивает строгую типизацию, так что в нее нельзя добавить другие типы данных. При извлечении элемента из универсальной коллекции не нужно определять или преобразовывать его тип данных.

В C# под коллекцией понимается группа объектов. Пространство имен System.Collections содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов. Коллекции упрощают программирование, предлагая уже готовые решения для построения структур данных, разработка которых "с нуля" отличается большой трудоемкостью. Речь идет о встроенных коллекциях, которые поддерживают, например, функционирование стеков, очередей и хеш-таблиц. Коллекции пользуются большой популярностью у всех C#-программистов.

Основное достоинство коллекций состоит в том, что они стандартизируют способ обработки групп объектов в прикладных программах. Все коллекции разработаны на основе набора четко определенных интерфейсов. Ряд встроенных реализаций таких интерфейсов, как ArrayList, Hashtable, Stack и Queue, вы можете использовать "как есть". У каждого программиста также есть возможность реализовать собственную коллекцию, но в большинстве случаев достаточно встроенных.

Среда .NET Framework поддерживает три основных типа коллекций: общего назначения, специализированные и ориентированные на побитовую организацию данных. Коллекции общего назначения реализуют ряд основных структур данных,

включая динамический массив, стек и очередь. Сюда также относятся словари, предназначенные для хранения пар ключ/значение. Коллекции общего назначения работают с данными типа object, поэтому их можно использовать для хранения данных любого типа.

Коллекции специального назначения ориентированы на обработку данных конкретного типа или на обработку уникальным способом. Например, существуют специализированные коллекции, предназначенные только для обработки строк или однонаправленного списка.

Классы коллекций, ориентированных на побитовую организацию данных, служат для хранения групп битов. Коллекции этой категории поддерживают такой набор операций, который не характерен для коллекций других типов. Например, в известной многим бит-ориентированной коллекции BitArray определены такие побитовые операции, как И и исключающее ИЛИ.

Основополагающим для всех коллекций является реализация перечислителя (нумератора), который поддерживается интерфейсами IEnumerator и IEnumerable.

Перечислитель обеспечивает стандартизованный способ поэлементного доступа к содержимому коллекции. Поскольку каждая коллекция должна реализовать интерфейс IEnumerable, к элементам любого класса коллекции можно получить доступ с помощью методов, определенных в интерфейсе IEnumerator. Следовательно, после внесения небольших изменений код, который позволяет циклически опрашивать коллекцию одного типа, можно успешно использовать для циклического опроса коллекции другого типа. Интересно отметить, что содержимое коллекции любого типа можно опросить с помощью нумератора, используемого в цикле foreach. И еще. Если вы знакомы со средствами C++-программирования, то вам будет интересно узнать, что C#-классы коллекций по сути аналогичны классам стандартной библиотеки шаблонов (Standard Template Library -— STL), определенной в C++. То, что в C++ называется контейнером, в C# именуется коллекцией. То же справедливо и для Java. Если вы знакомы с Java-средой Collections Framework, то очень легко освоите использование C#-коллекций.

Классы коллекций общего назначения

Классы общего назначения можно использовать для хранения объектов любого типа. Битовые предназначены для хранения битовой информации. Коллекции специального назначения разрабатываются для обработки данных конкретного типа. Этот раздел посвящен классам коллекций общего назначения. В таблице 1 представлены классы коллекций общего назначения.

Таблица 1 – Классы коллекций общего назначения

ArrayList	Динамический массив, т.е. массив который при необходимости может увеличивать свой размер
Hashtabie	Хеш-таблица для пар ключ/значение
Queue	Очередь, или список, действующий по принципу: первым прибыл — первым обслужен
SortedList	Отсортированный список пар ключ/значение
Stack	Стек, или список, действующий по принципу: первым прибыл — последним обслужен

Класс ArrayList

Класс ArrayList предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или сокращаться. В C# стандартные массивы имеют фиксированную длину, которая не может измениться во время выполнения программы. Это означает, что программист должен знать заранее, сколько элементов будет храниться в массиве. Но иногда до выполнения программы нельзя точно сказать, массив какого размера понадобится. В таких случаях и используется класс ArrayList. Объект класса ArrayList представляет собой массив переменной длины, элементами которого являются объектные ссылки. Любой объект класса ArrayList создается с некоторым начальным размером. При превышении этого размера коллекция автоматически его увеличивает. В случае удаления объектов массив можно сократить. Коллекция класса ArrayList, пожалуй, наиболее употребимая, поэтому ее стоит рассмотреть в деталях.

Класс ArrayList реализует интерфейсы ICollection, IList, IEnumerable и ICloneable. В классе ArrayList определены следующие конструкторы:

public ArrayList()

public ArrayList(ICollection c)

public ArrayList(int capacity)

Первый конструктор предназначен для создания пустого ArrayList-массива с начальной емкостью, равной 16 элементам. Второй служит для построения массива, который инициализируется элементами и емкостью коллекции, заданной параметром c. Третий конструктор создает список с заданной начальной емкостью. Емкость (или вместимость) — это размер массива для хранения элементов. При добавлении элементов в ArrayList-массив его емкость автоматически увеличивается, причем каждый раз, когда список должен расшириться, его емкость удваивается.

Помимо методов, определенных в интерфейсах, которые реализует класс ArrayList, в нем определены и собственные методы. Коллекцию класса ArrayList можно отсортировать с помощью метода Sort(). В отсортированной коллекции можно эффективно выполнять поиск элементов, используя метод BinarySearch(). При необходимости содержимое ArrayList-коллекции можно реверсировать, вызвав метод Reverse(). Класс ArrayList поддерживает ряд методов, которые действуют в некотором диапазоне элементов коллекции. Например, вызвав метод insertRange(), можно вставить в ArrayList-массив другую коллекцию. С помощью метода RemoveRange() можно удалить из коллекции заданный диапазон элементов. А если нужно заменить элементы заданного диапазона одной коллекции элементами другой, используйте метод SetRange(). Сортировать можно не только всю коллекцию, но и заданный диапазон внутри нее. То же справедливо и для поиска.

По умолчанию коллекция класса ArrayList не синхронизирована. Чтобы поместить коллекцию в синхронизированную оболочку, вызовите метод Synchronized().

В таблице 2 представлены основные методы класса ArrayList.

Таблица 2 - основные методы класса ArrayList

public virtual void AddRange(ICollection c)	Добавляет элементы из коллекции c в конец вызывающей коллекции
public virtual int BinarySearch (object v)	В вызывающей коллекции выполняет поиск значения, заданного параметром v. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
public virtual int BinarySearch(object v, IComparer comp)	В вызывающей коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром comp. Возвращает индекс найденного элемента. Если искомое

	значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
public virtual int BinarySearch(int startldx, int count, object v, IComparer comp)	В вызывающей коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром comp. Поиск начинается с элемента, индекс которого равен значению startldx, и включает count элементов. Метод возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
public virtual void CopyTo(Array ar, int startldx)	Копирует содержимое вызывающей коллекции, начиная с элемента, индекс которого равен значению startldx, в массив, заданный параметром a г. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции
public virtual void CopyTo(int srcldx, Array ar, int destldx, int count)	Копирует count элементов вызывающей коллекции, начиная с элемента, индекс которого равен значению srcidx, в массив, заданный параметром a г, начиная с элемента, индекс которого равен значению destldx. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции
public virtual ArrayList GetRange(int idx, int count)	Возвращает часть вызывающей коллекции типа ArrayList. Диапазон возвращаемой коллекции начинается с индекса idx и включает count элементов. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
public static ArrayList FixedSize(ArrayList ar)	Превращает коллекцию ar в ArrayList-массив с фиксированным размером и возвращает результат
public virtual void InsertRange(int startldx, ICollection c)	Вставляет элементы коллекции, заданной параметром c, в вызывающую коллекцию, начиная с индекса, заданного параметром startldx

public virtual int LastlndexOf(object v)	Возвращает индекс последнего вхождения объекта v в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение -1
public static ArrayList Readonly(ArrayList ar)	Превращает коллекцию ar в ArrayList-массив, предназначенный только для чтения, и возвращает результат 8
public virtual void ReraoveRange(int idx, int count)	Удаляет count элементов из вызывающей коллекции, начиная с элемента, индекс которого равен значению idx
public virtual void Reverse()	Располагает элементы вызывающей коллекции в обратном порядке
public virtual void Reverse(int startldx, int count)	Располагает в обратном порядке count элементов вызывающей коллекции, начиная с индекса startldx
public virtual void SetRange(int startldx, ICollection c)	Заменяет элементы вызывающей коллекции, начиная с индекса startldx, элементами коллекции, заданной параметром с
public virtual void Sort()	Сортирует коллекцию по возрастанию
public virtual void Sort(IComparer comp)	Сортирует вызывающую коллекцию на основе метода сравнения объектов, заданного параметром comp. Если параметр comp имеет нулевое значение, для каждого объекта используется стандартный метод сравнения
public virtual void Sort (int startidx, int endidx, icomparer comp)	Сортирует часть вызывающей коллекции на основе метода сравнения объектов, заданного параметром comp. Сортировка начинается с индекса startidx и заканчивается индексом endidx. Если параметр comp имеет нулевое значение, для каждого объекта используется стандартный метод сравнения

public static ArrayList Synchronized(ArrayList list)	Возвращает синхронизированную версию вызывающей коллекции
public virtual object [ ] ToArray ()	Возвращает массив, который содержит копии элементов вызывающего объекта
public virtual Array ToArray (Type type)	Возвращает массив, который содержит копии элементов вызывающего объекта. Тип элементов в этом массиве задается параметром type
public virtual void TrimToSize()	Устанавливает свойство Capacity равным значению свойства Count

Использование динамического массива типа ArrayList демонстрируется в следующей программе. Сначала она создает ArrayList-массив, а затем добавляет в него символы и отображает их. После удаления элементов список отображается снова. Последующее добавление новых элементов приводит к автоматическому увеличению емкости используемого списка. На "десерт" демонстрируется возможность изменения содержимого его элементов.

```
// Демонстрация использования ArrayList-массива.
using System;
using System.Collections;
class ArrayListDemo {
public static void Main() {
// Создаем динамический массив.
ArrayList al = new ArrayListO;
Console.WriteLine("Начальная емкость: " + al.Capacity);
Console.WriteLine("Начальное количество элементов: al.Count);
Console.WriteLine();
Console.WriteLine("Добавляем 6 элементов.");
// Добавляем элементы в динамический массив.
al.Add('C');
```

```
al.Add(‘A’);
al.Add(‘B’);
al.Add(‘D’);
al.Add(‘F’);
Console.WriteLine("Текущая емкость: " + al.Capacity);
Console.WriteLine("Количество элементов: " + al.Count);
// Отображаем массив, используя индексацию.
Console.Write("Текущее содержимое массива: ") ;
for(int i=0; i < al.Count; i++)
Console.Write (al[i] + " ");
Console.WriteLine("\n");
Console.WriteLine("Удаляем 2 элемента.");
// Удаляем элементы из динамического массива.
al.Remove(‘F’);
al.Remove(‘A’);
Console.WriteLine("Текущая емкость: " + al.Capacity);
Console.WriteLine("Количество элементов: " + al.Count);
// Для отображения массива используем цикл foreach.
Console.Write("Содержимое: ") ;
foreach(char c in al)
Console.Write(c + " ") ;
Console.WriteLine("\n");
Console.WriteLine("Добавляем еще 20 элементов.");
// Добавляем такое количество элементов в массив,
// которое заставит его увеличить свой размер.
for(int i=0; i < 20; i++)
al.Add((char)('a' + i) ) ;
Console.WriteLine("Текущая емкость: " + al.Capacity);
Console.WriteLine(
"Количество элементов после добавления 20 новых: " + al.Count);
Console.Write("Содержимое: ");
```

```
foreach(char c in al)
Console.Write(c + " ") ;
Console.WriteLine("\n");
// Изменяем содержимое массива, используя индексацию.
Console.WriteLine("Изменяем первые три элемента.");
al[0] = 'X';
al[l] = 'Y';
al[2] = 'Z';
Console.Write("Содержимое: ");
foreach(char c in al)
Console.Write(c + " ");
Console.WriteLine();
}
}
```

Результаты выполнения программы:

Начальная емкость: 16

Начальное количество элементов: 0

Добавляем 6 элементов.

Текущая емкость: 16

Количество элементов: 6

Текущее содержимое массива: C A E B D F

Удаляем 2 элемента.

Текущая емкость: 16

Количество элементов: 4

Содержимое: C E B D

Добавляем еще 20 элементов.

Текущая емкость: 32

Количество элементов после добавления 20 новых: 24

Содержимое: C E B D a b c d e f g h i j k l m n o p q r s t

Изменяем первые три элемента.

Содержимое: X Y Z D a b c d e f g h i j k l m n o p q r s t

Обратите внимание на то, что коллекция с самого начала работы этой программы пуста, но ее начальная емкость равна 16. При необходимости коллекция увеличивается, причем каждый раз, когда возникает такая необходимость, емкость удваивается.

Класс Hashtable

Класс Hashtable предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. Возможно, многим известно, что в хеш-таблице для хранения информации используется механизм, именуемый хешированием (hashing). Суть хеширования состоит в том, что для определения уникального значения, которое называется хэш-кодом, используется информационное содержимое соответствующего ему ключа. Хэш-код затем используется в качестве индекса, по которому в таблице отыскиваются данные, соответствующие этому ключу. Преобразование ключа в хэш-код выполняется автоматически, т.е. сам хэш-код вы даже не увидите. Но преимущество хеширования — в том, что оно позволяет сохранять постоянным время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации. Класс Hashtable реализует интерфейсы IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback и ICloneable. В классе Hashtable определено множество конструкторов, включая следующие (они используются чаще всего):

public Hashtable()

public Hashtable(IDictionary c)

public Hashtable(int capacity)

public Hashtable(int capacity, float fillRatio)

Первая форма позволяет создать стандартный объект класса Hashtable. Вторая для инициализации Hashtable-объекта использует элементы заданной коллекции c. Третья инициализирует емкость создаваемой хэш-таблицы значением capacity, а четвертая — как емкость (значением capacity), так и коэффициент заполнения (значением fillRatio). Значение коэффициента заполнения (также именуемого коэффициентом нагрузки), которое должно попадать в диапазон 0,1-1,0, определяет степень заполнения хеш-таблицы, после чего ее размер увеличивается. В частности, размер

таблицы увеличится, когда количество элементов станет больше емкости таблицы, умноженной на ее коэффициент заполнения. Для конструкторов, которые в качестве параметра не принимают коэффициент заполнения, используется значение 1,0. В классе Hashtable помимо методов, определенных в реализованных им интерфейсах, также определены собственные методы. Наиболее употребимые из них перечислены в таблице 3. Чтобы определить, содержит ли Hashtable-коллекция заданный ключ, достаточно вызвать метод ContainsKey (). А чтобы узнать, хранится ли в интересующей вас хеш-таблице заданное значение, вызовите метод ContainsValue (). Для опроса элементов Hashtable-коллекции необходимо получить нумератор типа IDictionaryEnumerator, вызвав метод GetEnumerator (). Вспомните, что для опроса содержимого коллекции, в которой хранятся пары ключ/значение, используется именно класс IDictionaryEnumerator.

Таблица 3 – Собственные методы класса Hashtable

public virtual bool ContainsKey (object k)	Возвращает значение true, если в вызывающей Hashtable-коллекции содержится ключ, заданный параметром k. В противном случае возвращает значение false
public virtual bool ContainsValue (object v)	Возвращает значение true, если в вызывающей Hashtable-коллекции содержится значение, заданное параметром v. В противном случае возвращает значение false
public virtual IDictionaryEnumerator GetEnumerator()	Возвращает для вызывающей Hashtable-коллекции нумератор типа IDictionaryEnumerator.
public static Hashtable Synchronized (Hashtable ht)	Возвращает синхронизированную версию вызывающей Hashtable-коллекции, переданной в параметре ht

В классе Hashtable, помимо свойств, определенных в реализованных им интерфейсах, также определены два собственных public-свойства. Используя следующие свойства, можно из Hashtable-коллекции получить коллекцию ключей или значений:

public virtual ICollection Keys { get; }

public virtual ICollection Values { get; }

Поскольку в классе Hashtable не предусмотрена поддержка упорядоченных коллекций, при получении коллекции ключей или значений заданный порядок элементов не достигается. В классе Hashtable также определены два protected-свойства с именами Hcp и Comparer, которые доступны для производных классов.

В классе Hashtable пары ключ/значение хранятся в форме структуры типа DictionaryEntry, но по большей части вас это не будет касаться, поскольку свойства и методы обрабатывают ключи и значения отдельно. Например, при добавлении элемента в Hashtable-коллекцию необходимо вызвать метод Add(), который принимает два отдельных аргумента: ключ и значение.

Важно отметить, что Hashtable-коллекция не гарантирует сохранения порядка элементов. Дело в том, что хеширование обычно не применяется к отсортированным таблицам.

Рассмотрим пример, который демонстрирует использование Hashtable-коллекции:

```
// Демонстрация использования Hashtable-коллекции.
using System;
using System.Collections;
class HashtableDemo {
public static void Main() {
// Создаем хэш-таблицу.
Hashtable ht = new Hashtable();
// Добавляем элементы в хэш-таблицу.
ht.Add("здание", "жилое помещение");
ht.Add("автомобиль", "транспортное средство");
ht.Add("книга", "набор печатных слов");
ht.Add("яблоко", "съедобный фрукт");
// Добавлять элементы можно также с помощью индексатора,
ht["трактор"] = "сельскохозяйственная машина";
// Получаем коллекцию ключе*й.
ICollection c = ht.Keys;
```

```
// Используем ключи для получения значений,
foreach(string str in c)
Console.WriteLine(str + ": " + ht[str]);
}
}
```

Результаты выполнения этой программы таковы:

```
яблоко: съедобный фрукт
здание: жилое помещение
трактор: сельскохозяйственная машина
автомобиль: транспортное средство
книга: набор печатных слов
```

Как видно по приведенным результатам, пары ключ/значение хранятся отнюдь не в упорядоченном виде. Обратите внимание на то, как было получено и отображено содержимое хеш-таблицы ht. Во-первых, коллекция ключей считывается с помощью свойства Keys. Каждый ключ затем используется в качестве индекса хеш-таблицы ht, который позволяет найти и отобразить значение, соответствующее каждому ключу. Не забывайте, что индексатор, определенный в интерфейсе IDictionary и реализованный в классе Hashtable, использует ключ в роли индекса.

2.2.3. Класс SortedList

Класс SortedList предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу. Класс SortedList реализует интерфейсы IDictionary, ICollection, IEnumerable и ICloneable. В классе SortedList определено несколько конструкторов, включая следующие:

public SortedList()

public SortedList(IDictionary c)

public SortedList(int capacity)

public SortedList(IComparer comp)

Первый конструктор позволяет создать пустую коллекцию с начальной емкостью, равной 16 элементам. Второй создает SortedList-коллекцию, которая инициализируется элементами и емкостью коллекции, заданной параметром c. Третий конструктор предназначен для построения пустого SortedList-списка, который инициализируется емкостью, заданной параметром capacity. Как вы помните, емкость— это размер базового массива, который используется для хранения элементов коллекции. Четвертая форма конструктора позволяет задать метод сравнения, который должен использоваться для сравнения объектов списка. С помощью этой формы создается пустая коллекция с начальной емкостью, равной 16 элементам.

Емкость SortedList-коллекции увеличивается автоматически, если в этом возникает необходимость, при добавлении элементов. Если окажется, что текущая емкость может быть превышена, она удваивается. Преимущество задания емкости при создании SortedList-списка состоит в минимизации затрат системных ресурсов, связанных с изменением размера коллекции. Конечно, задавать начальную емкость имеет смысл только в том случае, если вы знаете, какое количество элементов должно храниться в коллекции.

В классе SortedList помимо методов, определенных в реализованных им интерфейсах, также определены собственные методы. Наиболее употребимые приведены в таблице 4. Чтобы определить, содержит ли SortedList-коллекция заданный ключ, достаточно вызвать метод ContainsKey(). А чтобы узнать, хранится ли в списке заданное значение, вызовите метод ContainsValue(). Для опроса элементов SortedList-коллекции необходимо получить нумератор типа IDictionaryEnumerator, вызвав метод GetEnumerator (). Вспомните, что для опроса содержимого коллекции, в которой хранятся пары ключ/значение, используется именно класс IDictionaryEnumerator. Синхронизированную версию SortedList-коллекции можно получить с помощью метода Synchronized().

Таблица 4 – Собственные методы класса SortedList

public virtual bool ContainsKey(object k)	Возвращает значение true, если в вызывающей SortedList-коллекции содержится ключ, заданный параметром k. В противном случае возвращает значение false
public virtual bool ContainsValue(object v)	Возвращает значение true, если в вызывающей SortedList-коллекции содержится значение, заданное параметром k. В противном случае возвращает значение false

public virtual object GetBylndex(int idx)	Возвращает значение, индекс которого задан параметром idx
public virtual IDictionaryEnumerator GetEnumerator()	Возвращает нумератор типа IDictionaryEnumerator для вызывающей SortedList-коллекции
public virtual object GetKeyUnt idx)	Возвращает ключ, индекс которого задан параметром idx
public virtual iList GetKeyList()	Возвращает iList-коллекцию ключей, хранимых в вызывающей SortedList-коллекции
public virtual IList GetValueList()	Возвращает iList-коллекцию значений, хранимых в вызывающей SortedList-коллекции
public virtual int IndexOfKey(object k)	Возвращает индекс ключа, заданного параметром к. Возвращает значение -1, если в списке нет заданного ключа
public virtual int IndexOfValue(object v)	Возвращает индекс первого вхождения значения, заданного параметром v. Возвращает -1, если в списке нет заданного ключа
public virtual void SetBylndex(int idx, object v)	Устанавливает значение по индексу, заданному параметром idx, равным значению, переданному в параметре v
public static SortedList Synchronized(SortedList sl)	Возвращает синхронизированную версию sortedList-коллекции, переданной в параметре sl
public virtual void TrimToSize()	Устанавливает свойство capacity равным значению свойства Count

Существуют различные способы установки и считывания ключа либо значения. Чтобы получить значение, связанное с заданным индексом, вызовите метод

GetBylndex (), а чтобы установить значение, заданное индексом, — метод SetBylndex (). Получить ключ, связанный с заданным индексом, можно с помощью метода GetKey (). Для получения списка всех ключей используйте метод GetKeyList (), а для получения списка всех значений— метод GetValueList (). Получить индекс ключа можно с помощью метода IndexOf Key (), а индекс значения— с помощью метода IndexOf Value (). Класс SortedList также поддерживает индексатор, определенный интерфейсом iDictionary, благодаря чему можно устанавливать или считывать значение, заданное соответствующим ключом. В классе SortedList помимо свойств, определенных в реализованных им интерфейсах, определены два собственных свойства. Получить предназначенную только для чтения коллекцию ключей или значений, хранимых в SortedList-коллекции, можно с помощью таких свойств:

public virtual ICollection Keys { get; }

public virtual ICollection Values { get; }

Порядок следования ключей и значений в полученных коллекциях отражает порядок SortedList-коллекции. Подобно Hashtable -коллекции, SortedList-список хранит пары ключ/значение в форме структуры типа DictionaryEntry, но с помощью методов и свойств, определенных в классе SortedList, программисты обычно получают отдельный доступ к ключам и значениям. Использование отсортированного списка типа SortedList демонстрируется в программе, которая представляет собой переработанную и расширенную версию программы из предыдущего раздела, демонстрировавшей Hashtable-коллекцию. Изучая результаты выполнения этой программы, обратите внимание на то, что SortedList-коллекция отсортирована по ключу.

```
// Демонстрация SortedList-коллекции.
using System;
using System.Collections;
class SLDemo {
public static void Main() {
// Создаем упорядоченную коллекцию типа SortedList.
SortedList si = new SortedList();
// Добавляем в список элементы.
sl.Add("здание", "жилое помещение");
sl.Add("автомобиль", "транспортное средство");
sl.Add("книга", "набор печатных слов");
```

```
sl.Add("яблоко", "съедобный фрукт");
// Добавлять элементы можно также с помощью индексатора
sl["трактор"] = "сельскохозяйственная машина";
// Получаем коллекцию ключей.
ICollection c = sl.Keys;
// Используем ключи для получения значений.
Console.WriteLine(
"Содержимое списка, полученное с помощью " +
"индексатора.");
foreach(string str in c)
Console.WriteLine(str + ": " + sl[str]);
Console.WriteLine();
// Отображаем список, используя целочисленные индексы.
Console.WriteLine(
"Содержимое списка, полученное с помощью " +
"целочисленных индексов.");
for(int i=0; i<sl.Count; i++)
Console.WriteLine(sl.GetBylndex(i));
Console.WriteLine();
// Отображаем целочисленные индексы элементов списка.
Console.WriteLine(
"Целочисленные индексы элементов списка.");
foreach(string str in c)
Console.WriteLine(str + ": " + sl.IndexOfKey(str));
}
}
```

Результаты выполнения этой программы таковы:

Содержимое списка, полученное с помощью индексатора.

автомобиль: транспортное средство

здание: жилое помещение

книга: набор печатных слов

трактор: сельскохозяйственная машина

яблоко: съедобный фрукт

Содержимое списка, полученное с помощью целочисленных индексов.

транспортное средство

жилое помещение

набор печатных слов

сельскохозяйственная машина

съедобный фрукт

Целочисленные индексы элементов списка.

автомобиль: 0

здание: 1

книга: 2

трактор: 3

яблоко: 4

Класс Stack

Вероятно, большинству читателей известно, что стек представляет собой список, добавление и удаление элементов к которому осуществляется по принципу "последним пришел — первым обслужен" (last-in, first-out — LIFO). Чтобы понять, как работает стек, представьте себе груду тарелок на столе. Тарелку, поставленную на стол первой, можно будет взять лишь последней, т.е. когда будут сняты все поставленные сверху тарелки. Стек — наиболее востребованная структура данных в программировании. Ее часто используют в системном программном обеспечении, компиляторах и программах из области создания искусственного интеллекта (в частности, в сфере программирования с обратным слежением).

Класс коллекции, предназначенный для поддержки стека, называется Stack. Он реализует интерфейсы ICollection, IEnumerable и ICloneable. Стек — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять

для хранения новые элементы, причем каждый раз, когда стек должен расшириться, его емкость удваивается.

В классе stack определены следующие конструкторы:

public Stack()

public Stack(int capacity)

public Stack(ICollection c)

Первый конструктор предназначен для создания пустого стека с начальной емкостью, равной 10 элементам. Второй создает пустой стек с начальной емкостью, заданной параметром capacity. Третий конструктор служит для построения стека, который инициализируется элементами и емкостью коллекции, заданной параметром с. Помимо методов, определенных в интерфейсах, которые реализует класс stack, в нем определены также собственные методы, перечисленные в таблице 5. По описанию этих методов можно судить о том, как используется стек Чтобы поместить объект в вершину стека, вызовите метод Push(). Чтобы извлечь верхний элемент и удалить его из стека, используйте метод Pop(). Если при вызове метода Pop() окажется, что вызывающий стек пуст, генерируется исключение типа InvalidOperationException. Метод Peek() позволяет вернуть верхний элемент, не удаляя его из стека.

Таблица 5 – Собственные методы класса Stack

public virtual bool Contains(object v)	Возвращает значение true, если объект v содержится в вызывающем стеке. В противном случае возвращает значение false
public virtual void Clear()	Устанавливает свойство count равным нулю, тем самым эффективно очищая стек
public virtual object Peek()	Возвращает элемент, расположенный в вершине стека, но не удаляет его
public virtual object Pop()	Возвращает элемент, расположенный в вершине стека, и удаляет его

public virtual void Push(object v)	Помещает объект v в стек
public static Stack Synchronized(Stack stk)	Возвращает синхронизированную версию stack-списка, переданного в параметре stk
public virtual object[] ToArray()	Возвращает массив, который содержит копии элементов вызывающе-го стека

Рассмотрим пример создания стека и его использования: поместим в него несколько объектов класса Integer, а затем извлечем их.

```
// Демонстрация использования класса Stack.
using System;
using System.Collections;
class StackDemo {
static void showPush(Stack st, int a) {
st.Push(a);
Console.WriteLine(
"Помещаем в элемент стек: Push(" + a + ")"
Console.Write("Содержимое стека: ") ;
foreach(int i in st)
Console.Write(i + " ") ;
Console.WriteLine();
static void showPop(Stack st) {
Console.Write("Извлекаем элемент из стека: Pop -> ")
int a = (int) st.Pop();
Console.WriteLine(a);
Console.Write("Содержимое стека: ") ;
foreach(int i in st)
```

```
Console.Write(i + " ") ;
Console.WriteLine();
public static void Main() {
Stack st = new Stack();
foreach(int i in st)
Console.Write(i + " ");
Console.WriteLine();
showPush (st, 22);
showPush(st, 65);
showPush(st, 91);
showPop(st);
showPop(st);
showPop(st);
try {
showPop(st);
} catch (InvalidOperationException) {
Console.WriteLine("Стек пуст.") ;
}
}
}
```

Ниже приведены результаты выполнения этой программы. Обратите внимание на то, как обрабатывается исключительная ситуация (исключение типа InvalidOperationException), возникающая при попытке извлечь элемент из пустого стека (эта ситуация называется незагруженностью стека).

```
Помещаем элемент в стек: Push(22)
Содержимое стека: 22
Помещаем элемент в стек: Push(65) ^
Содержимое стека: 65 22
Помещаем элемент в стек: Push(91)
```

Содержимое стека: 91 65 22

Извлекаем элемент из стека: Pop -> 91

Содержимое стека: 65 22

Извлекаем элемент из стека: Pop -> 65

Содержимое стека: 22

Извлекаем элемент из стека: Pop -> 22

Содержимое стека:

Извлекаем элемент из стека: Pop -> Стек пуст.

2.2.5. Класс Queue

Еще одной распространенной структурой данных является очередь. Добавление элементов в очередь и удаление их из нее осуществляется по принципу "первым пришел — первым обслужен" (first-in, first-out — FIFO). Другими словами, первый элемент, помещенный в очередь, первым же из нее и извлекается. Ну кто не сталкивался с очередями в реальной жизни? Например, каждому приходилось, вероятно, стоять в очереди к билетной кассе в кинотеатре или к кассе в супермаркете, чтобы оплатить покупку. В программировании очереди используются для организации таких механизмов, как выполнение нескольких процессов в системе и поддержка списка незаконченных транзакций (в системах ведения баз данных) или пакетов данных, полученных из Internet. Очереди также часто используются в области имитационного моделирования.

Класс коллекции, предназначенный для поддержки очереди, называется Queue. Он реализует интерфейсы iCollection, IEnumerable и ICloneable. Очередь — это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда такая необходимость возникает, текущий размер очереди умножается на коэффициент роста, который по умолчанию равен значению 2,0.

В классе Queue определены следующие конструкторы:

public Queue()

public Queue (int capacity)

public Queue (int capacity, float growFact)

public Queue (ICollection c)

Первый конструктор предназначен для создания пустой очереди с начальной емкостью, равной 32 элементам, и коэффициентом роста 2,0. Второй создает пустую очередь с начальной емкостью, заданной параметром capacity, и коэффициентом роста 2,0. Третий отличается от второго тем, что позволяет задать коэффициент роста посредством параметра growFact. Четвертый конструктор служит для создания очереди, которая инициализируется элементами и емкостью коллекции, заданной параметром c.

Помимо методов, определенных в интерфейсах, которые реализует класс Queue, в нем определены также собственные методы, перечисленные в таблице 6. О работе очереди можно получить представление по описанию этих методов. Чтобы поместить объект в очередь, вызовите метод Enqueue(). Чтобы извлечь верхний элемент и удалить его из очереди, используйте метод Dequeue(). Если при вызове метода Dequeue() окажется, что вызывающая очередь пуста, генерируется исключение типа InvalidOperationException. Метод Peek() позволяет вернуть следующий объект, не удаляя его из очереди.

Таблица 6 - Собственные методы класса Queue

public virtual bool Contains (object v)	Возвращает значение true, если объект v содержится в вызывающей очереди. В противном случае возвращает значение false
public virtual void clear ()	Устанавливает свойство Count равным нулю, тем самым эффективно очищая очередь
public virtual object Dequeue ()	Возвращает объект из начала вызывающей очереди, Возвращаемый объект из очереди удаляется
public virtual void Enqueue(object v)	Добавляет объект v в конец очереди

public virtual object Peek ()	Возвращает объект из начала вызывающей очереди, но не удаляет его
public static Queue Synchronized(Queue q)	Возвращает синхронизированную версию очереди, заданной параметром g
public virtual ob j ect [ ] тоАггау ()	Возвращает массив, который содержит копии элементов из вызывающей очереди
public virtual void TrimToSize()	Устанавливает свойство Capacity равным значению свойства Count

Рассмотрим пример, в котором демонстрируется использование класса Queue:

```
// Демонстрация класса Queue.
using System;
using System.Collections;
class QueueDemo {
static void showEnq(Queue q, int a) {
q.Enqueue(a) ;
Console.WriteLine(
"Помещаем элемент в очередь: Enqueue(" + a + ")");
Console.Write("Содержимое очереди: ") ;
foreach(int i in q)
Console.Write(i + " ") ;
Console.WriteLine();
static void showDeq(Queue q) {
Console.Write(
"Извлекаем элемент из очереди: Dequeue -> ") ;
int a = (int) q.Dequeue();
Console.WriteLine(a);
```

```
Console.Write("Содержимое очереди: ") ;
foreach(int i in q)
Console.Write(i + " ") ;
Console.WriteLine();
public static void Main() {
Queue q = new Queue();
foreach(int i in q)
Console.Write(i + " ") ;
Console.WriteLine() ;
showEnq(q, 22) ;
showEnq(q, 65) ;
showEnq(q, 91);
showDeq(q);
showDeq(q);
showDeq(q);
try {
showDeq(q);
} catch (InvalidOperationException) {
Console.WriteLine("Очередь пуста.");
}
}
}
```

Результаты выполнения программы таковы:

Помещаем элемент в очередь: Enqueue(22)

Содержимое очереди: 22

Помещаем элемент в очередь: Enqueue(65)

Содержимое очереди: 22 65

Помещаем элемент в очередь: Enqueue(91)

Содержимое очереди: 22 65 91

Извлекаем элемент из очереди: Dequeue -> 22

Содержимое очереди: 65 91

Извлекаем элемент из очереди: Dequeue -> 65

Содержимое очереди: 91

Извлекаем элемент из очереди: Dequeue -> 91

Содержимое очереди:

Извлекаем элемент из очереди: Dequeue -> Очередь пуста.

Класс List<T>

Класс List<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

void Add(T item): добавление нового элемента в список

void AddRange(ICollection collection): добавление в список коллекции или массива

int BinarySearch(T item): бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.

int IndexOf(T item): возвращает индекс первого вхождения элемента в списке

void Insert(int index, T item): вставляет элемент item в списке на позицию index

bool Remove(T item): удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true

void RemoveAt(int index): удаление элемента по указанному индексу index

void Sort(): сортировка списка

В примерах этого раздела используется универсальный класс List<T>, который позволяет работать со строго типизированными списками объектов.

В приведенном ниже примере создается список строк, а затем выполняется перебор строк с помощью оператора foreach.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
   Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Для перебора коллекции можно использовать оператор for вместо оператора foreach. Для этого доступ к элементам коллекции осуществляется по позиции индекса. Индекс элементов начинается с 0 и заканчивается числом, равным количеству элементов минус 1.

В приведенном ниже примере выполняется перебор элементов коллекции с помощью оператора for вместо foreach.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
   Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

В приведенном ниже примере элемент удаляется из коллекции путем указания удаляемого объекта.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };
```

```
// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
   Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

Для типа элементов в List<T> можно также определить собственный класс. В приведенном ниже примере класс Galaxy, который используется объектом List<T>, определен в коде.

```
private static void IterateThroughList()
{
   var theGalaxies = new List<Galaxy>
      {
         new Galaxy() { Name="Tadpole", MegaLightYears=400},
         new Galaxy() { Name="Pinwheel", MegaLightYears=25},
         new Galaxy() { Name="Milky Way", MegaLightYears=0},
         new Galaxy() { Name="Andromeda", MegaLightYears=3}
      };

   foreach (Galaxy theGalaxy in theGalaxies)
   {
      Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.Mega-
LightYears);
   }

   // Output:
   //  Tadpole  400
   //  Pinwheel  25
   //  Milky Way  0
   //  Andromeda  3
}

public class Galaxy
{
   public string Name { get; set; }
   public int MegaLightYears { get; set; }
}
```

Обобщающий пример работы с списком List:

```
using System;
using System.Collections.Generic;

namespace Collections
{
   class Program
   {
      static void Main(string[] args)
      {
         List<int> numbers = new List<int>() { 1, 2, 3, 45 };
         numbers.Add(6); // добавление элемента

         numbers.AddRange(new int[] { 7, 8, 9 });

         numbers.Insert(0, 666); // вставляем на первое место в списке
число 666

         numbers.RemoveAt(1); //  удаляем второй элемент

         foreach (int i in numbers)
         {
            Console.WriteLine(i);
         }

         List<Person> people = new List<Person>(3);
         people.Add(new Person() { Name = "Том" });
         people.Add(new Person() { Name = "Билл" });

         foreach (Person p in people)
         {
            Console.WriteLine(p.Name);
         }

         Console.ReadLine();
      }
   }

   class Person
   {
      public string Name { get; set; }
   }
}
```

Здесь создаются два списка: один для объектов типа int, а другой - для объектов Person. В первом случае выполняется начальная инициализация списка:

```
List<int> numbers = new List<int>() { 1, 2, 3, 45 };
```

Во втором случае используется другой конструктор, в который передается начальная емкость списка:

```
List<Person> people = new List<Person>(3);
```

Указание начальной емкости списка (capacity) позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Также начальную емкость можно установить с помощью свойства Capacity, которое имеется у класса List.

Не рекомендуется использовать ArrayList класс для новой разработки. Вместо этого рекомендуется использовать универсальный List<T> класс. ArrayListКласс предназначен для хранения разнородных коллекций объектов. Однако это не всегда обеспечивает наилучшую производительность. Вместо этого рекомендуется следующее:

Для разнородной коллекции объектов используйте List<Object> тип (в C#).

Для однородной коллекции объектов используйте List<T> класс.

Задание на лабораторную работу

Задание 1

Вариант	**№**	**Задания**
Вариант № 1	1	Определить, какое из двух слов длиннее и на сколько.
	2	Дана последовательность, содержащая от 1 до 30 слов, в каждом из которых от 1 до 5 строчных латинских букв; между соседними словами -- запятая, за последним словом -- точка. Напечатать: эту же последовательность слов, но в обратном порядке;
	3	Удвоить все согласные буквы.
	1	Определить, является ли какое-нибудь из двух слов частью другого.

Вариант № 2	2	Дана последовательность, содержащая от 1 до 30 слов, в каждом из которых от 1 до 5 строчных латинских букв; между соседними словами -- запятая, за последним словом -- точка. Напечатать: слова из последовательности, расположив их в алфавитном порядке
	3	Удалить из данного слова все согласные буквы.
Вариант № 3	1	Определить, есть ли в записи квадрата данного числа цифра n
	2	Дана последовательность, содержащая от 1 до 30 слов, в каждом из которых от 1 до 5 строчных латинских букв; между соседними словами -- запятая, за последним словом -- точка. Напечатать: все слова, которые встречаются в последовательности по одному разу;
	3	Удвоить все гласные буквы.
Вариант № 4	1	Поменять в слове первую и последнюю буквы
	2	Дана последовательность, содержащая от 1 до 30 слов, в каждом из которых от 1 до 5 строчных латинских букв; между соседними словами -- запятая, за последним словом -- точка. Напечатать: все слова, которые встречаются в последовательности по несколько раз.
	3	Удалить из данного слова все гласные буквы.
Вариант № 5	1	По последнему символу определить тип предложения (повествовательное, вопросительное, восклицательное).
	2	Дана последовательность, содержащая от 2 до 50 слов, в каждом из которых от 1 до 8 строчных латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать те слова последовательности, которые отличны от последнего слова и удовлетворяют следующему свойству: длина слова максимальна;
	3	Проверить, имеются ли в данном слове одинаковые буквы.
Вариант № 6	1	Заменить в арифметическом выражении знаки "+" на знаки "-", а знаки "-" на знаки "+".
	2	Дана последовательность, содержащая от 2 до 50 слов, в каждом из которых от 1 до 8 строчных латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать те слова последовательности, которые отличны от последнего слова и удовлетворяют следующему свойству: в слове гласные буквы (a, e, i, o, u) чередуются с согласными.
	3	Для подсчета количества слов в предложении, учитывая что между словами может быть несколько пробелов.
Вариант № 7	1	Удалить все буквы "f" в данном слове.
	2	Дана последовательность, содержащая от 2 до 50 слов, в каждом из которых от 1 до 8 строчных латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка.

		Напечатать те слова последовательности, которые отличны от последнего слова и удовлетворяют следующему свойству: слово симметрично;
	3	Выяснить, можно ли из символов заданного слова составить заданное слово
Вариант № 8	1	Если в слове нечетное число букв, то удвоить среднюю.
	2	Дана последовательность, содержащая от 2 до 50 слов, в каждом из которых от 1 до 8 строчных латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать те слова последовательности, которые отличны от последнего слова и удовлетворяют следующему свойству: длина слова минимальна;
	3	Даны две литеры -- латинская буква (от a до h) и цифра (от 1 до 8), например a2 или g5. Рассматривая их как координаты поля шахматной доски, на котором находится пешка, вывести значения клеток поля, которые находятся под ударом пешки.
Вариант № 9	1	Удалить все пробелы в данном предложении.
	2	Дана последовательность, содержащая от 2 до 30 слов, в каждом из которых от 2 до 10 латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать все слова, отличные от последнего слова, предварительно преобразовав каждое из них по следующему правилу: удалить из слова все последующие вхождения первой буквы;
	3	Заменить в тексте все маленькие латинские буквы на большие.
Вариант № 10	1	Поменять в слове первую и последнюю буквы
	2	Дана последовательность, содержащая от 2 до 30 слов, в каждом из которых от 2 до 10 латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать все слова, отличные от последнего слова, предварительно преобразовав каждое из них по следующему правилу: удалить из слова первую и последнюю буквы;
	3	Заменить в тексте все большие латинские буквы на маленькие.
Вариант № 11	1	Удвоить каждую букву данного слова.
	2	Дана последовательность, содержащая от 2 до 30 слов, в каждом из которых от 2 до 10 латинских букв; между соседними словами -- не менее одного пробела, за последним словом -- точка. Напечатать все слова, отличные от последнего слова, предварительно преобразовав каждое из них по следующему правилу: если слово нечетной длины, то удалить его среднюю букву.

	3	Даны две литеры -- латинская буква (от a до h) и цифра (от 1 до 8), например a2 или g5. Рассматривая их как координаты поля шахматной доски, на котором находится конь, вывести значения клеток поля, которые находятся под ударом коня.

Задание 2

Создать однородную (однотипную) коллекцию объектов List<T> и добавить основные методы работы с ним.

Задание 3

Создать разнородную коллекцию объектов List<T>. Для этого создать класс описания свойств объекта согласно предметной области варианта, который будет использоваться объектом List<T>.

Вариант	**Класс**
1	Человек
2	Студент
3	Преподаватель
4	Мотоцикл
5	Легковой автомобиль
6	Грузовой автомобиль
7	Вертолет
8	Самолет
9	Жилой дом
10	Административное здание
11	Завод

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Выполнить полученное задание в консольном приложении (.NET Framework).

5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.
7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 7

«Разработка пользовательских интерфейсов для программирования подпрограмм на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ программирования подпрограмм на языке C# на платформе .NET Framework.

Краткие теоретические сведения

Понятие процедур и функций

В языке C# нет специальных ключевых слов - procedure и function, но присутствуют сами эти понятия. Синтаксис объявления метода позволяет однозначно определить, чем является метод - процедурой или функцией.

Функция отличается от процедуры двумя особенностями:

1. Она всегда вычисляет некоторое значение, возвращаемое в качестве результата функции;

2. И вызывается в выражениях.

Пример:

```
public static int GetAverage(int[] array)
```

Процедура C# имеет свои особенности:

1. Она возвращает формальный результат void, указывающий на отсутствие результата;

2. Вызов процедуры является оператором языка;

3. И она имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

Обычно метод предпочитают реализовать в виде функции тогда, когда он имеет один выходной аргумент, рассматриваемый как результат вычисления значения функции. Возможность вызова функций в выражениях также влияет на выбор в пользу реализации метода в виде функции. В других случаях метод реализуют в виде процедуры.

Пример:

```
public static void Srednee(int[] mas)
```

Методы

Метод представляет собой блок кода, содержащий набор инструкций. Программа инициирует выполнение операторов, вызывая метод и задавая необходимые аргументы метода. В C# все инструкции выполняются в контексте метода. Метод Main является точкой входа для каждого приложения C#, и вызывается он средой CLR при запуске программы.

Следует отметить, что официальная терминология C# делает различие между функциями и методами. Согласно этой терминологии, понятие "функция-член" включает не только методы, но также другие члены, не являющиеся данными, класса или структуры. Сюда входят индексаторы, операции, конструкторы, деструкторы, а также — возможно, несколько неожиданно — свойства. Они контрастируют с данными-членами: полями, константами и событиями.

Объявление методов

В C# определение метода состоит из любых модификаторов (таких как спецификация доступности), типа возвращаемого значения, за которым следует имя метода, затем списка аргу-ментов в круглых скобках и далее — тела метода в фигурных скобках:

```
[модификаторы] тип_возврата ИмяМетода([параметры])
{
// Тело метода
}
```

Каждый параметр состоит из имени типа параметра и имени, по которому к нему можно обратиться в теле метода. Вдобавок, если метод возвращает значение, то для указания точки выхода должен использоваться оператор возврата return вместе с возвращаемым значением.

Если метод не возвращает ничего, то в качестве типа возврата указывается void, поскольку вообще опустить тип возврата невозможно. Если же он не принимает аргументов, то все равно после имени метода должны присутствовать пустые круглые скобки. При этом включать в тело метода оператор возврата не обязательно — метод возвращает управление автоматически по достижении закрывающей фигурной скобки.

Возврат из метода и возврат значения

В целом, возврат из метода может произойти при двух условиях. Во-первых, когда встречается фигурная скобка, закрывающая тело метода. И во-вторых, когда выполняется оператор return. Имеются две формы оператора return: одна — для методов типа void (возврат из метода), т.е. тех методов, которые не возвращают значения, а другая — для методов, возвращающих конкретные значения (возврат значения).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class MyMathOperation
   {
      public double r;
      public string s;

      // Возвращает площадь круга
      public double sqrCircle()
      {
         return Math.PI * r * r;
      }

      // Возвращает длину окружности
      public double longCircle()
      {
         return 2 * Math.PI * r;
      }

      public void writeResult()
      {
```

```
            Console.WriteLine("Вычислить площадь или длину? s/l:");
            s = Console.ReadLine();
            s = s.ToLower();
            if (s == "s")
            {
               Console.WriteLine("Площадь круга равна {0:#.###}",sqrCir-
cle());
               return;
            }
            else if (s == "l")
            {
               Console.WriteLine("Длина окружности равна {0:#.##}",longCir-
cle());
               return;
            }
            else
            {
               Console.WriteLine("Вы ввели не тот символ");
            }
         }
      }

      class Program
      {
         static void Main(string[] args)
         {
            Console.WriteLine("Введите радиус: ");
            string radius = Console.ReadLine();

            MyMathOperation newOperation = new MyMathOperation { r = dou-
ble.Parse(radius) };
            newOperation.writeResult();

            Console.ReadLine();
         }
      }
   }
```

Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

```
   class Program
   {
      static void Main(string[] args)
      {
```

```
            Sum(10, 15);        // параметры передаются по значению
            Console.ReadKey();
        }
        static int Sum(int x, int y)
        {
            return x + y;
        }
    }
```

Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор ref:

```
static void Main(string[] args)
{
    int x = 10;
    int y = 15;
    Addition(ref x, y); // вызов метода
    Console.WriteLine(x);   // 25

    Console.ReadLine();
}
// параметр x передается по ссылке
static void Addition(ref int x, int y)
{
    x += y;
}
```

Обратите внимание, что модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

Сравнение передачи по значению и по ссылке

В чем отличие двух способов передачи параметров? При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

Рассмотрим два аналогичных примера. Первый пример - передача параметра по значению:

```
class Program
{
   static void Main(string[] args)
   {
      int a = 5;
      Console.WriteLine($"Начальное значение переменной a = {a}");

      //Передача переменных по значению
      //После выполнения этого кода по-прежнему a = 5, так как мы передали лишь ее копию
      IncrementVal(a);
      Console.WriteLine($"Переменная a после передачи по значению равна = {a}");
      Console.ReadKey();
   }
   // передача по значению
   static void IncrementVal(int x)
   {
      x++;
      Console.WriteLine($"IncrementVal: {x}");
   }
}
```

Консольный вывод:

```
Начальное значение переменной a = 5
IncrementVal: 6
Переменная a после передачи по значению равна = 5
```

При вызове метод IncrementVal получает копию переменной a и увеличивает значение этой копии. Поэтому в самом методе IncrementVal мы видим, что значение параметра x увеличилось на 1, но после выполнения метода переменная a имеет прежнее значение - 5. То есть изменяется копия, а сама переменная не изменяется.

Второй пример - аналогичный метод с передачей параметра по ссылке:

```
class Program
{
   static void Main(string[] args)
   {
      int a = 5;
      Console.WriteLine($"Начальное значение переменной a  = {a}");
      //Передача переменных по ссылке
      //После выполнения этого кода a = 6, так как мы передали саму
переменную
      IncrementRef(ref a);
      Console.WriteLine($"Переменная a после передачи ссылке равна =
{a}");

      Console.ReadKey();
   }
   // передача по ссылке
   static void IncrementRef(ref int x)
   {
      x++;
      Console.WriteLine($"IncrementRef: {x}");
   }
}
```

Консольный вывод:

```
Начальное значение переменной a = 5
IncrementRef: 6
Переменная a после передачи по ссылке равна = 6
```

В метод IncrementRef передается ссылка на саму переменную a в памяти. И если значение параметра в IncrementRef изменяется, то это приводит и к изменению переменной a, так как и параметр и переменная указывают на один и тот же адрес в памяти.

Выходные параметры. Модификатор out

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out:

```
static void Sum(int x, int y, out int a)
{
   a = x + y;
}
```

Здесь результат возвращается не через оператор return, а через выходной параметр. Использование в программе:

```
static void Main(string[] args)
{
   int x = 10;

   int z;

   Sum(x, 15, out z);

   Console.WriteLine(z);

   Console.ReadKey();
}
```

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:

```
static void Sum(int x, int y, out int a)
{
   Console.WriteLine(x+y);
}
```

Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не один вариант, а несколько. Например:

```
static void Main(string[] args)
{
   int x = 10;
   int area;
   int perimetr;
   GetData(x, 15, out area, out perimetr);
   Console.WriteLine("Площадь : " + area);
   Console.WriteLine("Периметр : " + perimetr);

   Console.ReadKey();
}
static void GetData(int x, int y, out int area, out int perim)
{
   area= x * y;
   perim= (x + y)*2;
}
```

Здесь у нас есть метод GetData, который, допустим, принимает стороны прямоугольника. А два выходных параметра мы используем для подсчета площади и периметра прямоугольника.

По сути, как и в случае с ключевым словом ref, ключевое слово out применяется для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

Стоит отметить, что начиная с версии C# 7.0 можно определять переменные в непосредственно при вызове метода. То есть вместо:

```
int x = 10;
int area;
int perimetr;
GetData(x, 15, out area, out perimetr);
Console.WriteLine($"Площадь : {area}");
Console.WriteLine($"Периметр : {perimetr}");
```

Мы можем написать:

```
int x = 10;
GetData(x, 15, out int area, out int perimetr);
Console.WriteLine($"Площадь : {area}");
Console.WriteLine($"Периметр : {perimetr}");
```

Входные параметры. Модификатор in

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором in. Модификатор in указывает, что данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить. Например, возьмем следующий метод:

```
static void GetData(in int x, int y, out int area, out int perim)
{
    // x = x + 10; нельзя изменить значение параметра x
    y = y + 10;
    area = x * y;
    perim = (x + y) * 2;
}
```

В данном случае через параметры x и y в метод передаются значения, но в самом методе можно изменить только значение параметра y, так как параметр x указан с модификатором in.

Перегрузка методов

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов. Перегрузка методов относится к одному из способов реализации полиморфизма в C#.

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Сигнатура складывается из следующих аспектов:

- Имя метода
- Количество параметров
- Типы параметров
- Порядок параметров
- Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

```
public int Sum(int x, int y)
{
   return x + y;
}
```

У данного метода сигнатура будет выглядеть так: Sum(int, int)

И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

- Количеству параметров
- Типу параметров
- Порядку параметров
- Модификаторам параметров

Таким образом, перегрузка методов – это объявление в классе методов с одинаковыми именами при этом с различными параметрами.

Имея некий метод, чтобы его перегрузить, другой метод с таким же именем должен отличаться от него количеством параметров и/или типами параметров. Отличия только типами возвращаемых значений методами недостаточно для перегрузки, но если методы отличаются параметрами, тогда перегружаемые методы могут иметь и различные типы возвращаемых значений.

Пример того, как может быть перегружен метод:

```
public void SomeMethod()
{
  // тело метода
}

public void SomeMethod(int a) // от первого отличается наличием параметра
{
  // тело метода
}

public void SomeMethod(string s) // от второго отличается типом параметра
{
  // тело метода
}
```

```
public int SomeMethod(int a, int b) // от предыдущих отличается количеством параметров (плюс изменен тип возврата)
{
  // тело метода
  return 0;
}

Пример того, как не может быть перегружен метод:

public void SomeMethod(int a)
{
  // тело метода
}

public void SomeMethod(int b) // имени параметра недостаточно
{
  // тело метода
}

public int SomeMethod(int a) // типа возвращаемого значения недостаточно
{
  // тело метода
  return 0;
}
```

Задание на лабораторную работу

Задание 1

Вариант	№	Задание
1	1	Написать программу, в которой будет реализована процедура вычисления количества строк двумерного массива, в которых среднее арифметическое элементов меньше нуля
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения суммы элементов одномерного массива A длины N.
2	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых все элементы меньше нуля
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения максимального элемента одномерного массива

3	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых хотя бы один элемент не равен нулю
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения минимального элемента одномерного массива
4	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых есть элементы разных знаков
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения номера максимального элемента одномерного массива
5	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых все элементы упорядочены по возрастанию
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения номера минимального элемента одномерного массива
6	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых все элементы упорядочены по убыванию
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения количества отрицательных элементов одномерного массива
7	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых сумма модуля отрицательных больше суммы положительных элементов
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения номера последнего нуля одномерного массива
8	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых содержится максимальный по модулю элемент массива
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения кол-ва нулей одномерного массива
9	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых содержится минимальный по модулю элемент массива
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения количества положительных элементов одномерного массива
10	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых присутствуют отрицательные элементы

	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения номера последнего отрицательного элемента одномерного массива
11	1	Написать программу, в которой будет реализована процедура вычисления кол-ва строк двумерного массива, в которых сумма элементов меньше суммы элементов, расположенных выше главной диагонали
	2	Составить программу, содержащую процедуру вычисления значений выражений по заданным формулам (из 1 л.р.)
	3	Составить программу, содержащую функцию для нахождения произведения положительных элементов одномерного массива

Задание 2

Даны действительные числа a, b. Получить u = min(a, b-a), y = min(ab, a+b),

k = min(u+v*2, 3.14).

Вычисление u, v и k оформите в виде функций с обязательным применением параметров ref, out, in.

Задание 3

Методы, реализованные в задании 2, оформить в виде перегружаемых методов.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Выполнить полученное задание в консольном приложении (.NET Framework).

5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 8

«Разработка интерфейса приложения для работы с файловой системой на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ построения интерфейса для работы с файловой системой на платформе .NET Framework.

Краткие теоретические сведения

Файл – это набор данных, который хранится на внешнем запоминающем устройстве (например, на жестком диске). Файл имеет имя и расширение. Расширение позволяет идентифицировать, какие данные и в каком формате хранятся в файле.

Под работой с файлами подразумевается:

- создание файлов;
- удаление файлов;
- чтение данных;
- запись данных;
- изменение параметров файла (имя, расширение…);
- другое.

В C# есть пространство имен System.IO, в котором реализованы все необходимые классы для работы с файлами. Чтобы подключить это пространство имен, необходимо в самом начале программы добавить строку using System.IO. Для использования кодировок еще добавим пространство using System.Text;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

Для создания пустого файла, в классе File есть метод Create(). Он принимает один аргумент – путь. Ниже приведен пример создания пустого текстового файла new_file.txt на диске D:

```
static void Main(string[] args)
{
  File.Create("D:\\new_file.txt");
}
```

Если файл с таким именем уже существует, он будет переписан на новый пустой файл.

Метод WriteAllText() создает новый файл (если такого нет), либо открывает существующий и записывает текст, заменяя всё, что было в файле:

```
static void Main(string[] args)
{
  File. WriteAllText("D:\\new_file.txt", "текст");
}
```

Метод AppendAllText() работает, как и метод WriteAllText() за исключением того, что новый текст дописывается в конец файла, а не переписывает всё что было в файле:

```
static void Main(string[] args)
{
  File.AppendAllText("D:\\new_file.txt", "тест метода AppendAllText ()"); //допишет текст в конец файла
}
```

Метод Delete() удаляет файл по указаному пути:

```
static void Main(string[] args)
{
  File.Delete("d:\\test.txt"); //удаление файла
}
```

Кроме того, чтобы читать/записывать данные в файл с C# можно использовать потоки.

Поток – это абстрактное представление данных (в байтах), которое облегчает работу с ними. В качестве источника данных может быть файл, устройство ввода-вывода, принтер.

Класс Stream является абстрактным базовым классом для всех потоковых классов в C#. Для работы с файлами нам понадобится класс FileStream (файловый поток).

FileStream - представляет поток, который позволяет выполнять операции чтения/записи в файл.

```
static void Main(string[] args)
{
  FileStream file = new FileStream("d:\\test.txt", FileMode.Open, FileAccess.Read); //открывает файл только на чтение
}
```

Режимы открытия FileMode:

- Append – открывает файл (если существует) и переводит указатель в конец файла (данные будут дописываться в конец), или создает новый файл. Данный режим возможен только при режиме доступа FileAccess.Write.

- Create - создает новый файл(если существует – заменяет)

- CreateNew – создает новый файл (если существует – генерируется исключение)

- Open - открывает файл (если не существует – генерируется исключение)

- OpenOrCreate – открывает файл, либо создает новый, если его не существует

- Truncate – открывает файл, но все данные внутри файла затирает (если файла не существует – генерируется исключение)

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\file1.txt", FileMode.CreateNew); //создание нового файла
  FileStream file2 = new FileStream("d:\\file2.txt", FileMode.Open); //открытие существующего файла
  FileStream file3 = new FileStream("d:\\file3.txt", FileMode.Append); //открытие файла на дозапись в конец файла
}
```

Режими доступа FileAccess:

- Read – открытие файла только на чтение. При попытке записи генерируется исключение

- Write - открытие файла только на запись. При попытке чтения генерируется исключение

- ReadWrite - открытие файла на чтение и запись.

Чтение из файла

Для чтения данных из потока нам понадобиться класс StreamReader. В нем ре-ализовано множество методов для удобного считывания данных. Ниже приведена программа, которая выводит содержимое файла на экран:

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open); //создаем фай-ловый поток
  StreamReader reader = new StreamReader(file1); // создаем «потоковый чита-тель» и связываем его с файловым потоком
  Console.WriteLine(reader.ReadToEnd()); //считываем все данные с по-тока и вы-водим на экран
  reader.Close(); //закрываем поток
  Console.ReadLine();
}
```

Метод ReadToEnd() считывает все данные из файла. ReadLine() – считывает од-ну строку (указатель потока при этом переходит на новую строку, и при следующем вызове метода будет считана следующая строка).

Свойство EndOfStream указывает, находится ли текущая позиция в потоке в конце потока (достигнут ли конец файла). Возвращает true или false.

Запись в файл

Для записи данных в поток используется класс StreamWriter. Пример записи в файл:

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\test.txt", FileMode.Create); //создаем фай-ловый поток
```

```
        StreamWriter writer = new StreamWriter(file1); //создаем «потоковый писатель» и связываем его с файловым потоком
        writer.Write("текст"); //записываем в файл
        writer.Close(); //закрываем поток. Не закрыв поток, в файл ничего не запишется
    }
```

Метод WriteLine() записывает в файл построчно (то же самое, что и простая запись с помощью Write(), только в конце добавляется новая строка).

Нужно всегда помнить, что после работы с потоком, его нужно закрыть (освободить ресурсы), использовав метод Close().

Кодировка, в которой будут считываться/записываться данные указывается при создании StreamReader/StreamWriter:

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open);
  StreamReader reader = new StreamReader(file1, Encoding.Unicode);
  StreamWriter writer = new StreamWriter(file1, Encoding.UTF8);
}
```

Кроме того, при использовании StreamReader и StreamWriter можно не создавать отдельно файловый поток FileStream, а сделать это сразу при создании StreamReader/StreamWriter:

```
static void Main(string[] args)
{
  StreamWriter writer = new StreamWriter("d:\\test.txt"); //указываем путь к фай-лу, а не поток
  writer.WriteLine("текст");
  writer.Close();
}
```

С помощью статического метода CreateDirectory() класса Directory:

```
static void Main(string[] args)
{
 Directory.CreateDirectory("d:\\new_folder");
}
```

Для удаления папок испоьзуется метод Delete():

```
static void Main(string[] args)
{
  Directory.Delete("d:\\new_folder"); //удаление пустой папки
}
```

Если папка не пустая, необходимо указать параметр рекурсивного удаления – true.

Задание на лабораторную работу

Задание 1 (по вариантам)

1) Написать программу, которая создает файл, содержащий действительные числа, и находит сумму наибольшего и наименьшего из чисел, содержащихся в файле.

2) Написать программу, которая создает файл, содержащий целые числа, и находит произведение всех чисел, содержащихся в файле.

3) Написать программу, которая создает файл, содержащий действительные числа, и находит сумму квадратов чисел, содержащихся в файле.

4) Написать программу, которая создает файл, содержащий целые числа, и находит модуль суммы и квадрат произведения чисел, содержащихся в файле.

5) Написать программу, которая создает файл, содержащий действительные числа, и находит последнее из чисел, содержащихся в файле.

6) Написать программу, которая создает файл, содержащий целые числа, и находит наименьшее из чисел, содержащихся в файле.

7) Написать программу, которая создает файл, содержащий действительные числа, и находит разность первого и последнего из чисел, содержащихся в файле.

8) Написать программу, которая создает файл, содержащий целые числа, и находит количество четных чисел, содержащихся в файле.

9) Написать программу, которая создает файл, содержащий действительные числа, и находит сумму нечетных чисел, содержащихся в файле.

10) Написать программу, которая создает файл, содержащий целые числа, и находит произведение четных чисел, содержащихся в файле.

11) Написать программу, которая в его конце запишет следующие данные: количество строк, количество символов в каждой строке, количество элементов в каждой строке.

Задание 2

Разработать интерфейс приложения и реализовать функции создания папки.

Задание 3

Разработать интерфейс приложения и реализовать функции создания текстового файла различными методами.

Задание 4

Разработать интерфейс приложения и реализовать функции чтения из файла.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Выполнить полученное задание в консольном приложении (.NET Framework).

5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 9

«Разработка интерфейса приложения для работы с двоичными (бинарными) файлами на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ построения интерфейса для работы с двоичными (бинарными) файлами на платформе .NET Framework.

Краткие теоретические сведения

Общие теоретические сведения

Файл – последовательность элементов одного типа, которой присвоено общее имя.

Количество элементов файла заранее не определяется. Максимальный размер файла, размещенного во внешней памяти, ограничивается лишь техническими возможностями вычислительной системы.

Доступ к элементу файла осуществляется через указатель файла. При выполнении операции чтения или записи указатель автоматически перемещается на следующий элемент.

Любой элемент становиться доступен, лишь после того, как будут последовательно пройдены все предыдущие значения. После каждого элемента автоматически ставиться признак конца элемента, а в конце файла ставиться признак конца файла.

В зависимости от способа представления информации различают три типа файлов:

– типизированные файлы;

– нетипизированные файлы;

– текстовые файлы.

Нетипизированные файлы – это файлы с произвольным доступом, которые используют для организации скоростного обмена между внешней и оперативной памятью.

Типизированный файл – файл с объявленным типом элементов, т.е. файл с одной и той же структурой элементов.

Текстовые файлы – это файлы, содержащие строки переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. В сущности, тестовые файлы представляют собой типизированные файлы строкового типа, но для удобства работы с ними в C# разработаны специальные классы, с которыми вы познакомились в предыдущей лабораторной работе.

Классы BinaryWriter и BinaryReader для работы с двоичными файлами

Класс BinaryWriter

Для работы с бинарными файлами предназначена пара классов BinaryWriter и BinaryReader. Эти классы позволяют читать и записывать данные в двоичном формате.

Класс BinaryWriter служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных.

BinaryWriter(Stream output)

где output обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра output может быть указан объект, создаваемый средствами класса FileStream. Если же параметр output оказывается пустым, то генерируется исключение ArgumentNullException. А если поток, определяемый параметром output, не был открыт для записи данных, то генерируется

исключение ArgumentException. По завершении вывода в поток типа BinaryWriter его нужно закрыть. При этом закрывается и базовый поток.

В классе BinaryWriter определены методы, предназначенные для записи данных всех встроенных в C# типов. Ниже приведен наиболее часто употребляемый конструктор этого класса:

Основные метода класса BinaryWriter
Close(): закрывает поток и освобождает ресурсы
Flush(): очищает буфер, дописывая из него оставшиеся данные в файл
Seek(): устанавливает позицию в потоке
Write(): записывает данные в поток

Класс BinaryReader

Класс BinaryReader служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных. Ниже приведен наиболее часто употребляемый конструктор этого класса:

BinaryReader(Stream input)

где input обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра input может быть указан объект, создаваемый средствами класса FileStream. Если же поток, определяемый параметром input, не был открыт для чтения данных или оказался недоступным по иным причинам, то генерируется исключение ArgumentException. По завершении ввода из потока типа BinaryReader его нужно закрыть. При этом закрывается и базовый поток.

В классе BinaryReader определены методы, предназначенные для чтения данных всех встроенных в C# типов.

Основные метода класса BinaryReader
Close(): закрывает поток и освобождает ресурсы
ReadBoolean(): считывает значение bool и перемещает указатель на один байт
ReadByte(): считывает один байт и перемещает указатель на один байт

ReadChar(): считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке

ReadDecimal(): считывает значение decimal и перемещает указатель на 16 байт

ReadDouble(): считывает значение double и перемещает указатель на 8 байт

ReadInt16(): считывает значение short и перемещает указатель на 2 байта

ReadInt32(): считывает значение int и перемещает указатель на 4 байта

ReadInt64(): считывает значение long и перемещает указатель на 8 байт

ReadSingle(): считывает значение float и перемещает указатель на 4 байта

ReadString(): считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа int занимает 4 байта, поэтому BinaryReader считает 4 байта и переместит указать на эти 4 байта.

Применение методов для работы с двоичными файлами

Рассмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

```
struct State
{
    public string name;
    public string capital;
    public int area;
    public double people;

    public State(string n, string c, int a, double p)
    {
        name = n;
        capital = c;
        people = p;
        area = a;
    }
}

class Program
{
    static void Main(string[] args)
```

```
        {
            State[] states = new State[2];
            states[0] = new State("Германия", "Берлин",  357168,  80.8);
            states[1] = new State("Франция", "Париж", 640679, 64.7);

            string path= @"C:\SomeDir\states.dat";

            try
            {
                // создаем объект BinaryWriter
                using (BinaryWriter writer = new BinaryWriter(File.Open(path, File-
Mode.OpenOrCreate)))
                {
                    // записываем в файл значение каждого поля структуры
                    foreach (State s in states)
                    {
                        writer.Write(s.name);
                        writer.Write(s.capital);
                        writer.Write(s.area);
                        writer.Write(s.people);
                    }
                }
                // создаем объект BinaryReader
                using (BinaryReader reader = new BinaryReader(File.Open(path,
FileMode.Open)))
                {
                    // пока не достигнут конец файла
                    // считываем каждое значение из файла
                    while (reader.PeekChar() > -1)
                    {
                        string name = reader.ReadString();
                        string capital = reader.ReadString();
                        int area = reader.ReadInt32();
                        double population = reader.ReadDouble();

                        Console.WriteLine("Страна: {0}  столица: {1}  площадь {2}
кв. км  численность населения: {3} млн. чел.",
                            name, capital, area, population);
                    }
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            Console.ReadLine();
```

```
    }
}
```

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком эти значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса BinaryReader также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))

В цикле while считываем данные. Чтобы узнать окончание потока, вызываем метод PeekChar(). Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы BinaryWriter и BinaryReader очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более комплексных объектов, например, объектов классов, лучше подходит другое решение - сериализация.

Файлы с произвольным доступом

В предыдущих примерах использовались последовательные файлы, т.е. файлы со строго линейным доступом, байт за байтом. Но доступ к содержимому файла может быть и произвольным. Для этого служит, в частности, метод Seek(), определенный в классе FileStream. Этот метод позволяет установить указатель положения

в файле, или так называемый указатель файла, на любое место в файле. Ниже приведена общая форма метода Seek():

long Seek(long offset, SeekOrigin origin)

где offset обозначает новое положение указателя файла в байтах относительно заданного начала отсчета (origin). В качестве origin может быть указано одно из приведенных ниже значений, определяемых в перечислении SeekOrigin:

SeekOrigin.Begin – поиск от начала файла
SeekOrigin.Current – поиск от текущего положения
SeekOrigin.End – поиск от конца файла

Следующая операция чтения или записи после вызова метода Seek() будет выполняться, начиная с нового положения в файле, возвращаемого этим методом. Если во время поиска в файле возникает ошибка, то генерируется исключение IOException. Если же запрос положения в файле не поддерживается базовым потоком, то генерируется исключение NotSupportedException. Кроме того, могут быть сгенерированы и другие исключения.

В приведенном ниже примере программы демонстрируется ввод-вывод в файл с произвольным доступом. Сначала в файл записываются прописные буквы английского алфавита, а затем его содержимое считывается обратно в произвольном порядке.

```
// Продемонстрировать произвольный доступ к файлу

using System;
using System.IO;

class RandomAccessDemo {
    static void Main() {
        FileStream f = null;
        char ch;
        f = new FileStream("random.dat", FileMode.Create);
        // Записать английский алфавит в файл.
        for (int i=0; i < 26; i++)
              f.WriteByte((byte)('A'+i));
```

```
            // А теперь считать отдельные буквы английского алфавита.
            f.Seek(0, SeekOrigin.Begin); // найти первый байт
            ch = (char) f.ReadByte();
            Console.WriteLine("Первая буква: " + ch);

            f.Seek(1, SeekOrigin.Begin); // найти второй байт
            ch = (char) f.ReadByte();
            Console.WriteLine("Вторая буква: " + ch);

            f.Seek(4, SeekOrigin.Begin); // найти пятый байт
            ch = (char) f.ReadByte();
            Console.WriteLine("Пятая буква: " + ch);

            Console.WriteLine ();

            // А теперь прочитать буквы английского алфавита через одну.
            Console.WriteLine("Буквы алфавита через одну: ");
            for(int i=0; i < 26; i += 2) {
                  f.Seek(i, SeekOrigin.Begin); // найти i-й символ
                  ch = (char) f.ReadByte();
                  Console.Write(ch + " ");
            }

            if(f != null) f.Close();
            Console.WriteLine();
      }
}
```

При выполнении этой программы получается следующий результат.

```
Первая буква: A
Вторая буква: B
Пятая буква: E
Буквы алфавита через одну:
A C E G I K M O Q S U W Y
```

Несмотря на то что метод Seek() имеет немало преимуществ при использовании с файлами, существует и другой способ установки текущего положения в файле с помощью свойства Position. Свойство Position доступно как для чтения, так и для записи. Поэтому с его помощью можно получить или же установить текущее

положение в файле. В качестве примера ниже приведен фрагмент кода из предыдущей программы записи и чтения из файла с произвольным доступом random.dat, измененный с целью продемонстрировать применение свойства Position.

```
Console.WriteLine("Буквы алфавита через одну: ");
for(int i=0; i < 26; i += 2) {
        f.Position = i; // найти i-й символ посредством свойства Position
        ch = (char) f.ReadByte();
        Console.Write(ch + " ");
}
```

Класс File

В среде .NET Framework определен класс File, который может оказаться полезным для работы с файлами, поскольку он содержит несколько статических методов, выполняющих типичные операции над файлами. В частности, в классе File имеются методы для копирования и перемещения, шифрования и расшифровывания, удаления файлов, а также для получения и задания информации о файлах, включая сведения об их существовании, времени создания, последнего доступа и различные атрибуты файлов (только для чтения, скрытых и пр.).

Кроме того, в классе File имеется ряд удобных методов для чтения из файлов и записи в них, открытия файла и получения ссылки типа FileStream на него. В классе File содержится слишком много методов для подробного их рассмотрения, поэтому мы уделим внимание только трем из них. Сначала будет представлен метод Copy(), а затем — методы Exists() и GetLastAccessTime().

На примере этих методов вы сможете получить ясное представление о том, насколько удобны методы, доступные в классе File. И тогда вам станет ясно, что класс File определенно заслуживает более тщательного изучения.

Ряд методов для работы с файлами определен также в классе FileInfo. Этот класс отличается от класса File одним, очень важным преимуществом: для операций над файлами он предоставляет методы экземпляра и свойства, а не статические методы. Поэтому для выполнения нескольких операций над одним и тем же файлом лучше воспользоваться классом FileInfo.

Копирование файлов с помощью метода Copy()

Задача копирования файлов не представляет особых трудностей, ее можно полностью автоматизировать с помощью метода Copy(), определенного в классе File. Ниже представлены две формы его объявления.

static void Copy (string имя_исходного_файла, string имя_целевого_файла)

static void Copy (string имя_исходного_файла, string имя_целевого_файла, boolean overwrite)

Метод Copy() копирует файл, на который указывает имя_исходного_файла, в файл, на который указывает имя_целевого_файла. В первой форме данный метод копирует файл только в том случае, если файл, на который указывает имя_целевого_файла, еще не существует. А во второй форме копия заменяет и перезаписывает целевой файл, если он существует и если параметр overwrite принимает логическое значение true. Но в обоих случаях может быть сгенерировано несколько видов исключений, включая IOException и FileNotFoundException.

В приведенном ниже примере программы метод Copy() применяется для копирования файла. Имена исходного и целевого файлов указываются в командной строке.

```
/* Скопировать файл, используя метод File.Copy().
Чтобы воспользоваться этой программой, укажите имя исходного и целевого
файлов. Например, чтобы скопировать файл FIRST.DAT в файл SECOND.DAT,
введите в командной строке следующее:
CopyFile FIRST.DAT SECOND.DAT
*/
using System;
using System.IO;
class CopyFile {
   static void Main(string[] args) {
        if(args.Length != 2) {
              Console.WriteLine("Применение : CopyFile Откуда Куда");
              return;
        }
        // Копировать файлы.
        try {
              File.Copy(args[0], args[1]);
        } catch(IOException exc) {
```

```
                Console.WriteLine("Ошибка копирования файла\n" + exc.Mes-
            sage);
            }
        }
    }
```

Как видите, в этой программе не нужно создавать поток типа FileStream или освобождать его ресурсы. Все это делается в методе Copy() автоматически. Обратите также внимание на то, что в данной программе существующий файл не перезаписывается.

Поэтому если целевой файл должен быть перезаписан, то для этой цели лучше воспользоваться второй из упоминавшихся ранее форм метода Copy().

Применение методов Exists() и GetLastAccessTime()

С помощью методов класса File очень легко получить нужные сведения о файле.

Рассмотрим два таких метода: Exists() и GetLastAccessTime(). Метод Exists() определяет, существует ли файл, а метод GetLastAccessTime() возвращает дату и время последнего доступа к файлу. Ниже приведены формы объявления обоих методов.

static bool Exists(string путь)
static DateTime GetLastAccessTime(string путь)

В обоих методах путь обозначает файл, сведения о котором требуется получить.

Метод Exists() возвращает логическое значение true, если файл существует и доступен для вызывающего процесса. А метод GetLastAccessTime() возвращает структуру DateTime, содержащую дату и время последнего доступа к файлу. (Структура DateTime описывается далее в этой книге, но метод ToString() автоматически приводит дату и время к удобочитаемому виду.) С указанием недействительных аргументов или прав доступа при вызове обоих рассматриваемых здесь методов может быть связан целый ряд исключений, но в действительности генерируется только исключение IOException.

В приведенном ниже примере программы методы Exists() и GetLastAccessTime() демонстрируются в действии. В этой программе сначала определяется, существует ли файл под названием test.txt. Если он существует, то на экран выводит время последнего доступа к нему.

```
// Применить методы Exists() и GetLastAccessTime().
using System;
using System.IO;
class ExistsDemo {
    static void Main() {
        if(File.Exists("test.txt"))
            Console.WriteLine("Файл существует. В последний раз он был
        доступен " + File.GetLastAccessTime("test.txt"));
        else
            Console.WriteLine("Файл не существует");
    }
}
```

Кроме того, время создания файла можно выяснить, вызвав метод GetCreationTime(), а время последней записи в файл, вызвав метод GetLastWriteTime(). Имеются также варианты этих методов для представления данных о файле в формате всеобщего скоординированного времени (UTC). Попробуйте поэкспериментировать с ними.

Задание на лабораторную работу

Задание 1 (по вариантам)

1) Написать программу, которая создает **двоичный** файл, содержащий действительные числа, и находит сумму наибольшего и наименьшего из чисел, содержащихся в файле.

2) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит произведение всех чисел, содержащихся в файле.

3) Написать программу, которая создает **двоичный** файл, содержащий действительные числа, и находит сумму квадратов чисел, содержащихся в файле.

4) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит модуль суммы и квадрат произведения чисел, содержащихся в файле.

5) Написать программу, которая создает **двоичный** файл, содержащий действительные числа, и находит последнее из чисел, содержащихся в файле.

6) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит наименьшее из чисел, содержащихся в файле.

7) Написать программу, которая создает **двоичный** файл, содержащий действительные числа, и находит разность первого и последнего из чисел, содержащихся в файле.

8) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит количество четных чисел, содержащихся в файле.

9) Написать программу, которая создает **двоичный** файл, содержащий действительные числа, и находит сумму нечетных чисел, содержащихся в файле.

10) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит произведение четных чисел, содержащихся в файле.

11) Написать программу, которая создает **двоичный** файл, содержащий целые числа, и находит модуль разности и квадрат деления чисел, содержащихся в файле.

Задание 2

В имеющуюся программу добавить функционал, который в случайно заданную позицию получившегося в первом задании файла, вставляет заданное пользователем число.

Задание 3

Разработать интерфейс приложения и реализовать функции копирования заданного пользователем файла.

Задание 4

Разработать интерфейс приложения и реализовать функции получения информации о заданном файле, а именно: время создания файла, время последнего доступна к файлу, время последней записи файла. Информацию вывести на экран и сохранить в файле.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.

2. Запустить приложение MS Visual Studio.

3. Создать новый проект.

4. Выполнить полученное задание в консольном приложении (.NET Framework).

5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).

6. Сохранить результаты лабораторной работы.

7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 10

«Разработка интерфейса приложения, реализующего принципы инкапсуляции на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ принципов инкапсуляции в приложениях на платформе .NET Framework.

Краткие теоретические сведения

Классы в C#

Класс — это логическая структура, позволяющая создавать свои собственные пользовательские типы путем группирования переменных других типов, методов и событий. Класс подобен чертежу. Он определяет данные и поведение типа. Если класс не объявлен статическим, то клиентский код может его использовать, создав объекты или экземпляры, назначенные переменной. Переменная остается в памяти, пока все ссылки на нее не выйдут из области видимости. В это время среда CLR помечает ее пригодной для сборщика мусора. Если класс объявляется статическим, то в памяти остается только одна копия и клиентский код может получить к ней доступ только посредством самого класса, а не переменной экземпляра. Для получения дополнительной информации см. Статические классы и члены статических классов (Руководство по программированию в C#).

В отличие от структур классы поддерживают наследование, фундаментальную характеристику объектно-ориентированного программирования. Для получения дополнительной информации см. Наследование (Руководство по программированию на C#).

Данные-члены

это те члены, которые содержат данные класса — поля, константы, события. Данные-члены могут быть статическими (static). Член класса является членом экземпляра, если только он не объявлен явно как static. Давайте рассмотрим виды этих данных:

Поля (field)

Это любые переменные, ассоциированные с классом.

Константы

Константы могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова const. Если она объявлена как public, то в этом случае становится доступной извне класса.

События

Это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них.

Функции-члены

Функции-члены — это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы:

Методы (method)

Это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора static.

Свойства (property)

Это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.

Индексаторы (indexer)

Позволяют индексировать объекты таким же способом, как массив или коллекцию.

Конструкторы (constructor)

Это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.

Финализаторы (finalizer)

Вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды. Предсказать точно, когда будет вызван финализатор, невозможно.

Операции (operator)

Это простейшие действия вроде + или -. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако C# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая перегрузка операции).

Принципы ООП: Инкапсуляция

К основным принципам ООП относятся следующие: инкапсуляция, наследование и полиморфизм.

Инкапсуляция предполагает возможность скрытия данных объекта от пользователя. Это свойство языка программирования (объектно-ориентированного), позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (то, что у него внутри), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов),

- Пользователь может взаимодействовать с объектом только через интерфейс.
- Пользователь не может использовать закрытые данные и методы.

Класс представляет собой единство трех сущностей – полей, методов и свойств. Объединение этих сущностей в единое целое называется инкапсуляцией. Инкапсуляция позволяет во многом изолировать класс от остальных частей программы, сделать его автономным для решения конкретной задачи. В результате класс всегда несет в себе некоторую функциональность. Например, класс TForm содержит (инкапсулирует в себе) все необходимое для создания Windows-окна, класс TMemo представляет собой полнофункциональный текстовый редактор, класс TTimer обеспечивает работу программы с таймером и т. д.

Структура класса

Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы:

```
class имя_класса {
 // Объявление переменных экземпляра.
 доступ тип переменная1;
 доступ тип переменная2;
 //...
 доступ тип переменнаяN;

 // Объявление методов.
 доступ возращаемый_тип метод1 (параметры) {
       // тело метода
 }
 доступ возращаемый_тип метод2 (параметры) {
       // тело метода
 }
 //. . .
 доступ возращаемый_тип методN(параметры) {
       // тело метода
 }
}
```

Обратите внимание на то, что перед каждым объявлением переменной и метода указывается доступ. Это спецификатор доступа, например public, определяющий порядок доступа к данному члену класса. Члены класса могут быть как закрытыми (private) в пределах класса, так открытыми (public), т.е. более доступными. Спецификатор доступа определяет тип разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса.

Для объявления объекта произвольного типа используется следующая конструкция:

<тип класса> имя переменной = new <тип класса>();

```
InfoUser myinfo = new InfoUser();
```

Инициализаторы объектов

Инициализаторы объектов предоставляют способ создания объекта и инициализации его полей и свойств. Если используются инициализаторы объектов, то вместо обычного вызова конструктора класса указываются имена полей или свойств, инициализируемых первоначально задаваемым значением. Следовательно, синтаксис инициализатора объекта предоставляет альтернативу явному вызову конструктора класса. Синтаксис инициализатора объекта используется главным образом при создании анонимных типов в LINQ-выражениях. Но поскольку инициализаторы объектов можно, а иногда и нужно использовать в именованном классе, то ниже представлены основные положения об инициализации объектов.

Ниже приведена общая форма синтаксиса инициализации объектов:

new имя_класса {имя = выражение, имя = выражение, ...}

где имя обозначает имя поля или свойства, т.е. доступного члена класса, на который указывает имя_класса. А выражение обозначает инициализирующее выражение, тип которого, конечно, должен соответствовать типу поля или свойства.

Инициализаторы объектов обычно не используются в именованных классах, хотя это вполне допустимо. Вообще, при обращении с именованными классами используется синтаксис вызова обычного конструктора.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class autoCar
   {
      public string marka;
      public short year;
   }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // используем инициализаторы
            autoCar myCar = new autoCar { marka = "Renault", year = 2004 };

            Console.ReadLine();
        }
    }
}
```

Конструкторы

Конструкторы являются одной из разновидностей методов класса. Конструкторы — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа.

Инициализация объекта происходит автоматически. Программисту не приходится беспокоиться, допустим, о поиске свободной памяти, в которой будет размещен новый объект. Однако иногда возникают ситуации, когда на этапе инициализации может потребоваться выполнение каких-либо дополнительных действий. Например, очень часто бывает необходимо инициализировать данные, хранящиеся в объекте. Как раз это и входит в функции конструктора.

У каждого объекта имеется конструктор, используемый по умолчанию, который представляет собой метод без параметров. Его имя совпадает с именем самого класса.

Определение класса может включать в себя несколько конструкторов. Они могут иметь отличающиеся сигнатуры, которые используются для создания экземпляра объекта.

Для присваивания начальных значений данным, хранящимся внутри объекта, часто применяются конструкторы с параметрами. В C# конструктор можно вызвать, введя ключевое слово new. Экземепляры также могут создаваться с помощью конструктора, используемого не по умолчанию. Как и имя конструктора по умолчанию, имена этих конструкторов совпадают с именем класса, но у них имеются еще и параметры. Используются они аналогичным образом.

Ниже приведен пример конструктора, который инициализирует все поля.

```
class sotrudnik
        {
          public String Surname;// фамилия сотрудника
          public String Name; //Имя сотрудника
          public int Age; //Возраст
          public int Money; //зарплата

          //конструктор
           public sotrudnik(string surname, string name, int age, int money)
          {
                        this.Surname=surname;
                  this.Name = name;
                        this.Age = age;
                        this.Money = money;
          }
        }

       //И сразу попробуем поработать с ним
        static void Main(string[] args)
        {
           sotrudnik s=new sotrudnik("Иванов", "Иван", 32, 30000);
           System.Console.WriteLine(s.Surname);
           System.Console.WriteLine(s.Name);
           System.Console.WriteLine(s.Age);
           System.Console.WriteLine(s.Money);
           System.Console.ReadLine();
        }
```

Обратите внимание на то что конструктор также, как и переменные создается, используя модификатор доступа public. Так же обратите внимание на то как задаются значения полей для класса. Но естественно значения полей после конструктора можно переопределить тем же способом что и в предыдущем примере.

Свойства

Свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения частного (private) поля. Свойства можно использовать, как если бы они являлись открытыми членами данных, хотя в действительности они являются специальными методами, называемыми методами доступа. Это обеспечивает простой доступ к данным и позволяет повысить уровень безопасности и гибкости методов.

```
тип имя {
  get
  {
     // код аксессора для чтения из поля
  }

  set
  {
     // код аксессора для записи в поле
  }
}
```

В данном примере, класс TimePeriod хранит сведения о периоде времени. Внутри класса время хранится в секундах, но свойство с именем Hours позволяет клиенту задать время в часах. Методы доступа для свойства Hours выполняют преобразование между часами и секундами.

```
class TimePeriod
{
   private double seconds;

   public double Hours
   {
      get { return seconds / 3600; }
      set { seconds = value * 3600; }
   }
}

class Program
{
   static void Main()
   {
      TimePeriod t = new TimePeriod();

      // Assigning the Hours property causes the 'set' accessor to be called.
      t.Hours = 24;

      // Evaluating the Hours property causes the 'get' accessor to be called.
      System.Console.WriteLine("Time in hours: " + t.Hours);
   }
}
// Output: Time in hours: 24
```

Свойства позволяют классу предоставлять общий способ получения и задания значений, скрывая при этом код реализации или проверки.

Метод доступа свойства get используется для возврата значения свойства, а метод доступа set используется для назначения нового значения. Эти методы доступа могут иметь различные уровни доступа. Дополнительные сведения см. в разделе Ограничение доступности методов доступа (Руководство по программированию на C#).

Ключевое слово value используется для определения значения, присваиваемого методом доступа set.

Свойства, которые не реализуют метод доступа set, доступны только для чтения.

Для простых свойств, не требующих пользовательского кода метода доступа, рассмотрите возможность использования автоматически реализуемых свойств.

Задание на лабораторную работу

Задание 1

Создать класс, описывающий геометрическую фигуру. Описать характеристики геометрической фигуры, добавив соответствующие поля.

Варианты:

Вариант	Геометрическая фигура
1	Конус
2	Цилиндр
3	Сфера
4	Куб
5	Треугольная призма
6	Треугольная пирамида
7	Прямоугольный параллелепипед
8	Четырехугольная призма
9	Четырехугольная пирамида
10	Шестиугольная призма
11	Шестиугольная пирамида

Задание 2

Создать несколько объектов класса с разными параметрами и инициализировать их с помощью конструктора.

Задание 3

Добавить функционал, позволяющий редактировать поля класса посредством свойств (property). При редактировании добавить возможность проверки допустимости вводимых значений.

Задание 4

Добавить методы для расчета площади и периметра основания геометрической фигуры, а также объема геометрической фигуры.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.
2. Запустить приложение MS Visual Studio.
3. Создать новый проект.
4. Выполнить полученное задание в консольном приложении (.NET Framework).
5. Выполнить полученное задание в приложении Windows Forms (.NET Framework).
6. Сохранить результаты лабораторной работы.
7. Подготовить отчет по лабораторной работе.

Лабораторная работа № 11

«Разработка многооконного пользовательского интерфейса на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ программирования многооконного пользовательского интерфейса на основе наследования классов на платформе .NET Framework.

Краткие теоретические сведения

Наследование

Наследование (inheritance) является одним из ключевых моментов ООП. Его смысл состоит в том, что мы можем расширить функциональность уже существующих классов за счет добавления нового функционала или изменения старого. Пусть у нас есть следующий класс Person, описывающий отдельного человека:

```
class Person {
    private string _firstName;
    private string _lastName;

    public string FirstName     {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public string LastName     {
        get { return _lastName; }
        set { _lastName = value; }
    }
     public void Display()    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}
```

Затем потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником) от класса Person, который, в свою очередь, называется базовым классом или родителем:

```
class Employee : Person
{
}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person.

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

– Не поддерживается множественное наследование, класс может наследоваться только от одного класса. Хотя проблема множественного наследования реализуется с помощью концепции интерфейсов, о которых мы поговорим позже.

– При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа internal, то производный класс может иметь тип доступа internal или private, но не public.

– Если класс объявлен с модификатором sealed, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

Хоть Employee наследует весь функционал от класса Person, но в следующем случае код не сработает и выдаст ошибку, так как переменная _firstName объявлена с модификатором private и поэтому к ней доступ имеет только класс Person.

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_firstName);
    }
}
```

Но зато в классе Person определено общедоступное свойство FirstName, которое мы можем использовать, поэтому следующий код будет работать правильно:

```
class Employee : Person
{
   public void Display()
   {
      Console.WriteLine(FirstName);
   }
}
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами public, internal, protected и protected internal.

Области видимости членов класса

Ключевые слова C#, указывающие уровень доступности - классификаторы доступности:

public – помечает метод, как доступный из объектной переменной, а также из всех производных классов

private – помечает метод, как доступный только из класса, определяющего этот метод. В C# любой член по умолчанию определяется, как private

protected – помечает метод, как доступный для определяющего класса, а так же для любого производного класса. Однако защищенные методы не доступны из объектной переменной

internal – определяет метод, как доступный для любого типа только внутри данного компоновочного блока, но не снаружи.

protected internal – определяет метод, доступ к которому ограничивается рамками текущего компоновочного блока или типами, созданными из определяющего класса в данном компоновочном блоке.

Классификатор доступности необязателен и если он не указан, то по умолчанию подразумевается, что член закрыт (*private*) в рамках класса, где он определен.

Наследуемый класс не может иметь более открытый классификатор доступности, чем его предок.

В C# структуры не поддерживают наследования. Поэтому спецификатор доступа protected в объявлении данных — членов и функций — членов структур не применяется.

Переопределение членов базового класса

При объявлении членов производного класса в C# разрешено использовать те же самые имена, которые применялись при объявлении членов базового класса. Это касается как методов, так и данных — членов объявляемых классов. В этом случае соответствующие члены базового класса считаются переопределенными.

Доступность типов

Типы (классы, интерфейсы, структуры, перечни и делегаты) могут использовать классификаторы доступности, но только public или internal. Public гарантирует, что тип будет доступен для других типов как в текущем компоновочном блоке, так и во внешних компоновочных блоках. Internal (внутренний) тип может использоваться только компоновочным блоком, в котором этот тип определен. По умолчанию для типов используется доступность internal.

Вызов конструкторов базового класса

В дополнение к наследуемым методам производный класс автоматически содержит все поля из базового класса. Обычно при создании объекта эти поля требуют инициализации. Как правило, инициализация такого рода выполняется в конструкторе. Вспомним, что все классы имеют как минимум один конструктор (при этом если вы не предоставляете конструктор сами, компилятор создает для вас пассивный конструктор).

Полезно, чтобы в качестве составной части инициализации конструктор в производном классе вызывал конструктор своего базового класса, что позволит конструктору базового класса выполнить требующуюся ему дополнительную инициализацию. Для вызова конструктора базового класса можно при определении конструктора наследующего класса указать ключевое слово base:

```
class Mammal // базовый класс
{
    public Mammal(string name) // конструктор базового класса
    {
      ...
    }
    ...
}

class Horse : Mammal // производный класс
{
    public Horse(string name) : base(name) // вызывает Mammal(name)
    {
      ...
    }
    ...
}
```

Если в конструкторе производного класса нет явного указания на вызов конструктора базового класса, компилятор перед выполнением кода в конструкторе производного класса пытается по умолчанию вставить вызов пассивного конструктора базового класса. Если взять ранее показанный пример, то компилятор переписывает код

```
class Horse : Mammal
{
        public Horse(string name) {
                ...
        }
        ...
}
```

в следующий код:

```
class Horse : Mammal
{
        public Horse(string name): base() {
                ...
        }
        ...
}
```

Этот код работает в том случае, если у класса Mammal имеется открытый пассивный конструктор. Но такой конструктор есть не у всех классов (к примеру, стоит вспомнить, что компилятор создает пассивный конструктор, только если вами не созданы какие-либо активные конструкторы), и если в таком случае забыть вызвать правильный конструктор базового класса, это приведет к ошибке в ходе компиляции.

Задание на лабораторную работу

Задание 1

Создать классы необходимые для описания сведений об абитуриенте и методы для управления полученными данными. В программе необходимо обрабатывать информацию о паспортных данных абитуриента, об уровне образования, об экзаменах, о выбранных специальностях и прочее (на свое усмотрение) посредством создания классов соответствующих документов. При этом необходимо выстроить иерархию классов.

Задание 2

Разработать многооконный интерфейс, добавляя правой кнопкой в дерево проекта новые компоненты «форма» и добавляя кнопки для открытия (закрытия) форм, например:

```
Form2 f2 = new Form2();
```

Задание 3

Разработать визуальное приложение, содержащее базовые и наследуемые классы. Проверить их работу с помощью многооконного интерфейса.

Задание 4

Учесть области видимости в классах.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.
2. Запустить приложение MS Visual Studio.
3. Создать новый проект.
4. Выполнить полученное задание в приложении Windows Forms (.NET Framework).
5. Сохранить результаты лабораторной работы.
6. Подготовить отчет по лабораторной работе.

Лабораторная работа № 12

«Разработка визуального приложения, реализующего принципы полиморфизма на платформе .NET Framework»

Цель работы

Целью лабораторной работы является изучение основ построения визуального приложения, реализующего принципы полиморфизма на платформе .NET Framework.

Краткие теоретические сведения

Определение полиморфизма

Полиморфизм часто называется третьим столпом объектно-ориентированного программирования после инкапсуляции и наследования. Полиморфизм — слово греческого происхождения, означающее "многообразие форм" и имеющее несколько аспектов.

Во время выполнения объекты производного класса могут обрабатываться как объекты базового класса в таких местах, как параметры метода и коллекции или массивы. Когда возникает полиморфизм, объявленный тип объекта перестает соответствовать своему типу во время выполнения.

Базовые классы могут определять и реализовывать виртуальные методы, а производные классы — переопределять их, т. е. предоставлять свое собственное определение и реализацию. Во время выполнения, когда клиент вызывает метод, CLR выполняет поиск типа объекта во время выполнения и вызывает перезапись виртуального метода. В исходном коде можно вызвать метод в базовом классе и обеспечить выполнение версии метода, относящейся к производному классу.

Виртуальные методы позволяют работать с группами связанных объектов универсальным способом. Представим, например, приложение, позволяющее пользователю создавать различные виды фигур на поверхности для рисования. Во время компиляции вы еще не знаете, какие именно виды фигур создаст пользователь. При этом приложению необходимо отслеживать все различные типы создаваемых фигур и обновлять их в ответ на движения мыши. Для решения этой проблемы можно использовать полиморфизм, выполнив два основных действия.

Создать иерархию классов, в которой каждый отдельный класс фигур является производным из общего базового класса.

Применить виртуальный метод для вызова соответствующего метода на любой производный класс через единый вызов в метод базового класса.

Виртуальные методы и реализация полиморфизма

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с использованием процесса, который называется переопределением метода (method overriding). Чтобы пересмотреть текущий дизайн, нужно понять значение ключевых слов virtual и override.

Виртуальным называется такой метод, который объявляется как virtual в базовом классе. Виртуальный метод отличается тем, что он может быть переопределен в одном или нескольких производных классах. Следовательно, у каждого производного класса может быть свой вариант виртуального метода. Кроме того, виртуальные методы интересны тем, что именно происходит при их вызове по ссылке на базовый класс. В этом случае средствами языка C# определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем это делается во время выполнения. Поэтому при ссылке на разные типы объектов выполняются разные варианты виртуального метода. Иными словами, вариант выполняемого виртуального метода выбирается по типу объекта, а не по типу ссылки на этот объект.

Так, если базовый класс содержит виртуальный метод и от него получены производные классы, то при обращении к разным типам объектов по ссылке на базовый класс выполняются разные варианты этого виртуального метода.

Метод объявляется как виртуальный в базовом классе с помощью ключевого слова virtual, указываемого перед его именем. Когда же виртуальный метод переопределяется в производном классе, то для этого используется модификатор override. А сам процесс повторного определения виртуального метода в производном классе называется переопределением метода. При переопределении имя, возвращаемый тип и сигнатура переопределяющего метода должны быть точно такими же, как и у того виртуального метода, который переопределяется. Кроме того, виртуальный метод не может быть объявлен как static или abstract.

Переопределение метода служит основанием для воплощения одного из самых эффективных в C# принципов: динамической диспетчеризации методов, которая представляет собой механизм разрешения вызова во время выполнения, а не компиляции. Значение динамической диспетчеризации методов состоит в том, что именно благодаря ей в C# реализуется динамический полиморфизм.

Если при наличии многоуровневой иерархии виртуальный метод не переопределяется в производном классе, то выполняется ближайший его вариант, обнаруживаемый вверх по иерархии.

И еще одно замечание: свойства также подлежат модификации ключевым словом virtual и переопределению ключевым словом override. Это же относится и к индексаторам.

Используя виртуальные методы, можно вызывать различные версии одного и того же метода, основываясь на типе объекта, который определяется динамически в ходе выполнения программы. Рассмотрим следующие примеры классов, определяющих один из вариантов ранее рассмотренной иерархии Mammal:

```
class Mammal {
        ...
        public virtual string GetTypeName() {
                return "This is a mammal" ;
        }
}

class Horse : Mammal {
        ...
        public override string GetTypeName() {
                return "This is a horse";
        }
}

class Whale : Mammal {
        ...
        public override string GetTypeName() {
                return "This is a whale";
        }
}

class Aardvark : Mammal {
        ...
}
```

А теперь изучим следующий блок кода:

```
Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
```

```
Aardvark myAardvark = new Aardvark(...);
myMammal = myHorse;

Console.WriteLine(myMammal.GetTypeName()); // Horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Aardvark
```

Этот же код можно переписать иначе:

```
Mammal myMammal = new Horse(...);
Console.WriteLine(myMammal.GetTypeName()); // Horse

Mammal myMammal = new Whale (...);
Console.WriteLine(myMammal.GetTypeName()); // Whale

Mammal myMammal = new Aardvark (...);
Console.WriteLine(myMammal.GetTypeName()); // Aardvark
```

Что будут выводить на экран консоли три различные инструкции Console.WriteLine? На первый взгляд от всех них следует ожидать вывода на экран строки «This is a mammal», поскольку в каждой инструкции метод GetTypeName вызывается в отношении переменной myMammal, имеющей тип Mammal. Но в первом случае можно увидеть, что myMammal фактически ссылается на Horse. (Вспомним, что Horse-переменную можно присвоить Mammal-переменной, потому что класс Horse наследуется из класса Mammal.) Поскольку метод GetTypeName определен в качестве виртуального, среда выполнения определяет, что нужно вызвать метод Horse.GetTypeName, поэтому инструкция выведет на экран сообщение «This is a horse». Точно такая же логика применяется ко второй инструкции Console.WriteLine, которая выводит сообщение «This is a whale». Третья инструкция вызывает метод Console.WriteLine в отношении объекта Aardvark. Но в классе Aardvark нет метода GetTypeName, поэтому вызывается исходный метод, находящийся в классе Mammal и возвращающий строку «This is a mammal».

Этот механизм вызова с помощью одной и той же инструкции различных методов в зависимости от имеющегося контекста называется полиморфизмом, что буквально означает «множество форм».

Абстрактные классы

Иногда требуется создать базовый класс, в котором определяется лишь самая общая форма для всех его производных классов, а наполнение ее деталями предоставляется каждому из этих классов. В таком классе определяется лишь характер методов, которые должны быть конкретно реализованы в производных классах, а не в самом базовом классе. Подобная ситуация возникает, например, в связи с невозможностью получить содержательную реализацию метода в базовом классе.

Создавая собственные библиотеки классов, вы можете сами убедиться в том, что у метода зачастую отсутствует содержательное определение в контексте его базового класса. Подобная ситуация разрешается двумя способами. Один из них состоит в том, чтобы просто выдать предупреждающее сообщение. Такой способ может пригодиться в определенных ситуациях, например, при отладке, но в практике программирования он обычно не применяется. В подобных случаях требуется какой-то способ, гарантирующий, что в производном классе действительно будут переопределены все необходимые методы. И такой способ в C# имеется. Он состоит в использовании абстрактного метода.

Абстрактный метод создается с помощью указываемого модификатора типа *abstract*. У абстрактного метода отсутствует тело, и поэтому он не реализуется в базовом классе. Это означает, что он должен быть переопределен в производном классе, поскольку его вариант из базового класса просто непригоден для использования. Нетрудно догадаться, что абстрактный метод автоматически становится виртуальным и не требует указания модификатора virtual. В действительности совместное использование модификаторов virtual и abstract считается ошибкой. Для определения абстрактного метода служит приведенная ниже общая форма:

abstract тип имя(список_параметров);

Как видите, у абстрактного метода отсутствует тело. Модификатор abstract может применяться только в методах экземпляра, но не в статических методах (static). Абстрактными могут быть также индексаторы и свойства.

Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный, и для этого перед его объявлением class указывается модификатор abstract. А поскольку реализация абстрактного класса не опреде-

ляется полностью, то у него не может быть объектов. Следовательно, попытка создать объект абстрактного класса с помощью оператора new приведет к ошибке во время компиляции.

Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как abstract. Таким образом, атрибут abstract наследуется до тех пор, пока не будет достигнута полная реализация класса.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   // Создаем абстрактный класс
   abstract class UserInfo
   {
      protected string Name;
      protected byte Age;

      public UserInfo(string Name, byte Age)
      {
         this.Name = Name;
         this.Age = Age;
      }

      // Абстрактный метод
      public abstract string ui();
   }

   class UserFamily : UserInfo
   {
      string Family;

      public UserFamily(string Family, string Name, byte Age) : base (Name,
Age)
      {
         this.Family = Family;
      }
```

```
        // Переопределяем метод ui
        public override string ui()
        {
            return Family + " " + Name + " " + Age;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            UserFamily user1 = new UserFamily("Erohin", "Alexandr", 26);
            Console.WriteLine(user1.ui());

            Console.ReadLine();
        }
    }
}
```

В данном примере создается абстрактный класс UserInfo в котором инкапсулируется абстрактный метод ui(), который, в свою очередь, переопределяется в классе UserFamily.

Следует отметить, что в абстрактные классы вполне допускается (и часто практикуется) включать конкретные методы, которые могут быть использованы в своем исходном виде в производном классе. А переопределению в производных классах подлежат только те методы, которые объявлены как abstract.

Интерфейсы

1. Назначение интерфейсов. Особенности применения интерфейсов в C#

Интерфейс определяет ряд методов (свойств, индексаторов, событий), которые должны быть реализованы в классе, который наследует (реализует) данный интерфейс. Интерфейсы используются для того, чтобы указать классам, что именно нужно реализовать в этих классах. Реализовывать нужно методы (свойства, индексаторы, события). Таким образом, интерфейс описывает функциональные возможности без конкретной реализации. Иными словами, интерфейс определяет спецификацию, но не реализацию.

Использование интерфейсов есть эффективным в случаях, когда нужно создать альтернативу множественного наследования. Любой класс может унаследовать несколько интерфейсов. При этом все методы унаследованных интерфейсов должны быть реализованы в классе.

В интерфейсах можно указывать следующие элементы:

– методы;

– свойства;

– индексаторы;

– события.

Так как свойства, индексаторы и события так или иначе для компилятора представляют собой методы, то можно сказать, что интерфейс содержит методы. Но при этом интерфейсы не могут содержать:

– члены данных;

– конструкторы;

– деструкторы;

– операторные методы.

Интерфейс может быть реализован в любом количестве классов.

В одном классе может быть реализовано любое количество интерфейсов.

Структура так же, как и класс может реализовывать любое количество интерфейсов.

Особенности интерфейсов:

– в интерфейсе нельзя вписывать реализацию его элементов;

– невозможно создать экземпляр интерфейса;

– можно создать ссылку на интерфейс;

– в интерфейсе не может быть конструкторов;

– интерфейс не может содержать поля;

– в интерфейсе не может быть осуществлена перегрузка операторов;

– все методы интерфейса по умолчанию объявлены как public.

При использовании интерфейсов в классах-наследниках:

– запрещено изменять модификатор доступа для метода при его реализации;

– невозможно объявить методы интерфейса как virtual;

– запрещено объявлять методы интерфейса с ключевым словом static (как статические).

2. Отличие между интерфейсами и абстрактными классами

В языке программирования C# существуют следующие отличия между интерфейсами и абстрактными классами:

- В интерфейсе запрещено прописывать реализацию его членов. В абстрактном классе часть членов может иметь реализацию. Иными словами, интерфейс это тот же абстрактный класс, у которого все методы абстрактные.
- В интерфейсе запрещено описывать поля (переменные, объекты), в абстрактном классе можно.
- Интерфейс не может содержать конструктор. В абстрактном классе может быть объявлен конструктор.
- Любой класс может быть унаследован от нескольких интерфейсов. При этом любой класс может быть унаследован только от одного абстрактного класса (и не более).

3. Синтаксис интерфейса

Интерфейсы объявляются с помощью ключевого слова interface. Общая форма описания интерфейса, в котором определяются методы, следующая:

```
interface имя
{
   возвращаемый_тип1 имя_метода1(параметры1);
   возвращаемый_тип2 имя_метода2(параметры2);
   // ...
   возвращаемый_типN имя_методаN(параметрыN);
}
```

где

имя – конкретное имя интерфейса;
имя_метода1, имя_метода2, …, имя_методаN – имена методов интерфейсов;

возвращаемый_тип1, возвращаемый_тип2, …, возвращаемый_типN – типы, которые возвращаются методами интерфейса;

параметры1, параметры2, …, параметрыN – списки параметров методов интерфейса.

Кроме методов, в интерфейсах можно указывать свойства, события и индексаторы.

4. Общая форма реализации интерфейса в классе

Общая форма реализации интерфейса в классе имеет следующий вид:

```
class имя_класса : имя_интерфейса
{
   // тело класса
   ...
}
```

где имя_интерфейса – имя интерфейса, методы (свойства, индексаторы, события) которого реализуются в классе. Класс обязательно должен реализовать все методы интерфейса.

5. Общая форма класса, реализующего несколько интерфейсов

Класс может реализовать несколько интерфейсов. В этом случае все интерфейсы определяются списком через запятую.

Общая форма класса реализующего несколько интерфейсов:

```
class имя_класса : имя_интерфейса1, имя_интерфейса2, ..., имя_интерфейсаN
{
   // тело класса
   ...
}
```

где имя_интерфейса1, имя_интерфейса2, …, имя_интерфейсаN – имена интерфейсов, которые должен реализовать класс. Класс должен реализовать все методы всех интерфейсов.

6. Пример объявления интерфейса и класса, наследующего этот интерфейс

В данном примере интерфейсу присваивается имя IMyInterface. Рекомендовано к имени интерфейса добавить префикс ‘I’ в соответствии с общераспространенной практикой.

Интерфейс объявлен как public.

```
public interface IMyInterface
{
   int MyGetInt(); // метод, возвращающий число типа int
   double MyGetPi(); // метод, возвращающий число Pi
   int MySquare(int x); // метод, возвращающий x в квадрате
   double MySqrt(double x); // метод возвращающий корень квадратный
из x
}
```

В данном примере, в интерфейсе объявлено описание четырех методов, которые должны быть реализованы во всех классах, определяющих эти интерфейсы. Это методы: MyGetInt(), MyGetPi(), MySquare(), MySqrt().

Пример описания класса использующего этот интерфейс.

```
public class MyClass : IMyInterface
{
   // модификатор доступа public
   public int MyGetInt()
   {
      return 25;
   }

   public double MyGetPi()
   {
      return Math.PI;
   }

   public int MySquare(int x)
   {
      return (int)(x * x);
   }
```

```
    public double MySqrt(double x)
    {
        return (double)Math.Sqrt(x);
    }
}
```

Все методы, которые определяются в классе, должны иметь тип доступа public. Если установить другой тип доступа (private или protected), то Visual Studio выдаст следующее сообщение:

"MyClass does not implement interface member MyFun() because it is not public."

где MyFun() – название функции, которая реализована в классе с модификатором доступа private или protected.

Это связано с тем, что в самом интерфейсе эти методы неявно считаются открытыми (public). Поэтому их реализация должна быть открытой.

7. Пример объявления двух интерфейсов и класса, который реализует методы этих интерфейсов

В нижеследующем примере объявлено два интерфейса с именами MyInterface и MyInterface2. Первый интерфейс содержит 4 методы. Второй интерфейс содержит 1 метод.

Также объявлен класс MyClass, использующий эти два интерфейса. Класс обязательно должен реализовать все методы обоих интерфейсов, то есть в сумме 5 методов.

```
public interface IMyInterface
{
    int MyGetInt(); // метод, возвращающий число типа int
    double MyGetPi(); // метод, возвращающий число Pi
    int MySquare(int x); // метод, возвращающий x в квадрате
    double MySqrt(double x); // метод, возвращающий корень квадратный
из x
}


public interface IMyInterface2
{
```

```
    double MySqrt2(double x); // корень квадратный из x
}

public class MyClass : IMyInterface, IMyInterface2
{
    // методы из интерфейса MyInterface
    public int MyGetInt()
    {
        return 25;
    }

    public double MyGetPi()
    {
        return Math.PI;
    }

    public int MySquare(int x)
    {
        return (int)(x * x);
    }

    public double MySqrt(double x)
    {
        return (double)Math.Sqrt(x);
    }

    // метод из интерфейса MyInterface2
    public double MySqrt2(double x)
    {
        return (double)Math.Sqrt(x);
    }
}
```

8. Пример использования ссылки на интерфейс для доступа к методам класса

В C# допускается описывать ссылки на интерфейс. Если описать переменную-ссылку на интерфейс, то с ее помощью можно вызвать методы класса, который использует этот интерфейс.

Пример.

```
public interface IMyInterface
{
   double MyGetPi(); // метод, возвращающий число Pi
}

class MyClass : IMyInterface
{
   // методы из интерфейса MyInterface
   public double MyGetPi()
   {
      return Math.PI;
   }

}

// вызов из программного кода
private void button1_Click(object sender, EventArgs e)
{
   MyClass mc = new MyClass(); // создание объекта класса mc
   IMyInterface mi; // ссылка на интерфейс
   double d;

   mi = mc; // mi ссылается на объект класса mc
   d = mi.MyGetPi(); // d = 3.14159265358979

   label1.Text = d.ToString();
}
```

В данном примере создается объект (экземпляр) класса MyClass с именем mc. Затем описывается ссылка на интерфейс IMyInterface с именем mi.

Строка «mi=mc;» приводит к тому, что ссылка mi указывает на объект класса mc. Таким образом, через ссылку mi можно иметь доступ к методам класса MyClass, так как класс MyClass реализует методы интерфейса IMyInterface.

С помощью ссылки на интерфейс можно иметь доступ к методам классов, которые реализуют описанные в этом интерфейсе методы.

9. Описание свойства в интерфейсе

Свойство описывается в интерфейсе без тела. Общая форма объявления интерфейсного свойства следующая:

```
тип имя_свойства
{
   get;
   set;
}
```

Если свойство предназначено только для чтения, то используется один только аксессор get.

Если свойство предназначено для записи, то используется только один аксессор set.

Пример. Описывается интерфейс и класс. Класс возвращает свойство MyPi.

```
public interface IMyInterface
{
   double MyGetPi(); // метод, возвращающий число Pi

   // свойство, возвращающее число Pi
   double MyPi
   {
      get;
   }
}

class MyClass : IMyInterface
{
   // метод
   public double MyGetPi()
   {
      return Math.PI;
   }

   // реализация свойства в классе
   public double MyPi
   {
      get
      {
         return Math.PI;
      }
   }
```

```
    }

    // использование интерфейсного свойства в обработчике события клика
на кнопке
    private void button1_Click(object sender, EventArgs e)
    {
        MyClass mc = new MyClass(); // создание объекта класса mc

        label1.Text = mc.MyPi.ToString(); // чтение свойства
    }
```

10. Пример описания индексатора в интерфейсе

Общая форма объявления интерфейсного индексатора имеет вид:

```
тип this[int индекс]
{
  get;
  set;
}
```

Пример описания и использования интерфейсного индексатора, который считывает элемент из массива, состоящего из 5 элементов типа double.

```
public interface IMyInterface
{
    // интерфейсный индексатор
    double this[int index]
    {
        get;
    }
}

class MyClass : IMyInterface
{
    double[] mas = { 3, 2.9, 0.5, 7, 8.3 };
    public double this[int index]
    {
        get
        {
            return mas[index];
        }
```

```
    }
}

private void button1_Click(object sender, EventArgs e)
{
    MyClass mc = new MyClass(); // создание объекта класса mc
    double d;
    d = mc[2]; // d = 0.5
    label1.Text = d.ToString();
}
```

15. Как работает механизм наследования интерфейсов?

Интерфейс может наследовать другой интерфейс. Синтаксис наследования интерфейсов такой же, как и у классов.

Общая форма наследования интерфейса следующая:

interface имя_интерфейса : имя_интерфейса1, имя_интерфейса2, ..., имя_интерфейсаN

```
{
    // методы, свойства, индексаторы и события интерфейса
    ...
}
```

где имя_интерфейса – имя интерфейса, который наследует другие интерфейсы;

имя_интерфейса1, имя_интерфейса2, …, имя_интерфейсаN – имена интерфейсов-предков.

Пример. В данном примере класс MyClass использует интерфейс, который наследует другой интерфейс. В классе нужно реализовать все методы (свойства, индексаторы, события) интерфейса MyInterface1 и интерфейса MyInterface2.

```
// базовый интерфейс
interface MyInterface1
{
    void Int1_Meth();
}
```

```
// интерфейс, который наследует другой интерфейс
interface MyInterface2 : MyInterface1
{
   void Int2_Meth();
}

// класс, который использует интерфейс MyInterface2
class MyClass : MyInterface2
{
   // реализация метода интерфейса MyInterface1
   public void Int1_Meth()
   {
      // тело метода
      // ...
      return;
   }

   // реализация метода интерфейса MyInterface2
   public void Int2_Meth()
   {
      // тело метода
      // ...
      return;
   }
}
```

11. Явная реализация члена интерфейса

Если перед именем метода (свойства, индексатора, события) стоит имя интерфейса через разделитель ‘ . ‘ (точка), то это называется явной реализацией члена интерфейса.

Пример явной реализации.

```
// базовый интерфейс
interface MyInterface1
{
   void Method();
}

// класс, который реализует интерфейс MyInterface1
class MyClass : MyInterface1
{
```

```
    // явная реализация метода интерфейса MyInterface1
    void MyInterface1.Method() // указывается имя интерфейса
    {
        // тело метода
        // ...
        return;
    }
}
```

12. Пример явной реализации члена интерфейса

Явная реализация члена интерфейса применяется в следующих случаях:

– когда нужно, чтобы интерфейсный метод был доступен по интерфейсной ссылке, а не по объекту класса, реализующего данный интерфейс. В этом случае интерфейсный метод не является открытым (public) членом класса (см. пример 1);

– когда в одном классе реализованы два интерфейса, в которых методы имеют одинаковые имена и сигнатуру (см. пример 2).

Пример 1. Явная реализация интерфейсного метода. По интерфейсной ссылке метод есть доступен, а по объекту класса недоступен.

```
// Интерфейс
interface MyInterface1
{
    void Method();
}

// класс, вызывающий интерфейс
class MyClass : MyInterface1
{
    // явная реализация метода интерфейса MyInterface1
    // модификатор доступа, должен отсутствовать
    void MyInterface1.Method() // указывается имя интерфейса
    {
        // тело метода
        // ...
        return;
    }

    void InternalMethod()
    {
        MyInterface1 mi = this; // mi - интерфейсная ссылка
```

```
            mi.Method(); // работает!

            MyClass mc = this; // mc - объект класса MyClass
            // mc.Method() - невозможно вызвать - метод не открыт для объекта
        }
    }
```

Пример 2. Есть два интерфейса MyInterface1 и MyInterface2. Каждый из них имеет методы с одинаковыми именами и сигнатурами. В данном случае это метод Method(), не возвращающий параметров (void). С помощью явной реализации класс распознает эти методы.

```
    // интерфейс 1
    interface MyInterface1
    {
        void Method();
    }

    // интерфейс 2
    interface MyInterface2
    {
        void Method();
    }

    // класс, который использует два интерфейса
    class MyClass : MyInterface1, MyInterface2
    {
        // явная реализация - модификатор доступа (public) должен отсутство-
вать
        // метод из интерфейса MyInterface1
        void MyInterface1.Method()
        {
            // тело метода
            // ...
            return;
        }

        // метод из интерфейса MyInterface2
        void MyInterface2.Method()
        {
            // тело метода
            // ...
            return;
        }
    }
```

Задание на лабораторную работу

Задание 1

В рамках иерархии классов, полученной в предыдущей лабораторной работе, добавить виртуальный метод в базовый класс и перегрузить его в производных классах.

Задание 2

Реорганизовать иерархию классов, добавив абстрактный класс и абстрактные методы (или заменить какой-либо класс абстрактным). В случае замены какого-либо класса абстрактным, реорганизовать код таким образом, чтобы методы абстрактного класса были реализованы в производных классах.

Задание 3

Реорганизовать иерархию классов, добавив интерфейс (interface). Вынести в интерфейс общие методы для нескольких классов.

Задание 4

Реализовать объекты полученных классов.

Порядок выполнения лабораторной работы

1. Получить задание у преподавателя.
2. Запустить приложение MS Visual Studio.
3. Создать новый проект.
4. Выполнить полученное задание в приложении Windows Forms (.NET Framework).
5. Сохранить результаты лабораторной работы.
6. Подготовить отчет по лабораторной работе.