В.В.Подбельский

Язык C#

Решение задач

В.В.Подбельский

Язык C#

Решение задач

Рекомендовано
УМО вузов РФ по образованию в области экономики,
менеджмента, логистики и бизнес-информатики,
а также УМО вузов РФ по образованию в области
прикладной математики и управления качеством
в качестве учебного пособия
для студентов высших учебных заведений,
обучающихся по направлениям подготовки
“Программная инженерия”, “Бизнес-информатика”,
“Прикладная математика”

МОСКВА
“ФИНАНСЫ И СТАТИСТИКА”
2014

УДК 004.438
ББК 32.973.2
П44

РЕЦЕНЗЕНТЫ:

Кафедра «Программное обеспечение ЭВМ и информационные технологии» (ИУ7) МГТУ им. Н.Э. Баумана;

В.А. Дударев,
канд. техн. наук, доц., старший научный сотрудник ИМЕТ РАН

П44 **Подбельский В.В.**
Язык Си#. Решение задач: учеб. пособие / В.В. Подбельский. – М.: Финансы и статистика, 2014. – 296 с.: ил.

ISBN 978-5-279-03553-3

Учебное пособие представляет собой сборник задач с решениями в виде программ на языке C# в интегрированной среде разработки Microsoft Visual Studio 2010, 2012. Решения задач снабжены подробными объяснениями всех алгоритмических и конструктивных особенностей кода. Задачи книги расположены по нарастающей сложности и позволяют читателю, начав с элементов языка C#, получить навыки программирования достаточно высокого уровня. От консольных программ читатель переходит к созданию библиотек классов и полноэкранным Windows-приложениям. В последних главах книги рассмотрено применение методологии эволюционного подхода к проектированию программ, а последняя задача посвящена разработке библиотеки классов для численного интегрирования систем дифференциальных уравнений и применению свободно распространяемой графической библиотеки ZedGraph.

Книга рассчитана на широкий круг программистов, переходящих на язык C#, на студентов, изучающих язык C# в курсах программирования, и читателей, самостоятельно изучающих язык C#.

УДК 004.438
ББК 32.973.2

ISBN 978-5-279-03553-3

ПРЕДИСЛОВИЕ

Изучать синтаксис и семантику конструкций языка нужно и полезно по учебным пособиям и учебникам, но нельзя стать хорошим программистом, не истратив достаточного количества времени на программирование и анализ образцов кодов конкретных программ.

Существуют многочисленные фундаментальные книги по языку C#. Но обычно в них подробно рассматриваются синтаксис и механизмы C# и недостаточно уделено внимания практическому применению его конструкций. Точнее сказать, примеры кода на языке C# обычно в публикациях не велики и демонстрируют только те или иные его частные возможности, обсуждаемые в конкретном разделе книги.

На практике программист сталкивается с конкретными задачами, для решения которых обычно недостаточно отдельных средств языка. Создаваемая программа часто требует применения целого набора конструкций языка и понимания разных возможностей среды программирования.

Однако нельзя начинать изучение ремесла программирования на C# с громоздкой задачи, требующей для своего решения большого количества разнообразных средств, и среды .NET, и языка C#. Несмотря на то, что такие примеры в литературе по программированию существуют (см., например, [17], где читатель должен последовательно разбирать особенность разработки информационно-справочной системы из медицинской области) для читателя удобнее учиться на небольших задачах, понимание постановки которых не требует глубокого знакомства с предметной областью, на которую они ориентированы. Здесь, конечно, существует опасность слишком упростить постановку задачи и свести её программную реализацию к примерам, которыми снабжает программиста справочная система MSDN. Поэтому в данной книге:

- задачи подбирались по нарастающей сложности;
- темы выделены в соответствии с классической схемой изучения языков программирования и методологией объектно-ориентированного подхода к разработке программ;
- по мере изложения материала постановки задач приближаются к уровню заданий на конкретные программные продукты.

Таким образом, в нашей книге мы начнём с простых задач, постепенно от темы к теме расширяя набор применяемых конструкций языка C# и инструментов платформы .NET.

К настоящему времени задачников, практикумов и решебников по программированию издано и существует в электронном виде очень много. Они написаны для разных алгоритмических языков и обычно достаточно полно представляют те виды и классы задач, которые принято решать с помощью программирования. Существуют прекрасные книги (например, Вирт Н. [7], Кубенский А.А. [15]), посвященные структурам данных и алгоритмам обработки этих структур.

Цель нашей книги – обеспечить читателя образцами не алгоритмов и не отвлечённых структур данных, а программ именно на языке C#. При этом основное внимание уделяется не собственно синтаксису языка C#, а тем особенностям конструкций языка, которые позволяют наиболее эффективно использовать его для решения тех или иных практических задач.

Предлагаемая книга начинается главой (темой 01) о среде разработки Microsoft Visual Studio. Основное внимание уделено версии MS VS 2010, но приведены сведения и о русскоязычном варианте MS VS 2012. По существу, тема 01 является введением к последующим главам, включающим содержательные по постановке и решению задачи. Назначение темы 01 – познакомить читателя с основными приёмами взаимодействия с интегрированной средой разработки MS VS при создании консольных программ на языке C#. (Созданию Windows-приложений посвящены темы 12 и 13.)

Темы 02, 03, 04 охватывают вопросы, связанные со взаимодействием программ на языке C# с библиотечными классами платформы .NET, предназначенными для работы с консольным экраном и клавиатурой.

Глава вторая (тема 02) посвящена проблеме вывода информации в консольное окно, представляемое в программе библиотечным классом Console. Рассматриваемые здесь задачи не сложны и в постановке, и в решении. Основная цель – научиться форматировать выводимые данные при создании их строковых представлений, выводимых в консольное окно.

Задачи и программы темы 03 посвящены преобразованию информации при её вводе с клавиатуры. Показано как приме-

нять методы Parse() и TryParse() базовых типов и как пользоваться возможностями класса System.Convert.

Тема 04 знакомит читателя с механизмами, позволяющими программно получать и изменять характеристики консоли в зависимости от требований решаемой задачи. Первая программа иллюстрирует особенности получения информации о нажимаемых пользователем клавишах при выполнении программы. Затем показано как получать и изменять свойства класса Console, определяющие внешний вид (размеры, цвет, заголовок) консольного окна и размеры буфера консоли. Последние две программы темы 04 посвящены управлению звуковыми сигналами, подачу которых обеспечивает метод Console.Beep().

Всякая программа на языке C# представляет собой совокупность классов (библиотечных и написанных вручную специально для решения конкретной задачи). Если при разработке кода программы программист применяет только статические члены подготовленных им классов, то C# можно воспринимать как императивный язык программирования. Именно такой подход позволяет применять при работе на языке C# процедурное программирование. Особенностям такого стиля программирования на языке C# посвящены задачи и программы тем 05–08.

Тема 05, в соответствии с заголовком, посвящена выражениям, условным операторам и циклам. Данные в программах этой темы представляют собой переменные в традиционном для алгоритмических языков понимании (*имя* + *значение*). Разные задачи предлагают обработку данных разных типов. В большинстве задач используются целочисленные и вещественные переменные, но в задаче 05-04 анализируются коды символьных данных и обращается внимание читателя на тот факт, что при обработке кодов символов зачастую нет необходимости знать их конкретные числовые значения. Программа задачи 05-03 иллюстрирует возможности тернарной (условной) операции. В программе задачи 05-02 показано как применять операцию контроля за целочисленным переполнением. В задаче 05-06 показано как выполнять итерационные вычисления с точностью до машинного нуля. В задаче 05-07 применяется библиотечный класс System.Random, позволяющий создавать случайные числовые последовательности. В программе задачи 05-08 построены вложенные циклы, позволяющие перебирать в регулярном порядке значения нескольких логических перемен-

ных. Делается это на примере построения таблицы истинности заданной логической функции.

Тема 06 посвящена таким структурам данных как массивы и переключатели. Программы в задачах 06-01 – 06-04 выполняют обработку одномерных массивов, элементы которых имеют арифметические значения. В задаче 06-05 элементы массивов имеют символьные значения. В задаче 06-06 реализуется косвенная индексация элементов массива, позволяющая получать доступ ко всем элементам массива в запланированном порядке, отличном от последовательного перебора. Традиционные для языка C# массивы создаются с неизменяемым числом элементов. В задаче 06-07 показано, как увеличить одномерный массив в процессе выполнения программы, а в программе из задачи 06-08 изменяются (уменьшаются) размеры двумерного массива. Задачи 06-09 и 06-13 знакомят читателя с применением методов классов System.string и System.Array. В программах задач 06-10 и 06-11 используются переключатели. В задаче 06-10 реализован ввод значений именованных данных. При таком вводе во входной строке (набираемой пользователем) указывается и имя переменной, и её значение. Задача 06-11 достаточно традиционная – программа формирует строку с шестнадцатеричным изображением заданного целого числа. В задаче 06-12 показано, как создавать и использовать непрямоугольный массив, т.е. массив ссылок на одномерные массивы с символьными элементами.

Тема 07 включает задачи, предусматривающие разработку и использование статических методов. В первой задаче метод с параметрами, передаваемыми по значениям, возвращает их наибольший общий делитель. Во второй задаче параметры методов передаются по ссылкам, что позволяет изменять значения внешних переменных, ссылки на которые применимы в качестве аргументов. Программа иллюстрирует перегрузку методов и применение в теле одного метода обращений к другому. Задача 07-03 вычисляет с точностью до машинного нуля значение функции sin(x). Особенность программы – применение в ней двух методов, первый из которых формирует массив разложения в ряд функции sin(1), а второй, принимая в качестве исходных данных ссылку на этот массив, вычисляет значение sin(x). Для построения треугольника Паскаля в задаче 07-04 создан метод формирования непрямоугольного массива, элементами которого служат биноминальные коэффициенты. В задаче

07-05 создан метод, позволяющий читать из входного потока (с клавиатуры) вещественные числа, в которых дробная часть отделена, по желанию пользователя, от целой либо точкой, либо запятой. В задаче 07-06 разработан рекурсивный метод формирования массива с заранее неизвестным числом элементов. В задаче 07-07 для упорядочения строк матрицы, представленной в виде массива ссылок на одномерные массивы, применяется библиотечный метод Array.Sort(). Для указания правила упорядочения создан вспомогательный метод сравнения одномерных массивов, ссылка на который используется в качестве аргумента метода Array.Sort().

Тема 08 знакомит читателя с особенностями разработки библиотек классов. В первой задаче в библиотеку помещается класс со статическими методами для работы с массивами. Далее в задачах 08-02, 08-03, 08-05, 08-06 класс дополняется методами для создания, ввода, вывода одномерных и двумерных массивов с вещественными аргументами. Особняком стоит задача 08-04. В ней на основе метода из задачи 07-05 реализован метод чтения из строки числового значения типа **double**. Задача 08-07 посвящена разработке методов для представления элементов массивов в виде строк.

Программы, в которых используются классы, обычно решают достаточно сложные задачи. В очень хороших книгах по объектно-ориентированным языкам программирования часто механизмы языка и методы объектно-ориентированного программирования (ООП) излагаются на основе рассмотрения одной большой задачи, для решения которой создается и последовательно реализуется средствами языка достаточно сложный проект. Например, рассматривается информационная система управления автотранспортного предприятия или информационные потоки в больнице. Такой подход знакомит читателя с реальными задачами, но зачастую затрудняет изучение возможностей языка, так как заставляет читателя для понимания смысла задачи разбираться в незнакомой ему предметной области.

В нашей книге в отличие от указанного подхода для объяснения техники ООП выбраны короткие задачи, иллюстрирующие применение конкретных механизмов и средств языка. В ряде случаев мы даже отказались от «смысловой» части формулировки задачи, ограничившись требованием применить

в программе указанным способом конкретные средства языка. Например, «определить класс с полями таких-то типов, объявить и инициализировать массив со ссылками на объекты этого класса, а затем упорядочить этот массив в соответствии со значениями одного из свойств объектов класса».

Исходя из сказанного об ООП, можно в этом предисловии при описании задач Тем 09, 10, 11 ограничиться несколькими словами. В ряде случаев для разработанных классов приведены их графические UML-представления. В задаче 09-03 показано, как создавать массивы объектов пользовательских классов, как использовать ссылку на объект в качестве параметра метода. В задаче 09-04 ссылка на объект класса используется как статическое поле того же класса и иллюстрируются возможности статических конструкторов. В задаче 09-05 параметром метода одного из взаимно независимых классов служит ссылка на объект второго класса.

Тема 10 посвящена перегрузке операций для объектов определяемых программистом классов: «активное электрическое сопротивление», «полином», «матрица», «правильная дробь», «вектор многомерного пространства».

В Теме 11 рассматриваются особенности наследования классов, возможности абстрактных классов и виртуальных членов. В задаче 11-04 на примере интерфейса структуры данных «стек» показано, как реализовать интерфейс с помощью разрабатываемых программистом классов.

В Теме 12 осуществлён переход от консольных приложений к программам с визуальным полноэкранным интерфейсом пользователя. В начале на простых задачах показано, как пользоваться редактором форм (графическим дизайнером), позволяющим выполнять визуальное проектирование пользовательского интерфейса. Затем объясняется механизм управления программой с помощью событий и иллюстрируются особенности создания обработчиков событий. Так как количество элементов управления и компонентов, которые доступны разработчику при создании Windows-приложений, весьма велико, то описать и применить все их никак невозможно в нашей книге. Гораздо важнее показать на примерах, что каждый компонент и каждый элемент управления – это объект некоторого класса. Эти объекты можно «настраивать», задавая начальные значения их свойств, и можно изменять в процессе выполнения програм-

мы. Кроме того, обращено внимание на тот факт, что основное окно Windows-программы (форма) представляет собой отображение объекта особого класса (по умолчанию имеющего имя Form1), код которого модифицирует программист при разработке Windows-приложения. Останавливаясь на особенностях некоторых задач этой темы, следует обратить внимание на задачу 12-06, в которой используется очень важный и функционально богатый элемент управления **DataGridView**. В задаче 12-07 разработана программа, иллюстрирующая особенности простейшей анимации с помощью графических средств и таймера. В задаче 12-08 программа строит график функции плотности вероятностей биноминального распределения. Значения этой функции вычисляются в отдельном классе с итератором. Для размещения графика и элементов управления на форме в программе используется контейнер **TableLayoutPanel**. Рисование графика выполняется в контексте элемента **PictureBox**.

Тема 13 включает два больших примера, с помощью которых читатель знакомится с методикой эволюционного проектирования программ и некоторыми возможностями свободно распространяемой библиотеки ZedGraph. Применение методики эволюционного проектирования позволяет создавать программу последовательными шагами, каждый из которых увеличивает функциональность программы. На каждом шаге (или этапе разработки) программист принимает некоторое проектное решение, реализует его в коде, а затем компилирует и выполняет программу, проверяя её соответствие принятому проектному решению. При этом важно, чтобы на каждом шаге принималось минимальное количество проектных решений, и что при завершении шага программа работоспособна. Такая схема разработки очень хорошо поддерживается возможностями, представляемыми программисту интегрированной средой разработки, и может быть рекомендована при создании программ средней сложности с помощью MS VS. В задаче 13-02 создаётся библиотека классов для численного интегрирования систем дифференциальных уравнений методом Рунге-Кутта. Для её тестирования разрабатывается программа, выполняющая интегрирование системы дифференциальных уравнений 2-го порядка, для которой известно аналитическое решение. Графики полученных численного и аналитического решений выводятся на экран и сравниваются. Для вывода графиков применяется библиотека

ZedGraph. Для вывода графиков можно было бы написать программу целиком вручную, как это сделано в задаче 12-08. Но цель состояла в том, чтобы обратить внимание читателя на существование общедоступных библиотек классов C# и показать особенности их применения.

Для решения задач в этой книге использовалась Microsoft Visual Studio 2010 Professional Edition и русскоязычная версия Microsoft Visual Studio 2012.

Книга рассчитана на программистов, имеющих некоторый опыт программирования (не обязательно на языке C#) и знакомых с основными сведениями о синтаксисе языка C#. Предварительное знакомство с языком C# может быть получено, например, из пособия автора «Язык C#. Базовый курс».

Изучение кодов приведённых в книге программ будет достаточно для освоения профессионального программирования на языке C#. Для дальнейшего совершенствования мастерства станут более понятны и доступны многочисленные справочные материалы и документация по языку C#, среде Microsoft Visual Studio и платформе Microsoft NET Framework.

Автор благодарен рецензентам: доценту, к.т.н. В.А. Дудареву, доценту, к.т.н. В.А. Крищенко и заведующему кафедрой «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана, к.т.н. И.В. Рудакову.

ТЕМА 01

СРЕДА РАЗРАБОТКИ И КОНСОЛЬНЫЕ ПРИЛОЖЕНИЯ

При изучении языка программирования нужно уметь работать с соответствующим компилятором и средствами выполнения готовых программ. Удобнее всего кодировать, компилировать, отлаживать и выполнять программы на языке C#, применяя интегрированную среду разработки (ИСР), например, Microsoft Visual Studio (MS VS). Поэтому первая тема книги содержит описание основных приёмов работы программиста с ИСР MS VS. Возможности этой интегрированной среды разработки настолько широки и разнообразны, что детальное знакомство с ними зачастую оказывается слишком трудоемким даже для подготовленного читателя. Мы начнем с достаточно простых сведений. В настоящее время кроме англоязычных версий ИСР MS VS доступны их локализованные (например, русскоязычные) аналоги. Будем ориентироваться на англоязычную версию MS VS 2010 и русскоязычную версию MS VS 2012.

Начнем со сведений об англоязычной версии MS VS 2010. При входе в интегрированную среду разработки, открывается ее окно со стартовой страницей **StartPage** и, возможно, с окном **SolutionExplorer** («Обозреватель решения»). В заголовке основного окна (рис. 1.1) размещено главное меню с пунктами **File**, **Edit**, **View**, **Debug**, **Team**, **Data**, **Tools**, **Test**, **Window**, **Help**. Нам понадобятся только некоторые пункты главного меню, некоторые элементы стартовой страницы и окно проводника решений.

Изучая программирование на C#, будем разрабатывать программы трех видов:

- консольные приложения;
- библиотеки классов;
- приложения на основе Windows Form.

Независимо от того, какого вида программа разрабатывается, в Visual Studio необходимо создать решение (Solution) и в

Рис. 1.1. Стартовая страница Microsoft Visual Studio 2010

этом решении проект (Project). Остановимся на этих двух понятиях.

Проект – это «контейнер» исходных кодов вашей программы и результата их трансляции в исполнимый код. Эти коды размещаются в физически существующих файлах проекта. Кроме того, в проект входят файлы с описаниями свойств (**Properties**) проекта и ссылок (**References**) на другие проекты.

Решение – это средство для объединения нескольких проектов, хотя в решении может быть только один проект. Решение позволяет определять свойства, которые будут отнесены к нескольким проектам (или к одному проекту, если в решении всего один проект). Сами по себе решения не выполняют никаких действий, они лишь «контейнеры» для проектов. Очень важно понимать, что Visual Studio имеет дело только с решениями. Проект доступен и «понятен» для среды разработки, когда он находится в конкретном решении и это решение загружено в Visual Studio. Сразу же отметим, что в каждый момент вре-

мени Visual Studio может работать только с одним решением, и если необходимо на компьютере одновременно использовать несколько решений, то для каждого из них нужно запустить отдельный экземпляр Visual Studio.

Итак, текст (код) программы может быть обработан средой Visual Studio, когда он помещен в проект, а проект – включен в решение. В одно решение могут одновременно входить проекты разных программ. Зачастую в одно решение помещают взаимосвязанные проекты, например, использующие одни и те же библиотеки классов (возможно помещенные также в виде проектов в это решение).

Из этих самых общих сведений о проектах и решениях ясно, что для создания и выполнения даже самой простой программы в Visual Studio необходимо создать проект (Project) с кодом программы, и поместить этот проект в решение, которое должно быть загружено в среду разработки.

Создание пустого (без проектов) решения чаще всего не имеет смысла, поэтому решение будет автоматически создано при создании нового проекта. Но прежде чем описать последовательность действий, необходимых для этого, заметим, что с проектами и решениями удобно работать, когда они каким либо образом систематизированы. В нашем случае будем помещать решения в один каталог, который можно создать с помощью стандартных средств операционной системы. Этот каталог можно разместить на своем рабочем столе либо на любом подходящем для вас диске. Для конкретности предположим, что для наших решений на диске C:\ создан каталог с именем «C#_Практикум».

Как только среда разработки (в нашем случае Visual Studio 2010) запущена, выполните следующие шаги.

Создание нового проекта

Это возможно двумя способами. Во-первых, на стартовой странице (см. рис. 1.1) можно выбрать (активировать мышью) пункт New Project.

Второй вариант – применить возможности главного меню:

File -> New -> Project (или сочетание клавиш: **Ctrl+Shift+N**).

В обоих случаях открывается окно **New Project** (рис. 1.2). В этом окне на левой панели Recent Templates выберите язык **Visual C#**. На центральной панели выберите вид приложения **Console Application** (консольное приложение).

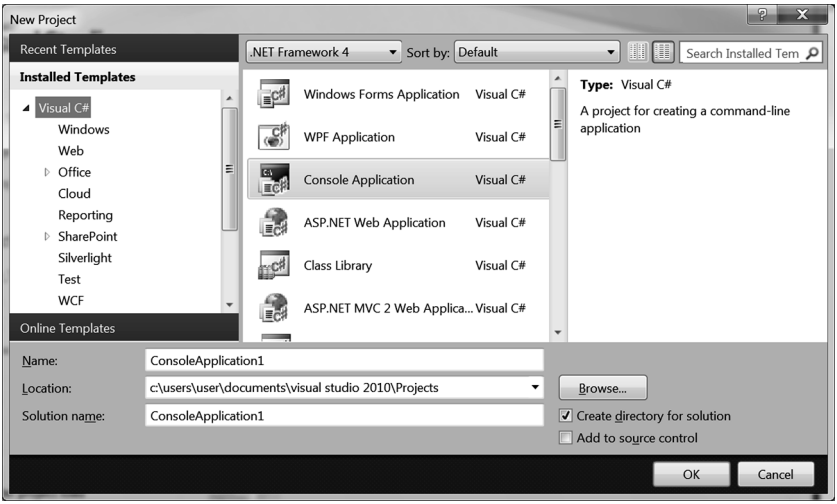


Рис. 1.2. Окно выбора нового проекта (поля заполнены по умолчанию)

В поле **Name** (см. рис. 1.2 и рис. 1.3) вместо предлагаемого по умолчанию имени ConsoleApplication1 напечатайте выбранное вами имя проекта, например, Program_1. В поле **Location** введите полное имя папки (каталога), в которой будет сохранено решение, например, C:\C#_Практикум.

Для решения запланируйте создание папки решения, поставив «галочку» в окошке **Create directory for solution.** По умолчанию и этой папке и одновременно решению (Solution) приписывается имя его первого проекта. В нашем примере на рис. 1.2 в поле **Solution name** для решения предлагается название ConsoleApplication1. Как только мы заменим имя проекта в поле

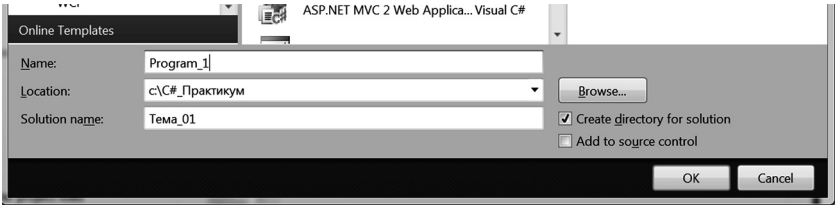


Рис. 1.3. Переименования для проекта Program_1 в решении Тема_01, размещаемом в каталоге C:\C#_Практикум

name, в этом поле **Solution name** появится имя Program_1. Чтобы не возникало путаницы (одинаковым именем названы разные вещи – и проект и решение), выберите и внесите в поле **Solution name** другое имя решения, например, Тема_01. Названные изменения показаны на рис. 1.3.

Кнопкой **OK** запускаем процесс создания проекта (и решения и папки). Среда Visual Studio создает папку C:\C#_Практикум\Тема_01 для решения, собственно решение, проект приложения и открывает окно редактора кода с шаблоном (с заготовкой) для текста консольной программы на языке C# (см. рис 1.4). Если по неясной для вас причине окно обозревателя решения не появится – откройте его через главное меню: **View -> Solution Explorer** (или сочетанием клавиш **Ctrl+W,S**).

Чтобы увидеть в окне редактора текст кода программы (если его нет), щелкните в окне Solution Explorer пункт Program.cs. Или, выделив пункт Program.cs, нажмите клавишу **F7**.

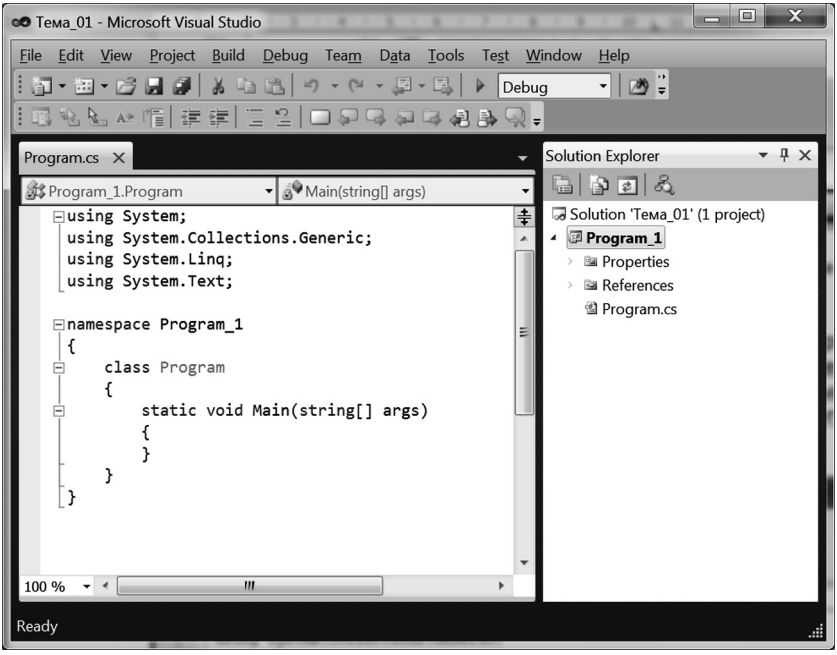


Рис. 1.4. Окно текстового редактора и обозревателя решения

Не анализируя предлагаемый средой разработки текст из файла Program.cs, показанный в окне редактора кода, отметим, что, несмотря на отсутствие функциональности, это код работоспособной программы. То есть ее можно транслировать и в результате трансляции получить исполнимый модуль приложения, которое выполняется, но ничего не делает. Прежде чем наполнять эту заготовку функциональностью, проверим ее корректность.

Компиляция и исполнение

Чтобы откомпилировать и запустить на исполнение код приложения, размещенный в окне текстового редактора, используйте возможности главного меню редактора кода:

Debug -> Start Without Debugging

(или сочетание клавиш **Ctrl+F5**).

В результате выполнения этой команды откроется консольное окно (см. рис. 1.5) с единственной фразой: «Для продолжения нажмите любую клавишу...». Это сообщение среды разработки, завершающее исполнение консольного приложения. Само приложение ничего «не сообщает» пользователю.

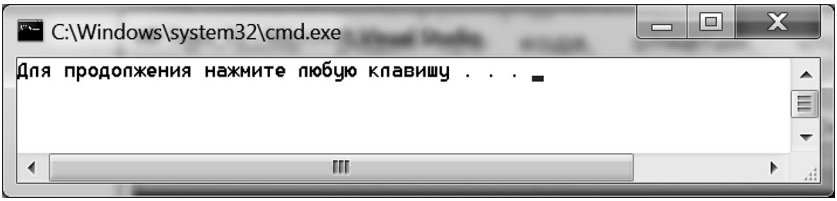


Рис. 1.5. Консольное окно с результатом выполнения пустого приложения

Дополним созданную средой разработки заготовку (прототип, шаблон) кода консольного приложения одним оператором для вывода сообщения. Получим, например, такой код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Program_1
{
  class Program
```

```
  {
    static void Main(string[] args)
{
      Console.WriteLine("Это первая программа!");
}
  }
}
```

Console.WriteLine – обращение к методу WriteLine() класса Console, принадлежащего пространству имен System. Назначение метода – вывод строки-аргумента в консольное окно.

По нажатию клавиш Ctrl+F5 программа откомпилируется, выполнится, и в консольное окно будут выведены две фразы:

```
Это первая программа!
Для продолжения нажмите любую клавишу...
```

Убедившись, что нам доступны и понятны элементарные действия, необходимые для создания консольных приложений, разберем текст кода программы. Большая часть кода сформирована без явного участия программиста. Программист добавил единственный исполняемый оператор:

```
Console.WriteLine("Это первая программа!");
```

Этот оператор размещен в теле метода с фиксированным заголовком:

```
Static void Main(string[ ] args)
```

Метод Main() принадлежит классу **class** Program { }. Класс с предложенным средой разработки, но не обязательным, именем Program помещен в пространство имен **namespace** Program_1 { }.

Название пространства имен Program_1 предложено средой разработки. Оно совпадает с именем проекта, но это имя не является обязательным – программист может его заменить, как и имя класса.

Перед пространством имен Program_1 размещены операторы **using**, обеспечивающие доступ к нескольким пространствам имен, которым принадлежат классы стандартной библиотеки Visual Studio, точнее, платформы .NET Framework. Назначение оператора **using** – сделать доступным для программы то пространство имен, имя которого указано в этом операторе.

Так как наше приложение и программы, изучаемые в первых темах, очень просты и требуют применения только базовых средств стандартной библиотеки классов, то в тексте заготовка программы, автоматически созданной средой Visual Studio, присутствуют лишние для нашей программы элементы. Что же является избыточным?

Во-первых, вместо четырех операторов **using** можно обойтись одним

using System;

Во-вторых, нет необходимости в явном указании параметра **string**[] args в заголовке метода Main( ). Этот заголовок можно записать так:

static void Main()

Конструкция **string**[] args никак не используется в нашем приложении и может быть удалена. Назначение этой конструкции будет рассмотрено в последующих программах, где мы ее при необходимости восстановим в заголовке метода Main( ).

Третья особенность заготовки – наличие объявления пространства имен самой программы:

namespace Program_1...{ }

Это объявление вводит для программы ее собственное пространство имен с обозначением Program_1. Для простых программ, с которых мы начинаем изучение программирования на языке C#, этот оператор **namespace** и скобки{ }, в которые заключен весь последующий текст, могут быть опущены. В этом случае программа использует стандартное пространство имен. Это вполне допустимо для тех программ, которые мы будем приводить.

Результат названных упрощений в виде кода работоспособной консольной программы может быть таким:

```
// 01_01.cs – файл с первой программой, проект Program_1
using System;
class Program {
    static void Main( ) {
    Console.WriteLine("Это первая программа!");
    }
}
```

В первой строке кода – комментарий. При обработке программы компилятором эта строка будет отброшена (не учитывается).

Задача 1-2 – Программа должна «спросить» у пользователя его имя и «поздороваться» с ним по имени.

Будем разрабатывать программу как еще один проект Program_2 уже созданного нами решения с названием Тема_01. Для этого откроем Visual Studio (рис. 1.1) и среди недавних проектов (Recent Projects) выберем решение Тема_01. Как только в Visual Studio загружено решение, откроем окно обозревателя решения (Solution Explorer – см. рис. 1.6).

View -> Solution Explorer

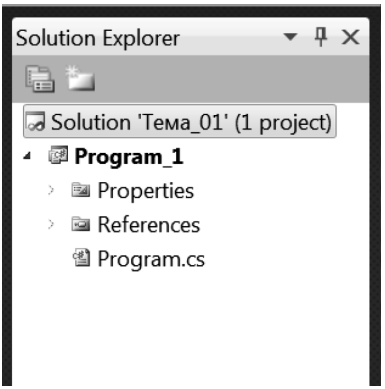


Рис. 1.6. Окно решения Тема_01

В этом окне (рис. 1.6) щелкнем правой кнопкой мыши на имени решения. (В нашем случае на строке с именем **Solution 'Тема_1'**.) В выпадающем меню выберем:

Add -> New Project

Откроется окно нового проекта, которое отличается от окна, показанного на рис. 1.2 и рис. 1.3 отсутствием поля **Solution name** и окошка **Create directory for solution**. Это естественное отличие, так как решение уже присутствует в Visual Studio. Для нового проекта укажем имя (например) «Program_2». Каталог (Location) изменять не будем. После нажатия кнопки OK в окне редактора кода проекта Program_2 отобразится заготовка для кода новой программы, а окно обозревателя решения Тема_01 примет вид, показанный на рис. 1.7.

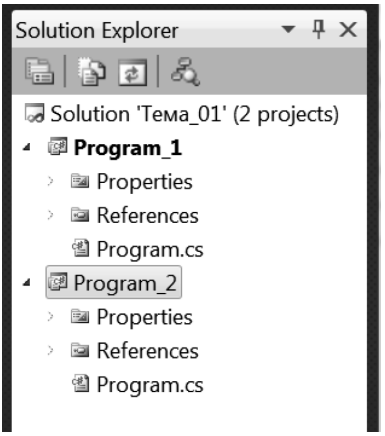


Рис. 1.7. Решение с двумя проектами

Заменим весь текст заготовки (то есть файла Program.cs) проекта Program_2 следующим кодом:

```
// 01_02.cs – Программа с диалогом, проект Program_2
using System;
class Program {
  static void Main( ) {
    string name;
    Console.WriteLine("Введите Ваше имя: ");
    name = System.Console.ReadLine();
    Console.WriteLine("Приветствую Вас, " + name + "!");
  }
}
```

В отличие от прототипа, предложенного средой разработки, в этой программе только один оператор **using**, отсутствует объявление пространства имен (нет **namespace**), у метода Main() нет параметра.

В коде оператор **string** name; определил переменную name строкового типа. Оператор

```
Console.WriteLine("Введите Ваше имя: ");
```

уже знаком – в этой программе он выведет в консольное окно приглашение, то есть строку "Введите Ваше имя: ".

Рассмотрим оператор

```
name =Console.ReadLine();
```

Справа от знака операции присваивания «=» находится обращение к методу ReadLine() класса Console, определенного в пространстве имен System. Метод ReadLine() считывает из входного консольного потока (от клавиатуры) строку символов. Операция присваивания связывает переменную с именем name со строкой, полученной с помощью метода ReadLine().

Рассмотрим следующий оператор:

```
Console.WriteLine("Приветствую Вас, " + name + "!");
```

Новое в нем – конкатенация трех строк в аргументе метода. Для обозначения операции конкатенации применяется знак +. Эту роль знак + играет только в том случае, если левый операнд – строка. (При использовании с арифметическими операндами знак +, как принято в математике, обозначает операцию сложения.)

После подготовки текста программы проекта Program_2 нужно сделать этот проект «стартовым» для решения Тема_01. Если этого не делать, то код второго проекта можно компилировать, но результат компиляции не будет запущен на исполнение из среды Visual Studio.

Дело в том, что в решении в каждый момент стартовым является только один проект. Если ничего не менять, то стартовым проектом решения является тот первый проект, при котором решение создано. В нашем случае стартовым остается проект Program_1. Если запустить компиляцию и выполнение:

Debug -> Start Without Debugging

(или использовать сочетание клавиш Ctrl+F5) – откроется консольное окно с фразами первого проекта:

```
Первая программа
Для продолжения нажмите любую клавишу. . .
```

Чтобы заменить стартовый проект решения, открываем окно обозревателя решения (если оно закрыто):

View -> Solution Explorer

В этом окне щелкнем правой кнопкой мыши на имени проекта, который хотим сделать стартовым. (В нашем случае на строке с именем ‘Program_2’.) В выпадающем меню выберем

Set As StartUp Project

После этого проект Program_2 станет стартовым для данного решения. Теперь по нажатию клавиш Ctrl+F5 программа откомпилируется, начнет выполнение, выведет приглашение: «Введите Ваше имя: » и в ответ на введенное имя – «поздоровается». Пример результатов выполнения:

```
Введите Ваше имя:
Арсений<ENTER>
Приветствую Вас, Арсений!
Для продолжения нажмите любую клавишу . . .
```

В приведенных результатах выполнения программы присутствует конструкция <ENTER>, которой нет в консольном окне. С ее помощью в тексте книги будем обозначать нажатие клавиши Enter, выполненное пользователем при диалоге с программой. Отметим, что только после нажатия клавиши Enter выполняется ввод данных и метод ReadLine() «прочитывает» в виде строки информацию, набранную на клавиатуре.

Файлы решения и его проектов

Чтобы познакомиться со структурой решения и составом его проектов, откройте вне Visual Studio папку «C:\C#_Практикум», в которую мы поместили решение, создавая его первый проект. В папку «C#_Практикум» входит каталог следующего уровня с именем решения «Тема_01». В свою очередь в папку этого решения входят два каталога проектов «Program_1», «Program_2» и файл Тема_01.sln:

```
[C:\C#_Практикум]
  [Тема_01]
    [Program_1]
    [Program_2]
    Тема_01.sln
```

Если вне Visual Studio активировать файл Тема_01.sln, дважды нажав на его иконке левую кнопку мыши, то автоматически запускается новый экземпляр Visual Studio с загруженным в среду разработки решением Тема_01. Такой способ запуска Visual Studio довольно удобен и часто применяется на практике для работы с уже существующими решениями.

Теперь обратим внимание на структуру каталогов проектов [Program_1] и [Program_2]. Так как в нашем примере они идентичны, то приведем состав одного из каталогов:

[Program_1]
[bin]
[obj]
[Properties]
Program.cs
Program_1.csproj

На этом уровне знакомства с интегрированной средой разработки MS VS наиболее интересен для практического применения текстовый файл Program.cs. В нем находится код написанной нами программы на языке C#. Этот код виден в окне редактора кодов Visual Studio, но его можно посматривать и редактировать обычными текстовыми редакторами (например, блокнотом).

Далее полезно открыть каталог [bin] и вложенный в него каталог [Debug]:

[Program_1]
[bin]
[Debug]
Program_1.exe
Program_1.vshost.exe
Program_1.vshost.exe.manifest
Program_1.pdb

Здесь интересен файл Program_1.exe с исполнимым кодом проекта. Остальные файлы этого каталога и папки верхних уровней нам пока не понадобятся.

Таким образом, чтобы найти файл с исполнимым модулем нужной программы, следует открыть папку решения, в ней – каталог проекта, в нем – каталог [bin] и затем каталог [Debug]. Например, в папке [Debug], папки [bin], папки [Program_2], папки [Тема_01] размещен файл Program_2.exe. Файл Program_2.exe содержит готовый к исполнению код проекта Program_2, который реализует функциональность соответствующего (второго) приложения. Такой код подготовлен Visual Studio на промежуточном языке и выполняется средствами среды исполнения, реализованными в Microsoft .NET Framework. До сего времени мы запускали этот файл из Visual Studio, но он может быть выполнен вне интегрированной среды разработки Visual Studio. (На компьютере обязательно должна присутствовать соответствующая версия платформы .NET Framework.)

Теперь рассмотрим особенности выполнения исполнимого модуля приложения вне Visual Studio.

Запустить исполнимый модуль на выполнение можно разными способами. Если открыто консольное окно, то в рабочей строке наберите полный путь к exe-файлу:

```
C:\C#_Практикум\Тема_01\Program_2\bin\debug\Program_2.exe
```

По нажатию клавиши Enter программа начнет диалог.

Запуск исполнимого модуля можно выполнить щелчком клавиши мыши на иконке модуля Program_2.exe. В этом случае консольное окно открывается только на время диалога. Зачастую это очень неудобно – программа принимает информацию от пользователя, и консольное окно моментально «захлопывается» после вывода результатов работы. В случае проекта Program_2 программа выведет строку «Введите Ваше имя: ». Как только пользователь после набора имени нажмет клавишу Enter, приветствие мелькнет на экране и консольное окно закроется. Это в большинстве случаев неудобно. Хотелось бы посмотреть на результаты! Простейшее решение состоит в добавлении в конец метода Main() следующих двух операторов:

```
Console.WriteLine("Для выхода из программы нажмите ENTER.");
Console.ReadLine();
```

Для реализации этого замысла создадим в том же решении Тема_01 еще один проект Program_3. Поместим в файл Program_3.cs этого проекта следующий код:

```
// 01_03.cs - Программа с диалогом, проект Program_3
using System;
class Program {
static void Main( ) {
    string name;
    System.Console.WriteLine("Введите Ваше имя: ");
    name = Console.ReadLine();
    Console.WriteLine("Приветствую Вас, " + name + "!");
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Сделаем проект Program_3 стартовым для решения с названием Тема_01. Выполнив в Visual Studio компиляцию и ис-

полнения программы этого проекта получим, например, такие результаты выполнения программы:

Введите Ваше имя:
Юрий<ENTER>
Приветствую Вас, Юрий!
Для выхода из программы нажмите ENTER.
<ENTER>
Для продолжения нажмите любую клавишу . . .

Исполнимый модуль этого проекта удобно выполнять, запуская его и вне Visual Studio. Автоматически закрывать консольное окно программа «не позволит».

Немного о русскоязычной версии MS VS 2012

Начальная страница MS VS 2012 показана на рис. 1.8. Как видите, почти все названия приведены на русском языке. Нас пока интересуют только возможности создания, компиляции и отладки программ на языке C#.

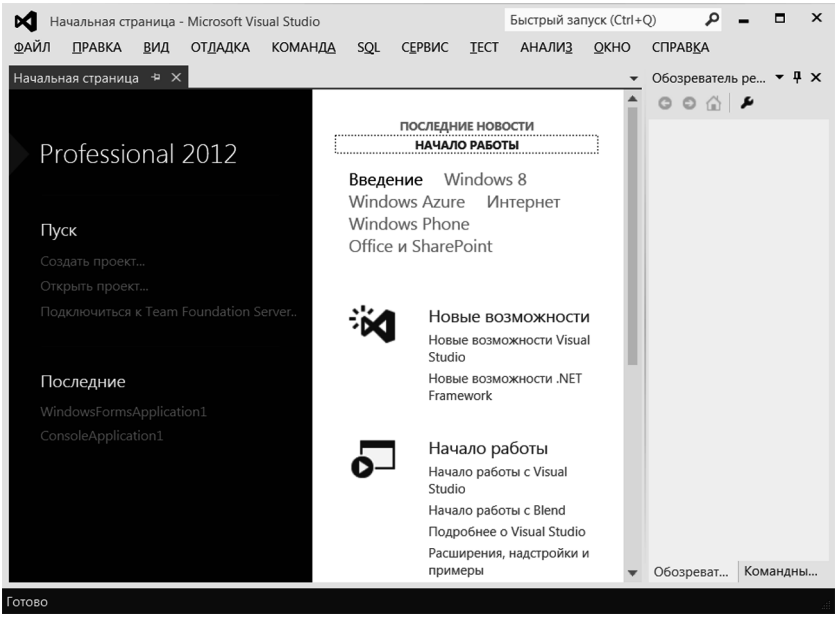


Рис. 1.8. Начальная страница русскоязычной версии MS VS 2012

Создание нового проекта в MS VS 2012

Это возможно несколькими способами. Во-первых, на стартовой странице (см. рис. 1.8) можно выбрать (активировать мышью) пункт **Создать проект**.

Второй вариант – применить возможности главного меню:

Файл -> Создать -> Проект (или сочетание клавиш: **Ctrl+Shift+N**).

В обоих случаях открывается окно **Создать проект** (рис. 1.9). В этом окне на левой панели **Установленные** среди **Шаблонов** выберите язык Visual C#. На центральной панели выберите вид приложения Консольное приложение.

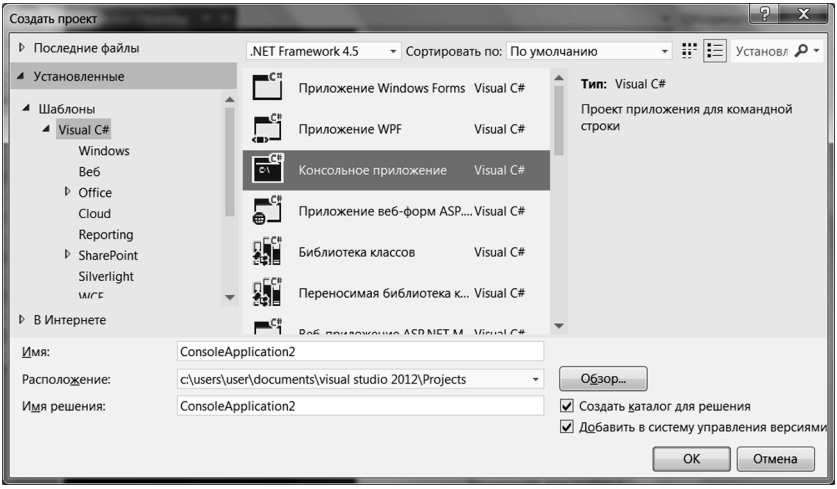


Рис. 1.9. Окно выбора вида проекта

В поле **Имя** (см. рис. 1.9) вместо предлагаемого по умолчанию имени ConsoleApplication1 напечатайте выбранное вами имя проекта, например Program_4. В поле **Расположение** введите полное имя папки (каталога), в которой будет сохранено решение, например, C:\C#_Практикум. В поле **Имя решения** укажите имя решения, например «Тема_01» (и имя папки для него). Если проект должен быть помещен в уже существующее решение, то путь к этому решению нужно указать в поле **Расположение** и снять выделение пункта **Создать каталог для решения**. Этот вариант показан на рис. 1.10.

Рис. 1.10. Проект поместить в существующее решение

Результат выполнения этого пункта – код заготовки программы в окне редактора (см. рис. 1.11).

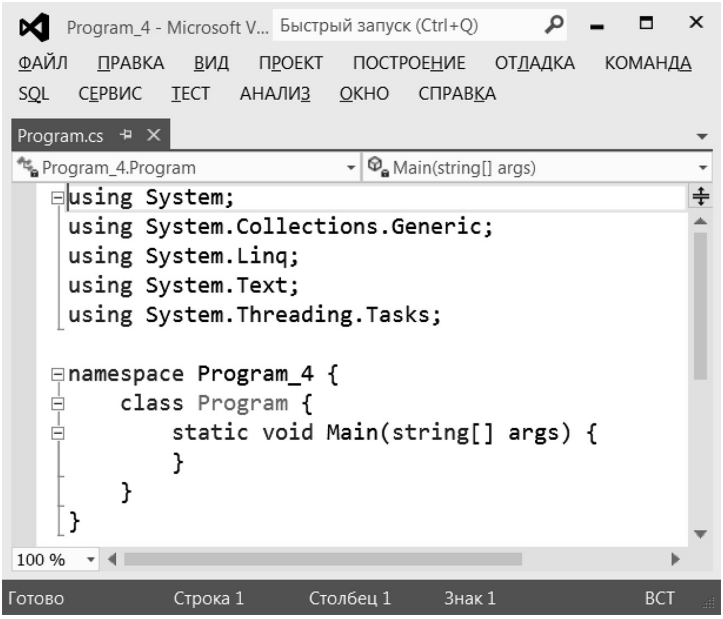


Рис. 1.11. Код прототипа программы в окне редактора кода

Окно обозревателя решения можно открыть через главное меню:

ВИД ->Обозреватель решений (или сочетанием клавиш **Ctrl + W,S**).

Чтобы увидеть в окне редактора текст кода программы (если его там нет), щелкните в окне **Обозреватель решений** пункт Program.cs, или, выделив его, нажмите клавишу **F7.**

Компиляция и исполнение

Чтобы откомпилировать и запустить на исполнение код приложения, размещенный в окне текстового редактора, используйте возможности главного меню редактора кода:

Отладка -> Запуск без отладки (или сочетание клавиш **Ctrl+F5**).

ТЕМА 02

КОНСОЛЬНЫЙ ВЫВОД

Для ввода и для вывода информации (данных) консольные программы используют последовательности символов, при вводе – набираемые пользователем на клавиатуре, при выводе – воспроизводимые на консольном экране. Для программиста важно, как эти символьные последовательности представлены в программе. Тут допустимы разные варианты. Нам сейчас удобно представлять в программах и выводимую, и читаемую последовательности символов в виде строк. При этом выводимая информация в программе может быть представлена либо литеральной строковой константой, либо значением переменной типа **string**, либо выражением, формирующим строку. С некоторыми возможностями вывода текстовой информации мы уже познакомились в предыдущих программах, где для этих целей применялся метод Console.WriteLine( ). Сейчас на примерах рассмотрим особенности вывода значений базовых типов и возможности управляющих символов.

Задача 02-01. Используя в строках управляющие символы (эскейп-последовательности) выведите в виде таблицы из двух колонок суффиксы и названия типов арифметических констант: **int, long, uint, ulong, double, float, decimal**.

```
// 02_01.cs – суффиксы арифметических констант
using System;
class Program {
static void Main( ) {
    string line = "<пусто>\t- int\t\t<пусто>\t- double";
    Console.WriteLine(line);
    string line1 = "L, l\t- long\t\tF, f\t- float ";
    Console.WriteLine(line1);
    string line2 = "U, u\t- uint\t\tM, m\t- decimal ";
    Console.WriteLine(line2);
    string line3 = "UL, ul\t- ulong\tD, d\t- double";
    Console.WriteLine(line3);
```

```
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
<пусто> - int      <пусто> - double
L, l    - long     F, f    - float
U, u    - uint     M, m    - decimal
UL, ul  - ulong    D, d    - double
Для выхода из программы нажмите ENTER.
<ENTER>
```

В приведённой программе использован метод Console. WriteLine() с аргументом типа **string**. Если обратиться к документации, то увидим, что имеется 20 вариантов метода Console. WriteLine( ), отличающихся спецификациями параметров. Метод автоматически формирует строковое представление информации, представленной его аргументами. Кроме того, при конкатенации строки с операндом другого типа выполняется автоматическое преобразование значения этого (не строкового) операнда в его строковое представление. Воспользуемся этой возможностью в следующей программе, которая выводит изображения и значения логических и арифметических выражений.

Задача 02-02. Вывести с помощью метода Console.WriteLine( ) изображения и значения нескольких арифметических и логических выражений.

```
// 02_02.cs – изображения и значения выражений
using System;
class Program {
static void Main( ) {
  Console.WriteLine("5 > 8 = "+(5 > 8));                    // bool
  Console.WriteLine("5+3 == 8 = "+(5+3 == 8));              // bool
  Console.WriteLine("8/5 = "+(8/5));                        // int
  Console.WriteLine("8.0*5.0 = "+(8.0*5.0));                // double
  Console.WriteLine("5.0/3.0 = "+(5.0/3.0));                // double
  Console.WriteLine("Для выхода из программы нажмите ENTER.");
  Console.ReadLine();
  }
}
```

Результаты выполнения программы:

```
5 > 8 = False
5+3 == 8 = True
8/5 = 1
8.0*5.0 = 40
5.0/3.0 = 1,66666666666667
Для выхода из программы нажмите ENTER.
<ENTER>
```

Анализируя результаты, обратим внимание, что при выводе значений логические величины представлены их системными обозначениями (т.е. слова **True**, **False** пишутся с большой буквы). Значение 8/5 при целочисленном делении округлено до меньшего целого.

Подробнее рассмотрим последний результат – выведенное значение 1,66666666666667 выражения 5.0/3.0. Во-первых, значение выражения 5.0/3.0 есть число иррациональное и его нельзя представить конечной десятичной дробью. В программе на языке C# вещественные числа имеют тип **double** и могут быть представлены только с конечной точностью. Программа выводит все значащие цифры представления числа этого типа. Во вторых, в записи числа для отделения дробной части числа от целой части использована запятая. Это соответствует европейскому (немецкому, французскому, русскому и др.) формату записи вещественных чисел. Но в тексте программ используется английский или американский формат записи чисел – не запятая, а точка отделяет целую часть от дробной. Это внешне незначительное отличие может приводить к неприятным последствиям при использовании готовых программ (как при просмотре результатов, так и при вводе с клавиатуры вещественных чисел). Национальный формат (культуру) можно устанавливать в настройках операционной системы Windows. В среде .NET, где выполняются программы, написанные на C#, тоже есть возможности учитывать и изменять тот национальный формат среды (потока), в которой выполняется программа. Покажем, как это сделать, на следующей задаче.

Задача 02-03. Вывести в европейском, американском, и вновь в европейском форматах значение числа π. Число π доступно как константное поле класса Math (см. Приложение 2).

```
// 02_03.cs – форматы записи вещественного числа
using System;                    // Для ConsoleиMath
using System.Threading;  // Для Thread
using System.Globalization;  // Для CultureInfo
class Program {
static void Main( ) {
    double pi = Math.PI;
    Console.WriteLine(pi);  // Европейский по настройке Windows
    Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
    Console.WriteLine(pi);  // Американский
    Thread.CurrentThread.CurrentCulture = new CultureInfo("ru-Ru");
    Console.WriteLine(pi);  // Европейский
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результаты выполнения программы:

```
3,14159265358979
3.14159265358979
3,14159265358979
Для выхода из программы нажмите ENTER.
<ENTER>
```

Для доступа к средствам .NET, позволяющим «переключать» национальные форматы представления вещественных чисел (и форматы представления дат), в программу включены операторы

```
using System.Threading;
using System.Globalization;
```

Их присутствие обеспечивает доступ к средствам классов Thread (Поток) и CultureInfo (Информация о культуре). Переход к формату другой культуры (в нашем примере к американской, а затем к российской) выполняет оператор:

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
```

В качестве аргумента в обращении к конструктору CultureInfo( ) используется строковая константа, которую в документации называют Language Name. В программе применены две такие константы: “ru-Ru” – европейский формат и “en-US” – американский формат.

Так как и значение 5.0/3.0, и число π есть числа иррациональные, то при их выводе по умолчанию выводятся все значащие цифры, которые может представить тип **double**. Такое количество значащих цифр в изображении чисел не всегда нужно на практике. Существует несколько способов влиять на символьное представление арифметических значений. Используем метод ToString( ), унаследованный всеми типами языка C# от общего базового класса **object**. Применяя метод ToString( ) к арифметическим данным, можно с помощью аргумента задавать формат их символьного представления. В приложении 1 приведена таблица спецификаторов формата числовых типов. В следующей программе показано, как можно их применять.

Задача 02-04. Определить переменную типа **double**, инициализируя ее тем приближенным значением числа π, которое предоставляет класс System.Math. Вывести строковые представления значения переменной и результатов ее умножения и деления на 10. Применить в методе ToString( ) строки формата “0.000” и “0.000E0”.

```
// 02_04.cs - форматирование изображений вещественных чисел
using System;
class Program {
static void Main( ) {
    double x = Math.PI;
    Console.WriteLine("Выражение\tФормат\t\tЗначение");
    Console.WriteLine("x\t\t0.000\t\t"+
                      x.ToString("0.000"));
    Console.WriteLine("x*10\t\t0.000\t\t"+
                      (x*10).ToString("0.000"));
    Console.WriteLine("x/10\t\t0.000\t\t"+
                      (x/10).ToString("0.000"));
    Console.WriteLine("x\t\t0.000E0\t\t"+
                      x.ToString("0.000E0"));
    Console.WriteLine("x*10\t\t0.000E0\t\t"+
                      (x*10).ToString("0.000E0"));
    Console.WriteLine("x/10\t\t0.000E0\t\t"+
                    (x/10).ToString("0.000E0"));
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
Выражение   Формат      Значение
X           0.000       3,142
x*10        0.000       31,416
x/10        0.000       0,314
x           0.000E0     3,142E0
x*10        0.000E0     3,142E1
x/10        0.000E0     3,142E-1
Для выхода из программы нажмите ENTER.
<ENTER>
```

Программа иллюстрирует возможности трех так называемых «пользовательских» спецификаторов формата. Спецификатор «точка» определяет положение десятичной запятой. Спецификатор 0 «отводит» позицию для размещения любой цифры, в том числе и незначащего нуля. Спецификатор E предусматривает применение при выводе экспонентной (научной) нотации.

Формирование изображений числовых значений при получении их строковых представлений возможно не только с помощью метода ToString(). Это можно делать с помощью строки форматирования, например, в методах Write() и WriteLine().

Задача 02-05. Используя строку форматирования в качестве первого аргумента метода WriteLine(), вывести целую и дробную части вещественного числа 1234,4321. Не использовать методы класса Math.

```
// 02_05.cs – форматирование при выводе чисел
using System;
class Program {
static void Main( ) {
    double real = 1234.4321;      // Вещественное число
    int integer = (int)real;      // Округление до целого
    double fraction = real - integer;      // Дробная часть числа
    Console.WriteLine("Целая часть: {0:D4}, дробная часть: {1:F4}",
                integer, fraction);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
Целая часть: 1234, дробная часть: 0,4321
Для выхода из программы нажмите ENTER.
<ENTER>
```

Значение переменной real инициализировано – задано вещественным числом в формате, предусмотренном синтаксисом языка C# (дробная часть отделена от целой не запятой, а точкой). Переменным integer и fraction присвоены значения целой и дробной частей числа real. Для округления вещественного значения переменной real до целого значения используется операция приведения типов (**int**). В строке форматирования метода WriteLine( ) два поля подстановок. Первое {0:D4} – определяет вид символьного представления целочисленного аргумента integer, второе {1:F4} – относится к вещественному аргументу fraction.

Задача 02-06. Используя строку форматирования в качестве аргумента метода ToString() и символ табуляции, вывести на консольный экран таблицу:

Выражение	*Формат*	***Изображение***
(5.0/3.0)	F	1,67
(5.0/3.0)	F4	1,6667
(5.0/3.0)	E	1,666667E+000
(5.0/3.0)	E5	1,66667E+000
(5.0/3.0)	G	1,66666666666667
(5.0/3.0)	G3	1,67
(5.0/3e10)	G3	1,67E-10
(5.0/3e-10)	G	16666666666,6667
(5.0/3e20)	G	1,66666666666667E-20

Следующая программа решает задачу.

```
// 02_06.cs – таблица спецификаторов формата
using System;
class Program    {
static void Main( )      {
string top = "|*Выражение*|*Формат*|" +
             "  *** Изображение ***   |";
Console.WriteLine(top);
   string line = "|----------|--------|" +
            "-------------------------|";
```

```
    Console.WriteLine(line);
    string str = "| (5.0/3.0) |  F    | " +
              (5.0/3.0).ToString("F")+"\t\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3.0) |  F4   | " +
              (5.0/3.0).ToString("F4")+"\t\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3.0) |  E    | " +
              (5.0/3.0).ToString("E")+"\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3.0) |  E5   | " +
              (5.0/3.0).ToString("E5")+"\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3.0) |  G    | " +
(5.0/3.0).ToString("G")+"\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3.0) |  G3   | " +
              (5.0/3.0).ToString("G3")+"\t\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3e10)|  G3   | " +
              (5.0/3e10).ToString("G3")+"\t\t\t|";
    Console.WriteLine(str);
    str = "|(5.0/3e-10)|  G    | " +
              (5.0 / 3e-10).ToString("G")+"\t\t|";
    Console.WriteLine(str);
    str = "| (5.0/3e20)|  G    | " +
              (5.0 / 3e10).ToString("G")+"\t\t|";
    Console.WriteLine(str);
    line = "*-----------*--------*" +
           "-------------------------*";
    Console.WriteLine(line);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Таблица и код программы наглядно демонстрируют основные принципы применения строк форматирования непосредственно в качестве аргументов метода ToString().

Задача 02-07. Используя строку форматирования в качестве аргумента метода WriteLine(), выведите коды русских и латинских букв A, C, E, предварительно присвоив символьным переменным соответствующие значения.

```
// 02_07.cs – коды русских и латинских букв
using System;
class Program    {
static void Main( )     {
    char rusA = 'A', rusC = 'C', rusE = 'E',
            latA = 'A', latC = 'C', latE = 'E';
    string format=
    "{2} символ: {0},\tкод10: {1:d4},\tкод16: 0x{1:x4}";
    Console.WriteLine(format, rusA, (int)rusA, "Русский");
    Console.WriteLine(format, latA, (int)latA, "Латинский");
    Console.WriteLine(format, rusC, (int)rusC, "Русский");
    Console.WriteLine(format, latC, (int)latC, "Латинский");
    Console.WriteLine(format, rusE, (int)rusE, "Русский");
    Console.WriteLine(format, latE, (int)latE, "Латинский");
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Результат выполнения программы:

```
Русский символ: A,      код10: 1040,    код16: 0x0410
Латинский символ: A,    код10: 0065,    код16: 0x0041
Русский символ: C,      код10: 1057,    код16: 0x0421
Латинский символ: C,    код10: 0067,    код16: 0x0043
Русский символ: E,      код10: 1045,    код16: 0x0415
Латинский символ: E,    код10: 0069,    код16: 0x0045
Для выхода из программы нажмите ENTER.
<ENTER>
```

В обращениях к методу Console.WriteLine( ) первый аргумент – строка форматирования

```
string format="{2} символ: {0},\tкод10: {1:d4},\tкод16: 0x{1:x4}";
```

В этой строке заранее запланированы размещения и внешний вид значений трех последующих аргументов. Обратите внимание, что в строке format четыре поля подстановок. Нумерация внутри полей не соответствует последовательности размещения аргументов в обращении к методу. Значения символа (он имеет тип char) и строки с указанием языка (константа типа string) выводятся без преобразований и поэтому поля {2} и {0} не содержат спецификаторов. Поля {1:d4} и {1:x4} предназначены для одного и того же аргумента, который в списке находится на втором месте (после аргумента format).

ТЕМА 03

КОНСОЛЬНЫЙ ВВОД

Для ввода с клавиатуры используют методы ReadLine(), Read(), ReadKey(). Это статические методы класса Console. Первый из них считывает строку, набранную на клавиатуре, второй – код отдельного символа, третий – код нажатой клавиши.

Начнем с возможностей, предоставляемых методом ReadLine(). Прочитать строку символов, набранных пользователем на клавиатуре, мы умеем – в первой теме приведена программа, читающая имя пользователя. Непосредственно числовое значение с помощью метода ReadLine() с клавиатуры прочитать невозможно. Вначале считывается строка – символьное изображение, набранное на клавиатуре, а затем его преобразуют в числовое значение нужного типа. Остановимся на второй части этой процедуры, так как чтение строки нам уже понятно.

Получить числовое или логическое значение по его строковому представлению можно с помощью методов Parse(), TryParse(), входящих во все базовые типы (**int**, **double**, **long**, ...) , и методов класса System.Convert.

Задача 03-01. Получить из строковых представлений значения разных типов.

Программа, решающая задачу методами Parse() из классов базовых типов:

```
// 03_01.cs – методы Parse()
using System;
class Program {
static void Main( ) {
    string bLine = “true”;  // представление логического значения
    bool bValue = bool.Parse(bLine);
    Console.WriteLine(“bValue = “ + bValue);
    string iLine = “2010”;  // представление целого числа
    int iValue = int.Parse(iLine);
    Console.WriteLine(“iValue = “ + iValue.ToString(“d6”));
    string dLine = “22,44”; // представление вещественного числа
```

```
    double dValue = double.Parse(dLine);
    Console.WriteLine("dValue = " + dValue.ToString("E"));
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результаты выполнения программы:

```
bValue = True
iValue = 002010
dValue = 2,244000E+001
Для выхода из программы нажмите ENTER.
<ENTER>
```

Напомним еще раз, что при выводе логического значения используется его системное обозначение, т.е. True вместо **true**. Обратите внимание, что применение метода Parse() требует «знания» того типа, значение которого представлено в символьной строке. Имя этого типа используется в качестве префикса в обращении к методу. Например, с помощью метода int.Parse() выполняется «чтение» целочисленного значения и строковое представление этого значения не должно содержать ошибок. Если в нашей программе использовать обращения **int**.Parse(bLine) или **int**.Parse(dLine), то это приведёт к аварийной ситуации на этапе выполнения программы. (Компилятор не выявит этой ошибки.)

Решение задачи методами класса Convert:

```
// 03_01_1.cs – методы класса Convert
using System;
class Program {
static void Main( ) {
string bLine = "false";  // представление логического значения
bool bValue = Convert.ToBoolean(bLine);
  Console.WriteLine("bValue = " + bValue);
  string iLine = "2010";  // представление целого числа
  int iValue = Convert.ToInt32(iLine);
  Console.WriteLine("iValue = " + iValue.ToString("d6"));
  string dLine = "22,44"; // представление вещественного числа
  double dValue = Convert.ToDouble(dLine);
  Console.WriteLine("dValue = " + dValue.ToString("E"));
  Console.WriteLine("Для выхода из программы нажмите ENTER.");
  Console.ReadLine();
  }
 }
```

Результаты выполнения программы:

```
bValue = False
iValue = 002010
dValue = 2,244000E+001
Для выхода из программы нажмите ENTER.
<ENTER>
```

Для наглядности в этой программе по сравнению с программой 03_01.cs по другому выполнена инициализация одной переменной: **string** bLine = “false”; что изменило результат.

Так как по умолчанию при выполнении программы стандартным считается формат записи вещественных чисел, установленный в настройках операционной системы, то при попытке решить ту же задачу, инициализируя переменную dLine значением “22.44”, выполнение методов double.Parse() или Convert.ToDouble( ) приведет (при европейском формате) к аварийной ситуации. При такой аварийной ситуации программа немедленно завершается. В консольное окно выводится большое сообщение, первая строка которого имеет вид:

Unhandled Exception: System.FormatException: Input string was not in a correct format. (Входная строка имеет неверный формат.)

Это результат реакции метода **double**. Parse( ) на неверные данные во входной строке-аргументе, которую он «желает видеть» только как правильную запись вещественного числа. Таким образом, при неверных данных метод Parse( ) и методы класса Convert посылают исключение, обработка которого в нашей программе не предусмотрена, и программа завершится с сообщением о необработанном исключении. Такая же аварийная ситуация возникнет и при любых других ошибках в символьной записи преобразуемого значения. Обрабатывать исключения мы пока не станем, а сейчас покажем еще один способ получения арифметических значений из их символьных представлений.

Задача 03-02. Присвоить переменным n и m значения, представленные в программе строкой со значением “63” и строковой константой (литералом) “37”. Использовать метод TryParse(). Вывести полученные значения переменных n и m.

```
// 03_02.cs – "чтение" числа из строки
using System;
class Program {
static void Main( ) {
string nLine = "63";       // Символьное представление числа
   int n, m;
   int.TryParse(nLine, out n);     // Число из строки
   int.TryParse("37", out m);      // Число из строковой константы
   Console.WriteLine("n = {0:D},  m = {1:D}", n, m);
   Console.WriteLine("Для выхода из программы нажмите ENTER.");
   Console.ReadLine();
   }
}
```

Результаты выполнения программы:

```
n = 63,  m = 37
Для выхода из программы нажмите ENTER.
<ENTER>
```

В применении метода TryParse( ) есть некоторые особенности, которые для начинающего могут оказаться непонятными. Эти методы, как и методы Parse( ), включены во все базовые классы (**int**, **double**, **float**...). Первый аргумент представляет в виде строки источник данных, второй аргумент – та переменная, которая получит формируемое значение. Второй аргумент всегда снабжается модификатором **out**. Необходимо, чтобы тип второго аргумента совпадал с типом того класса, которому принадлежит метод TryParse( ). Если это условие нарушено, то возникает ошибка компиляции, т.е. компилятор «не позволит» построить неверную программу. Однако возможность ошибиться в записи изображения числового значения остаётся – компилятор не проверяет правильность строки – первого аргумента. Если первый аргумент метода TryParse( ) неверно представляет значение соответствующего типа, то второй аргумент получает умалчиваемое для своего типа значение (для чисел 0), а возвращаемое значение метода TryParse() равно **false**. Таким образом, метод не посылает исключений, и тем самым не завершается аварийно выполнение программы при отсутствии их обработки.

Познакомившись со средствами получения арифметических и логических данных из строковых представлений, перейдем к чтению данных из стандартного консольного потока, который в программе представляет ввод данных с клавиатуры. В задаче

01-02 использовался метод Console.ReadLine( ), передающий в программу строку символов, набранную на клавиатуре. Если в прочитанной строке помещено символьное изображение значения базового типа, то, используя методы Parse( ) и TryParse( ) (а также средства класса Convert) можно в программе получить соответствующие числовые и логические значения. Единственное, но очень важное условие – необходимо знать тип вводимого значения, и его строковое представление не должно содержать ошибок. Как бороться с ошибками, допускаемыми при вводе, – это будет рассмотрено позже. Сейчас будем рассчитывать на идеального пользователя, который не ошибается.

Задача 03-03. Ввести значение вещественного числа и вывести значение ближайшего к нему по абсолютной величине целого числа. Для получения вещественного числа из введённой пользователем строки применить метод **double**.TryParse().

Для решения задачи можно было бы воспользоваться методом Math.Round() – округление вещественного числа. Результат применения этого метода к аргументу с дробным значением 0.5 – ближайшее чётное целое число. Предположим, что это нас не устраивает и, когда дробная часть числа равна 0.5, мы хотим всегда получать ближайшее целое число, по модулю большее вещественного числа.

```
// 03_03.cs – "чтение" числа из стандартного входного потока
using System;
class Program {
static void Main( ) {
      double x,              // Значение числа
      d;                     // "Поправка"
      int res;               // Ближайшее целое
      string line;       // Строка для приема данных
      Console.Write("Введите вещественное число:  ");
      line = Console.ReadLine();        // Чтение строки от клавиатуры
      double.TryParse(line, out x);     // Получение числа
      d = x > 0 ? 0.5 : -0.5;                  // Поправка
      res = (int)(x + d);                 // Приведение типов
      Console.WriteLine("Ближайшее целое: " + res);
      Console.WriteLine("Для выхода из программы нажмите ENTER.");
      Console.ReadLine();
         }
}
```

Результаты выполнения программы:

```
Введите вещественное число:  74,5 <ENTER>
Ближайшее целое: 75
Для выхода из программы нажмите ENTER.
<ENTER>
```

В программе используется метод **double**.TryParse( ). Если во входной строке изображение вещественного числа будет ошибочным, то метод присвоит нулевое значение второму аргументу.

Результаты выполнения программы при ошибке во входных данных:

```
Введите вещественное число:  34  фис 44<ENTER>
Ближайшее целое: 0
Для выхода из программы нажмите ENTER.
<ENTER>
```

Применяя метод TryParse(), удобно организовать защиту от ошибок пользователя при вводе исходных данных. В следующей задаче показано, как это делать.

Задача 03-04. Ввести значения двух вещественных переменных x и y, представляющих декартовы координаты точки на плоскости. Вычислить расстояние от точки до начала координат. Использовать метод TryParse( ) и предусмотреть повторный ввод при ошибках во входной строке. При выводе значения вычисленного расстояния использовать пять значащих цифр.

Для организации повторного ввода при неверно введённых данных используем оператор цикла с постусловием:

```
do тело_цикла while (условие);
```

В *теле_цикла* – ввод строки, в *условии* – проверка правильности строкового представления данных.

С учётом сказанного, программу можно написать так:

```
// 03_04.cs – синтаксический контроль входной строки
using System;
class Program {
static void Main( ) {
    double x, y;   // Координаты точки
    double range;  // Расстояние от начала координат
    string line;     // Строка для приема данных
```

```
    do {   // цикл синтаксического контроля ввода
        Console.Write("Введите x: ");
        line = Console.ReadLine();  // Чтение строки от клавиатуры
        }
    while(!double.TryParse(line, out x));
    do    // цикл синтаксического контроля ввода
     Console.Write("Введите y: ");
     while(!double.TryParse(Console.ReadLine(), out y));
     range = Math.Sqrt(x*x + y*y);
     Console.WriteLine("Расстояние от начала координат:
     {0:G5}", range);
     Console.WriteLine("Для выхода из программы нажмите ENTER.");
     Console.ReadLine();
     }
}
```

Результаты выполнения программы:

```
Введите x: 33 , 55<ENTER>
Введите x: radius<ENTER>
Введите x: 45<ENTER>
Введите y: 32,79<ENTER>
Расстояние от начала координат: 55,679
Для выхода из программы нажмите ENTER.
<ENTER>
```

В первых двух попытках ввести значение переменной x сделаны ошибки, и программа требует повторить ввод. Такое «поведение» обеспечивает оператор цикла с постусловием:

```
do {   // цикл синтаксического контроля ввода
    Console.Write("Введите x: ");
    line = Console.ReadLine();  // Чтение строки от клавиатуры
    }
while(!double.TryParse(line, out x));
```

В условии повторения итераций цикла проверяется результат выполнения метода TryParse(). Если первый аргумент – строка с правильной записью числа, то метод TryParse( ) возвращает значение true и выполнение цикла прерывается. В противном случае повторяется обращение к пользователю "Введите x: "...

Для ввода значения переменной с именем y структура цикла изменена. Сделано это только в иллюстративных целях. В теле цикла один оператор:

```
Console.Write("Введите y: ");
```

Чтение входной строки перенесено в обращение к методу TryParse(), где результат, возвращаемый методом Console.ReadLine(), используется в качестве первого аргумента.

Обратите внимание на спецификатор формата G5, использованный в операторе

```
Console.WriteLine("Расстояние от начала координат: {0:G5}", range);
```

Его роль при больших значениях выводимого значения демонстрируют следующие результаты:

```
Введите x: 10000000<ENTER>
Введите y: 23452354325<ENTER>
Расстояние от начала координат: 2,3452E+10
Для выхода из программы нажмите ENTER.
<ENTER>
```

Для ввода отдельных символов удобно применять метод Console.Read(), который читает из входного потока один символ и возвращает его код в виде значения типа int. В следующей задаче используется эта возможность.

Задача 03-05: Введите значения двух символьных переменных. Вывести их числовые коды и введенные значения (символы). Применить для ввода метод Console.Read().

Программу можно написать так:

```
// 03_05.cs – "чтение" символов из входного потока
using System;
class Program {
static void Main( ) {
    char one,  two;                 // Символьные переменные
    int cod1, cod2;                 // Коды символов
    Console.Write("Введите one: ");
    cod1 = Console.Read();          // Чтение кода символа от клавиатуры
    one = (char)cod1;               // Преобразование код -> символ
    Console.ReadLine();             // "Очистка" буфера
    Console.Write("Введите two: ");
    cod2 = Console.Read();          // Чтение кода символа от клавиатуры
    two = (char)cod2;               // Преобразование код -> символ
    Console.ReadLine();             // "Очистка" буфера
    Console.WriteLine("Коды:\t\tcod1={0},  cod2={1}",cod1,cod2);
    Console.WriteLine("Символы:\tone=\'{0}\',  two=\'{1}\'", one, two);
```

```
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результаты выполнения программы:

```
Введите one: 8<ENTER>
Введите two: ы<ENTER>
Коды:        cod1=56,  cod2=1099
Символы:     one='8',   two='ы'
Для выхода из программы нажмите ENTER.
<ENTER>
```

Нужно отметить, что читается методом Console.Read() всегда один символ. Если во входной строке несколько символов, то все они при нажатии клавиши ENTER заносятся во входной буфер, но прочитывается только первый. Если во входной строке только один символ, то даже в этом случае при нажатии клавиши ENTER в буфер дополнительно заносится ее код, точнее коды со значениями 13 (возврат каретки) и 10 (перевод строки). До чтения следующего символа необходима очистка буфера. Ее можно выполнить методом Console.ReadLine(). Именно это сделано в программе после чтения каждого символа.

Для сравнения возможностей методов Console.ReadLine() и Console.Read() приведем решение той же задачи с помощью метода Console.ReadLine().

Задача 03–06: Введите значения двух символьных переменных. Вывести введенные значения (символы) и их числовые коды. Применить для ввода метод Console.ReadLine(). «Извлекать» из строки символ методом **char**.Parse( ).

```
// 03_06.cs – "чтение" символов из строки. Коды символов.
using System;
class Program {
static void Main( ) {
    char one,  two;                 // Символьные переменные
    int cod1, cod2;                 // Коды символов
    string input;                   // Читаемая строка
    Console.Write("Введите one: ");
    input = Console.ReadLine(); // Чтение изображения символа
    one = char.Parse(input);        // Получение символа
    cod1 = (int)one;                // Преобразование код -> символ
    Console.Write("Введите two: ");
```

```
    input = Console.ReadLine(); // Чтение изображения символа
    two = char.Parse(input);    // Получение символа
    cod2 = (int)two;            // Преобразование код -> символ
    Console.WriteLine("Коды:\t\tcod1={0}, cod2={1}",cod1,cod2);
    Console.WriteLine("Символы:\tone=\'{0}\', two=\'{1}\'", one, two);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результаты выполнения программы:

```
Введите one: 7<ENTER>
Введите two: п<ENTER>
Коды:          cod1=55,          cod2=1087
Символы:       one='7', two='п'
Для выхода из программы нажмите ENTER.
<ENTER>
```

Как и ранее в предыдущих программах из консольного потока метод ReadLine() прочитывает строку и присваивает результат переменной input. Затем строковое изображение символа преобразуется в значение символьной переменной методом char.Parse(input). С помощью операции приведения типов (**int**) из символа формируется его числовой код.

С точки зрения пользователя как будто ничего не изменилось. (Смотрите результаты выполнения программы.) Но при ошибочном вводе не одного, а нескольких символов (то есть при вводе строки длиннее одного символа) программа завершится аварийно. Метод **char**.Parse() не может (не умеет) выделить из строки-аргумента один символ. Второе отличие чтения символов методом ReadLine() – отсутствие необходимости в очистке буфера после ввода каждого отдельного символа. Метод ReadLine() не только читает строку, но и удаляет из буфера коды, соответствующие нажатию клавиши ENTER. Можно после чтения методом ReadLine() строки с нужным символом выделять этот символ методами класса String и затем получать значение символа. Но эти возможности удобнее рассматривать, изучая методы работы со строками.

ТЕМА 04

УПРАВЛЕНИЕ КОНСОЛЬЮ

Прошли те времена, когда консольный экран и клавиатура были основными инструментами для общения пользователя с выполняемой программой. Но и сейчас консоль – это наиболее удобное средство для того, чтобы изучать язык программирования и не утонуть в богатейших возможностях той или иной среды, предназначенной для автоматизации разработки полноэкранных программ с графическим интерфейсом. В ряде случаев применение консольных приложений весьма полезно и вне изучения языка программирования. Вот в этих случаях целесообразно уметь программно получать характеристики консоли и изменять их в зависимости от требований решаемой задачи.

В начале этой темы отметим, что в дальнейшем изложении мы будем использовать в дополнении к статическим консольным методам ReadLine( ), Read( ), WriteLine( ), Write( ), применяемым в темах 2 и 3, только метод Console.ReadKey() и необходимую для его применения структуру ConsoleKeyInfo. (см. Приложение 3).

Console.ReadKey( ) – это статический метод класса Console, который даёт возможность получать информацию о действиях пользователя в консольном режиме выполнения программы. С его помощью программа получает код очередной нажатой клавиши, отличной от функциональных ALT, CTRL, SHIFT. Метод перегружен. Один из его вариантов – метод с параметром булевского типа – позволяет управлять выводом на экран изображения символа нажатой клавиши. Заголовок метода:

public static ConsoleKeyInfo ReadKey(bool** intercept**)

Прежде чем говорить о типе значения, которое возвращает метод ReadKey(), отметим особенности его выполнения. Метод «приостанавливает» выполнение программы и ожидает нажатия клавиши. Клавиша может быть нажата в сочетании с одной из функциональных клавиш ALT, CTRL, SHIFT, однако метод «не реагирует» на отдельное нажатие только

функциональной клавиши. Когда клавиша (одна или в сочетании с функциональной клавишей) нажата, метод ReadKey() возвращает соответствующее значение (структуру) типа ConsoleKeyInfo. Одновременно изображение символа появится в консольном окне, если аргумент, заменяющий параметр intercept, имеет значение **false**.

Структура типа ConsoleKeyInfo, возвращаемая методом ReadKey(), содержит информацию, позволяющую узнать, какую клавишу и в сочетании с какой функциональной клавишей, нажал пользователь на клавиатуре. Полные сведения о членах структуры, возвращаемой методом ReadKey(), нам не понадобятся. (С ним можно познакомиться, обратившись к справочной системе MSDN.)

Для применения метода Console.ReadKey() обычно определяют переменную типа ConsoleKeyInfo и присваивают этой переменной значение, возвращаемое методом. Например, так:

```
ConsoleKeyInfo клавиша = Console.ReadKey();
```

После этого можно с помощью переменной *клавиша* обращаться к членам структуры и получать нужную информацию о нажатых клавишах.

Среди членов структуры ConsoleKeyInfo обратим внимание на следующие два свойства:

Key – содержит информацию о нажатой клавише, представленной структурой;
KeyChar – содержит символ в кодировке Unicode, представленный структурой.

Имеется ещё свойство Modifiers, позволяющее получить сведения о тех функциональных клавишах, в сочетании с которыми нажата клавиша.

Свойство Key имеет тип перечисления ConsoleKey. Его члены – именованные константы – представляют все клавиши, имеющиеся на клавиатуре за исключением функциональных ALT, CTRL, SHIFT. Вот некоторые из них:

```
Backspace, Tab, Enter, Escape, End, Home ...
```

Для ознакомления с возможностями метода Console. ReadKey() и свойствами Key и KeyChar рассмотрим следующую задачу.

Задача 04-01. Составьте программу, которая выводит значения кодов нажимаемых пользователем клавиш и соответствующие им символы. Программа должна работать до тех пор, пока пользователь не нажмет клавишу Escape.

```
// 04_01.cs. Нажимая клавиши, наблюдайте за значениями на экране
using System;
class Program {
static void Main( ) {
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    Console.WriteLine("Нажмите любую клавишу!");
    ConsoleKeyInfo клавиша; // Нажимаемая пользователем клавиша
    do  {
            клавиша = Console.ReadKey(true);
            Console.WriteLine("клавиша.Key.ToString()="+
            клавиша.Key.ToString());
            Console.WriteLine("(int)клавиша.KeyChar="+
            (int)клавиша.KeyChar);
    } while (клавиша.Key != ConsoleKey.Escape);
 }
}
```

Прежде чем привести результаты выполнения программы, обратимся к ее тексту. В первых строках нет ничего нового. В операторе

```
ConsoleKeyInfo клавиша;
```

объявлена переменная с типом структуры ConsoleKeyInfo. Далее цикл с постусловием, вводимый служебным словом **do**. В теле цикла переменной с именем *клавиша* присваивается результат выполнения метода Console.ReadKey(). Метод выполняется при нажатии любой клавиши, отличной от функциональных (ALT, CTRL, SHIFT). Возвращаемое значение метода – ссылка на экземпляр структуры ConsoleKeyInfo, представляющий данные о нажатой клавише. В качестве аргумента при обращении к методу ReadKey() использовано булевское значение **true**. Тем самым метод, формируя результат, запрещает вывод на экран изображения символа, соответствующего клавише.

В следующих двух операторах используются два свойства объекта структуры ConsoleKeyInfo. Свойство Key принимает значение той из констант перечисления ConsoleKey, которая соответствует клавише, представленной объектом ConsoleKeyInfo. Чтобы получить его символьное представление использовано вы-

ражение *клавиша.Key.ToString()*. Значение свойства KeyChar – это код в кодировке юникод символа для клавиши, представленной объектом ConsoleKeyInfo. Это значение выводится на экран с помощью оператора

```
Console.WriteLine("клавиша.KeyChar="+клавиша.KeyChar);
```

Строка кода

```
} while (клавиша.Key != ConsoleKey.Escape);
```

завершает тело цикла и в скобках содержит условие продолжения итераций цикла. Значение свойства *клавиша.Key* сравнивается с константой ConsoleKey.Escape, соответствующей нажатию клавиши Escape. Если значения не равны, то результат равен **true** и тело цикла выполняется вновь.

Результаты выполнения программы:

```
Для выхода из программы нажмите ESC.
Нажмите любую клавишу!
клавиша.Key.ToString()=Z
(int)клавиша.KeyChar=1263
клавиша.Key.ToString()=Z
(int)клавиша.KeyChar=122
клавиша.Key.ToString()=Enter
(int)клавиша.KeyChar=13
клавиша.Key.ToString()=Spacebar
(int)клавиша.KeyChar=32
клавиша.Key.ToString()=Escape
(int)клавиша.KeyChar=27
<ESC>
```

В результатах выполнения программы обратите внимание на то, что нажатие клавиши Z первый раз выполнено в русской кодировке. При русской кодировке числовым кодом свойства KeyChar будет 1263. Но Key.ToString() вернет символ, соответствующей клавиши, т.е. ‘Z’. Далее пользователь, используя сочетание функциональных клавиш, переключил клавиатуру на ввод латинских символов. Это не отражается в результатах, так как программа, точнее метод Console.ReadKey( ), на нажатие функциональных клавиш не реагирует. Но следующее нажатие клавиши Z приводит к получению структуры, для которой числовой код свойства KeyChar равен 122.

Кроме собственно вывода на экран значений свойств нажимаемых клавиш приведенная задача иллюстрирует способ по-

строения программ, позволяющих пользователю многократно вводить новые и новые исходные данные, не повторяя запусков программы на исполнение.

В общем виде код, обеспечивающий многократное исполнение программы без повторного запуска, может быть таким:

```
Console.WriteLine("Для выхода из программы нажмите ESC.");
do   {
       // код, обеспечивающий функциональность программы
       } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
```

В приведённом образце кода отсутствует переменная с типом ConsoleKeyInfo. Так как обычно не требуется знать, какую клавишу (отличную от Escape) нажал пользователь для продолжения работы программы, то переменная типа ConsoleKeyInfo не нужна в явном виде. В условии продолжения цикла непосредственно анализируется свойство Key результата, возвращаемого методом Console.ReadKey(true).

Задача 04-02. Выведите значения свойств класса Console, опредяющих внешний вид консольного окна: WindowHeight, WindowWidth, BufferHeight, BufferWidth, BackgroundColor, ForegroundColor. Используя константы перечисления ConsoleColor, измените свойства BackgroundColor и ForegroundColor, и вновь выведите их значения. С помощью свойства Console.Title выведите в заголовок консольного окна номер задачи.

```
// 04_02.cs - Изучение свойств класса Console.
using System;
class Program {
static void Main( ) {
  Console.Title = "Задача 04-02.";
  Console.WriteLine("Console.WindowHeight = "+
        Console.WindowHeight);
  Console.WriteLine("Console.WindowWidth = "+
        Console.WindowWidth);
  Console.WriteLine("Console.BufferHeight = "+
        Console.BufferHeight);
  Console.WriteLine("Console.BufferWidth = "+
        Console.BufferWidth);
  Console.WriteLine("BackgroundColor="+
        Console.BackgroundColor);
```

```
    Console.WriteLine("ForegroundColor="+
            Console.ForegroundColor);
    Console.BackgroundColor=ConsoleColor.Yellow;
    Console.ForegroundColor=ConsoleColor.Red;
    Console.WriteLine("BackgroundColor="+
            Console.BackgroundColor);
    Console.WriteLine("ForegroundColor="+
            Console.ForegroundColor);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

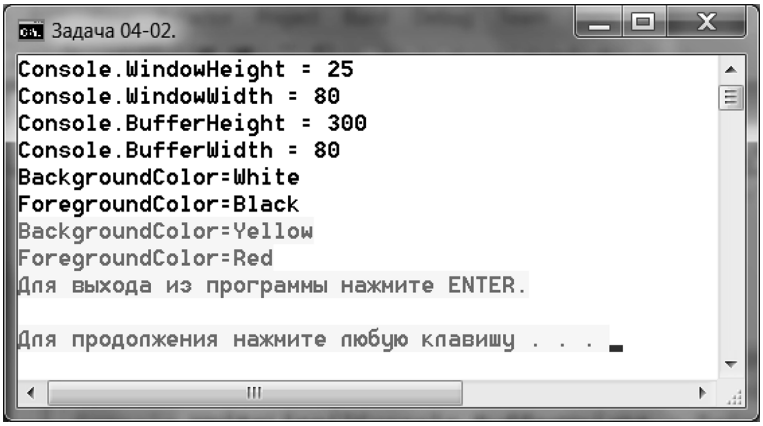


Рис. 4.1. Консольное окно при выполнении программы 04_02

Хотя это не видно при черно-белой печати изображения экрана (см. рис. 4.1), но после выполнения операторов

```
Console.BackgroundColor=ConsoleColor.Yellow;
Console.ForegroundColor=ConsoleColor.Red;
```

фон стал желтым, а текст выводится красными буквами.

Задача 04-03. Выведите значения 350 натуральных чисел, размещая каждое число на отдельной строке и «окрашивая» каждое десятое в красный цвет.

Из результатов выполнения предыдущей программы (см. рис. 4.1) видно, что буфер консоли (Console.BufferHeight) по

умолчанию вмещает 300 строк. Поэтому для просмотра трехсот пятидесяти строк придется увеличить буфер консоли, иначе первые 50 строк «пропадут». Программа, решающая поставленную задачу, может быть такой:

```
// 04_03.cs – Свойства класса Console и перечисление
ConsoleColor.
using System;
class Program {
static void Main( ) {
Console.BufferHeight =400; // размер  буфера
// запомнить  цвет  шрифта:
ConsoleColor old = Console.ForegroundColor;
for(int i=1; i <= 350; i++){
    if (i%10 == 0)   Console.ForegroundColor=ConsoleColor.Red;
    Console.WriteLine(i);
    Console.ForegroundColor=old;
    }
    Console.WriteLine("Размеры консольного буфера:");
    Console.WriteLine("BufferHeight="+Console.BufferHeight);
    Console.WriteLine("BufferWidth="+Console.BufferWidth);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

В программе размер буфера консоли увеличен до 400. Затем объявлена переменная old типа ConsoleColor. Тип ConsoleColor – это перечисление, константы которого представляют основные цвета, применяемые для окраски фона и текстовой информации. Переменная old инициализирована значением свойства Console.ForegroundColor, представляющего текущее значение цвета шрифта в консольном окне. В цикле при выводе каждого десятого значения это свойство изменяется. Ему в этот момент присваивается значение ConsoleColor.Red, то есть текст выводится красным шрифтом. После выполнения оператора Console.WriteLine(i); свойству Console.ForegroundColor присваивается сохраненное в переменной old исходное значение. Таким образом, красными становятся только числа, кратные 10.

Результаты (не полные) выполнения программы:

```
1
2
.....................................................
349
350
Размеры консольного буфера:
BufferHeight=400
BufferWidth=80
Для выхода из программы нажмите ENTER.
<ENTER>
```

Задача 04-04. Составить программу, позволяющую пользователю изменять размеры консольного окна в процессе выполнения программы.

Размеры консольного окна определяют свойства Console.WindowWidth и Console.WindowHeight. В программе установим их исходные значения, а затем в зависимости от нажимаемых пользователем клавиш, будем увеличивать и уменьшать текущие значения высоты и ширины, изменяя значения названных свойств. Так как существуют системные требования к размерам консольного окна, то ширину окна разрешим изменять от 10 до 128, а высоту – от 10 до 36. Для «управления» размерами используем клавиши со стрелками:

DownArrow – уменьшить высоту на 10%;
UpArrow – увеличить высоту на 10%;
RightArrow – увеличить ширину на 10%;
LeftArrow – уменьшить ширину на 10%.

Изменять на 10% будем текущее значение соответствующего свойства с учетом указанных выше предельных значений. Размеры консольного окна определяются количеством знакомест и всегда целочисленные. Поэтому при их изменении на 10% будем округлять получаемые значения до целых чисел, отбрасывая дробные части.

Текст программы:

```
// 04_04.cs. Нажимая клавиши, наблюдайте за размерами экрана
using System;
class Program {
static void Main( ) {
```

```
    Console.Title = "Задача 04-04.";
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    Console.WriteLine("Нажмите клавишу со стрелкой!");
    ConsoleKeyInfo key; //Нажимаемая пользователем клавиша
    Console.WindowWidth = 40;
    Console.WindowHeight = 15;
    do  {
        key = Console.ReadKey(true);
        Console.WriteLine("клавиша: " + key.Key.ToString());
        if(key.Key == ConsoleKey.RightArrow)// увеличить  ширину
        Console.WindowWidth =
        Math.Min(128,(int)(Console.WindowWidth*1.1));
        if(key.Key == ConsoleKey.LeftArrow) // уменьшить  ширину
        Console.WindowWidth =
        Math.Max(10,(int)(Console.WindowWidth*0.9));
        if(key.Key == ConsoleKey.UpArrow)  // увеличить  высоту
        Console.WindowHeight =
        Math.Min(36,(int)(Console.WindowHeight*1.1));
        if(key.Key == ConsoleKey.DownArrow) // уменьшить высоту
        Console.WindowHeight =
        Math.Max(10,(int)(Console.WindowHeight*0.9));
        } while (key.Key != ConsoleKey.Escape);
    }
}
```

В программе объявлена переменная key типа ConsoleKeyInfo. Возможности этого типа уже рассматривались в разборе программы 04_01. В цикле с постусловием переменная key получает значение, формируемое методом Console.ReadKey(true) при каждом нажатии пользователя на клавишу, отличную от функциональной. Символьное представление присвоенного переменной key значения выводится на экран оператором

```
Console.WriteLine("клавиша: " + key.Key.ToString());
```

Если нажата клавиша ESC, то работа программы завершается. На клавиши, отличные от стрелок, размеры окна не реагируют. При нажатии какой-либо клавиши со стрелкой выполняется соответствующий условный оператор и размер консольного окна изменяется на 10% по одному из измерений. На рис. 4.2 показан пример консольного окна при завершении работы программы.

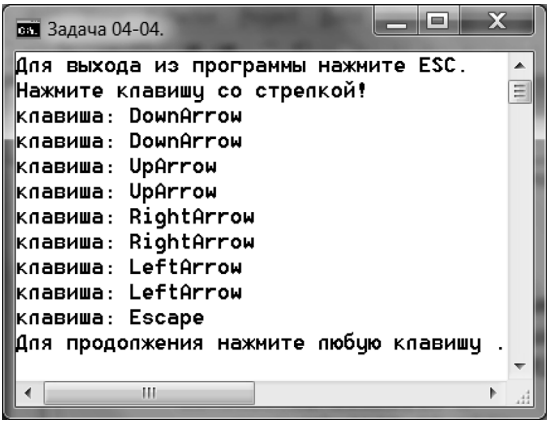


Рис. 4.2. Консольное окно при завершении программы 04_04.cs

Задача 04-05. Выведите белым шрифтом в центр консольного окна подсказку пользователю: “Для выхода из программы нажмите ESC.”. При каждом нажатии клавиши, отличной от ESC, окраска (цвет) фона должна меняться, циклически принимая разные цвета.

```
// 04_05.cs – циклическое изменение цвета фона окна
using System;
class Program {
static void Main( ) {
    Console.Title = “Циклическое изменение цвета фона окна”;
    // Массив констант перечисления, определяющего цвет:
    ConsoleColor [] color = {ConsoleColor.Red,
    ConsoleColor.Yellow, ConsoleColor.Green,
    ConsoleColor.DarkBlue,
    ConsoleColor.Blue, ConsoleColor.DarkRed  };
    int back = 0;  // Счетчик
    Console.ForegroundColor = ConsoleColor.White;
    do  {
        Console.BackgroundColor = color[back++%color.Length];
        Console.Clear();
        Console.CursorLeft = Console.WindowWidth/4;
        Console.CursorTop = Console.WindowHeight/2;
        Console.WriteLine(“Для выхода из программы нажмите ESC.”);
        } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

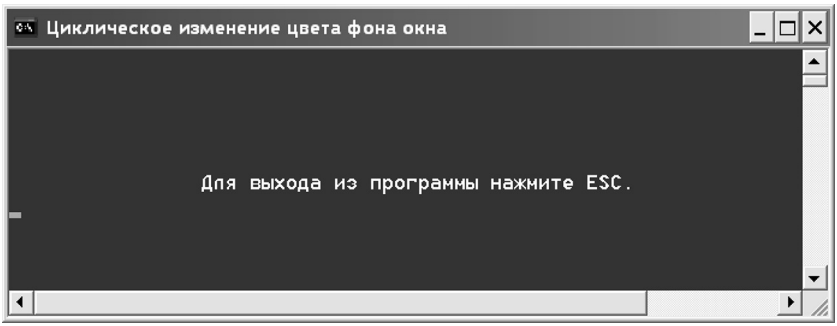


Рис. 4.3. Консольное окно, изменяющее цвет фона

В программе 04_05.cs для перебора цветов используется массив типа ConsoleColor[], инициализированный константами перечисления ConsoleColor. В языке C# есть возможность не указывать конкретные имена констант перечисления, а получать их с помощью статического метода GetNames(). Параметром этого метода должен быть тип перечисления. Для получения типа перечисления к имени конкретного перечисления применяется операция **typeof**. Метод GetNames() возвращает массив строк с именами констант перечисления. Для нашей задачи соответствующий оператор будет таким:

```
string [ ] names = ConsoleColor.GetNames(typeof(ConsoleColor));
```

Массив, именуемый ссылкой names, будет содержать имена всех констант перечисления ConsoleColor. В следующей программе этот массив используется.

```
// 04_05_1.cs - циклическое изменение цвета фона окна
using System;
class Program {
static void Main( ) {
    // Массив имен констант перечисления, определяющего цвет:
    string [] names = ConsoleColor.GetNames(typeof(ConsoleColor));
    int back = 0;  // Счетчик
     do  {
        Console.ForegroundColor =
        names[back%names.Length] != "White" ? ConsoleColor.White:
        ConsoleColor.Black;
        Console.BackgroundColor =
        (ConsoleColor)Enum.Parse(typeof(ConsoleColor),
        names[back++%names.Length]);
```

```
        Console.Clear();
        Console.CursorLeft = Console.WindowWidth/4;
        Console.CursorTop = Console.WindowHeight/2;
        Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы 04_05_1.cs здесь нет необходимости приводить, они подобны результатам программы 04_05.cs.

При выполнении реальных программ достаточно часто появляется ситуация, когда процесс счета достаточно продолжителен. Таковы программы моделирования физических процессов, обработки сложных запросов с обращениями к большим базам данных и т.д. В таких случаях при выполнении программы пользователю зачастую нет необходимости постоянно находиться у консоли. Его роль – подготовка исходных данных, запуск программы на выполнение и принятие решений при анализе промежуточных результатов. О необходимости вмешательства пользователю удобно следить по звуковым сигналам, выдаваемым при завершении того или иного этапа обработки. Подачу таких звуковых сигналов можно реализовать, включив в программу соответствующие фрагменты кода. При работе в консольном режиме для этих целей используют метод Console.Beep(). В консольном классе их два с такими заголовками:

```
public static void Beep()
public static void Beep(int frequency, int duration)
```

Выполнение каждого из этих методов приводит к выдаче звукового сигнала. Метод без параметров выдает сигнал фиксированной длительности и постоянной частоты. Метод с параметрами позволяет создавать разнообразные звуковые эффекты. Первый параметр frequency задает частоту звукового сигнала (в герцах). Второй параметр служит для задания длительности звучания (в миллисекундах).

Задача 04-06. Для иллюстрации возможностей метода Console.Beep() рассмотрим в качестве примера задачу генерации музыкальной гаммы с нарастающей продолжительностью звучания каждой ноты.

```
// 04_06.cs - Изучение возможностей метода Beep()
// Воспроизведение музыкальной гаммы
using System;
class Program {
static void Main( ) {
    // Массив частот (нот) звуковых сигналов:
    int [] notes = {262,294,330,349,392,440,494};
    // Массив длительностей звуковых сигналов:
    int [] durations = {1000,1100,1200,1300,1400,1500,1600};
    do  { // Цикл "проигрывания" нот:
    for(int i=0; i < notes.Length; i++)
    Console.Beep(notes[i], durations[i]);
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы можно услышать...

В качестве прототипа для программы 04_06.cs использована программа на языке Паскаль из книги В.Б. Попова "Turbo Pascal для школьников". Оттуда же взята формулировка следующей задачи.

Задача 04-07. Воспроизведите программно звук сирены машины со спецсигналом. Частота (в герцах) изменяется от 500 до 2000 и от 2000 до 500 с шагом 5. Продолжительность звучания каждой частоты 1 миллисекунда.

```
// 04_07.cs - Изучение возможностей метода Beep()
// Воспроизведение звука сирены
using System;
class Program {
static void Main( ) {
     do  {        // Цикл перебора частот в прямом направлении:
     for(int i=500; i < 2000; i+=5)
     Console.Beep(i, 1);
     for(int i=2000; i > 500; i-=5)
     Console.Beep(i, 1);
     Console.WriteLine("Для выхода из программы нажмите ESC.");
     } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы можно услышать.

ТЕМА 05

ВЫРАЖЕНИЯ, УСЛОВНЫЕ ОПЕРАТОРЫ, ЦИКЛЫ

Цели темы – иллюстрация особенностей применения традиционных средств императивных языков программирования.

Задача 05-01. Найти трехзначное десятичное число s, все цифры которого одинаковы и которое представляет собой сумму первых членов натурального ряда, то есть s = 1 + 2 + 3 + 4 +... Вывести полученное число и изображение соответствующих ему членов ряда в виде суммы.

```
// 05_01 - анализ суммы первых членов натурального ряда
using System;
class Program {
static void Main() {
    int s = 0,                  // сумма  членов
    i = 0;                      // номер и значение члена
    string report = "1";        // Запись  суммы
    do  {
       s += ++i;                                  // Вычисление  суммы
       report += i == 1 ? "":"+" + i;             // Изображение  ряда
       if (s <= 99) continue;
       intc1 = s/100,                             // Первая цифра
       c2 = (s - c1*100)/10,                      // Вторая цифра
       c3 = s - c1*100 - c2*10;                   // Младшая цифра
       if (c1 == c2 && c2 == c3) break;           // прерываем цикл
       } while (s <= 999); // сумма меньше четырехзначного числа
       if (s > 999) {
          Console.WriteLine("Нужное число не найдено!");
          return; }
       Console.WriteLine(report);
       Console.WriteLine("Сумма членов ряда: "+ s);
       Console.Write("Для выхода из программы нажмите Enter.");
       Console.ReadLine();
  }
}
```

Результаты выполнения программы:

```
1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20+21
+22+23+24+25+26+27+28+29+30+31+32+33+34+35+36
Сумма членов ряда: 666
Для выхода из программы нажмите Enter.
<ENTER>
```

Программа не требует ввода исходных данных. Во внешнем блоке определены две целочисленных переменных *i* – для представления очередного слагаемого суммы и *s* – для значения суммы. Для формирования изображения суммируемого ряда объявлена строковая переменная report. Она инициализирована изображением цифры "1".

В теле цикла с постусловием значение суммы последовательно увеличивается на величину очередного члена ряда. Если значение члена отлично от 1, то к строке report приписывается изображение выражения "+"+i. При выполнении условия s <= 99 следует переход к очередной итерации цикла (оператор **continue**), так как по условию искомая сумма должна быть трехзначным числом. Из трехзначных чисел, представляющих последовательные суммы, выделяются значащие цифры (целочисленные переменные c1, c2, c3). Если они равны, то искомая сумма найдена (выполняется оператор **break**). Обратим внимание на условие проверки их равенства c1 == c2 && c2 == c3. В нем использована условная логическая операция конъюнкции &&. Если первый операнд равен **false**, то в выраражении с этой операцией нет необходимости проверять значение второго операнда.

Цикл может завершиться в двух случаях, во-первых, когда найдено нужное значение суммы и, во вторых, когда сумма превысит значение 999, то есть ее значение станет четырехзначным числом. В этом втором случае выводится сообщение *"Нужное число не найдено!"* и оператор **return** завершает выполнение программы. Если число найдено, то выводится изображение суммы и ее значение, как показано в результатах выполнения программы.

Задача 05-02. Введя натуральное n, вычислите приближенное значение n-го члена ряда Фибоначчи по формуле Бине:

$$U_n = (b^n - (-b)^{-n}) / \sqrt{5}, \text{ где } b = (1 + \sqrt{5}) / 2.$$

Выведите точное значение n-го числа Фибоначчи, вычислив его по рекуррентной формуле: $F_i = F_{i-1} + F_{i-2}$, где $F_1 = F_2 = 1$, $i > 2$.

Для вычисления по формуле Бине применим переменные и константы вещественного типа (типа **double**). Для демонстрации отличий целочисленной арифметики от вычислений с действительными числами, используем при непосредственном вычислении членов ряда по рекуррентной формуле переменные целого типа (типа **int**).

```
// 05_02 - формула Бине для чисел Фибоначчи
using System;
class Program  {
public static void Main()  {
  uint n;                  // Номер члена ряда
  double b,                // Вспомогательная переменная
  un;                      // Оценка по формуле Бине
  do {                     // Цикл повторения решений
       do              // Цикл с проверкой введенных данных
       Console.Write("Введите номер члена ряда: ");
       while (uint.TryParse(Console.ReadLine(), out n) == false
              || n == 0);    // Семантическая проверка
       b = (1 + Math.Sqrt(5)) / 2;
       un = (Math.Pow(b, n) - Math.Pow(-b, -n)) / Math.Sqrt(5);
       // Рекурентное вычисление члена ряда:
       int f1 = 1, f2 = 1, fn=1;
       for(int k=2; k<n; k++) {
            fn = f1 + f2;   // fn = checked(f1 + f2);
            f1 = f2; f2 = fn;
            }
       Console.WriteLine("un = " + un + ", число Фибоначчи: " + fn);
       Console.WriteLine("Для выхода из программы нажмите ESC.");
       } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
  } //Конец определения метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Введите номер члена ряда: 7<ENTER>
un = 13, число Фибоначчи: 13
Для выхода из программы нажмите ESC.
Введите номер члена ряда: 46<ENTER>
un = 1836311903, число Фибоначчи: 1836311903
Для выхода из программы нажмите ESC.
Введите номер члена ряда: 47<ENTER>
un = 2971215073, число Фибоначчи: -1323752223
Для выхода из программы нажмите ESC.
<ESC>
```

Недостаток программы – неверный результат, формируемый при вычислении очередного 47-го члена ряда по рекуррентной формуле. При суммировании двух целых положительных чисел результат выходит за разрядную сетку и интерпретируется как отрицательное число. В нашей программе эта аварийная ситуация очевидна, но в больших программах такая ошибка может быть не замечена, что приведет к непредсказуемым последствиям.

Возможная защита – применение специальной операции контроля за арифметическими переполнениями **checked**.

После замены оператора fn = f1 + f2; на оператор

```
fn = checked(f1 + f2);
```

при вводе числа, превышающего 46, программа завершится аварийно с выдачей сообщения о необработанном исключении, связанным с переполнением при арифметической операции.

Результаты выполнения программы будут такими:

```
Введите номер члена ряда: 47<ENTER>
Unhandled Exception: System.OverflowException:
Arithmetic operation resulted in an overflow.
at Program.Main( ) in
C:\C#_Практикум\Тема_05\Program_02\Program_02.cs:line 19
```

Примечание. Сообщение может быть выведено на русском языке в таком виде:
Необработанное исключение: System.OverflowException:
Переполнение в результате выполнения арифметической операции в Program.Main() в C:\C#_Практикум\Тема_05\Program_02\Program_02.cs:строка 19

В сообщении указан характер события, приведшего к возникновению исключения, и номер строки в коде программы, где это событие произошло.

Задача 05-03. Ввести значения трех переменных x, y, z. Применяя выражения с тернарной операцией ?: (и не применяя оператор **if**), обменяйте значения переменных, чтобы выполнилось требование: x <= y <= z.

```
// 05_03 – упорядочение значений трех переменных
using System;
class Program  {
public static void Main()  {
```

```
    int x, y, z;
    do  { // Цикл повторения решения
    do
        Console.Write("Введите значение x: ");
    while (int.TryParse(Console.ReadLine(), out x) == false);
    do
        Console.Write("Введите значение y: ");
    while (int.TryParse(Console.ReadLine(), out y) == false);
    do
        Console.Write("Введите значение z: ");
    while (int.TryParse(Console.ReadLine(), out z) == false);
    int a1 = 0, a2 = 0, a3 = 0;
    a1 = x < y ? (z < x ? z : x) : (y < z ? y : z);
    a3 = x > y ? (z > x ? z : x) : (y > z ? y : z);
    a2 = x + y + z - a1 - a3;
    x = a1; y = a2; z = a3;
    Console.WriteLine("x = " + x);
    Console.WriteLine("y = " + y);
    Console.WriteLine("z = " + z);
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} //Конец определения метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Введите значение x: 9<ENTER>
Введите значение y: 8<ENTER>
Введите значение z: 5<ENTER>
x = 5
y = 8
z = 9
Для выхода из программы нажмите ESC.
<ESC>
```

Для тех, кто понимает семантику тернарной операции, комментарии, по-видимому, не требуются. При необходимости обратитесь к одному из пособий по языку C#.

Задача 05-04. Ввести целое число. Сообщить, является ли это число кодом цифры, кодом буквы русского алфавита (прописной либо строчной), или не тем и не другим. Для ознакомления с числовыми диапазонами конкретных кодов вывести коды «граничных» букв.

Основная цель этой задачи – обратить внимание на тот факт, что во многих случаях для работы с кодами символов не нужно знать конкретные числовые значения их кодов.

```
// 05_04 – анализ числовых значений кодов символов
using System;
class Program  {
public static void Main()  {
      uint code;
      string str;                  // Строка для приема данных
      string report;               // Строка с заключением о коде
      uint код_А = (uint)'А',  // Числовое значение кода буквы А
          код_а = (uint)'а',
          код_я = (uint)'я',
          код_Я = (uint)'Я',
          код_0 = (uint)'0';    // Числовое значение кода цифры 0
      Console.WriteLine("Коды граничных символов:");
      Console.WriteLine("Код А = " + код_А +
          ";\nКод Я = " + код_Я + ";\nКод а = " + код_а +
          ";\nКод я = " + код_я + ";\nКод нуля = " + код_0);
      do  { // Цикл повторения решения
           do
              Console.Write("Введите значение code: ");
           while (uint.TryParse(Console.ReadLine(), out code) == false);
           report = code <= '9' && code >= 0 ?
               "Это цифра: " + (char)code: code <= 'Я' && code >= 'А'
               ? "Прописная буква: " + (char)code
               : code <= 'я' && code >= 'а'
               ? "Строчная буква: " + (char)code
               : "Неизвестный символ!";
           Console.WriteLine(report);
           Console.WriteLine("Для выхода из программы нажмите ESC.");
     } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} //Конец определения метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Коды граничных символов:
Код А = 1040;
Код Я = 1071;
Код а = 1072;
Код я = 1103;
Код нуля = 48
Введите значение code: 1050<ENTER>
```

```
Прописная буква: K
Для выхода из программы нажмите ESC.
Введите значение code: 50<ENTER>
Это цифра: 2
Для выхода из программы нажмите ESC.
Введите значение code: 123<ENTER>
Неизвестный символ!
Для выхода из программы нажмите ESC.
<ESC>
```

В коде программы стоит обратить внимание на выражения, в которых символы входят наряду с целочисленными значениями (например, code <= ‘Я’).

Задача 05-05. Расчет сложных процентов выполняется по формуле: s = k * (1 + r / 100)n, где k – начальный капитал; r – годовая процентная ставка; n – число лет; s – итоговая сумма. Ввести значения k, r, n. Вывести итоговую сумму в конце каждого года.

```
//05_05.cs - сложные проценты
using System;
class Program  {
static void Main()    {
    ConsoleKeyInfo клавиша; // Нажатая  пользователем  клавиша
    double k,          // начальный капитал
    r,                 // годовая процентная ставка
    s,                 // итоговая сумма в конце года
    term;              // вспомогательная переменная
    uint n;            // число лет
    do  // цикл для повторения решений задачи
    {  // Конструкция для ввода значения:
     do Console.Write("Введите начальный капитал: ");
     while (!double.TryParse(Console.ReadLine(), out k)
          | k <= 0);          // Проверка логики
     do Console.Write("Введите годовую процентную ставку: ");
     while (!double.TryParse(Console.ReadLine(), out r)
          | r <= 0);          // Проверка логики
     do Console.Write("Введите число лет: ");
     while (!uint.TryParse(Console.ReadLine(), out n)
          | n <= 0);          // Проверка логики
     term = 1 + r / 100;
     for (int i = 1; i <= n; i++, term *= 1 + r / 100)     {
          s = k * term;
```

```
            Console.WriteLine("n ={0,2:D}\ts = {1,8:F2}", i, s);
            }
        Console.WriteLine("Для выхода из программы нажмите ESC");
        клавиша = Console.ReadKey(true);
        } while (клавиша.Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите начальный капитал: 1000<ENTER>
Введите годовую процентную ставку: 17<ENTER>
Введите число лет: 5<ENTER>
n = 1   s =  1170,00
n = 2   s =  1368,90
n = 3   s =  1601,61
n = 4   s =  1873,89
n = 5   s =  2192,45
Для выхода из программы нажмите клавишу ESC
<ESC>
```

В коде программы следует обратить внимание на заголовок цикла:

```
for (int i = 1; i <= n; i++, term *= 1 + r / 100)
```

Значение годовой процентной ставки (переменная r) вводится в процентах. Для расчетов в коде программы объявлена переменная term, которой до цикла присвоено значение 1+r/100. Затем в заголовке цикла после каждого выполнения его тела этой переменной присваивается значение term *(1+r/100). Таким образом, возведение в степень заменено последовательностью умножений, и переменная term на каждой итерации цикла равна значению (1+r/100)*i, где i – параметр цикла.

Задача 05-06. Вычислить sin x с точностью до машинного нуля, используя приближенную формулу:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} \ldots (-1)^m \frac{x^{2m+1}}{(2m+1)!} \ldots,$$

где m – номер очередного члена, начиная с m = 0. Вывести последовательно вычисляемые значения частных сумм ряда, и соответствующие члены ряда. После достижения требуемой точности напечатать полученное значение sin x, значе-

ние отброшенного члена ряда и количество просуммированных членов. Сравнить получаемый результат со значением библиотечного метода Math.Sin( ). Учесть периодичность функции sin и, перед вычислением ряда, привести значение аргумента к диапазону [0; 2π).

Вычисление сходящегося ряда с точностью до машинного нуля продолжается пока прибавление (или вычитание) очередного члена изменяет накапливаемое значение суммы ряда.

```
//05_06.cs – вычисление значения функции sin(x)
using System;
class Program  {
public static void Main()  {
      double angle,  // Введенный угол в радианах
      x,             // Аргумент x (приведенный угол)
      sin,           // Сумма ряда (текущая)
      sinOld,        // Сумма ряда (предыдущая)
      memb;          // Очередной член ряда
      do {           // Цикл повторения решений
      do Console.Write("Введите значение угла (в радианах): ");
      while (double.TryParse(Console.ReadLine(), out angle) == false);
      x = angle % (2*Math.PI);
      Console.WriteLine("Приведенный угол x = {0}",x);
      int m; // число просуммированных членов ряда
      for(m = 1, sin = memb = x, sinOld = 0;
            sin != sinOld; m++)   { // Цикл вычисления ряда
      Console.WriteLine("sin({0:F5})={1:G5}  \tmemb = {2:G5}",
            x, sin, memb);
      sinOld = sin;
      memb *= - x*x/2/m/(2*m+1);
      sin +=memb;
      }
      Console.WriteLine("Число членов: {0}, отброшен член: {1}",
            m-1, memb);
      Console.WriteLine("Результат: sin({0}) = {1}", angle, sin);
      Console.WriteLine("Библиотечный метод: Math.Sin({0}) = {1}",
            angle, Math.Sin(angle));
      Console.WriteLine("Для выхода из программы нажмите ESC.");
      } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} //Конец метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Введите значение угла (в радианах): 32
Приведенный угол x = 0,584073464102069
sin(0,58407)=0,58407   memb = 0,58407
sin(0,58407)=0,55086   memb = -0,033209
sin(0,58407)=0,55143   memb = 0,00056644
sin(0,58407)=0,55143   memb = -4,6009E-06
sin(0,58407)=0,55143   memb = 2,1799E-08
sin(0,58407)=0,55143   memb = -6,7606E-11
sin(0,58407)=0,55143   memb = 1,4784E-13
sin(0,58407)=0,55143   memb = -2,4017E-16
Число членов: 8, отброшен член: 3,01216369918166E-19
Результат: sin(32) = 0,551426681241692
Библиотечный метод: Math.Sin(32) = 0,551426681241691
Для выхода из программы нажмите ESC.
<ESC>
```

Для приведения введенного значения угла (переменная *angle*) к диапазону [0; 2π) в программе использован оператор:

```
x = angle % (2*Math.PI);
```

Для завершения суммирования членов ряда необходимо, чтобы сумма осталась неизменной после добавления очередного члена. Это условие достижимо, так как модули членов ряда монотонно убывают, а точность представления вещественных чисел в компьютере конечная. Для проверки выполнения указанного условия в программе кроме переменной sin, накапливающей текущее значение частной суммы ряда, определена переменная sinOld, сохраняющее значение предыдущей суммы. Проверка условия необходимости продолжать суммирование выполняется в заголовке цикла:

```
for(m = 1, sin = memb = x, sinOld = 0; sin != sinOld; m++).
```

Параметром цикла служит переменная m – номер очередного члена ряда. Обратите внимание, что эта переменная объявлена вне цикла, так как ее значение нужно знать после завершения цикла. Еще одна особенность этого цикла – его инициализатор, список выражений которого выполняет действия первого шага суммирования.

При вычислении значений членов ряда, определяемых выражением, $(-1)^m \frac{x^{2m+1}}{(2m+1)!}$, важно не вычислять напрямую для

каждого из них степень аргумента x и факториал в знаменателе. Для получения очередного члена нужно предыдущий член умножить на $-x^2$ и разделить на (2*m+1)*2*m. В программе для представления очередного члена ряда используется вещественная переменная memb.

В результатах выполнения программы обратим внимание, что значения некоторых вещественных переменных специально выводятся без форматирования, то есть выводятся все цифры чисел типа **double**. Можно отметить в результатах весьма хорошее совпадение полученной суммы ряда со значением, возвращаемым библиотечным методом Math.Sin( ).

Задача 05-07. Существуют многочисленные способы вычисления приближенного значения числа π. Один из них (не самый лучший) – статистический метод, программная реализация которого на языке Фортран описана в книге Дрейфус М., Ганглоф К.[9].

Для решения задачи выбирается четверть круга с центром в начале координат и единичным радиусом, размещенная в первом квадранте системы координат (рис. 5.1). Около этой четверти круга описан квадрат. Для всех случайно выбираемых точек квадрата с координатами (x, y) выполняются неравенства: $0 \le x < 1$ и $0 \le y < 1$. Если распределение точек внутри квадрата равномерно, то число точек, попавших внутрь четверти круга, пропорционально π/4. Выполняя достаточно большое количество испытаний и имея хороший датчик равномерно распределенных случайных чисел, можно, проводя разное количество испытаний, с разной точностью оценивать значение числа π.

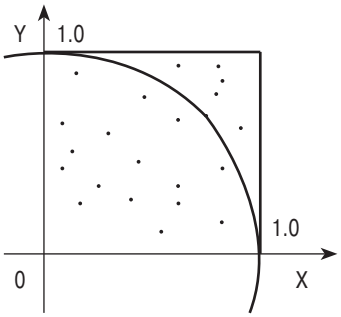


Рис. 5.1. Схема случайных испытаний к задаче 5-07

Для получения случайных чисел из диапазона [0,1) используем метод NextDouble() объекта класса Random. (См. Приложение 4.) Будем вычислять приближенные значения числа π с вводимой пользователем точностью. Вычисления и оценку точности будем выполнять после каждой 1000 испытаний. Для наблюдения за процессом после каждых 100000 испытаний выведем количество проверенных точек, оценку значения π и разность между очередным и предыдущим значениями. Цикл испытаний завершим по достижению заданной пользователем точности ε или когда число испытаний достигнет предельного значения, установленного для целочисленных значений (**int.** MaxValue). Для оценки достигнутой точности будем сравнивать абсолютное значение разности текущего и предыдущего значений π с введенной пользователем точностью.

Программа, решающая задачу:

```
//05_07.cs – вычисление значения числа пи
using System;
class Program  {
public static void Main()  {
    uint numb,          // Число испытаний
    into;               // Количество точек внутри четверти круга
    double x, y;        // Координаты точки в квадрате
    double eps;         // Требуемая точность
    double piNew,       // Очередное приближение
    piOld;              // Предыдущее приближение
    do {                // Цикл повторения решений
    do Console.Write("Введите требуемую точность: ");
    while (double.TryParse(Console.ReadLine(), out eps) == false ||
                eps <= 0);
    Random gen = new Random(); // Объект – генератор случайных чисел
    into = 0;
    piNew = piOld = 0;
    string format = "numb={0,-10} pi={1,-10:g5} eps={2,-6:g5}";
    Console.WriteLine("\tПромежуточные результаты: ");
    for (numb = 1; numb < int.MaxValue; numb++)  {
    x = gen.NextDouble();
    y = gen.NextDouble();
    if (x*x + y*y < 1) into++;
    if (numb % 1000 == 0) {
    piOld = piNew;  // Запомнить предыдущее значение
    piNew = (double)into/numb*4;  // Новое значение
    if (Math.Abs(piNew - piOld) < eps) break;
```

```
    }
    if (numb % 100000 == 0)          // Промежуточная печать
    Console.WriteLine(format, numb, piNew, piNew-piOld);
    Console.WriteLine("Итоговый результат:");
    Console.WriteLine(format, numb, piNew, piNew-piOld);
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} //Конец метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Введите требуемую точность: 1E-5<ENTER>
        Промежуточные результаты:
numb=100000    pi=3,1439    eps=-0,00092808
Итоговый результат:
numb=101000    pi=3,1439    eps=1,1881e-06
Для выхода из программы нажмите ESC.
Введите требуемую точность: 1E-6<ENTER>
        Промежуточные результаты:
numb=100000    pi=3,1543    eps=0,00025939
numb=200000    pi=3,1477    eps=-7,8693e-05
numb=300000    pi=3,1454    eps=0,00011559
Итоговый результат:
numb=335000    pi=3,1437    eps=8,9373e-07
Для выхода из программы нажмите ESC.
<ESC>
```

Для генерации случайных чисел (то есть координат точек) в программе следующим образом определен объект класса Random:

```
Random gen = new Random();
```

При таком определении, то есть при использовании конструктора *Random()* без аргумента, последовательности создаваемых случайных значений разные при каждом выполнении программы.

Генерация случайных координат точек и все вычисления выполняются в цикле с заголовком:

```
for (numb = 1; numb < int.MaxValue; numb++)
```

После получения очередных координат точки в следующем операторе проверяется ее принадлежность кругу:

```
if (x*x + y*y < 1) into++;
```

Здесь into – счетчик точек в четверти круга с радиусом 1. При numb, кратном 1000, выполняются операторы, включенные в следующий условный оператор:

```
if (numb % 1000 == 0) {
  piOld = piNew;  // Запомнить предыдущее значение
  piNew = (double)into/numb*4;   // Новое  значение
  if (Math.Abs(piNew - piOld) < eps) break;
}
```

Именно при выполнении этих операторов вычисляется очередное приближенное значение числа π (переменная piNew) и выполняется его сравнение с предыдущим приближением (переменная piOld).

Из приведенных результатов выполнения программы видно, что точность такого вычисления значения числа π невысокая. Объясняется это и выбранным алгоритмом и, возможно, недостаточной равномерностью распределения случайных чисел.

Задача 05-08. Вывести на экран таблицу истинности логической функции трех переменных: F = (a || !b) && c.

Для решения задачи нужно вычислять значения функции F для всех возможных сочетаний значений логических переменных a, b, c. Каждая из логических переменных может иметь два значения **true** и **false**. Перебирать значения переменных можно с помощью трех вложенных циклов. Наиболее удобно применить оператор цикла с постусловием.

Программа, решающая задачу:

```
//05_08.cs - таблицу истинности логической функции
using System;
class Program {
static void Main()      {
        bool ba=true, // Значение первой  переменной
        bb,           // Значение второй  переменной
        bc,           // Значение третьей переменной
        bf;           // Значение функции
        Console.WriteLine(" A \t B \t C \t F");
        do { bb = true;
          do { bc = true;
             do {
                  bf = (ba | !bb) &bc;    // вычисление значения функции
                  Console.WriteLine("{0}\t{1}\t{2}\t{3}", ba, bb, bc, bf);
```

```
                bc = !bc;
              } while (!bc);
              bb = !bb;
            } while (!bb);
            ba = !ba;
          } while(!ba);
        Console.WriteLine("Для выхода из программы нажмите ENTER.");
        Console.ReadLine();
  }
}
```

Результаты выполнения программы:

```
A       B       C       F
True    True    True    True
True    True    False   False
True    False   True    True
True    False   False   False
False   True    True    False
False   True    False   False
False   False   True    True
False   False   False   False
Для выхода из программы нажмите ENTER.
<ENTER>
```

Задача 05-09. Ввести положительное целое число и вывести его цифры, разделив их знаками подчеркивания.

Если бы нужно было вывести цифры числа в обратном порядке, то программа была бы совсем простой. В этом случае все действия свелись бы к последовательному получению остатка от деления числа на 10 (операция %) и целочисленному уменьшению числа в 10 раз (делению числа на 10).

Для получения цифр числа в естественном порядке проще всего применить рекурсивный метод, но методами мы займёмся в теме 7, поэтому оценим порядок введенного числа и получим делитель – максимальную степень числа 10, не превышающую введенного числа. Например, для числа 853 таким делителем будет 100. Дальше все просто – при целочисленном делении результатом будет первая цифра числа. Вычитая из числа произведение делителя на первую цифру, получим остаток. Делим этот остаток на уменьшенный в 10 раз делитель – получаем следующую цифру и т.д.

Программа, решающая задачу:

```
// 05_09.cs – цифры целого числа
using System;
class Program  {
public static void Main()  {
        uint numb,        // Введенное число
        top,              // степень 10, не превышающая numb
        n,                // вспомогательная переменная
        fig;              // очередная цифра числа
        do {              // Цикл повторения решений
    do Console.Write("Введите положительное целое: ");
    while (uint.TryParse(Console.ReadLine(), out numb) == false);
    top = 1;
    while(top * 10 <= numb) top *= 10;
    n = numb;
    do {
    fig = n/top;
    n -= fig * top;
    top /= 10;
    Console.Write(fig);
    if(top != 0) Console.Write("_");
    } while (top != 0);
    Console.WriteLine("\nДля выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} //Конец метода Main()
} //Конец объявления класса Program
```

Результаты выполнения программы:

```
Введите положительное целое: 0<ENTER>
0
Для выхода из программы нажмите ESC.
Введите положительное целое: 10<ENTER>
1_0
Для выхода из программы нажмите ESC.
Введите положительное целое: 400320<ENTER>
4_0_0_3_2_0
Для выхода из программы нажмите ESC.
<ESC>
```

В данной программе предполагается, что в представлении числа нет ведущих (левых) нулей (точнее – левые нули не переносятся в изображении числа). Однако для нулевого числа сделано исключение.

ТЕМА 06

МАССИВЫ, СТРОКИ, ПЕРЕКЛЮЧАТЕЛИ

Задача 06-01. Ввести размер массива **N** и значения его элементов. Нормировать элементы массива, разделив их на значение максимального по модулю элемента. Вывести значения элементов измененного массива.

```
// 06_01.cs – нормирование элементов массива
using System;
class Program {
static void Main()  {
      Console.Title = "Нормирование элементов массива";
      uint N;    // размер массива
      do      {   // цикл для повторения решений задачи
      do Console.Write("Введите размер массива: ");
      while (!uint.TryParse(Console.ReadLine(), out N)|| N ==0);
      double[] doubleAr = new double[N];  // Массив чисел
      double elemMax = 0; // Максимальный по модулю элемент
      for (int i = 0; i < N; i++) // Ввод и выбор максимального
          {
          do Console.Write("Введите значение doubleAr[{0}]: ", i);
          while (!double.TryParse(Console.ReadLine(), out doubleAr[i]));
          if (Math.Abs(elemMax) < Math.Abs(doubleAr[i]))
          elemMax =  doubleAr[i];
          }
      if (elemMax == 0)    { // Если все элементы нулевые
      Console.WriteLine("Все элементы нулевые!!!");
      Console.WriteLine("Для выхода из программы нажмите ESC ");
      continue;
      }
      for (int i = 0; i < N; i++) // Нормируем элементы
            doubleAr[i] /= elemMax;
      Console.WriteLine("Нормированный массив:");
      for (int i = 0; i < N; i++)
            Console.Write("{0:f3}\t",doubleAr[i]);
      Console.WriteLine();
```

```
    Console.WriteLine("Для выхода из программы нажмите ESC");
     } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите размер массива: 3<ENTER>
Введите значение doubleAr[0]: 1<ENTER>
Введите значение doubleAr[1]: 2<ENTER>
Введите значение doubleAr[2]: 3<ENTER>
Нормированный массив:
0,333  0,667  1,000
Для выхода из программы нажмите ESC
<ESC>
```

Программа 06_01.cs демонстрирует: создание массива, количество элементов которого определяется пользователем в ходе выполнения программы, ввод значений элементов массива и традиционные средства их обработки.

Задача 06-02. Определить и инициализировать целочисленный массив из 10-ти элементов. Ввести целое число и заменить им значение максимального элемента в массиве.

```
// 06_02.cs – поиск и изменение элемента в массиве
using System;
class Program {
static void Main()  {
      int[] arInt = {22, 5, 12, 63, -6, -52, 77, 41, 35, 23};
      int numb;
      Console.Title = "Инициализация  массива";
      do   { // цикл для повторения ввода числа
         do Console.Write("Введите целое число: ");
         while (!int.TryParse(Console.ReadLine(), out numb));
      int index = 0; // индекс максимального элемента массива
      for (int i = 0; i < arInt.Length; i++) // поиск
      if (arInt[i] > arInt[index]) index = i;
Console.WriteLine("Заменяем arInt[{0}]={1} на {2}",
      index, arInt[index], numb);
arInt[index] = numb;   // замена значения
for (int i = 0; i < arInt.Length; i++) // вывод массива
      Console.Write(arInt[i] + "\t");
Console.WriteLine();
Console.WriteLine("Для выхода из программы нажмите ESC ");
} while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите целое число: 34<ENTER>
Заменяем arInt[6]=77 на 34
22   5   12   63   -6   -52   34   41   35
Для выхода из программы нажмите ESC
Введите целое число: 22<ENTER>
Заменяем arInt[3]=63 на 22
22   5   12   22   -6   -52   34   41   35
Для выхода из программы нажмите ESC
<ESC>
```

Программа 06_02.cs демонстрирует: создание массива с инициализацией; поиск индекса элемента с нужным свойством; и обычные приемы обработки массивов.

Задача 06-03. Вычислить K простых чисел. Значение K ввести с клавиатуры. Вывести значения чисел, размещая их по 10 на строке.

Первые три простых числа 1, 2, 3 будем считать известными и предложим пользователю вводить значение K большее трех. Для сохранения найденных простых чисел определим массив из K элементов. Первым трем элементам явно присвоим значения 1, 2, 3. Кандидатом на роль очередного простого числа будет увеличенное на 2 последнее (самое большое) уже найденное простое число. Если оно не делится ни на одно меньшее его простое, то это очередное простое число, значение которого заносится в массив.

```
// 06_03.cs – массив простых чисел
using System;
class Program {
static void Main()  {
    uint K;                  // Количество простых чисел
    uint candidate;          // число-претендент на простое
    Console.Title = "Массив простых чисел";
    do   // цикл для повторения решений задачи
       {  // Конструкция для ввода значения:
      do Console.Write("Введите натуральное число, большее трех: ");
      while (!uint.TryParse(Console.ReadLine(), out K)
            || K <= 3); // Первые три простых известны
      uint[] numbers = new uint[K];  // Массив простых чисел
      numbers[0] = 1;
      numbers[1] = 2;
      numbers[2] = 3;
```

```
    for (int i = 3; i < K; i++)  {
        candidate = numbers[i - 1] + 2;
        for (int j = 2; j < i; j++)
            if (candidate % numbers[j] == 0)  {
                candidate += 2;
                j = 2;  // изменяем параметр цикла
            }
          numbers[i] = candidate;
        }
    for (int i = 0; i < K; i++)
       if ((i + 1) % 10 != 0)
            Console.Write(numbers[i] + "\t");
       else Console.WriteLine(numbers[i]);
            Console.WriteLine();
       Console.WriteLine("Для выхода из программы нажмите ESC ");
        } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
    }
}
```

Результаты выполнения программы:

```
Введите натуральное число, большее трех: 8<ENTER>
1    2    3    5    7    11    13    17
Для выхода из программы нажмите ESC
<ESC>
```

В программе стоит обратить внимание на цикл поиска очередного простого числа:

```
for (int j = 2; j < i; j++)
    if (candidate % numbers[j] == 0)  {
        candidate += 2;
        j = 2;  // изменяем параметр цикла
    }
```

Особенность цикла в том, что количество итераций заранее не определено и зависит от значения переменной candidate, и от уже известных элементов массива numbers[ ]. Если переменная candidate делится без остатка на значение элемента numbers[j], то гипотеза о простоте значения переменной candidate отвергается. Претендентом на роль следующего простого числа становится значение candidate+2, и цикл перебора уже найденных простых чисел начинается с начала (параметру цикла – переменной j присваивается значение 2). Цикл завершается только в том случае, если значение переменной candidate не делится нацело ни на одно из значений уже найденных простых чисел (начальных элементов массива).

Задача 06-04. Определить целочисленный массив из K элементов. Присвоить элементам случайные значения из диапазона [A,B). Найти индексы минимального и максимального элементов массива. Вывести значения элементов, расположенных между найденными (включая найденные).

Предположим, что размер массива (значение переменной K) пользователь будет вводить с клавиатуры. Границы диапазона случайных значений определим в коде программы, введя целочисленные константы A = –50 и B = 50. Тогда программа может быть такой:

```
// 06_04.cs – "вырезка" из массива элементов от макс. до мин.
using System;
class Program  {
  static void Main()  {
      uint K;                                  // размер массива
      int[] row;                               // ссылка на массив
      Random gen = newRandom();                // датчик случайных чисел
      const int A = -50, B = 50;               // предельные значения чисел
      do  {  // цикл для повторения решения
         do Console.Write("Введите размер массива (K>1): ");
      while (!uint.TryParse(Console.ReadLine(), out K)
                || K < 2);
      row = new int[K];                        // экземпляр массива
      int iMax = 0, iMin = 0;                  // индексы искомых значений
        // Заполняем массив и оцениваем значения его элементов:
      for (int i = 0; i < row.Length; i++) {
          row[i] = gen.Next(A, B);
          iMax = row[i] > row[iMax] ? i : iMax;
          iMin = row[i] < row[iMin] ? i : iMin;
        }
      Console.WriteLine("Элементы массива:");
      foreach (int memb in row)
          Console.Write(" {0,4} ", memb);
      Console.WriteLine("\nВыделенные элементы:");
      int jMin = Math.Min(iMax, iMin),
          jMax = Math.Max(iMin, iMax);
      for (int j = jMin; j <= jMax; j++)
          Console.Write(" {0,4} ", row[j]);
      Console.WriteLine("\nДля выхода из программы нажмите ESC ");
      } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
  }
}
```

Результаты выполнения программы:

```
Введите размер массива (K>1): 7<ENTER>
Элементы массива:
 –19  28  43  13  –33  19  –12
Выделенные элементы:
  43  13  –33
Для выхода из программы нажмите ESC
<ESC>
```

Задача 06-05. Определить символьный массив из K элементов. Присвоить элементам случайные значения букв русского алфавита. Создать новый массив, поместив в него только согласные буквы из первого массива. Значение K ввести с клавиатуры. Вывести элементы обоих массивов.

```
// 06_05.cs – выбор согласных русских букв из массива
using System;
class Program  {
static void Main()  {
    uint K;  // размер  первого массива
    char[] line, newLine; // ссылки на массивы
    char[] vowel = { 'а','е','ё','и','й','о','у', 'ы','э','ю','я' };  // гласные
    Array.Sort(vowel);  // упорядочивание символов
    Random gen = new Random( );  // датчик случайных чисел
    do  {   // цикл для повторения решения
       do Console.Write("Введите размер массива (K>1): ");
       while (!uint.TryParse(Console.ReadLine(), out K)|| K < 2);
       line = new char[K]; // экземпляр  массива
           // Заполняем  массив:
       for (int i = 0; i < line.Length; i++)
           line[i] = (char)gen.Next('а', 'я' + 1);
       int numb = 0;   // количество гласных
       Console.WriteLine("Первый массив:");
       for (int n = 0, res; n < line.Length; n++) {
           Console.Write(" {0,3} ", line[n]);
           res = Array.BinarySearch(vowel, line[n]);
           if (res >= 0) { numb++; line[n] = '*'; }
       }
       newLine = new char[line.Length - numb];
       int j = 0;  // счетчик  элементов
       foreach (char ch in line)
           if (ch != '*') newLine[j++] = ch;
       Console.WriteLine("\n Получен массив: ");
       foreach (char memb in newLine)
           Console.Write(" {0,3} ", memb);
```

```
        Console.WriteLine("\n Для выхода из программы нажмите ESC ");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите размер массива (K>1): 7<ENTER>
Первый массив:
  ш  м  е  я  г  й  щ
Получен массив:
  ш  м  г  щ
Для выхода из программы нажмите ESC
<ESC>
```

В программе использованы несколько символьных массивов и методы класса Array. Ссылки line и newLine представляют, соответственно, исходный массив и формируемый массив согласных букв. Для заполнения исходного массива из K элементов используется метод Next( ) объекта класса Random. Очередной элемент массива получает значение при выполнении оператора:

```
line[i] = (char)gen.Next('а', 'я' + 1);
```

Предельные значения получаемого результата определены кодами символов 'а' и 'я'. Так как метод Next( ) возвращает целое число, то выполнено приведение типов с помощью явной операции (**char**).

Для определения количества согласных букв в массиве line[ ] из него "удаляются" все гласные буквы и подсчитывается их количество (переменная numb). Основой этих действий служит оператор

```
res = Array.BinarySearch(vowel, line[n]);
```

Первый аргумент метода BinarySearch( ) – ссылка на вспомогательный массив, элементам которого присвоены значения гласных букв русского алфавита. Второй аргумент – очередной элемент (символ) исходного массива. Метод выполняет поиск в массиве значения второго аргумента. Возвращаемое значение – индекс найденного элемента, либо –1, если искомый элемент в массиве отсутствует. Метод BinarySearch( ) применим только в тех случаях, кода элементы массива упорядочены. Для соблюдения этого условия вначале определен вспомогательный массив гласных букв:

```
char[] vowel = { 'а','е','ё','и','й','о','у', 'ы','э','ю','я' };
```

Предположив, что никаких сведений об упорядоченности числовых значений символов, представляющих русские буквы, у нас нет, то выполняем упорядочение массива:

```
Array.Sort(vowel);
```

Если в результате выполнения оператора res = Array. BinarySearch(vowel, line[n]); переменная res получает неотрицательное значение, то найдена гласная русская буква. Чтобы при повторных обращениях эта буква более не учитывалась, соответствующему элементу массива присваивается условное значение: line[n]='*';

Остальное просто. Создается новый массив:

```
newLine = new char[line.Length - numb];
```

и его элементам присваиваются отличные от символа '*' значения из массива line[].

Задача 06-06. Определить вещественный массив из 10-ти элементов. Присвоить элементам случайные значения из диапазона [–10, 10). Сформировать массив индексов, которые нумеруют элементы первого массива в порядке возрастания их значений.

Цель этой задачи – обратить внимание на возможность с помощью косвенной адресации (косвенного индексирования), не меняя местоположения элементов массива, перебирать элементы массива в нужном порядке. Создав несколько индексных массивов, можно с их помощью по разным правилам получать доступ к элементам одного и того же неизменяемого массива. В нашем случае нужен только один индексный массив.

Программа, решающая задачу:

```
// 06_06.cs – сортировка массива с помощью косвенной индексации
using System;
class Program  {
static void Main()  {
    double[] arX = new double[10];          // первый массив
    int[] index = new int[arX.Length];      // массив индексов
    Random gen = new Random( );             // генератор случайных чисел
    const int minX = -10, maxX = 10;        // предельные значения
    do  { // цикл для повторения решения
```

```
    Console.WriteLine("Вещественный массив:");
    for (int i = 0; i < arX.Length; i++)  {
            arX[i] = gen.Next(minX, maxX) + gen.NextDouble();
            Console.Write("[{0}]={1:f3}\t", i, arX[i]);
            index[i] = i;   // инициализация массива индексов
            }
    // Сортировка  массива  индексов:
    for (int i = 0; i < arX.Length - 1; i++)
        for (int j = i, temp; j < arX.Length; j++)
            if (arX[index[i]] > arX[index[j]])  {
                temp = index[i]; index[i] = index[j]; index[j] = temp;
                }
    Console.WriteLine("Косвенное обращение: ");
    foreach (int ind in index)
                Console.Write("[{0}]={1:f3}\t", ind, arX[ind]);
    Console.WriteLine("Для выхода из программы нажмите ESC ");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Вещественный массив:
[0]=9,752    [1]=5,693    [2]=4,298    [3]=-2,541    [4]=-5,039
[5]=3,274    [6]=0,672    [7]=9,276    [8]=-8,805    [9]=8,187
Косвенное обращение:
[8]=-8,805    [4]=-5,039    [3]=-2,541    [6]=0,672    [5]=3,274
[2]=4,298    [1]=5,693    [9]=8,187    [7]=9,276    [0]=9,752
Для выхода из программы нажмите ESC
<ESC>
```

В программе два массива: основной **double**[] arX = **new double**[10] и вспомогательный массив индексов **int**[] index = **new int**[arX.Length]. Первый получает случайные значения элементов, второй – значения индексов первого массива. Далее во вложенных циклах сравниваются значения элементов первого массива, индексируемые значениями элементов второго:

```
if (arX[index[i]] > arX[index[j]])
```

Если в результате сравнения обнаружено нарушение порядка, то меняются местами значения элементов индексирующего массива. Последующий опосредованный перебор элементов основного массива (при прямом переборе массива индексов) обеспечивает вывод их значений в порядке возрастания. (См. результаты работы программы.)

Задача 06-07. Определить целочисленные массивы A и B из K элементов в каждом. Присвоить элементам массивов случайные значения из диапазона [1,10). Добавить в конец массива A те элементы из массива B, которых нет в A.

Цель этой задачи – обратить внимание на невозможность изменять размеры уже определенного массива. То есть массив не может расти за счет добавления к нему новых элементов. Поэтому для решения задачи определим в программе вспомогательный массив заведомо больших размеров, нежели нужно для результата. Поместим в него все элементы массива A. После них запишем подходящие элементы из B. Зная общее число элементов, помещенных во вспомогательный массив, определим новый массив и копируем в него начало (заполненную часть) вспомогательного массива. Этот последний массив и будет результатом.

Программа, решающая задачу:

```
// 06_07.cs – добавление в одномерный массив новых элементов
using System;
class Program {
static void Main()  {
    int K;                                  // размер массивов
    int[] A, B;                             // ссылки на массивы
    Random gen = new Random( );   // датчик случайных чисел
    const int min = 1, max = 10;        // предельные значения
    do  { // цикл для повторения решения
       do Console.Write("Введите размер массива (K>1): ");
    while (!int.TryParse(Console.ReadLine(), out K) || K < 2);
    A = new int[K];                         // экземпляр  массива
    B = new int[K];
        // Заполняем массивы:
    for (int i = 0; i < K; i++)  {
        A[i] = gen.Next(min, max);
        B[i] = gen.Next(min, max);
        }
    Console.WriteLine("Элементы массива A:");
    foreach (int memb in A)
        Console.Write(" {0,2} ", memb);
    Console.WriteLine("\nЭлементы массива B:");
    foreach (int memb in B)
        Console.Write(" {0,2} ", memb);
    int[] tempAr = new int[2 * K];          // Вспомогательный  массив
    int ind = K - 1;
    Array.Copy(A, tempAr, K);  //
     for (int i = 0; i < K; i++) {                  // перебираем элементы из B
```

```
        bool flag = false;  //
        for (int j = 0; j < K; j++)              // перебираем элементы из A
          if (B[i] == A[j])
               { flag = true; break; }
          if (flag) continue;
        tempAr[++ind] = B[i];
        }
      A = new int[ind + 1];
      Array.Copy(tempAr, A, ind + 1);
      Console.WriteLine("\nЭлементы массива tempAr:");
      foreach (int memb in tempAr)
                Console.Write(" {0,2} ", memb);
      Console.WriteLine("\nЭлементы нового массива A:");
      foreach (int memb in A)
                Console.Write(" {0,2} ", memb);
      Console.WriteLine("\n Для выхода из программы нажмите ESC ");
      } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите размер массива (K>1): 7<ENTER>
Элементы массива A:
 3  5  6  4  1  1  3
Элементы массива B:
 1  7  4  9  6  3  9
Элементы массива tempAr:
 3  5  6  4  1  1  3  7  9  9  0  0  0  0
Элементы нового массива A:
 3  5  6  4  1  1  3  7  9  9
Для выхода из программы нажмите ESC
<ESC>
```

В программе для "переноса" элементов массива A[ ] во вспомогательный массив tempAr[ ] использовано обращение к методу Array.Copy(A, tempAr, K). Первый аргумент – источник данных, второй – цель (мишень) копирования, третий – количество копируемых элементов. Тот же метод затем используется для копирования значений элементов из вспомогательного массива в массив-результат:

```
A = new int[ind + 1];
Array.Copy(tempAr, A, ind + 1);
```

Обратите внимание, что массив-результат представлен в программе переменной-ссылкой A, которая вначале адресовала

исходный массив. Исходный массив (объект типа **int**[ ] из K элементов) теперь в программе недоступен и будет удален сборщиком мусора.

Задача 06-08. Построить двумерный целочисленный массив с размерами 5 на 5 (квадратную матрицу) Элементам массива присвоить случайные значения от –50 до +50. Удалить из матрицы строку и столбец, на пересечении которых находится элемент с максимальным значением. (Если таких элементов несколько – выбрать любой.) При каждом повторении решения матрица уменьшается («сжимается»). Окончание обработки – вырожденная матрица из одного элемента.

При решении этой задачи проиллюстрируем особенности изменения размеров многомерных массивов.

```
// 06_08.cs – принудительное изменение размеров матрицы
using System;
class Program  {
static void Main() {
    uint N = 5;                          // размеры массива (матрицы)
    int[,] matr = new int[N, N];         // исходный  массив
    Random gen = new Random();   // датчик случайных чисел
    // Заполняем и печатаем матрицу:
    Console.WriteLine("Исходная матрица:");
    for (int i = 0; i < N; i++, Console.WriteLine())
      for (int j = 0; j < N; j++) {
          matr[i, j] = gen.Next(-50, 51);
          Console.Write("{0,4} ", matr[i, j]);
          }
  do  { // цикл уменьшения размеров матрицы
    int max = matr[0, 0];   //
    int xMax = 0, yMax = 0; //
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
          if (matr[i, j] > max)
          { xMax = i; yMax = j; max = matr[i, j]; }
          // Новый массив для матрицы:
          int[,] newMatr = newint[N - 1, N - 1];
          Console.WriteLine("Максимальный: [{0},{1}]={2}",
                  xMax, yMax, max);
          // Заполняем и печатаем матрицу:
          Console.WriteLine("Измененная матрица:");
          for (int i = 0, inew = 0; i < N; i++)  {
             for (int j = 0, jnew = 0; j < N; j++)
              if (i != xMax && j != yMax)   {
```

```
                newMatr[inew, jnew] = matr[i, j];
                Console.Write("{0,4} ", newMatr[inew, jnew++]);
                }
            if (i != xMax) { inew++; Console.WriteLine(); }
            }
        matr = newMatr;   // "Забываем" предыдущий массив
        N--;              // Размеры нового массива
        if (N == 1)  break;  // Матрица "выродилась"
        Console.WriteLine("Для прерывания цикла: ESC");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
Исходная матрица:
 -7 -17 -39 -25  -6
-10  49 -30 -12   7
-46 -26 -36 -34  28
 46 -23 -50 -36  28
-33  13   5  23   5
Максимальный: [1,1]=49
Измененная матрица:
 -7 -39 -25  -6
-46 -36 -34  28
 46 -50 -36  28
-33   5  23   5
Для прерывания цикла: ESC
Максимальный: [2,0]=46
Измененная матрица:
 -39 -25  -6
-36 -34  28
  5  23   5
Для прерывания цикла: ESC
Максимальный: [1,2]=28
Измененная матрица:
 -39 -25
  5  23
Для прерывания цикла: ESC
Максимальный: [1,1]=23
Измененная матрица:
 -39
Для выхода из программы нажмите ENTER.
<ENTER>
```

В тексте программы обратим внимание на некоторые детали. Во-первых, для представления матрицы определяются объект типа **int**[,] и адресующая его ссылка:

```
int[,] matr = new int[N, N];
```

Для заполнения и одновременного вывода на экран значений элементов матрицы используются вложенные циклы, заголовок внешнего из которых имеет вид:

```
for (int i = 0; i < N; i++, Console.WriteLine())
```

Его особенность – обращение к методу Console.WriteLine() в завершающем выражении заголовка. Так как параметр этого цикла служит первым индексом при обращениях к элементам массива, то перед выводом каждой следующей строки матрицы выполняется переход на новую строку печатающего устройства.

Вложенные циклы затем применяются для поиска максимального элемента матрицы и для заполнения нового массива:

```
int[,] newMatr = new int[N - 1, N - 1];
```

После того, как создан и заполнен новый массив, ссылке matr, представляющей исходный массив, присваивается новое значение:

```
matr = newMatr;
```

Тем самым предыдущий массив “забыт” и на следующей итерации цикла **do** выполняется обработка матрицы меньших размеров.

Задача 06-09. Упорядочить по убыванию цифры натурального числа, вводимого с клавиатуры.

Эту задачу можно решать по-разному. Воспользуемся тем фактом, что число вводится как символьная строка, и применим методы библиотечных классов String и Array. Тогда последовательность действий может быть такой. Читаем строку с изображением числа; удаляем из нее возможно присутствующие пробелы слева и справа; проверяем отсутствие символов, отличных от цифр; преобразуем строку в символьный массив; упорядочиваем по возрастанию элементы массива; реверсируем элементы массива (меняем на обратный порядок элементов); формируем из массива строку и выводим ее как результат.

```
// 06_09.cs – упорядочить цифры (символы) натурального числа
using System;
class Program  {
static void Main() {
  string number, newNumber;
  char[] arNumb;
  string sample = "0123456789";
  do  { // цикл для повторения решения
    int t = 0;   // вспомогательная переменная
    do {         // цикл с проверкой ввода
       Console.Write("Введите натуральное число: ");
       number = Console.ReadLine();   // Вводим строку с числом
       number.Trim();  // убираем пробелы слева и справа
       t=1; // цикл  проверки  символов  строки:
       foreach (char ch in number)
          t = Math.Min(sample.IndexOf(ch), t);
       }
    while(t == -1);
    arNumb = number.ToCharArray();     // массив  из  строки
    Array.Sort(arNumb);                // сортировка по возрастанию
    Array.Reverse(arNumb);             // реверсирование
    newNumber = new string(arNumb);   // из массива в строку
    Console.WriteLine("Результат: "+newNumber);
    Console.WriteLine("Для выхода из программы нажмите ESC ");
} while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите натуральное число: 14952<ENTER>
Результат: 95421
Для выхода нажмите клавишу ESC
Введите натуральное число: 34E-3<ENTER>
Введите натуральное число: 00567<ENTER>
Результат: 76500
Для выхода из программы нажмите ESC
<ESC>
```

Комментарии в коде программы объясняют назначение операторов, но на один из них обратим особое внимание. Рассмотрим следующий оператор

```
t = Math.Min(sample.IndexOf(ch), t);
```

В строке sample набор символов, представляющих десятичные цифры. Метод IndexOf( ) позволяет найти в строке индекс

первого вхождения символа-аргумента. Метод возвращает -1, если в строке sample нет искомого символа. Это случится, когда символ-аргумент будет отличен от цифры. В противном случае возвращаемое значение неотрицательное. Если после проверки всех символов строки number значение переменной t окажется равным –1, то введенная строка записана неверно – это не целое число и ввод нужно повторить.

Задача 06-10. Рассчитать общее сопротивление электрической цепи из двух последовательно соединенных групп активных сопротивлений. В первой группе три параллельно соединенных сопротивления R1, R2, R3, во второй – два параллельно соединенных сопротивления R4, R5. Предусмотреть инициализацию значений всех сопротивлений (R1, .., R5) и явно вводить значения только тех, которые нужно изменить. Общее сопротивление цепи вычисляется по формуле: $R = \frac{1}{G_1} + \frac{1}{G_2}$, где G_1, G_2 – проводимости первой и второй групп: $G_1 = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}$, $G_2 = \frac{1}{R_4} + \frac{1}{R_5}$.

В основной формулировке задачи нет ничего сложного и нового – нужно по известным значениям пяти сопротивлений вычислить по заданным формулам общее сопротивление цепи. Но вторая часть задания (“явно вводить значения только тех, которые нужно изменить”) предполагает, что программа должна распознавать, какие именно значения сопротивлений изменил с клавиатуры пользователь при очередном вычислении.

Примем следующие решения. При определении переменных, представляющих в программе сопротивления, инициализируем их единичными значениями. Пользователь при каждом расчете должен ввести дополнительную информацию в виде одной строки. Если строка пустая (пользователь нажал клавишу ENTER), то вычисления выполняются для имеющихся значений сопротивлений. Для того чтобы изменить значение i-го сопротивления, в строку заносится конструкция вида: “Ri=числовое_значение”. Таких конструкций в строке может быть до пяти. Отделены они друг от друга пробелами. В программе нужно проанализировать введенную строку, определить какие сопротивления должны быть изменены, и присвоить соответствующим переменным новые значения. Далее выполня-

ются вычисления, программа сохраняет имеющиеся значения сопротивлений и предлагает пользователю при необходимости вновь изменить их значения.

Программа, решающая задачу:

```
// 06_10.cs – сопротивление электрической цепи
using System;
class Program {
static void Main( ) {
  double r1=1,r2=1,r3=1,r4=1,r5=1;  // Значения сопротивлений
  double r;     // Общее сопротивление цепи
  double g1, g2; // Проводимости параллельных групп
  do  { // цикл для повторения решений задачи
     Console.WriteLine("\tИсходные данные:");
     Console.WriteLine(
        "R1={0:F2}  R2={1:F2}  R3={2:F2}  R4={3:F2}  R5={4:F2}",
              r1,r2,r3,r4,r5);
     // Конструкция для ввода новых значений сопротивлений:
     string text =
        "\tНажмите ENTER для решения или измените данные.";
     Console.WriteLine(text);
     Console.Write("Для замены значения переменной Ri наберите: ");
     Console.WriteLine("\"Ri=значение\"");
     string input;      // вводимая строка
     string name;     // имя (обозначение) сопротивления
     string image;    // значение сопротивления в виде строки
     string[] data;    // массив пар "имя=значение"
     double value;    // числовое значение сопротивления
     while(true)  {    // цикл ввода изменений данных
       input = Console.ReadLine();
       // удалить пробелы в начале и в конце строки:
       input = input.Trim();
       if(input == "") break; // Ничего  не  введено
       int gap = 0; // индекс двойного пробела во входной строке
       // удаляем повторяющиеся пробелы:
       while((gap = input.IndexOf("  ",gap)) != -1)
              input = input.Remove(gap, 1);
      data = input.Split(' ');   // массив пар "имя=значение"
      for (int i=0; i<data.Length; i++) {
          if (data[i].IndexOf('=') < 0) {
           Console.WriteLine("Ошибка во входных данных! ");
           continue;
          }
          name = data[i].Split('=')[0];      // имя  сопротивления
          image = data[i].Split('=')[1];     // значение сопротивления
          if(!double.TryParse(image, out value))
                {Console.WriteLine("Ошибка ввода!"); break;}
```

```
            switch(name)  {
                case "R1": r1 = value; break;
                case "R2": r2 = value; break;
                case "R3": r3 = value; break;
                case "R4": r4 = value; break;
                case "R5": r5 = value; break;
                default:
                      Console.WriteLine(name + " – неизвестное имя!");
                      continue;
                } // end of switch(name)
          } // end of for (int i…
          Console.WriteLine("\tДанные для расчета:");
          Console.WriteLine(
                "R1={0:F2} R2={1:F2} R3={2:F2} R4={3:F2} R5={4:F2}",
                      r1,r2,r3,r4,r5);
          break;
        } // end of while(true)
        g1 = 1/r1 + 1/r2 + 1/r3;
        g2 = 1/r4 + 1/r5;
        r = 1/g1 + 1/g2;
        Console.WriteLine("Общее сопротивление цепи: {0:F4}", r);
        Console.WriteLine("Для выхода из программы нажмите ESC ");
        } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
    Исходные данные:
R1=1,00 R2=1,00 R3=1,00 R4=1,00 R5=1,00
    Нажмите ENTER для решения или измените данные.
Для замены значения переменной Ri наберите: "Ri=значение"
R3=6<ENTER>
    Данные для расчета:
R1=1,00 R2=1,00 R3=6,00 R4=1,00 R5=1,00
Общее сопротивление цепи: 0,9615
Для выхода из программы нажмите ESC
<ENTER>
    Исходные данные:
R1=1,00 R2=1,00 R3=6,00 R4=1,00 R5=1,00
    Нажмите ENTER для решения или измените данные.
Для замены значения переменной Ri наберите: "Ri=значение"
T6=8 R5=4 R1=7<ENTER>
T6 - неизвестное имя!
    Данные для расчета:
R1=7,00 R2=1,00 R3=6,00 R4=1,00 R5=4,00
Общее сопротивление цепи: 1,5636
Для выхода из программы нажмите ESC
<ESC>
```

Наиболее важная часть программы – анализ входной строки и изменение значений конкретных переменных, представляющих изменяемые сопротивления.

Первый шаг анализа – удаление из введенной строки пробелов до и после введенных данных, выполняет оператор

```
input = input.Trim();
```

Если строка пустая, то анализ завершен, выполняется оператор **break**;

В противном случае в цикле из входной строки удаляются повторяющиеся пробелы. Для этого в цикле с заголовком `while((gap = input.IndexOf(“ “,gap)) != -1)` в строке input разыскиваются все парные пробелы и один из них удаляется методом `input.Remove(gap,1)`. Между конструкциями “Ri=числовое_значение” остаются только одиночные пробелы. Это позволяет преобразовать строку входных данных в массив строк, каждый элемент которого – строка “Ri=числовое_значение”. Выполняет это преобразование оператор

```
data = input.Split(‘ ‘);
```

Далее в цикле выполняется разбор каждого элемента массива data. Во-первых, проверяется наличие символа **‘=’**. Если этот знак присваивания отсутствует, то элемент не учитывается и выполняется переход к анализу следующего элемента (выдается сообщение об ошибке и выполняется оператор **continue**;). При наличии знака присваивания (**‘=’**) он служит разделителем в операторах

```
name = data[i].Split(‘=’)[0];      // обозначение сопротивления
image = data[i].Split(‘=’)[1];     // значение сопротивления
```

Для получения числового значения из строки image используются операторы

```
if(!double.TryParse(image, out value))
{   Console.WriteLine(“Ошибка ввода!”); break;   }
```

Если в представлении числового значения ошибок нет, выполняется переключатель – оператор множественного ветвления.

```
switch(name)   {
        case “R1”: r1 = value; break;
        case “R2”: r2 = value; break;
        case “R3”: r3 = value; break;
```

```
        case "R4": r4 = value; break;
        case "R5": r5 = value; break;
        default: Console.WriteLine(name + " – неизвестное имя!");
                 continue;
    } // end of switch(name)
```

Каждая ветвь переключателя служит для изменения значения соответствующей переменной, представляющей одно из сопротивлений. В каждой такой ветви находится оператор **break;**, который передает управление за пределы переключателя. Ветвь, вводимая служебным словом **default**, играет иную роль – при неверном имени выполняется оператор **continue;**. Он передает управление на следующую итерацию цикла, то есть выполняется анализ следующей конструкции "Ri=числовое_значение".

Результаты выполнения программы дополняют приведённые объяснения.

Задача 06-11. Получить из целого числа с основанием 10 его строковое представление с основанием 16.

```
// 06_11.cs – запись числа с основанием 16
using System;
class Program {
static void Main( ) {
   uint numb;   // Вводимое десятичное число
   do { // цикл для повторения решения
   Console.Write("Введите натуральное число: ");
   while (!uint.TryParse(Console.ReadLine(), out numb));
   uint temp = numb;            // вспомогательная переменная
   string hexNumb = "";         // строка результата
   uint figure;                 // очередная цифра
   uint hex = 16;               // основание системы счисления
   while(temp > 0) {  // цикл преобразования числа
   figure = temp % hex;   // очередная младшая цифра
   if(figure < 10)
      hexNumb = figure.ToString()+hexNumb;
   else switch(figure) {
      case 10: hexNumb = "A" + hexNumb; break;
      case 11: hexNumb = "B" + hexNumb; break;
      case 12: hexNumb = "C" + hexNumb; break;
      case 13: hexNumb = "D" + hexNumb; break;
      case 14: hexNumb = "E" + hexNumb; break;
      case 15: hexNumb = "F" + hexNumb; break;
      }
      temp /= hex;
```

```
    } // end of while
      hexNumb = "0x" + hexNumb;
      Console.WriteLine("{0} ==> {1}", numb,hexNumb);
      Console.WriteLine("Для выхода из программы нажмите ESC ");
   } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите натуральное число: 0<ENTER>
0 ==> 0x
Для выхода из программы нажмите ESC
Введите натуральное число: 16<ENTER>
16 ==> 0x10
Для выхода из программы нажмите ESC
Введите натуральное число: 30<ENTER>
30 ==> 0x1E
Для выхода из программы нажмите ESC
<ESC>
```

В программе нет конструкций, которые не использовались в предыдущих программах. Комментарии в тексте кода и пример результатов выполнения программы достаточно красноречивы.

Задача 06-12. Построить двумерный символьный массив (матрицу) с размерами 6 на 10. Элементам массива присвоить случайные значения букв латинского и русского алфавитов. Сформировать массив символьных массивов с русскими буквами из строк матрицы. Вывести матрицу и массив массивов русских букв.

В задаче требуется построить массив одномерных массивов, длины которых заранее неизвестны и в общем случае все разные. Такие конструкции в литературе почему-то называют массивами «непрямоугольными», «ступенчатыми», «зубчатыми», «ломаными» (по-английски «jagged»). Однако синтаксически массив массивов это обычный массив, элементами которого являются ссылки на массивы. В свою очередь элементы-ссылки адресуют массивы одного и того же типа, но на размеры этих массивов никаких ограничений не налагается.

```
// 06_12.cs – массив символьных массивов («зубчатый» массив)
using System;
class Program {
static void Main()  {
```

```
  char[,] symbols;   // ссылка на массив-матрицу букв
  const uint row=6, col=10;  // размеры массива-матрицы
  char[][] russian;  // ссылка на массив массивов русских букв
  Random generator = new Random();   // датчик случайных чисел
  symbols = new char[row, col]; // объект-массив (матрица)
  // Массив для количеств русских букв в строках:
  uint[] rus = new uint[symbols.GetLength(0)];
  // Заполняем и печатаем матрицу:
  Console.WriteLine("Исходная матрица:");
  for (int i = 0; i <= symbols.GetUpperBound(0);
           i++, Console.WriteLine())
     for (int j = 0; j <= symbols.GetUpperBound(1); j++) {
        if (generator.Next(0, 2) == 0)  {
           symbols[i, j] = (char)generator.Next('a', 'я' + 1);
           rus[i]++;  // подсчет русских букв в строке
           }
        else
           symbols[i, j] = (char)generator.Next('a', 'z' + 1);
        Console.Write(symbols[i, j] + " ");
        }
    russian = new char [symbols.GetLength(0)][]; // массив ссылок
   // Формируем массив массивов букв из строк матрицы:
   for (int i = 0; i <= symbols.GetUpperBound(0); i++)  {
   russian[i] = new char[rus[i]]; // массив для букв строки
   for (int j = 0, k =0; j <= symbols.GetUpperBound(1); j++)
   if ('a' <= symbols[i,j] & symbols[i,j] <= 'я')
   russian[i][k++] = symbols[i,j];
   }
   // Вывод значений элементов массива массивов:
   Console.WriteLine("\nРезультаты выбора: ");
   for (int i = 0; i < rus.Length; i++)  {
      Console.Write("Строка {0}: ",i+1);
      foreach (char memb in russian[i])
         Console.Write(memb +"  ");
      Console.WriteLine();
    }
    Console.WriteLine("Для выхода из программы нажмите ENTER");
    Console.ReadLine( );
}
}
```

Результаты выполнения программы:

Исходная матрица:

м	*t*	*n*	*z*	*й*	*a*	*r*	*y*	*e*	*ф*
l	*f*	*г*	*g*	*x*	*q*	*p*	*л*	*c*	*l*
k	*э*	*ь*	*s*	*п*	*p*	*l*	*ы*	*o*	*v*
k	*п*	*j*	*в*	*d*	*п*	*ц*	*h*	*ч*	*d*
з	*г*	*я*	*k*	*e*	*y*	*d*	*k*	*x*	*a*
s	*ю*	*м*	*я*	*x*	*a*	*p*	*i*	*й*	*ч*

Результаты выбора:
Строка 1: м й у е ф
Строка 2: г л с
Строка 3: э ь п ы
Строка 4: п в п ц ч
Строка 5: з г я е х а
Строка 6: ю м я й ч
Для выхода из программы нажмите ENTER
<ENTER>

В программе матрица представлена двумерным массивом symbols[,]. Массив массивов с русскими буквами адресует ссылка **char**[][] russian. Для подсчета количества русских букв в строках матрицы введен вспомогательный массив, представляемый ссылкой **uint**[] rus. Значения его элементов определяют размеры одномерных массивов, содержащих русские буквы строк матрицы. Создание и инициализация массива массивов выполняется в два «этапа». Вначале определяется массив неинициализированных ссылок:

```
russian = new char [symbols.GetLength(0)][];
```

Затем в цикле формируются конкретные символьные массивы «нижнего» уровня:

```
russian[i] = new char[rus[i]];
```

Стоит обратить внимание на особенности индексации при присваивании элементам значений:

```
russian[i][k++] = symbols[i,j];
```

Задача 06-13. Поменять на обратный порядок слов предложения. Слова разделены пробелами, в конце предложения нет знака препинания.

Если бы в условии требовалось поменять на обратный порядок символов (букв) всего предложения, то для решения можно было бы воспользоваться такой схемой. С помощью нестатического метода ToCharArray( ) класса **string** преобразовать строку в символьный массив. Затем, используя полученный массив в качестве аргумента метода Array.Reverse( ), реверсировать элементы массива. И, наконец, с помощью конструктора класса **string** сформировать из преобразованного массива нужную строку, которая окажется зеркально симметричной для исходной.

Описанная схема могла бы служить основой для решения нашей задачи. Но возможен более простой подход. Создадим с помощью метода Split( ) из строки массив ссылок на строки, каждая из которых – отдельное слово предложения. Выполним с помощью метода Array.Reverse( ) реверсирование массива ссылок, а затем с помощью метода **string**.Join( ) объединим элементы массива в одну строку. В этой строке слова исходного предложения будут размещены в обратном порядке.

Применение метода Split( ) требует аккуратности. Метод делит строку на части по указанным в параметрах разделителям. В нашей задаче таким разделителем будет пробел. Если в обрабатываемой строке встретятся два подряд разделителя (пробела), то метод формирует пустую строку, которая никак не должна восприниматься как отдельное слово предложения. Поэтому до обработки строки методом Split( ) следует оставить между словами по одному пробелу.

Для сокращения текста программы строку с предложением определим непосредственно в коде как значение строковой переменной **string** sentence. С учетом указанных соглашений решение задачи может быть таким:

```
// 06_13.cs – обратный порядок слов предложения
using System;
class Program1 {
static void Main( ) {
   string sentence =       // исходное предложение:
   " поменять    на обратный порядок слов предложения ";
   // Удалить пробелы слева и справа:
   sentence = sentence.Trim();
   // Удалить повторяющиеся пробелы:
   int n=-1;
   while((n = sentence.IndexOf(' ', n+1)) != -1)
   if(sentence[n+1] == ' ')
   sentence = sentence.Remove(n--, 1);
   //Сформировать массив ссылок на строки - слова предложения:
   string [] words = sentence.Split(' ');
   // Повернуть массив ссылок на строки:
   Array.Reverse(words);
   // Сформировать результирующую строку:
   sentence = string.Join(" ",words);
   Console.WriteLine(sentence);
   Console.WriteLine("Для выхода из программы нажмите ENTER.");
   Console.ReadLine();
}
}
```

Результаты выполнения программы:

предложения слов порядок обратный на поменять
Для выхода из программы нажмите ENTER.
<ENTER>

Примечание: «Метод Split() дополнительно принимает перечисление StringSplitOptions, которое имеет опцию для удаления пустых элементов: это полезно, когда слова в строке разделяются несколькими разделителями» ([5], стр.227). Указанная возможность позволяет упростить программы 06_10 и 06_13, исключив из них операторы, заменяющие в строке несколько повторяющихся пробелов на один пробел.

ТЕМА 07

СТАТИЧЕСКИЕ МЕТОДЫ (МЕТОДЫ КЛАССОВ)

В языке C# можно определять методы-функции и методы-процедуры. Метод-процедура не возвращает в точку вызова значений (возвращает значение типа **void**) и не может использоваться в выражениях. Метод, играющий роль функции, возвращает значение отличное от **void**, и обращение к нему может служить операндом соответствующего выражения. Параметры и тех, и других методов могут иметь ссылочные типы и могут иметь типы значений. Возвращаемое методом-функцией значение может иметь тип значения и может иметь тип ссылки. Кроме того, в методе может быть предусмотрена передача соответствующих параметрам аргументов по ссылкам и по значениям. Перечисленные разнообразия методов создают весьма гибкие возможности их применения. Рассмотрим эти возможности на основе статических методов (методов классов). Напомним, что в языке C# методы существуют только как члены классов. В предыдущих темах каждая программа включала один класс с одним статическим методом, имеющим фиксированное имя Main(). Но статических методов в классе может быть несколько.

Задача 07-01. Определить функцию (статический метод) для вычисления наибольшего общего делителя двух целых натуральных чисел (Greatest Common Measure). В основной программе, используя функцию, сократить неотрицательную обыкновенную дробь. Дробь вводится с клавиатуры в виде неотрицательного числителя и положительного знаменателя.

```
// 07_01.cs. – вычисление наибольшего общего делителя
using System;
class Program{
// Метод для вычисления НОД:
static uint GCM(uint a, uint b)     {
while (a != b)
```

```
    if (a > b) a -= b;
    else b -= a;
    return a;
    }
static void Main( )    {
    uint num,              // Числитель  и
    den,                   // знаменатель исходной дроби
    gcm,                   // Наибольший общий делитель
    numNew,    // Числитель и
    denNew;    // знаменатель сокращенной дроби
    do       {
    Console.Clear();
    do Console.Write("Введите отличный от нуля числитель: ");
    while (!uint.TryParse(Console.ReadLine(), out num||num==0));
    do Console.Write("Введите знаменатель: ");
    while (!uint.TryParse(Console.ReadLine(), out den));
    if (den == 0)  {
    Console.WriteLine("Нулевой знаменатель!");
    Console.WriteLine("Начинайте с начала!");
    Console.WriteLine("Для выхода нажмите клавишу ESC");
    continue;
    }
    gcm = GCM(num, den);
    numNew = num / gcm;
    denNew = den / gcm;
    Console.WriteLine("{0}/{1}  ===>  {2}/{3}",
                      num, den, numNew, denNew);
    Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Введите числитель: 54<ENTER>
Введите знаменатель: 0<ENTER>
Нулевой знаменатель!
Начинайте сначала!
Для выхода нажмите клавишу ESC
<ENTER>
Введите   числитель: 34<ENTER>
Введите знаменатель: 12<ENTER>
34/12  ===>  17/6
Для выхода нажмите клавишу ESC
<ESC>
```

Для решения задачи в программе использован классический вариант алгоритма Евклида. Его реализует статический метод –

функция с заголовком **static uint** GCM(**uint** a, **uint** b). Метод имеет параметры с типами значений (**uint**) и возвращает значение (делитель) типа **uint**. Обращение к методу используется в правой части оператора присваивания gcm = GCM(num, den);. С помощью аргументов в метод передаются значения числителя и знаменателя введенной пользователем дроби. Переменной gcm присваивается вычисленное значение наибольшего общего делителя. Остальное должно быть очевидно из текста программы, результатов и комментариев. Отметим только, что метод GCM() правильно работает только при натуральных (ненулевых положительных) значениях аргументов.

Задача 07-02. Ввести три целых числа, вывести их в порядке возрастания значений.

Задачу можно решить (как в задаче 05-03), не применяя вспомогательных методов. Однако изящным будет решение с применением двух методов-процедур. Первый метод упорядочивает значения двух параметров, а второй использует первый метод для упорядочивания трех параметров. Заметим, что упорядочиваются не параметры, а значения адресуемые аргументами. Для этого в предлагаемых методах параметры с типом значений передаются по ссылкам.

Программа, решающая задачу:

```
// 07_02.cs. – сортировка значений скалярных переменных
Using System;
classProgram {
// Вспомогательный метод (обратите внимание на перегрузку!):
static void order(ref int x, ref int y){
      if(x>y) {int temp=x; x=y; y=temp;}
}
// метод для упорядочения значений трех целых аргументов:
static void order(ref int a, ref int b, ref int c)  {
      order(ref a, ref b);
      order(ref b, ref c);
      order(ref a, ref b);
}
static void Main()  {
      int x, y, z;        // Вводимые числа
      do                  // цикл для повторения решений задачи
      {                   // Конструкция для ввода значения:
      do Console.Write("Введите x: ");
```

```
        while (!int.TryParse(Console.ReadLine(), out x));
        do Console.Write("Введите y: ");
        while (!int.TryParse(Console.ReadLine(), out y));
        do Console.Write("Введите z: ");
        while (!int.TryParse(Console.ReadLine(), out z));
        order(ref x, ref y, ref z);
        Console.WriteLine("Результат: x={0}, y={1}, z={2},",x,y,z);
        Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} // Main()
} // class Program
```

Результаты выполнения программы:

```
Введите x: 7<ENTER>
Введите y: 4<ENTER>
Введите z: 5<ENTER>
Результат: x=4, y=5, z=7,
Для выхода из программы нажмите ESC.
<ESC>
```

В программе два одноименных метода-процедуры с заголовками:

```
static void order(ref int x,ref int y)
static void order(ref int a,ref int b,ref int c)
```

В каждом из методов параметры специфицированы модификатором **ref**, то есть передаются по ссылкам. В этом случае операторы тела метода имеют возможность изменять значения аргументов. Первый из методов предназначен для упорядочения значений двух аргументов, второй, трижды обращаясь к первому методу, сортирует значения трех аргументов.

В основной программе (в методе Main()) вводятся значения трех переменных и выполняется обращение к методу сортировки:

```
order(ref x, ref y, ref z);
```

Обратите внимание, что модификатор **ref** обязательно используется как при определении метода в спецификации параметра, так и в обращении к методу перед именем аргумента.

Результаты выполнения программы и комментарии в ее тексте дополняют приведенные объяснения.

Задача 07-03. Вычислить sin x, используя приближенную формулу:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} ... (-1)^m \frac{x^{2m+1}}{(2m+1)!} ...,$$

где m – номер очередного члена, начиная с m = 0. Вычисление суммы ряда выполнить с использованием 10-ти членов ряда. В основной программе вводить значения x и выводить соответствующие значения синуса.

Собственно алгоритм суммирования этого ряда подробно рассмотрен в задаче 05-06. Поэтому сейчас основное внимание уделим методам. Для решения задачи в программе определим два метода. Первый метод-функция с именем rowSin должен формировать массив из N членов разложения в ряд функции sin(1). Второй метод mySin будет вычислять значение sin(x), принимая в качестве параметров значение x и массив, полученный с помощью первого метода. Здесь же будет учтена периодичность функции sin x, для чего перед вычислением синуса значение аргумента x будет приводиться к диапазону [0; 2π).

Программа, решающая задачу:

```
// 07_03.cs. – методы для вычисления sin x
using System;
class Program   {
static double[] rowSin(int n) {
    double[] tempAr = new double[n];
    double memb = 1; // Очередной член ряда
    for (int i = 0; i < tempAr.Length; i++)   {
        tempAr[i] = memb;
        memb /= -((i + 1) * 2) * ((i + 1) * 2 + 1);
        }
    return tempAr;
}
static double mySin(double x, double[ ] row) {
    double temp = x % (2 * Math.PI), x2 = temp*temp,
        sinX = 0;
    foreach (double elem in row)    {
        sinX += elem * temp;
        temp *= x2;
        }
    return sinX;
}
static void Main()      {
    int n = 10;                 // количество членов ряда
```

```
    double [] members;      // члены ряда
    double x,               // аргумент
    sinX;                   // значение синуса
    members = rowSin(n); // Вычисление членов ряда
    while (true)  { // цикл для ввода разных значений аргумента
      do Console.Write("Введите аргумент x:  ");
      while(!double.TryParse(Console.ReadLine(), out x));
      sinX = mySin(x, members);
      Console.WriteLine("sin({0:g3}) = {1:g4}",x,sinX);
      Console.WriteLine("Для выхода из программы нажмите ESC.");
      if (Console.ReadKey(true).Key == ConsoleKey.Escape) break;
    }  // while (true)
  }
}
```

Результаты выполнения программы:

```
Введите аргумент x:  44<ENTER>
sin(44) = 0,0177
Для выхода нажмите клавишу ESC
<ENTER>
Введите аргумент x:  3,14<ENTER>
sin(3,14) = 0,001593
Для выхода из программы нажмите ESC.
<ESC>
```

Как запланировано, в коде программы в классе Program, кроме метода Main() два взаимодействующих статических метода. Первый с заголовком

static double[ ] rowSin(intn)

формирует массив с n элементами типа **double**, присваивает элементам значения n членов ряда разложения sin(1) и возвращает значение ссылки на этот массив.

Второй метод с заголовком

static double mySin(double x, double[ ] row)

умножает члены ряда, представленные элементами массива-параметра, на величину $x^{(2m+1)}$, где m – индекс очередного элемента массива. Массив передается в метод по значению параметра-ссылки **double**[] row. Еще раз обратим внимание, что параметр передается по значению, а тип параметра – ссылка на массив с элементами типа **double**. Метод возвращает вычисленное значение sin(x) как величину типа **double**.

Особенность основной программы состоит в том, что метод rowSin(), формирующий массив членов ряда, вызывается только один раз. Результат его выполнения присваивается ссылке **double** [] members. Затем в цикле пользователь может вводить разные значения переменной x, и для каждого значения выполняется оператор ***sinX = mySin(x, members);***

Таким образом, программа «экономит» количество операций, необходимых для многократных вычислений синуса, но «платит» за эту экономию необходимостью хранить заранее подсчитанные значения членов ряда.

Задача 07-04. Треугольник Паскаля образован биномиальными коэффициентами:
0-й уровень – строка из одного элемента со значением C(0,0)=1,
1-й уровень – строка из 2-х элементов C(1,0)=C(1,1)=1.
2-й уровень – строка из 3-х элементов:
C(2,0)=C(2,2)=1, C(2,1)=2.
.....
n-й уровень – строка из n+1 элементов:
C(n,0)=C(n,n)=1,.....C(n,k)=C(n-1,k-1)+C(n-1,k).
Вводя неотрицательные значение n (не большее 12) построить массив массивов со значениями биномиальных коэффициентов и вывести его на экран, размещая значения элементов каждого массива нижнего уровня по строкам и соблюдая традиционный формат представления треугольника Паскаля.

Выделим из задачи две части. Часть первая - построение массива ссылок на массивы с элементами, равными значениям биномиальных коэффициентов. Часть вторая – вывод треугольника Паскаля, представленного массивом массивов. Каждую часть оформим в виде отдельного метода. С учетом сказанного, программа может быть такой:

```
// 07_04.cs – методы для построения треугольника Паскаля
using System;
class Program  {
// метод построения массива биномиальных коэффициентов:
static int[ ][ ] triPaskal(int n)    {
      int[][] tabl = new int[n][ ];          // Массив ссылок на массивы
     for (int i = 0; i < tabl.Length; i++)     {
       tabl[i] = new int[i + 1];             // массив элементов типа int
       tabl[i][0] = tabl[i][i] = 1;
```

```
        for (int j = 1; j < i; j++)
        tabl[i][j] = tabl[i - 1][j - 1] + tabl[i - 1][j];
        }
        return tabl;
} // triPaskal()
// Метод вывода треугольника Паскаля:
static void printPaskal(int[][] ar)    {
     string gap = "  ", // вспомогательный "заполнитель"
          line;              // строка треугольника Паскаля
     int h = ar.Length;  // высота треугольника Паскаля
// цикл по строкам треугольника:
     for (int k = 0; k < h; k++)  {
       line = "";
       for (int j = 0; j < h - k; j++) // «пустое» начало строки
             line += gap;
       foreach (int cnk in ar[k])  { // перебор  элементов
                line += string.Format("{0,3:d}", cnk);
                line += gap;   // пробелы  между  коэффициентами
                }
       Console.WriteLine(line);
  }  // for(int k ...
}  // printPaskal()
static void Main()   {
    int[ ][ ] paskal; // ссылка на массив ссылок на массивы
    int n;
    do  { // цикл для повторения решения
    do Console.Write("Введите n (неболее 12): ");
    while (!int.TryParse(Console.ReadLine(), out n)
                    || n < 0 || n > 12);
     paskal = triPaskal(n);          // создать  массив
     printPaskal(paskal);            // вывести  массив
     Console.WriteLine("Для выхода из программы нажмите ESC.");
  } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
} // Main()
}
```

Результаты выполнения программы:

```
Введите n (не более 12): 7
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
Для выхода из программы нажмите ESC.
<ESC>
```

Задача 07-05. В задаче 02-03 показано, как по-разному должна быть выполнена при вводе запись вещественного числа в зависимости от «настройки» операционной системы. При вводе числовой информации с клавиатуры пользователь зачастую делает ошибку, отделяя целую часть числа от дробной точкой (или запятой, если консоль «настроена» на американский формат записи вещественных чисел). Для устранения этого неудобства разработать метод, позволяющий в изображении числа равноправно вводить либо точку, либо запятую.

Ограничим задачу созданием метода для «надежного» ввода вещественных чисел только в форме с фиксированной точкой (запятой). Метод не будет пригоден для ввода числа с указание мантиссы и порядка. Допустимо будет вводить знак числа, последовательность десятичных цифр и единственный знак «точка» или «запятая», отделяющий целую часть числа от дробной. При вводе метод будет «игнорировать» нечисловые символы, а также неверно (не на месте) использованные символы «знак числа», «точка», «запятая». Метод будет правильно работать при настройке среды исполнения на европейский формат («ru-Ru») представления вещественных чисел. См. задачу 02-03.

В основной программе вводить вещественные числа, специально допуская ошибки при вводе, и выводить «воспринятые» числа.

Следующая программа решает поставленную задачу:

```
// 07_05.cs – Ввод вещественного числа (точкой либо запятой)
using System;
class Support  {
public static double readD()  {
   double res = 0;
   string numb = “”;       // Строка для результата ввода
   do  {// цикл для повторения ввода при ошибках
      Console.Write(“Введите вещественное число: “);
      ConsoleKeyInfo key;      // Нажатая клавиша
      numb = “”;                  // Строка для результата ввода
      do  {   // цикл для формирования правильного числа
         key = Console.ReadKey(true);     // Читать без отображения
         char ch = key.KeyChar;              // Символ от клавиши
          if (((ch == ‘-’ | ch == ‘+’) & numb == “”) |
                (ch >= ‘0’ & ch <= ‘9’))
             {
```

```
            Console.Write(ch);
            numb += ch;
            continue;
            }
          if (ch == '.') ch = ',';
          if (ch == ',' & numb.IndexOf(',') < 0)
             {
             Console.Write(ch);
             numb += ch;
             }
      } while (key.Key != ConsoleKey.Enter);
      Console.WriteLine();
       // преобразовать строку в число:
      } while (!double.TryParse(numb, out res));
      return res;
}  // readD
}  // class Support

class Program  {
static void Main()  {
   do  {  // цикл повторения решений
       double x;
       x = Support.readD();   // Прочитать числовое значение x
       Console.WriteLine("Значение: {0:g6}", x);
       Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}  // Main
}  // Program
```

Результаты выполнения программы:

```
Введите вещественное число: 34,986<ENTER>
Значение: 34,986
Для выхода нажмите клавишу ESC
<ENTER>
Введите вещественное число: 769769697665<ENTER>
Значение: 7,6977e+11
Для выхода из программы нажмите ESC.
<ESC>
```

В приведённых результатах выполнения программы нет никаких признаков того, что при вводе данных допускались ошибки. Объясняется это тем, что чтение набираемых на клавиатуре символов выполняется методом Console.ReadKey(**true**). без их отображения на экране. Прочитанный символ анализируется и выводится на экран только в том случае, если он допустим в изображении вещественного числа.

В отличие от предыдущих программ, в данной программе два класса: **class** Support и **class** Program. Основной метод с именем Main находится в классе Program. Метод чтения данных от клавиатуры, выполняющий всю полезную работу, размещен в классе Support. Его заголовок:

public static double readD()

Размещение методов по разным классам типично для реальных программ на языке C#. Для того, чтобы метод одного класса был доступен из методов другого класса, в его объявлении используется модификатор доступа **public**. Как и все методы этой темы, этот метод принадлежит своему классу (метод статический), что определено модификатором **static**. Метод возвращает в точку вызова значение типа **double**. Имя метода *readD*, и у метода нет параметров. Особенность метода в том, что он получает данные только из консольного потока.

В теле метода *readD()* два цикла. Внешний цикл предусматривает повторение ввода при полностью неверной набранной последовательности символов. (Когда среди введенных пользователем символов нет ни одной цифры.) Внутренний цикл продолжается до появления во входной строке кода нажатия клавиши ENTER. В этом внутреннем цикле оператор *key = Console.ReadKey(true)*; присваивает переменной *key* код очередной нажатой пользователем клавиши. Далее анализируется соответствующий символ (переменная ch) и, если он допустим в записи числа – его добавляют в конец строки представленной ссылкой (переменной) *numb*. При появлении кода клавиши ENTER цикл завершается и делается попытка преобразовать строку в число – в значение переменной *res*. При удаче завершается внешний цикл, и выполняется оператор **return** *res*;. В точку вызова будет передано значение вещественного числа. При неудачном преобразовании – повторяется приглашение пользователю.

В основной программе (в методе Main() из класса Program) цикл для возможного повторения решений и новое для нас – обращение к методу другого класса в операторе присваивания x = Support.readD();. Результат как значение переменной x выводится с использованием форматирования (спецификация g6).

Задача 07-06. Присваивая последовательным элементам массива случайные значения от 1 до 9, создать массив с минимальным количеством элементов, сумма которых не превышает заданного пользователем числа.

Решая эту задачу, создадим рекурсивный метод, параметр которого – заданное значение суммы, возвращаемое значение – ссылка на массив, удовлетворяющий условию задачи. Особенность и сложность построения такого метода не только в его рекурсивности, но и в невозможности на языке C# непосредственно создавать растущие массивы. В то же время до начала решения задачи точно определить размер массива в общем случае нельзя. Поэтому рост массива будем моделировать, переписывая на каждом уровне рекурсии полученный массив в новый массив, число элементов которого увеличено на 1.

Полный текст программы:

```
// 07_06.cs – рекурсивное формирование массива
using System;
class Program  {
static void Main()  {
    int sum;
    int[] ar;
    do {  // Цикл повторения решений
      do     Console.Write("Введите пороговое значение суммы: ");
      while (!int.TryParse(Console.ReadLine(), out sum) | sum <= 0);
      ar = newArray(sum);
      foreach (int e in ar)
        Console.Write(e + " ");
      Console.WriteLine("\nДля выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}  // Main()
static Random gen = new Random( );
static int[] newArray(int numb)  {
    int next = gen.Next(1, 10);  // очередной элемент
    if (next >= numb) return new int[] { next };
    int[] temp = newArray(numb - next);
    int len = temp.Length;  // длина уже созданного массива
    int[] res = new int[len + 1];
    temp.CopyTo(res, 0);
    res[len] = next;
    Array.Reverse(res); // в порядке создания элементов
    return res;
}  // newArray()
}
```

Результаты выполнения программы:

```
Введите пороговое значение суммы: 17<ENTER>
9 8 8
Для выхода из программы нажмите ESC.
<ENTER>
Введите пороговое значение суммы: 17<ENTER>
7 8 1 5
Для выхода из программы нажмите ESC.
<ESC>
```

В основной программе (в методе Main) определена переменная **int** sum для представления требуемой суммы и ссылка **int**[] ar на формируемый массив. Эта ссылка получает значение после выполнения оператора ar = newArray(sum);. Первый оператор тела метода newArray( ) – вычисляет очередной элемент массива и присваивает его вспомогательной переменной:

```
int next = gen.Next(1, 10);
```

Для получения случайного числа здесь используется объект-генератор и его метод Next(). Ссылка gen на этот объект класса Random определена как статический член класса Program. Это не случайно. Синтаксически не было бы ошибкой определить объект-генератор в теле метода newArray(). Однако, при многократных рекурсивных входах в метод newArray() значения, формируемые таким генератором, могли оказаться одинаковыми.

Следующий оператор сравнивает значение переменной next с требуемым значением суммы numb:

```
if (next >= numb) return new int[ ] { next };
```

Если условие выполнено, то создается массив из одного элемента со значением next и ссылка на него возвращается в точку вызова метода.

В противном случае выполняется оператор

```
int[ ] temp = newArray(numb - next);
```

В этом операторе определена временная ссылка на массив, и ей присваивается результат рекурсивного обращения к методу newArray(). Аргумент метода – уменьшенное на величину очередного члена значение требуемой суммы. Такие рекурсивные обращения продолжаются до тех пор, пока сумма элементов последовательности меньше требуемой суммы. В этом случае в операторе **int**[]

res = **new int**[len + 1]; создается массив с нужным количеством элементов и в следующих операторах элементам этого массива присваиваются значения всех сформированных членов последовательности. Рекурсивность описанной процедуры приводит к тому, что массив заполнен в обратном порядке по сравнению с порядком вычисления элементов последовательности. Для получения «правильного» порядка значений элементов массива выполняется оператор Array.Reverse(res);.

Задача 07-07. Сформировав квадратную целочисленную матрицу со случайными значениями элементов, упорядочить ее строки по возрастанию сумм их элементов. Пусть элементы принимают значения в диапазоне от –50 до +50.

Программа, решающая сформулированную задачу, может быть построена по-разному. Первый вопрос о данных в программе – как представить матрицу? Это может быть двумерный массив, либо массив ссылок на одномерные массивы – строки матрицы. В зависимости от выбора ответа на первый вопрос могут быть применены разные средства для сортировки строк матрицы. Если матрица представлена двумерным массивом, то нужно написать отдельный метод для упорядочивания строк матрицы. Если для представления матрицы используется массив массивов, то можно применить статический метод Sort() библиотечного класса Array. (Применить этот метод для сортировки двумерного массива нельзя – метод Sort() «работает» только с одномерными массивами.)

Чтобы продемонстрировать особенности применения метода Array.Sort(), будем представлять матрицу массивом массивов. В программе определим три метода: метод getMatrix() для формирования квадратной матрицы заданных размеров; метод matrPrint( ) для вывода на экран матрицы в виде таблицы; метод compareRow( ), задающий правила сравнения строк матрицы. Имя последнего метода будет использовано в качестве второго аргумента в обращении к методу сортировки Array.Sort().

Программа, решающая задачу:

```
// 07_07.cs – сортировка строк матрицы в виде массива массивов
using System;
class Program  {
```

```
// Напечатать матрицу (массив массивов):
static void matrPrint(int[ ][ ] ar)   {
// Число  строк  матрицы:
   int nn = ar.GetUpperBound(0);
   for (int i = 0; i <= nn; i++, Console.WriteLine())
      for (int j = 0; j <= ar[i].GetUpperBound(0); j++)
         Console.Write("{0,4:d}", ar[i][j]);
}
// Сформировать матрицу со случайными элементами:
static int[][] getMatrix(int n) {
   Random generator = new Random();
   int[ ][ ] matr = new int[n][];   // Массив  ссылок  на  массивы
   for (int i = 0; i < n; i++)   {
   matr[i] = new int[n]; // объект-массив элементов типа int
   for (int j = 0; j < n; j++)
      matr[i][j] = generator.Next(-50, 51);
   }
   return matr;
}
// Метод сравнения сумм элементов двух массивов:
static int compareRow(int[] row1, int[] row2) {
   int sum1 = 0, sum2 = 0;
   for (int i = 0; i < row1.Length; i++) {
      sum1 += row1[i];
      sum2 += row2[i];
      }
   if (sum1 > sum2) return +1;
   if (sum1 < sum2) return -1;
   return 0;
}
static void Main()   {
   int[ ][ ] matrix;     // ссылка на массив массивов
   int size;             // размер квадратной матрицы
   do {                  // Цикл повторения решений
    do                   // Цикл контроля ввода:
     Console.Write("Введите размер матрицы: ");
     while (!int.TryParse(Console.ReadLine(), out size) | size <= 0);
     matrix = getMatrix(size);
     Console.WriteLine("Исходная матрица:");
     matrPrint(matrix);
     Array.Sort(matrix,compareRow);
     Console.WriteLine("Упорядоченная матрица:");
     matrPrint(matrix);
     Console.WriteLine("Для выхода из программы нажмите ESC.");
} while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}  // Main()
}
```

Результаты выполнения программы:

```
Введите размер матрицы: 5<ENTER>
Исходная матрица:
 14 -42  34   3  43
-39 -43 -45  14   1
 -7 -15 -31  21  40
 31   4  11  17  42
-10  15 -27 -49   3
Упорядоченная матрица:
-39 -43 -45  14   1
-10  15 -27 -49   3
 -7 -15 -31  21  40
 14 -42  34   3  43
 31   4  11  17  42
Для выхода из программы нажмите ESC.
<ESC>
```

В программе первым определен метод, позволяющий выводить на консольный экран целочисленные значения элементов массива массивов. Этот массив массивов может представлять матрицу не квадратную и даже не прямоугольную. Рассмотрим отдельно код этого метода:

```
static void matrPrint(int[][] ar)   {
// Число строк матрицы:
    int nn = ar.GetUpperBound(0);
    for (int i = 0; i <= nn; i++, Console.WriteLine())
        for (int j = 0; j <= ar[i].GetUpperBound(0); j++)
            Console.Write("{0,4:d}", ar[i][j]);
}
```

Параметр **int**[][] ar – это ссылка на одномерный массив ссылок на массивы с целочисленными элементами. Локальной вспомогательной переменной nn присваивается максимальное значение индекса массива ссылок. В цикле с параметром i перебираются элементы массива ссылок, то есть строки матрицы. Параметр j вложенного цикла изменяется от 0 до предельного значения индекса соответствующей строки. Строки в общем случае могут иметь разные количества членов (разные длины).

Метод с заголовком **static int**[][] getMatrix(**int** n), формирует квадратную матрицу в виде массива массивов и присваивает ее элементам случайные значения.

Внимания заслуживает метод с заголовком

```
static int compareRow(int[ ] row1, int[ ] row2)
```

Этот метод в программе явно как-будто не вызывается. Его имя используется в качестве второго аргумента в обращении

```
Array.Sort(matrix,compareRow);
```

Назначение метода compareRow( ) – сравнивать суммы элементов двух массивов, предаваемых в метод с помощью параметров. Если сумма элементов первого массива больше суммы второго, то метод возвращает +1, и это считается нарушением требуемого порядка размещения строк матрицы. В этом случае метод *Array.Sort( )* меняет местами соответствующие строки матрицы. (Следует отметить, что меняются местами не строки матрицы, а только соответствующие значения ссылок.)

Результаты выполнения программы и комментарии в ее тексте дополняют приведенные объяснения.

Задача 07-08. В одномерном целочисленном массиве со случайными значениями элементов определите все локальные минимумы. То есть выведите индексы всех элементов, значения которых меньше значений соседних элементов. При оценке минимальности первого и последнего элементов каждый из них будем сравнивать только с одним соседом.

В программе, решающей задачу, кроме основного метода Main() определим методы: minimum( ) – метод, возвращающий массив индексов локальных минимумов; array( ) – метод, формирующий одномерный массив со случайными значениями элементов.

На примере этой программы рассмотрим вопрос о надежности методов с точки зрения их устойчивости по отношению к исходным данным.

Программа, решающая задачу:

```
// 07_08.cs - массив индексов локальных минимумов
using System;
class Program  {
// Возвращает массив индексов локальных минимумов:
static int[] minimum(int[] vect)  {
   int d = vect.Length;
   if (d < 2)  //****
      { Console.Write("Нет решения!"); return null; }
   int x = vect[0];
   bool[] mark = new bool[d];       // Флаги  минимумов
```

```
    int ind = 0;                          // счетчик  минимумов
    if (vect[0] < vect[1])                // первый  элемент
       { mark[0] = true;  ind++; }
    for (int i = 1; i < d - 1; i++)
       if (vect[i - 1] > vect[i] & vect[i + 1] > vect[i])
             { mark[i] = true;  ind++; }
    if (vect[d - 2] > vect[d - 1])        // последний  элемент
       { mark[d - 1] = true;  ind++; }
    int[] result = new int[ind];          // массив  результатов
    int k = 0;
    for (int i = 0; i < d; i++)
       if (mark[i]) result[k++] = i;
    return result;
}  // minimum( )

// Создание массива со случайными значениями элементов:
static int[] array(int n, int mi, int ma)   {
    if (n < 1)  //****
    { Console.Write("Нельзя создать массив!"); return null; }
    // xmin, xma – границы интервала:
    int xmi = (int)Math.Min(mi, ma);
    int xma = (int)Math.Max(mi, ma);
    Random gen = new Random( );  // датчик  случайных  чисел
    int[] A = new int[n];
    for (int i = 0; i < n; i++)
          A[i] = gen.Next(xmi, xma);
    return A;
}  // array()

static void Main()   {
    int n;  //размер массива
    int xMin = -10, xMax = 10;        // Границы диапазона значений
    do    {                           // Для повторения решений…
       Console.Clear();
       do Console.Write("Введите размер массива: ");
       while (!int.TryParse(Console.ReadLine(), out n)
                  | n < 2);   //****
       int[] row = array(n, xMin, xMax);
       foreach (double v in row)
             Console.Write("{0,5} ", v);
       Console.WriteLine();
       int[] localMin = minimum(row);
       Console.WriteLine("Индексы локальных минимумов: ");
       foreach (int v in localMin)
             Console.Write("{0,5} ", v);
       Console.WriteLine();
       Console.WriteLine("Для выхода из программы нажмите ESC.");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}  //  Main()
}
```

Результаты выполнения программы:

```
Введите размер массива: -3<ENTER>
Введите размер массива: 0<ENTER>
Введите размер массива: 1<ENTER>
Введите размер массива: 7<ENTER>
  -6   -8   -1    5    3   -2   -3
Индексы локальных минимумов:
   1    6
Для выхода из программы нажмите ESC.
<ESC>
```

В коде программы комментариями вида // **** отмечены операторы, которые не нужны в том случае, если и программист и пользователь никогда не ошибаются. Но пользователь может ошибиться при вводе исходных данных, а программист – в выборе значений аргументов при обращении к методам.

Фрагмент кода

```
if (d< 2)    // ****
 { Console.Write("Нет решения!"); return null; }
```

защищает метод minimum() от обработки слишком маленького массива, в котором не может быть минимального элемента.

В методе array() три параметра, и можно сделать несколько ошибок при обращении к нему. Проверка **if** (n < 1) вводит защиту от неверного выбора размера массива. Использование вспомогательных переменных xmi и xma защищает от потенциальной ошибки при обращении к методу Next(). Значение первого аргумента этого метода не может превышать значения второго аргумента.

В данной конкретной уже составленной программе отмеченные операторы «защиты» методов array() и minimum() излишни. В методе Main() предусмотрена проверка вводимого значения размера массива (n < 2), а предельные значения элементов массива заданы явно и правильно (xMin = -10, xMax = 10). Однако, проектируя методы, целесообразно вводить в их код операторы проверки тех исходных данных, которые будут им переданы при обращениях. В данной программе предусмотрен вывод в консольное окно предупреждающих сообщений. Но в методах, предназначенных для повторного использования сторонними программистами, следует использовать механизм исключений.

ТЕМА 08

МЕТОДЫ В БИБЛИОТЕКЕ КЛАССОВ

Даже на начальном этапе изучения программирования полезно накапливать свои результаты и сохранять коды для последующего применения в новых программах. Делать это можно с помощью разных подходов. Сейчас нам доступен один из них – создание набора методов, которые по каким-то показателям представляются нам достаточно универсальными. Такой набор обычно функционально близких методов язык C# позволяет оформлять в библиотеке классов. Методы в этом случае часто создаются как статические, и обращаться к ним нужно с помощью квалифицированных имен, включающих имя соответствующего класса, которому они принадлежат.

Задача 08-01. Создать библиотеку классов и разместить в ней класс с именем Methods со статическими методами для работы с массивами. В качестве первого шага «заполнения» библиотеки определить в классе Methods один метод для вывода на консоль значений элементов одномерного целочисленного массива.

Библиотеку классов создадим как отдельный проект Library нового решения с тем же именем Library в каталоге нашего практикума «c:\C#_Практикум». Для этого в Visual Studio и открываем окно «New Project» (рис. 8.1). В левой панели выбираем Visual C#, на центральной панели активируем пункт ClassLibrary. Вводим: имя проекта Name = Library; указываем размещение каталога с проектом: Location = c:\C#_Практикум; выбираем имя решения: Solution name = Library.

Нажав клавишу OK, запускаем процесс создания проекта Library и решения Library. Результат – шаблон кода в текстовом редакторе и каталог файлов решения (см. рис. 8.2).

Чтобы в библиотеке класс имел выбранное нами имя Methods, в окне решения правой кнопкой активируем имя Class1.cs и в выпавшем меню выбираем пункт Rename. Вводим

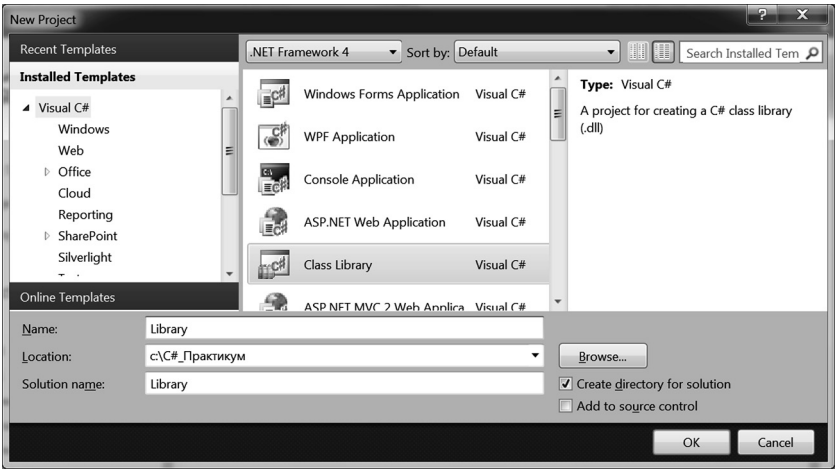


Рис. 8.1. Окно выбора проекта для библиотеки классов

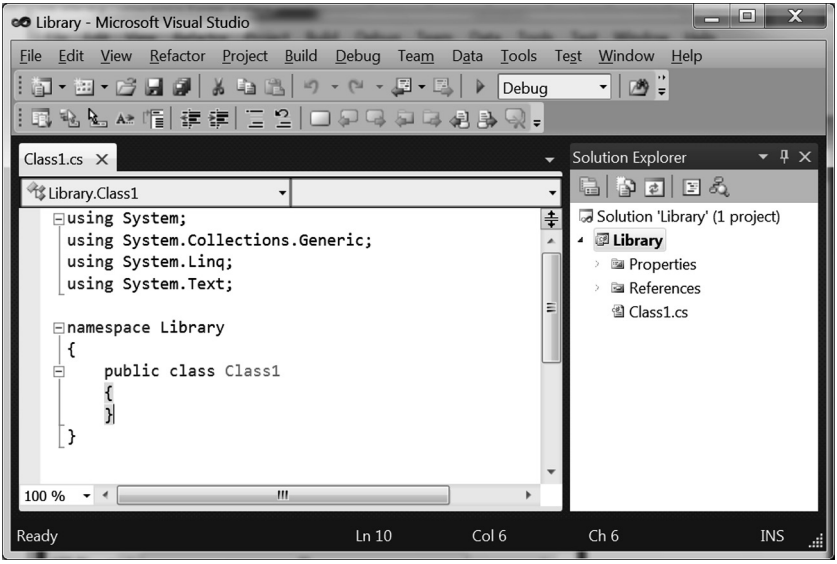


Рис. 8.2. Окна текстового редактора и Solution Explorer

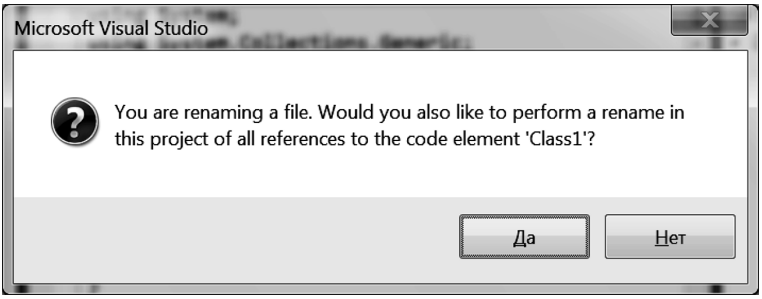


Рис. 8.3. Окно с предложением внести изменение имени в ссылки

новое имя класса «Methods.cs» и соглашаемся внести это имя во все ссылки(см. рис. 8.3).

Изменив указанным способом имя класса в проекте, введём в предлагаемый шаблон кода объявление метода для печати одномерного массива. Удалив лишние операторы **using**, получим в окне текстового редактора следующий код:

```
// Methods - класс методов в библиотеке Library
using System;
namespace Library  {
public class Methods  {
// Метод для вывода элементов целочисленного массива:
public static void printArray(int[] a, string format)  {
   for (int i = 0; i < a.Length; i++)
      Console.Write(format, a[i]);
   Console.WriteLine();
}  // printArray()
}  // Methods
}  // Library
```

Метод printArray( ) объявлен в открытом классе Methods как открытый (**public**) статический (**static**). Его первый параметр – ссылка на целочисленный одномерный массив. Второй параметр – строка, применяемая в теле метода для форматирования значения одного элемента массива при его выводе. В цикле с параметром **int** i выполняется перебор всех элементов массива. Обращение в конце тела метода (вне цикла) к методу Console. WriteLine() обеспечивает перевод строки при окончании вывода всех элементов массива.

Для дальнейшего применения метода printArray( ) в других проектах важно, что метод открытый, что он статический, что

он принадлежит открытому классу Methods, и что этот класс определен в пространстве имен Library.

Для того чтобы получить исполнимый код библиотеки классов, то есть в нашем примере файл «Library.dll», запустите компиляцию библиотеки, используя главное меню:

Build -> Build Library или сочетание клавиш ***Shift+F6***

Путь к созданному исполнимому коду библиотеки:

C:\C#_Практикум → Library → Library → bin → Debug → Library.dll

Библиотека создана, в ней пока только один класс, в этом классе только один метод. Для тестирования метода нужен исполнимый модуль (exe-сборка), в котором присутствует статический метод Main() (точка входа в программу) и есть обращение к библиотечному методу printArray( ). Создать исполнимый модуль можно разными путями. Во-первых, можно в том же решении Library, где определена библиотека классов, определить новый проект, подключить к нему библиотеку Library.dll и в коде проекта вызывать метод printArray( ). Но полезнее показать как применять метод библиотеки не только из другого проекта, но и из другого решения. Поэтому, как обычно, для новой темы создаем решение «Тема_08» и в нем проект консольного приложения Program_1 со следующим кодом:

```
// 08_01.cs – иллюстративная программа для работы с библиотекой
using System;
using Library; // Пространство имен библиотеки
class Program   {
static void Main( )   {
   int[ ] testArray = new int []
      {0,10,20,30,40,50,60,70,80,90};
   Methods.printArray (testArray, "{0} ");
   Console.WriteLine("Для выхода из программы нажмите
ENTER.");
Console.ReadLine();
}
}
```

В коде программы объявлена ссылка testArray на массив, инициализированный списком целых чисел. Этот массив нужно вывести в консольное окно с помощью библиотечного метода printArray(). Для связи с пространством имен библиотеки в код добавлен оператор **using** Library;. Но этого для обраще-

ния из проекта к методу библиотеки Library недостаточно. Необходимо в число ссылок проекта Program_1 добавить ссылку на Library.dll. Для этого в окне решения Тема_08 активируем правой кнопкой имя проекта Program_1. Выбираем пункт меню **Add Reference**. В меню окна выбираем "закладку" **Browse** и «добираемся» (спускаясь по дереву из каталога C#_Практикум) до Library.dll. В результате в число ссылок проекта будет включена ссылка на библиотеку Library, и её можно увидеть в окне Solution Explorer (см. рис. 8.4).

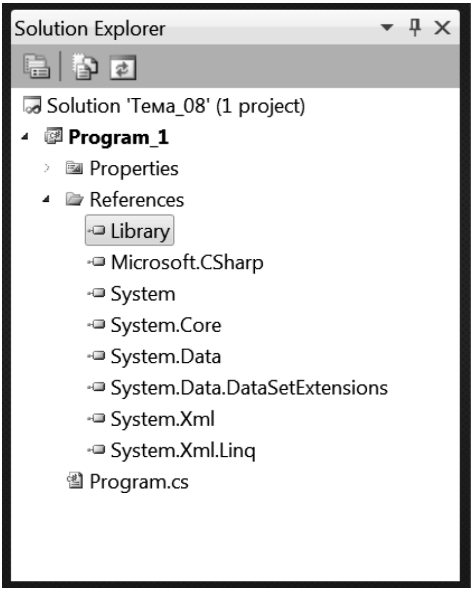


Рис. 8.4. Ссылки проекта Program_1

Только теперь можно без ошибок компилировать проект и получить результаты.

Результаты выполнения программы:

```
0 10 20 30 40 50 60 70 80 90
Для выхода из программы нажмите ENTER.
<ENTER>
```

Задача 08-02. Разработать библиотечные методы для вывода в консольное окно (для «печати»): 1) одномерного масси-

ва с вещественными значениями элементов; 2) матрицы с целочисленными значениями элементов; 3) матрицы с вещественными значениями элементов.

Для решения задачи используем возможность перегрузки методов. Следующий метод будет отличаться от метода printArray( ) из предыдущей задачи только типом первого параметра:

```
// Метод для вывода элементов вещественного массива:
public static void printArray(double[] a, string format)  {
   for (int i = 0; i < a.Length; i++)
      Console.Write(format, a[i]);
      Console.WriteLine();
      }  // printArray()
```

Поместив код статического метода в библиотеку (в класс Methods библиотеки Library), выполним ее компиляцию, и получим обновленный вариант модуля Library.dll. Теперь можно проверить работоспособность уже двух библиотечных методов. Для этого в уже существующем решении «Тема_08» создаем новый проект консольного приложения Program_2 со следующим кодом:

```
// 08_02.cs - программа с методами печати массивов
using System;
using Library;  // Пространство  имен  библиотеки
class Program   {
static void Main()  {
   int[ ] testArray = new int[ ]
                { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90 };
   Methods.printArray(testArray, "{0} ");
   double[ ] dArray = new double[] { 0, 1, 2, 3, 4 };
   Methods.printArray(dArray, "{0:f2}  ");
   Console.WriteLine("Для выхода из программы нажмите ENTER.");
   Console.ReadLine();
}
}
```

Как уже описано выше, подключаем к проекту Program_2 библиотеку Library.dll (добавляем ссылку на неё) и, оттранслировав приложение, получим следующие результаты выполнения программы:

```
0 10 20 30 40 50 60 70 80 90
0,00  1,00  2,00  3,00  4,00
Для выхода из программы нажмите ENTER.
<ENTER>
```

Наиболее важное в программе – обращения к разным библиотечным методам с одинаковым именем Methods.printArray. В первом случае выполняется метод для печати целочисленного массива, во втором – метод для вывода значений элементов вещественного массива.

Для методов печати элементов матриц можно было бы использовать то же имя printArray, но для наглядности выберем для них имя printMatrix. Методы будут отличаться только типом первого параметра.

Код метода печати двумерного целочисленного массива:

```
// Печать целочисленной матрицы по строкам:
public static void printMatrix(int[,] matr, string format)  {
   for (int i = 0; i < matr.GetLength(0); i++, Console.WriteLine())
      for (int j = 0; j < matr.GetLength(1); j++)
         Console.Write(format, matr[i, j]);
}  // printMatrix
```

Код метода для двумерного вещественного массива:

```
// Печать вещественной матрицы по строкам:
public static void printMatrix(double[,] matr, string format)  {
  for (int i = 0; i < matr.GetLength(0); i++, Console.WriteLine())
     for (int j = 0; j < matr.GetLength(1); j++)
        Console.Write(format, matr[i, j]);
}  // printMatrix
```

В методах printMatrix() обратите внимание на заголовок внешнего цикла, точнее на его завершающее выражение i++, Console.WriteLine(). После окончания вложенного цикла, в котором перебираются элементы очередной i-й строки, значение параметра i увеличивается на 1, и обращение к методу Console. WriteLine() обеспечивает переход к новой строке на экране.

Поместив коды статических методов в библиотеку (в класс Methods библиотеки Library), выполним ее компиляцию, и получим обновленный вариант модуля Library.dll.

Так как иллюстрация возможностей методов печати матриц требует формирования матриц, то отложим проверку созданных библиотечных методов до решения следующей задачи.

Задача 08-03. Разработать библиотечные методы для создания одномерных и двумерных массивов со случайными значениями элементов.

Приведем вначале два одноименных метода для создания одномерных массивов. В каждом из них три параметра. Первый параметр определяет размер массива (количество элементов), два следующих задают интервал, из которого выбираются случайные значения элементов массива.

Код метода для создания одномерного массива с целочисленными случайными элементами:

```
// Создание массива со случайными значениями элементов
public static int[] randArray(int n, int mi, int ma)  {
// n - число элементов масива, min, ma - границы интервала.
   if (n < 1 || mi > ma) return null;
   Random gen = new Random(); // датчик случайных чисел
   int[] A = new int[n];
   for (int i = 0; i < n; i++)
       A[i] = gen.Next(mi,ma);
   return A;
}  // randArray
```

Этот метод randArray() создает и инициализирует одномерный массив из n элементов типа **int**. Значения элементов выбираются случайно из интервала [mi,ma). При обращениях к методу должно соблюдаться условие: mi<=ma. Кроме того, число элементов массива, определяемое значением первого параметра n, должно быть положительным. Метод возвращает ссылку на созданный массив. При ошибках в значениях аргументов массив не создается, и возвращается значение **null**. Параметры, определяющие интервал значений элементов, должны иметь тип **int**.

Код метода для создания одномерного массива со случайными вещественными элементами:

```
// Создание массива со случайными значениями элементов
public static double[ ] randArray(int n, double mi, double ma)  {
// n - число элементов масива, min, ma - границы интервала.
    if (n < 1 || mi > ma) return null;
    Random gen = new Random();          // датчик случайных чисел
   double[ ] A = new double[n];
   for (int i = 0; i < n; i++)
       A[i] = gen.NextDouble() * (ma - mi) + mi;
   return A;
}  // randArray
```

Метод randArray() создает и инициализирует массив из n элементов типа **double**. Значения элементов выбираются случайно из интервала [mi,ma). При обращениях к методу долж-

но соблюдаться условие: mi<=ma. Кроме того, число элементов массива, определяемое значением первого параметра n, должно быть положительным. Метод возвращает ссылку на созданный массив. При ошибках массив не создается, и возвращается значение **null**. Параметры, определяющие интервал значений элементов, должны иметь тип **double**.

Два одноименных метода randArray() различаются способом получения значений элементов. В целочисленном массиве используется метод gen.Next(mi,ma) объекта класса Random. Обязательное условие – аргументы, заменяющие параметры mi, ma, должны иметь тип **int**. Во втором методе создается массив с элементами типа **double**, и для вычисления их значений применяется выражение с обращением к методу gen.NextDouble(). Обязательное условие для вызова именно этого метода – хотя бы один из аргументов, заменяющих параметры mi, ma, должен иметь тип **double**.

Следующие два метода возвращают ссылки на двумерные массивы, представляющие прямоугольные матрицы. Построены они по той же схеме, что и методы для формирования одномерных массивов, за исключением естественных отличий, связанных с тем, что размерность массива равна двум и изменено имя методов.

```
// Создание матрицы с целочисленными случайными элементами:
public static int[,] randMatrix(int n, int m, int mi, int ma)  {
    // mi, ma - границы интервала случайных значений элементов.
    if (n < 1 || m < 1 || mi > ma) return null;
    Random gen = new Random();          // датчик случайных чисел
    int[,] matr = new int[n, m];
    for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        matr[i, j] = gen.Next(mi, ma);
    return matr;
}  // randMatrix

// Создание матрицы с вещественными случайными элементами:
public static double[,] randMatrix(int n, int m, double mi, double ma) {
// mi, ma - границы интервала случайных значений элементов.
    if (n < 1 || m < 1 || mi > ma) return null;
    Random gen = new Random();          // датчик случайных чисел
    double[,] matr = new double[n, m];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            matr[i, j] = gen.NextDouble() * (ma - mi) + mi;
    return matr;
}  // randMatrix
```

Поместив коды методов в библиотеку (в класс Methods библиотеки Library), выполним ее компиляцию, и получим обновленный вариант модуля Library.dll.

Теперь можно тестировать методы. Для этого создадим в существующем решении «Тема_08» новый проект консольного приложения Program_3 со следующим кодом:

```
// 08_03.cs - программа с библиотечными методами
using System;
using Library; // Пространство имен библиотеки
class Program  {
static void Main()  {
    Console.WriteLine("Массив с элементами типа int:");
    int[ ] intArray = Methods.randArray(7, 1, 6);
    if (intArray != null) Methods.printArray(intArray, "{0} ");
    Console.WriteLine("Массив с элементами типа double:");
    double[] doubleArray = Methods.randArray(3, 2.0, 8.0);
    if (doubleArray != null) Methods.printArray(doubleArray, "{0:G3}  ");
    Console.WriteLine("Матрица с элементами типа int:");
    int[,] intMatr = Methods.randMatrix(3, 4, 3, 9);
    if (intMatr != null) Methods.printMatrix(intMatr, "{0} ");
    Console.WriteLine("Матрица с элементами типа double:");
    double[,] doubleMatr = Methods.randMatrix(2, 4, 3.0, 9.0);
    if (doubleMatr != null) Methods.printMatrix(doubleMatr, "{0:F3}  ");
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}
}
```

В программе нет никаких тонкостей. Последовательно с помощью методов из библиотечного класса Methods randArray(), printArray(), randMatrix(), printMatrix() создаются и выводятся на экран вначале одномерные массивы, затем матрицы.

Результаты выполнения программы:

```
1 4 3 5 3 3 2
Массив с элементами типа double:
2,79  6,58  5,2
Матрица с элементами типа int:
3 7 6 8
6 6 5 7
5 5 7 8
Матрица с элементами типа double:
3,793  7,580  6,204  8,205
6,580  6,352  5,302  7,745
Для выхода из программы нажмите ENTER.
<ENTER>
```

Перед обращением к каждому методу печати проверяется значение соответствующей ссылки на экземпляр массива. Если значение ссылки отлично от **null**, то массив явно существует, и его можно печатать. Необходимость в проверках обусловлена тем, что при создании каждого массива могли быть ошибки в аргументах, и массив не был создан.

Задача 08-04. Разработать библиотечный метод для «чтения» из строки значения вещественного числа типа **double**. Предусмотреть возможность в записи числа (в строке) отделять целую часть от дробной как точкой, так и запятой.

Похожую задачу чтения с клавиатуры чисел, как в европейской, так и в американской записи, мы уже решали (задача 07-05). Но предлагаемый сейчас метод чтения не от клавиатуры (не из входного потока), а из строки можно использовать не только в консольных приложениях, но и, например, для получения данных из полей форм. Поэтому такой метод полезно иметь в библиотеке.

Пусть код нашего метода будет корректировать только одну ошибку – неверное однократное применение точки или запятой в записи числа. В остальных случаях, а их много, синтаксический анализ строкового представления чисел поручим библиотечному методу **double**.TryParse(). Тогда код метода может быть таким:

```
// Метод для чтения вещественного числа из строки с учетом
// американской (точка) и европейской (запятая) записей чисел.
public static bool readDouble(string st, out double res)
{
    if (double.TryParse(st, out res) == true) return true;
    string temp = null;
    int p = st.IndexOf('.');              // ищем  точку
    if (p >= 0)                           // в стороке есть точка
       temp = st.Replace('.', ',');       // заменяем точку запятой
     else
    {
       p = st.IndexOf(',');                     // ищем запятую
       if (p >= 0)                              // в стороке есть запятая
          temp = st.Replace(',', '.');          // заменяем запятую точкой
       else return false;
    }
    if (double.TryParse(temp, out res) == true) return true;
    return false;
}  // readDouble()
```

Параметры метода readDouble() – строка с записью числа и вещественная переменная с модификатором **out**. С помощью второго параметра метод возвращает прочитанное из строки значение. Имя параметра res с модификатором **out** в теле метода используется в качестве аргумента в обращениях к методу **double**.TryParse(). Если попытки чтения не удаются, то метод возвращает в точку вызова логическое значение **false**. Таким образом, применение метода readDouble() аналогично использованию метода **double**.TryParse(). Комментарии в коде поясняют детали.

Поместим метод readDouble() в нашу библиотеку Library (в класс Methods) и выполним ее компиляцию. Для иллюстрации возможностей метода используем следующую программу, построенную как консольное приложение Program_4 в решении Тема_08.

```
// 08_04.cs - тестирование библиотечного метода readDouble()
using System;
using Library; // Пространство имен библиотеки
class Program   {
static void Main( )    {
    double x;
    Methods.readDouble("3.14159", out x);
    Console.WriteLine("Значение 3.14159: {0:g6}", x);
    do              // цикл повторения решений
    {
       do  Console.Write("Введите значение x =  ");
       // Прочитать вещественное значение x:
       while (!(Methods.readDouble(Console.ReadLine(), out x)));
       Console.WriteLine("Значение: {0:g6}", x);
       Console.WriteLine("Для выхода нажмите клавишу ESC");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
}
```

Результаты выполнения программы:

```
Значение 3.14159: 3,14159
Введите значение x =  33,66.44<ENTER>
Введите значение x =  33,44<ENTER>
Значение: 33,44
Для выхода нажмите клавишу ESC
<ENTER>
Введите значение x =  3.234<ENTER>
Значение: 3,234
Для выхода нажмите клавишу ESC
<ESC>
```

Первое обращение к методу Methods.readDouble() не требует ввода с клавиатуры – метод преобразует в числовое значение типа **double** строковый литерал “3.14159”. В диалоге с программой пользователь в ответ на приглашение "Введите значение x =" ввел в первый раз неверную запись числа 33,66.44 и программа «предложила» вновь ввести значение x.

Задача 08-05. Разработать библиотечные методы для ввода с клавиатуры значений элементов одномерных и двумерных целочисленных массивов.

Метод для чтения с клавиатуры значений элементов одномерного целочисленного массива:

```
// Метод для ввода элементов целочисленного массива:
public static void readArray(int[] a) {
    string str;
    int temp;
    for (int i = 0; i < a.Length; i++      {
        Console.Write(“[“ + i + “] = “);
        str = Console.ReadLine();
        if (!int.TryParse(str, out temp)) i--;
        else  a[i] = temp;
        }
} // readArray
```

В методе обратите внимание на анализ результата обращения к библиотечному методу **int**.TryParse(). Если пользователь сделает ошибку при вводе и изображение числа в прочитанной с консоли строке str будет неверным, то очередной элемент массива значения не получает, параметр цикла уменьшается на 1 и повторяется запрос на ввод нового значения.

```
// Метод для ввода элементов целочисленной матрицы:
public static void readMatrix(int[,] matr)  {
    string str;
    int temp;
    for (int i = 0; i < matr.GetLength(0); i++, Console.WriteLine())
     for (int j = 0; j < matr.GetLength(1); j++)    {
        Console.Write(“[“ + i + “, “+j+”] = “);
        str = Console.ReadLine();
        if (!int.TryParse(str, out temp)) j--;
        else matr[i,j] = temp;
        }
}         // readMatrix
```

Метод readMatrix() для чтения с клавиатуры значений элементов матрицы отличается от предыдущего метода ввода одномерного массива только вторым (вложенным) циклом, где перебираются столбцы матрицы. Ну, естественно, у элементов массива два индекса.

Для иллюстрации применения методов чтения значений элементов массивов создадим следующую программу (проект Program_5 в решении Тема_08):

```
// 08_05.cs - ввод элементов целочисленных массивов
using System;
using Library; // Пространство имен библиотеки
class Program  {
static void Main()  {
  int[] intArray = new int[4];
  Console.WriteLine("Введите элементы одномерного массива:");
  Methods.readArray(intArray);
  Console.WriteLine("Введены элементы массива типа int[]:");
  Methods.printArray(intArray, "{0}  ");
  int [,] intMatr = new int [2,3];
  Console.WriteLine("Введите элементы двумерного массива:");
  Methods.readMatrix(intMatr);
  Console.WriteLine("Введены элементы массива типа int[,]:");
  Methods.printMatrix(intMatr, "{0}  ");
  Console.WriteLine("Для выхода из программы нажмите ENTER.");
  Console.ReadLine();
}
}
```

В программе созданы два массива фиксированных размеров. Для чтения значений их элементов с клавиатуры выполняется обращение к методам readArray() и readMatrix(). Затем с помощью двух других библиотечных методов выводятся значения элементов одномерного массива и матрицы.

Результаты выполнения программы поясняют особенности ввода:

```
Введите элементы одномерного массива:
[0] = 3,7<ENTER>
[0] = 3<ENTER>
[1] = 7<ENTER>
[2] = 3ghj9<ENTER>
[2] = 3<ENTER>
[3] = 9<ENTER>
Введены элементы массива типа int[]:
3 7 3 9
Введите элементы двумерного массива:
```

```
[0, 0] = 4<ENTER>
[0, 1] = 7<ENTER>
[0, 2] = 9<ENTER>

[1, 0] = 2m7<ENTER>
[1, 0] = 2<ENTER>
[1, 1] = 7<ENTER>
[1, 2] = 3<ENTER>

Введены элементы массива типа int[,]:
4 7 9
2 7 3
Для выхода из программы нажмите ENTER.
<ENTER>
```

При вводе значений некоторых элементов допущены ошибки и это приводит к повторению ввода значений тех же элементов.

Задача 08-06. Разработать библиотечные методы для ввода с клавиатуры значений элементов одномерных и двумерных вещественных массивов.

В отличие от чтения значений элементов целочисленных массивов при кодировании этих методов используем уже созданный нами в задаче 08-04 и помещенный в библиотеку статический метод readDouble() класса Methods.

```
// Метод для ввода элементов вещественного массива:
public static void readArray(double[] a)  {
   string str;
   double temp;
   for (int i = 0; i < a.Length; i++)     {
      Console.Write("[" + i + "] = ");
      str = Console.ReadLine();
      if (!readDouble(str, out temp)) i--;
      else a[i] = temp;
      }
}     // readArray

// Метод для ввода элементов вещественной матрицы:
public static void readMatrix(double[,] matr)
{
   string str;
   double temp;
   for (int i = 0; i < matr.GetLength(0); i++, Console.WriteLine())
```

```
      for (int j = 0; j < matr.GetLength(1); j++)      {
         Console.Write("[" + i + ", " + j + "] = ");
         str = Console.ReadLine();
         if (!readDouble(str, out temp)) j--;
         else matr[i, j] = temp;
         }
}     // readMatrix
```

Так как и приведенные методы readArray() и readMatrix() и метод readDouble() принадлежат одному классу Methods, то в обращениях к методу readDouble() опущен префикс «Methods.». Это было бы ошибкой, если бы методы принадлежали разным классам.

Консольное приложение Program_6, иллюстрирующее возможности методов «чтения» вещественных массивов:

```
// 08_06.cs - ввод элементов вещественных массивов
using System;
using Library; // Пространство имен библиотеки
class Program   {
static void Main( ) {
   double[] realArray = new double[3];
   Console.WriteLine("Введите элементы вещественного массива:");
   Methods.readArray(realArray);
   Console.WriteLine("Введены элементы массива типа double[]:");
   Methods.printArray(realArray, "{0:f2}  ");
   double[,] realMatr = new double[2, 3];
   Console.WriteLine("Введите элементы двумерного массива:");
   Methods.readMatrix(realMatr);
   Console.WriteLine("Введены элементы массива типа double[,]:");
   Methods.printMatrix(realMatr, "{0:f2}  ");
   Console.WriteLine("Для выхода из программы нажмите ENTER.");
   Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
Введите элементы вещественного массива:
[0] = 3,7<ENTER>
[1] = 4m,7<ENTER>
[1] = 4.6<ENTER>
[2] = 5,8<ENTER>
Введены элементы массива типа double[]:
3,70  4,60  5,80
Введите элементы двумерного массива:
[0, 0] = 4,8<ENTER>
```

[0, 1] = 3.7<ENTER>
[0, 2] = 5..44<ENTER>
[0, 2] = 5.4<ENTER>
[1, 0] = 4<ENTER>
[1, 1] = 7,8<ENTER>
[1, 2] = 9.4<ENTER>
Введены элементы массива типа double[,]:
4,80 3,70 5,40
4,00 7,80 9,40
Для выхода из программы нажмите ENTER.
<ENTER>

Задача 08-07. Разработать и поместить в библиотеку методы для формирования строкового представления значений элементов одномерных массивов с целочисленными и вещественными элементами, а также методы для представления в виде массива строк элементов матриц.

По существу, задача создания таких методов совпадает с задачами 08-01 и 08-02, где построены методы для вывода в выходной поток строковых представлений значений элементов массивов. Отличие только в возвращаемом результате. Для одномерного массива нужно вернуть строку с изображениями значений элементов. Для матрицы будем возвращать массив строк, каждая из которых содержит символьное представление значений элементов соответствующей строки матрицы. Вывода в консольный поток методы выполнять не должны.

```
// Метод для "изображения" элементов целочисленного массива:
public static string arrayToString(int [] a, string format  {
    string res = "";
    for (int i = 0; i < a.Length; i++)
    res += string.Format(format, a[i]);
    return res;
}  // arrayToString

// Метод для "изображения" элементов вещественного массива:
public static string arrayToString(double[] a, string format)  {
    string res = "";
    for (int i = 0; i < a.Length; i++)
    res += string.Format(format, a[i]);
    return res;
}  // arrayToString
```

```
// Метод для "изображения" элементов целочисленной матрицы:
public static string[] matrToString(int[,] ar, string format)  {
    string[] lines = new string[ar.GetUpperBound(0) + 1];
    for (int i = 0; i < ar.GetLongLength(0); i++)  {
        lines[i] = "";
        for (int j = 0; j <= ar.GetUpperBound(1); j++)
            lines[i] += string.Format(format, ar[i, j]);
       }
    return lines;
}  // matrToString

// Метод для "изображения" элементов вещественной матрицы:
public static string[] matrToString(double[,] ar, string format)  {
    string[] lines = new string[ar.GetUpperBound(0) + 1];
    for (int i = 0; i < ar.GetLongLength(0); i++)  {
        lines[i] = "";
        for (int j = 0; j <= ar.GetUpperBound(1); j++)
            lines[i] += string.Format(format, ar[i, j]);
       }
        return lines;
}  // matrToString
```

Разместив методы в библиотечном классе Methods и оттранслировав библиотеку классов Library.dll, можно продемонстрировать выполнение методов, подготовив следующую программу:

```
// 08_07.cs - тестирование методов
using System;
using Library;  // Пространство  имен  библиотеки
class Program  {
static void Main()  {
    Console.WriteLine("Массив с элементами типа int:");
    int[] intAr = new int [] {9, 7, 5, 3, 1, 0};
    string elem = Methods.arrayToString(intAr, "{0}  ");
    Console.WriteLine(elem);

    Console.WriteLine("Массив с элементами типа double:");
    double[] doubAr = new double[] { 9, 7, 5, 3, 1, 0 };
    string line = Methods.arrayToString(doubAr, "{0:F2}  ");
    Console.WriteLine(line);

    Console.WriteLine("Матрица с элементами типа int:");
    int[,] intMatr = Methods.randMatrix(2, 6, 1, 6);
    string[] image = Methods.matrToString(intMatr, "{0}  ");
    for (int i = 0; i < image.Length; i++)
       Console.WriteLine(image[i]);

    Console.WriteLine("Матрица с элементами типа double:");
```

```
    double[,] doubleMatr = Methods.randMatr(2, 3, 1.0, 6.0);
    string[] list = Methods.matrToString(doubleMatr, "{0:G3}  ");
    Methods.printArray(list, "{0} \n");
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
}  // Main()
}  // class Program
```

Результаты выполнения программы:

```
Массив с элементами типа int:
9 7 5 3 1 0
Массив с элементами типа double:
9,00 7,00 5,00 3,00 1,00 0,00
Матрица с элементами типа int:
3  5  4  2  1  1
5  4  1  3  1  4
Матрица с элементами типа double:
3,81  5,97  4,76
2,96  1,19  1,95
Для выхода из программы нажмите ENTER.
<ENTER>
```

Текст программы почти «линеен» и состоит из последовательности обращений к библиотечным методам. При выводе строк с изображениями элементов матриц в первом случае использован явный цикл, а во втором – массив list с элементами типа **string** выводится с помощью библиотечного метода. В этом случае стоит обратить внимание на параметр форматирования: **Methods.printArray(list, "{0} \n");.** Если убрать из строки формата эскейп-последовательность \n, то строки массива при выводе разместятся подряд.

Отметим, что методы преобразования массивов в строки удобны, например, для программ, создаваемых на основе Windows Forms.

ТЕМА 09

КЛАССЫ И ИХ ОБЪЕКТЫ

Задача 09-01. Периметр P правильного n-угольника, описанного около окружности радиуса r, равен 2*n*r*tg(PI/n), площадь этого многоугольника S равна n*r*r*tg(PI/n). Ввести значения n и r, проверить их корректность и вывести полную информацию о многоугольнике. Проверяемые при вводе условия: n >= 3 и r > 0.

Данную задачу можно решать средствами процедурного программирования. Однако в программах данной темы будем применять объектный подход и использовать объекты классов, создаваемых программистом.

Поэтому для решения сформулированной задачи определим класс "правильный многоугольник, описанный около окружности заданного радиуса". Назовем этот класс Polygon. Закрытые поля объектов класса: радиус окружности, число сторон многоугольника. Свойства: периметр многоугольника, площадь многоугольника. Кроме этих членов в класс войдет конструктор общего вида и нестатический метод ToString() для получения в виде строки полной информации о конкретном многоугольнике.

Кроме класса Polygon в программу войдет класс Program со статическим методом Main(). Назначение этого метода – обеспечить диалог с пользователем, создавать объекты-многоугольники и выводить сведения о них.

Программа в целом может быть такой:

```
// 09_01.cs – Правильный многоугольник
using System;
public class Polygon{       // Класс многоугольников
   int numb;                // Число сторон
   double radius;           // Радиус вписанной окружности
public Polygon(int n, double r)   { // конструктор
        numb = n;
        radius = r;
        }
public double Perimeter{ // Периметр  многоугольника
```

```
    get {  // аксессор  свойства
          double term = Math.Tan(Math.PI / numb);
          return 2 * numb * radius * term;
           }
   }
public double Area         { // Площадь  многоугольника
   get              {  // аксессор  свойства
         return Perimeter * radius / 2;
     }
   }
public new string ToString()        {  // Перегрузка  метода
   string res =
   string.Format("N={0}; R={1}; P={2,2:F3}; S={3,4:F3}",
   numb, radius, Perimeter, Area);
   return res;
}
}  // Polygon

class Program   {
static void Main() {
    Polygon polygon;
    double rad;
    int number;
    do  {
          do Console.Write("Введите число сторон: ");
          while (!int.TryParse(Console.ReadLine(), out number)
                  | number< 3);
          do Console.Write("Введите радиус: ");
          while (!double.TryParse(Console.ReadLine(), out rad)
                      | rad< 0);
          polygon = new Polygon(number, rad);
          Console.WriteLine("Сведения о многоугольнике:");
          Console.WriteLine(polygon.ToString());
          Console.WriteLine("Для выхода нажмите клавишу ESC");
     } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}  // Main()
}  // Program
```

Результаты выполнения программы:

```
Введите число сторон: 7<ENTER>
Введите радиус: 3<ENTER>
Сведения о многоугольнике:
N=7; R=3; P=20,226; S=30,339
Для выхода нажмите клавишу ESC
<ENTER>
Введите число сторон: 4<ENTER>
Введите радиус: 10<ENTER>
```

Сведения о многоугольнике:
N = 4; R = 10; P = 80,000; S = 400,000
Для выхода нажмите клавишу ESC
<ESC>

Класс Polygon содержит все перечисленные выше члены. Поля **int** numb (число сторон) и **double** radius (радиус вписанной окружности) по умолчанию имеют статус доступа **private** и тем самым закрыты («невидимы») извне класса и его объектов. Свойства **double** Perimeter (периметр многоугольника) и **double** Area (площадь многоугольника), конструктор Polygon(**int** n, **double** r) и метод ToString() объявлены с модификатором **public**, то есть доступны для внешнего доступа и в совокупности составляют внешний интерфейс класса и его объектов. В каждом из свойств класса присутствует только один аксессор **get**, так как изменение свойств извне не имеет смысла. В аксессоре свойства Perimeter выполнено обращение к полям объекта. А аксессор свойства Area содержит обращение к свойству Perimeter. У конструктора объектов класса два параметра, определяющие число сторон многоугольника и радиус вписанной окружности. В объявлении метода ToString() обратите внимание на модификатор **new**. Он «сообщает» компилятору, что данный метод «экранирует» метод с тем же именем, который наследуется каждым классом от общего базового класса **Object**.

Статический метод Main() находится в классе Program. В теле этого метода объявлена ссылка Polygon polygon, с которой в процессе выполнения программы ассоциируются конкретные объекты, представляющие многоугольники. Для создания этих объектов используется конструктор, вызываемый в операторе

polygon = newPolygon(number, rad);

Аргументы конструктора – переменные **double** rad, **int** number, объявленные в методе Main(), получают значения при диалоге с пользователем. После выполнения приведенного оператора присваивания переменная polygon представляет очередной созданный объект класса. Для вывода на консоль сведений о соответствующем многоугольнике выполняется обращение к методу ToString() в аргументе метода Console.WriteLine().

Результаты диалога с программой и комментарии в тексте программы дополняют приведенные пояснения.

Разрабатывая программы с классами полезно использовать для графического представления классов символику UML (Unified Modeling Language – Унифицированный язык моделирования). Среда разработки MS VS позволяет получить изображения классов с помощью диаграммы классов. Для её построения в окне обозревателя решения (Solution Explorer) щелкните правой кнопкой на имени проекта. В выпадающем меню выберите пункт View Class Diagram. Результат построения диаграммы классов приведён на рис. 9.1.

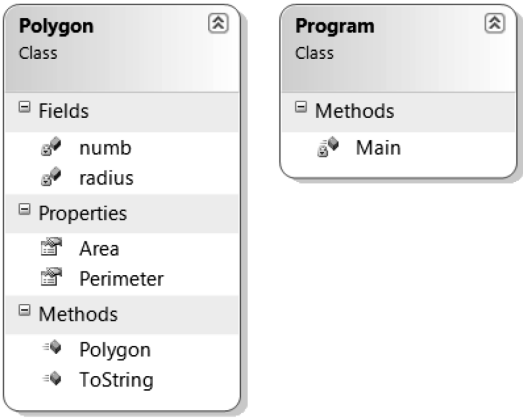


Рис. 9.1. Графическое представление классов программы 09-01

Как видно из диаграммы, в программе два класса Poligon и Program. В классе Poligon показаны поля (Fields), свойства (Properties) и методы (Methods). Поля numb и radius специальным значком обозначены как закрытые (**private**). Отсутствие этого специального значка означает, что свойства Area и Perimeter открыты (**public**) для «внешних» обращений. Методы Poligon() и ToString() также обозначены как открытые (**public**). В классе Program нет ни свойств, ни полей. Метод Main() обозначен как закрытый (**private**). К нему может получить доступ только исполняющая система.

В предыдущей программе с классом правильных многоугольников все объекты при выделении им памяти занимают участки одинаковых размеров, и каждый объект содержит только поля с типами значений. Это зачастую не так. Объекты класса вполне

могут иметь поля с типами ссылок, и эти ссылки могут быть связаны с объектами (например, массивами) разных размеров. В качестве примера рассмотрим класс, каждый объект которого «хранит» целое неотрицательное число и представление его значения в виде массива символов-цифр шестнадцатеричной системы счисления.

Задача 09-02. Определить класс HexNumber – "число и его шестнадцатеричные цифры". Поля класса (его объектов): целое неотрицательное число (number), шестнадцатеричное представление числа в виде массива символов (hexView). Свойства: Number – для доступа к значению числа (к полю number); HexView – для обращения к массиву (hexView) его шестнадцатеричных символов (записи числа); Record – для получения строкового представления числа в шестнадцатеричном виде.

В основной программе определить объект класса и, вводя разные числовые значения его свойства Number, изменять «состояние» объекта и выводить сведения о нем.

Программа, решающая сформулированную задачу, может быть такой:

```
//09_02.cs – Класс "число и его шестнадцатеричные цифры"
using System;
public class HexNumber{ // представление неотрицательных целых
   uint number;              // целое неотрицательное число
   char [ ] hexView;         // Шестнадцатеричное представление
public HexNumber(uint n)     { // конструктор общего вида
   number = n;
   hexView = series(n);
   }
public HexNumber(): this(0) {  }   // конструктор умолчания
public uint Number           {          // Свойство: десятичное целое
   get { return number; }
   set {
        number = value;
        hexView = series(value);
        }
   }
public char[ ] HexView   {        // Свойство: массив символов-цифр
   get { return hexView; }
}
public string Record {        // Свойство: строковое представление
   get {
```

```
    string str = new String(hexView);
    return "0x" + str;
    }
}

// Возвращает массив шестнадцатеричных цифр параметра:
char[ ] series(uint num)  {
    int arLen = num == 0 ? 1 : (int)Math.Log(num, 16) + 1;
    char[ ] res = new char[arLen];
    for (int i = arLen - 1; i >= 0; i--)  {
        uint temp = (uint)(num % 16);
        if(temp >= 0 & temp <=9) res[i] = (char)('0'+temp);
        else res[i] = (char)('A'+temp % 10);
        num /= 16;
        }
    return res;
}  // series
}  // HexNumber


class Program  {
static void Main()  {
    HexNumber hex;       // ссылка с типом класса
    hex = new HexNumber(0);      // объект класса
    uint number;
    while (true){ // цикл для ввода разных значений числа
        do Console.Write("Введите целое неотрицательное число:  ");
        while (!uint.TryParse(Console.ReadLine(), out number));
        hex.Number = number;    // Изменяем объект через свойство
        Console.WriteLine("Свойство Number: " + hex.Number);
        Console.Write("Шестнадцатеричные цифры числа: ");
        foreach (char h in hex.HexView) Console.Write("{0} ", h);
        Console.WriteLine("\nШестнадцатеричная константа: "
            + hex.Record);
        Console.WriteLine("Для выхода нажмите клавишу ESC");
        if (Console.ReadKey(true).Key == ConsoleKey.Escape) break;
        }  // while
}  // Main()
}  // Program
```

Результаты выполнения программы:

Введите целое неотрицательное число: 6666<ENTER>
Свойство Number: 6666
Шестнадцатеричные цифры числа: 1 A 0 A
Шестнадцатеричная константа: 0x1A0A
Для выхода нажмите клавишу ESC
<ENTER>

Введите целое неотрицательное число: 255<ENTER>
Свойство Number: 255
Шестнадцатеричные цифры числа: F F
Шестнадцатеричная константа: 0xFF
Для выхода нажмите клавишу ESC
<ESC>

В программе два класса: HexNumber и Program. Первый из них позволяет создавать объекты, каждый из которых содержит два представления неотрицательного целого числа. Соответствующие поля **int** number (целое неотрицательное число) и **char** [] hexView (ссылка на массив символов записи значения числа с шестнадцатеричным основанием) по умолчанию имеют статус **private**. Три открытых (**public**) свойства: **int** Number, **char**[] HexView, **string**, Record и конструктор общего вида HexNumber(**int** n) обеспечивают внешний интерфейс объектов класса. В классе определен закрытый вспомогательный метод **char**[] series(**int** num) для построения массива шестнадцатеричных цифр целочисленного параметра. Обращения к этому методу используются в конструкторе и в аксессоре изменений (**set**-аксессор) свойства Number. В определениях свойств HexView и Record такие аксессоры изменений (**set**-аксессоры) отсутствуют. Следовательно, с помощью этих свойств невозможно изменить состояние объекта. Обратите внимание на конструкторы класса, точнее, на конструктор без параметров. При обращении к нему выполняется вызов конструктора с параметром, а в качестве аргумента используется нулевое значение.

В классе Program только один статический метод Main(), где определена переменная (ссылка) hex с типом класса HexNumber и переменная **uint** number. Переменной hex присвоен результат обращений к конструктору с нулевым аргументом. Создаваемый объект представляет число с нулевым значением. Следующие операторы поддерживают в цикле диалог с пользователем. На каждой итерации вводится значение переменной number, которая используется для обращения к свойству Number объекта, связанного с переменной hex. Приведенные результаты выполнения программы иллюстрируют возможности свойств. Обратите внимание, что изменять состояние объекта можно только с помощью свойства Number. Для того чтобы можно было бы изменять объект с помощью свойств HexView и Record, необходимо дополнить объявления этих свойств аксессорами изме-

нений (**set**-аксессорами). Читатель может проделать это самостоятельно.

Задача 09-03. Определить класс Circle – «окружность на плоскости». Радиус окружности и координаты центра окружности должны быть определены неявно с помощью автореализуемых свойств: Radius– радиус окружности; Xc – абсцисса центра; Yc – ордината центра. В классе объявить нестатический метод для оценки пересечения двух окружностей.

В основной программе определить и инициализировать массив ссылок на объекты класса “окружность на плоскости”. В диалоге: вводить значения радиуса и координат центра окружности и выводить информацию о тех окружностях из массива, которые с нею пересекаются.

```
// 09_03 – Массив ссылок на объекты класса “Окружность”
using System;
public class Circle {                  // Окружность на плоскости
    public double Radius               // Свойство - радиус
    { get; set; }
    public double Xc                   // Свойство – координата  центра
    { get; set; }
    public double Yc                   // Свойство – координата  центра
    { get; set; }
// Конструктор:
    public Circle(double xi, double yi, double ri)    {
     Xc = xi; Yc = yi;
     Radius = ri;
     }
// Метод проверки пересечения:
    public bool crossing(Circlecir)  {
       double distance = (Xc - cir.Xc) * (Xc - cir.Xc) +
                         (Yc - cir.Yc) * (Yc - cir.Yc);
       double rad2 = (Radius + cir.Radius) *
                          (Radius + cir.Radius);
       if (distance > rad2) return false;
       else return true;
    }
}  // class Circle

class Program  {
static void Main()  {
    Circle[] set = { new Circle(2, 4, 9),
    new Circle(-4, 12, 2), new Circle(5, -8, 3),
    new Circle(4, 9, 7), new Circle(7, 8, 11),
    new Circle(11, 4, 8), new Circle(4, -6, 10)};
```

```
    double rad, xc, yc;
    Circle newCirc;
    do  { newCirc = null;
    Console.WriteLine("Введите сведения об окружности: ");
    do Console.Write("xc=");
    while (!double.TryParse(Console.ReadLine(), out xc));
    do Console.Write("yc=");
    while (!double.TryParse(Console.ReadLine(), out yc));
    do Console.Write("rad=");
    while (!double.TryParse(Console.ReadLine(),
                  outrad) || rad< 0);
    // Создаем объект класса "Окружность":
    newCirc = new Circle(xc, yc, rad);
    bool flag = false;
    foreach (Circle cir in set)  {
        if (newCirc.crossing(cir))  {
        if (!flag)
        Console.WriteLine("С этими окружностями пересекается:");
        flag = true;
        Console.WriteLine("xc={0} yc={1} radius={2}",
        cir.Xc, cir.Yc, cir.Radius);
        }
    }
    if(!flag) Console.WriteLine("Окружности не пересекаются!");
    Console.WriteLine("Для выхода нажмите клавишу ESC");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
  }  // Main()
}  // classProgram
```

Результаты выполнения программы:

```
Введите сведения об окружности:
xc=66<ENTER>
yc=77<ENTER>
rad=5<ENTER>
Окружности не пересекаются!
Для выхода нажмите клавишу ESC
<ENTER>
Введите сведения об окружности:
xc=2<ENTER>
yc=5<ENTER>
rad=4<ENTER>
С этими окружностями пересекается:
xc=2 yc=4 radius=9
xc=4 yc=9 radius=7
xc=7 yc=8 radius=11
xc=11 yc=4 radius=8
```

xc=4 yc=-6 radius=10
Для выхода нажмите клавишу ESC
<ESC>

В классе Circle пять открытых членов: автореализуемые свойства Radius, Xc, Yc; конструктор общего вида Circle(**double** xi, **double** yi, **double** ri); нестатический метод **public bool** crossing(Circle cir). Так как этот метод нестатический, то его можно вызвать только для существующего объекта класса Circle и предать ему в качестве аргумента ссылку на другой объект этого же класса. Свойства имеют тип **double**, что предполагает, что вещественными являются и соответствующие им поля. При объявлении автореализуемого свойства компилятор создает закрытое поле того же типа для «поддержки» именно этого свойства. Свойство обеспечивает запись и чтение значения поддерживающего поля без проверки правильности значений и без какой-либо обработки. Поддерживающие поля данных скрыты за соответствующими им свойствами. Имена полей неизвестны программисту. Поэтому в конструкторе и в методе crossing() для изменения или получения значений полей объектов используются обращения к свойствам.

В основной программе, в методе Main() объявлена ссылка Circle[] set на массив ссылок на объекты класса Circle. Массив инициализирован списком из семи выражений, создающих объекты класса Circle. Далее объявлены три вещественные переменные rad, xc, yc. Они служат далее для представления радиуса и координат центра новой окружности. В цикле с постусловием пользователь вводит значения переменных rad, xc, yc, которые затем используются в качестве аргументов при обращении к конструктору Circle(). Созданный при этом объект представлен далее в коде ссылкой newCirc.

После создания объекта в цикле **foreach** выполняется перебор массива ссылок на объекты класса Circle. Параметр cir (переменная) цикла **foreach** имеет тип Circle. В теле цикла результат обращения к методу newCirc.crossing(cir) анализируется в условном операторе. Если окружности, представляемые ссылками newCirc и cir, пересекаются, то метод crossing() возвращает значение **true**, и сведения об окружности выводятся в консольное окно. Чтобы фраза о пересечении окружностей “С этими окружностями пересекается:” выводилась только один раз, объявлена и применяется вспомогательная переменная **bool**

flag. Другие особенности кода поясняют результаты выполнения программы.

Задача 09-04. Определить класс «материальная точка в трехмерном пространстве». Поля объектов: координаты точки, масса точки. Поля класса – центр масс всех точек, число точек (всех созданных объектов класса). Метод для вывода информации о центре масс. В диалоге нужно вводить значения координат и массу очередной точки, выводить центр масс точек, который изменяется при добавлении точки.

В механике под материальной точкой понимают тело, размерами и формой которого в конкретном случае можно пренебречь. Для представления материальной точки достаточно знать ее координаты (x, y, z) и массу m. Известно, что центр масс системы материальных точек можно представить как отдельную (воображаемую) материальную точку, масса которой m_c равна сумме масс всех точек системы, а координаты (x_c, y_c, z_c,) вычисляются следующим образом:

$$x_c = \frac{1}{m_c}\sum x_i m_i;\ y_c = \frac{1}{m_c}\sum y_i m_i;\ z_c = \frac{1}{m_c}\sum z_i m_i.$$

При добавлении новой точки (x', y', z', m') в существующую систему материальных точек масса системы становится равной:

$$m'_c = m_c + m'.$$

Координаты центра масс пересчитываются по следующим формулам:

$$x'_c = (x_c \times m_c + x' \times m')/(m_c + m'),$$
$$y'_c = (y_c \times m_c + y' \times m')/(m_c + m'),$$
$$z'_c = (z_c \times m_c + z' \times m')/(m_c + m').$$

где (x_c, y_c, z_c, m_c) – координаты и масса центра масс до добавления точки, (x'_c, y'_c, z'_c, m'_c) – значения после добавления точки.

```
// 09_04.cs – материальная точка в трехмерном пространстве
using System;
public class Mass_point{       // материальная точка
    double x, y, z;            // координаты точки
    double mass;               // масса точки
```

```
    static Mass_point mass_center;  // Центр масс
static Mass_point()                 // статический конструктор
    {  mass_center = newMass_point(); }
Mass_point( )             // конструктор без параметров
      { x = y = z = mass = 0.0; }
    //  Конструктор общего вида:
public Mass_point(double xi, double yi, double zi, double mi)   {
    x = xi; y = yi; z = zi; mass = mi;
    double newMass = mass_center.mass + mi;
    mass_center.x = (mass_center.x *
       mass_center.mass + xi * mi) / newMass;
    mass_center.y = (mass_center.y *
       mass_center.mass + yi * mi) / newMass;
    mass_center.z = (mass_center.z *
       mass_center.mass + zi * mi) / newMass;
    mass_center.mass = newMass;
}
// Метод для вывода центра масс:
public static void printCenter()  {
    string format = "xc={0,4:F3}; yc={1,4:F3};" +
                    " zc={2,4:F3}; mc={3,4:F3}";
    string res = string.Format(format, mass_center.x,
       mass_center.y, mass_center.z, mass_center.mass);
    Console.WriteLine(res);
  }
}  // Mass_point

class Program      {
static void Main()     {
    Mass_point mp = null;
    double x, y, z, m;
    do  {
          Console.WriteLine("Введите координаты и массу точки: ");
          do Console.Write("x=");
          while (!double.TryParse(Console.ReadLine(), out x));
          do Console.Write("y=");
          while (!double.TryParse(Console.ReadLine(), out y));
          do Console.Write("z=");
          while (!double.TryParse(Console.ReadLine(), out z));
          do Console.Write("m=");
          while (!double.TryParse(Console.ReadLine(), out m));
          mp = new Mass_point(x, y, z, m);
          Console.WriteLine("Сведения о центре масс всех точек:");
          Mass_point.printCenter();
          Console.WriteLine("Для выхода нажмите клавишу ESC");
          } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
     }  // Main()
}  // Program
```

Результаты выполнения программы:

```
Введите координаты и массу точки:
x=1<ENTER>
y=1<ENTER>
z=1<ENTER>
m=1<ENTER>
Сведения о центре масс всех точек:
xc=1,000; yc=1,000; zc=1,000; mc=1,000
Для выхода нажмите клавишу ESC
<ENTER>
Введите координаты и массу точки:
x=-1<ENTER>
y=-1<ENTER>
z=-1<ENTER>
m=1<ENTER>
Сведения о центре масс всех точек:
xc=0,000; yc=0,000; zc=0,000; mc=2,000
Для выхода нажмите клавишу ESC
<ESC>
```

В классе Mass_point вещественные поля *x*, *y*, *z*, mass представляют координаты и массу отдельной материальной точки – объекта класса Mass_point. Статическое поле mass_center – ссылка на объект класса Mass_point, который будет представлять общий для всех объектов класса центр масс системы точек, представляемых объектами. Для инициализации этой ссылки в класс введен статический конструктор, в теле которого ссылке mass_center присваивается результат обращения к нестатическому конструктору без параметров. Тем самым создается объект с нулевыми значениями полей – основа для «накапливания» сведений о центре масс. В конструкторе общего вида с помощью параметров определяются значения полей очередного объекта «материальная точка» и изменяется объект, представляющий центр масс. Статический метод printCenter() предназначен для вывода сведений о центре масс.

В классе Program объявлены вещественные переменные *x*, *y*, *z*, *m* (для ввода сведений о новой материальной точке) и переменная *mp* – ссылка на объект класса Mass_point, представляющий эту точку. Еще раз обратим внимание, что при каждом обращении к конструктору общего вида изменяются значения полей объекта, представляющего центр масс, поэтому метод Mass_point.printCenter() всегда выводит актуальную информацию о центре масс.

Задача 09-05. Определить классы «Точка на плоскости» и «Прямая на плоскости». Поля класса «точка на плоскости» должны представлять ее декартовы координаты. Поля класса «прямая» должны представлять коэффициенты уравнения прямой на плоскости: A*x + B*y + C = 0. В классе «прямая» определить метод ToString() для строкового представления уравнения прямой с заданными (конкретными) значениями коэффициентов.

В основной программе определить и инициализировать (явно задав значения коэффициентов уравнения) объект класса «прямая». Считывая вводимые пользователем значения координат объекта класса «точка», вычислять уклонение точки от прямой. Напоминаем, что уклонение точки с координатами x_i; y_i от прямой A*x + B*y + C = 0 равно значению выражения $A*x_i + B*y_i + C$.

Вычислять в программе уклонение точки от прямой можно разными способами. Можно, например, непосредственно в методе Main() вычислять выражение, обращаясь к полям или свойствам объектов классов. Можно объявить в одном из классов программы статический метод с параметрами, представляющими объекты классов точка и прямая. Но удобнее в класс «Прямая на плоскости» ввести нестатический метод с параметром – ссылкой на объект класса «Точка на плоскости». Так и поступим.

Программа, решающая задачу:

```
// 09_05.cs – Уклонение точки от прямой
using System;
public class Point  {  // Точка  на  плоскости
// координаты точки (автореализуемые свойства):
      public double X  {  get; set; }
      public double Y  {  get; set; }
      }  // Point
public class Line { // Прямая на  плоскости
// Коэффициенты  уравнения  прямой:
      public double A  {  get; set; }
      public double B  {  get; set; }
      public double C  {  get; set; }
// Конструктор:
public Line (double a, double b, double c)
      {  A = a; B = b; C = c; }
public new string ToString() {
      string format =
```

```
        "Уравнение прямой: {0:F2}*x + {1:F2}*y + {2:F2} = 0";
        return string.Format(format, A,B,C);
}
// Уклонение точки от прямой:
public double deviation (Point pt)
        { return A*pt.X + B*pt.Y + C; }
}  // Line

class Program    {
static void Main()    {
        Point point = new Point();
        Line line = new Line(-2, 3, 6);
        Console.WriteLine(line.ToString());
        double x, y;
        do   {
                Console.WriteLine("Введите координаты точки: ");
                do Console.Write("x=");
                while (!double.TryParse(Console.ReadLine(), out x));
                do Console.Write("y=");
                while (!double.TryParse(Console.ReadLine(), out y));
                point.X = x;
                point.Y = y;
                Console.WriteLine("Уклонение точки от прямой: {0:F2}",
                                        line.deviation(point));
                Console.WriteLine("Для выхода нажмите клавишу ESC");
        } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
    }  // Main()
}  // Program
```

Результаты выполнения программы:

```
Уравнение прямой: -2,00*x + 3,00*y + 6,00 = 0
Введите координаты точки:
x=2<ENTER>
y=7<ENTER>
Уклонение точки от прямой: 23,00
Для выхода нажмите клавишу ESC
<ENTER>
Введите координаты точки:
x=3<ENTER>
y=0<ENTER>
Уклонение точки от прямой: 0,00
Для выхода нажмите клавишу ESC
<ESC>
```

В программе три класса: Point – класс, объекты которого представляют точки на плоскости; Line – класс для пред-

ставления уравнений прямых на плоскости; Program – класс со статическим методом Main(), в котором создаются объекты и ведется диалог с пользователем. В классах Point и Line использован механизм автореализуемых свойств – объявлены открытые свойства X, Y, A, B, C, аксессоры которых не имеют выполняемых операторов тела. (Тело каждого аксессора представлено пустым оператором – точкой с запятой.) Свойствам соответствуют поля, имена которых программисту неизвестны, обращения к ним возможны только через автореализуемые свойства.

В классе Point кроме свойств X и Y нет других членов. Компилятор автоматически добавляет конструктор без параметров, создающий объект-точку с нулевыми значениями полей (координат). В классе Line конструктор определен явно и есть два метода. Параметры конструктора определяют коэффициенты уравнения прямой. Метод ToString() переопределяет (экранирует) метод ToString(), унаследованный каждым классом от класса **Object**. Метод deviation() может быть вызван для существующего объекта класса Line. Его параметр – ссылка на объект класса Point. В теле метода обращения к полям объектов классов Point и Line выполняются через соответствующие свойства. Результат выполнения метода – уклонение точки от прямой – вещественное значение.

В методе Main() практически нет особенностей. Объявляется ссылка point с типом Point, создается соответствующий ей объект с нулевыми значениями полей. Создается ссылка line типа Line и объект, представляющий прямую на плоскости. Запись уравнения прямой, формируемая методом ToString(), выводится на консольный экран. В цикле повторения решений пользователь вводит координаты точки, которые присваиваются соответствующим полям объекта, представляемого ссылкой point. Затем эта ссылка используется в качестве аргумента при обращении к методу deviation(), вызванному для объекта, адресованного ссылкой line. Результат – уклонение точки от прямой – выводится в консольное окно.

Результаты диалога с программой дополняют объяснения кода. На рис. 9.2 приведена диаграмма классов для этой программы.

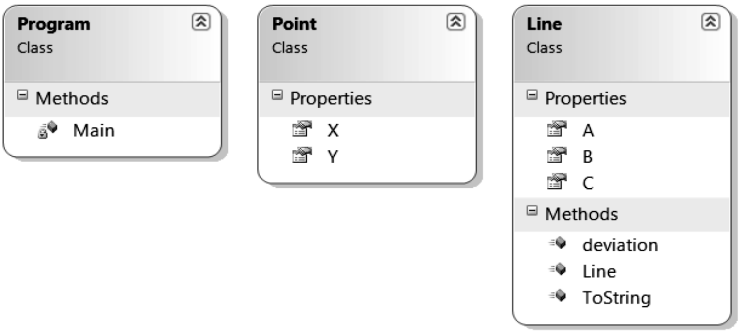


Рис. 9.2. Графическое представление классов программы 09-05

В языке C# каждый массив – это средство для представления данных из фиксированного количества элементов одного типа. Особенность массива – последовательное размещение его элементов. Над массивом определена операция индексирования, позволяющая получить непосредственный доступ к любому из его элементов. Если при выполнении этой операции индекс выходит за пределы (за фиксированные для каждого массива границы), то генерируется исключение. Недостатком или особенностью массива в C# является невозможность после создания массива изменить его размер (число элементов) при дальнейшем выполнении программы.

Задача 09-06. С целью ликвидировать некоторые ограничения стандартных массивов языка C#, разработать класс для представления одномерных массивов с плавающей верхней границей индекса.

Определим функциональность проектируемого класса динамических массивов. Для обращения к его элементам используем индексацию. Нижняя граница индекса пусть будет, как и принято в C#, равна 0. (Это ограничение не обязательно, можно сделать плавающими и обе границы индекса, как, например, в языке Алгол-60, но это добавит не много нового в предлагаемый класс массивов с динамически изменяемыми размерами.) Верхняя граница индекса определяется при создании экземпляра класса и может увеличиваться при присваивании значения элементу, индекс которого превышает верхнюю границу.

В классе динамических массивов нам потребуется внутренняя память для хранения элементов экземпляра (объекта) динамического массива. Размер этой памяти (Capacity – предельная текущая емкость массива) будет меняться при присваивании значения элементам, индекс которых превышает емкость массива. Теоретически размер массива в языке C# может быть произвольно большим. Однако при выполнении программы существуют ограничения, налагаемые средой исполнения на размер непрерывного участка памяти, выделяемого для массива. Поэтому введем верхнюю границу и для индекса при записи в массив. Выберем ее равной Int32.MaxValue/3.

Реальное число элементов в динамическом массиве (Count) пусть определяется максимальным значением индекса, использованном при записи данных в массив. При этом в последовательности элементов динамического массива могут появиться элементы с умалчиваемыми значениями. Нижняя граница индекса и при чтении значения элемента и при присваивании ему значений одинаковы и равны 0. Верхние границы будут различны для записи и для чтения. При записи верхняя граница (определяемая значением Capacity) будет увеличиваться, если индекс ее превысил, но индекс не должен превысить значения Int32.MaxValue/3. При чтении верхняя граница (определяемая значением Count) фиксирована, и выход за нее должен рассматриваться как аварийная ситуация. Другими словами, при записи будем выполнять расширение (в заданных пределах) динамического массива, но читать данные за пределами его верхней границы недопустимо. При попытке обращения к экземпляру динамического массива со значением индекса, который меньше 0 или превышает верхнюю границу (Count – при чтении, Int32.MaxValue/3 – при записи) предусмотрим генерацию исключения IndexOutOfRangeException.

Класс динамических массивов можно создать для элементов разных типов как стандартных, так и определенных пользователем. Наиболее универсальное решение предлагает механизм обобщений. Но для простоты отложим его применение, а создадим класс динамических массивов, элементы которых будут иметь тип **char**. Программа с эти классом и кодом для иллюстрации его возможностей может быть такой:

```
// 09_06.cs - класс динамических массивов
using System;
class cArray{  // класс динамических массивов
// Число элементов в массиве:
      public int Count { get; private set; }
// Размер памяти (ёмкость массива в элементах):
      public int Capacity {get; private set;}
      char [] memory;  // Ссылка на внутренний массив
// Конструктор:
public cArray(int n = 4) {
       Capacity = n;
       Count = 0;
       memory = newchar[n];
}
// Индексатор:
public char this [int ind] {
     get   {
        if(ind > Count | ind< 0)
        throw new IndexOutOfRangeException();
        return memory[ind];
        }
     set {
        if(ind < 0 | ind > Int32.MaxValue / 3)
        throw new IndexOutOfRangeException();
        if(ind < Capacity) {
            memory[ind] = value;
            Count = Count >ind ?Count :ind + 1;
            return; }
        Capacity = ((ind-Capacity)/Capacity+2)*Capacity;
        char [ ] temp = new char[Capacity];
        Array.Copy(memory, temp, memory.Length);
        temp[ind] = value;
        memory = temp;
        Count = ind+1;
        }
  }  //  this []
public string ToString(int n) {
     string res = "";
     for(int i =0; i< Count; i++) {
        res += "["+i+"]=" + memory[i] + "  ";
        if ((i + 1) % n == 0) res += '\n';
        }
        return res;   }
}  // class cArray

class Program     {
static void Main( ) {
   cArray myArray = new cArray();
```

```
    myArray[1] = '1';
    myArray[5] = '5';
    myArray[4] = '4';
    Console.WriteLine(myArray.ToString(3));
    Console.WriteLine("Вводите символы (окончание 0):");
    charch; int t=0;
    do { ch = (char)Console.Read();
        myArray[t++] = ch;
       } while(ch != '0');
    Console.WriteLine(myArray.ToString(5));
    Console.WriteLine("Для выхода нажмите ENTER");
    Console.ReadLine();
}  // Main( )
}  // Program
```

Результаты выполнения программы:

```
[0]=  [1]=1  [2]=
[3]=  [4]=4  [5]=5
Вводите символы (окончание 0):
12345678abcdef0ghi<ENTER>
[0]=1   [1]=2   [2]=3   [3]=4   [4]=5
[5]=6   [6]=7   [7]=8   [8]=a   [9]=b
[10]=c  [11]=d  [12]=e  [13]=f  [14]=0
Для выхода нажмите ENTER
<ENTER>
```

В классе cArray два автореализуемых свойства Count (реальное число элементов в массиве) и Capacity (текущая емкость динамического массива) предназначены только для чтения значений поддерживающих их полей. Точнее, эти свойства закрыты для изменений извне объявления класса. Закрытое (**private**) поле memory – это ссылка на внутренний массив, в котором хранятся элементы динамического массива. Конструктор с заголовком **public** cArray(**int** n = 4) интересен тем, что его параметр n имеет умалчиваемое значение. Параметр определяет размер выделенной в конструкторе памяти для массива, адресуемого ссылкой memory. Тем самым определяется и первоначальная емкость Capacity динамического массива. При создании объекта-массива число элементов в нем (Count) равно нулю. Инициализация внутреннего массива выполняется по умолчанию, как это принято для массивов C#.

Основная функциональность нашего объекта-массива сосредоточена в индексаторе. Его аксессоры не имеют явных модификаторов и поэтому их доступность определена модификатором

public индексатора. Аксессор чтения (**get**-аксессор) проверяет принадлежность значения индекса диапазону существования значащих элементов массива. При выявлении нарушения – генерируется исключение. В противном случае аксессор возвращает значение нужного элемента массива. При выполнении аксессора записи (**set**-аксессора) возможны три ситуации. Если индекс меньше нуля или превысил значение Int32.MaxValue/3 – генерируется исключение; если индекс меньше емкости (Capacity), то в соответствующий элемент массива memory[ind] записывается присваиваемое значение и корректируется общее число элементов Count динамического массива; если индекс превышает или равен емкости динамического массива, то вычисляется новое значение емкости:

Capacity = ((ind-Capacity)/Capacity+2)*Capacity;

Затем создается новый массив больших размеров, адресуемый временной переменной – ссылкой **char** [] temp. После копирования массива memory в начало нового массива, выполняется присваивание temp[ind] = **value**;. Ссылке memory присваивается значение temp и определяется новое значение числа элементов в динамическом массиве (Count).

Класс cArray дополнен методом ToString(**int** n), для представления элементов массива в виде строки, приспособленной для вывода элементов с индексами в консольное окно. Параметр метода – количество элементов в одной строке консольного экрана. В формируемую методом строку вставлены эскей-последовательности ‘\n’, вызывающие переходы на новые строки при выводе.

В основной программе объявлен объект класса cArray, адресуемый ссылкой myArray. Его трем элементам (индексируемым в произвольном порядке) присвоены некоторые значения. Затем в обращении к консольному методу WriteLine() в качестве аргумента используется вызов метода ToString(3) для объекта myArray. В выходной консольный поток выводятся шесть значений элементов динамического массива (по три в каждой экранной строке). Для элементов с индексами 0, 2 и 3 выведены умалчиваемые значения. Далее программа предлагает пользователю вводить произвольные символы, и каждый из них записывается в очередной элемент динамического массива. Окончание ввода – появление во входном потоке символа 0. Результаты одного сеанса работы с программой дополняют объяснения.

ТЕМА 10

ПЕРЕГРУЗКА ОПЕРАЦИЙ

Объектно-ориентированное программирование становится особенно привлекательным в тех случаях, кода программисту-пользователю авторы библиотеки классов дают возможность применять к объектам операции, соответствующие смысловому значению предметной области. Яркий тому пример – класс комплексных чисел, реализованный в библиотеке классов .NET. Но классы, к объектам которых применимы перегруженные операции, могут быть не только в стандартных библиотеках.

В тех случаях, когда объекты в программе отображают набор понятий той предметной области, к которой относится решаемая программой задача, удобно и полезно иметь возможность применять к этим объектам в коде программы операции, специфические для понятий предметной области. Именно в таких случаях применяется механизм перегрузки операций.

Задача 10-01. Определить класс для представления активных сопротивлений электрической цепи. Выполнить в этом классе перегрузку двух операций, соответствующих последовательному и параллельному соединению двух сопротивлений. В основной программе рассчитать общее сопротивление электрической цепи из двух последовательно соединенных групп активных сопротивлений. В первой группе три параллельно соединенных сопротивления R1, R2, R3, во второй – два параллельно соединенных сопротивления R4, R5. Затем вычислить сопротивление другой цепи, в которой группа из трех последовательно соединенных сопротивления (R1, R2, R3) соединена параллельно с группой из двух (R4, R5) последовательно соединенных сопротивления. Для упрощения принять, что значения всех пяти сопротивлений одинаковы и каждое равно одному Ому (1 Ом).

Напомним (см. задачу 06-10), что при последовательном соединении нескольких сопротивлений общее сопротивление цепочки равно сумме сопротивлений. При параллельном соедине-

нии суммируются проводимости, и общее сопротивление цепи вычисляется как обратная величина от суммы проводимостей.

Определим класс «активное электрическое сопротивление» и присвоим классу имя Resistor. Для объекта класса потребуется только одно поле данных **public double** r – значение сопротивления. Конструктор традиционен. Операцию, моделирующую последовательное соединение сопротивлений, обозначим знаком '+', операцию, обозначающую параллельное соединение, – знаком '|'. Соответствующие методы перегрузки операций должны быть открытыми и статическими. Для формирования сведений о сопротивлении в виде строки перегрузим метод ToString(). Обратите внимание на использование в заголовке этого метода модификатора **new**.

Программа с названным классом, решающая поставленную задачу:

```
// 10_01.cs – операции с электрическими сопротивлениями
using System;
class Resistor {
    public double r;
public Resistor(double ri) { r = ri; }
public static Resistor operator + (Resistor r1, Resistor r2)   {
        return new Resistor(r1.r + r2.r);
        }
public static Resistor operator |(Resistor r1, Resistor r2)   {
        double temp = r1.r * r2.r / (r1.r + r2.r);
        return new Resistor(temp);
        }
public new string ToString() {
        return string.Format("R = {0:F2}", r);
        }
}  // Resistor

class Program   {
static void Main( )      {
   Resistor r1 = new Resistor(1);
   Resistor r2 = new Resistor(1);
   Resistor r3 = new Resistor(1);
   Resistor r4 = new Resistor(1);
   Resistor r5 = new Resistor(1);
   Resistor result = (r1 | r2 | r3) + (r4 | r5);
   Console.WriteLine(result.ToString());
   result = r1 + r2 + r3 | r4 + r5;
   Console.WriteLine(result.ToString());
   Console.WriteLine("Для выхода из программы нажмите ENTER.");
   Console.ReadLine();
}  // Main
}  // Program
```

Результаты выполнения программы:

```
R = 0,83
R = 1,20
Для выхода из программы нажмите ENTER.
<ENTER>
```

В основной программе определены пять объектов класса Resistor, ссылке result последовательно присвоены значения выражений (r1|r2|r3)+(r4|r5) и r1+r2+r3|r4+r5. Ранги (приоритеты) операций при перегрузках не изменяются. Поэтому в первом выражении для правильной последовательности выполнения вычислений приходится использовать скобки. Для второй цепи расчетное выражение можно записать без скобок. Для единичных значений всех сопротивлений правильность полученных результатов легко проверить устным подсчетом.

Задача 10-02. Определить класс полиномов с вещественными коэффициентами. Ввести операции для суммирования полиномов и операции для умножения полинома на число (один операнд полином, второй – арифметическое значение) и числа на полином. Предусмотреть метод ToString() для представления записи полинома в виде строки. В строке изображать операцию возведения в степень символом '∧'. Например, полином $P(x) = 2*x^4 - 2*x^3 + 6$ нужно представить в виде «2*x∧4-2*x∧3+6».

Многочленом (полиномом) n-й степени относительно аргумента x называют выражение вида

$$P(x) = a_0*x^n + a_1*x^{n-1} + a_2*x^{n-2} + ... + a_{n-1}*x + a^n,$$

где n – неотрицательное целое (степень полинома), a_0, a_1, a_2, ... a_{n-1}, a_n – коэффициенты полинома, причем коэффициент a_0 не равен нулю. Каждое слагаемое в записи полинома называют членом полинома.

Для многочленов определены операции сложения, вычитания, умножения, деления и сравнения на равенство. Так как отдельное число – это частный случай полинома нулевой степени, то перечисленные операции допустимы в выражениях, где один операнд – число, а второй операнд – полином.

```
// 10_02.cs - Перегрузка операций в классе полиномов.
// Класс полиномов с вещественными коэффициентами.
using System;
class polynom  {
    int size;  // Степень полинома
    double[] coef;  // Коэффициенты  полинома
public polynom(params double[] ar)   { // Конструктор
    size = ar.Length;
    coef = new double[size];
    int i = 0;
    foreach (double dd in ar)
        coef[i++] = dd;
}
public double value(double x)  {  // Значение  полинома
    double sum = coef[0];
    for (int i = 1; i < coef.Length; i++)
        sum = sum * x + coef[i];
    return sum;
}
// Умножение полинома на число:
public static polynom operator *(polynom p, double k)   {
    polynom temp = new polynom();
    temp.size = p.size;
    temp.coef = new double[p.size];
    for (int i = 0; i < p.size; i++)
        temp.coef[i] = p.coef[i] * k;
    return temp;
}
// Умножение числа на полином :
public static polynom operator *(double k, polynom p)
    { return p*k;  }
// Суммирование полиномов:
public static polynom operator +(polynom p1, polynom p2){
    polynom p;
    if (p1.size > p2.size)   {
       p = new polynom(p1.coef);
       for (int i = 0, s = p1.size - p2.size; i < p2.size; i++)
           p.coef[i + s] += p2.coef[i];
    }
    else
    {
       p = new polynom(p2.coef);
       for (int i = 0, s = p2.size - p1.size; i < p1.size; i++)
           p.coef[i + s] += p1.coef[i];
    }
    return p;
}
```

```
public new string ToString()   { // Изображение  полинома
    string [] im = new string[size];
    bool flag = true;
    for (int i = 0; i < size; i++)   {
       if (coef[i] == 0) continue;
       if (i == 0 | flag)  {
          flag = false;
          im[i] = (coef[i] == 1) ? string.Format("x^{0}", size - i - 1) :
             (coef[i] == -1) ? string.Format("-x^{0}", size - i - 1) :
             string.Format("{0}*x^{1}", coef[i], size - i - 1);
          continue;
          }
       if (i == size - 1) {
          im[i] = (coef[i] > 0) ?  string.Format("+{0}", coef[i]) :
             string.Format("{0}", coef[i]);
          continue;
          }
       if (i == size - 2) {
          im[i] = (coef[i] == 1) ? string.Format("+x") :
             (coef[i] == -1) ? string.Format("-x") :
             (coef[i] > 0) ? string.Format("+{0}*x", coef[i]) :
             string.Format("{0}*x", coef[i]);
          continue;
          }
       im[i] = (coef[i] == 1) ?  string.Format("+x^{0}", size - i - 1) :
          (coef[i] == -1) ?  string.Format("-x^{0}", size - i - 1) :
          (coef[i] > 0) ? string.Format("+{0}*x^{1}", coef[i], size - i - 1):
          string.Format("{0}*x^{1}", coef[i], size - i - 1);
    }
    return string.Join("",im);
}
}  // polynom
class Program   {
static void Main()   {
    polynom p1 = new polynom(1, -1, 0, 0, 3);
    polynom p2 = new polynom(2, 0, -1, 2);
    polynom res = p1 + p2;
    Console.WriteLine("Полиномр1: \t" + p1.ToString());
    Console.WriteLine("Полиномр2: \t" + p2.ToString());
    Console.WriteLine("Полиномр1+ p2: \t" + (p1 + p2).ToString());
    Console.WriteLine("Полиномр1*2: \t" + (p1*2).ToString());
    Console.WriteLine("Полином 2*p1: \t" + (2*p1).ToString());
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результат выполнения программы:

```
Полином p1:   x^4-x^3+3
Полином p2:   2*x^3-x+2
Полином p1+p2: x^4+x^3-x+5
Полином p1*2:  2*x^4-2*x^3+6
Полином 2*p1:  2*x^4-2*x^3+6
Для выхода из программы нажмите ENTER.
<ENTER>
```

В классе полиномов только один конструктор с заголовком **public** polynom(**params double**[] ar). Наличие модификатора **params** позволяет задавать в качестве аргументов список коэффициентов полинома a_0, a_1, a_2, ... + a_{n-1}, a_n. Количество аргументов в этом случае должно быть на 1 больше степени полинома. Порядок аргументов должен соответствовать порядку в списке коэффициентов. Если какой-либо из членов полинома отсутствует, то для него нужно задать нулевой коэффициент.

Следует обратить внимание на метод ToString(). Его код получился достаточно сложным, так как в нем учтены разные варианты записи членов полинома, зависящие как от знака коэффициента, так и от степени члена полинома. Например, если изображается старший член полинома, то не требуется знак + при положительном значении коэффициента. Если степень члена равна 1, то не нужно в его изображении использовать возведение в степень, а достаточно записать «x» и т. д.

В классе определены три статических метода для перегрузки операций. Обратите внимание на код метода умножения числа на полином:

```
public static polynom operator *(double k, polynom p)
  { return p*k; }
```

В теле этого метода выполняется неявное обращение к методу с заголовком **public static** polynom **operator** *(polynom p, **double** k), который выполняет умножение полинома на число.

Метод Main(), комментарии в коде и результаты выполнения программы дополняют пояснения.

В программе не иллюстрируются возможности метода **public double** value(**double** x), так как в нем нет ничего нового для темы о перегрузке операций. Метод позволяет вычислить значение полинома по схеме Горнера для заданного аргументом значения переменной x.

Задача 10-03. Определить класс матриц с индексатором для обращения к элементам матрицы и перегруженной операцией умножения матриц.

```
// 10_03.cs – Перегрузка операции умножения матриц
using System;
public class Matrix {//класс матриц с операцией умножения
    int row, col;        // размеры матрицы
    double[,] elem;  // элементы матрицы
public Matrix(int row, int col) {  // конструктор
    elem = new double[row, col];
    this.row = row;
    this.col = col;
}
public int Row { get { return row; } }
public int Col { get { return col; } }
// Индексатор:
public double this[int indexR, int indexC] {
    get { return elem[indexR, indexC];  }
    set { elem[indexR, indexC] = value;  }
}
// Вывод матрицы на экран:
public void print() {
    for (int i = 0; i < row; i++, Console.WriteLine())
        for (int j = 0; j < col; j++)
            Console.Write("{0:F2}  ", this[i, j]);
    }
// Перегрузка операции умножения матриц:
public static Matrix operator * (Matrix m1, Matrix m2) {
    if (m1.Col != m2.Row)  return null;
    Matrix res = new Matrix(m1.Row, m2.Col);
    for (int i = 0; i < m1.Row; i++)
        for (int j = 0; j < m2.Col; j++)
            for (int k=0; k< m2.Row; k++)
            res[i,j] += m1[i,k]*m2[k,j];
    return res;
}
}  // Matrix

class Program  {
static void Main()  {
    Matrix m1 = new Matrix(2, 2);
    m1[0, 0] = 1; m1[0, 1] = 2;
    m1[1, 0] = 3; m1[1, 1] = 4;
    Matrix m2 = new Matrix(2, 2);
    m2[0, 0] = 2; m2[0, 1] = 0;
    m2[1, 0] = 3; m2[1, 1] = -1;
```

```
    Matrix r1 = m1 * m2;
    Console.WriteLine("Произведение матриц m1 * m2:");
    r1.print();
    Matrix r2 = m2 * m1;
    Console.WriteLine("Произведение матриц m2 * m1:");
    r2.print();
    Matrix m3 = new Matrix(2, 3);
    m3[0, 0] = 1; m3[0, 1] = 1; m3[0, 2] = 1;
    m3[1, 0] = 0.5; m3[1, 1] = 0.5; m3[1, 2] = 0.5;
    Matrix r3 = m1 * m3;
    Console.WriteLine("Произведение матриц m1 * m3:");
    r3.print();
    Console.WriteLine("Для выхода из программы нажмите ENTER.");
    Console.ReadLine();
    }
}
```

Результат выполнения программы:

```
Произведение матриц m1 * m2:
8,00  -2,00
18,00  -4,00
Произведение матриц m2 * m1:
2,00  4,00
0,00  2,00
Произведение матриц m1 * m3:
2,00  2,00  2,00
5,00  5,00  5,00
Для выхода из программы нажмите ENTER.
<ENTER>
```

В теле класса кроме полей и свойств определен индексатор:

```
    public double this[int indexR, int indexC] {
        get { return elem[indexR, indexC];  }
        set { elem[indexR, indexC] = value;  }
    }
```

Так как поле **double**[,] elem закрытое, то вне класса только с помощью индексатора можно получить доступ к элементам матрицы, представленной объектом класса. Для обращения к индексатору в теле нестатического метода print() используется выражение **this**[i, j]. Вне класса и в его статическом методе синтаксис применения индексатора аналогичен синтаксису обращений к элементам массива с помощью операции индексирования: m1[i,k].

Известно, что умножение матриц допустимо, когда число столбцов первого операнда (левой матрицы) равно числу строк второго операнда (матрицы справа). Именно это условие в коде метода перегрузки операции умножения проверяет оператор

```
if (m1.Col != m2.Row)  return null;
```

Если проверяемое условие нарушено (отношение в скобках истинно), метод перегрузки операции умножения возвращает значение **null**.

Анализируя результаты выполнения программы, обратите внимание, что операция умножения матриц не обладает переместительным свойством (см. Гантмахер Ф.Г. «Теория матриц»). Для иллюстрации этого свойства в основной программе приведены два разных произведения двух одних и тех же квадратных матриц.

Задача 10-04. Определить класс «правильная дробь» и с помощью перегрузки операций интегрировать объекты класса в систему целых типов языка C#.

Для небольшого сокращения кода реализуем в классе не все операции отношений и не все арифметические операции над дробями. Выполним перегрузку операций сложения, вычитания, изменения знака и операций сравнения дробей (больше и меньше). Кроме того, для интеграции объектов класса в систему целых типов языка C# в классе дробей понадобится операция преобразования значения типа **int** к типу дроби.

```
// 10_04.cs - Перегрузка операций в классе рациональных дробей.
using System;
classFraction {  // класс  рациональных  дробей.
    int num;   // числитель
    int den;   // знаменатель
public Fraction(int n, int d)   { // конструктор
    if (n >= 0 && d > 0) { num = n; den = d; return; }
    if (n >= 0 && d < 0) { num = -n; den = -d; return; }
    if (n <= 0 && d > 0) { num = n; den = d; return; }
    if (n <= 0 && d < 0) { num = -n; den = -d; return; }
    Console.WriteLine("Нулевой знаменатель: {0}/{1}", n, d);
    return;
    }
public new string ToString()
    { return string.Format("{0}/{1}", num, den); }
// унарный  минус:
```

```
static public Fraction operator -(Fraction f)
    { return new Fraction(-f.num, f.den); }
// Сложение дробей:
static public Fraction operator +(Fraction f1, Fraction f2)  {
       int n = f1.num * f2.den + f1.den * f2.num;
       int d = f1.den * f2.den;
       return new Fraction(n, d);
    }
// Вычитание дробей:
static public Fraction operator -(Fraction f1, Fraction f2)
     { return f1+(-f2); }
 //Преобразование целого к дроби:
static public implicit operator Fraction(int x)
     { return new Fraction(x, 1); }
// Сравнение дробей:
static public bool operator < (Fraction f1, Fraction f2)    {
     if (f1.num * f2.den < f1.den * f2.num) return true;
       else return false;
     }
static public bool operator >(Fraction f1, Fraction f2)   {
     if (f1.num * f2.den > f1.den * f2.num) return true;
        else return false;
     }
}  // Fraction
class Program  {
static void Main()   {
     Fraction a = new Fraction(1, 4);
     Console.WriteLine("a = " + a.ToString());
     Fraction b = new Fraction(3, 5);
     Console.WriteLine("b = " + b.ToString());
     Fraction d = a + b;
     Console.WriteLine("a + b = " + d.ToString());
     Console.WriteLine("b - a = " + (b-a).ToString());
     Console.WriteLine("a - 1 = " + (a-1).ToString());
     Console.WriteLine("{0} > {1} == {2}",
                    a.ToString(), b.ToString(), a > b);
     Console.WriteLine("{0} < {1} == {2}",
                    d.ToString(), 1, d < 1);
     Console.WriteLine("Для выхода из программы нажмите ENTER.");
     Console.ReadLine();
     }
}
```

Результат выполнения программы:

```
a = 1/4
b = 3/5
a + b = 17/20
```

```
d = 17/20
b - a = 7/20
a - 1 = -3/4
1/4 > 3/5 == False
17/20 < 1 == True
Для выхода из программы нажмите ENTER
<ENTER>
```

Так как по определению рациональной дроби её знаменатель всегда положительный, а знак значения дроби определяется знаком её числителя, то в конструкторе анализируются знаки параметров, представляющих, соответственно, числитель (**int** n) и знаменатель (**int** d). Реальные значения полей num (числитель) и den (знаменатель) выбираются всегда так, что den>0. В коде класса Fraction ещё нужно обратить внимание на то, что в теле метода вычитания дробей применяются перегруженные операции изменения знака и сложения дробей.

Особую роль играет метод перегрузки неявного преобразования типов:

```
static public implicit operator Fraction(int x)
{ return new Fraction(x, 1); }
```

Обращения к этому методу автоматически выполняются в тех случаях, когда в выражениях с двумя операндами один из операндов имеет тип **int**, а второй имеет тип класса Fraction. Примерами таких выражений в методе Main() служат (a–1) и (d<1).

Задача 10-05. Вектор многомерного пространства над полем вещественных чисел всегда можно представить в языке программирования в виде одномерного массива. Однако более эффективно для дальнейшего применения создать класс многомерных векторов и включить в этот класс такие средства автоматизации кодирования, как перегрузка операций, выполняемых над векторами.

Следуя указанному замыслу, создадим класс Vector и включим в него набор статических методов для перегрузки: операции сложения векторов (+); операции вычитания векторов (-); операция поэлементного (покоординатного) умножения двух векторов (*); операции умножения вещественного числа на век-

тор (*); операции умножения вектора на вещественное число (*) и, наконец, несколько необычную операцию для получения суммы значений элементов (координат) вектора (унарная операция +). Полем объекта класса Vector будет ссылка cood на одномерный массив вещественных координат вектора. У объекта класса Vector будет свойство Length, позволяющее получить количество его координат, т.е. размерность того пространства, в котором определён вектор (объект класса).

Для удобного отображения вектора введём метод ToString(), позволяющий представить значения координат вектора в символьном виде. Чтобы сохранить возможность обращаться к компонентам вектора, как к элементам одномерного массива, определим в классе Vector индексатор. В классе объявим три конструктора: с параметром **int** n, формирующий вектор из n нулевых координат; с параметром **params double**[ ]z, создающий вектор, количество и значения координат которого определены аргументами; с параметром типа Vector – это конструктор копирования, позволяющий создать копию уже существующего объекта класса Vector.

Код класса Vector может быть таким:

```
// Vector.cs – класс многомерных векторов
using System;
namespace Library  {
public class Vector  {
   public double[] coord;  // координаты  вектора
public Vector(int n) { // конструктор
         coord = new double[n];
      }
public Vector(params double[] z) { // конструктор
   coord = new double[z.Length];
   for (int i = 0; i < z.Length; i++)
      coord[i] = z[i];
   }
public Vector(Vector z)  { // конструктор копирования
   coord = new double[z.coord.Length];
   Array.Copy(z.coord, coord, z.coord.Length);
   }
// Операция  поэлементного  сложения  векторов (+):
public static Vector operator +(Vector x, Vector y)  {
   if (x.coord.Length != y.coord.Length)
   throw new Exception("Несоответствие размерностей!");
   int n = x.coord.Length;
   Vector res = new Vector(n);
```

```
    for (int i = 0; i < n; i++)
        res.coord[i] = x.coord[i] + y.coord[i];
    return res;
}
// Операция поэлементного вычитания векторов (-):
public static Vector operator -(Vector x, Vector y)  {
    if (x.coord.Length != y.coord.Length)
    throw new Exception("Несоответствие размерностей!");
    int n = x.coord.Length;
    Vector res = new Vector(n);
    for (int i = 0; i < n; i++)
        res.coord[i] = x.coord[i] - y.coord[i];
    return res;
}
// Операция умножения числа на вектор (*):
public static Vector operator *(double x, Vector y)  {
    int n = y.coord.Length;
    Vector res = new Vector(n);
    for (int i = 0; i < n; i++)
        res.coord[i] = x * y.coord[i];
    return res;
}
// Операция умножения вектора на число (*):
public static Vector operator *(Vector x, double y) {
    int n = x.coord.Length;
    Vector res = new Vector(n);
    for (int i = 0; i < n; i++)
        res.coord[i] = x.coord[i] * y;
    return res;
}
// Операция поэлементного умножения векторов (*):
public static Vector operator *(Vector x, Vector y) {
    if (x.coord.Length != y.coord.Length)
    throw new Exception("Несоответствие размерностей!");
    int n = x.coord.Length;
    Vector res = new Vector(n);
    for (int i = 0; i < n; i++)
        res.coord[i] = x.coord[i] * y.coord[i];
    return res;
}
// Операция суммирования элементов вектора (+):
public static double operator +(Vector x) {
    int n = x.coord.Length;
    double temp = 0;
    for (int i = 0; i < n; i++)
        temp += x.coord[i];
    return temp;
}
```

```
// индексатор
public double this[int k] {
set {
   if (k < 0 || k > coord.Length)
      throw new IndexOutOfRangeException();
   coord[k] = value;
   }
get {
   if (k < 0 || k > coord.Length)
      throw new IndexOutOfRangeException();
   return coord[k];
   }
}
public int Length  // Свойство
   { get { return coord.Length; } }
public new string ToString()  {
   string res = "";
   for (int k = 0; k < Length; k++)
       res += coord[k] + " ";
   return res;
   }
}  // class Vector
}  // namespace Library
```

Обратите внимание, что в классе Vector не определён явно конструктор умолчания (так как в классе объявлены явно другие конструкторы, то конструктор умолчания автоматически не создаётся), но вместо него используется конструктор с параметром, имеющим модификатор **params**.

Поместим этот код в качестве отдельного файла Vector.cs в проект Library из решения Library, т.е. в библиотеку классов, где уже размещён класс Methods, созданный при решении задач из Темы_08.

Напомним порядок действий по добавлению в проект нового файла с кодом класса. Открываем в Visual Studio решение Library; выделяем в окне Solution Explorer правой кнопкой проект Library; выбираем пункт Add; затем New Item. В окне выбираем слева Visual C#, в центре – Class. В поле Name заменим имя Class1.cs на Vector.cs. Система предложит прототип кода класса, который нужно заменить приведённым кодом класса Vector. Для получения новой версии кода библиотеки классов (теперь уже с классом Vector) выполните компиляцию библиотеки. Для этого используйте главное меню (Build -> Build Library).

Библиотечный класс векторов с именем Vector будет полезен для решения задач в одной из следующих тем книги, а в данный

момент нужно выполнить несколько операций над его объектами, чтобы удостовериться в корректности объявления класса.

Создадим в решении Тема_10 новый проект Program_5 консольного приложения и разместим в нём следующий код:

```
// 10_05.cs – операции над многомерными векторами
using System;
using Library;
class Program {
static void Main( ) {
    Vector x = new Vector(1, 0, 0);
    Vector y = new Vector(0, 1, 0);
    Vector z = new Vector(0, 0, 1);
    Vector res = x + y + z;
    Console.WriteLine("res = "+res.ToString());
    Console.WriteLine("res*10 = " + (res * 10).ToString());
    Console.WriteLine("+res = "+ +res);
    Console.Write("Для выхода из программы нажмите Enter.");
    Console.ReadLine();
}
}
```

Результаты выполнения программы:

```
res = 1 1 1
res*10 = 10 10 10
+res = 3
Для выхода из программы нажмите Enter.
<ENTER>
```

В методе Main() проекта Program_5 объявлены три переменных x, y, z типа Vector, представляющих единичные векторы трехмерного пространства с декартовыми координатами. Третий вектор (представляемый ссылкой res) – результат суммирования векторов x, y, z, имеет координаты (1,1,1). Этот вектор (объект класса Vector) создан статическим методом Vector.operator+(), обращение к которому дважды неявно выполняет компилятор, обрабатывая выражение x+y+z. Для вывода значения вектора, представляемого ссылкой res, в методе WriteLine() используется обращение к нестатическому методу ToString(), принадлежащему классу Vector. В следующих двух обращениях к методу WriteLine() иллюстрируются применения к вектору операции умножения на число (res*10) и получения суммы координат вектора (выражение +res). Результаты выполнения программы дополняют пояснения. Отметим, что программа иллюстрирует не все возможности класса Vector, но это сейчас и не обязательно, т.к. этот класс будет ещё использоваться в следующих темах.

ТЕМА 11

НАСЛЕДОВАНИЕ КЛАССОВ И ИНТЕРФЕЙСЫ

Достаточно часто на практике возникает необходимость формировать и обрабатывать коллекции разнотипных элементов. Например, на складе может потребоваться единый список товаров, каждый из которых должен иметь свои собственные характеристики. Программа, обрабатывающая такой список, должна быть в некотором смысле независимой от конкретных товаров и в то же время должна отличать один товар от других. Язык C# позволяет решать такие задачи с помощью наследования классов или реализации интерфейсов. В первой из задач темы используем наследование классов.

Задача 11-1. Пусть в коллекции в случайном порядке размещены объекты (ссылки на объекты) классов, представляющих геометрические фигуры «круг» и «квадрат» на плоскости. Требуется упорядочить элементы коллекции по возрастанию площадей фигур, а затем выполнить сортировку элементов той же коллекции по убыванию периметров фигур.

Прежде чем приступить к программированию, необходимо принять несколько «архитектурных» решений. Для представления коллекции будем использовать массив ссылок на объекты класса «точка плоскости», а классы «круг» и «квадрат» создадим как производные от класса «точка на плоскости». Объявления классов разместим в отдельном файле – в библиотеке классов с именем Figures.

Итак, в библиотеке разместим классы: Point – «точка на плоскости»; Circle – «круг с центром в точке»; Square – «квадрат с центром в точке». Класс Point будет базовым, а классы Circle и Square определим как производные от Point. Напомним, что производный класс наследует код базового и обычно расширяет его функциональность.

Члены класса Point: поля – координаты точки – заданы автореализуемыми свойствами; виртуальный метод display() для вывода характеристик объекта «точка»; виртуальное свойство Area для получения площади фигуры (объекта). Конструктор с умалчиваемыми значениями параметров позволит использовать его как конструктор умолчания и как конструктор общего вида.

Члены класса Circle: поле – «радиус круга», заданное автореализуемым свойством Rad; виртуальный метод display() для вывода характеристик фигуры (объекта круг); виртуальное свойство Area для получения площади круга; свойство Len для получения длины окружности (границы круга). Конструктор общего вида с тремя параметрами, определяющими координаты центра и радиус круга.

Члены класса Square: поле – «сторона квадрата», заданное автореализуемым свойством Side; виртуальный метод display() для вывода характеристик квадрата; виртуальное свойство Area для получения площади квадрата; свойство Len для получения периметра квадрата. Конструктор общего вида с тремя параметрами – координаты центра и сторона квадрата.

Код классов из библиотеки классов Figures.

```
// Библиотека  классов Figures
using System;
namespace Figures  {
// Класс “точка на плоскости”:
public class Point  {
    public double X { get; set; }
    public double Y { get; set; }
public Point (double x = 0, double y = 0)
    { X = x; Y = y; }
virtual // – модификатор виртуального члена
    public void display() // Вывести характеристики
        { Console.WriteLine(“Point: X={0}, Y={1}; “, X, Y); }
virtual // – модификатор виртуального члена
    public double Area  // Площадь фигуры (свойство)
        { get { return 0; } }
}  // Point

// Класс “круг на плоскости”:
public class Circle : Point {
public Circle(int xi, int yi, double r) // конструктор
    : base(xi, yi) // Вызов базового конструктора
    { Rad = r; }
```

```
// Автореализуемое свойство для радиуса круга:
public double Rad { get ; set ; }
// Свойство для получения значения длины окружности:
public double Len { get { return 2 * Rad * Math.PI; } }
override // – модификатор переопределения виртуального члена
    public double Area
    { get { return Rad * Rad * Math.PI; } }
override // – модификатор переопределения виртуального члена
    public void display()  { // Вывести характеристики
          Console.WriteLine("Circle: X={0}, Y={1}; " +
          "Radius={2:f2}, Length={3,6:f2}",
          X, Y, Rad, Len);
          }
}  // Circle

// Класс "квадрат на плоскости":
public class Square : Point  {
public Square(int xi, int yi, double s) // конструктор
     : base(xi, yi) // Вызов базового конструктора
    { Side = s; }
// Автореализуемое свойство для стороны квадрата:
public double Side { get; set; }
// Свойство для получения значения периметра квадрата:
public double Len { get { return 4 * Side; } }
override // – модификатор переопределения виртуального члена
    public double Area { get { return Side * Side; } }
override // – модификатор переопределения виртуального члена
    public void display() { // Вывести характеристики
      Console.WriteLine("Square: X={0}, Y={1}; " +
            "Side={2:f2}, Perimeter={3,6:f2}",
            X, Y, Side, Len);
     }
    }  // Square
} // namespace Figures
```

Поместим библиотеку классов в виде отдельного проекта с именем Figures в решение Тема_11. Соотношение классов библиотеки удобно представить с помощью диаграммы классов (см. рис. 11.1), где линия со стрелкой указывает на отношение наследования классов. (Стрелка обращена к базовому классу.)

В том же решении Тема_11 создайте в виде проекта Program_1 консольное приложение. Включите в число ссылок (References) проекта Program_1 ссылку на библиотеку Figures, а в заголовок

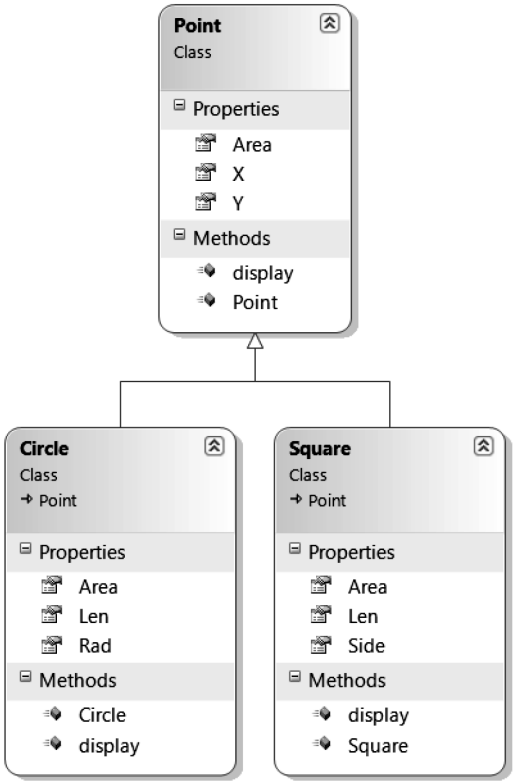


Рис. 11.1. Диаграмма классов библиотеки для задачи 11-01

текста программы поместите оператор **using** Figures; В классе Program приложения определите статический метод figArray( ) для создания массива ссылок типа Point на случайно формируемые объекты классов Circle и Square. Параметр метода figArray( ) определяет размер массива ссылок и количество создаваемых этим методом объектов производных классов. В основной программе создайте массив из выбранного вами числа элементов. Подсчитайте в массиве количества объектов каждого из классов, и выведите сведения об их характеристиках. Используя метод Sort() библиотечного класса **Array**, упорядочите массив по возрастанию площадей фигур, представляемых объектами массива, затем выполните сортировку массива по возрастанию периметров фигур.

Программа, решающая задачу:

```
// 11_01.cs – обработка массивов с разнотипными элементами
using System;
using Figures;
class Program  {
// Метод для создания "случайного" массива:
static Point[] figArray(int n)  {
    Point[] temp = new Point[n];
    Random gen = new Random();
    for (int i = 0; i < n; i++)
        if (gen.Next(0, 2) == 0)
          temp[i] = new Circle(gen.Next(1, 10),
              gen.Next(1, 9), 10 * gen.NextDouble());
        else
          temp[i] = new Square(gen.Next(1, 10),
              gen.Next(1, 9), 10 * gen.NextDouble());
    return temp;
}  // figArray

static void Main()  {
    Point[] arFif = figArray(5);
    int numC = 0, numS = 0;  // Количество объектов
    Console.WriteLine("Исходный массив:");
     foreach (Point p in arFif)  {
        Console.Write("Area={0:f2}\t", p.Area);
        p.display();
        if (p is Circle) numC++;
        if (p is Square) numS++;
        }
    Console.WriteLine("Число объектов Circle: " + numC);
    Console.WriteLine("Число объектов Square: " + numS);
    Array.Sort(arFif, compArea);
    Console.WriteLine("Массив упорядочен по площадям фигур:");
    foreach (Point p in arFif)  {
        Console.Write("Area={0:f2}\t", p.Area);
        p.display();
        }
    Array.Sort(arFif, compPerimeter);
    Console.WriteLine("Упорядочен по убыванию периметров:");
    foreach (Point p in arFif)
        p.display();
    Console.Write("Для выхода из программы нажмите Enter.");
    Console.ReadLine();
}  // Main
```

```
// Метод сравнения площадей:
static int compArea(Point x, Point y) {
    if (x.Area > y.Area) return 1;
    if (x.Area == y.Area) return 0;
    return -1;
} // compArea
 // Метод сравнения периметров:
static int compPerimeter(Point x, Point y) {
    double lx = (x is Circle) ?
        ((Circle)x).Len : ((Square)x).Len;
    double ly = (y is Circle) ?
        ((Circle)y).Len : ((Square)y).Len;
    if (lx < ly) return 1;
    if (lx == ly) return 0;
    return -1;
} // compPeremeter
} // Program
```

В методе Main() после создания массива с элементами типа Point, эти элементы адресуют объекты классов Circle и Square в случайном порядке. Переменная цикла **foreach** (ссылка Point p) последовательно ссылается на эти объекты. Так как метод display() виртуальный и переопределён в производных классах, то выражение p.display() обеспечивает обращение к методу display() того класса, объект которого в данный момент адресован ссылкой p. Для подсчёта объектов класса Circle и Square необходимо определять, какой именно объект адресует в конкретный момент переменная p. Для этого используются в условных операторах выражения с операцией **is**.

Для сортировки массива ссылок Point[] temp на объекты разных классов с помощью метода Array.Sort в класс Program введены два статических метода. Оба метода имеют параметры типа Point. В методе compArea() сравниваются значения виртуального свойства Area, присутствующего в базовом классе Point и переопределяемого в производных классах. Метод compPerimeter() должен сравнивать значения свойства Len, которого нет в базовом классе. Поэтому в теле метода с помощью операции **is** распознается тип каждого из параметров и соответствующим образом выполняется преобразование типов. Таким образом, в методе определяется значение невиртуального свойства производного класса через ссылку с типом базового класса.

Разместив текст программы в проекте Program_1, добавим к ссылкам (References) проекта ссылку на библиотеку классов Figures. Назначим проект Program_1 запускаемым и выполним компиляцию и решение задачи.

Результаты выполнения программы:

```
Исходный массив:
Area=0,33   Square:  X=4,  Y=7;  Side=0,57,  Perimeter= 2,29
Area=3,80   Square:  X=5,  Y=8;  Side=1,95,  Perimeter= 7,79
Area=54,07  Square:  X=2,  Y=1;  Side=7,35,  Perimeter= 29,41
Area=270,17 Circle:  X=4,  Y=7;  Radius=9,27, Length= 58,27
Area=302,44 Circle:  X=6,  Y=2;  Radius=9,81, Length= 61,65
Число объектов Circle: 2
Число объектов Square: 3
Массив упорядочен по площадям фигур:
Area=0,33   Square:  X=4,  Y=7;  Side=0,57,  Perimeter= 2,29
Area=3,80   Square:  X=5,  Y=8;  Side=1,95,  Perimeter= 7,79
Area=54,07  Square:  X=2,  Y=1;  Side=7,35,  Perimeter= 29,41
Area=270,17 Circle:  X=4,  Y=7;  Radius=9,27, Length= 58,27
Area=302,44 Circle:  X=6,  Y=2;  Radius=9,81, Length= 61,65
Упорядочен по убыванию периметров:
Circle:   X=6,  Y=2;  Radius=9,81,  Length= 61,65
Circle:   X=4,  Y=7;  Radius=9,27,  Length= 58,27
Square:   X=2,  Y=1;  Side=7,35,    Perimeter= 29,41
Square:   X=5,  Y=8;  Side=1,95,    Perimeter= 7,79
Square:   X=4,  Y=7;  Side=0,57,    Perimeter= 2,29
Для выхода из программы нажмите Enter.
<ENTER>
```

Объектно-ориентированная методология рекомендует и предусматривает размещение кода, который является общим (одинаковым) для нескольких классов, в базовом классе. Если при этом нет необходимости в создании объектов базового класса, то базовый класс объявляют абстрактным. Кроме кода, общего для всех производных классов, в абстрактный класс в виде прототипов помещают объявления виртуальных членов, функциональность которых появляется только в производных классах.

В рассмотренной программе класс Point – «точка на плоскости» служил базовым для классов, представляющих фигуры круг (Circle) и квадрат (Square). В основной программе не было создано ни одного объекта класса Point. Массив с элементами

типа Point[] был инициализирован ссылками на объекты производных классов Circle и Square, затем обрабатывались именно эти объекты. Ни метод display(), ни свойство Area класса Point в основной программе непосредственно не использовались, а обращение к конструктору Point() выполнялось только из конструкторов производных классов. При таких условиях применения класс Point можно было бы определить как абстрактный, т.е. как класс – единственное назначение которого – служить базовым для производных классов. Изменять уже созданный класс Point не будем, а в следующей задаче на другом примере продемонстрируем некоторые особенности и возможности абстрактных классов.

Задача 11-02. В библиотеке классов Figures определите абстрактный класс Dimensions для представления таких фигур на плоскости, для которых известны только габаритные размеры вдоль координатных осей. В классе Dimensions объявите: конструктор общего вида с умалчиваемыми значениями параметров; поля для представления габаритов фигуры; метод для изменения габаритов в заданное число раз; абстрактное свойство для получения площади фигуры. На базе класса Dimensions определите класс Ellipse – «эллипс, оси которого параллельны координатным осям» и класс Triangle – «треугольник, основание которого параллельно оси абсцисс». В каждый производный класс введите метод ToString(), переопределяющий соответствующий виртуальный метод общего базового класса **Object**.

Новые классы в библиотеке Figures:

```
// Абстрактный класс "габаритные размеры":
abstract public class Dimensions {
    protected double dimX, dimY;
protected Dimensions(double x = 0, double y =0 ) // Конструктор
    { dimX = x; dimY = y; }
public void change(double k) // Изменение  размеров в k раз
    { dimX *= k; dimY *= k; }
abstract public double Area { get; }  // Площадь  фигуры
}  // Dimensions
```

```
// Класс “эллипс”:
public class Ellipse : Dimensions {
public Ellipse(double x, double y) // Конструктор
        : base(x, y) { }
override public double Area  // Площадь  фигуры
    { get { return Math.PI * dimX * dimY / 4; } }
public override string ToString()  {
    string form = “Эллипс: “ +
    “dx={0:f2},\tdy={1:f2},\tArea={2:f2}”;
        return string.Format(form, dimX, dimY, Area);
}
}  // Ellipse

// Производный  класс “треугольник”:
public class Triangle : Dimensions  {
public Triangle(double x, double y) // Конструктор
      : base(x, y) { }
override public double Area // Площадь  фигуры
    { get { return dimX * dimY / 2; } }
public override string ToString()  {
    string form = “Треугольник:” +
        “ dx={0:f2},\tdy={1:f2},\tArea={2:f2}”;
    return string.Format(form, dimX, dimY, Area);
}
}  // Triangle
```

В коде стоит обратить внимание на то, что поля базового класса Dimensions и его конструктор снабжены модификатором **protected**. Свойство Area снабжено модификатором **abstract** и поэтому по умолчанию является виртуальным. Производные классы наследуют члены базового класса и заново определяют свойство Area. Кроме того, в производных классах переопределен метод ToString(). Соотношения между классами иллюстрирует диаграмма на рис 11.2.

В следующей программе, реализованной как проект Program_2 решения с именем Тема_11, иллюстрируются особенности наследования абстрактного класса.

Условие задачи тривиальное. Нужно создать два объекта производных классов, применить к ним методы тех же классов. Затем создать массив ссылок базового класса, «адресовать» ими объекты производных классов и вывести с помощью оператора **foreach** строковые представления элементов массива.

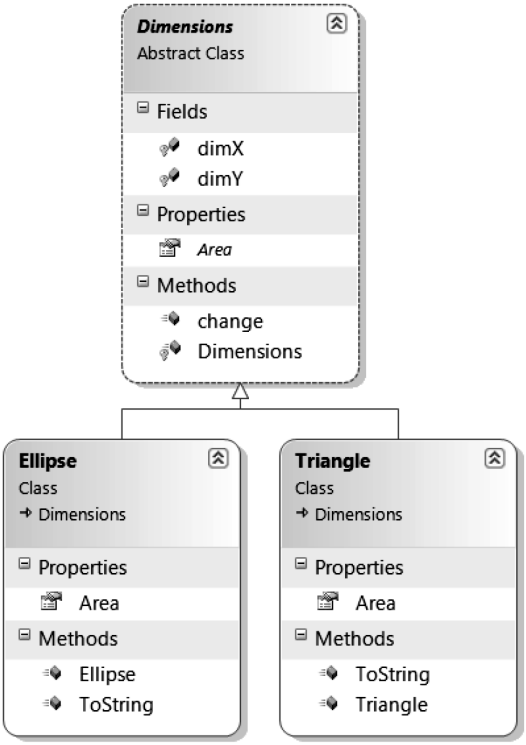


Рис. 11.2. Классы в библиотеке Figures для задачи 11-02

```
// 11_02.cs –классы – наследники абстрактного класса Dimensions
using System;
using Figures;
class Program  {
static void Main()  {
    Ellipse e = new Ellipse(3, 8);
    Console.WriteLine(e.ToString());
    e.change(10);
    Console.WriteLine(e.ToString());
    Triangle t = new Triangle(5, 4);
    Console.WriteLine(t.ToString());
    Dimensions[] fig = new Dimensions[4];
    fig[0] = e;
    fig[1] = t;
    fig[2] = new Ellipse(4, 6);
```

```
    fig[3] = new Triangle(2, 8);
    Console.WriteLine("Массив объектов:");
    foreach (Dimensions d in fig)
        Console.WriteLine(d.ToString());
    Console.Write("Для выхода из программы нажмите Enter.");
    Console.ReadLine();
}  // Main
}  // Program
```

Результаты выполнения программы:

```
Эллипс: dx=3,00,        dy=8,00,        Area=18,85
Эллипс: dx=30,00,       dy=80,00,       Area=1884,96
Треугольник: dx=5,00,   dy=4,00,        Area=10,00
Массив объектов:
Эллипс: dx=30,00,       dy=80,00,       Area=1884,96
Треугольник: dx=5,00,   dy=4,00,        Area=10,00
Эллипс: dx=4,00,        dy=6,00,        Area=18,85
Треугольник: dx=2,00,   dy=8,00,        Area=8,00
Для выхода из программы нажмите Enter.
<ENTER>
```

В коде программы стоит обратить внимание на обязательное присутствие оператора **using** Figures; и не забыть добавить в проект ссылку на библиотеку Figures.

Задача 11-03. Определить класс, объект которого представляет совокупность материальных точек на плоскости, принадлежащих кругу с заданными центром и радиусом. В классе декларировать свойство для представления центра тяжести точек совокупности.

Материальная точка – тело, размерами и формой которого можно пренебречь. В теме 9 уже рассмотрен класс для представления материальных точек в трёхмерном пространстве. В этой задаче нужен класс для представления материальных точек на плоскости. Назовём этот класс MassPoint.

Для иллюстрации возможности наследования этот класс определим в библиотеке Figures как производный от класса Point. Определим множество материальных точек на плоскости, как совокупность точек, принадлежащих кругу с заданными центром и радиусом. В библиотеке Figures объявим класс SetOfMassPoint, представляющий такое множество. Названные классы разместить в библиотеке Figures.

```
// Класс материальная точка – наследник класса Point:
class MassPoint: Point  {
public double mass; //масса точки
// Конструктор:
public MassPoint(double x, double y, double mass)
    base (x,y);
    { this.mass = mass;  }
public double Mass { get {return mass;} }  // масса точки
public new string ToString()   {
    string format = "x={0,2:F3};\ty={1,2:F3};\tmass={2,2:F3}";
    string res = string.Format(format, X, Y, mass);
    return res;
    }
// расстояние между точками:
public double distance(MassPoint pt) {
    double dx = X - pt.X;
    double dy = Y - pt.Y;
    return Math.Sqrt(dx * dx + dy * dy);
}
}  // MassPoint
```

Комментарии в тексте поясняют назначение членов класса. Отметим только основное. Класс MassPoint наследует поля для представления координат точки из базового класса Point. Для обращения к ним в производном классе используются автореализуемые унаследованные свойства X и Y класса Point. Для строкового представления сведений о материальной точке в классе объявлен метод ToString(). В классе MassPoint также объявлен нестатический метод distance(), позволяющий вычислять расстояние между материальными точками (без учета их масс).

```
// Класс множество материальных точек (агрегация классов):
public class SetOfMassPoint {
    MassPoint[] set;        // набор  точек  множества
    double rad;             // радиус множества
// Конструктор выбирает из массива точки:
public SetOfMassPoint(MassPoint[] collection,
            MassPoint ps,       // центр круга
            double rad)         // радиус круга    {
     this.rad = rad;
     MassPoint [] temp = new MassPoint[collection.Length];
     int count = 0;                  // Счетчик  точек  множества
     foreach (MassPoint mp in collection)
          if ((mp.distance(ps) <= rad)) temp[count++] = mp;
     set = new MassPoint[count];
```

```
      Array.Copy(temp, set, count);
      }
// Мощность (число  точек) множества:
public int Power { get { return this.set.Length; } }
// Центр масс множества материальных точек:
public MassPoint MassCentre   {
    get     {
        if (Power == 0) return null;
        double xc = 0, yc = 0, mc = 0;
        foreach (MassPoint mp in set)    {
              mc += mp.Mass;
              xc += mp.Mass * mp.X;
              yc += mp.Mass * mp.Y;
              }
           return new MassPoint(xc / mc, yc / mc, mc);
        }
} // MassCentre
// переопределение  метода
public new string ToString()   {
     string format = "Сведения о множестве: R={0},\tPower={1}";
     string res = string.Format(format, rad, Power);
     return res;
}
}  // SetOfMassPoint
```

В классе SetOfMassPoint два поля: MassPoint[] set и **double** rad, конструктор, свойство Power для получения числа точек во множестве, свойство MassCentre для вычисления центра тяжести точек множества и метод ToString() для строкового представления сведений о множестве точек.

Класс SetOfMassPoint включает объекты класса MassPoint на основе отношения агрегации. Конструктору класса SetOfMassPoint с помощью параметра MassPoint[] collection передается массив ссылок на уже существующие объекты класса MassPoint. Из этих объектов конструктор выбирает только те, которые принадлежат кругу с центром в точке MassPoint ps и радиусом, определяемым параметром **double** rad. Для оценки принадлежности точки кругу в конструкторе используется метод distance() из класса MassPoint. Все точки, попадающие в круг, «накапливаются во временном массиве temp», их число подсчитывается и определяет размер массива MassPoint[] set.

Поместив классы MassPoint и SetOfMassPoint в библиотеку классов Figures, перейдем к программе (проект Programs_3),

формирующей массив материальных точек со случайными значениями полей и создающей на основе этого массива объект класса «множество материальных точек». Размер массива пользователь должен ввести с клавиатуры.

Соотношение между классами задачи 11-03 показано на рис. 11.3. Связь со стрелкой между изображениями классов Point и MassPoint указывает, что эти классы и находятся в отношении наследования, а отношение агрегации классов MassPoint и SetOfMassPoint никак не отображено.

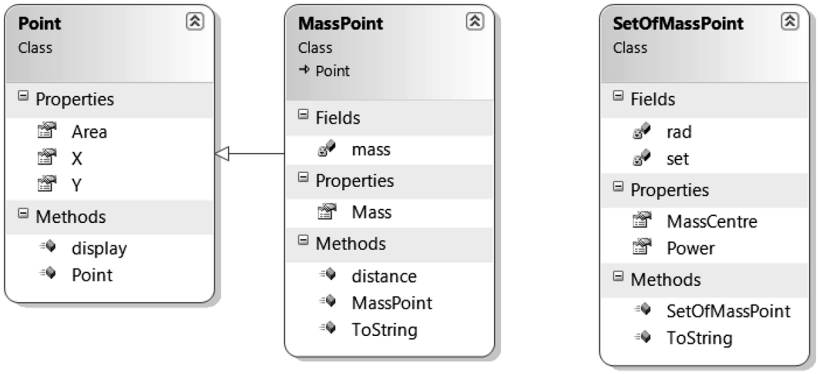


Рис. 11.3. Диаграмма классов для задачи 11-03

В объект класса SetOfMassPoint должны входить только точки плоскости, принадлежащие кругу с заданными центром и радиусом. Поэтому в программе после создания массива материальных точек пользователь вводит значение радиуса круга с центром в начале координат, и из массива конструктор выбирает точки множества. Выводятся сведения о множестве.

```
// 11_03.cs - множество материальных точек на плоскости
using System;
using Figures;
class Program {
static void Main()  {
    MassPoint[] elements;
    int N; // количество точек на плоскости
    Random gen = new Random();
    do Console.Write("Введите количество точек на плоскости: ");
    while (!int.TryParse(Console.ReadLine(), out N) || N < 1);
```

```
elements = new MassPoint[N];
for (int i = 0; i < elements.Length; i++) {
    double x = gen.Next(-10, 11), y = gen.Next(-10, 11);
    elements[i] = new MassPoint(x, y, gen.Next(1, 6));
    Console.WriteLine(elements[i].ToString());
    }
SetOfMassPoint real;
do {
    double R = 0;
    do Console.Write("Введите радиус множества: ");
    while (!double.TryParse(Console.ReadLine(), out R) || R < 0);
    real =
        new SetOfMassPoint(elements, new MassPoint(0, 0, 0), R);
    if (real.Power != 0) {
        Console.WriteLine(real.ToString());
        Console.WriteLine("Центр масс: \n"
              + real.MassCentre.ToString());
        }
    else Console.WriteLine("Множество пусто!");
    Console.WriteLine("Для завершения работы нажмите Escape ");
    } while (Console.ReadKey(true).Key != ConsoleKey.Escape);
}
```

В коде программы объявлены ссылка **elements** на массив с элементами типа MassPoint и целочисленная переменная N – количество элементов в массиве. Диалог с пользователем начинается с ввода значения переменной N. Затем определяется массив из N элементов, которым в цикле присваиваются ссылки на объекты класса MassPoint. Аргументы конструктора, формирующего объекты, – случайные величины. Координаты точек выбираются из интервала [-10,10], массы точек – случайные значения из интервала [1,5].

После создания массива с элементами типа MassPoint объявлена ссылка *real* на объект класса SetOfMassPoint, и пользователю в цикле с постусловием предлагается ввести радиус круга (переменная R). Если введённое значение R положительно, конструктор класса SetOfMassPoint создаёт объект, используя в качестве аргументов: массив точек (ссылка elements); центр круга (объект класса MassPoint с нулевыми значениями полей) и радиус круга (переменная R).

Анализируя значение свойства Power (число точек множества), можно оценить его пустоту. Если множество не пусто (real.Power != 0) – выводим сведения о нём и об его элементах, в противном случае выводится сообщение «Множество пусто!».

Результаты выполнения программы:

```
Введите количество точек на плоскости: 5<ENTER>
x=5,000;      y=7,000;      mass=4,000
x=1,000;      y=-6,000;     mass=3,000
x=9,000;      y=-1,000;     mass=5,000
x=-5,000;     y=-4,000;     mass=3,000
x=3,000;      y=-1,000;     mass=5,000
Введите радиус множества: 3<ENTER>
Множество пусто!
Для завершения работы нажмите Escape
<ENTER>
Введите радиус множества: 7<ENTER>
Сведения о множестве: R=7,                Power=3
Центр масс:
x=0,273;      y=-3,182;     mass=11,000
Для завершения работы нажмите Escape
<Esc>
```

В результате разработки предыдущих программ этой темы в библиотеку классов Figures включены классы, диаграмма которых приведена на рис. 11-4.

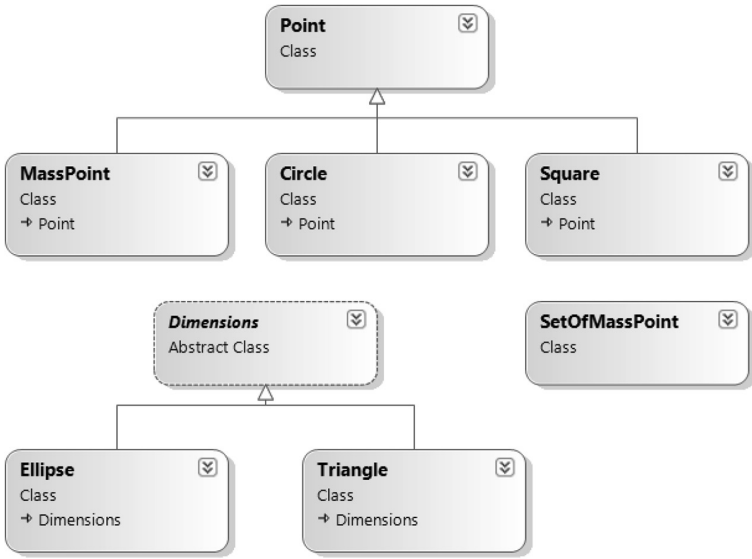


Рис. 11.4. Диаграмма классов библиотеки Figures

Задача 11-04. Разработать интерфейс, определяющий функциональность стека с элементами типа **object**. Реализовать интерфейс стека с помощью двух разных классов. В первом из них для хранения элементов использовать список типа ArrayList, во втором из классов, реализующих интерфейс стека, применить для хранения элементов стека массив с элементами типа **object**. Объявить статический метод для вывода на экран сведений об элементах стека, ссылка на который использована в качестве аргумента.
В основной программе, создав объекты классов, реализовавших интерфейс стеков, продемонстрировать их особенности и возможности статического метода для вывода элементов стека.

Стек – это классическая структура данных, для которой определены следующие операции: поместить элемент на вершину стека, извлечь элемент с (из) вершины стека, получить значение элемента с вершины стека, не удаляя его из стека. Кроме того, обычно в стеке имеется возможность проверить, есть ли элементы в стеке (пуст ли стек). (В библиотеке обобщенных классов присутствует класс обобщенных стеков, имеющих все перечисленные характеристики, но мы его не будем использовать.)

Для разработки стека могут быть использованы разные возможности и средства языка программирования. Было бы удобно, чтобы, несмотря на разные варианты реализации, стек можно было применять, не обращая внимания на различия в реализациях. Такую возможность обеспечивает применение интерфейса, определяющего функциональность целого семейства стеков.

Определим такой интерфейс стеков следующим образом:

```
// Интерфейс стеков:
public interface IStack {
   int Count { get; }              // Число элементов в стеке
   object Pop();                   // Вытолкнуть элемент из стека
   object Peek { get; }            // Посмотреть верхний элемент
   bool Push(object item);         // Втолкнуть элемент в стек
   string[] ToStringArray();       // Массив записей об элементах
}
```

В соответствии с приведенным выше описанием операций над стеком, в интерфейсе IStack включены прототипы методов Pop() – вытолкнуть элемент из стека; Push( ) – втолкнуть эле-

мент в стек. Свойство Peek позволит получить значения элемента на вершине стека (не удаляя его из стека), а свойство Count даст возможность узнать значение количества элементов в стеке. Если значение Count равно нулю, то стек пуст, поэтому отдельного средства для проверки пустоты стека не требуется. Интерфейс IStack не конкретизирует тип элементов стека, так как и параметр метода Push(), и возвращаемое значение метода Pop(), и значение свойства Peek имеют тип **object**. Для подтверждения успешности размещения в стеке очередного элемента метод Push() возвращает логическое значение. Эта возможность может потребоваться, если при реализации стека для хранения элементов будет использоваться фиксированный участок памяти, например, массив. В дополнение к традиционным для стеков средствам в интерфейс стека IStack включен прототип метода ToStringArray() для получения массива строк, содержащих сведения о включенных в стек элементах.

В соответствии с условием создадим две реализации интерфейса стеков. В первой из них для хранения элементов стека применим объект класса ArrayList, принадлежащего пространству имен System.Collections.

Код первого класса, реализующего интерфейс IStack, будет таким:

```
class StackList : IStack { // Стек без ограничения объёма
   ArrayList list; // Список элементов в стеке
// Число элементов в стеке:
   public int Count { get { return list.Count; } }
   public StackList() {   // Конструктор
   list = new ArrayList();
   }
public Object Pop()  { // Вытолкнуть элемент из стека
   Object ob = null;
   if (Count != 0)
      { ob = list[Count-1]; list.RemoveAt(Count-1);}
   return ob;
   }
public Object Peek { // «Посмотреть» верхний элемент
   get {
      if (Count != 0) return list[Count-1];
      else return null;
      }
   }
```

```
// Втолкнуть элемент в стек:
public bool Push(object item) {
   list.Insert(Count, item);
   return true;
   }
public string[] ToStringArray() { // Записи об элементах
   string[] result = new string[list.Count];
   for (int k = 0; k < list.Count; k++)
      result[k] = list[k].ToString();
   return result;
   }
} // class StackList
```

Первое поле класса StackList – ссылка list с типом **ArrayList** – адресует растущий список, каждый элемент которого имеет объявленный тип **object**. Так как тип **object** является базовым для всех типов языка C#, то это позволяет хранить в списке (то есть в стеке) элементы любых классов. Конструктор класса StackList создает пустой список и связывает его со ссылкой list. Свойство Count возвращает счетчик элементов списка. При извлечении («выталкивании») элемента с вершины стека метод Pop() вернет либо ссылку с типом **object** и уменьшит размер списка, либо значение **null**, если в стеке нет элементов.

Похожим образом работает свойство Peek за одним исключением – элемент из списка (из стека) не удаляется. Так как список для хранения элементов стека в данном классе не ограничен, то метод Push(), поместив элемент в стек, всегда возвращает значение **true**.

В методе ToStringArray() создается массив строк, и каждой строке присваивается результат применения метода ToString() к элементу стека. При этом список стека сохраняется, но в массив строк записи помещаются в хронологическом порядке (первый пришел – первый в списке).

Во второй реализации интерфейса IStack используем массив фиксированных размеров с элементами типа **object** для хранения элементов стека.

Код второго класса, реализовавшего интерфейс IStack:

```
class StackArray : ISteck { // Стек ограниченных размеров
   object[] arr;  // элементы стека
   int top;       // число элементов в стеке
public StackArray(int capacity) {  // Конструктор
```

```
    arr = new object [capacity];
    top = 0;
    }
public int Count { get { return top; } }
public Object Pop() { // Вынуть элемент из стека
    Object ob = null;
    if (Count != 0) ob = arr[--top];
    return ob;
}
public Object Peek { // "Посмотреть" верхний элемент
        get {
        if (Count != 0) return arr[top - 1];
        else return null;
        }
    }
public bool Push(object item) { // Втолкнуть элемент в стек
    if (top == arr.Length) return false;
    arr[top++] = item;
    return true;
    }
public string[] ToStringArray()
    { // Массив записей об элементах:
    string[] result = new string[Count];
    for (int k = 0; k < Count; k++)
        result[k] = arr[k].ToString();
    return result;
    }
} // class StackArray
```

Параметр **int** capacity конструктора класса StackArray определяет потенциальную емкость стека – размер массива **object**[] arr для его элементов. Число элементов в стеке (поле **int** top) не может превысить емкость стека. Значение поля top увеличивается при добавлении в стек элемента и уменьшается при выталкивании элемента из стека. Если стек полон, то метод Push() возвращает значение **false** и пополнения стека не происходит. В остальном классы StackList и StackArray очень схожи и функционально неразличимы до тех пор, пока стек класса StackArray не будет переполнен.

Статистический метод PrintStack() для вывода информации о стеке объявим в основном классе программы. Его основная особенность – применение параметра с типом интерфейса IStack. При обращении к этому методу аргументом может быть ссылка

на объект любого класса, реализовавшего интерфейсIStack. Возможности метода и особенности стеков, формируемых как объекты классов StackList и StackArray, показаны в методе Main().

```
// Объекты стеков, реализовавших интерфейс IStack.
class Program
// Метод для вывода информации об элементах в стеке:
static void printStack(IStack ist) {
    foreach (var elem in ist.ToStringArray())
        Console.WriteLine(elem);
    }
static void Main( ) {
    StackList stlist = new StackList();
    stlist.Push("StackList_bottom");
    stlist.Push(new { a = 4, g = 3 });
    stlist.Push(2);
    stlist.Push("3");
    stlist.Push('4');
    stlist.Push("StackList_top");
    printStack(stlist);
    Console.WriteLine("stlist.Pop() = " + stlist.Pop());
    StackArray stArray = new StackArray(10);
    stArray.Push("StackArray_bottom");
    stArray.Push(1);
    stArray.Push(2);
    stArray.Push(new { x = 12, y = 6, z = 3 });
    stArray.Push(new { x = 0, y = 0, z = 0 });
    stArray.Push("StackArray_top");
    printStack(stArray);
     Console.Write("Для выхода из программы нажмите Enter.");
    Console.ReadLine();

} // Main()
}
```

Результаты выполнения программы:

```
StackList_bottom
{ a = 4, g = 3 }
2
3
4
StackList_top
stlist.Pop( ) = StackList_top
StackArray_bottom
```

```
1
2
{ x = 12, y = 6, z = 3 }
{ x = 0, y = 0, z = 0 }
StackArray_top
Для выхода из программы нажмите Enter.
<ENTER>
```

В методе Main() создан «расширяемый» стек как объект класса StackList. Стек представлен ссылкой StList. С помощью метода Push() в стек занесены элементы разных типов. На «дне» стека – строка “StackList_botton”; над ней – объект безымянного класса с полями a = 4, g = 3; далее целое число 2; затем строка “3”; потом символ ‘4’ и наверху стека строка “StackList_Top”. Именно в такой последовательности на экран выводит значения элементов стека метод printStack(stliist). После выполнения метода стек не изменился и обращение stliist.Pop() позволяет «показать» элемент на вершине стека “StackList_Top”.

Далее создан объект класса StackArray с ёмкостью 10 элементов. В этот стек, представляемый ссылкой stArray, занесены шесть элементов разных типов, и их значения выводит на экран метод printStack(stArray).

Результаты выполнения программы дополняют приведённое описание кода.

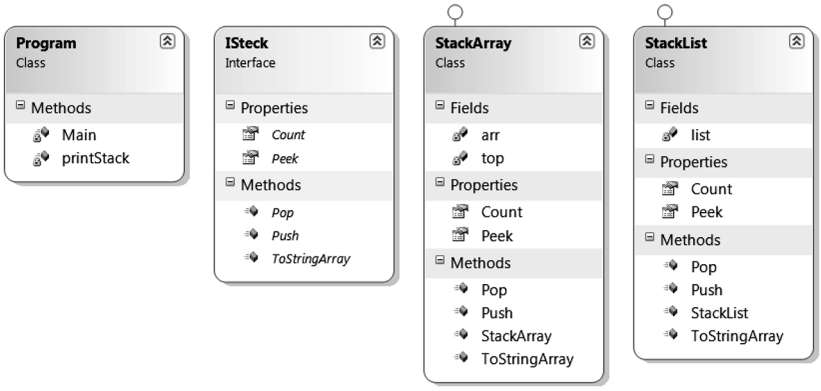


Рис. 11.5. Диаграмма классов программы с реализацией интерфейса стеков

ТЕМА 12

ПРОГРАММЫ С ВИЗУАЛЬНЫМ ИНТЕРФЕЙСОМ ПОЛЬЗОВАТЕЛЯ

Для того, чтобы в интегрированной среде разработки (например, в MS VS) разрабатывать программы с графическим интерфейсом пользователя, нужно:

1) научиться проектировать формы, размещая на них требуемые по смыслу задачи элементы пользовательского интерфейса (элементы управления) и компоненты;

2) понимать особенности механизма управления программами с помощью событий;

3) уметь добавлять в создаваемую программу шаблоны (заготовки) обработчиков событий, возникающих при воздействии пользователя (прямо или косвенно) на элементы управления;

4) уметь дополнять заготовки обработчиков событий программным кодом, реализующим требования, предъявляемые к программе смыслом решаемой задачи.

В программах данной темы поясним особенности первого, третьего и четвёртого из четырёх перечисленных выше пунктов, необходимых для разработки в MS VS программ с графическим интерфейсом. Основное внимание будет уделяться не подробному описанию конкретных элементов Windows Form, а базовым механизмам программирования на языке C# при разработке программ с графическим интерфейсом.

Задача 12-01. Разработать Windows-приложение, на экранной форме которого размещены два элемента пользовательского интерфейса Button (Кнопка) и Label (Метка). При каждом нажатии кнопки выводить в текстовое поле элемента Label значение очередного члена ряда Пелла:

$$p_1 = 1,\ p_2 = 2,\ p_3 = 5,\ ...p_i = p_{i-2} + 2^* p_{i-1}...$$

Для представления членов ряда использовать переменные типа **int**. При возникновении целочисленного переполнения

вывести в диалоговое окно MessageBox сообщение “Переполнение! Ряд начнем с начала!”.

Желаемый вид результатов выполнения программы показан на рис. 12.1.

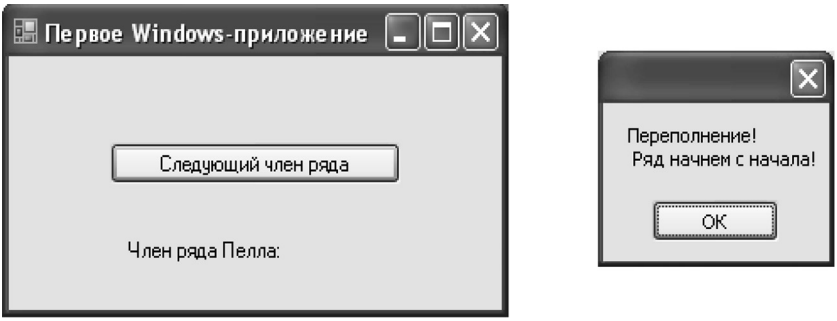


Рис. 12.1. Экранная форма и диалоговое окно работающей программы

Как и при создании консольных приложений, откроем окно выбора нового проекта (см. рис. 1.2), но вместо “Console Application Visual C#” на центральной панели выберем вид приложения “Windows Forms Application Visual C#” (приложение с экранной формой). Введем: имя решения «Тема_12», имя проекта «Program_1», имя каталога «C:\C#_Практикум». После создания решения и проекта страница MS VS 2010 примет вид, показанный на рис. 12.2. Слева – окно Form1.cs [Design] для проектирования пользовательского интерфейса, справа – окно обозревателя решения (Solution Explorer). Если окно проектирования по каким-то причинам не открыто – его можно открыть, активируя мышью пункт Form1.cs в окне обозревателя.

При работе в рускоязычной версии MS VS 2012 (см. рис. 1.9) выбираем «Приложение Windows Forms Visual C#». Введя имена решения и проекта, получим окно проектирования («Конструктор») с заготовкой формы.

В окне проектирования изображена заготовка экранной формы, с минимальной функциональностью (экранная форма с именем Form1, которое показано в ее заголовке). Если выполнить компиляцию и запуск приложения (**Ctrl+F5**), то получим именно такую экранную форму, которая изображена в окне проектирования. Приложению, которое формирует на экране компьютера экранную форму с минимальной функциональ-

ностью, соответствует код, который размещается в нескольких файлах и создан автоматически. Никакого «ручного» программирования для этого не понадобилось.

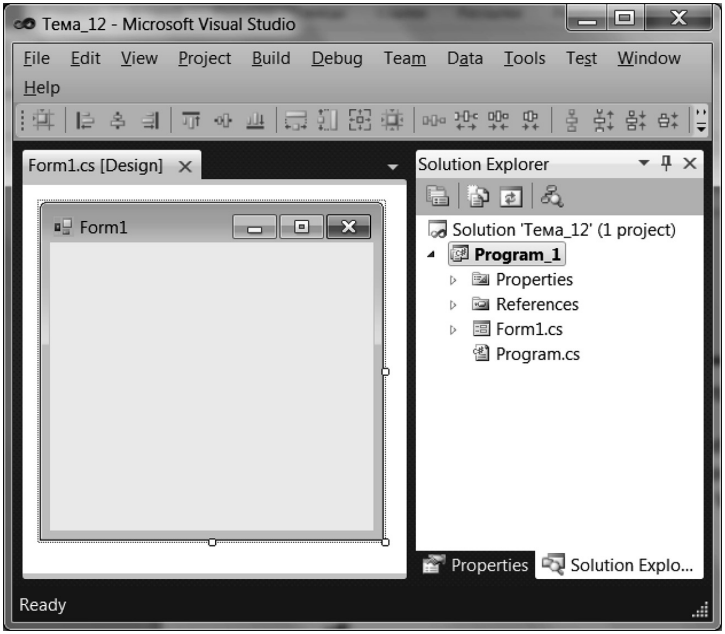


Рис. 12.2. «Заготовка» для приложения с экранной формой

Для дополнения экранной формы элементами управления проще всего использовать панель инструментов (Toolbox), см. рис. 12.3. Если панель инстументов не показана, то ее можно открыть через главное меню:

View –> Toolbox или сочетание клавиш (**Ctrl+W,X**).

При работе в рускоязычной версии MS VS 2012 используйте последовательность:

ВИД –>Другие окна –>Панель элементов или **Ctrl+W,X**.

В панели элементов откройте раздел **«Стандартные элементы управления»**.

Последовательно выбирая из списка на панели инструментов элементы **Button** и **Label**, «перетаскиваем» их мышью на экранную форму. После «перетаскивания» элементы будут помечены над-

писями: «button1» и «label1» (рис. 12.4). Для того, чтобы изменить указанные надписи, необходимо обратиться к спискам свойств элементов Button и Label. Для этого щелкните левой кнопкой мыши по изображению элемента и активируйте в выпадающем меню пункт «Properties». Появится окно свойств данного элемента (для элемента **Button** см. рис. 12.4). Заменим надпись «Button1» в поле **Text** на «Следующий член ряда». Можно было бы проделать те же действия для элемента **Label**, заменив текст «label1» на «Член ряда Пелла: », однако не будем этого делать, чтобы показать как можно изменить свойство элемента из кода программы.

Свойства имеются не только у элементов управления, но и у самой экранной формы. Для нее умалчиваемое значение свойства **Text** – это «Form1», и именно эта строка выводится в ее заголовке (см. рис. 12.2-12.4). Заменим «Form1» на «Первое Windows-приложение».

Уменьшив вертикальный размер изображения экранной формы и растянув по горизонтали изображение элемента **Button**, получим окно, соответствующее условию задачи (рис 12.1), за одним исключением – элемент **Label**, сохранит надпись «label1».

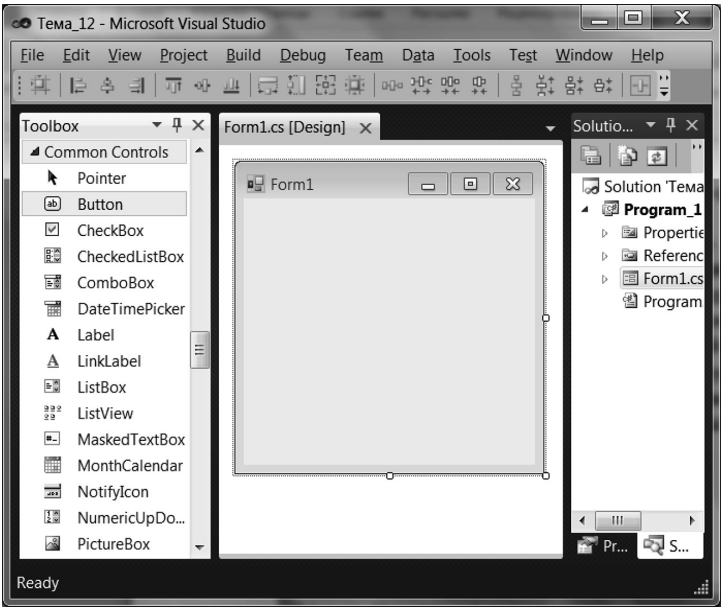


Рис. 12.3. Панель инструментов (элементов и компонентов)

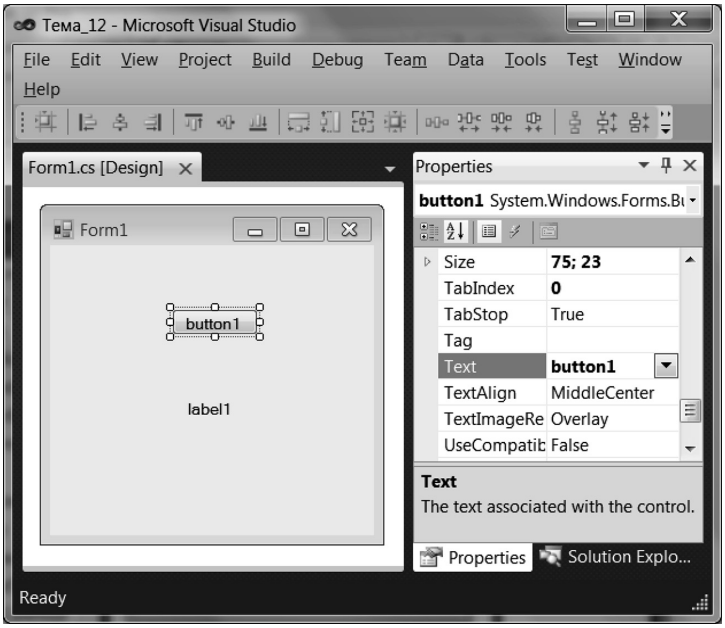


Рис. 12.4. Редактор экранной формы и панель свойств элемента Button

На этом этапе мы показали как выполнять в VS первый шаг цикла проектирования – как размещать на форме элементы графического интерфейса и изменять их свойства (пока только свойство **Text**). Транслируя и выполняя программу, мы получим на экране окошко, почти соответствующее условию задачи. Но программа ещё не решает задачу. У нашего приложения нехватает функциональности. Кнопка **Button** не реагирует на нажатия, метка **Label** всегда отображается без изменений как текст «label1». Но приложение можно транслировать и запустить на выполнение, можно изменять размеры экранной формы, можно перемещать ее по экрану, можно с помощью стандартных для Windows кнопок в заголовке формы раскрыть ее на весь экран, можно свернуть, можно завершить исполнение приложения.

Подготовленное нами приложение может обрабатываться как простейшая Windows-программа только потому, что автоматизированная среда разработки сгенерировала код (в нашем случае на языке C#), соответствующий нашему проекту экран-

ной формы. Такой код размещается в нескольких файлах, из которых только один нужно вручную дополнять и модифицировать. Этот код находится в файле Form1.cs. Чтобы вывести его в окно редактора кода,можно выделить в окне обозревателя решения имя этого файла и нажать **F7**. Вторая возможность – щелкнуть правой кнопкой по изображению экранной формы и в выпадающем меню выбрать пункт **View Code**. Прежде чем дополнять этот код, приведем его в том виде, как он создан средой:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace Program_1 {
   public partial class Form1:Form {
   public Form1( ) {
      InitializeComponent( );
       }
   }
}
```

Последовательно рассмотрим строки кода из файла Form1.cs. Во-первых, для нашей задачи нужны не все операторы **using**. Система автоматизации программирования на всякий случай обеспечивает заготовку кода средствами доступа к большинству наиболее употребительных пространств имен платформы .NET. Именно поэтому в тексте восемь операторов **using**. Для нашей программы достаточно иметь доступ только к пространству **System.Windows.Forms**.

Оператор **namespace Program_1** вводит собственное пространство имен данного приложения.

Далее размещен код части класса Form1. То, что это только часть класса Form1, указывает модификатор **partial** в его заголовке. Те его части, которые обычно не требуют ручного программирования, система автоматически помещает в другие файлы, изменять которые нам не понадобится.

Класс Form1 построен как наследник библиотечного класса Form, и от него получает большую часть своей функциональности. В приведенной заготовке класса всего один метод – конструктор без параметров. В теле этого конструктора только один оператор – обращение к унаследованному классом Form1 методу инициализации компонентов (элементов управления) экранной формы.

Как уже говорилось, сейчас нами построено Windows-приложение, выполнение которого приводит к созданию экранной формы, показанной на рис. 12.5. Экранной форме соответствует объект класса Form1. Формирование кода той части программы, где размещен метод Main() и где создается объект класса Form1, среда MS VS выполняет автоматически без ручного программирования. Однако требуется дополнить автоматически создаваемый код программы операторами, соответствующими целям решаемой программой задачи.

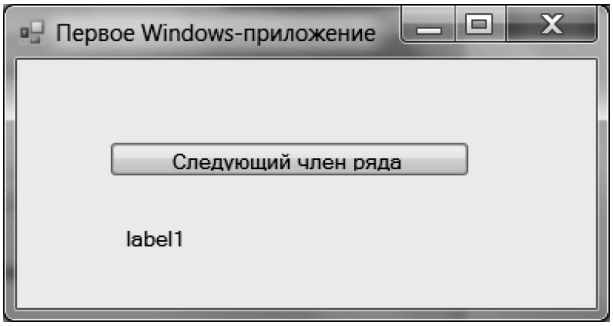


Рис. 12.5. Экранная форма с минимальной функциональностью

Для нашей задачи нужно, во-первых, добавить в программу заготовку для обработки события, соответствующего нажатию на кнопку. В интегрированной среде разработки, какой является MS VS, этот процесс автоматизирован. Если, находясь в дизайнере форм, дважды щелкнуть левой клавишей мыши по изображению элемента **Button**, то в файл автоматически добавится метод – обработчик события «button1_Click» – «нажатие на кнопку button1». То же самое можно сделать через панель свойств элемента **Button**, включив на ней список событий с помощью иконки с изображением молнии (см. рис. 12.6) и активировав пункт **Click**.

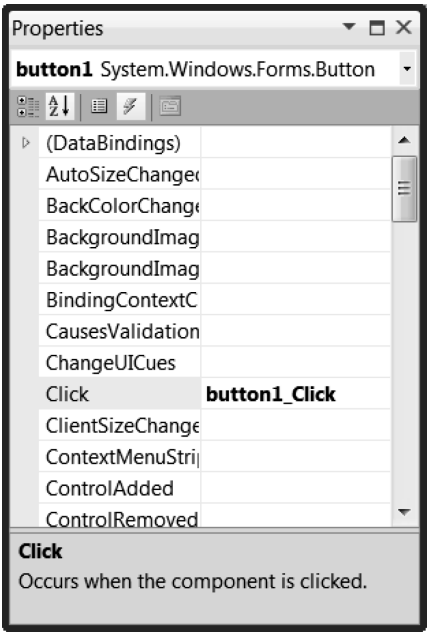


Рис. 12.6. Список событий для элемента button1

После создания заготовки для метода button1_Click() автоматически открывается окно редактора кода, в который выведен текст файла Form1.cs.

Заголовок метода для обработки названного события:

```
private void button1_Click(object sender, System.EventArgs e)
```

Первый параметр определяет источник (происхождение) события. Второй параметр – передает методу сведения о событии (характеристики события). На первых этапах ознакомления с разработкой программ с оконным интерфейсом ни один из параметров метода обработки событий нам не понадобится. Достаточно знать, что при нажатии на клавишу button1 формируется соответствующее событие и автоматически выполняется метод button1_Click(). Для того, чтобы программа решала задачу генерации чисел-элементов ряда Пелла, достаточно дополнить тело метода button1_Click() соответствующими операторами, ввести в объявление класса Form1 дополнительные поля и слегка изменить тело его конструктора.

Текст файла Form1.cs для нашей задачи может быть таким:

```
// 12_01.cs – члены ряда Пелла
using System.Windows.Forms;
namespace Program_1 {
public partial class Form1:Form {
public Form1( ) {
    InitializeComponent( );
    label1.Text = text;
}
// Поля объектов класса Form1:
  int old = 1, last = 0;
  string text = "Член ряда Пелла: ";
// Метод обработки события "Нажатие на клавишу":
private void button1_Click (object sender, System.EventArgs e) {
// Защита от переполнения:
  if (old > int.MaxValue - last - last)  {
// Диалоговое окно, захватывающее фокус:
      MessageBox.Show("Переполнение!" +
          " \n Ряд начнем с начала!");
      last = 0; old = 1;
      }
// Локальная переменная метода:
  int now = old + 2 * last;
  old = last; last = now;
  label1.Text = text + now.ToString();
  }
  }
}
```

В класс введены три поля: old – для первого, а затем предыдущего членов ряда; last – для второго и затем очередного членов ряда; text – для строки с текстом "Член ряда Пелла:".

В теле конструктора (который выполняется при создании объекта класса Form1) добавлен оператор label1.Text = text;. Тем самым после запуска программы текст "label1" на экранной форме будет заменен текстом "Член ряда Пелла:".

В теле метода обработки события button1_Click() два фрагмента кода. Второй начинается с объявления вспомогательной переменной **int** now, которой присваивается значение очередного члена ряда. Затем обновляются значения переменных old и last и изображение значения переменной now (дополненное по-

яснительным текстом) присваивается полю label1.Text. На этом можно было бы завершить получение значений членов ряда Пелла, но эти значения растут бесконечно, и в конце концов очередное превысит **int**.MaxValue. Если не предусмотреть защиту от этой ситуации, то произойдёт переполнение и программа выдаст неверный результат.

Для защиты от переполнения в метод button1_Click() включен условный оператор, в теле которого выполняется обращение к диалоговому окну MessageBox.Show(). При визуализации диалогового окна в него выводится предупреждающее сообщение (см. рис. 12.7). При выходе из диалогового окна переменным last и old присваиваются начальные значения и последующие нажатия на клавишу button1 приведут к повторению ряда с его начала.

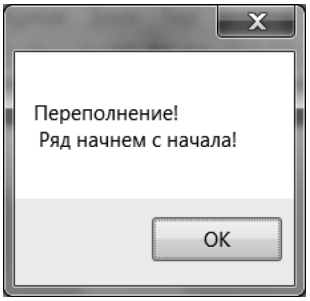


Рис. 12.7. Диалоговое окно при предельном значении элемента ряда Пелла

Обратите внимание на тот факт, что диалоговое окно MessageBox «захватывает» фокус ввода. (Такие окна называют модальными.) До тех пор, пока пользователь не нажмёт клавишу OK в диалоговом окне, программа не продолжит выполнения, а попытки нажать клавишу «Следующий член ряда» на основной форме будут игнорироваться.

Задача 12-02. Разработать Windows-приложение. В поле TextBox вывести в виде списка элементы массива строк. Отредактировав (изменив) список на экране, вывести его в диалоговое окно MessageBox. Обеспечить возможность восстановления начального состояния списка.

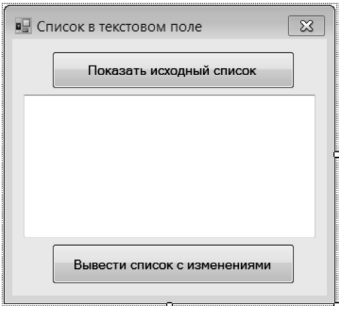


Рис. 12.8. Изображение окна в редакторе форм для задачи 12-02

Настройка свойств элементов:
Form1.Text = Список в текстовом поле
Form1StartPosition = CenterScreen
button1.Text = Показать исходный список
button2.Text = Вывести список с изменениями
textbox1:Multiline = True
textbox1.Anchor = Top, Bottom, Left, Right

```
// 12_02.cs – редактируемый список в текстовом поле
using System.Windows.Forms;
namespace Program_2 {
public partial class Form1:Form {
public Form1( ) {
        InitializeComponent( );
        button2.Visible = false; // скрыть  кнопку 2
        }
    string [] lines = new string[]
        {"Каждый – ","Охотник – ","Желает – ","Знать – ",
            "Где – ","Сидит – ", "Фазан – " };
private void button1_Click(object sender, System.EventArgs e) {
        textBox1.Lines = lines;     // Вывести  строки  массива
        button2.Visible = true;     // показать  кнопку 2
        }
private void button2_Click(object sender, System.EventArgs e) {
        string res = string.Join("\n", textBox1.Lines);
        MessageBox.Show("Результат изменений:\n"+res);
        }
} // Form1
}
```

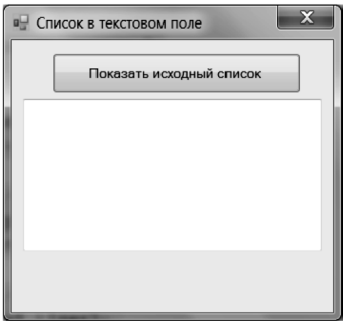


Рис. 12.8. Форма после загрузки

При создании объекта класса Form1 свойству Visible элемента button2 в конструкторе присваивается значение **false** и изображение кнопки button2 на форме не появляется (см. рис. 12.8).

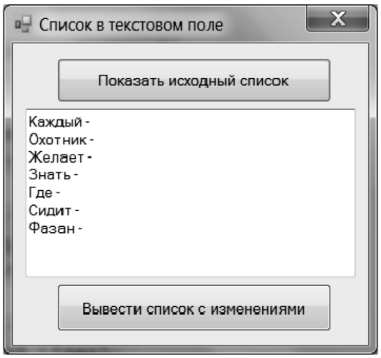


Рис. 12.9. Исходный список в текстовом поле

В теле класса Form1 только одно поле lines – ссылка на инициализированный массив строк. При выполнении обработчика button1_Click() значение этой ссылки присваивается свойству Lines элемента (объекта) textBox1. Это приводит к выводу строк в текстовое поле. Свойство Visible элемента button2 получает значение **true** и кнопка button2 становится видимой на форме (рис. 12.9). Дополнение пользователем текста в окне textBox1 показано на рис. 12.10 слева. Справа – диалоговое окно MessageBox с результатами обработки текста.

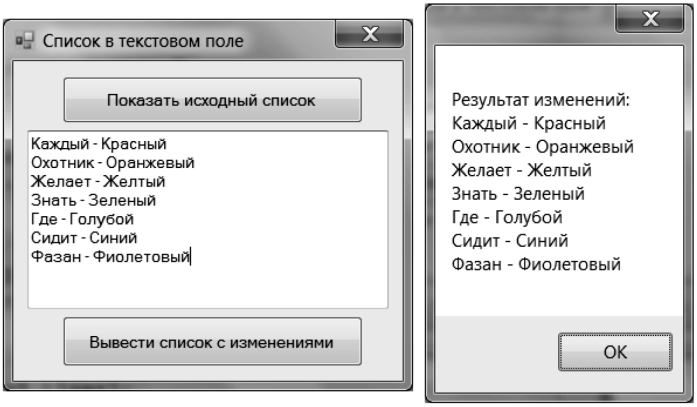


Рис. 12.10. Редактирование текста и результат в диалоговом окне

Задача 12-03. Упорядочить по убыванию цифры натурального числа, вводимого с клавиатуры в текстовое поле.

Эту задачу можно решать по-разному. Воспользуемся тем фактом, что число вводится как символьная строка, и применим методы библиотечных классов **String** и **Array**. Тогда последовательность действий может быть такой. Читаем строку с изображением числа; удаляем из нее (возможно присутствующие) пробелы слева и справа; проверяем отсутствие символов, отличных от цифр; преобразуем строку в символьный массив; упорядочиваем по возрастанию элементы массива; реверсируем элементы массива (меняем на обратный порядок элементов); формируем из массива строку и выводим ее как результат.

Строку с цифрами натурального числа будем вводить в однострочное текстовое поле, формируемое элементом **TextBox**. Для вывода результата сортировки используем элемент **Label**. При ошибках ввода будем выводить соответствующие сообщения в диалоговое окно (**MessageBox**).

Итак, нам понадобится форма (см. рис. 12.11) со следующими элементами: **Label** – для приглашения пользователю; **TextBox** – для ввода числа; **Button** – кнопка для запуска счета; **Label** – для вывода упорядоченных цифр числа.

Настройки свойств элементов:
Form1.Text =Сортировка цифр числа
Form. StartPosition = CenterScreen

Label1.Text = Введите натуральное число:
Label1.Font.Size = 12
Label1.Font.Bold = True

button1.Text = Разместить цифры по убыванию
button1.Font.Size = 12
button1.Font.Bold = True

Label2.Text = Результат:
Label2.Font.Size = 12
Label2.Font.Bold = True

TextBox1.Font.Bold = True
TextBox1.Font.Size = 12

Из набора операторов **using**, которые автоматически включаются в заготовку файла Form1.cs, удалим все кроме **using System.Windows.Forms;** и **using System;**. Второй оператор нужен для применения в коде библиотечных классов **String** и **Array**.

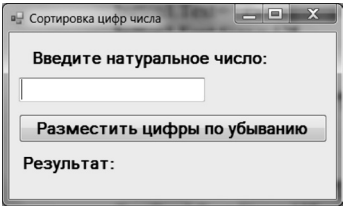


Рис. 12.11. Исходная форма

Текст программы:

```
// 12-03.cs – сортировка цифр вводимого числа
using System;
using System.Windows.Forms;
namespace Program_3 {
public partial class Form1:Form {
public Form1( ) {  // Конструктор
   InitializeComponent( );
   }
  string text;  // Текст элемента label2
  string sample = "0123456789";  // строка цифр
// Обработчик события "Загрузка формы":
```

```
private void Form1_Load(object sender, System.EventArgs e) {
   text = label2.Text;
}
// Метод проверяет присутствие в str2 символов из str1,
// если в str1 только символы из str2 - результат true
bool find(string str1, string str2) {
   int t = -1;       // вспомогательная  переменная
   foreach(char ch in str1) {
      t = str2.IndexOf(ch);
      if(t < 0) return false;
   }
   return true;
}
// Обработчик  события:
private void button1_Click(object sender, System.EventArgs e) {
   string number = textBox1.Text;
   number.Trim();  // убираем  пробелы
   if(number.Length == 0 || !find(number, sample)) {
// Диалоговое окно, захватывающее фокус:
      MessageBox.Show("Это не натуральное число!");
      textBox1.Focus();  // установить фокус ввода
      return;  // покидаем button1_Click
   }
   char [] digits = number.ToCharArray();   // массив из строки
   Array.Sort(digits);              // сортировка по возрастанию
   Array.Reverse(digits);       // реверсирование
   label2.Text =  text + new string(digits);      // строка  из  массива
}  // button1_Click
}
}
```

В классе Form1 два закрытых (**private**) поля типа **string**. В первое поле с именем text обработчик Form1_Load, который выполняется непосредственно при загрузке формы, записывает текст (строку), подготовленный при проектировании формы в свойстве label2.Text. Затем эта строка применяется в обработчике button1_Click для формирования результата выполнения программы. Поле sample используется во вспомогательном методе find() для проверки отсутствия во введённой пользователем строке символов, отличных от цифр. Собственно алгоритм решения задачи реализован в обработчике событий button1_Click(). Локальной переменной **string** number присваивается строка с изображением числа, введённого поль-

зователем в поле элемента textBox1. Логическое выражение number.Length==0||!find(number,sample) обеспечивает проверку корректности введённого изображения числа. Остальное поясняют комментарии в коде программы. Обратим внимание на оператор textBox1.Focus(). Он выполняется только при ошибках во введённом числе и обеспечивает передачу фокуса ввода текстовому полю, где находится ошибочно записанное число.

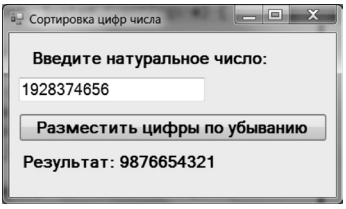


Рис. 12.12. Результат выполнения программы 12-03

Задача 12-04. Поместить в центр формы одну кнопку и в её обработчике события «нажатие на кнопку» изменять размеры формы. В начале, при каждом нажатии на кнопку размеры формы уменьшать, но как только форма достигнет минимальных (заданных при разработке) размеров – увеличивать ее при нажатии на ту же кнопку. Когда форма достигнет максимальных размеров – переключить кнопку на уменьшение и т. д. Начальный вид формы показан на рис. 12.13.

Данная задача показывает, что элементы управления формы и сама форма представляют собой объекты, характеристики которых можно изменять в коде программы, и тем самым изменять визуальное представление окна и его элементов в процессе выполнения программы.

Настройка свойств элементов:
Form1.Text = Изменение размеров формы
Form1.StartPosition = CenterScreen
Form1.MaximumSize = 1000; 700
Form1.MinimumSize = 220; 100
Form1.MaximizeBox = False
button1.Text = Уменьшить форму
button1.Anchor = None
button1.Size = 150; 30

Для использования в программе библиотечных типов **Point** и **Size** включим в заголовок кода оператор:

```
using System.Drawing;
```

Назначение пространств имён ***System*** и ***System.Windows***. Forms мы уже объяснили при разработке программ предыдущих задач.

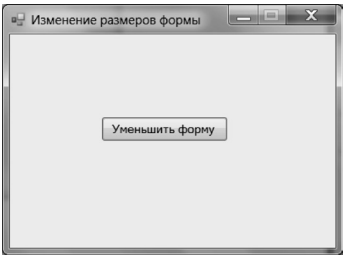


Рис. 12.13. Начальный вид формы

```
// 12_04.cs - программное изменение размеров формы
using System;
using System.Drawing;
using System.Windows.Forms;
namespace Program_4 {
public partial class Form1:Form {
public Form1( ) {
      InitializeComponent( );
      }
private void Form1_Load(object sender, EventArgs e) {
       Size s = this.ClientSize;  // размеры клиентской области
       // Размещение кнопки в клиентской области формы:
       button1.Location = new Point(s.Width/2 - button1.Width/2,
                        s.Height/2 - button1.Height/2);
       }
 bool grow = false;  // Флажок  изменения  размеров
// Направление (true – рост, false - уменьшение формы)
private void button1_Click(object sender, EventArgs e) {
   int w = this.Size.Width;                    // ширина  формы
   int h = this.Size.Height;                   // высота  формы
    if (grow == false)                         // Уменьшаем  форму
    if (w > this.MinimumSize.Width || h > this.MinimumSize.Height)
        { this.Size= new Size(w/3*2 , h/3*2);
        this.CenterToScreen();  return; }
```

```
    else { grow = true; button1.Text="Увеличить форму"; return;}
  else                  // Увеличиваем  форму
    if (w < this.MaximumSize.Width || h < this.MaximumSize.Height)
       { this.Size= new Size(w/2*3 , h/2*3);
         this.CenterToScreen();  return; }
    else
       { grow = false; button1.Text="Уменьшить форму"; return; }
} // button1_Click
}
}
```

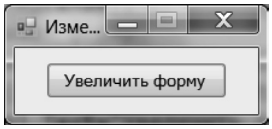


Рис. 12.14. Уменьшенная до минимума форма

При загрузке формы кнопка помещается в центр клиентской области формы. Для этого в обработчике события Form1_Load() определяются размеры клиентской области формы и соответствующим образом вычисляются координаты левой верхней вершины прямоугольного изображения кнопки. Свойство button1.Location – это объект библиотечного типа **Point** («точка на экране»).

Основная функциональность программы в обработчике button1_Click(). При каждом нажатии на кнопку вычисляются текущие размеры формы: w – ширина, h – высота. Если установлен флажок уменьшения формы (grow==**false**), то линейные размеры формы уменьшаются в 2/3 раза. Это уменьшение производится только в тех случаях, когда w и h не меньше размеров, заданных свойствами формы. В противном случае флажок grow получает значение **true**, изменяется текст на изображении кнопки и при последующих нажатиях на кнопку размеры формы увеличиваются в 3/2 раза. Остальное очевидно из кода метода. Отметим только, что в программе (точнее в обработчике событий button1_Click) не только изменяются размеры формы, но форма принудительно переносится в центр экрана. Делается это за счет обращения к методу this.CenterToScreen() после каждого изменения размеров формы.

Задача 12-05. Периметр p правильного n-угольника, описанного около окружности радиуса r, равен 2*n*r*tg(PI/n). Ввести значения n и r, проверить их корректность и вывести значение периметра. Проверяемые условия: n >= 3 и r > 0; отсутствие во входной строке нецифровых данных.

Размещение управляющих элементов на форме выполнить в относительных координатах. Ввести ограничения на минимизацию формы. (Изображения элементов не должны «налезать» друг на друга при уменьшении размеров формы.)

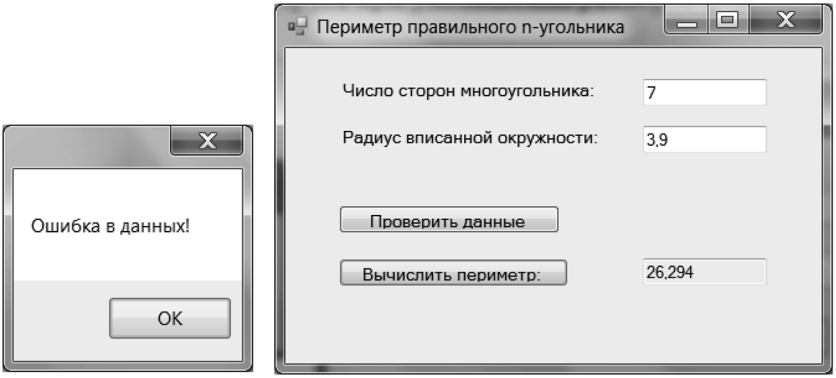


Рис. 12.15. Диалоговое окно и форма к задаче 12-05

Настройка свойств элементов:

Form1.Text = Периметр правильного n-угольника
Form1.StartPosition = CenterScreen
Form1.MinimumSize = 450; 300
Form1.MaximumSize = 600; 400

label1.Text = Число сторон многоугольника:
label2.Text = Радиус вписанной окружности:

button1.Text = Проверить данные
button2.Text = Вычислить периметр

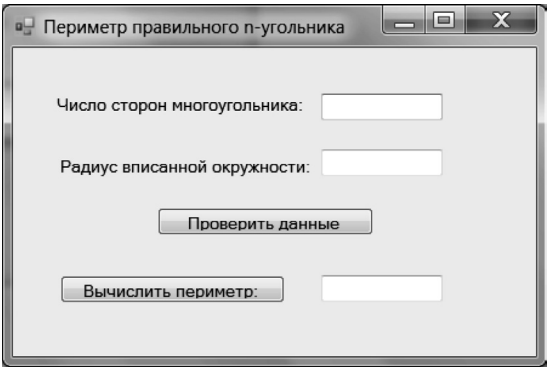


Рис. 12.16. Спроектированная форма (вид в редакторе форм)

Обработчики событий, которые нужно добавить в код формы:

Form1_Load *– загрузка формы;*
Form1_Paint *– перерисовать форму;*
Form1_Resize *– изменение размеров формы;*
button1_Click *– нажатие на клавишу 1;*
button2_Click *– нажатие на клавишу 2;*

Программа, решающая задачу:

```
// 12_05 – периметр правильного n-угольника
using System;          // Для класса Math
using System.Drawing; // Point, Size
using System.Windows.Forms;

namespace Program_5 {
public partial class Form1:Form {
public Form1( ) { // Конструктор  формы
    InitializeComponent( );
}
// Обработчик  события “Загрузка формы”:
private void Form1_Load(object sender, EventArgs e) {
// Запретить внешние изменения поля:
    textBox3.ReadOnly = true;
    button2.Enabled = false;  // блокируем  кнопку 2
}
// Поля объекта класса Form1:
int n;              // число  сторон  многоугольника
double r, p;     // радиус и периметр многоугольника
int dx, dy;       // размеры клиентской области формы
```

```
// Обработчик события Paint:
private void Form1_Paint(object sender, PaintEventArgs e) {
   dx = this.ClientSize.Width;
   dy = this.ClientSize.Height;
   label1.Location = new Point(dx /10, dy/10);
   label2.Location = new Point(dx /10, dy/4);
   textBox1.Location = new Point(2*dx /3, dy/10);
   textBox2.Location = new Point(2*dx /3, dy/4);
   button1.Location = new Point(dx/10,dy/2);
   button2.Location = new Point(dx/10,2*dy/3);
   textBox3.Location = new Point(2*dx /3, 2*dy/3);
}
// Обработчик события Resize:
private void Form1_Resize(object sender, EventArgs e) {
   Refresh();
}
// Обработчик события button1_Click:
private void button1_Click(object sender, EventArgs e) {
   if (!int.TryParse(textBox1.Text, out n)|n < 3) {
      MessageBox.Show("Ошибка в данных!");
      textBox1.Focus();
      return;
      }
   if (!double.TryParse(textBox2.Text, out r)|r <= 0) {
      MessageBox.Show("Ошибка в данных!");
      textBox2.Focus();
      return;
      }
   button2.Enabled = true;
   textBox3.Text = "";
}
// Обработчик события button2_Click:
private void button2_Click(object sender, EventArgs e) {
   p = 2 * n * r * Math.Tan(Math.PI / n);
   textBox3.Text = String.Format("{0:g5}", p);
}
}
}
```

В коде достаточно подробно прокомментированы особенности и назначение отдельных членов класса Form1. В дополнение к комментариям обратим внимание на код обработчика Form1_Paint(). При его выполнении определяются текущие размеры рабочей (клиентской) области формы и, исходя из их значений, определяются местоположения всех элементов

управления. Таким образом, при каждом возникновении события Paint(), пересчитываются координаты всех элементов управления и форма перерисовывается заново. Чтобы событие Paint создавалось при изменении размеров формы, а также при разворачивании свернутой формы, в обработчике события Form1_Resize() выполняется обращение к методу Refresh().

Так как при загрузке формы в полях ввода отсутствуют данные, то в обработчике Form1_Load() свойству button2.Enabled присваивается значение **false**. Тем самым кнопка button2 «Вычислить периметр» не реагирует на внешние воздействия до тех пор, пока пользователь не введёт без ошибок значения числа сторон и радиуса. После этого в обработчике button1_Click() свойство button2.Enabled получит значение **true** и кнопка button2 станет доступной для внешних (пользовательских) воздействий.

В обработчике button2_Click() вычисляется значение периметра многоугольника, и полученное значение изображается в поле textBox3. Для обращения к методу Math.Tan() требуется доступ к пространству имён System. Поэтому в заголовке файла присутствует оператор **using System**.

Задача 12_06. Ввести размер единичной матрицы и построить ее, используя элемент управления DataGridView. (Требуемый результат показан на рис. 12.17).

Результат проектирования формы с элементами **Label**, **TextBox**, **Button**, **DataGridView** показан на рис. 12.16.

Настройка свойств элементов:

Form1.Text = Единичная матрица
Form1.StartPosition = CenterScreen
Form1.MinimumSize = 450; 300
Form1.MaximumSize = 600; 400
lable1.Text = Размер матрицы:
button1.Text = Построить
DataGridView.ColumnHeadersVisible = **False** (убрать заголовки столбцов);
DataGridView.RowHeadersVisible = **False** (убрать названия строк);
DataGridView.AutoSizeColumnMode = Fill («растянуть» строки по ширине элемента).
DataGridView.Anchor = Top, Bottom, Left, Right

В код программы добавить обработчик событий button1_ Click().

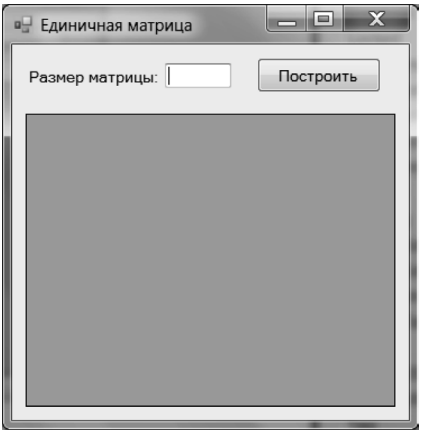
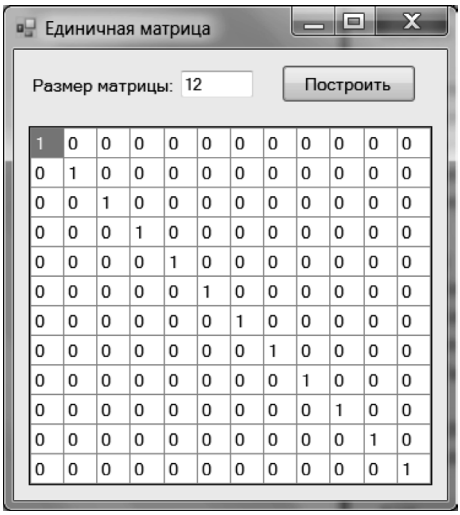


Рис. 12.16. Спроектированная форма (вид в редакторе форм)

Код файла с классом Form1:

```
// 12_06 - Построение единичной матрицы
using System;
using System.Windows.Forms;
namespace Program_6 {
public partial class Form1:Form {
public Form1( ) {
   InitializeComponent( );
}
int n;  // Размер  матрицы
private void button1_Click(object sender, EventArgs e) {
    while (!int.TryParse(textBox1.Text, out n) | n < 1)  {
          MessageBox.Show("Ошибка в данных!");
          textBox1.Focus();
          return;
          }
    dataGridView1.ColumnCount= dataGridView1.RowCount = n;
    for (int i = 0; i < n; i++)
    for (int j = 0; j <n; j++)
       dataGridView1[i, j].Value = i == j ? "1" : "0";
} // button1_Click
}
}
```

Несмотря на предполагаемую сложность исходной задачи, программа очень проста. Объясняется это огромными возможностями элемента управления DataGridView. В данной программе использована только малая часть функциональности этого элемента. После ввода значения n, которое читается в обработчике button1_Click(), определяются число строк (RowCount) и число столбцов (ColumnCount) таблицы, и во вложенных циклах элементам таблицы присваиваются значения 0 или 1. Единицы размещаются на диагонали, и тем самым формируется единичная матрица, показанная на рис. 12.17.



1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1

Рис. 12.17. Результат выполнения программы в задаче 12-06

Задача 12-07. Изобразить с помощью анимации запуск спутника земли. Земля и спутник – закрашенные круги. Земля в центре панели **PictureBox**. По нажатию кнопки **Button** спутник начинает движение по спирали от поверхности земли до достижения орбиты максимального (фиксированного) радиуса. Предусмотреть масштабирование рисунка, то есть корректное выполнение программы при изменении размеров формы.

Для решения задачи разработаем Windows-приложение с двумя элементами пользовательского интерфейса **Button** и **PictureBox** и компонентом **Timer**.

Настройка свойств элементов:

Form1.StartPosition = CenterScreen,
Form1.Text = Полет спутника
button1.Text = Запуск
button1.Font.Size = 12
button1.Font.Bold = True
PictureBox.Anchor = Top, Bottom, Left, Right
PictureBox.BorderStyle = FixedSingle

Проектируя форму, добавим к ней компонент Timer (таймер). Он никак не отображается на самой форме, но условным значком изображается в поле компонентов редактора форм.

Рис. 12.18. Результат проектирования формы для задачи 12-07

Обработчики событий, связанные с элементами формы:

button1_Click_1– нажатие на клавишу 1;
Form1_ClientSizeChanged – изменение размеров формы;
Form1_Load – загрузка формы;
pictureBox1_Paint – перерисовка элемента PictureBox;
timer1_Tick – «срабатывание» таймера.

```
// 12_07 – Иллюстрация полета спутника
using System;                    // Для класса Math
using System.Drawing;            // Для поддержки графики
using System.Windows.Forms;      // Для базового класса Form
```

```
namespace Program_ {
public partial class Form1:Form {
public Form1( ) {
   InitializeComponent( );
     }
   float xz, yz;    // абсцисса и ордината центра земли
   float one;       // единица масштаба
   float rz;        // радиус земли
   float wz, hz;    // левый верхний угол изображения земли
   int k = 0;       // счетчик тиков
// начальный угол положения спутника:
float teta0 = (float)(5 * Math.PI / 4);
float R0;           // начальное расстояние от земли до спутника
   float rs;        // радиус спутника
   float ws, hs;    // левый верхний угол изображения спутника
   float dt = (float)(Math.PI / 100);          // приращение угла
   int kz = 15, ks = 4, kr = 1, kOne = 100;  // коэффициенты
// Обработчик нажатия кнопки:
private void button1_Click(object sender, EventArgs e) {
       timer1.Enabled = true; // Запустить таймер
       }
// Обработчик загрузки формы:
private void Form1_Load(object sender, EventArgs e) {
      timer1.Enabled = false;  // Остановлен таймер
      timer1.Interval = 100;    // Интервал тиков
      pictureData();
      }
// Обработчик изменения размеров формы:
private void Form1_ClientSizeChanged(object sender, EventArgs e) {
    pictureData();
   pictureBox1.Refresh();
   }
// Обработчик события Paint
private void pictureBox1_Paint(object sender, PaintEventArgs e) {
       Graphics target = e.Graphics;
       Pen blackPen = new Pen(Color.Black);
       Pen greenPen = new Pen(Color.Green);
       target.FillEllipse(blackPen.Brush, ws, hs, 2*rs, 2*rs);
       target.FillEllipse(greenPen.Brush, wz, hz, 2*rz, 2*rz);
       }
// Подготовка и масштабирование данных для рисунка:
private void pictureData()  {
  // абсцисса центра земли:
  xz = pictureBox1.Size.Width / 2;
```

```
  // ордината центра земли:
  yz = pictureBox1.Size.Height / 2;
  one = Math.Min(xz, yz)/kOne;   // единица масштаба
  rz = one * kz;                 // радиус земли
  // левый верхний угол изображения земли:
  wz = xz - rz; hz = yz - rz;
  rs = one * ks;                 // радиус спутника
  ws = wz-2; hs = hz-2;          // левый верхний угол спутника
  float R;       // расстояние от земли до спутника
  R0 = (float)Math.Sqrt((ws-xz)*(ws-xz)+(hs-yz)*(hs-yz));
  float dR = one * kr;
  R = Math.Min(R0 + k * dR, one * 80);
  ws = (float)(R * Math.Cos(teta0 + k * dt)) + xz;
  hs = (float)(R * Math.Sin(teta0 + k * dt)) + yz;
  } // pictureData
// Обработчик "тиков" таймера:
private void timer1_Tick(object sender, EventArgs e) {
  // Подготовка и масштабирование данных для рисунка:
  pictureData();
  k++;   // счетчик тиков
  pictureBox1.Refresh();
} // timer1_Tick
}
}
```

Остановимся на особенностях реализации класса Form1. В нём достаточно большое количество полей, определяющих «геометрию» изображения земли и спутника. Так как необходимо, чтобы изображение было масштабируемым и изменялось в течение времени, то размеры и положения земли и спутника должны достаточно часто пересчитываться в процессе работы программы. Необходимые вычисления объединены в методе pictureData(). Этот метод вызывается из обработчиков событий Form1_Load(), Form1_ClientSizeChanged(), timer1_Tick(). В первом из них таймер остановлен и pictureData() вычисляет геометрические соотношения, соответствующие статическому изображению «спутник на земле перед стартом» (см. рис. 12.19). Собственно прорисовка изображения, выполняется в обработчике pictureBox1.Paint(), который вызывается сначала после загрузки формы, затем в ответ на обращения к методу pictureBox1.Refresh(), выполняемые из обработчиков Form1_ClientSizeChanged() и timer1_Tick(). Во втором случае обязательно учитывается «течение времени», то есть изменение

счётчика **int** k. В методе pictureBox1.Paint() создаётся контекст графического устройства и ссылка target на него используется для рисования двух закрашенных окружностей – изображения земли и изображения спутника.

Поясним необходимость именно этих обработчиков. При выполнении программы с графическим интерфейсом операционная система сообщает программе о необходимости перерисовки, возбуждая событие Paint. Класс Form, на базе которого построен класс нашей формы, имеет встроенный обработчик этого события. Поэтому в программе нужно предусмотреть перерисовку не всей формы, а только элемента pictureBox1, где будет размещен рисунок. Поэтому в код добавляется обработчик pictureBox1_Paint.

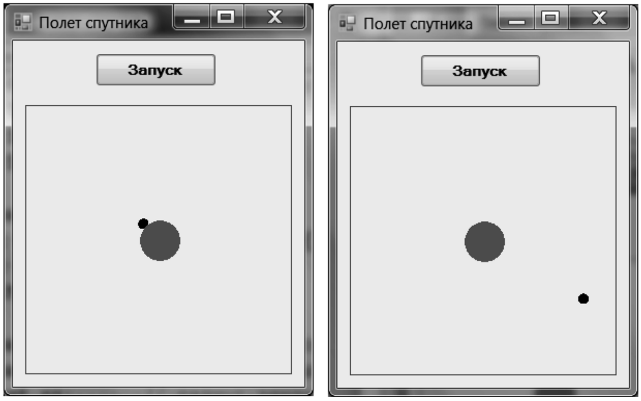


Рис. 12.19. Два этапа выполнения программы (начальное состояние и «полёт»)

Задача 12-08. Написать программу для построения графика функции плотности вероятностей биномиального распределения при разных значениях его параметров.

Приведем необходимые сведения справочного характера (см., например, http://ru.wikipedia.org/wiki/).

Пусть X_1, ..., X_n — конечная последовательность независимых случайных величин с распределением Бернули, то есть

$$X_i = \begin{cases} 1, & p \\ 0, & q \equiv 1 - p \end{cases}, i = 1,...,n.$$

где p – вероятность значения 1, q – вероятность значения 0 (0<=p<=1).

Используя совокупность таких случайных величин, можно следующим образом построить случайную величину с биномиальным распределением:

$$Y = \sum_{i=1}^{n} X_i.$$

Функция плотности вероятности задаётся формулой:

$$p_Y(k) \equiv \mathbb{P}(Y = k) = \binom{n}{k} p^k q^{n-k}, k = 0,\ldots,n,$$

где $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ – биномиальный коэффициент.

Для решения задачи вначале определим класс Distribution для представления биномиального распределения. Поля класса: n, p, q. Для вычисления последовательных значений $P_Y(k)$ удобно применить в классе итератор.

Класс разместим в отдельном файле проекта Program_8 решения Тема_12.

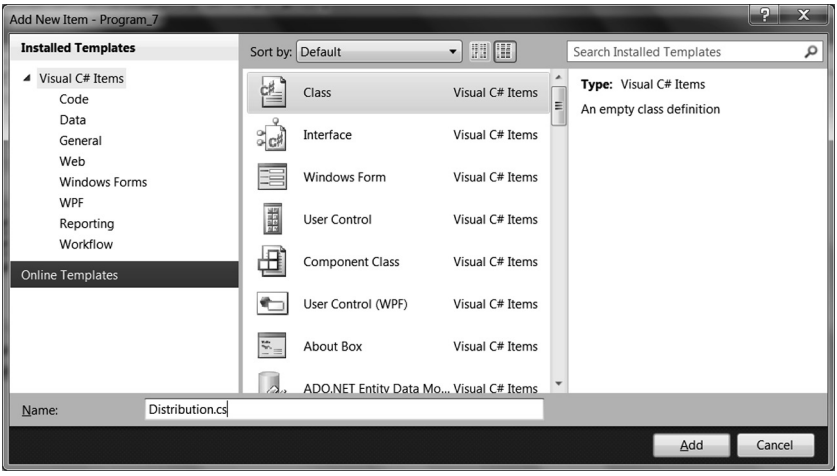


Рис. 12.20. Окно подключения к проекту нового файла

Для создания отдельного файла в существующем проекте вначале, как обычно, создайте в решении Тема_12 проект Windows-приложения с именем Program_8. Затем откройте окно решения (в нашем случае с именем Тема_12) и правой кнопкой активируйте пункт Program_8. В появившемся меню выберите пункт **Add**, в следующем меню – пункт **Class** (перечисленные шаги можно заменить комбинацией **Shift+Alt+C**). Откроется окно (рис. 12.20) для создания и подключения к проекту нового файла. Введем имя создаваемого файла Distribution.cs.

В окне редактора кода появится автоматически подготовленная заготовка кода класса Distribution:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Program_8 {
   class Distribution {
   }
}
```

Автоматически созданный код в файле Distribution.cs сократим, убрав лишние операторы **using**, и дополним функционально значимым кодом. Во-первых, объявим поля данных для задания характеристик биномиального распределения. Во-вторых, введем конструктор класса, позволяющий задавать и проверять значения указанных характеристик. Затем определим вспомогательный метод factorial( ) для вычисления факториала числа и метод-итератор, возвращающий ссылку на перечислимую последовательность, каждый элемент которой - очередное значение функции плотности вероятности биномиального распределения. Особенность перечислимой последовательности – тот факт, что при ее переборе (например, оператору **foreach**) автоматически по очереди становятся доступны все ее элементы (в нашей задаче – значения $p_y(k)$).

Код класса в отдельном файле:

```
// 12_08 - class Distribution
using System;
namespace Program_8 {
class Distribution {
   int n;    // количество членов
   double p, q;   // вероятности
```

```
// Конструктор:
public Distribution(int ni, double pi) {
    if (ni <= 0 || pi < 0 || pi > 1)
          throw new Exception();
    n = ni;
    p = pi;
    q = 1 - p;
}
// Итератор - возвращает значение типа IEnumerable
public System.Collections.IEnumerable getMemb()  {
    for (int k = 0; k <= n; k++)   {
       yield return factorial(n) /
             factorial(n - k) / factorial(k) *
             Math.Pow(p, k) * Math.Pow(q, n-k);
}
}
// Вспомогательный рекурсивный метод:
double factorial(int x)   {
    if (x < 0) return -1;
    if (x == 0) return 1;
    return x * factorial(x-1);
}
} // class Distribution
}
```

«Слабым местом» предлагаемого класса является метод для вычисления факториала, так как при больших значениях его аргумента (превышающим 170) возникает целочисленное переполнение. Это ограничение учтем в основной программе, где ограничим число слагаемых (n – количество независимых случайных величин), входящих в совокупность, формирующую случайную величину с распределением Бернулли.

Подготовив файл с объявлением класса Distribution, перейдем к разработке формы. В визуальном конструкторе форм проектируем форму со следующими характеристиками:

Form1.Text = Биномиальное распределение;
Form1.Size = 600;500;
Form1.StartPosition = CenterScreen

Помещаем на форму контейнер TableLayoutPanel из двух строк и двух столбцов и задаем его свойство:

TableLayoutPanel.Dock = Fill

В ячейки первой строки контейнера TableLayoutPanel помещаем элемент PictureBox. Он сначала разместится в первой ячейке первой строки, но мы так зададим значения его свойств:

PictureBox.Dock = Fill;
PictureBox.ColumnSpan = 2;
PictureBox.BorderStyle = Fixed3D.

В первую ячейку второй строки элемента TableLayoutPanel помещаем контейнер FlowLayoutPanel.

В него помещаем две метки и два текстовых поля со следующими свойствами (см. рис. 12.21):

label1.Text = n=;
textBox1.Text = 7;
label2.Text = p=;
textBox2.Text = 0,4.

Во вторую ячейку второй строки элемента TableLayoutPanel помещаем кнопку со следующими свойствами:

button1.Text = Построить график.
button1.BackColor = Yellow;
button1.Anchor = Top, Bottom, Left, Right.

Для всех элементов на форме установим свойства:

Font.Size = 10;
Font.Bold = True.

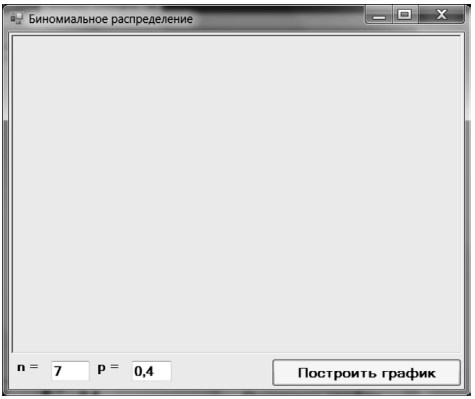


Рис. 12.21. Результат проектирования формы для задачи 12-08

Код заготовки проекта (удалены лишние операторы using):

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace Program_8 {
public partial class Binar:Form {
public Binar( ) {
  InitializeComponent( );
    }
  }
}
```

Чтобы обратить внимание на некоторую независимость класса, представляющего форму Form1 от его базового класса Form, при создании заготовки кода класс Form1 переименован в Binar с помощью средств системы проектирования. Для такого переименования в окне решения щелкните правой кнопкой на имени файла Form1.cs. В меню выберите пункт **Rename**, выделенное при этом имя Form1.cs замените на Binar.cs. Согласитесь на предложение системы заменить все использования имени Form1 на имя Binar.

Дополним заготовку кода обработчиками событий button1_Click(), pictureBox1_Paint(), Binar_SizeChanged().

Элемент button1 – это основной элемент управления программой со стороны пользователя (кроме текстовых полей для ввода значений n и p). При нажатии на эту кнопку должно выполняться чтение и проверка значений из полей ввода, а затем возбуждаться событие pictureBox1.Paint. Без обработчика события Binar_SizeChanged() можно было бы обойтись, если запретить изменения размеров формы, создаваемой на экране для объекта Binar. Единственное действие этого обработчика – возбудить событие pictureBox1.Paint и тем самым перерисовать изображение графика при изменении размеров основного окна программы. В задаче 12_07 объяснялась необходимость применения обработчика pictureBox1_Paint().

После добавления шаблонов обработчиков событий заготовка кода примет такой вид (с учетом удаленных операторов **using** и добавленных комментариев на русском языке):

```
// График биномиального распределения
using System;
using System.Drawing;
```

```
using System.Windows.Forms;
namespace Program_8 {
public partial class Binar:Form {
public Binar( ) { // Конструктор  объекта  формы
  InitializeComponent( );
}
private void button1_Click(object sender, EventArgs e) {
      // Чтение и анализ исходных данных,
      // генерация события pictureBox1_Paint
}
private void pictureBox1_Paint(object sender, PaintEventArgs e) {
      // Анализ состояния, подготовка контекста устройства,
      // данных для графика и рисование на элементе pictureBox1
}
private void Binar_SizeChanged(object sender, EventArgs e) {
      // Генерация события pictureBox1_Paint
}
} // class Binar
} // namespace Program_8
```

Следующий шаг разработки – наполнение функциональностью объявления класса Binar и обработчиков событий в нем. Введем в код следующие методы, каждый из которых решает частную задачу рисования графика на элементе pictureBox1.

void pictureData() – метод для расчета размеров рисунка. На основе размеров клиентской области элемента управления pictureBox1 вычисляются размеры графика и координаты опорных точек для его размещения.

void chartAxes() – метод для рисования осей. На основе размеров графика и опорных точек его размещения в клиентской области элемента pictureBox1 выполняется масштабирование будущего графика, и выводятся изображения его осей с нужными текстовыми обозначениями.

void chartGraphic() – метод, обращается к методу getMemb() объекта класса Distribution, получает значения функции и рисует ее график с учетом масштабов, определенных в методе chartAxes().

Для обменов данными между методами в класс Binar введём соответствующие поля объектов (в комментариях приведённого ниже кода класса указано их назначение). Остановимся на двух полях.

Graphics gr – ссылка на объект библиотечного класса System.Drawing.Graphics, представляющего контекст устройства.

В методе pictureBox1_Paint() создается объект этого класса, «настроенный» на рисование в элементе pictureBox1. Переменной gr присваивается ссылка на этот объект, и после этого методы chartAxes() и chartGraphic(), обращаясь к ссылке gr, выполняют рисование именно в клиентской части элемента pictureBox1.

int flag – флаг для управления рисованием используется для того, чтобы заблокировать рисование при загрузке формы. Дело в том, что при загрузке возникает событие Paint, когда еще не прочитаны значения переменных nn и pp, определяющих характеристики функции распределения. При создании объекта класса Binar значение переменной flag инициализируется нулевым значением. При таком нулевом значении переменной flag метод pictureBox1_Paint() завершается без выполнения каких-либо действий. При выполнении метода button1_Click() (при нажатии клавиши «Построить график») переменная flag получает единичное значение и обработчик события pictureBox1_Paint() при каждом обращении перерисовывает изображение на элементе pictureBox1.

Полный текст объявления формы для проекта построения графика биномиального распределения:

```
// График биномиального распределения
using System;
using System.Drawing;
using System.Windows.Forms;
namespace Program_8 {
public partial class Binar:Form {
public Binar( ) { // Конструктор объекта формы
  InitializeComponent( );
}
// Поля формы, доступные методам:
    int nn; // значение n (число слагаемых)
    double pp; // вероятность p
    // Ссылка на объект биномиального распределения:
    Distribution binar;
    Graphics gr; // Контекст  устройства
    int xMax, yMax;  // Размеры области для рисования
    // Опорные координаты и размеры графика (для рисунка):
    int xMinRis, xMaxRis, yMinRis, yMaxRis, hRis, wRis;
    double pMin, pMax;  // пределы по оси ординат
    // Максимальное значение n на оси абсцисс:
    int nMax;
    int flag = 0; // флаг для управления рисованием
```

```
// Метод для расчета размеров рисунка:
void pictureData() {
    // Оценить размеры области для рисования:
    xMax = pictureBox1.ClientSize.Width;
    yMax = pictureBox1.ClientSize.Height;
    // Опорные координаты и размеры для графика:
    xMinRis = xMax/10 < 5 ? 5: xMax/10;
    xMaxRis = xMax *49 / 50;
    wRis = xMaxRis - xMinRis;
    yMinRis = yMax/50 < 2 ? 2 : yMax/50;
    yMaxRis = yMax * 9 / 10;
    hRis = yMaxRis - yMinRis;
} // pictureData()
// Метод для рисования осей (рисование в pictureBox1):
void chartAxes()  {
    // Подготовка (окраска) холста:
    gr.Clear(Color.Azure);
    // Выделить поле для графика (для рисунка):
    gr.FillRectangle(new SolidBrush(Color.White),
          xMinRis, yMinRis, wRis, hRis);
   // Нарисовать  линии  осей:
   Pen pen1 = new Pen(Color.Black, 2);
   // Рисуем оси абсцисс и ординат:
   gr.DrawLine(pen1, xMinRis, yMaxRis, xMaxRis, yMaxRis);
   gr.DrawLine(pen1, xMinRis, yMaxRis, xMinRis, yMinRis);
    // Нанести насечки и оцифровать ось ординат:
    double alfa, pDel=0.1;
    pMin=0; pMax=0.5;
    for (double p = pMin; p <= pMax; p += pDel) {
       alfa = (p-pMin)/(pMax-pMin);
       double y = yMaxRis * (1 - alfa) + alfa * yMinRis;
       int yt = (int)y, dent = wRis/70;
       gr.DrawLine(pen1, xMinRis-dent, yt, xMinRis, yt);
       gr.DrawString(p.ToString("f2"), Font,
           new SolidBrush(Color.Black),
           xMinRis - 6 * dent, yt-dent);
       // Выбрать оцифровку и шаг оси абсцисс (10 или 20-100):
       int q = (int)Math.Ceiling((double)nn / 10);
       nMax = q * 10;
       int stn = (nMax <= 10 ? 1 : 10 );  // шаг по n
      // Нанести насечки и оцифровать ось абсцисс:
      for (int n = 0; n <= nMax; n += stn) {
         alfa = (double)(n-0)/(nMax-0);
         double x = xMinRis*(1-alfa) + xMaxRis*alfa;
```

```
            int xt = (int)x;
            dent = hRis / 60;
            gr.DrawLine(pen1, xt, yMaxRis, xt,
               yMaxRis + 3*dent/2);
            gr.DrawString(n.ToString("d"), Font,
               new SolidBrush(Color.Black),
               xt-dent, yMaxRis+2*dent);
            }
      }
} // chartAxes()
// Метод получает значения функции и рисует график:
void chartGraphic() {
      int nt = 0;
      foreach (double pro in binar.getMemb())  {
         double alfaX = (double)nt/nMax;
         double x = xMinRis*(1-alfaX) + xMaxRis*alfaX;
         int xt = (int)x;
         double alfaY = (double)(pro - pMin)/(pMax - pMin);
         double y = yMaxRis * (1 - alfaY) + yMinRis * alfaY;
         int yt = (int)y;
         gr.FillEllipse(new SolidBrush(Color.Red),
         xt-3, yt-3, 6, 6);
         nt++;
         }
}  // chartGraphic()
void button1_Click(object sender, EventArgs e) {
      if (!int.TryParse(textBox1.Text, out nn)
            || nn < 0 || nn > 170)  {
            MessageBox.Show("Ошибка!" +
            "Ограничение:\n    0 <= n <= 170");
            textBox1.Focus(); return;
      }
      if (!double.TryParse(textBox2.Text, out pp)
            || pp < 0 || pp > 1)   {
            MessageBox.Show("Ошибка!" +
            " Ограничение: \n    0 <= p <= 1");
            textBox2.Focus(); return;
      }
// Объект  биномиального  распределения:
       binar = new Distribution(nn, pp);
       flag = 1;
       pictureBox1.Refresh();
} // button1_Click
void pictureBox1_Paint(object sender, PaintEventArgs e) {
```

```
            if (flag == 0) return;
        gr = e.Graphics;
        pictureData();
        chartAxes();
        chartGraphic();
} // pictureBox1_Paint
void Binar_SizeChanged(object sender,  EventArgs e) {
        pictureBox1.Refresh();
} // Binar_SizeChanged()
} // class Binar
} // namespace Program_8
```

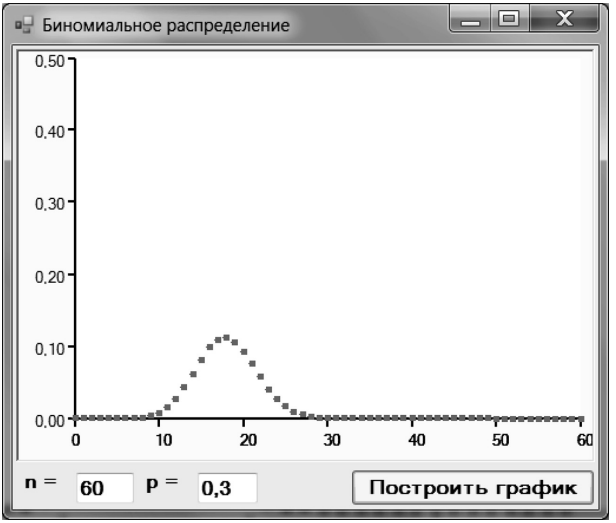


Рис 12.22. Результат построения графика функции плотности вероятности биномиального распределения

ТЕМА 13

ЭВОЛЮЦИОННЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММ

В предыдущих темах нашей целью было изучение на конкретных примерах особенностей применения конструкций языка C#, интегрированной среды разработки MS VS и платформы .NET Framework. Сейчас обратим внимание на дисциплину проектирования, которой удобно пользоваться, выполняя разработку программ в такой среде как MS VS.

В современных методологиях программирования всегда предусматривается итеративность, проект развивается поэтапно, и решения, принятые на очередном этапе, могут существенно влиять на последующие этапы проектирования. На первых этапах создаются версии программного продукта, обладающие минимальной функциональностью, и эта функциональность нарастает при дальнейшей разработке. Можно сказать, что программа при таком подходе эволюционно развивается. Итеративность разработки и постепенное увеличение функциональности проектируемого продукта очень хорошо применимы при использовании интегрированных сред разработки, к которым относится MS VS.

Следуя эволюционным методикам создания программных продуктов, сформулируем рекомендации по выполнению разработки.

Разработка должна выполняться по этапам (шагам), каждый из которых включает: проектирование (в том числе проектирование элементов визуального интерфейса); кодирование (включая синтаксическую и семантическую отладку кода) и тестирование.

1. Этапы разработки неформально делятся на два вида: исследовательские этапы и этапы собственно разработки, добавляющие создаваемому продукту новую функциональность в соответствии с техническим заданием (с условием задачи).

2. Каждый (даже первый) этап, независимо от его вида, предусматривает создание работающего программного продук-

та. (На первом этапе создается программа с минимальной функциональностью.)

3. В конце каждого этапа разработки выполняется анализ существующего (работоспособного) варианта программного продукта. Цель анализа – оценка возможности добавления средств (конструкций) для реализации новых, дополнительных требований, предусмотренных техническим заданием.

4. На основе результатов анализа либо выполняется реорганизация существующего варианта, либо осуществляется принятие новых проектных решений. В обоих случаях это новый этап разработки, включающий кодирование и тестирование.

5. После завершения каждого этапа разработки продукт, оставаясь работоспособным, приобретает новую (зачастую дополнительную) функциональность, либо изменяет уже имеющуюся функциональность, приближаясь к требованиям задания.

Таким образом, продукт постоянно находится в работоспособном состоянии и в него постоянно вносятся изменения. Другими словами, программный продукт от шага к шагу, независимо от вида шагов эволюционирует и его функциональность растет от минимальной до желаемой (указанной в техническом задании).

Покажем, как могут быть реализованы предложенные рекомендации на конкретных примерах.

Задача 13-01. Определите матрицу с размерами 12 на 12, элементы которой принимают случайные значения 0 или 1. Визуализируйте матрицу. Выделяя (например, курсором мыши) конкретный элемент на изображении матрицы, «закрасьте» все смежные элементы, имеющие в матрице те же значения, что и выделенный (начальный). Смежными считать все элементы, имеющие на изображении общую сторону и примыкающие к нему непосредственно и опосредовано (через промежуточные элементы с теми же значениями).

Пример требуемого результата выполнения программы приведен на рис.13.1. Будем называть совокупность выделенных элементов кластером. Таким образом, кластер – это совокупность элементов матрицы, имеющих общие стороны и одинаковые значения.

1	0	1	1	0	0	0	1	0	1	0	0
1	1	1	0	1	1	0	0	1	0	1	1
1	0	1	0	0	0	0	1	0	0	1	0
0	1	0	1	0	1	1	0	1	0	1	0
0	1	1	0	0	1	1	0	1	0	0	1
1	0	1	1	1	1	1	0	0	1	0	0
1	0	1	1	1	0	1	1	1	0	1	0
1	0	0	0	0	0	1	1	1	0	0	0

Рис. 13.1. Кластер ячеек с нулевыми значениями (иллюстрация задания)

Анализируя постановку задачи, отмечаем, что для ее решения необходимо сконструировать средства диалога (интерфейс пользователя) и разработать алгоритмы, обеспечивающие нужную функциональность программы.

Шаг 1. Проектируем форму

Следуя предложенной схеме эволюционного построения программ, начнем с шага, предполагающего создание программы с минимальной функциональностью. Как уже сказано, каждый шаг должен включать три действия: проектирование, кодирование и тестирование. Проектирование на первом шаге решения нашей задачи сводится к выбору элементов интерфейса. В качестве основы выбираем стандартное окно Windows-приложения (форму с заголовком «Выделение кластеров» в новом проекте с именем Clasters в решении Тема_13). Для размещения приглашения пользователю поместим на форму элемент **Label**, разместив на нем текст «Выделите ячейку:». Для изображения матрицы используем элемент **DataGridView**. В качестве еще одного средства диалога с пользователем введем кнопку (элемент **Button**) с надписью «Построить кластер». Отметим, что на первом шаге не предполагается обработки события «нажатие на кнопку», поэтому добавление элемента **Button** можно было бы отложить на второй шаг разработки. Однако в нашей несложной задаче интерфейс очень прост и визуальное конструирование формы может быть выполнено за один шаг разработки. Итак, минимальной функциональностью создаваемого программного продукта может быть форма (класс с именем Form1) с тремя элементами: **Label**, **DataGridView**, **Button**.

Поместив на форму указанные элементы, необходимо явно задать значения некоторых из свойств.

Form1.Text = «Выделение кластеров»
Form1.Font.Bold = True
Form1.StartPosition = CenterScreen
Form1.Size=392;400
Form1.MaximumSize=392;400
Form1.MinimumSize=392;400
Text = «Выделите ячейку:»
label1.Font.Bold = True
Text = «Построить кластер»
button1.Font.Bold = True
DataGridView1.ColumnHeadersVisible = False (убрать заголовки столбцов);
DataGridView1.RowHeadersVisible = False (убрать названия строк);
DataGridView1.AutoSizeColumnMode = Fill («растянуть» строки по ширине элемента).
DataGridView1.Anchor = Top, Bottom, Left, Right.

Кодирование. Для достижения минимальной функциональности достаточно поместить в код класса Form1.cs обработчик события Form1_Load.

```
usingSystem;
using System.Windows.Forms;
namespace Clusters {
public partial class Form1 : Form {
    int M=12, N=12;   // размеры матрицы: M - строки, N - столбцы.
public Form1() {
         InitializeComponent( );
     }
private void Form1_Load(object sender, EventArgs e) {
       dataGridView1.ColumnCount = N;
       dataGridView1.RowCount = M;
       dataGridView1.Rows[0].Cells[0].Selected = false; // снять выделение
}
  }
}
```

Тестирование на первом шаге сводится к трансляции и выполнению программы. Результат выполнения программы в конце первого шага (форма на экране) показан на рис. 13.2. По

результатам автор программы может, возвращаясь к визуальному проектированию формы, подравнять размеры ячеек (клеток элемента DataGridView), изменить взаимное расположение элементов и т. д. Кроме того, следует проверить возможность выделения мышью клеток таблицы DataGridView.

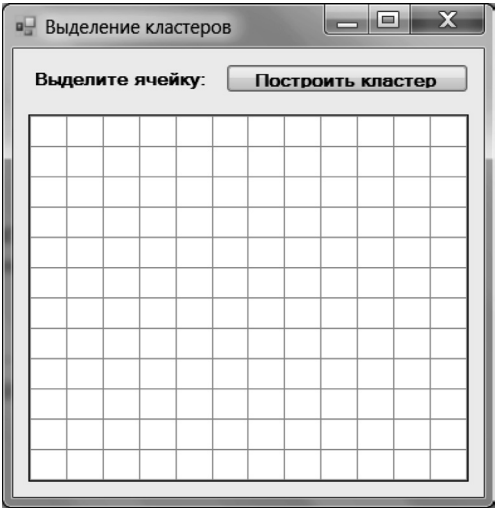


Рис.13.2. Результат выполнения программы на первом шаге

Шаг 2.

В проект Clusters добавим класс матриц со случайными (0 или 1) значениями элементов. Присвоим классу имя RandomMatrix.

```
namespace Clusters {
class RandomMatrix {
        int mm, nn;                          // размеры матрицы
        public sbyte[,] matrix;              // ссылка на массив элементов
public RandomMatrix(int m, int n) {    // конструктор
   mm=m; nn=n;
   matrix = new sbyte[m, n];
   Random gen = new Random();
   for (int i = 0; i < m; i++)
      for(int j = 0; j < n; j++)
          matrix[i,j] = (sbyte)gen.Next(2);
}  // конструктор
}
}
```

В Form1 добавим поле-ссылку на объект класса RandomMatrix:

```
RandomMatrix matr;    // матрица со случайными элементами 0, 1.
```

В конструктор формы Form1() добавим:

```
matr = new RandomMatrix(M, N);
```

В обработчик событий Form1_Load() добавим:

```
// поместить в таблицу значения элементов матрицы:
for (int i = 0; i < M; i++)
   for (int j = 0; j < N; j++);
     dataGridView1[j, i].Value =    // первый индекс - столбец!
     matr.matrix[i,j] == 0 ? "0" : "1";
```

В результате выполнения программы на шаге 2 получим изображение заполненной матрицы, показанное на рис. 13.3:

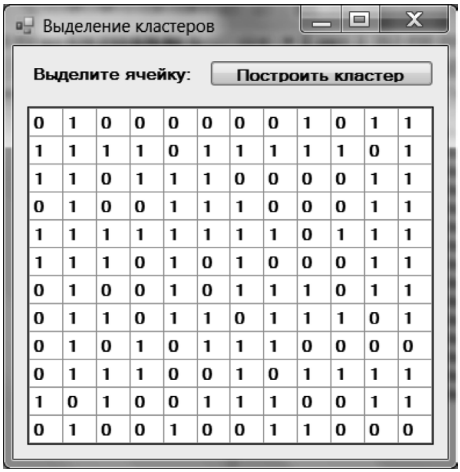


Рис. 13.3. Результат выполнения программы на шаге 2

Шаг 3.

В проект добавим класс Position (можно было бы использовать System.Prawing.Point), объект которого представляет координаты элемента матрицы и ячейки сетки:

```
namespace Clasters {
   class Position {
      public int x, y;
      public Position(int xi, int yi) {x=xi; y = yi;}
      public Position( Position p) {x = p.x; y = p.y;}
}
}
```

В код формы добавим новое поле:

```
Position pos;          // координаты выделенной ячейки
```

Усложним обработчик события button1_Click():

```
private void button1_Click(object sender, EventArgs e) {
  // Ищем выделенную ячейку (K - количество выделенных):
    int K=0;
    for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
       if (dataGridView1.Rows[i].Cells[j].Selected == true) {
       pos = new Position(j,i);  K++;
       }
    if (K == 0) { MessageBox.Show("Нет выделенной ячейки!");
       return;
    }
    if (K > 1) { MessageBox.Show("Выделите одну ячейку!");
       for (int i = 0; i < M; i++)
       for (int j = 0; j < N; j++)
       dataGridView1.Rows[i].Cells[j].Selected = false;
       return;
       }
       else
       MessageBox.Show("x="+pos.x+"  y=" + pos.y);
}  // button1_Click()
```

При выполнении программы с функциональностью, достигнутой на шаге 3, возможны три варианта результата (см. рис. 13.4).

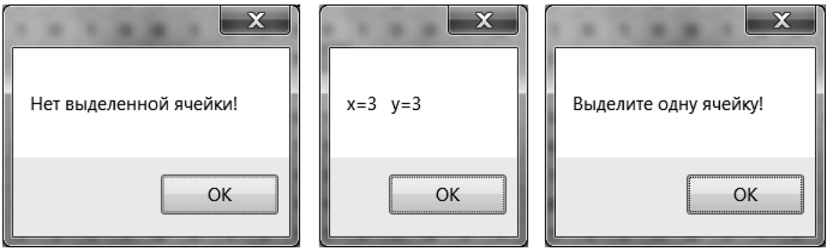


Рис. 13.4. Результаты выполнения программы на шаге 3

Шаг 4.

В класс RandomMatrix добавим метод region() для формирования массива смежных (ближайших, соседних к выделённой) ячеек:

```
public Position[] region(Position p) { // ближайшие  ячейки
   Position [] res = null;
   if (p.x > 0 & p.x < nn-1 & p.y > 0 & p.y < mm-1)
   { res = new Position[5];
     res[0] = new Position(p);
     res[1] = new Position(p.x+1, p.y);
     res[2] = new Position(p.x, p.y-1);
     res[3] = new Position(p.x-1, p.y);
     res[4] = new Position(p.x, p.y+1);
     }
   if (p.x == 0 & p.y ==0) {res = new Position[3]
      {new Position(p), new Position(p.x+1, p.y), new Position(p.x, p.y+1)};
      }
   if (p.x == nn-1 & p.y ==0) {res = new Position[3]
      {new Position(p), new Position(p.x-1, p.y), new Position(p.x, p.y+1)};
      }
   if (p.x == nn-1 & p.y == mm-1) {res = new Position[3]
      {new Position(p), new Position(p.x-1, p.y), new Position(p.x, p.y-1)};
      }
   if (p.x == 0 & p.y == mm-1) {res = new Position[3]
      {new Position(p), new Position(p.x+1, p.y), new Position(p.x, p.y-1)};
      }
   if (p.x == 0 & p.y != 0 & p.y != mm-1) {res = new Position[4]
      {new Position(p), new Position(p.x+1, p.y),
         new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
      }
   if (p.x == nn - 1 & p.y != 0 & p.y != mm-1) {res = new Position[4]
      {new Position(p), new Position(p.x-1, p.y),
         new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
      }
   if (p.y == 0 & p.x != nn-1 & p.x != 0) {res = new Position[4]
      {new Position(p), new Position(p.x+1, p.y),
         new Position(p.x-1, p.y), new Position(p.x, p.y+1)};
      }
   if (p.y == mm-1 & p.x != nn-1 & p.x != 0) {res = new Position[4]
      {new Position(p), new Position(p.x+1, p.y),
         new Position(p.x-1, p.y), new Position(p.x, p.y-1)};
      }
   return res;    // этот оператор будет заменён
}  // region()
```

В обработчик события button1_Click() вносим дополнение:

```
else {
     MessageBox.Show("x="+pos.x+"  y="+pos.y);
     foreach (Position s in matr.region(pos))
        dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
     }
```

После этих добавлений нажатие на клавишу приведёт к выделению всех элементов, смежных с выбранным, как показано на рис. 13.5.

Рис. 13.5. Результаты выполнения программы на шаге 4

Шаг 5.

Дополним метод region() операторами для удаления окраски тех клеток, указывающих на элементы матрицы, значения которых отличны от помеченного элемента. Для этого вместо оператора **return** res; в метод region() помещаем код:

```
// Проверка найденных позиций:
int  r=0;
for(int k=0; k < res.Length; k++) {
   if (matrix[res[0].y, res[0].x] ==
        matrix[res[k].y, res[k].x]) r++;}
   Position [] temp = new Position[r];
   for(int k=0, j=0; k < res.Length; k++)
      if (matrix[res[0].y, res[0].x] == matrix[res[k].y, res[k].x])
         temp[j++] = res[k];
   return temp;
}  // region()
```

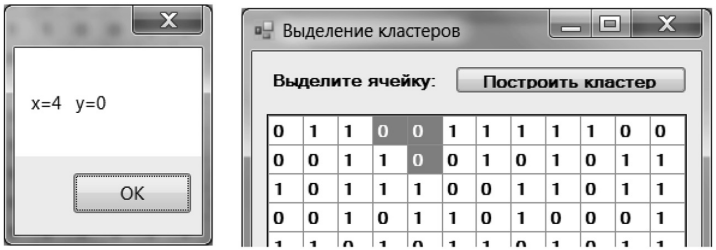


Рис. 13.6. Результаты выполнения программы на шаге 5

Шаг 6.

Теперь можно дополнять программу средствами для выделения кластера, то есть построения и окрашивания всей совокупности одинаковых смежных ячеек, прилегающих прямо и косвенно (через соседей) к начальной (выделенной).

На этом шаге придется заниматься разработкой алгоритма построения кластера. Для этого необходимо проанализировать каждого из соседей начального элемента. Так как у каждого из анализируемых элементов в свою очередь могут оказаться подходящие смежные элементы и т.д., то размер кластера заранее предсказать нельзя. Выбор элементов, принадлежащих кластеру, это типичная комбинаторная задача, в общем виде сводящаяся к требованию: «выделите все такие, и только такие элементы множества, которые удовлетворяют заданному условию».

Применим для решения нашей задачи один из вариантов основного метода комбинаторных вычислений – метода ветвей и границ, который еще называют методом перебором с возвратом. Основных подходов к реализации метода ветвей и границ два: применение стека, сохраняющего выбранные, но еще не включенные в результат элементы, и применение рекурсивного алгоритма. Остановимся на рекурсивном подходе.

Пусть P – позиция, определяющая элемент матрицы, для которого должно быть выделено множество соседей, удовлетворяющих требованию принадлежности кластеру.

Пусть R – множество позиций, уже включенных в кластер.

Пусть T – множество соседей элемента P, удовлетворяющих требованию принадлежности кластеру, но еще не включенных в R и в предшествующие множества T.

Пусть F(P, R) – рекурсивная процедура построения кластера R из позиции P.

Псевдокод процедуры можно представить так:

```
F(P,R)  {
Поместить позицию P в кластер R
Определить T(P) – множество соседей элемента P
Пометить P и все Pi из T как уже выбранные
Цикл по всем Pi из T
    Вызов F(Pi, R)
Выход из F()
}
```

До обращения к процедуре P – задано, R – пусто. После выхода из процедуры F(P,R) в R – множество позиций, включенных в кластер. Процедура имеет одну особенность – в ней должна быть предусмотрена защита от повторного включения в очередное множество соседей уже выбранных ранее элементов. Поэтому при определении T(P) нужно выбирать во множество соседей элементы, соответствующие кластеру, но еще не включенные в рассмотрение (не помеченные ранее). Для выбора множества соседей, пригодных для включения в кластер, хотелось бы применить разработанный нами метод RandomMatrix.region(). Однако в нем не предусмотрена оценка «помеченности» элементов. Помечать уже использованные элементы можно разными способами. С целью максимального использования без каких-либо изменений уже закодированных фрагментов нашей программы помечать выбранные элементы будем, изменяя значения элементов матрицы, на которой решается наша задача. Тем самым появляется возможность применить для получения множества T(P) метод RandomMatrix.region(). По условию задачи элементы матрицы имеют значения 0 и 1, метод region() выбирает соседние элементы, имеющие то же значение, что и начальный. Если изменить значения уже использованных элементов матрицы, то они не будут считаться подходящими для кластера, и метод region() «проигнорирует» их присутствие по соседству с начальным элементом.

Техническое решение: будем помечать рассмотренный элемент, увеличивая соответствующее значение элемента матрицы на 2. Не останавливаясь на других деталях программной реализации алгоритма, включим в класс RandomMatrix следующий рекурсивный метод для построения кластера:

```
public void spot(Position p, ref Position[] res) {
   int size = res.Length;
   Position [] temp = res;
   res = new Position [size+1];
   temp.CopyTo(res,0);
   res[size] = p;
   Position [] path = region(p);
   foreach (Position g in path)
   matrix[g.y, g.x] += 2; // Помечаем выделенные элементы
   for (int i=1; i<path.Length; i++)      {
         matrix[path[i].y, path[i].x] -= 2;
         spot(path[i], ref res);
         }
   return;
}  // spot()
```

Как сказано, в методе spot() уже рассмотренные элементы специальным образом помечаются. Для этого соответствующие элементы матрицы увеличиваются на 2. После этого они становятся «не похожими» на подключаемые к кластеру элементы. Тем самым исключается дублирование и возможное зацикливание. Однако для продолжения поиска новый начальный элемент приводится в исходное состояние (элемент матрицы уменьшается на 2).

Принудительное «окрашивание» элементов, уже включенных в кластер, требует их восстановления для возможности повторных построений того же кластера (возможно из другого начального элемента). Это приходится явно выполнять в коде. Таким образом, в обработчик события button1_Click() вносим следующие изменения:

```
else {  MessageBox.Show("x="+pos.x+"  y="+pos.y);
        Position [] cla = new Position[0];
        matr.spot(pos, ref cla);
        foreach (Position s in cla) {
           dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
           matr.matrix[s.y, s.x] -=2;
           }  // foreach
     }  // else
```

Этот код должен заменить следующий, исключаемый из метода button1.Click(), фрагмент:

```
else {
    MessageBox.Show("x="+pos.x+"  y="+pos.y);
    foreach (Position s in matr.region(pos))
        dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
    }
```

Самое главное в новом коде – замена обращения к методу matr.region(pos) на вызов рекурсивного метода matr.spot(pos, **ref** cla). Различия между этими методами и пометка («окрашивание») уже рассмотренных элементов определили особенности нового заменяющего кода.

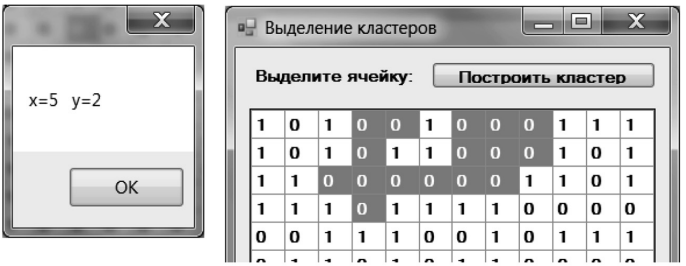


Рис. 13.7. Результаты выполнения программы на шаге 6

Чтобы подвести итог пошаговой разработки программы, приведем ее полный код (конечно, без файлов, формируемых автоматически средой разработки).

```
// Выделение кластеров в таблице на экране
using System;
using System.Windows.Forms;

namespace Clasters {
public partial class Form1:Form {
    int M=12, N=12;          // размеры матрицы: M - строки, N - столбцы
    RandomMatrixmatr;    // матрица со случайными элементами 0, 1.
    Position pos;            // координаты выделенной ячейки
public Form1( ) {
    InitializeComponent( );
    matr = new RandomMatrix(M, N);
    }
private void Form1_Load(object sender, EventArgs e) {
    dataGridView1.ColumnCount = N;
    dataGridView1.RowCount = M;
```

```
// снять выделение клетки (ячейки):
    dataGridView1.Rows[0].Cells[0].Selected = false;
// Поместить в таблицу значения элементов матрицы:
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        dataGridView1[j, i].Value =  // первый индекс - столбец!
            matr.matrix[i,j] == 0 ? "0" : "1";
    }
private void button1_Click(object sender, EventArgs e) {
   // Ищем выделенную ячейку (K - количество выделенных):
   int K=0;
   for (int i = 0; i < M; i++)
   for (int j = 0; j < N; j++)
      if (dataGridView1.Rows[i].Cells[j].Selected == true) {
      pos = new Position(j,i);  K++;
      }
   if (K == 0) { MessageBox.Show("Нет выделенной ячейки!");
     return;
   }
  if (K > 1) { MessageBox.Show("Выделите одну ячейку!");
     for (int i = 0; i < M; i++)
     for (int j = 0; j < N; j++)
     dataGridView1.Rows[i].Cells[j].Selected = false;
     return;
   }
      // Замененный фрагмент кода:
      //else {
      //   MessageBox.Show("x="+pos.x+"  y="+pos.y);
      //   foreach (Position s in matr.region(pos))
      //       dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
      //   } // else
      // Заменяющий фрагмент кода:
      else {
      MessageBox.Show("x="+pos.x+"  y="+pos.y);
          Position [] cla = new Position[0];
          matr.spot(pos, ref cla);
          foreach (Position s in cla) {
             dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
            matr.matrix[s.y, s.x] -=2;
           }  // foreach
          }  // else
}  // button1_Click
}  // class Form1
}
```

```
// Класс для представления координат в таблице и матрице
namespace Clasters {
class Position {
  public int x, y;
  public Position(int xi, int yi) {x=xi; y = yi;}
  public Position( Position p) {x = p.x; y = p.y;}
}
}

// Класс поддержки функциональности программы
usingSystem;
namespace Clasters {
class RandomMatrix {
    int mm, nn;                  // размеры матрицы
    public sbyte[,] matrix; // ссылка на массив элементов матрицы
public RandomMatrix(int m, int n) { // конструктор
    mm=m; nn=n;
    matrix = new sbyte[m, n];
    Random gen = new Random();
    for (int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
            matrix[i,j] = (sbyte)gen.Next(2);
}  // конструктор RandomMatrix()
// Выделение ближайших точек:
public Position[] region(Position p) {
   Position [] res = null;
   if (p.x > 0 & p.x < nn-1 & p.y > 0 & p.y < mm-1)
   { res = new Position[5];
     res[0] = new Position(p);
     res[1] = new Position(p.x+1, p.y);
     res[2] = new Position(p.x, p.y-1);
     res[3] = new Position(p.x-1, p.y);
     res[4] = new Position(p.x, p.y+1);
   }  //***************
   if (p.x == 0 & p.y ==0) {res = new Position[3]
      {new Position(p), new Position(p.x+1, p.y),
         new Position(p.x, p.y+1)};
   }
  if (p.x == nn-1 & p.y ==0) {res = new Position[3]
      {new Position(p), new Position(p.x-1, p.y),
         new Position(p.x, p.y+1)};
   }
  if (p.x == nn-1 & p.y == mm-1) {res = new Position[3]
      {new Position(p), new Position(p.x-1, p.y),
```

```
            new Position(p.x, p.y-1)};
    }
    if (p.x == 0 & p.y == mm-1) {res = new Position[3]
        {new Position(p), new Position(p.x+1, p.y),
            new Position(p.x, p.y-1)};
    } //*****************
    if (p.x == 0 & p.y != 0 & p.y != mm-1) {res = new Position[4]
        {new Position(p), new Position(p.x+1, p.y),
            new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
    }
    if (p.x == nn - 1 & p.y != 0 & p.y != mm-1) {res = new Position[4]
        {new Position(p), new Position(p.x-1, p.y),
            new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
    }
    if (p.y == 0 & p.x != nn-1 & p.x != 0) {res = new Position[4]
        {new Position(p), new Position(p.x+1, p.y),
            new Position(p.x-1, p.y), new Position(p.x, p.y+1)};
    }
    if (p.y == mm-1 & p.x != nn-1 & p.x != 0) {res = new Position[4]
        {new Position(p), new Position(p.x+1, p.y),
            new Position(p.x-1, p.y), new Position(p.x, p.y-1)};
}
//  return res;// Этот оператор потом заменяем.
// Проверка найденных позиций:
int r=0;
for(int k=0; k < res.Length; k++) {
    if (matrix[res[0].y, res[0].x] ==
                matrix[res[k].y, res[k].x]) r++;}
    Position [] temp = new Position[r];
    for(int k=0, j=0; k < res.Length; k++)
        if (matrix[res[0].y, res[0].x] ==
                matrix[res[k].y, res[k].x])
            temp[j++] = res[k];
    return temp;
} // region()
public void spot(Position p, ref Position[] res) {
    int size = res.Length;
    Position [] temp = res;
    res = new Position [size+1];
    temp.CopyTo(res,0);
    res[size] = p;
    Position [] path = region(p);
    foreach (Position g in path)
        matrix[g.y, g.x] += 2;  // Помечаем выделенные элементы
```

```
    for (int i=1; i<path.Length; i++)      {
          matrix[path[i].y, path[i].x] -= 2;
          spot(path[i], ref res);
          }
    return;
}  // spot()
}  // class RandomMatrix
}
```

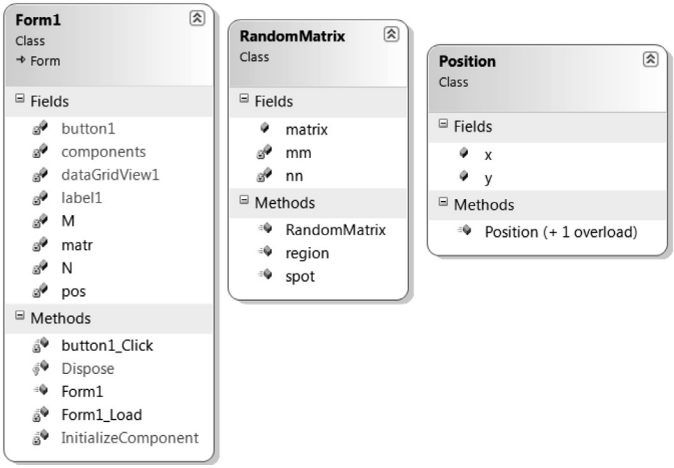


Рис. 13.8. Структура классов проекта Clusters, подготовленных вручную

Задача 13-02. Разработать библиотеку классов для численного решения систем дифференциальных уравнений методом Рунге-Кутта 4-го порядка. Для тестирования и проверки работоспособности созданной библиотеки решить дифференциальное уравнение второго порядка, описывающее колебания тела (шарика) массы m вдоль горизонтальной оси x при условии, что шарик с помощью пружин закреплен между неподвижными опорами и может двигаться без трения только горизонтально.

Вид уравнения гармонических колебаний:

$$x''(t)+\frac{k}{m}x(t)=0,$$

где t – время;
m – масса шарика;
k – коэффициент упругости пружины.

С точки зрения теории дифференциальных уравнений, задача проста и имеет следующее аналитическое решение:

$$x(t) = A\sin(\omega t + b),$$

где $\omega = \sqrt{\frac{k}{m}}$ (круговая, иначе угловая частота),

$A = \sqrt{x^2(t_n) + (\frac{x'(t_n)}{\omega})^2}$ (амплитуда колебаний),

$b = arctg\left(\frac{x(t_n) * \omega}{x'(t_n)}\right)$ (начальная фаза колебаний).

Существование аналитического решения позволяет сравнить с ним результат численного решения уравнения и оценить точность решения. Итак, начнем с создания программы (проект Oscilation в решении Тема_13), построения графика аналитически заданного решения задачи о колебаниях тела.

В программе предусмотреть графическое представление решения уравнения в виде построения зависимости x(t) при разных начальных условиях.

Исходными данными для программы должны быть:

tn – начало интегрирования;

tk – конец интегрирования;

dt – шаг интегрирования;

m – масса тела (шарика);

k – коэффициент упругости пружин;

x(tn) – начальное положение тела (уклонение от положения равновесия);

x'(tn) – начальная скорость тела.

Хотя основной целью нашего проекта является создание библиотеки классов для решения систем дифференциальных уравнений, но целесообразно начинать не с алгоритмов библиотеки, а с разработки вспомогательных модулей, которые будут использованы для тестирования библиотеки. Такой подход позволит следовать идеологии эволюционного программирования и постепенно «выращивать» программу, последовательно расширяя ее функциональность и обеспечивая тестирование библиотеки.

Для решения задачи требуется разработать интерфейс пользователя и реализовать алгоритмы, обеспечивающие нужную функциональность программы.

Шаг 1. Проектирование формы со стандартными элементами управления.

Следуя рекомендациям эволюционной методологии, начнем с шага, предусматривающего создание программы с минимальной функциональностью. В качестве основы создадим форму, показанную на рис. 13.9.

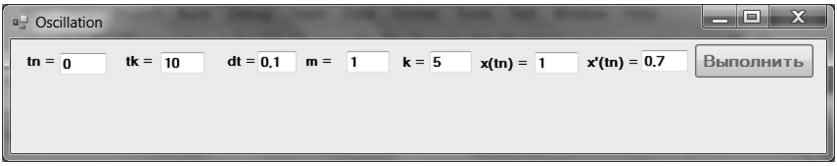


Рис. 13.9. Форма с элементами пользовательского интерфейса

При проектировании формы использованы стандартные средства панели инструментов (Toolbox). На форму помещены: элементы label1 - label7, играющие роль обозначений для полей ввода textBox1 – textBox7, и кнопка button1 с надписью «Выполнить».

Начальные значения свойств элементов формы:

Form1.Text = Oscilation
label1.Text = tn (начало интегрирования);
textBox1.Text = 0
label2.Text = tk (конец интегрирования);
textBox2.Text = 10
label3.Text = dt (шаг интегрирования);
textBox3.Text = 0.1;
label4.Text = m (масса тела);
textBox4.Text = 1;
label5.Text = k (коэффициент упругости пружины);
textBox5.Text = 5;
label6.Text = x(tn) (начальное положение тела);
textBox6.Text = 1;
label7.Text = x'(tn) (начальная скорость тела);
textBox7.Text = 0.7.

Программу (проект Oscillation) можно оттранслировать и выполнить. На экране появится изображение формы. Программа работоспособна, но ничего «не умеет» делать, кроме как изо-

бражать эту форму. Нужно переходить к следующей итерации, то есть наращивать функциональность программы.

Шаг 2. Добавление на форму компонента ZedGraph

Для построения изображения графика решения уравнения можно самостоятельно программировать соответствующий фрагмент программы, но более выгодно, надежно и эффективно использовать существующие программные решения. Воспользуемся свободно распространяемым программным продуктом – библиотекой ZedGraph (http://zedgraph.org). Его можно загрузить, например, с сайта http://sourceforge.net/projects/zedgraph/. Руководство по применению библиотеки ZedGraph.dll доступно по адресу http://www.codeproject.com/csharp/zedgraph.asp.

Для применения библиотеки ZedGraph.dll удобнее всего включить ее как компонент в набор инструментов (Toolbox) Visual Studio. Последовательность шагов:

1) Поместить файл ZedGraph.dll на диск (например, в каталог Library\...\Debug\).

2) В VS открыть окно ToolBox и, нажав правую кнопку мыши, выбрать пункт **ChooseItems**, затем .NET Framework Components, найти ZedGraphControl и нажать кнопку OK. **(ToolBox → Choose Items → .NET Framework Components → ZedGraphControl → OK).**

3) Управляющий элемент ZedGraphControl появился в списке элементов управления ToolBox.

В русскоязычной версии VS2012 последовательность будет такой:

ВИД → Другие окна → Панель элементов (или **Ctrl+W,X**) **→ стандартные элементы управления**

Для размещения экземпляра компонента ZedGraphControl на форме нашего проекта Oscillation нужно в редакторе формы «перетащить» ZedGraphControl на форму и настроить свойство Anchor, установив привязки Top, Bottom, Left, Right. Результат этих действий показан на рис. 13.10.

Надписи и оси на изображении компонента нанесены со значениями по умолчанию. Их нужно настроить программно. Подробное описание возможностей компонента ZedGraph выходит за рамки нашего изложения, поэтому объяснения приведем только для тех действий, которые будут прописаны в коде.

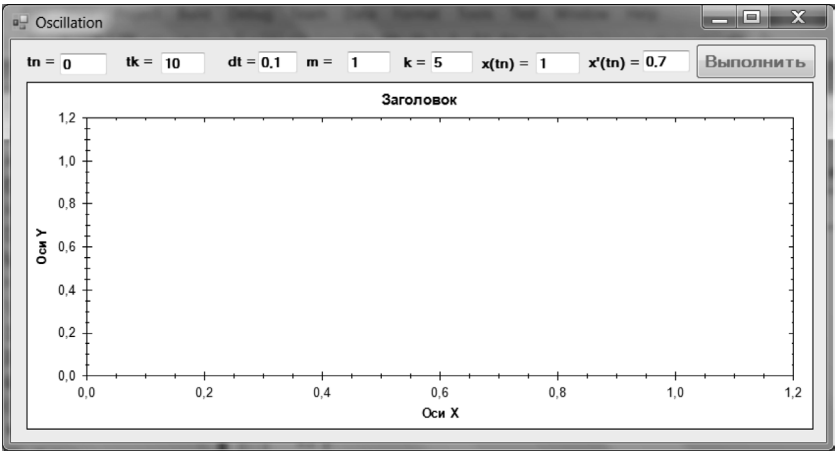


Рис. 13.10. Форма с компонентом ZedGraph в окне редактора

Компонент получает имя zedGraphControl1. Среди свойств компонента настроим свойство Visible = **False**. Это позволит убрать изображение компонента с экрана при загрузке формы. «Восстанавливать» изображение будем по нажатию кнопки «Выполнить», когда пользователь установит значения вводимых данных. Таким образом, при запуске программы форма будет соответствовать изображению на рис. 13.9.

Для того, чтобы использовать компонент zedGraphControl1 в программе (в проекте Oscillation) нужно включить в код Form1.cs оператор:

```
using ZedGraph;
```

Кроме того, в числе ссылок проекта Oscillation должна быть ссылка на библиотеку ZedGraph.dll. Для этого: в окне Solution Explorer выберем и активируем правой кнопкой пункт References. Активируем пункт Add Reference, затем в окне Add Reference выберем Browse и, найдя ZedGraph.dll, нажмем OK.

Теперь проект можно компилировать и выполнить, получив форму, показанную на рис. 13.9. Напоминаем, что элемент ZedGraph1 не будет виден, так как для его свойства Visible установлено значение **false**.

Шаг 3. Построение графика аналитического решения

Дополним код нашего проекта Oscillation методом для обработки нажатия на кнопку «Выполнить» и начнем расширение кода модуля Form1.cs. Результат:

```
// График аналитического решения уравнения
using System;
using System.Drawing;
using ZedGraph;
using System.Windows.Forms;
namespace Oscillation {
public partial class Form1:Form {
public Form1( ) {
    InitializeComponent( );
    }
    // Исходные данные для интегрирования:
    double tn, tk, dt, m, k, x0, x10;
    int size;   // Размер массивов результатов решения
    double[] xt, time;
    // Заглушка для метода чтения данных:
bool readData()  {
    tn = 0; tk = 10; dt = 0.1; m = 1;
        k = 5; x0 = 1; x10 = 0.7;
    size = (int)((tk - tn) / dt + 1);
    xt = new double[size];
    time = new double[size];
    return true;
    }
// Аналитическое решение уравнения:
void analytical_Solution() {
    double t = tn, sq = Math.Sqrt(k/m);
    double A = Math.Sqrt(x0*x0 + (x10/sq)*(x10/sq));
    double b = 0;
    if (x0 == 0 & x10 > 0)  b = 0;
    if (x0 == 0 & x10 < 0)  b = Math.PI;
    if (x0 != 0 & x10 == 0) b = Math.PI - Math.Atan(x0 * sq / x10);
    if (x0 != 0 & x10 > 0) b = Math.Atan(x0 * sq / x10);
    if (x0 != 0 & x10 < 0) b =  Math.PI + Math.Atan(x0 * sq / x10);
    for (int i = 0; i < size; i++)  {
        time[i] = t;
        xt[i] = A * Math.Sin(sq*t +b);
        t += dt;
        }
    }  // analytical_Solution
```

```
private void button1_Click(object sender, System.EventArgs e) {
      // Прочитать и проверить введенные данные:
      if(!readData()) return;
      // Аналитическое решение уравнения (получить xt[] и time[]):
      analytical_Solution();
      // Показать элемент zedGraph:
      zedGraphControl1.Visible = true;
      // Создать панель для рисования:
      GraphPanepane = zedGraphControl1.GraphPane;
      // Очистить список прошлых кривых:
      pane.CurveList.Clear();
      // Создадать список для точек графика:
      PointPairList list = new PointPairList();
      // Заполнить список точек:
      for (int i = 0; i < xt.Length; i++)
      // Добавить в список координаты одной точки:
      list.Add(time[i], xt[i]);
      // Создать кривую с названием "Аналитическое решение":
      LineItemmyCurve = pane.AddCurve("Аналитическое решение",
            list, Color.Black, SymbolType.None);
      // Включить "сглаживание" изображаемой кривой:
      myCurve.Line.IsSmooth = true;
       // Общий заголовок графика:
   pane.Title.Text = "Гармонические колебания";
   // Текст подписей для осей x и y:
   pane.XAxis.Title.Text = "Время";
   pane.YAxis.Title.Text = "Отклонения шарика";
   // Обновить график:
    zedGraphControl1.AxisChange();
    zedGraphControl1.Invalidate();
} // button1_Click
} // class Form1
} // Oscillation
```

Комментарии в коде многое поясняют, но некоторые объяснения будут не лишними. Для построения графика с помощью компонента ZedGraph требуется получить массив координат точек графика. Абсциссы этих точек будем представлять массивом time[], соответствующие им ординаты — массивом xt[]. Размеры массивов одинаковые и определяются исходными данными: size = (int)((tk - tn) / dt + 1);

Так как чтение и проверка вводимых пользователем данных требует аккуратного и трудоемкого кодирования, а нам хотелось бы в первую очередь проверить правильность подключения

компонента ZedGraph, то на этом шаге разработки проекта в код класса Form1 включен метод-заглушка readData(). В нем переменным tn, tk, dt, m, k, x0, x10 присваиваются фиксированные значения, вычисляется значение переменной size и определяются массивы time[size] и xt[size] для представления координат точек графика.

Метод analytical_Solution() присваивает элементам массивов time[size] и xt[size] конкретные значения координат точек графика аналитического решения уравнения гармонических колебаний шарика.

Названные методы вызываются в обработчике события button1_Click().

Как только методы ReadDate() и analytical_Solution() выполнены, оператор

```
zedGraphControl1.Visible = true;
```

восстанавливает показ изображения компонента zedGraphControl1 на форме.

Следующие операторы используют средства библиотеки zedGraph. Собственно «рисование» выполняется на панели GraphPane pane с помощью метода pane.AddCurve(). Его первый параметр – название графика, второй параметр – список точек изображения, третий параметр – цвет графика, четвертый параметр – стиль (мода) изображения кривой. Результат выполнения программы приведен на рис. 13.11.

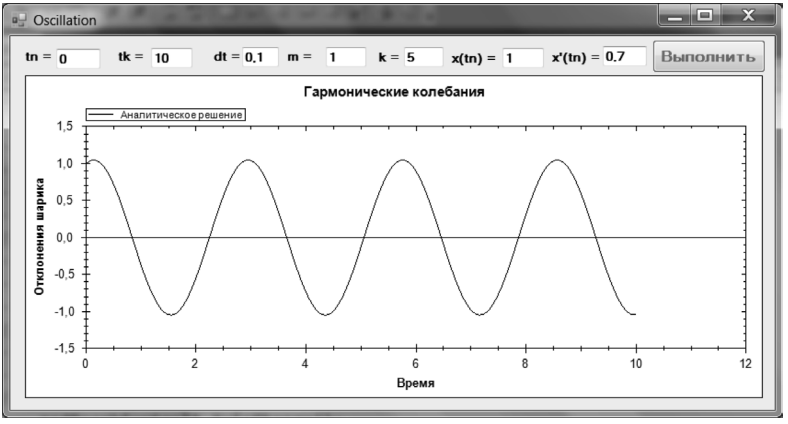


Рис. 13.11. График аналитического решения уравнения

Шаг 4. Ввод исходных данных

Для работы с программой у пользователя должна быть возможность вводить в поля формы исходные данные. Все данные имеют тип **double**, поэтому удобно предоставить при вводе возможность записывать значения, отделяя дробную часть от целой как точкой, так и запятой. Для решения этой вспомогательной задачи введем в программу метод для чтения вещественного числа из строки с учетом американской (точка) и европейской (запятая) записей чисел.

Такой метод был уже разработан при решении задачи 08-04. Метод принадлежит классу Methods, размещённому в библиотеке Library. Его заголовок:

```
public static bool readDouble(string st, out double res)
```

Для ипользования в нашем проекте этого метода необходимо в число ссылок (References) проекта Oscillation добавить ссылку на Library.dll, а в заголовок кода Form1.cs поместить оператор **using** Library;. На основе этого метода напишем метод чтения данных из полей формы, заменяя им предыдущую заглушку:

```
// Метод чтения и проверки данных:
bool readData() {
    if (Methods.readDouble(textBox1.Text, out tn) ==
                        false || tn < 0)    {
       MessageBox.Show("Ошибка в значении tn");
       textBox1.Focus();
       return false;
       }
    if (Methods.readDouble(textBox2.Text, out tk) ==
                        false || tk <= tn)   {
       MessageBox.Show("Ошибка в значении tk или tn");
       textBox2.Focus();
       return false;
       }
    if (Methods.readDouble(textBox3.Text, out dt) ==
                        false || dt <= 0 || dt > tk - tn)  {
       MessageBox.Show("Ошибка в значениях dt или tn или tk");
       textBox3.Focus();
       return false;
       }
    if (Methods.readDouble(textBox4.Text, out m) ==
                        false || m <= 0)   {
       MessageBox.Show("Ошибка в значении m");
       textBox4.Focus();
```

```
        return false;
        }
    if (Methods.readDouble(textBox5.Text, out k) ==
                        false || k <= 0)  {
        MessageBox.Show("Ошибка в значении k");
        textBox5.Focus();
        return false;
        }
    if (Methods.readDouble(textBox6.Text, out x0) == false) {
        MessageBox.Show("Ошибка в значении x0");
        textBox6.Focus();
        return false;
        }
    if (Methods.readDouble(textBox7.Text, out x10) == false) {
        MessageBox.Show("Ошибка в значении x10");
        textBox7.Focus();
        return false;
        }
    size = (int)((tk - tn) / dt + 1);
    xt = new double[size];
    time = new double[size];
    returntrue;
} // readData()
```

Теперь наша программа может работать при разных исходных данных (см. рис. 13.12).

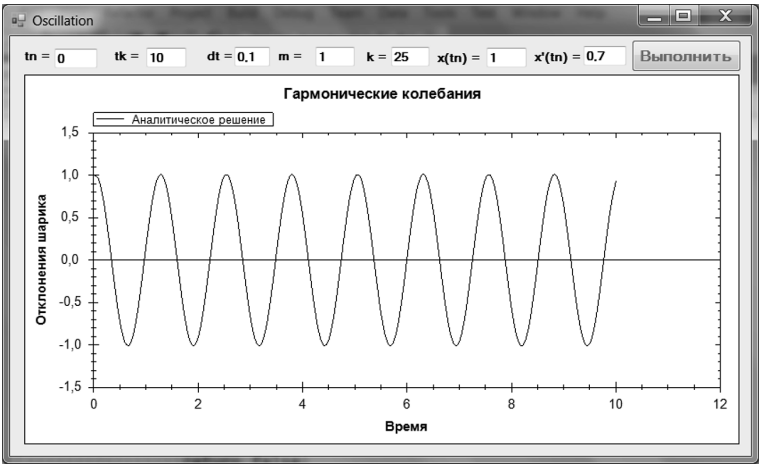


Рис. 13.12. Аналитическое решение при измененных данных

Шаг 5. Библиотека классов для численного решения дифференциальных уравнений

Наш проект уже пригоден для тестирования метода численного интегрирования систем дифференциальных уравнений. Как только метод будет готов, с его помощью можно получить массив точек графика решения дифференциального уравнения гармонических колебаний и сравнить этот график с аналитическим решением. В качестве основы для создания метода интегрирования возьмём метод Рунге-Кутта 4-го порядка (см., например, [16]).

Для разработки достаточно универсального метода, выполняющего численное интегрирование систем обыкновенных дифференциальных уравнений, удобно построить и использовать класс векторов. Объясним целесообразность такого подхода.

Решаемое дифференциальное уравнение должно быть представлено в форме Коши, то есть записано в виде системы:

$$x' = f(x, t),$$

где x', $f(x,t)$, x – вектор-столбцы, t – независимая переменная, изменяющаяся на интервале от t_n до t_k с шагом h. Начальное состояние, то есть вектор $x_n = x(t_n)$ – должно быть задано.

На каждом шаге интегрирования последовательно вычисляются векторы:

$$k_1 = h*f(t_n, x_n),$$
$$k_2 = h*f(t_n + h/2, x_n + k_1/2),$$
$$k_3 = h * f(t_n+\mathrm{h}/2, x_n + k_2/2),$$
$$k_4 = h * f(t_n + h, x_n + k_3),$$
$$x_{n+1} = x_n + (k_1 + 2 * k_2 + 2 * k_3 + k_4) / 6.$$

Полученный вектор x_{n+1} служит начальным значением для следующего шага.

Из приведенных соотношений видно, что необходим класс векторов, над которыми можно выполнять покомпонентные операции сложения, умножения числа на вектор, умножения вектора на вещественное число. Так как заранее неизвестна размерность решаемой системы, то следует предусмотреть возможность формирования векторов переменных размеров.

Указанным требованиям соответствует класс Vector, который разработан в теме 10 и размещён в библиотеке Library.

Приведём его UML-изображение (см. рис. 13.13) и напомним особенности класса Vector.

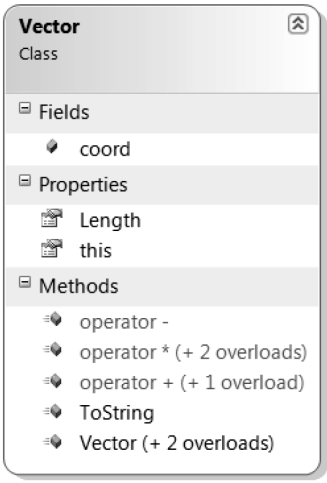


Рис. 13.13. Структура класса многомерных векторов

В этом классе только одно поле **double**[] coord для представления координат вектора в виде массива. Размер массива определяет размерность пространства векторов – объектов класса. Три конструктора позволяют по-разному определять объекты класса векторов. В классе определены методы для перегрузки операций сложения, вычитания и умножения векторов, а также умножения вектора на число и числа на вектор. Индексатор обеспечивает доступ к элементам вектора, а свойство **Length** позволяет получать размер вектора. Отметим, что для сокращения кода опущены операторы проверки корректности значений аргументов в методах класса.

В той же библиотеке Library определён класс Methods, в который поместим статические методы, реализующие алгоритм Рунге-Кутта. Кроме того в библиотеку Library включим делегат для представления методов, описывающих правые части систем дифференциальных уравнений:

```
public delegate Vector Proc(double x, Vector y);
```

Для численного интегрирования системы дифференциальных уравнений удобно использовать два метода. Первый из них

выполняет вычисления, относящиеся к одному шагу изменения переменной интегрирования:

```
// Один шаг интегрирования СДУ методом РК_4:
public static void RK_4_step(
   double xn,        // начальное значение независимой переменной
   double hx,        // шаг интегрирования
   Vector yn,        // начальные значения искомых функций
   out Vector yk,    // значения искомых функций в момент xn+hx
   Proc fun          // процедура вычисления (производных)
   )  {
    Vector k1, k2, k3, k4;
    k1 = fun(xn, yn);
    k2 = fun(xn + hx / 2, yn + hx / 2 * k1);
    k3 = fun(xn + hx / 2, yn + hx / 2 * k2);
    k4 = fun(xn + hx, yn + hx * k3);
    yk = yn + hx / 6 * (k1 + 2 * k2 + 2 * k3 + k4);
}  // RK_4_step()
```

Обратите внимание, что один из параметров метода RK_4_step() имеет тип делегата Proc и предназначен для представления метода, вычисляющего правые части интегрируемой системы дифференциальных уравнений.

Второй метод solveDif() предназначен для решения системы дифференциальных уравнений на заданном интервале изменения независимой переменной с фиксированным шагом ее изменения.

```
// Решение уравнений с шагом dt на интервале tn-tk.
// Результат - массив векторов res[], массив time[].
public static Vector[] solveDif(
    double tn, double tk, double dt, //
    Vector y0,          // начальные условия
    Proc fun,           // правые части уравнений
    out double [] time     // значения переменной интегрирования
    )  {
    // Размер массивов результатов решения:
    intsize = (int)((tk - tn) / dt + 1);
    Vector[] res = new Vector[size];
    res[0] = new Vector(y0);
    time = new double[size];
    time[0] = tn;
    double t = tn;
    for (int m = 1; m < res.Length; m++) {
        Methods.RK_4_step(t, dt, res[m - 1], out res[m], fun);
```

```
        t += dt;
        time[m] = t;
        }
    return res;
}  // solveDif()
```

Оттранслировав библиотеку классов и убедившись в отсутствии синтаксических ошибок, перейдем к следующему шагу проектирования.

Шаг 6. Интегрирование системы дифференциальных уравнений

Включим в код Form1.cs оператор

```
using Library;
```

и добавим в число ссылок ссылку на проект Library.

Теперь можно программировать численное решение уравнения

$$x''(t) + \frac{k}{m}x(t) = 0.$$

Но предварительно нужно привести уравнение к форме Коши, введя дополнительные переменные x_0 и x_1:

$$x_0' = x_1$$

$$x_1' = -\frac{k}{m}x_0$$

Начальные условия: $x_1(t_0) = x'(t_0)$; $x(t_0)$.

Дополним метод button1_Click() кодом, цель которого – получить численное решение приведенной системы дифференциальных уравнений, отобразить полученное решение в виде графика вместе с графиком аналитического решения, вычислить среднеквадратичное отклонение аналитического решения от численного.

Для получения решения, воспользуемся методом solveDif из нашей библиотеки классов.

Вызов метода будет выглядеть следующим образом:

```
// Вызов метода:
Vector[ ] result = Library.Methods.solveDif(tn, tk, dt,
      new Vector(new double[ ] { x0, x10 }),
      // Анонимный метод как аргумент:
      delegate(double t, Vector v)     {
          Vector res = new Vector(v.Length);
          res[0] = v[1];
          res[1] = -k / m * v[0];
          return res;
          },
       out trk
  );
```

Аргументы tn, tk, dt определяют начало, окончание и шаг интегрирования. Безымянный аргумент-вектор передает в метод начальное состояние интегрируемой системы. Правые части системы дифференциальных уравнений заданы анонимным методом с типом делегата Proc. Первый элемент массива res[0] определяет значение x(t), а второй, res[1] – координату x1(t). Результат выполнения метода solveDif () – массив векторов (представляемый ссылкой result) и массив значений переменной интегрирования (ссылка trk).

Полученные массивы позволяют построить график зависимости x(t). Для этого в методе button1_Click() создадим новый экземпляр класса PointPairList (ссылка listRK), заполняем его значениями из полученных массивов и добавляем на график соответствующую кривую:

```
// Создадим список для точек графика RK:
PointPairList listRK = new PointPairList();
// Заполняем  список  точек RK:
    for (int i = 0; i < xt.Length; i++)
// добавим в список координаты точки RK:
listRK.Add(trk[i], result[i][0]);
// Создадим кривую с названием "Численное решение":
LineItem myCurveRK = pane.AddCurve("Численное решение",
            listRK, Color.Red, SymbolType.Circle);
```

После этих дополнений результаты выполнения программы показаны на рис 13.14.

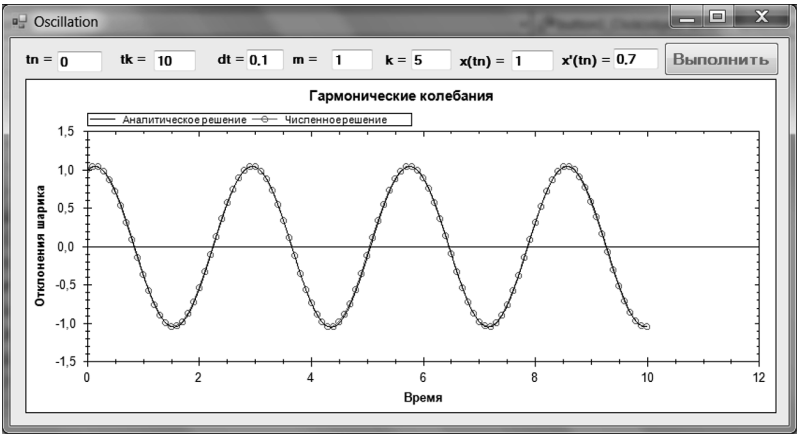


Рис. 13.14. Объединение графиков численного и аналитического решений

Шаг. 7. Оценка точности решения

Так как в нашей программе в одних и тех же точках (при одинаковых значениях переменной интегрирования) значения x(t) вычисляются и аналитически, и численно, то можно оценить погрешность численного вычисления. Для этого в метод button1_Click() добавим следующий фрагмент кода:

```
// Подсчитаем среднеквадратичное отклонение:
   double cko = 0;
   for (int i = 0; i < xt.Length; i++) {
      double del = xt[i] - result[i][0];
      cko += del * del;
      }
   double norm = Math.Sqrt(cko) / xt.Length;
// Напишем текст со значением СКО.
// Координаты размещения текста «привязаны» к осям.
TextObj text = new TextObj("СКО = " +
      norm.ToString("g5"), tk, 0.5);
// Отключим рамку вокруг текста
text.FontSpec.Border.IsVisible = false;
// Удалим предшествующую запись:
   pane.GraphObjList.Clear();
// Добавим текст в список отображаемых объектов
   pane.GraphObjList.Add(text);
```

Данный код кроме вычисления значения среднеквадратичного отклонения выполняет вывод его значения в поле элемента ZedGraph. Результаты выполнения программы, оценивающей точность численного интегрирования, приведены на рис. 13.15.

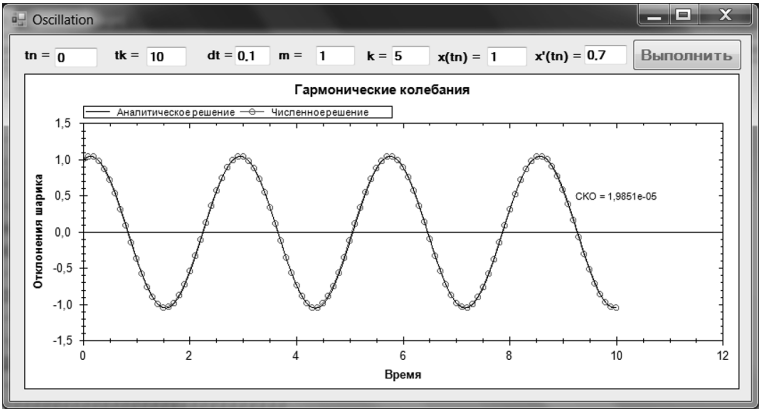


Рис. 13.15. Результаты интегрирования с оценкой погрешности вычислений

Приведем в заключение полный код файла Form1.cs проекта Oscilation.

```
// График аналитического решения уравнения
using System;
using System.Drawing;
using ZedGraph;
using System.Windows.Forms;
using Library;
namespace Oscillation {
public partial class Form1:Form {
public Form1( ) {
    InitializeComponent( );
    }
// Исходные данные для интегрирования:
double tn, tk, dt, m, k, x0, x10;
int size;  // Размер массивов результатов решения
double[] xt, time;
// Заглушка для метода чтения данных:
//bool readData()  {
//    tn = 0; tk = 10; dt = 0.1; m = 1;
```

```
//       k = 5; x0 = 1; x10 = 0.7;
//     size = (int)((tk - tn) / dt + 1);
//     xt = new double[size];
//     time = new double[size];
//     return true;
//   }

// Метод чтения и проверки данных:
bool readData() {
     if (Methods.readDouble(textBox1.Text, out tn) ==
     false || tn < 0)    {
     MessageBox.Show("Ошибка в значении tn");
     textBox1.Focus();
     return false;
     }
if (Methods.readDouble(textBox2.Text, out tk) ==
false || tk <= tn)   {
          MessageBox.Show("Ошибка в значении tk или tn");
          textBox2.Focus();
          return false;
          }
if (Methods.readDouble(textBox3.Text, out dt) ==
              false || dt <= 0 || dt > tk - tn)  {
    MessageBox.Show("Ошибка в значениях dt или tn или tk");
    textBox3.Focus();
    return false;
    }
if (Methods.readDouble(textBox4.Text, out m) ==
    false || m <= 0)   {
    MessageBox.Show("Ошибка в значении m");
    textBox4.Focus();
    return false;
    }
if (Methods.readDouble(textBox5.Text, out k) ==
         false || k <= 0)   {
         MessageBox.Show("Ошибка в значении k");
         textBox5.Focus();
         return false;
         }
if (Methods.readDouble(textBox6.Text, out x0) == false) {
         MessageBox.Show("Ошибка в значении x0");
         textBox6.Focus();
         return false;
         }
```

```
if (Methods.readDouble(textBox7.Text, out x10) == false) {
        MessageBox.Show("Ошибка в значении x10");
        textBox7.Focus();
        return false;
        }
    size = (int)((tk - tn) / dt + 1);
    xt = new double[size];
    time = new double[size];
    return true;
} // readData()

void analytical_Solution() {
    double t = tn, sq = Math.Sqrt(k/m);
    double A = Math.Sqrt(x0*x0 + (x10/sq)*(x10/sq));
    double b = 0;
    if (x0 == 0 & x10 > 0)  b = 0;
    if (x0 == 0 & x10 < 0)  b = Math.PI;
    if (x0 != 0 & x10 == 0) b = Math.PI - Math.Atan(x0 * sq / x10);
    if (x0 != 0 & x10 > 0) b = Math.Atan(x0 * sq / x10);
    if (x0 != 0 & x10 < 0) b =  Math.PI + Math.Atan(x0 * sq / x10);
    for (int i = 0; i < size; i++)  {
        time[i] = t;
       xt[i] = A * Math.Sin(sq*t +b);
       t + = dt;
       }
}  // analytical_Solution

private void button1_Click(object sender, System.EventArgs e) {
// Прочитать и проверить введенные пользователем данные:
   if(!readData()) return;
// Аналитическое решение уравнения (получить xt[] и time[]):
   analytical_Solution();
// Показать элемент zedGraph:
   zedGraphControl1.Visible = true;
// Создать панель для рисования:
   GraphPane pane = zedGraphControl1.GraphPane;
// Очистить список прошлых кривых:
   pane.CurveList.Clear();
// Создать список для точек графика:
   PointPairList list = new PointPairList();
// Заполнить список точек:
   for (int i = 0; i < xt.Length; i++)
// Добавить в список одной координаты точки:
       list.Add(time[i], xt[i]);
```

```
// Создать кривую с названием "Аналитическое решение":
   LineItem myCurve = pane.AddCurve("Аналитическое решение",
        list, Color.Black, SymbolType.None);
// Включить "сглаживание" изображаемой кривой:
   myCurve.Line.IsSmooth = true;

        //////Численное решение системы уравнений:
    double [] trk = null; // Точки оси абсцисс
// Вызов метода:
   Vector[ ] result = Library.Methods.solveDif(tn, tk, dt,
        new Vector(new double[ ] { x0, x10 }),
// Анонимный метод как аргумент:
   delegate(double t, Vector v)      {
         Vector res = new Vector(v.Length);
         res[0] = v[1];
         res[1] = -k / m * v[0];
         return res;
         }, out trk
  );
// Создадим список для точек графика RK:
   PointPairList listRK = new PointPairList();
// Заполняем список точек RK:
    for (int i = 0; i < xt.Length; i++)
// добавим в список координаты точки RK:
   listRK.Add(trk[i], result[i][0]);
// Создадим кривую с названием "Численное решение":
   LineItem myCurveRK = pane.AddCurve("Численное решение",
                listRK, Color.Red, SymbolType.Circle);
//////////////////////
// Подсчитаем среднеквадратичное отклонение:
   double cko = 0;
   for (int i = 0; i < xt.Length; i++) {
      double del = xt[i] - result[i][0];
      cko += del * del;
      }
   double norm = Math.Sqrt(cko) / xt.Length;
// Напишем текст со значением СКО.
// Координаты размещения текста "привязаны" к осям.
   TextObj text = new TextObj("СКО = " +
   norm.ToString("g5"), tk, 0.5);
// Отключим рамку вокруг текста
   text.FontSpec.Border.IsVisible = false;
// Удалим предшествующую запись:
    pane.GraphObjList.Clear();
```

```
// Добавим текст в список отображаемых объектов
   pane.GraphObjList.Add(text);
            ////////////////////////
    // Общий заголовок графика:
    pane.Title.Text = "Гармонические колебания";
// Текст подписей для осей x и y:
    pane.XAxis.Title.Text = "Время";
    pane.YAxis.Title.Text = "Отклонения шарика";
// Обновить график:
    zedGraphControl1.AxisChange();
    zedGraphControl1.Invalidate();
} // button1_Click
} // class Form1
} // Oscillation
```

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

Форматирование данных при создании строк

Строка форматирования метода **string**.Format() включает неизменяемые символы и конструкции, называемые полями подстановок. Структура поля подстановки: {N[,W][:S[R]]}, где N – номер аргумента; W - ширина поля; S - спецификатор формата; R – спецификатор точности. Квадратные скобки в поля подстановки не входят и обозначают необязательность ограниченного ими фрагмента (элемента) поля. В поле подстановки разрешено опускать все элементы, кроме фигурных скобок и номера аргумента. Именно поэтому все остальные элементы поля подстановки мы ограничили квадратными скобками.

Форматирование для чисел

Спецификатор формата S задаёт вид изображаемого значения. Для разных типов данных этот спецификатор выбирается по-разному. Следующая за ним цифра – спецификатор точности R – влияет на формируемое значение, и это влияние зависит от спецификатора формата.

Таблица П.1.1

Спецификаторы формата S и точности R

Спецификатор S	Формат	Роль спецификатора точности R
C или c	валютный	Количество десятичных разрядов
D или d	целочисленный	Минимальное число цифр
E или e	экспоненциальный	Число разрядов после точки
F или f	с фиксированной точкой	Число разрядов после точки
G или g	короткий из E или F	Подобен E или F
X или x	шестнадцатеричный	Минимальное число цифр
N или n	с разделителями триад	Количество знаков после точки
P или p	процентов	Количество знаков после точки
R или r	обратимое преобразование	

Напомним, что в поле подстановки может кроме номера аргумента использоваться ширина поля (W). Тем самым можно по-разному размещать символы выводимого строкового значения. Если значение W отрицательно, то выводимая строка (изображение значения аргумента) выравнивается по левому краю поля.

Таблица П.1.2

Пользовательские спецификаторы формата

Спецификатор	Смысл	Примеры
0	Любая цифра и нуль	5.6 и “00.00” => 05.60 .27 и “0.0” => 0.3 26.5 и “00” => 27
#	Любая цифра	26.5 и “##” => 27 5.6 и “##.##” => 5.6
.	Десятичная точка	См. другие примеры
,	Разделитель триад	987654321 и “#,#” => 9,876,544,321
%	Процент	0.005432 и “#0.##%” => 0,54%
E0 или e0 E-0 или e-0 E+0 или e+0	Экспонентная нотация	1234500 и «0.##E-00» => 1,23E06 1234500 и «0.##E00» => 1,23E06 1234500 и “0.##E+00” => 1,23E+06
\	Начало эскейп - последовательности	\n или \t
‘ABC’ «ABC»	Литеральная подстановка строки	30 и “ 0 ‘ руб.’” => 30 руб.
;	Разделитель секций в форматной строке	30 и «0;[0]» => 30 (-30) и «0;[0]» => [30]

ПРИЛОЖЕНИЕ 2

Константы и методы класса Math

Для работы с числовыми данными предназначены методы и константы класса Math, из пространства имен System. Методы и константы статические, поэтому их применение не требует создания объектов класса.

Константы определяют приближенные значения чисел π и e

public const double PI=3.14159265358979323846
public const double E=2.71828182845904523536

Методы класса Math представляют тригонометрические и алгебраические функции. В следующей таблице представлены наиболее употребительные из этих методов.

Таблица П.2.1

Некоторые методы класса Math

Метод	Назначение (результат)
mun Abs(*mun* x)	Абсолютное значение аргумента
double Acos(**double** x)	Арккосинус значения аргумента
double Asin(**double** x)	Арксинус значения аргумента
double Atan(**double** x)	Арктангенс значения аргумента
double Atan2(**double** y, **double***x)*	Арктангенс значения y/x
double Ceiling(**double***x*)	Наименьшее целое не меньшее аргумента
double Cos(**double** x)	Косинус значения аргумента
double Cosh(**double** x)	Гиперболический косинус значения аргумента
double Exp(**double** x)	Число e в степени значения аргумента
double Floor(**double***x*)	Наибольшее целое не большее аргумента
double Log(**double** x)	Натуральный логарифм значения аргумента
double Log10(**double** x)	Десятичный логарифм значения аргумента
mun Max(*mun* x1, *mun* x2)	Максимальное из значений аргументов

Продолжение таблицы П.2.1

Метод	Назначение (результат)
mun Min(*mun* x1, *mun* x2)	Минимальное из значений аргументов
double Pow (**double** y, **double** x)	y в степени x
double Round(**double***x*)	Ближайшее к значению аргумента целое
int Sign(*mun*x)	Знак аргумента (1 для +; -1 для -; 0 для нуля)
double Sin(**double** x)	Синус значения аргумента
double Sinh(**double** x)	Гиперболический синус значения аргумента
double Sqrt (**double** x)	Квадратный корень из значения аргумента
double Tan(**double** x)	Тангенс значения аргумента
double Tanh(**double** x)	Гиперболический тангенс значения аргумента
double Truncate(**double***x*)	Целая часть значения аргумента

Методы, параметры которых специфицированы условным обозначением «*mun*», перегружены и существуют в нескольких вариантах, каждый для конкретного арифметического типа.

Значения аргументов тригонометрических функций и возвращаемые значения обратных тригонометрических функций задаются и получаются в радианах.

ПРИЛОЖЕНИЕ 3

Событие, свойства и методы класса Console

При изучении языка C# вначале проще разрабатывать не приложения с изысканным графическим интерфейсом, а создавать консольные программы, для которых средством общения пользователя с исполняемой программой служат стандартные потоки ввода/вывода. Для представления этих потоков в программе на языке C# используется статический класс System.Console. У этого класса есть одно событие, а также набор методов и свойств, дающих возможность работать с клавиатурой и управлять характеристиками консольного окна.

Событие **CancelKeyPress** класса **Console**:
public static event ConsoleCancelEventHandler CancelKeyPress

Это событие возникает (происходит), когда в процессе выполнения консольной программы пользователь нажимает одновременно две клавиши: «CTRL+C».

Таблица П.3.1

Некоторые методы класса Console

Имя метода	Назначение (результат)
Beep	Звуковой сигнал консольного динамика
Clear	Очищает буфер консоли и консольное окно
MoveBufferArea	Копирует определенную исходную область буфера экрана в конкретную область памяти
OpenStandardError	Открывает стандартный поток ошибок
OpenStandardInput	Открывает стандартный входной поток
OpenStandardOutput	Открывает стандартный выходной поток
Read	Читает очередной символ из стандартного входного потока
ReadKey	Получает код нажатой символьной или функциональной клавиши в виде объекта структуры ConsoleKeyInfo
ReadLine	Читает очередную символьную строку из стандартного входного потока

Продолжение таблицы П.3.1

Имя метода	Назначение (результат)
ResetColor	Восстанавливает в консольном окне умалчиваемые значения цветов фона и переднего плана
SetBufferSize	Задает высоту и ширину области буфера дисплея
SetCursorPosition	Устанавливает позицию курсора
SetError	Связывает свойство Error с указанным объектом типа TextWriter
SetIn	Связывает свойство In с указанным объектом типа TextWriter
SetOut	Связывает свойство Out с указанным объектом типа TextWriter
SetWindowPosition	Устанавливает позицию консольного окна относительно буфера дисплея
SetWimdowSize	Задает высоту и ширину консольного окна
Write	Выводит в выходной поток последовательность символов (текст)
WriteLine	Выводит в выходной поток последовательность символов и код перехода на следующую строку

Таблица П.3.2

Некоторые свойства класса Console

Имя свойств	Назначение (описание)
BackgroundColor	Цвет фона консольного окна
BufferHeight	Высота буферной области
BufferWidth	Ширина буферной области
CapsLock	Признак включения/отключения режима CAPSLOCK
CursorLeft	Номер колонки (столбца) позиции курсора в буферной области
CursorSize	Высота курсора в символьной позиции
CursorTop	Номер строки позиции курсора в буферной области
CursorVisible	Индикатор видимости курсора
Error	Стандартный выходной поток ошибок
ForegroundColor	Цвет переднего плана консоли

Продолжение таблицы П.3.2

Имя свойств	Назначение (описание)
In	Стандартный входной поток
InputEncoding	Кодировка консоли, применяемая при вводе (чтении)
KeyAvailable	Индикатор доступности клавиши во входном потоке
LargestWindowHeight	Максимальное количество строк консольного окна
LargestWindowWidth	Максимальное количество столбцов консольного окна
NumberLock	Признак включения/отключения режима NUMLOCK
Out	Стандартный выходной поток
OutputEncoding	Кодировка консоли, применяемая при выводе
Title	Текст заголовка консольного окна
TreatControlCAsInput	Индикатор действия (восприятия) сочетания клавиш CTRL+C. Сочетание может восприниматься либо как обычные входные данные либо как прерывание, обрабатываемое операционной системой
WindowHeight	Высота области консольного окна
WindowLeft	Левая позиция консольного окна относительно буфера дисплея
WindowTop	Верхняя позиция консольного окна относительно буфера дисплея
WindowWidth	Ширина области консольного окна

Структура struct ConsoleKeyInfo

Объект этой структуры представляет информацию о нажатой в процессе выполнения программы клавише. Кроме того, в нем же содержится информация о состоянии функциональных клавиш SHIFT, ALT, CTRL. Объект содержит константу перечисления **ConsoleKey** и значение юникода символа, представляемого клавишей. Эта структура не входит в консольный класс, но ссылку на ее объект возвращает метод ReadKey(), поэтому программисту полезно ознакомиться с членами этой структуры.

Таблица П.3.3

Свойства структуры ConsoleKeyInfo

Имя свойства	Назначение (описание)
Key	Значение типа перечисления **ConsoleKey** для клавиши, представленной объектом **ConsoleKeyInfo**
KeyChar	Юникод символа для клавиши, представленной объектом **ConsoleKeyInfo**
Modifiers	Поразрядное представление модификаторов, соответствующих функциональным клавишам, нажатым одновременно с клавишей, представленной объектом **ConsoleKeyInfo**

Приводить список констант перечисления **ConsoleKey** нет необходимости – их всегда можно посмотреть, воспользовавшись справочной системой. В нашей книге мы использовали только две константы этого перечисления:

ConsoleKey.Enter – соответствует клавише ENTER,

ConsoleKey.Esc – соотнесена с клавишей ESCAPE.

При использовании методов Console.BackGroundColor() и ForegroundColor() обязательно требуются сведения о наборе констант перечисления **Console.Color.**

Таблица П.3.4

Константы перечисления public enum Console.Color

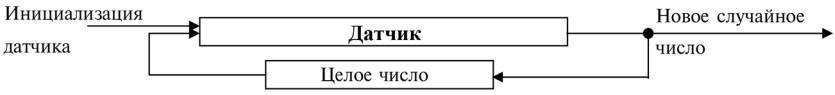
Имя константы	Описание (название цвета)
Black	Черный
DarkBlue	Темно-синий
DarkGreen	Темно-зеленый
DarkCyan	Темно-голубой (темно зелено-синий)
DarkRed	Темно-красный
DarkMagenta	Темно-багряный (темный фуксин)
DarkYellow	Темно-желтый (охра)
Gray	Серый
DarkGray	Темно-серый
Blue	Синий
Green	Зеленый
Cyan	Голубой (сине-зеленый)
Red	Красный
Magenta	Фуксин (красно-багрянистый)
Yellow	Желтый
White	Белый

ПРИЛОЖЕНИЕ 4

Генерация случайных чисел

Средства генерации случайных чисел определены в классе **Random** (пространство имен **System**). Для создания источника случайных чисел необходимо объявить ссылку на объект этого класса и создать этот объект с помощью конструктора, используя операцию **new**. Этот объект и будет играть роль датчика случайных чисел.

Алгоритм датчика построен так, что очередное случайное число вырабатывается на основе значения предыдущего случайного числа:



Самое первое «стартовое» число в датчик заносится при инициализации объекта одним из двух способов:

Random ran = **new** Random(start);

Random ran = **new** Random();

В первом случае в объект, представляемый ссылкой ran, заносится стартовое число start, поэтому при повторных запусках программы датчик всегда будет выдавать одну и ту же последовательность случайных чисел. Во втором случае объект, представляемый ссылкой ran, инициализируется текущим значением системного времени, поэтому при повторных запусках программы датчик будет выдавать разные последовательности случайных чисел.

Для получения случайного числа X предназначены методы объекта – датчика (случайных чисел):

int Next()	$0 \le X < $ **int**.MaxValue
int Next (**int**B)	$0 \le X < B$
int Next(**int** A, **int** B)	$A \le X < B$
double NextDouble ()	$0 \le X < 1$

Для получения случайного вещественного числа X из диапазона $A \le X < B$, пользуются выражением: X = A + (B-A)*NextDouble().

Примеры.

Получение случайного целого числа из диапазона от 10 до 100 включительно:

```
int  x1;
Random ran = new Random();
x1 = ran.Next(10, 101);
```

Случайное вещественного числа из диапазона от -3 до +3 (+3 не входит в диапазон):

```
double z2 = -3 + (3 + 3) * NextDouble().
```

ПРИЛОЖЕНИЕ 5

Члены классов Array и String

Таблица П.5.1

Наследуемые члены класса Array

Член (Member)	Тип (Type)	Время жизни (Lifetime)	Смысл (Meaning)
Rank	Свойство	Экземпляр	Количество измерений массива
Length	Свойство	Экземпляр	Общее количество элементов по всем измерениям массива
GetLength	Метод	Экземпляр	Длина конкретного измерения массива
Clear	Метод	Класс	Обнуляет число элементов (0 или **null**)
Copy	Метод	Класс	Поверхностное копирование массива или части его элементов
Sort	Метод	Класс	Сортирует элементы одномерного массива
BinarySearch	Метод	Класс	Бинарный поиск значения в одномерном массиве
Clone	Метод	Экземпляр	Поверхностное копирование массива
IndexOf	Метод	Класс	Возвращает индекс первого появления заданного значения в одномерном массиве
Reverse	Метод	Класс	Изменяет на обратный порядок элементов в одномерном массиве
GetUpperBound	Метод	Экземпляр	Верхняя граница конкретного измерения массива

Таблица П.5.2

Члены класса String

Член (Member)	Тип (Type)	Смысл (Meaning)
Length	Свойство	Длина строк
CompareTo	Метод объекта	Сравнивает две строки
Concat	Метод класса	Создаёт строку как конкатенацию строк-аргументов
Contains	Метод объекта	Возвращает логическое значение, указывающее является ли строка-аргумент подстрокой вызывающей строки
Copy	Метод объекта	Возвращает копию существующей строки
Format	Метод класса	Возвращает отформатированную строку
IndexOf	Метод объекта	Поиск в вызывающей строке подстроки
Insert	Метод объекта	Вставляет строку-аргумент в заданную позицию вызывающей строки
Join	Метод объекта	Объединяет строки массива–параметра
LastIndexOf	Метод объекта	Поиск в вызывающей строке подстроки
Remove	Метод объекта	Удаляет набор символов из вызывающей строки
Replace	Метод объекта	Заменяет символ или подстроку в вызывающей строке
Split	Метод объекта	Формирует массив строк из фрагментов вызывающей строки
Substring	Метод объекта	Выделяет подстроку из вызывающей строки
ToCharArray	Метод объекта	Копирует символы вызывающей строки в массив типа **char**[].
ToLower	Метод объекта	Возвращает копию вызывающей строки, где символы преобразованы к нижнему регистру
ToUpper	Метод объекта	Возвращает копию вызывающей строки, где символы преобразованы к верхнему регистру
Trim	Метод объекта	Удаляет вхождение заданных символов в начале и в конце строки.

ПРИЛОЖЕНИЕ 6

Общие методы списков

Таблица П.6.1

Методы списков List<T> и ArrayList

Член (Member)	Смысл (Meaning)
Add	Добавить объект в конец списка
AddRange	Добавить в конец списка элементы конкретной коллекции
BinarySearch	Найти в списке место размещения элемента с заданным значением. Метод перегружен. В нем используется алгоритм бинарного поиска
Clear	Удалить из списка все элементы
Contains	Выяснить, есть ли в списке элемент с заданным значением
CopyTo	Копировать в подходящий одномерный массив либо список, либо часть его элементов. Метод перегружен. Можно указывать позицию записи в массиве
GetEnumerator	Вернуть нумератор, обеспечивающий перебор элементов списка
GetRange	Создать поверхностную копию диапазона элементов из списка
IndexOf	Возвращает индекс первого элемента с заданным значением. Метод перегружен. Поиск выполняется или в списке, или в диапазоне его элементов
Insert	Вставить новый элемент в заданную индексом позицию списка
InsertRange	Вставить последовательность элементов в заданную индексом позицию списка
LastIndexOf	Возвращает индекс последнего элемента с заданным значением. Метод перегружен. Поиск выполняется или в списке, или в диапазоне его элементов.
Remove	Удаляет из списка первый элемент с заданным значением.
RemoveAt	Удаляет из списка элемент с заданным индексом
RemoveRange	Удаляет из списка элементы заданного диапазона

Продолжение таблицы П.6.1

Член (Member)	Смысл (Meaning)
Reverse	Меняет на обратный порядок элементов списка или его части. Метод перегружен.
Sort	Сортирует элементы списка или его части. Метод перегружен.
ToArray	Копирует элементы списка в новый массив.
ToString	Возвращает строку, которая представляет текущий объект. (Унаследован от класса Object.)

Реестр программ

Тема 01. Среда разработки и консольные приложения
// 01_01 – Первая программа, проект Program_1
// 01_02 – Программа с диалогом, проект Program_2
// 01_03 – Программа с диалогом, проект Program_3

Тема 02. Консольный вывод
// 02_01 – Суффиксы арифметических констант
// 02_02 – Изображения и значения выражений
// 02_03 – Форматы записи вещественного числа
// 02_04 – Форматирование изображений вещественных чисел
// 02_05 – Форматирование при выводе чисел
// 02_06 – Таблица спецификаторов формата
// 02_07 – Коды русских и латинских букв

Тема 03. Консольный ввод
// 03_01 – Методы Parse()
// 03_01_1 – Методы класса Convert
// 03_02 – «Чтение» числа из строки
// 03_03 – «Чтение» числа из стандартного входного потока
// 03_04 – Синтаксический контроль входной строки
// 03_05 – «Чтение» символов из входного потока
// 03_06 – «Чтение» символов из строки. Коды символов.

Тема 04. Управление консолью
// 04_01 – Нажимая клавиши, наблюдайте за значениями на экране
// 04_02 – Изучение свойств класса Console.
// 04_03 – Свойства класса Console и перечисление ConsoleColor.
// 04_04 – Нажимая клавиши, наблюдайте за размерами экрана
// 04_05 – Циклическое изменение цвета фона окна
// 04_05_1 – Циклическое изменение цвета фона окна
// 04_06 – Изучение возможностей метода Beep()
// 04_07 – Изучение возможностей метода Beep()
// 04_08 – Воспроизведение звука сирены

Тема 05. Выражения, условные операторы, циклы
// 05_01 – Анализ суммы первых членов натурального ряда
// 05_02 – Формула Бине для чисел Фибоначчи
// 05_03 – Упорядочение значений трех переменных
// 05_04 – Анализ числовых значений кодов символов
//05_05 – Сложные проценты
//05_06 – Вычисление значения функции sin(x)
//05_07 – Вычисление значения числа пи
//05_08 – Вывести таблицу истинности логической функции

Тема 06. Массивы, строки, переключатели
// 06_01 – Нормирование элементов массива
// 06_02 – Поиск и изменение элемента в массиве
// 06_03 – Массив простых чисел
// 06_04 – «Вырезка» из массива элементов от макс. до мин.
// 06_05 – Выбор из массива согласных русских букв
// 06_06 – Сортировка массива с помощью косвенной индексации
// 06_07 – Добавление в одномерный массив новых элементов
// 06_08 – Принудительное изменение размеров матрицы
// 06_09 – Упорядочить цифры (символы) натурального числа
// 06_10 – Сопротивление электрической цепи
// 06_11 – Запись числа с основанием 16
// 06_12 – Массив символьных массивов («зубчатый» массив)
// 06_13 – Обратный порядок слов предложения

Тема 07. Статические методы (методы классов)
// 07_01 – Вычисление наибольшего общего делителя
// 07_02 – Сортировка значений скалярных переменных
// 07_03 – Методы для вычисления sin x
// 07_04 – Методы для построения треугольника Паскаля
// 07_05 – Ввод вещественного числа (с точкой либо с запятой)
// 07_06 – Рекурсивное формирование массива
// 07_07 – Сортировка строк матрицы в виде массива массивов
// 07_08 – Массив индексов локальных минимумов

Тема 08. Методы в библиотеке классов
// Methods – класс методов в библиотеке Library
Методы библиотеки:
метод для вывода элементов целочисленного массива

метод для вывода элементов вещественного массива
печать целочисленной матрицы по строкам
печать вещественной матрицы по строкам
создание массива со случайными значениями элементов
создание массива со случайными значениями элементов
создание матрицы с целочисленными случайными элементами
создание матрицы с вещественными случайными элементами
метод для чтения вещественного числа из строки с учетом
американской (точка) и европейской (запятая) записей чисел
метод для ввода элементов целочисленного массива
метод для ввода элементов целочисленной матрицы
метод для ввода элементов вещественного массива
метод для ввода элементов вещественной матрицы
метод для «изображения» элементов целочисленного массива
метод для «изображения» элементов вещественного массива
метод для «изображения» элементов целочисленной матрицы
метод для «изображения» элементов вещественной матрицы
обобщенный метод для создания массива со случайными
значениями
обобщенный метод для вывода элементов массива
обобщенный метод для создания матрицы со случайными
элементами
обобщенный метод для печати матрицы по строкам:
обобщенный метод для «изображения» элементов массива
обобщенный метод для «изображения» матрицы
метод выполняет один шаг интегрирования СДУ методом РК_4
решение диф. уравнений с шагом dt на интервале [tn,tk]

Проекты темы 08:

08_01 – иллюстративная программа для работы с библиотекой
08_02 – программа с методами печати массивов
08_03 – программа с библиотечными методами
08_04 – тестирование библиотечного метода readDouble()
08_05 – ввод элементов целочисленных массивов
08_06 – ввод элементов вещественных массивов
08_07 – тестирование методов для работы с массивами

Тема 09. Классы и их объекты

// 09_01 – Правильный многоугольник
// 09_02 – Класс «число и его шестнадцатеричные цифры»
// 09_03 – Массив ссылок на объекты класса «Окружность»

// 09_04 – Материальная точка в трехмерном пространстве
// 09_05 – Уклонение точки от прямой
// 09_06 – Класс динамических массивов

Тема 10. Перегрузка операций
// 10_01 – Операции с электрическими сопротивлениями
// 10_02 – Перегрузка операций в классе полиномов
// 10_03 – Перегрузка операции умножения матриц
// 10_04 – Перегрузка операций в классе рациональных дробей
// 10_05 – Операции над многомерными векторами

// Перегруженные операции класса многомерных векторов
Vector.cs:
операция поэлементного сложения векторов (+)
операция поэлементного вычитания векторов (–)
операция умножения числа на вектор (*)
операция умножения вектора на число (*)
операция поэлементного умножения векторов (*)
операция суммирования элементов вектора (+)

Тема 11. Наследование классов и интерфейсы
// Библиотека классов Figures:
класс «точка на плоскости»
класс «круг на плоскости»
класс «квадрат на плоскости»
абстрактный класс «габаритные размеры»
производный класс «эллипс»
производный класс «треугольник»
класс материальная точка – наследник класса Point
множество материальных точек (агрегация классов)

// 11_01 – Обработка массивов с разнотипными элементами
// 11_02 – Наследники абстрактного класса Dimensions
// 11_03 – Множество материальных точек на плоскости
// 11_04 – Классы стеков, реализовавших интерфейс ISteck

Тема 12. Программы с визуальным интерфейсом пользователя
// 12_01 – Члены ряда Пелла
// 12_02 – Редактируемый список в текстовом поле
// 12_03 – Сортировка цифр вводимого числа

// 12_04 – Программное изменение размеров формы
// 12_05 – Периметр правильного n-угольника
// 12_06 – Построение единичной матрицы
// 12_07 – Иллюстрация полета спутника
// 12_08 – График биномиального распределения

Тема 13. Эволюционный подход к проектированию программ
// 13_01 – Выделение кластеров в таблице на экране
// 13_02 – Интегрирование дифференциальных уравнений

Литература

1. C# 4.0. Language Specification. Version 4.0. : Microsoft Corporation. 2010. – 525 pp.
2. ECMA-334. C# Language Specification. 4th Edition / June 2006, – Geneva (ISO/IEC 23270:2006). – 553 pp.
3. Абрамова С.А., Гнездилова Г.Г. и др. Задачи по программированию. – М.: Наука, 1988. – 224 с.
4. Абрамян М.Э. Visual C# на примерах. – СПб.: БХВ-Петербург, 2008. – 496 с.: ил. + CD-ROM.
5. Албахари Дж. C# 5.0. Справочник: Полное описание языка: Пер. с англ. / Албахари Дж., Албахари Б. – М.: Издательский дом «Вильямс», 2013. – 1008 с.
6. Ватсон Б. C# 4.0 на примерах. – СПб.: БХВ-Петербург, 2011. – 608 с.
7. Вирт Н. Алгоритмы + структуры данных = программы: Пер. с англ. – М.: Мир, 1985. – 406 с.
8. Гросс К. C# 2008 и платформа NET 3.5 Framework: базовое руководство. – 2-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 480 с.
9. Дрейфус М., Ганглоф К. Практика программирования на Фортране: Пер. с франц. – М.: Изд-во «МИР», 1978. – 224 с.
10. Давыдов В.Г. Технологии программирования C++. – СПб.: БХВ-Петербург, 2005. – 272 с.
11. Зиборов В.В. Visual C#2010 на примерах. – СПб.: БХВ-Петербург, 2011. – 432 с.
12. Касьянов В.Н., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ. – М.: Наука, 1986. – 272 с.
13. Климов А.П. C#. Советы программистам. – СПб.: БХВ-Петербург, 2008. – 544 с.: ил. + CD-ROM.
14. Культин Н.Б. Microsoft Visual C# в задачах и примерах. – СПб.: БХВ-Петербург, 2009. – 320 с.: ил. + CD-ROM.

15. Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++. – СПб.: БХВ-Петербург, 2004. – 464 с.

16. Мак-Кракен Д., Дорн У. Численные методы и программирование на Фортране. – М.: Изд-во «МИР», 1969. – 582 с.

17. Морган М. Java 2. Руководство разработчика. – М.: Издательский дом «Вильямс», 2000. – 720 с.

18. Нейгел К. и др. C# 4.0 и платформа .NET 4.0 для профессионалов. – М.: ООО «И.Д. Вильямс», 2011. – 1440 с.

19. Нэш Т. C# 2008. Ускоренный курс для профессионалов. – М.: ООО Издательский дом «Вильямс», 2008. – 576 с.

20. Павловская Т.А. C#. Программирование на языке высокого уровня. Учебник для вузов. – СПб.: Питер. 2009. – 432 с.

21. Пауэрс Л., Снелл М. Microsoft Visual Studio 2008. – СПб.: БХВ-Петербург, 2009. – 1200 с.

22. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. – М.: Издательско-торговый дом «Русская Редакция», 2002. Том 1. – 624 с.; Том 2. – 576 с.

23. Подбельский В.В. Язык C# Базовый курс: учеб. пособие. – 2-е изд. – М.: Финансы и статистика, 2013. – 408 с.

24. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер класс. – 2-е изд. исправ. – М.: Русская Редакция; СПб.: Питер, 2008. – 656 с.

25. Скит Дж. C#: Программирование для профессионалов. – 2-изд. – М.: ООО «И.Д. Вильямс», 2011. – 544 с.

26. Тондо К., Гимпел С. Язык Си. Книга ответов: Пер. с англ. – М.: Финансы и статистика, 1994. – 160 с.

27. Троелсен Э. Язык программирования C# и платформа .NET 3.5. – М.: ООО «И.Д. Вильямс», 2011. – 1344 с.

28. Фаронов В.В. Создание приложений с помощью C#. Руководство программиста. – М.: Эксмо, 2008. – 576 с.

29. Фленов М.Е. Библия C#. – 2-е изд., перераб. и доп. – СПб.: БХВ-Петербург, 2011. – 560 с.: ил. + CD-ROM.

30. Фролов А.В., Фролов Г.В. Визуальное проектирование приложений C#. – М.: КУДИЦ-ОБРАЗ, 2003. – 512 с.

31. Шень А. Программирование: теоремы и задачи. – М.: МЦНМО, 1995. – 264 с.

32. Шилдт Г. C# 4.0. Полное руководство. – М.: Издательский дом «Вильямс», 2013. – 1056 с.

Содержание

Предисловие 3

Тема 01. Среда разработки и консольные приложения . . 11
Тема 02. Консольный вывод 29
Тема 03. Консольный ввод 38
Тема 04. Управление консолью 48
Тема 05. Выражения, условные операторы, циклы 61
Тема 06. Массивы, строки, переключатели 77
Тема 07. Статические методы (методы классов) 102
Тема 08. Методы в библиотеке классов 121
Тема 09. Классы и их объекты 140
Тема 10. Перегрузка операций 161
Тема 11. Наследование классов и интерфейсы 176
Тема 12. Программы с визуальным интерфейсом пользователя 198
Тема 13. Эволюционный подход к разработке программ . 236

Приложение 1. Форматирование данных при создании строк 273
Приложение 2. Константы и методы класса Math . . . 275
Приложение 3. Событие, свойства и методы класса Console 277
Приложение 4. Генерация случайных чисел 281
Приложение 5. Члены классов Array и String 283
Приложение 6. Общие методы списков 285

Реестр программ 287
Литература 292

Учебное издание

Подбельский Вадим Валериевич

Язык C#. Решение задач

Заведующая редакцией *Н.Ф. Карпычева*
Мл. редактор *Н.В. Зеленюк*
Компьютерная верстка *Е.И. Аникеева*

ИБ № 5449

Подписано в печать 21.05.2014. Формат 60 × 90 $^{1}/_{16}$.
Гарнитура «Таймс». Печать офсетная.
Усл. п. л. 18,5. Уч.-изд. л. 18,2. Тираж 300 экз.
Заказ № «С» 106.

Издательство «Финансы и статистика»
101000, Москва, ул. Покровка, 7
Телефон (495) 625-35-02, 625-47-08. Факс (495) 625-09-57
E-mail: mail@finstat.ru http://www.finstat.ru