МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛОГОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА «УПРАВЛЯЮЩИЕ И ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ»

И Н Ф О Р М А Т И К А

Методические указания к лабораторным работам

Факультет – электроэнергетический

Направление 09.03.01 «Информатика и вычислительная техника»

ВОЛОГДА
2015

УДК 681.3.06(075)

Информатика: методические указания к лабораторным работам. – Вологда: ВоГУ, 2015. – 35 с.

В методических указаниях приведено описание двенадцати лабораторных работ по дисциплине информатика. Лабораторные работы посвящены вопросам выработки практических навыков программирования на языке C#.

Утверждено редакционно-издательским советом ВоГУ

Составитель: А. В. Машкин, канд. техн. наук, доцент каф. УВС ВоГУ

Рецензент: А. А. Суконщиков, канд. техн. наук, доцент, зав. каф. АВТ ВоГУ

ВВЕДЕНИЕ

До настоящего времени не существует установившегося и общепринятого определения термина информатика, в том смысле, в каком он используется в учебно-методической и научной литературе. Это связано прежде всего с тем, что очень трудно дать какое-либо общее определение термина информатика, точно так же, как например, трудно дать определение общего термина математика. Информатика – это и наука, и область прикладных исследований, и область междисциплинарных исследований, и учебная дисциплина. Тем не менее в [1] дается следующее определение термина информатика: «Информатика — молодая научная дисциплина, изучающая вопросы, связанные с поиском, сбором, хранением, преобразованием и использованием информации в самых различных сферах человеческой деятельности».

Согласно [1] информатика подразделяется на ряд разделов, одним из которых является прикладная информатика. Одним из центральных вопросов прикладной информатики является *инженерия программного обеспечения* (*Software Engineering*) под которой понимается систематический процесс разработки от этапа формирования идеи до этапа создания готового программного обеспечения (ПО).

Современные технологии разработки ПО ориентированы на использования языков высокого уровня (ЯВУ). Основной целью применения ЯВУ является облегчение и ускорение процесса разработки ПО. Существует достаточно большое количество ЯВУ, популярность которых может, например, оцениваться по значению TIOBE Index [2]. В данных методических указаниях описаны лабораторные работы, посвященные выработке навыков программирования на языке C#. C# является одним из языков программирования, входящих в интегрированную среду разработки Microsoft Visual Studio предназначен для создания управляемых приложений под платформу .NET. Выбор данного ЯВУ объясняется тем, что он используется в большинстве технологий программирования от компании Microsoft [3].

Большинство описанных лабораторных работ посвящено программированию консольных приложений, поскольку на данном этапе обучения основной целью является изучение и выработка практических навыков использования синтаксических конструкций самого языка C#. Сложные структуры данных и наиболее часто употребляемые алгоритмы обработки данных, а также профессионально использование языка C# в технологиях программирования от Microsoft изучаются в дальнейших учебных курсах («Инженерная и компьютерная графика», «Программирование», «Технологии программирования» и т.д.). Большинство заданий на лабораторные работы приведены в [4], в

случае изменения задания это указывается особо. Номер варианта задания на лабораторную работу определяется номером студента по журналу.

Лабораторная работа № 1
Линейные программы

1.1. Цель работы

Научится разрабатывать линейные программы и пользоваться возможностями интегрированных сред разработки (IDE) для отладки приложений.

1.2. Теоретические сведения

В принципе, исходный текст программы на ЯВУ можно написать в любом текстовом редакторе (например, в блокноте) и используя командную строку выполнить компиляцию исходного текста программы (естественно, лишь при наличии установленного в системе соответствующего компилятора). Для компиляции программ, написанных на языке C# используется программа *csc.exe* (*C Sharp Compiler* – компилятор языка C#). Параметры компиляции задаются при помощи ключей, записываемых после имени *csc*. Эти параметры должны сопровождаться префиксом либо в виде символа «-», либо символа «/». Наиболее часто используемые ключи компилятора *csc* приведены в таблице 1.1.

Таблица 1.1

Ключи компилятора *csc.exe*

Ключ	Описание
out	Этот ключ применяется для указания имени создаваемой сборки. По умолчанию сборке присваивается то же имя, что у входного файла *.cs
target:exe	Этот ключ позволяет создавать исполняемое консольное приложение. Сборка такого типа генерируется по умолчанию, потому при создании подобного приложения данный параметр можно пропустить
target:library	Этот ключ позволяет создавать однофайловую сборку *.dll
target:module	Этот ключ позволяет создавать модуль. Модули являются элементами многофайловых сборок
target:winexe	Хотя приложения с графическим пользовательским интерфейсом можно создавать с применением ключа /target: exe, параметр /target: winexe позволяет предотвратить открытие окна консоли под остальными окнами

Пусть, например, в программе Notepad (блокнот) создан следующий исходный файл на языке C# и сохранен под именем *Test.cs*:

```
using System;
```

```
class TestApplication
{ static void Main()
   {Console.WriteLine("Hello, world!");
  Console.ReadLine();
    }
}
```

Для компиляции программы при помощи командной строки удобно воспользоваться командной строкой из пакета Microsoft Visual Studio. Если использовать командную строку из операционной системы, то придется либо указывать полные пути к файлам, либо прописывать переменные окружения. Если исходный файл *Test.cs* был сохранен, например, в папке C:\MyProject, то для создания исполняемого файла *Test.exe* необходимо выполнить следующие действия:

1) Перейти с помощью команды *cd* к папке, где хранится исходный файл *Test.cs* (для рассматриваемого примера C:\MyProject).
2) Ввести и выполнить командную строку: *csc /t :exe Test.cs*.

Несмотря на то, что компилировать программы можно только при помощи командной строки, гораздо удобнее при разработке ПО пользоваться ИСР (интегрированными средами разработки) (англ. IDE - *Integrated development environment или integrated debugging environment)*.

Обычно среда разработки включает в себя:

- текстовый редактор
- компилятор и/или интерпретатор
- средства автоматизации сборки
- отладчик.

Современные IDE, как правило, также содержит разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. IDE, поддерживающие объектно-ориентированную методологию разработки, включают средства, облегчающие такую разработку, например: браузер классов, инспектор объектов, диаграмму иерархии классов и т.д. Существуют IDE, предназначенные для разработки ПО с использованием нескольких языков программирования — такие, как Eclipse, NetBeans или Microsoft Visual Studio, так и IDE предназначается для одного определённого языка программирования - как, например, Visual Basic, Dev-C++.

IDE были созданы для увеличения производительности труда программиста, что достигается путем тесной связи графического пользовательского интерфейса с внутренними компонентами IDE. В графических IDE-средах в конечном итоге все заканчивается предоставлением компилятору инструкций относительно того, что следует делать с входными файлами кода. Это позво-

ляет выполнять разработчику меньше действий для переключения режимов, по сравнению с использованием дискретных средств разработки.

При создании реальных программ неизбежно возникают ошибки. Для устранения логических ошибок (связанных с неправильными действиями программы с точки зрения разработчика) удобно воспользоваться пошаговым режимом работы отладчика. Поскольку место возникновения ошибки может находиться достаточно далеко от начала программы, то в этом случае удобнее воспользоваться точками останова.

Точку останова (breakpoint) в Visual Studio можно помещать на любую строку кода, которая в действительности выполняется. Самый простой способ – щелчок на необходимой строке в окне редактора кода внутри затененной области вдоль левого края окна документа (или выделение нужной строки и нажатие клавиши <F9>). Это приводит к размещению в данной строке точки останова, которая вызывает прерывание процесса выполнения и передачу управления отладчику. Точка останова обозначается большим кружком слева от соответствующей строки в окне редактора кода. Кроме того, Visual Studio выделяет саму строку, отображая ее текст и фон разными цветами. Щелчок на кружке приводит к удалению точки останова.

Если останов на определенной строке каждый раз не подходит для решения имеющейся проблемы, можно создать так называемую условную точку останова. Для этого необходимо выбрать в меню Debug (Отладка) пункт Windows --- Breakpoints (Окно --- Точки останова). Откроется диалоговое окно, позволяющее указать желаемые детали для точки останова. В этом окне можно выполнять следующие действия:

1) Указать, что выполнение должно прерываться лишь после прохождения точки останова определенное количество раз.

2) Указать, что точка останова должна вступать в действие при каждом n-ном достижении строки, например, при каждом 20-м ее выполнении (это удобно при отладке больших циклов).

3) Задать точки останова относительно переменных, а не команд. В таком случае наблюдение будет вестись за значением указанной переменной, и точки останова будут активизироваться при каждом изменении значения этой переменной.

Более подробную информацию об использовании возможностей IDE Microsoft Visual Studio можно найти в [5].

1.3. Программа работы

1) В зависимости от номера варианта задания написать программу [4]. После написания программы выполнить ее компиляцию с помощью IDE Microsoft Visual Studio.

2) В IDE следует установить точки останова и выполнить программу в пошаговом режиме работы.

3) С помощью любого файлового менеджера (FAR, Total Commander и т.п.) найти созданный исполняемый файл (файл с расширением *.exe) и удалить его. Далее с помощью командной строки из файла с исходным текстом вновь создать исполняемый файл при помощи командной строки Visual Studio (csc /t:exe *.cs). Предварительно следует при помощи командной строки перейти в каталог, где находится исходный текст программы (файл с расширением *.cs). Для смены диска в командной строке следует ввести его имя и двоеточие (например, F:). Для смены каталога внутри диска следует выдать команду cd `путь к каталогу` (например, cd F:\VC).

Лабораторная работа № 2
Использование условных операторов

2.1. Цель работы

Научиться использовать условные операторы для организации разветвляющихся вычислительных процессов.

2.2. Теоретические сведения

Условные операторы позволяют управлять потоком выполнения программы. В зависимости от условий, сложившихся в процессе вычислений, будет выполняться либо та, либо иная ветвь программы.

Для организации условного ветвления язык C# унаследовал от C и C++ конструкцию *if...else*. Ее синтаксис следующий:

`if (условие) оператор` [*`else оператор`*]

Здесь *оператор* – это простой или составной оператор (заключенный в фигурные скобки) оператор языка C#. В отличие от языков C и C++, в C# условный оператор *if* может работать только с булевскими выражениями, но не с произвольными значениями вроде -1 и 0. Если проверяемое условие истинно, то есть логическое выражение, стоящее в круглых скобках возвращает значение *true*, то будет выполняться оператор, стоящий за круглыми скобками. Если логическое выражение возвращает значение *false*, то будет выполняться оператор, стоящий после ключевого слова *else*. Часть, стоящая после ключевого слова *else* может отсутствовать, в этом случае при возврате выражением значения *false* произойдет переход к выполнению следующего оператора. В операторе *if* может осуществляться проверка не одного, а нескольких условий. В этом случае они должны быть связаны между собой с помощью логических операций И (в языке C# обозначается как &&) и ИЛИ (в языке C#

обозначается как ||). Операторы *if* можно вкладывать друг в друга, причем уровень вложенности практически ничем не ограничен.

Вторым оператором ветвления в языке C# является *switch*. Он обеспечивает многонаправленное ветвление программы. Несмотря на то что многонаправленное ветвление может быть организовано с помощью последовательного ряда вложенных операторов *if*, во многих случаях более эффективным оказывается применение оператора *switch*, поскольку он более нагляден и улучшает читабельность программы. Синтаксис оператора *switch* в языке C# выглядит следующим образом:

```
switch(выражение) {
case константа_1: оператор_1; оператор_перехода;
case константа_2:оператор_2; оператор_перехода;
case константа_3: оператор_3; оператор_перехода;
...
[default: оператор; оператор_перехода;]
}
```

Выполнение оператора *switch* начинается с вычисления значения выражения. В общем случае выражение может возвращать значение, принадлежащее перечислимому типу, а также к типам данных, для которых возможно неявное преобразование к одному из целочисленных типов, символьному типу (*char*) и строковому типу (*string*), либо непосредственно сами эти типы. Далее происходит сравнение вычисленного значения выражения со значениями констант. В более общем случае, вместо констант, могут также употребляться константные выражения. В случае совпадения будет выполнен оператор (простой или составной), находящейся на этой ветви программы. В качестве `оператор_перехода` в операторе *switch* может использоваться один из трех операторов: *break, return* или *goto*. Наиболее часто используется оператор *break*, который выполняет выход из самого внутреннего из объемлющих его операторов. В случае, если совпадений ни в одной из ветвей *case* не было найдено, то управление передается на ветвь `default`. После чего управление будет передано следующему за оператором *switch* оператору. Ветвь *default* не является обязательной. Несмотря на необязательность этой ветви ее рекомендуется всегда включать в оператор *switch*, поскольку это облегчает отладку программы. Более подробную информацию об использовании условных операторов можно найти в [4,6].

2.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 3
Использование операторов цикла

3.1. Цель работы

Научиться использовать операторы цикла для организации многократно повторяющихся вычислений.

3.2. Теоретические сведения

В языке C# имеются четыре различных вида циклов (*for, while, do...while и foreach*), позволяющих выполнять блок кода повторно до тех пор, пока удовлетворяется определенное условие. Многократно выполняющийся оператор (простой или составной) называется телом цикла. Один проход цикла называется *итерацией*. Проверка условия продолжения цикла выполняется на каждой итерации либо до входа в тело цикла (цикл с предусловием), либо после выхода из тела цикла (цикл с постусловием). Таким образом, тело цикла с предусловием может не выполниться ни разу, а тело цикла с постусловием обязательно выполняется хотя бы один раз (это может пригодится, когда операторы находящиеся в теле цикла обязательно должны быть выполнены, например, при вводе данных с клавиатуры). *Параметром цикла* называется переменная, которая используется при проверке условия продолжения выполнения цикла и принудительно изменяется на каждой итерации цикла, как правило, на одну и ту же величину. Если параметр цикла представляет из себя целочисленную переменную, то он называется *счетчиком цикла*. Количество проходов такого цикла можно определить заранее. Синтаксис оператора цикла с предусловием в языке C# выглядит следующим образом:

```
while(условие) оператор;
```

Если проверяемое условие истинно, т.е. логическое выражение возвращает значение *true*, то выполнение цикла продолжается. Когда выражение вернет значение *false*, то выполнение цикла прекратится. Синтаксис оператора цикла с предусловием в языке C# выглядит следующим образом:

```
do {оператор;} while (условие);
```

Если тело цикла состоит из одного простого оператора, то фигурные скобки могут быть пропущены. Проверка условия, осуществляется точно так же как и в цикле с предусловием.

Цикл с параметром *for* в языке C# реализован как цикл с предусловием. Синтаксис цикла с параметром выглядит следующим образом:

```
for (инициализатор; условие; модификации)оператор;
```

Инициализатор служит для объявления переменных, используемых в цикле, и присвоения им начальных значений. В нем можно записать несколько операторов, разделяя их запятой. Областью действия переменных, объяв-

ленных в инициализаторе, является весь цикл. Инициализация выполняется один раз в начале цикла. Если проверяемое условие истинно, то происходит выполнение тела цикла, в противном случае выполнение цикла прекращается. Модификации выполняются в каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую.

Цикл *foreach* позволяет осуществить перебор в специально организованной группе данных. Синтаксис цикла *foreach* выглядит следующим образом:

```
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. 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). Примерами коллекций могут служить массивы C#, классы коллекций из пространства имен System.Collection, а также пользовательские классы коллекций. Более подробные сведения о использовании операторов цикла содержатся в [4, 6].

3.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 4
Простейшие классы

4.1. Цель работы

Научиться создавать классы и использовать их при программировании в Visual C#.

4.2. Теоретические сведения

Классом в языке C# называется особая структура, которая может иметь в своем составе поля, свойства, методы и другие элементы. Такой тип данных также называется объектным типом. Описание класса на языке C# выглядит следующим образом:

```
[атрибуты][спецификаторы] class имя_класса [ : предки ]
        тело_класса
```

Обязательными являются только ключевое слово *class*, имя и тело класса. Тело класса — это список описаний его элементов, заключенный в фигурные скобки. Список может быть пустым, если класс не содержит ни одного элемента. Таким образом, простейшее описание класса может выглядеть так:

```
class MyObject {}
```

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов приведены в табл. 4.1.

Спецификаторы 2-6 (табл. 4.1) называются спецификаторами доступа. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут присутствовать в описании только в вариантах, приведенных в табл. 3.1, а также могут комбинироваться с другими спецификаторами. Для описания классов, которые непосредственно описываются в пространстве имен (не вложенные классы), допускается использовать только два спецификатора: `public` и `internal`. По умолчанию, т.е. если ни одни спецификатор доступа не указан, подразумевается спецификатор `internal`.

Чтобы использовать любой тип данных в программе, нужно, как минимум, объявить переменную этого типа. Переменная объектного типа называется экземпляром класса или просто объектом. Т.е. класс это описание, объект – то, что создано в соответствии с этим описанием. Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции *new*, например:

```
MyObject a = new MyObject();// создание экземпляра класса MyObject
MyObject b = new MyObject();// создание другого экземпляра класса MyObject
```

Таблица 4.1

Спецификаторы класса

№	Спецификатор	Описание
1	`new`	Используется для вложенных классов. Задает новое описание класса взамен унаследованного от предка.
2	`public`	Доступ не ограничен
3	`protected`	Используется для вложенных классов. Доступ только из элементов данного и производных классов
4	`iternal`	Доступ только из данной программы (сборки)
5	`protected internal`	Доступ только из данного и производных классов или из данной программы (сборки)
6	`private`	Используется для вложенных классов. Доступ только из элементов класса, внутри которых описан данный класс
7	`abstract`	Абстрактный класс. Применяется в иерархиях объектов
8	`sealed`	Бесплодный класс. Применяется в иерархиях объектов
9	`static`	Статический класс, введен во вторую версию языка Visual C#

Поля объекта аналогичны полям записи. Это – данные, уникальные для каждого созданного в программе экземпляра класса. Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. Кроме того, в классе могут присутствовать статические элементы, которые существуют в единственном экземпляре для всех объектов данного класса. Часто статические данные называют данными класса, а остальные — данными экземпляра.

В отличие от полей, методы у всех объектов одного класса общие, т.е. они и другие функциональные элементы класса не тиражируются и всегда хранятся в памяти в единственном экземпляре. Методы – это процедуры и функции, описанные внутри класса и предназначенные для выполнения операций над его полями. В состав класса входит указатель на специальную таблицу, где содержится вся информация, нужная для вызова методов. От обычных функций методы отличаются тем, что им при вызове неявно передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут поля именно того объекта, который их вызвал. Внутри метода указатель на вызвавший его объект доступен под зарезервированным словом *this*. Для работы с данными класса используются методы класса (статические методы), для работы с данными экземпляра — методы экземпляра, или просто методы. Поля и методы являются основными элементами класса. Кроме того, в классе можно задавать целый ряд других элементов: константы, свойства, индексаторы, конструкторы, деструкторы, операции, события, вложенные типы данных. Более подробную информацию о работе с классами можно найти в [4,6].

4.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 5
Одномерные массивы

5.1. Цель работы

Научиться работать с одномерными массивами в программах на языке C#.

5.2. Теоретические сведения

Массив представляет собой совокупность однотипных переменных с общим для обращения к ним именем. Различаются элементы массива по номеру, который в этом случае называется индексом. В C# массивы могут быть как одномерными, так и многомерными. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных.

В языке C# массивы реализованы в виде объектов. Поэтому в начале необходимо объявить переменную, которая позволит обращаться к массиву, а далее создать экземпляр массива при помощи операции *new*.

Следует иметь в виду, что если массив только объявляется, но явно не инициализируется, то каждый его элемент будет установлен в значение, принятое по умолчанию для соответствующего типа данных (например, элементы массива типа *bool* будут устанавливаться в *false*, а элементы массива типа *int* — в 0). Во всех версиях языка C# доступны следующие варианты описания одномерных массивов:

тип [] имя;
тип [] имя=new тип [размерность];
тип [] имя=new {список_инициализаторов};
тип [] имя=new тип []{список_инициализаторов};
тип [] имя= new тип [размерность]{список_инициализаторов};

В первом случае фактически описана только ссылка на массив, а память под его элементы будет выделена во время их инициализации. Начиная с третьей версии языка C# появилась возможность определять неявно типизированные массивы, используя для этого ключевое слово *var*, например:

```
var arr1 = new[] { 1, 2, 3 };//1
var arr2 = new[] { "One", "Two", "Three" };//2
```

Компилятор сам выведет тип данных элементов массива, основываясь на типе данных констант, приведенных в списке инициализаторов. Так для оператора 1 тип данных элементов массива будет *int*, а для оператора 2 *string*.

С элементами массива можно выполнять все те же самые действия, что и с отдельными переменными того же типа данных. Количество элементов массива называется размерностью. Нумерация элементов массива в языке C# на-

чинается с нуля. Для обращения к элементу массива используется оператор *[]*. Например, запись *Console.WriteLine(arr1[1])* приведет к выводу на консоль числа 2. Поскольку нумерация элементов массива начинается с нуля, то индекс последнего элемента массива будет на единицу меньше его размерности.

Все массивы в языке C# являются потомками класса *System.Array*, который наделяет их некоторыми важными свойствами и методами (табл. 5.1).

Таблица 5.1

Основные элементы класса Array

Название элемента	Тип	Описание
`Length`	Свойство	Количество элементов массива
`IndexOf`	Статический метод	Поиск первого вхождения указанного элемента в одномерный массив
`LastIndexOf`	Статический метод	Поиск последнего вхождения указанного элемента в массив
`Sort`	Статический метод	Упорядочивание элементов одномерного массива по возрастанию. Существует относительно большое количество версий данного перегруженного метода, например, те которые производят сортировку по паре: *ключ, значение*
`BinarySearch`	Статический метод	Двоичный поиск указанного элемента в отсортированном одномерном массиве
`Reverse`	Статический метод	Изменение порядка следования элементов на обратный
`GetValue`	Метод	Получение значения элемента массива
`SetValue`	Метод	Установка значения элемента массива

Более подробную информацию о работе с классами можно найти в [4,6].

5.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 6
Двумерные массивы

6.1. Цель работы

Научиться работать с двумерными массивами в программах на языке C#.

6.2. Теоретические сведения

Язык C# допускает использование многомерных массивов, отличительным признаком которых служат два или более измерений, причем доступ к каждому элементу такого массива осуществляется с помощью определенной

комбинации двух или более индексов. Элементы многомерного массива индексируются двумя или более целыми числами. Важным частным случаем многомерных массивов являются *двумерные (прямоугольные массивы)*. Во всех версиях языка C# доступны следующие варианты описания двумерных массивов:

тип [,] имя;

тип [,] имя=new тип [размерность_1, размерность_2];

тип [,] имя=new {список_инициализаторов};

тип [,] имя=new тип [,]{список_инициализаторов};

тип [] имя= new тип [размерность_1, размерность_2] {список_инициализаторов};

Для обращения к конкретному элементу двумерного массива используют *[]*. В них помещают два индекса, разделяя их запятой. Первый индекс интерпретируется как номер строки, второй как номер столбца, например:

```
int [,] MyArr=new int [{1,2,3},{4,5,6}, {7,8,9}];
Console.WriteLine(MyArr[2,1]);
```

После выполнения этого фрагмента кода на консоль будет выведено значение 8 (нумерация элементов и в строках и в столбцах начинается с нуля, также как и в одномерных массивах). Для обработки элементов двумерных массивов обычно применяются вложенные циклы. В следующем примере создается двумерный массив целочисленных переменных размерностью 4x5, после чего его элементы заполняются случайными числами и выводятся на консоль:

```
// Объявляем двумерный массив
int[,] myArr = new int[4, 5];
Random ran = new Random();
// Инициализируем данный массив
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 5; j++)
    {
       myArr[i, j] = ran.Next(1, 15);
       Console.Write("{0}\t", myArr[i, j]);
    }
    Console.WriteLine();
}
```

Более подробную информацию о работе с двумерными массивами можно узнать в [4,6].

6.3. Программа работы

В зависимости от номера варианта задания написать программу [4], за исключением номера варианта 11. Для варианта с номером 11 задание следующее:

Дана целочисленная квадратная матрица. Определить:

- сумму элементов в строках, в которых отсутствуют нулевые элементы.
- минимум среди произведений диагоналей параллельных побочной диагонали матрицы.

Лабораторная работа № 7
Строки

7.1. Цель работы

Научиться работать со строками в языке C#.

7.2. Теоретические сведения

Одной из наиболее распространенных задач в современном программировании является обработка тестовой информации. Для ее решения в языке C# присутствует богатый набор средств, к которым относятся: отдельные символы, массивы символов, неизменяемые и изменяемые строки, а также регулярные выражения.

Символьный тип *char* относится к встроенным типам данных зыка C#, ему соответствует тип данных *System.Char* библиотеки классов платформы .NET. Этот тип данных предназначен для работы с символами, представленными в кодировке Unicode. Основные методы класса *System.Char* приведены в таблице 7.1.

Точно также как и любой массив в языке C# массивы символов строятся на основе базового класса *System.Array*, использование унаследованных свойств и методов которого позволяет эффективно решать некоторые задачи. Массив символов можно проинициализировать либо непосредственно, либо используя метод *String.ToCharArray()*, который разбивает исходную строку на отдельные символы.

Для работы со строками символов в кодировке Unicode предназначен встроенный тип данных языка C# *string*. Для строк определены следующие операции: присваивание (=), проверка на равенство (==), проверка на неравенство (*!*=), обращение по индексу к элементу строки (*[]*), конкатенация строк (+).

Таблица 7.1

Основные методы класса System.Char

Имя	Описание
`GetNumericValue`	Возвращает числовое значение символа, если он является цифрой и -1 в противном случае
`IsControl`	Возвращает значение `true`, если символ является управляющим
`IsDigit`	Возвращает значение `true`, если символ является десятичной цифрой
`IsLetter`	Возвращает значение `true`, если символ является буквой
`IsLetterOrDigit`	Возвращает значение `true`, если символ является буквой или цифрой
`IsNumber`	Возвращает значение `true`, если символ является числом (десятичным или шестнадцатеричным)
`Parse`	Преобразует строку в символ. Строка должна содержать только один символ
`ToLower`	Преобразует строку в нижний регистр
`ToUpper`	Преобразует строку в верхний регистр

Несмотря на то, что в языке C# строки относятся к ссылочным типам данных, проверка на равенство и неравенство строк осуществляется по их значениям. Строки равны, если они имеют одинаковую длину и состоят из одних и тех же символов.

Содержимое экземпляра класса *string* не подлежит изменению. Это означает, что однажды созданную последовательность символов изменить нельзя. Данное ограничение способствует более эффективной реализации символьных строк. Так, если требуется строка в качестве разновидности уже имеющейся строки, то для этой цели следует создать новую строку, содержащую все необходимые изменения. А поскольку неиспользуемые строковые объекты автоматически собираются в "мусор", то о дальнейшей судьбе ненужных строк можно даже не беспокоиться (они в дальнейшем будут автоматически удалены сборщиком мусора).

Следует, однако, подчеркнуть, что переменные ссылки на строки (т.е. объекты типа *string*) подлежат изменению, а следовательно, они могут ссылаться на другой объект. Но содержимое самого объекта типа *string* не меняется после его создания.

В классе *System.String* описаны большое количество полей, свойств и методов, позволяющих выполнять со строками различные действия. Некоторые из элементов класса *System.String* приведены в табл. 7.2.

Требование неизменности экземпляров класса *String* может оказаться неудобным. В этом случае для работы со строками можно применить класс StringBuilder, определенный в пространстве имен *System.Text*.

Таблица 7.2

Основные методы класса System.String

Название элемента	Тип	Описание
Concat	Статический метод	Конкатенация строк. Метод допускает конкатенацию произвольного числа строк
Compare	Статический метод	Сравнение двух строк в алфавитном порядке. Разные реализации данного перегруженного метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и т.д.
CompareOrdinal	Статический метод	Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки.
ComareTo	Метод	Сравнение текущего экземпляра строки с другой строкой.
Copy	Статический метод	Создание копии строки
Empty	Статическое поле	Поле доступное только для чтения, обозначающее пустую строку
Insert	Метод	Вставка подстроки в строку, начиная с указанной позиции
Join	Статический метод	Слияние массива строк в единую строку. Между элементами массива вставляются разделители, переданные в метод в качестве параметров.
Length	Свойство	Содержит длину строки
Remove	Метод	Удаление подстроки из строки, начиная с указанной позиции
Replace	Метод	Замена всех вхождений подстроки или символа новой подстрокой или символом
Split	Метод	Разделяет строку на элементы, которые помещаются в массив строк. В качестве разделителей, используются символы переданные в метод в качестве параметров.
Substring	Метод	Выделение подстроки в строке, начиная с указанной позиции
ToCharArray	Метод	Преобразование строки в массив символов

Более подробную информацию о работе с со строками в языке C# можно узнать в [4,6].

7.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 8
Перегрузка операций в классах

8.1. Цель работы

Научиться выполнять перегрузку операций для собственных классов в языке C#.

8.2. Теоретические сведения

Класс является полноценным типом данных, поэтому программист может определять для него собственные операции, наподобие операций, которые используются с экземплярами стандартных типов данных языка C#. Перегрузка, т.е. переопределение операций, используется и в стандартных классах платформы .NET. Так, например, знак «+» для целочисленных и вещественных типов данных означает операцию сложения, а для строковых операцию конкатенации. Язык C# позволяет переопределить большинство операций таким, образом, чтобы при их использовании совместно с экземплярами классов, созданных программистом, выполнялись бы требуемые ему действия. Это в свою очередь позволит использовать знаки операций в составе выражений с входящими в них экземплярами собственных классов, точно также как и с экземплярами стандартных классов. Пусть имеется следующий фрагмент программы:

```
MyObject m1 =new MyObject();
MyObject m2=new MyObject();
MyObject m3 =m1-m1;
```

В последнем операторе выполняется операция вычитания, определенная программистом для класса MyObject (например, может производиться вычитание значений одноименных полей экземпляров или вычитание разноименных полей). Перегрузка операций, осуществляется с помощью методов специального вида, называемых также функциями-операциями. При перегрузке операций, новые обозначения для них вводить нельзя. Для операций класса сохраняются тоже количество аргументов, приоритеты и правила ассоциации, что и для операций, используемых вместе со стандартными типами данных.

Операция должна быть описана либо как открытый статический метод (спецификатор *public static*), либо как внешняя (спецификатор *extern*). Сигнатуры всех операций, описанных в классе должны различаться. Параметры в операцию могут передаваться только по значению. Синтаксис описания операции класса в языке C# выглядит следующим образом:

```
[атрибуты] спецификаторы объявитель_операции тело
```

Объявитель операции содержит ключевое слово *operator*. Тело операции определяет действия, которые выполняются при использовании в выражении.

В языке C# возможна перегрузка трех типов операций класса: унарные, бинарные и преобразования типа. Синтаксис объявителя унарной операции выглядит следующим образом:

```
тип operator унарная_операция (параметр)
```

В классе можно определять следующие унарные операции: +, -, *!*, ~, ++, --, *true, false*. Параметр, передаваемый в операцию должен являться экземпляром класса, для которого она определяется. Унарные операции +, -, *!*,и ~ могут возвращать величину любого типа; ++ и -- экземпляр класса, для которого они определяются; а *true* и *false должны возвращать величину типа bool.* Префиксный и постфиксный инкремент (++) и декремент (--) при реализации их в собственном классе различаться не будут.

Синтаксис объявителя бинарной операции выглядит следующим образом:

```
тип operator бинарная_операция (параметр_1, параметр_2)
```

В классе можно определять следующие бинарные операции: +, -, *, /, %, |, ^, <<, >>, ==, *!*=, >, <, >=, <=. Один из параметров, передаваемых в операцию, обязательно должен являться экземпляром класса, для которого она определяется. Второй параметр может принадлежать к любому типу данных. Бинарная операция может возвращать величину любого типа данных. Операции == и *!*=, > и <, >= и <= должны перегружаться парами и обычно возвращают значение типа *bool*. Наиболее часто в классах перегружают операции == и *!*=, чтобы обеспечить сравнение их экземпляров между собой по равенству значений полей, а не по равенству ссылок. Более подробную информацию о перегрузке операций в классах можно найти в [4,6].

8.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 9
Использование наследования в ООП

9.1. Цель работы

Научится создавать и использовать при программировании принципы объектно-ориентированного программирования (ООП).

9.2. Теоретические сведения

Существуют три основных принципа ООП – инкапсуляция, наследование и полиморфизм. Классическое правило ООП утверждает, что для обеспечения надежности разрабатываемых программ нежелателен прямой доступ к полям объекта: чтение и обновление их содержимого должно происходить

через вызов соответствующих методов. Это правило и называется инкапсуляцией. В языке C# для реализации этого принципа ООП имеется соответствующая синтаксическая конструкция. В C# пользователь объекта может быть полностью отгорожен от полей при помощи свойств. Свойство обычно определяется тремя элементами: полем и двумя методами, осуществляющими его чтение и запись. Синтаксис описания свойства в языке C# выглядит следующими образом:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
    [ get код_доступа ]
    [ set код_доступа ]
}
```

Спецификатор может принимать те же значения, что и у поля, а кроме того еще такие как: `abstract`, `override`, `sealed`, `virtual` и `extern`. Чаще всего свойства объявляются со спецификатором `public`. Код доступа представляет собой блоки операторов, которые выполняются при получении (`get`) или установке (`set`) свойства. Может отсутствовать либо часть `get`, либо `set`, но не обе одновременно.

Если отсутствует часть `set`, свойство доступно только для чтения (`read-only`), если отсутствует часть `get`, свойство доступно только для записи (`write-only`). Начиная со второй версии языка C# введена возможность задавать разный уровень доступа для частей `get` и `set`. Например, во многих классах возникает необходимость обеспечить неограниченный доступ для чтения и ограниченный для записи. Спецификаторы доступа для отдельной части должны задавать либо такой же, либо более ограниченный доступ, чем спецификатор доступа для свойства в целом. Например, если свойство описано как `public` , то его части могут иметь любой спецификатор доступа, а если свойство имеет видимость *protected internal*, то его части могут объявляться имеющими область видимости *protected internal, internal, protected* и *private.* Таким образом, синтаксис описания свойства, начиная со второй версии C# выглядит следующим образом:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
    [ атрибуты ] [ спецификаторы ] [ get код_доступа ]
    [ атрибуты ] [ спецификаторы ] [ set код_доступа ]
}
```

Пример описания свойства:

```
class MyObject
    {
        private double x;
```

```
        public double X
        {
            get
            {
                return x;
            }
            set
            {
                if (value>=0) x=value;
            }
         }
    }
```

В данном примере доступ к закрытому полю x экземпляра класса MyObject осуществляется через вызовы методов, прописанных в частях get и set. Однако в явном вызове этих методов нет необходимости достаточно в программе написать, что-то наподобие:

```
double y=34, z=5; MyObject Obj1=new MyObject();
z=Obj1.X; Obj1.X=y;
```

и компилятор сам оттранслирует эти синтаксические конструкции в вызовы соответствующих методов. Т.е. внешне свойство выглядит в точности как поле, но за каждым обращением к нему может скрываться выполнение требуемых программисту действий.

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других функций, зависящих от вносимых изменений. Если потребности в специальных функциях чтения и записи нет, то можно вместо имен методов применять просто имена полей.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод *get* должен содержать оператор *return,* возвращающий выражение того же типа, что и тип свойства, либо типа данных, для которого должно существовать неявное преобразование к типу свойства. В методе *set* используется параметр со стандартным значением *value*, которое содержит устанавливаемое значение. В отличие от открытых полей, свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом и, таким образом, упрощают внесение изменений в класс.

Вторым принципом ООП является наследование. Суть этого простого принципа заключается в следующем – если в программе требуется создать новый класс, лишь незначительно отличающийся от уже существующего, то совершенно нет необходимости заново переписывать все его свойства поля и методы. Достаточно в программе написать конструкцию вида:

Class NewObject: OldObject;

Эта конструкция означает, что вновь создаваемый класс *NewObject* является классом-наследником (потомком, дочерним, производным и т.п.) старого класса *OldObject*, который в этом случае называется классом-предком (родительским, базовым и т.п.). Далее во вновь создаваемый класс следует добавить описание новых полей, свойств и методов, отсутствующих в родительском классе, т.е. произвести переход от общего к частному. В языке C# в отличие от языка C++ не допускается множественное наследование, когда часть свойств потомок наследует от одного класса, а часть от других. В C# у класса может быть только один класс-предок и произвольное количество потомков. Класс описывается при помощи следующей синтаксической конструкции:

```
[атрибуты] [спецификаторы] class имя_класса  [:предки]
                        тело класса
```

Слово предки присутствует во множественном числе, потому что класс наряду с единственным предком может наследовать еще и от интерфейсов – специального вида классов, не имеющих реализации. Поскольку все классы в C# являются потомками класса *object*, то если вновь создаваемый класс порождается непосредственно от него, то в описании нового класса его можно не упоминать. Например, следующие две синтаксические конструкции описывают одно и то же:

```
Class NewObject: object {}
Class NewObject {}
```

В классах-потомках можно добавлять описание новых элементов или переопределять унаследованные от родительских классов, но нельзя удалять такие элементы. Конструкторы не наследуются, поэтому класс-потомок должен иметь собственные конструкторы.

Поля, свойства и обычные методы наследуются. Если имеет место совпадение имен методов и полей, то в этом случае говорят, что они перекрываются. Обычные методы и поля ведут себя при наследовании в классах-потомках одинаково: можно без ограничений перекрывать их старые имена. При этом код нового обычного метода полностью перекрывает (т.е. заменяет собой) код старого метода.

```
using System;
namespace ConsoleApplication1
{
    class FirstObject : Object
    {
        public FirstObject(){}
        public double i;
        public void SetData(double AValue)
```

```
        { i = AValue;}
    }
    class SecondObject : FirstObject
    {
        public SecondObject(){}
        new public int i;
        public void SetData(int AValue)
        { i = 1; base.SetData(4.0);}
    }
    class Program
    {
        static void Main()
        {
            //FirstObject Obj1 = new SecondObject();
            SecondObject Obj1 = new SecondObject();
            //Obj1.i = 34.67;                    //1
            Console.WriteLine(Obj1.i);
            Obj1.i = 12;                         //2
            Console.WriteLine(Obj1.i);
            Obj1.SetData(1.2);                   //3
            Console.WriteLine(Obj1.i);
            Obj1.SetData(10);                    //4
            Console.WriteLine(Obj1.i);
        }
    }
}
```

В этом примере два разных метода с одинаковым именем SetData присваивают значение разным полям с именем *i*. Перекрытое (одноименное) поле предка недоступно в потомке; поэтому два одноименных поля *i* приведены только для примера. Поэтому если удалить комментарий в строке (1), то будет получено сообщение об ошибке, так как в классе *SecondObject* поле *i* было описано как целое. Если закомментировать объявление поля *i* целочисленного типа в классе *SecondObject*, то в строке 1 не будет наблюдаться сообщения об ошибке, поскольку в этом случае поле i типа *double* будет унаследовано им от класса *FirstObject*. В строке (2) целочисленному полю *i* присваивается значение. А в строке (3) вызывается метод присвоения значения полю *i SetData()* с параметром типа *double*, т.е. метод, определенный в классе *FirstObject*, который присваивает значение этому полю *i*, определенному в классе *FirstObject*. Но поскольку поле *i* перекрыто в классе *SecondObject*, то на консоль будет выведено предыдущее его значение, т.е. 12. В строке (4) вызывается метод

присвоения значения полю *i SetData()* с параметром типа *int*, т.е. метод, определенный в классе *SecondObject*, который присваивает значение этому полю *i*, определенному в классе *SecondObject*. В отличие от поля, внутри других методов, перекрытый обычный метод доступен при использовании зарезервированного слова *base*. Внутри метода *SetData()* с целочисленным параметром осуществляется присвоение значения 1 целочисленному полю *i*, описанному в классе *SecondObject* и при помощи зарезервированного слова *base* вызывается метод родительского класса, который присваивает значение параметра полю *i* родительского класса *FirstObject*. Но поскольку поле *i* с типом *double* перекрыто в классе *SecondObject*, то на консоль будет выведено значение 1.

Поскольку поля, свойства и обычные методы класса наследуются, то при желании заменить элемент базового класса новым элементом (перекрыть этот элемент) следует явным образом указать компилятору свое намерение с помощью ключевого слова *new*. Если этого не сделать, то при компиляции будет получено предупреждение.

Более подробную информацию о принципах ООП можно узнать в [4, 6].

9.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 10
Структуры

10.1. Цель работы

Научится работать со структурами в языке C#.

10.2. Теоретические сведения

Структура - это частный случай класса. Исторически структуры используются в языках программирования раньше классов. В языках PL/1, C, Pascal они представляли собой только совокупность данных (полей класса), но не включали ни методов, ни событий. В языке C++ возможности структур были существенно расширены, и они стали настоящими классами, хотя и с некоторыми ограничениями. В языке C# сохранен именно такой подход к структурам. Структура – тип данных, аналогичный классу, но имеющий ряд важных отличий от него, которые заключаются в следующем:

1) Структура является значимым, а не ссылочным типом данных, т.е. экземпляр структуры хранит значение своих элементов, а не ссылки на них.

2) Структура не может участвовать в иерархиях наследования, она может реализовывать только интерфейсы.

3) В структуре запрещено определять конструктор по молчанию, поскольку он определен неявно и присваивает всем ее элементам значения по умолчанию (нули соответствующего типа).

4) В структуре нельзя описать деструктор.

Эти отличительные черты и определяют основную область использования структур – типы данных, содержащие незначительное количество полей. Поскольку накладные расходы на динамическое выделение памяти, при небольших ее объемах, могут весьма существенно снизить производительность программы, то гораздо рациональней описывать такие объекты как структура, а не как классы. Синтаксис структуры на языке C# выглядит следующим образом:

```
[атрибуты] [спецификаторы] struct имя_струкутры [:интерфейсы] тело_структуры[;]
```

В качестве спецификаторов при описании структур можно указывать только следующие значения: *public*, *internal* и *private*. В тело структуры могут входить: константы, поля, методы, свойства, события, индексаторы, операции, конструкторы и вложенные типы данных.

В следующем фрагменте кода, описывается структура, служащая для представления радиус-вектора на плоскости:

```
Struct RadiusVector
{
     public double R, Fi;
     public RadiusVector(double r, double fi)
     {
         R=r;Fi=fi;
     }
     public double ConvertToDecartX()
     {
         return R*Math.Cos(Fi);
     }
     public double ConvertToDecartY()
     {
         return R*Math.Sin(Fi);
     }
 }
```

Также в этой структуре описаны два метода, обеспечивающие преобразование полярных координат в прямоугольные декартовые.

Присваивание структур обладает значимой семантикой, т.е. при выполнении операции присваивания одного экземпляра структуры другому происходит создание копии значения его полей. То же самое верно и при передаче

экземпляров структур в качестве фактических параметров в методы. Наибольшей эффективности можно достичь, используя массивы структур вместо массивов классов. Например, для массива из 1000 экземпляров класса в языке C# требуется создать 1001, а для массива структур всего один объект.

Более подробную информацию об использовании структур при программировании на языке C# можно узнать в [4, 6].

10.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа №11
Интерфейсы и параметризованные коллекции

11.1. Цель работы

Научиться реализовывать стандартные интерфейсы платформы .NET в собственных классах, а также работать с параметризированными коллекциями.

11.2. Теоретические сведения

Особая роль параметризированных коллекций состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. В англоязычной литературе подобные классы обозначаются термином *generic*. В настоящее время установившегося перевода данного термина на русский язык не существует. Наиболее часто в качестве перевода термина *generic* предлагаются параметризированные классы и классы-прототипы. Однако, наряду с данным переводом термина *generic* встречаются такие, как: универсальные классы, родовые классы, обобщенные классы, шаблоны, обобщения.

С помощью классов-протоипов можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

Если в программе применяются классы-прототипы, то для улучшения читабельности программы, желательно их именовать таким образом, чтобы можно было по их названию отличить от обычных типов данных. Рекомендации по именованию обобщенных типов:

1) Имена обобщенных типов должны начинаться с буквы *T*.

2) Если класс-прототип может быть заменен любым классом, так как нет никаких специальных требований и используется только класс-прототип, то *T* — вполне подходящее имя для обобщенного типа, например:

```
public class List<T> { }
public class LinkedList<T> { }
```

Если к классу-прототипу предъявляются специальные требования (например, что тип должен реализовывать интерфейс или наследоваться от определенного класса), либо же используется более одного обобщенного типа в качестве параметров, то типам следует давать осмысленные имена, например:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) ;
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class SortedList<TKey, TValue> { }
```

Синтаксис объявления класса-прототипа в языке C# выглядит следующим образом:

```
class имя_класса<список_параметров_типа>
тело_класса
```

Синтаксис объявления ссылки на класс-прототип:

```
имя_класса<список_аргументов_типа> имя_переменной =
new имя_класса<список_параметров_типа>
(список_аргументов_конструктора);
```

В библиотеке классов платформы .NET описано множество стандартных интерфейсов, которые позволяют задать требуемое поведение объектов. Например, реализация интерфейса *IComparable* в классе позволит сравнивать его экземпляры на больше-меньше, что в свою очередь даст возможность выполнять их сортировку. Реализация интерфейсов *IEnumerable* и *IEnumerator* позволяет осуществить перебор содержимого экземпляра класса. Многие стандартные классы платформы .NET реализуют стандартные интерфейсы. Так, например, перебор содержимого массива с помощью цикла *foreach* возможен лишь потому, что в классе *System.Array* реализованы интерфейсы *IEnumerable* и *IEnumerator*. Можно реализовывать стандартные интерфейсы и в собственных типах данных. Это даст возможность использовать экземпляры таких классов в составе выражений, точно также как и экземпляры стандартных типов. Более подробную информацию о реализации стандартных интерфейсов в собственных классах и применения классов-прототипов при программировании можно найти в [4,6].

11.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

Лабораторная работа № 12
Создание приложений Windows Forms

12.1. Цель работы

Научиться создавать Windows-приложения с GUI (Graphic User Interface – графический интерфейс пользователя).

12.2. Теоретические сведения

Несмотря на то, что начиная с третьей версии платформы .NET появилась альтернатива создания приложений с графическим интерфейсом в виде WPF (Windows Presentation Foundation), в эксплуатации находится достаточно большое количество приложений с графическим интерфейсом Windows Forms.

Среда Visual Studio содержит удобные средства разработки Windows-приложений, выполняющие вместо программиста рутинную работу – создание шаблонов приложений и форм, заготовок обработчиков событий, организацию цикла обработки сообщений и т.д. При создании нового проекта **Visual C#** и выбора шаблона **Windows Application** среда Visual Studio сформирует готовый шаблон Windows-приложения. В этом шаблоне имеется вкладка заготовки формы **Form1.cs[Design]**, которая располагается в основной части экрана. Форма представляет собой окно и предназначена для размещения компонентов (элементов управления) – меню, текста, кнопок, списков, изображений и т.д. Среда создает не только заготовку формы, но и шаблон текста приложения. Перейти к нему можно, щелкнув в окне **Solution Explorer** (**View► Solution Explorer**) правой кнопкой мыши на файле **Form1.cs** и выбрав в контекстном меню команду **View Code**. При этом откроется вкладка с кодом формы, листинг которого имеет примерно следующий вид:

Листинг 12.1 Шаблон Windows-приложения

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace WindowsApplication4
{
    public partial class Form1 : Form
    {
```

```
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Приложение начинается с директив использования пространств имен библиотеки .NET. Для пустой формы, не содержащей ни одного компонента, необходимыми являются только две директивы:

```
using System;
using System.Windows.Forms;
```

Пространство имен *System.Windows.Forms* содержит элементы графического интерфейса пользователя, такие как формы, кнопки и т.д. Также в листинге программы содержится описание части класса *Form1*, порождаемого от класса *Form*:

```
public partial class Form1 : Form
```

Для улучшения читабельности программ, начиная с версии Visual Studio 2005 (второй версии языка Visual C#), введена возможность разбивать описание типа на части и хранить их в разных физических файлах, создавая так называемые частичные типы (partial types). Для описания отдельной части класса используется модификатор `partial`. Он может применяться к классам, структурам и интерфейсам, например:

```
public partial class A
{
 …
}
public partial class A
{
 …
}
```

После совместной компиляции этих двух частей получается такой же код, как если бы класс был описан обычным образом. Все части одного и того же частичного типа должны компилироваться одновременно, иными словами добавление новой части к уже скомпилированной не допускается. Модификатор `partial` не является ключевым словом и должен стоять непосредственно перед одним из ключевых слов `class`, `struct` или `interface` в каждой из частей. Все части определения одного класса должны быть описаны в одном и том же пространстве имен. Если модификтор `partial` указывается для типа, описание которого состоит только из одной части, то это не является ошибкой.

Поскольку, как следует из листинга 12.1, класс *Form1* описан как частичный тип, то для получения полного исходного описания этого класса, находясь на вкладке **Form1.Designer.cs**, либо **Form1.cs*** (рис. 12.1) следует выбрать из левого раскрывающегося списка либо пункт **components**, либо **Dispose (bool disposing)**, либо **InitializeComponent()**. При выборе одного из этих пунктов откроется текст, представленный в листинге 12.2 (комментарии из него удалены).

Листинг 12.2. Шаблон Windows-приложения **(**продолжение)

```
namespace WindowsApplication4
{
    partial class Form1
    {
  private System.ComponentModel.IContainer
   components = null;
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.components = new
                 System.ComponentModel.Container();
            this.AutoScaleMode =

System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "Form1";
        }
        #endregion
    }
}
```

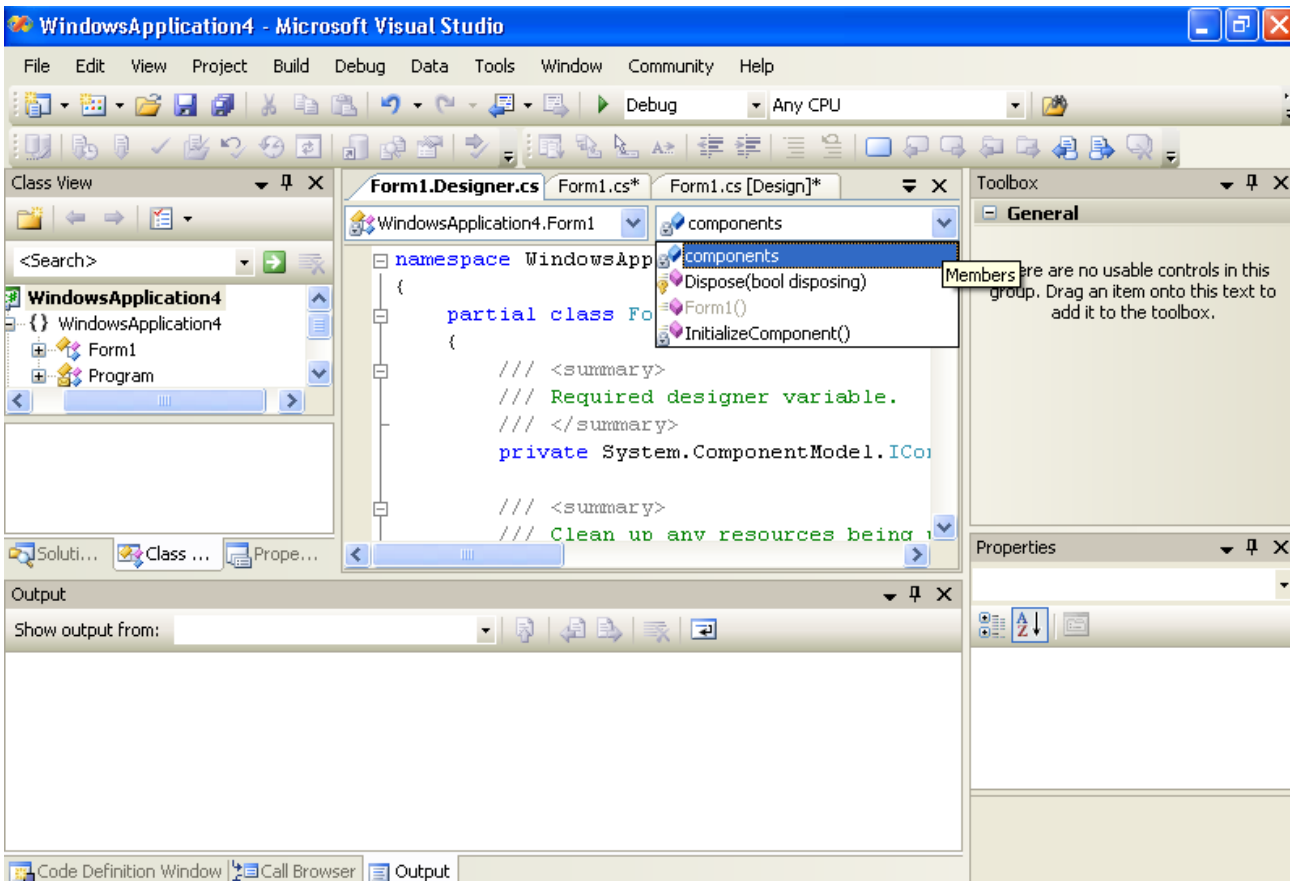

Рис. 12.1. Окно, используемое для доступа к различным частям класса

Класс *Form1* наследует от своего предка множество элементов, а в нем самом, как видно из листинга 12.2, описано новое закрытое поле *components* – контейнер для хранения компонентов, которые можно добавить в класс формы. Поскольку во вновь созданной форме компонентов не имеется, то значение этого поля равно *null*. Конструктор формы вызывает закрытый метод *InitializeComponen()t*, автоматически формируемый средой (код этого метода скрыт между директивами препроцессора *#region* и *#endregion*). Этот метод обновляется средой при добавлении элементов управления на форму, а также при изменении свойств формы и свойств содержащихся на ней элементов управления. Например, если изменить цвет фона формы с помощью окна свойств (**Properties**), в методе появится примерно такая строка:

this.BackColor=System.Drawing.SystemColors.ControlDarkDark;

Метод освобождения ресурсов `Dispose` вызывается автоматически при закрытии формы.

Процесс создания Windows-приложения состоит из двух основных этапов:

1) Визуальное проектирование, т.е. задание внешнего облика

приложения.

2) Определение поведения приложения путем написания процедур обработки событий.

На этапе визуального проектирования на форму помещаются компоненты (элементы управления) и задаются их свойства и свойства самой формы при помощи окна свойств (рис. 12.2). Если его не видно, то можно воспользоваться командой меню **View ►Properties Window**. Свойства отображаются либо в алфавитном порядке, либо сгруппированными по категориям. Способ отображения выбирается с помощью кнопок, расположенных в верхней части окна свойств (рис. 12.2).

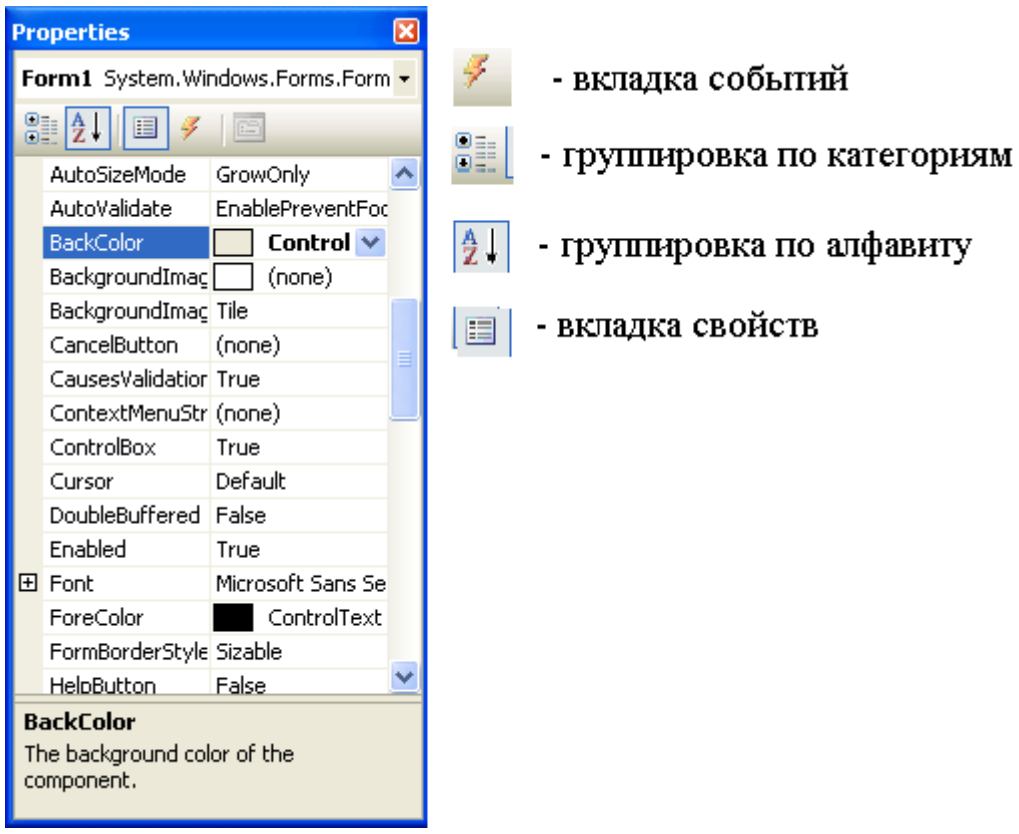

Рис. 12.2. Окно свойств

Для размещения элементов управления на форме следует воспользоваться палитрой компонентов **Toolbox**. Если ее не видно, то следует воспользоваться командой меню **View ►Toolbox**. Поместить элемент управления на форму можно, либо дважды щелкнув по нему, либо при нажатой левой кнопке мыши перетащить компонент на форму. Можно также сделать один щелчок на палитре и еще один щелчок в том месте формы, куда планируется поместить элемент управления. Задание свойств выполняется либо выбором имеющихся в списке вариантов, либо вводом требуемого значения с клавиатуры.

Определение поведения программы начинается с принятия решений, какие действия должны выполняться при щелчке на кнопках, вводе текста, выборе пунктов меню и т.д., иными словами, по каким событиям будут выполняться действия, реализующие функциональность программы. Заготовка шаблона обработчика события формируется двойным щелчком на поле, расположенном справа от имени соответствующего события на вкладке **Events** окна свойств, при этом появляется вкладка окна редактора кода с заготовкой соответствующего обработчика. Для каждого класса определяется свой набор событий, на которые он может реагировать. Наиболее часто используемые события:

1) `Activated` – получение формой фокуса ввода.
2) `Click, DoubleClick` – одинарный и двойной щелчки мышью.
3) `Closed` – закрытие формы.
4) `Load` – загрузка формы.
5) `KeyDown, KeyUp` – нажатие и отпускание любой клавиши и их сочетаний.
6) `KeyPress` – нажатие клавиши, имеющей ASCII-код.
7) `MouseDown, MouseUp` – нажатие и отпускание кнопки мыши.
8) `MouseMove` – перемещение мыши.
9) `Paint` – возникает при необходимости прорисовки формы.

Более подробные сведения о программировании приложений *Windows Forms* можно получить в [4].

12.3. Программа работы

В зависимости от номера варианта задания написать программу [4].

СПИСКОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Википедия [Электронный ресурс]. – Информатика. – Режим доступа: https://ru.wikipedia.org/wiki/%C8%ED%F4%EE%F0%EC%E0%F2%E8%EA%E0

2. Википедия [Электронный ресурс]. – TIOBE Index. – Режим доступа: https://ru.wikipedia.org/wiki/TIOBE_Index

3. CyberForum.ru - форум программистов и сисадминов [Электронный ресурс]. – Форум программистов Microsoft .NET. Программирование с использованием платформы Microsoft.NET– Режим доступа: http://www.cyberforum.ru/net-framework/

4. Павловская, Т. А. C#: Программирование на языке высокого уровня: учебник для вузов/ Т. А. Павловская. – Санкт-Петербург: Питер, 2013. – 432 с.

5. Голощапов, А. Л. Microsoft Visual Studio 2010/ А. Л. Голощапов. – Санкт-Петербург: БХВ-Петербург, 2011. – 544 с.

6. Шилдт, Г. C# 4.0: полное руководство/ Г. Шилдт. – Москва: И. Д. Вильямс, 2011. – 1056 с.

И Н Ф О Р М А Т И К А

Методические указания к лабораторным работам

Подписано в печать . Формат 60 × 84/16.
Усл. п. л. 2,25. Тираж экз. Заказ №

РИО ВоГУ. 160000, г. Вологда, ул. С. Орлова, 6.