КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт Вычислительной Математики и Информационных Технологий

АНДРИАНОВА А.А.,
ИСМАГИЛОВ Л.Н.,
МУХТАРОВА Т.М.

Объектно-ориентированное программирование на C#

КАЗАНЬ - 2012

УДК 004.43

ББК 32.973.26 – 018.1

Печатается по постановлению редакционно-издательского совета Института вычислительной математики и информационных технологий Казанского (Приволжского) федерального университета

Рецензенты:

кандидат технических наук, доцент кафедры систем информационной безопасности КНИТУ им. А.Н. Туполева **Л.А. Александрова**

старший преподаватель кафедры системного анализа и информационных технологий Казанского (Приволжского) федерального университета **Р.Р. Тагиров**

Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М.
Объектно-ориентированное программирование на C# : Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжский) федеральный университет, 2012. – 140 с.

Учебное пособие посвящено принципам разработки программ на языке программирования C#, использующим объектно-ориентированный подход, и предназначено для ведения практических занятий на 1, 2 курсах направлений «Прикладная информатика», «Прикладная математика и информатика», «Бизнес-информатика», а также может использоваться студентами других специальностей.

Оглавление

Введение

Учебное пособие посвящено принципам разработки программ на языке C#, использующим объектно-ориентированный подход. В настоящее время этот подход является основным для различных языков программирования. Объектно-ориентированное программирование отличается от процедурного программирования в первую очередь тем, что при проектировании основной акцент ставится на разработку структур хранения и управления данными, а не на алгоритмы, реализованные в программе. Согласно объектно-ориентированному подходу любая программа представляет собой набор взаимодействующих друг с другом объектов, имеющих состояние и поведение. Разработка программы сводится к определению этого набора объектов.

Язык программирования C# в настоящее время является одним из наиболее распространенных средств разработки в рамках объектно-ориентированного подхода. Он был разработан в 1998—2001 годах группой инженеров под руководством Андерса Хейлсберга в компании *Microsoft* как основной язык разработки приложений для платформы *Microsoft .NET Framework*.

.NET Framework – это программная платформа, выпущенная компанией *Microsoft* в 2002 году, основой которой является виртуальная машина *Common Language Runtime* (*CLR*), способная выполнять как обычные настольные программы, так и веб-приложения. Отличительной особенностью *.NET Framework* является способность выполнять программы, написанные на разных языках программирования (C#, C++, Visual Basic и др.). Основой платформы *.NET Framework* является набор библиотек классов, которыми можно пользоваться при разработке приложений на всех поддерживаемых языках.

Данное учебное пособие предназначено для студентов и преподавателей Института Вычислительной математики и информационных технологий при изучении принципов объектно-ориентированного программирования в рамках различных учебных дисциплин.

Учебное пособие состоит из семи глав. В первой главе вводится понятие объектной модели приложения. Вторая глава посвящена краткой характеристике принципов объектно-ориентированного программирования и их использованию при проектировании программ – абстрагированию, инкапсуляции, использованию классов и объектов в программе, наследованию, полиморфизму и параметризации классов. Третья глава предоставляет краткий обзор особенностей платформы *.NET Framework* и языка программирования C#. В четвертой главе речь идет о применении принципов инкапсуляции и абстрагирования в программах при разработке собственных классов на языке C#. Здесь описываются основные синтаксические конструкции языка, которые позволяют определить различные элементы класса. В завершении главы подробно рассматривается пример создания класса для решения квадратного уравнения. Пятая глава посвящена таким возможностям языка C#, как делегаты и события. С их помощью можно хранить информацию о методах класса и вызывать эти методы по необходимости. Наличие этих средств делает программы более гибкими. В шестой главе описываются правила применения принципов наследования и полиморфизма в языке C# и демонстрируется их применение на нескольких примерах. Седьмая глава дает представление о понятиях обобщенного класса, метода, интерфейса и др.

Все главы, начиная с четвертой, содержат большое количество примеров программ, демонстрирующих возможности языка C#. Эти программы разработаны и отлажены с помощью оболочки проектирования *Microsoft Visual Studio 2010*.

Глава 1. Объектная модель

1.1. Основные элементы объектной модели

На начальном этапе развития компьютерной техники и программирования основными являлись вычислительные задачи. Здесь центральным понятием являлся алгоритм – предписание выполнить точно определенную последовательность операций, которая преобразовывает входные данные в результат. Программа представлялась как средство реализации некоторого алгоритма.

Со временем вычислительные задачи становились все сложнее, и решающие их программы увеличивались в размерах. Это привело к изменению подходов в программировании. Программы приходилось разделять на все более мелкие фрагменты, которые решали конкретные подзадачи. Основой для такого разбиения стала процедурная (функциональная) декомпозиция. Программа, таким образом, превратилась в совокупность процедур, каждая из которых представляет собой законченную последовательность действий, направленных на решение отдельной задачи. Отдельно выделялась главная процедура, определяющая процесс решения задачи путем вызова в определенном порядке отдельных процедур. Такой подход в методологии создания программ назвали ***структурным программированием***. Одна из основных особенностей такой методологии заключалась в том, что появилась возможность создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. Период наибольшей популярности идей структурного программирования пришелся на конец 1970-х – начало 1980-х годов.

В 1980-е годы, когда массовое распространение получили персональные компьютеры, вычислительные и расчетно-алгоритмические задачи стали занимать второстепенное место. Компьютер перестал восприниматься в качестве простого вычислителя, он превратился в среду решения различных прикладных задач обработки и манипулирования данными. На первый план вышли задачи организации простого и удобного человеко-машинного взаимодействия, разработка программ с удобным графическим

интерфейсом, создание автоматизированных систем управления и пр. При решении этих задач принципы структурного программирования стали неэффективны, поскольку в процессе разработки таких приложений часто могли изменяться функциональные требования, что усложняло процесс создания программного обеспечения. Также увеличивались размеры программ, что требовало привлечения немалого числа программистов и дополнительных ресурсов для организации их согласованной работы.

В этот момент ведущим при разработке программ стал ***объектно-ориентированный подход***, который основан на использовании в программах набора моделей объектов, описывающих предметную область решаемой задачи. Такие ***модели*** называют ***объектными***, или ***объектно-информационными***.

Человечество в своей деятельности (научной, образовательной, технологической, культурной) постоянно создает и использует модели для описания окружающего мира (макет строящегося жилищного комплекса, модель самолета, эскизы картин и т.д.). Модели позволяют представить в наглядной форме объекты, взаимосвязи между ними, и процессы.

Основой объектно-информационной модели являются объекты. ***Объект*** – это часть окружающей нас действительности, воспринимаемая человеком как единое целое. Объекты могут быть материальными (предметы и явления) и нематериальными (идеи и образы), например, стол, стул, собака, сердце – это материальные объекты; матрица, теория относительности, законы Ньютона, философское учение Платона – примеры нематериальных объектов. Отдельно можно выделить объекты, которые добавляются в процессе программной реализации и не имеют никакого отношения к окружающей нас реальности – вектор, динамический список, бинарное дерево, хэш-таблица и пр.

Каждый объект характеризуется множеством ***свойств***. Информационная модель объекта выделяет из этого множества только некоторые свойства, существенные для решения конкретной задачи, позволяющие отделить этот объект от других. Объекты могут находиться в различных состояниях. ***Состояние*** объекта характеризуется перечнем всех его свойств и их текущими значениями.

В Таблице 1.1 следующей таблице приведены примеры некоторых объектов, их свойств и значений.

Обычно выделяется свойство (набор свойств), однозначно идентифицирующее объект во множестве объектов того же типа. Это свойство (набор свойств) называют ***идентичностью***. Например, свойством, отвечающим за идентичность объекта «Студент», является номер его зачетной книжки (в таблице номер зачетной книжки студента Петрова А.С. – 09-155).

Таблица 1.1. Примеры свойств объектов.

Имя объекта	**Наименование свойства**	**Значения свойств**
Студент 09-155	Имя Специальность Курс Форма обучения Номер зачетной книжки Номер учебной группы	Петров Андрей Сергеевич Математические методы в экономике 1 курс Контрактная форма 09-155 900Э
Мой жесткий диск	Объем Количество занятой памяти	200 Гб 103 Гб
Матрица А	Количество строк Количество столбцов Элементы матрицы	5 5 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

Как правило, объекты не остаются неизменными. Изменение состояния объекта отражается в его модели изменением значений его свойств. Например, на жестком диске изменяется объем занятой памяти, студенты переводятся на следующие курсы и пр.

В объектно-информационной модели отражаются не только свойства, но также и поведение объекта. ***Поведение объекта*** – действия, которые могут выполняться над объектом или которые может выполнять сам объект. Именно поведение объекта определяет его переход от одного состояния в другое. Опишем поведение объектов из нашего примера в Таблице 1.2.

Таблица 1.2. Примеры поведения объектов.

Имя объекта	**Поведение (действия)**
Студент 09-155	Посещение занятий Решение контрольных работ Ответ на семинарах Сдача зачетов Сдача экзаменов Перевод на следующий курс Отчисление Оплата обучения
Мой жесткий диск	Форматирование Копирование
Матрица А	Транспонирование Определение, является ли матрица квадратной Вычисление обратной матрицы Сложение матриц

Множество объектов с одинаковым набором свойств и поведением называется ***классом***. Все студенты обладают одним и тем же набором свойств (имя, специальность, курс, форма обучения, номер зачетной книжки, номер учебной группы, посещение занятий, сдача зачетов, сдача экзаменов и др.) и поэтому образуют класс объектов «Студент». Каждый конкретный студент – экземпляр этого класса (или объект). Следовательно, «Студент 09-155» – экземпляр класса «Студент». Аналогично можно ввести класс «Жесткий диск», объединив в нем все жесткие диски. Тогда «Мой жесткий диск» – экземпляр класса «Жесткий диск».

Таким образом, ***экземпляр класса*** – это конкретный предмет или объект, а класс определяет множество объектов с одинаковым набором свойств и поведением. Класс может порождать произвольное число объектов, однако любой объект относится к строго фиксированному классу. Класс объекта – это его неявное свойство.

1.2. Отношения между объектами и классами

Проектирование объектной модели сводится не только к определению классов, которые описывают предметную область. Классы не существуют автономно – они взаимодействуют между собой. Поэтому в объектную модель включается также описание связей (отношений) между классами.

Наиболее распространенными при описании предметной области модели являются следующие три типа связей – ассоциация, обобщение и зависимость.

Ассоциацией называется структурное отношение, показывающее, что объекты одного типа связаны с объектами другого типа. Например, высказывание «студент учится в вузе» определяет ассоциацию между объектами классов «Студент» и «Вуз». Эта ассоциация является простой, т.е. ни один из классов, участвующих в ней, не является более важным, чем другой. Отношение ассоциации изображено на Рис.1.1.

Рис. 1.1. Ассоциация «Студент-Вуз»

Ассоциации обычно описываются именем, отражающим природу отношения между объектами. На рисунке именем ассоциации служит «учится в». При определении ассоциации указывается, какое количество объектов каждого класса участвует в отношении. Это количество называют ***кратностью ассоциации***. Так, в примере ассоциации «Студент-Вуз» кратность характеризуется высказыванием «В одном вузе учится много студентов, но каждый студент учится только в одном вузе». Заметим, что любой человек может учиться в нескольких вузах одновременно, но в этом случае роль студента он выполняет для каждого вуза в отдельности.

Особым видом ассоциации является ***агрегирование*** – отношение типа «является частью» («is-part-of»), когда объект-целое состоит из нескольких объектов-частей. Например, высказывание «группа состоит из студентов» определяет отношение агрегации между объектами классов «Группа» и «Студент» (Рис.1.2).

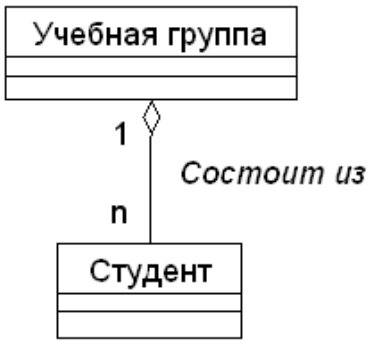

Рис.1.2. Агрегирование «Учебная группа-Студент»

Частным случаем агрегирования является ***композиция*** – отношение, когда время жизни частей и целого совпадают. Примером такой связи является отношение «Вуз-Факультет» – после ликвидации вуза факультеты как самостоятельные единицы существовать не могут (Рис.1.3).

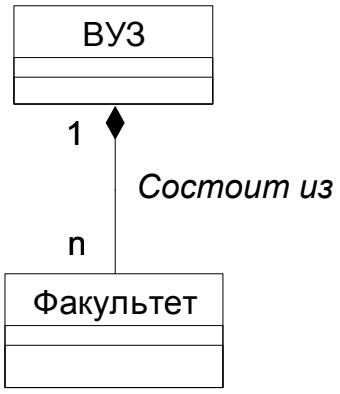

Рис.1.3. Композиция «Вуз-Факультет»

В отличие от этого агрегация «Группа-Студент» не обладает таким свойством – при распределении студентов по специализациям осуществляется переформирование групп (прежние группы упраздняются, новые группы формируются). При этом объекты-студенты не уничтожаются.

Обобщение – это отношение между общим классом (суперклассом, родителем) и одной или несколькими его вариациями (подклассами, потомками). Обобщение объединяет классы по их общим свойствам и поведению, что обеспечивает структурирование описания объектов.

Обобщение иногда называют отношениями типа ***«является» («is-a»)***, имея в виду, что одна сущность (класс «Студент-контрактник») является частным случаем другой, более общей (класс «Студент»). Обобщение означает, что объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот. Потомок может быть подставлен вместо родителя. При этом он наследует свойства родителя – его атрибуты и операции. Часто, хотя и не всегда, у потомков есть и свои собственные атрибуты и операции, помимо тех, что существуют у родителя (Рис.1.4).

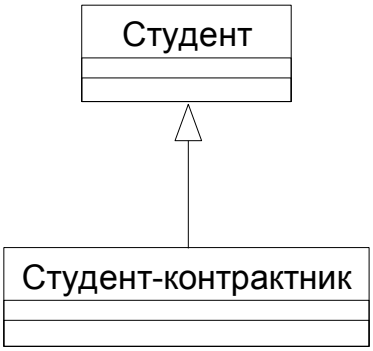

Рис.1.4. Обобщение между классами «Студент»
и «Студент-контрактник»

В случаях, когда класс-потомок не содержит собственных атрибутов и операций, но реализация некоторых унаследованных им методов отличается от родительских, определяется отношение типа ***«является подобным» («is-like-a»)***. Примером такого отношения является иерархия классов для студентов разных курсов. Все студенты сдают зачеты и экзамены, количество которых может быть различным в зависимости от курса обучения. Поэтому методы допуска к экзаменационной сессии, перевода на следующий курс и т.д. будут иметь одинаковый прототип, но их реализация будет различной.

Отношение ***зависимости*** – это такой тип отношения, при котором изменение в определении одного класса приводит к изменению реализации другого класса. Например, изменение в классе «Студент» (добавление новых методов, изменение прототипов существующих методов и пр.) может привести к изменениям в классе «Учебная группа». Чаще всего такая связь возникает в случаях, когда классы находятся в отношении агрегации или когда объекты одного класса являются параметрами методов другого класса.

1.3. Объектная модель вузовской информационной системы

Приведем несколько примеров объектных моделей, связанных с различными приложениями вузовской информационной системы. Как уже было сказано, для определения объектной модели необходимо описать объекты предметной области, относящиеся к решению задач, и отношения между ними.

Пример 1. Объектная модель системы «Абитуриент».

Классами, которые определяют основные объекты системы «Абитуриент» и ее предметную область, являются:

- «Абитуриент», свойствами которого являются ФИО, место жительства, номер школы, паспортные данные, выбранная специальность, результаты ЕГЭ и т.д;
- «Результат вступительных испытаний», определяющий баллы, которые получил конкретный абитуриент при сдаче экзамена по одному предмету (ЕГЭ или вступительные экзамены);
- «Специальность», которая описывается следующими атрибутами: код, название, план приема, стоимость обучения, проходной балл и т.д.
- «Факультет», у которого задаются: код, название факультета, список специальностей;
- «Приказ о зачислении», характеризующийся номером, датой и списком абитуриентов, зачисленных в студенты по различным специальностям;
- «Студент», основная информация о котором совпадает с атрибутами класса «Абитуриент», а также имеются новые свойства, характерные для других приложений.

Опишем связи между этими классами, необходимые для данного приложения.

1. Между классами «Факультет» и «Специальность» существует ассоциация, показывающая, что на каждом факультете ведется подготовка специалистов по конкретным специальностям. На одном факультете может быть несколько специальностей,

подготовка по каждой специальности может вестись только на одном факультете.

2. Между классами «Абитуриент» и «Специальность» существует ассоциация, показывающая, что абитуриент подает заявление на участие в конкурсе по конкретной специальности. Один абитуриент может подать заявления на несколько специальностей, на каждую специальность может быть подано большое количество заявлений от абитуриентов.

3. Между классами «Абитуриент» и «Результат экзамена» существует ассоциация, демонстрирующая, что любой абитуриент должен предоставить свои результаты ЕГЭ или сдать вступительный экзамен для участия в конкурсе. Один абитуриент предоставляет результаты нескольких экзаменов, каждый результат принадлежит только одному абитуриенту.

4. Между классами «Абитуриент», «Специальность» и «Приказ о зачислении» существует тернарная (между тремя классами) ассоциация, которая описывает формирование приказов о зачислении абитуриентов на специальности. Приказ может содержать списки абитуриентов, зачисленных в студенты по разным специальностям. Каждый абитуриент при этом может быть зачислен только на одну специальность, т.е. его фамилия присутствует в приказе только один раз.

5. Между классами «Приказ» и «Студент» существует ассоциация, которая производит изменение статуса зачисленных абитуриентов в студенты (Рис. 1.5).

6. Как правило, выделяются различные типы абитуриентов – граждане РФ и иностранные граждане. Отличия этих типов состоит в отсутствии результатов ЕГЭ у иностранных граждан. Следовательно, можно создать два новых класса – «Абитуриент – гражданин РФ» и «Абитуриент – иностранный гражданин». Эти классы наследуют свойства и методы класса «Абитуриент», определяя тем самым отношение обобщения между ними (отношение типа «is-like-a»).

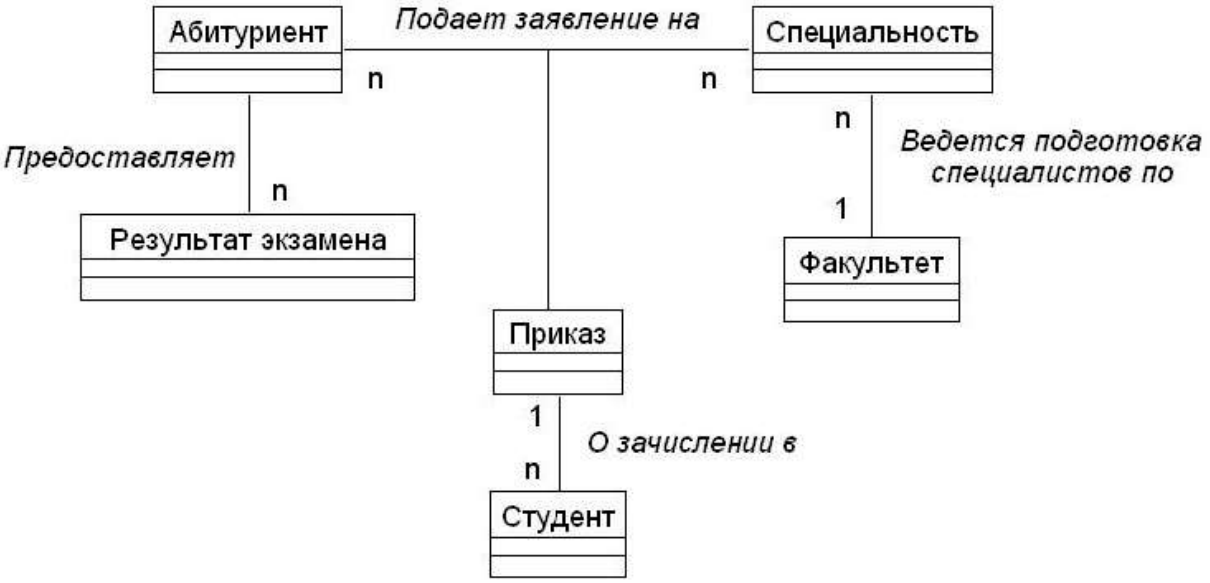

Рис.1.5. Объектная модель системы «Абитуриент».

7. Предположим, что только абитуриентам – гражданам РФ могут быть предоставлены льготные условия при поступлении (сиротам, военнослужащим, инвалидам и пр.). Тогда между классами «Абитуриент – гражданин РФ» и новыми классами «Абитуриент с льготами» и «Абитуриент без льгот» можно также выделить отношение обобщения (Рис. 1.6).

8. Другой вариант представления этой связи приводит к появлению отношения зависимости между классом «Абитуриент» и «Типы абитуриентов», который будет в себя включать перечисление типов абитуриентов (имеющие или не имеющие льготы). Зачисление в студенты будет происходить в зависимости от типа абитуриента, т.е. будет по-разному реализовано. Такое представление взаимосвязей между классами считается более правильным (Рис. 1.7).

Для простоты не будем рассматривать зачисление абитуриентов для получения второго высшего образования или зачисления в слушатели.

Рис. 1.6. Отношения обобщения в системе «Абитуриент»

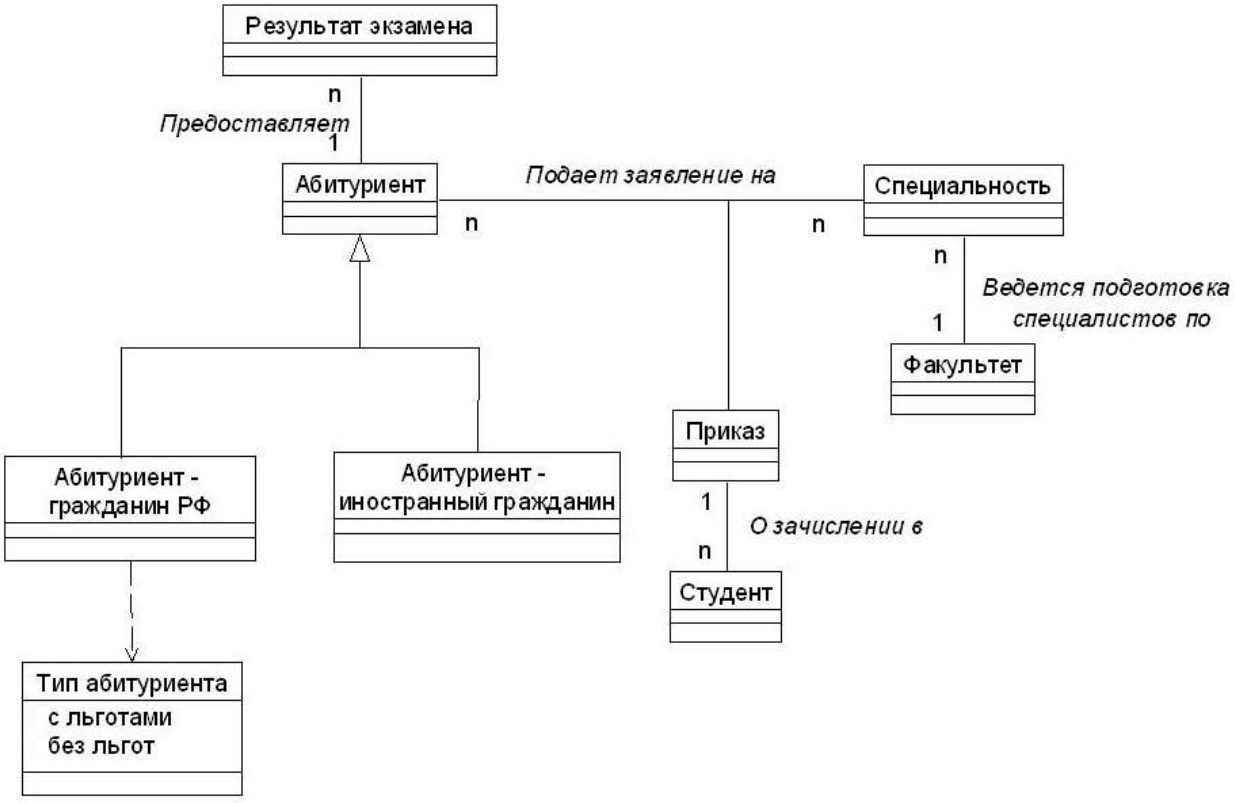

Рис. 1.7. Отношение зависимости в системе «Абитуриент»

Пример 2. Объектная модель системы «Формирование учебного расписания».

Основные объекты системы «Формирование учебного расписания» определяются следующими классами:

- «Расписание». Его свойствами является название факультета и список занятий;
- «Учебное занятие». Оно задается названием предмета, типом занятия (лекция, практика, консультация и т.д.), преподавателем, который его проводит, учебной группой, днем недели и временем проведения, аудиторией;
- «Преподаватель», проводящий занятие;
- «Предмет», по которому проводятся занятия;
- «Учебная группа», для которой проводится занятие;
- «Аудитория», в которой проводится занятие.

Расписание содержит в себе информацию обо всех учебных занятиях, т.е. определяет отношение типа «часть/целое» с классом «Учебное занятие». Данное отношение является отношением композиции, поскольку учебные занятия не существуют как самостоятельные сущности без расписания. Между классом «Учебное занятие» и оставшимися классами существуют ассоциации. Одно учебное занятие проводит один преподаватель, при этом один преподавать может проводить несколько занятий в разное время. Одно учебное занятие может проводиться для студентов нескольких учебных групп, при этом каждая группа может посещать несколько занятий в разное время. Одно учебное занятие ведется по конкретному предмету, но для каждого предмета может быть несколько занятий в расписании. Каждое учебное занятие проходит только в одной аудитории, но в этой же аудитории в другое время могут проходить другие занятия.

Как и в предыдущем примере, типы занятий можно выделить в отдельный класс, который будет накладывать ограничение на аудитории, в которых может проводиться занятие (в зависимости от вместительности аудитории или наличии определенной техники). Тем самым, образуется отношение зависимости между классами «Тип занятия» и «Аудитория».

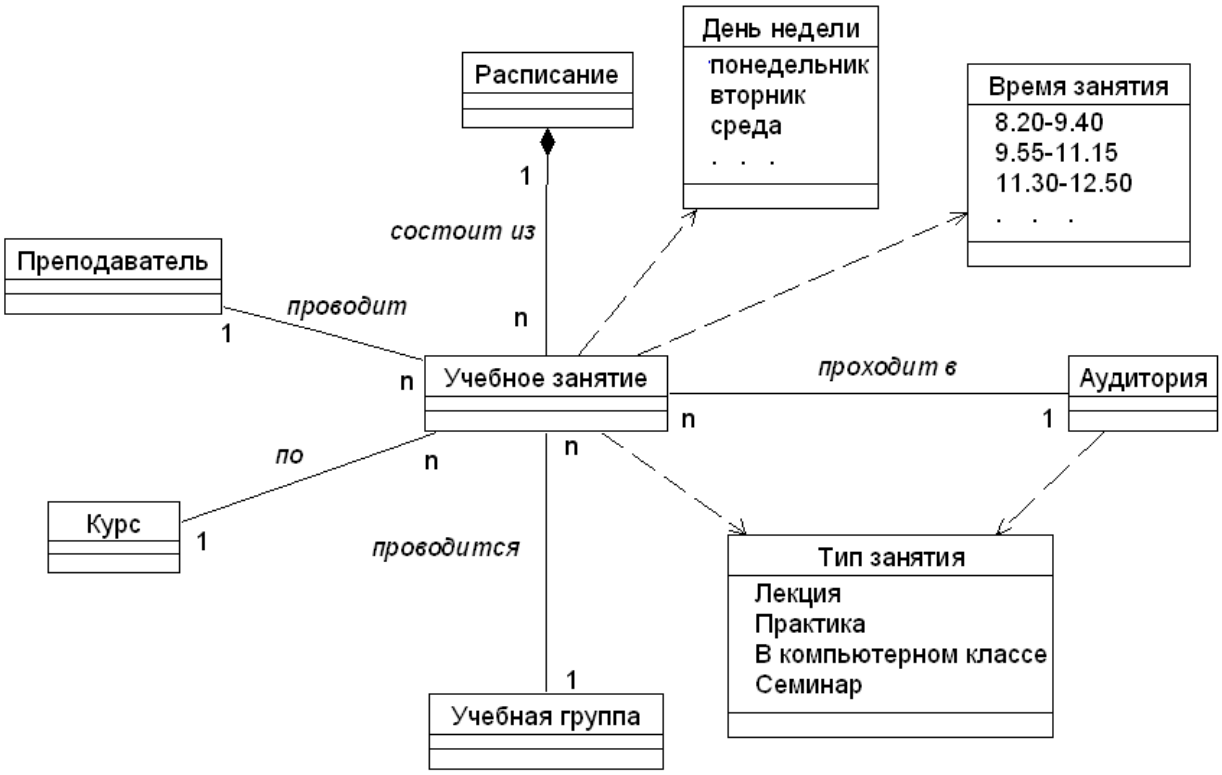

Рис.1.8. Объектная модель системы
«Формирование учебного расписания».

В заключение данного раздела отметим, какие преимущества при разработке программ в стиле объектно-ориентированного программирования дает построение объектной модели:

- объектная модель более естественная для разработки, поскольку ориентирована на человеческое восприятие мира, а не на компьютерную реализацию;
- объектная модель сложной программной системы может быть разделена на объектные модели ее подсистем. Поэтому процесс разработки носит обычно эволюционный характер, что предполагает развитие системы на базе уже разработанных небольших подсистем;
- объектная модель в дальнейшем позволяет использовать выразительные возможности объектных и объектно-ориентированных языков программирования, таких как C++, C#, Java и т.д.

Глава 2. Основные принципы объектно-ориентированного программирования

В объектно-ориентированной технологии используется особый подход к разработке программ, основанный на использовании объектных моделей и нескольких базовых концепциях. К этим концепциям относятся абстрагирование, инкапсуляция, полиморфизм, наследование.

2.1. Абстрагирование

Любая объектная модель содержит описание объектов, необходимых для работы приложения, и их взаимосвязей. Любой объект обладает большим количеством различных свойств. Каждый человек воспринимает объект по-своему, исходя из того, какие задачи приходится ему решать, работая с этим объектом. В этом случае для описания объекта выделяется некоторое количество его характеристик, существенных для решения задачи. К характеристикам объекта относятся его свойства, как с точки зрения его структуры, так и с точки зрения его поведения. Например, при приобретении велосипеда покупатель обращает внимание на:

- структурные характеристики: возрастная группа велосипедиста (детский, подростковый, взрослый), тип велосипеда (спортивный, прогулочный, горный, шоссейный), размер колес, количество передач, материал, из которого сделан велосипед, фирма-производитель, цвет, стоимость и др.;
- и поведение: переключение скорости, движение, торможение и др.

Из всех этих характеристик пользователь в данный момент выделяет только существенные. Предположим, покупатель выбирает велосипед для обучения ребенка езде на нем. В этом случае несущественными могут быть следующие характеристики: количество передач, фирма-производитель, цвет, переключение скорости и др.

Так и в программировании разработчики концентрируют свое внимание на существенных свойствах, необходимых для описания объекта, и на операциях, которые описывают его поведение. В этом и заключается ***абстрагирование***.

При абстрагировании выделяются те характеристики объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы с точки зрения наблюдателя.

Классы объектной модели представляют собой абстракции сущностей предметной области задачи. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Абстракция структурной характеристики объекта определяется своим именем и множеством значений, которые она может принимать. Например, свойство велосипеда «цвет» может принимать значения «красный», «зеленый», «серебристый» и др., свойство «количество передач» – 1, 4, 8 и др., «стоимость» - 15 000 рублей, 40 000 рублей и т.д. С точки зрения программирования множество значений определяется типом данных переменной, которая будет хранить значение этой характеристики.

В процессе использования объекта значения структурных характеристик могут изменяться. Например, увеличилась стоимость велосипеда или поменялся его цвет. Эти изменения, как правило, происходят в результате воздействия на объект, которое порождает некоторую ответную реакцию самого объекта. Действия, которые можно выполнить по отношению к данному объекту, и реакция объекта на них определяют абстракцию поведенческой характеристики. С точки зрения программирования она задается специальной функцией (***методом***).

Абстракцию множества объектов, которые имеют общий набор свойств и обладают одинаковым поведением, называют ***классом***. Каждый объект в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых характеристик или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

Например, в класс «Велосипед» добавим характеристики, которые описывают текущее состояние велосипеда в процессе использования –

СостояниеВелосипеда (стоит или двигается), ТекущаяСкорость и НомерПередачи. Изменяют эти свойства соответствующие им методы. Например, метод Остановить() может изменить поле ТекущаяСкорость на значение 0.0 и поле СостояниеВелосипеда на значение false.

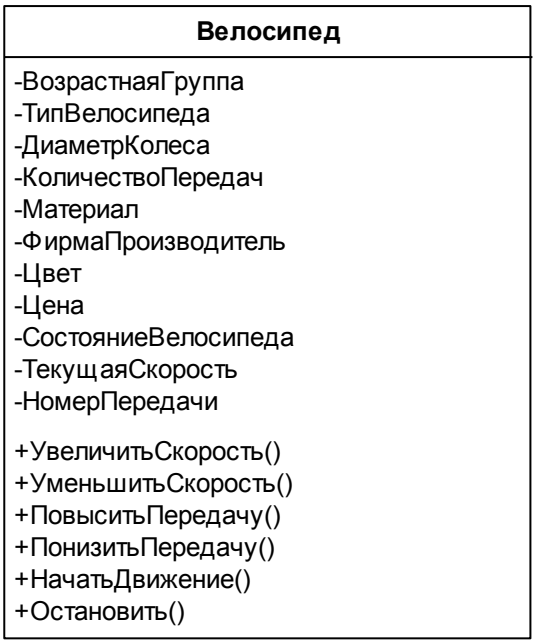

Рис.2.1. Класс «Велосипед».

Приведем еще один пример – класс «Матрица».

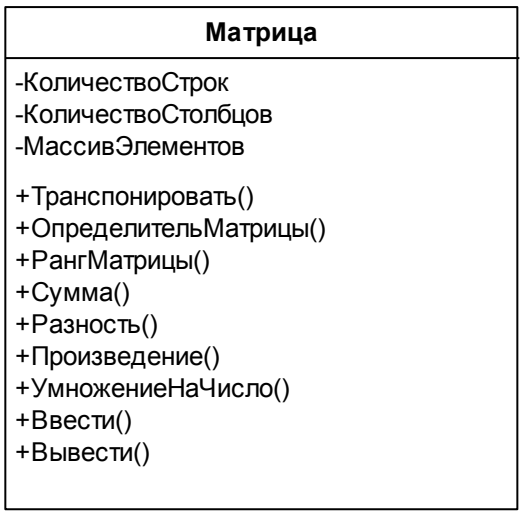

Рис. 2.2. Класс «Матрица».

Для создания объектов класса служат специальные методы, которые называют конструкторами. Они необходимы для корректной инициализации объекта. Например, при создании матрицы заданных размеров конструктор должен выделить память для хранения элементов этой матрицы. Уничтожением объекта также занимается специальный метод класса, который называют деструктором. Его задача – освободить ресурсы, занимаемые объектом (закрыть используемые файлы, соединения с базами данных и пр.).

2.2. Инкапсуляция

Следующий принцип объектно-ориентированного программирования (ООП) – ***инкапсуляция***. Этот термин характеризует сокрытие отдельных деталей внутреннего устройства класса от внешних по отношению к нему объектов или пользователей. Действительно, пользователю нет необходимости знать, из каких частей состоит экземпляр класса и каким образом реализовано его поведение. Реализация присущих классу структурных и поведенческих характеристик, является его собственным делом. Более того, отдельные свойства и методы вообще могут быть невидимы за пределами этого класса.

Покажем на примере класса «Матрица» (`Matrix`) применение принципа инкапсуляции. Этот класс содержит переменные, доступ к которым может быть осуществлен только из методов класса. Такой подход используется для того, чтобы защитить переменные от несанкционированного доступа. Например, если дать возможность программисту, который будет работать с объектом класса `Matrix`, изменять напрямую значения переменных для хранения размера матрицы, то эти значения могут стать некорректными (отрицательными, очень большими, или не соответствующими действительным размерам). А, выполняя то же самое с помощью методов класса, можно включить в них проверку корректности введенных значений. Для нашего класса таким методом может быть метод ввода матрицы.

Помимо структурных характеристик класса сокрытию подлежит и реализация операций класса. Для пользователя нет необходимости

знать, как реализован тот или иной метод. Надо знать только то, что метод выполняет, как к нему обратиться и как воспользоваться результатом его работы. Например, определитель квадратной матрицы можно вычислить различными способами: методом исключений Гаусса или по формуле Лапласа – разложение по строке или столбцу. Для пользователя не имеет значения, какой алгоритм заложен в методе класса.

Для реализации принципа сокрытия данных в объектно-ориентированных языках программирования явно определяется способ доступа к данным и методам класса. Доступ к элементам класса может быть открытым (`public`), защищенным (`protected`), закрытым (`private`) и др. К закрытым данным и методам можно обращаться только в методах самого класса. Защищенные данные и методы будут доступны для методов класса и классов, которые связаны с исходным отношением «родитель-потомок». К открытым (общедоступным) данным и методам можно обращаться из любого места программы.

Принципы абстрагирования и инкапсуляции используются совместно при разработке классов, дополняя друг друга. Уже на этапе выбора структурных и поведенческих характеристик класса, т.е. при применении принципа абстрагирования, определяется способ доступа к этим свойствам (применяется принцип инкапсуляции). Тем самым определяется внешний интерфейс класса – набор средств, которыми можно пользоваться извне при работе с объектами этого класса. Каждое такое средство определяет некоторое внешнее поведение объекта. Внутренняя же реализация этих средств скрыта от других объектов.

2.3. Наследование

Для любого объектно-ориентированного приложения необходимо определить объекты, которые оно будет содержать. Иногда эти объекты могут быть схожими по своей структуре и поведению, но при этом иметь существенные различия. Также бывают ситуации, когда некоторые объекты являются частями других объектов. Подобные взаимосвязи образуют иерархию объектов – их упорядочивание. Основными видами иерархических структур применительно к

объектно-ориентированным приложениям являются структура классов (иерархия «is-a» – «является», «is-like-a» – «является подобным») и структура объектов (иерархия «is-part-of» – «является частью»), которые определяются отношениями агрегации и обобщения.

Иерархия «is-part-of» была представлена ранее в объектной модели программы, автоматизирующей работу деканата. Ее определяет взаимосвязь классов «учебная группа» и «студент» – «студент является частью учебной группы».

В той же модели определены два класса студентов: бюджетной («Студент») формы обучения и контрактной («Студент-контрактник»). Большая часть их структурных и поведенческих характеристик совпадают. Объекты этих классов описываются фамилией, именем и отчеством студента, его датой рождения, номером зачетной книжки, средним баллом успеваемости и т.д. И те и другие сдают зачеты и экзамены, получают баллы, переводятся на следующий курс. При этом у студентов-контрактников имеются дополнительные характеристики: объекты этого класса могут содержать информацию об общей стоимости обучения за год, внесенной сумме, оставшейся задолженности, методы доступа и изменения этих свойств, методы проверки задолженностей по оплате обучения и т.д. Между этими двумя классами можно определить отношение обобщения, которое связано с использованием принципа наследования в объектно-ориентированном программировании.

Наследование – это механизм, который позволяет создавать новые классы на основе существующих, используя их структурные и поведенческие характеристики. Новые классы называют дочерними (производными или подклассами), а классы, на основе которых происходит наследование, – родительскими (базовыми или суперклассами). Кроме наследуемых свойств дочерние классы обладают дополнительными характеристиками, которые и отличают их от родительских.

В нашем примере в качестве родительского класса выступает класс «Студент», который наследуется дочерним классом «Студент-контрактник». Данное отношение классов образует иерархию вида «is-a» – «студент-контрактник является студентом».

Использование принципа наследования позволяет расширить базовый класс посредством создания производного класса, содержащего дополнительно:

- новые данные, которые будут определять дочерний класс. Например, расширяя структурные свойства класса «Студент», добавим переменные для хранения значений общей стоимости обучения за год, внесенной суммы, оставшейся задолженности и, тем самым, образуем структурные характеристики нового класса «Студент-контрактник»;
- новые поведенческие характеристики дочернего класса. Например, новым поведением объектов класса «Студент-контрактник» может служить внесение оплаты за обучение;
- переопределенные поведенческие характеристики базового класса. Например, если для студентов бюджетной формы обучения для определения допуска к экзамену необходимо наличие определенного количества набранных баллов, то для студентов-контрактников, кроме этого, необходимо отсутствие задолженности по оплате. Это приведет к переопределению в дочернем классе метода базового класса, осуществляющего проверку допуска студента к экзамену. Переопределенный метод имеет тот же прототип и скрывает в производном классе метод базового класса.

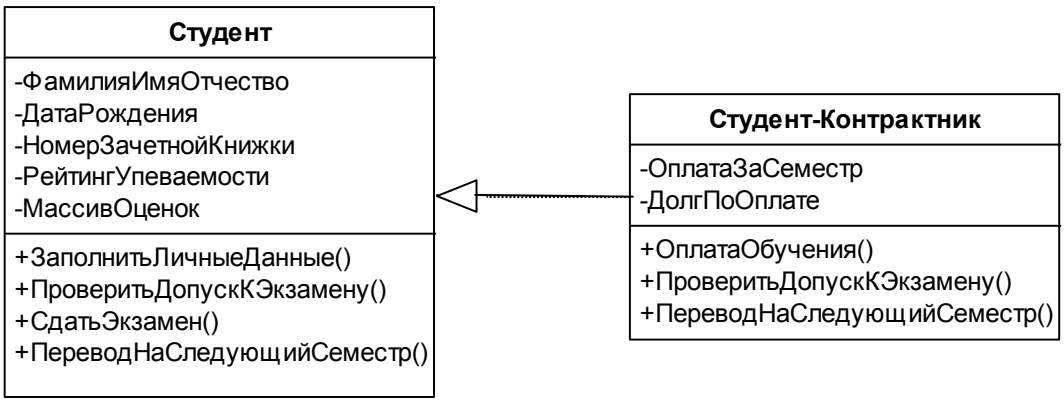

Рис.2.3. Отношение обобщения
«Студент»- «Студент-контрактник».

2.4. Полиморфизм

Слово «полиморфизм» означает «имеющий множество форм». В программировании под полиморфизмом понимают использование одного и того же имени для выполнения различных задач.

Полиморфизм – достаточно широкое понятие, в котором можно выделить следующие формы:

- перегрузка методов;
- перегрузка операций;
- использование методов с одним и тем же именем в различных классах, включая виртуальные функции;
- обобщенные классы (шаблоны классов).

Часто приходится разрабатывать методы, выполняющие одинаковые действия с различными типами данных. Например, методы сортировки массивов, содержащих элементы различных типов (целого, вещественного или символьного типов), удобно было бы называть одинаково. Поэтому в языках программирования предусмотрена возможность создавать методы с одинаковыми именами, но различными параметрами (параметры должны различаться количеством и/или типом данных). Такие ***методы*** называются ***перегруженными***. Типы возвращаемых значений у них также могут отличаться, однако использование методов, которые отличаются только типом возвращаемого значения недопустимо.

При проектировании классов, характеризующих поведение математических объектов, удобно использовать традиционные математические знаки операций для выполнения соответствующих действий. Например, при сложении двух матриц было бы понятней использовать операцию "+", а не вызывать метод `Summ()` (тем более, что другой программист может назвать его иным именем). Для таких ситуаций используется ***перегрузка операций***.

Несколько классов могут иметь ***методы с одинаковыми именами***. Это означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из классов относится тот или иной метод. Например, классы «Учебная группа» и «Студент» могут содержать методы для печати информации о соответствующем объекте, которые имеют одинаковое имя,

например, `Печать()`. При использовании метода `Печать()` будет выводиться информация о том объекте, для которого он вызван: при вызове метода для объекта класса «Учебная группа» будет печататься список студентов этой группы, а при вызове для объекта класса «Студент» – информация о студенте (ФИО, № зачетной книжки, № группы и др.). Это возможно за счет того, что каждый объект знает, какому классу он принадлежит, что позволяет вызвать метод именно этого класса.

Другой способ использования методов с одинаковыми именами в различных классах основан на понятии виртуального метода. Механизм использования ***виртуальных методов*** основывается на возможности хранения в переменной родительского класса объекта дочернего класса. По умолчанию выбор вызываемого метода осуществляется в соответствии с типом объекта, хранящегося в переменной. Если в родительском классе вызываемый метод был определен как виртуальный, а в дочернем классе он был переопределен, то будет вызываться последний.

Например, в одной группе могут учиться и бюджетные, и контрактные студенты. Тогда класс «Учебная группа» может содержать список переменных класса «Студент», некоторые из которых могут хранить объекты класса «Студент-контрактник». В классе «Студент» могут быть определены виртуальные методы (например, `ПроверитьДопускКЭкзамену()`, `Печать()`), которые должны быть переопределены в классе «Студент-контрактник». Тогда в методах класса «Учебная группа» информация о студентах может обрабатываться единообразно (посредством вызова виртуальных методов).

Обобщенные классы (шаблоны классов) определяют описание класса для обобщенного типа данных. При обращении к шаблону класса указывается конкретный тип данных (`int`, `double` или `класс`), который подставляется на место обобщенного типа. Таким образом, компилятор генерирует класс для этого конкретного типа. Подставляя разные типы данных, можно создать множество классов на основе шаблона, реализующих единый алгоритм обработки данных. Например, на основе шаблона класса «Матрица» могут создаваться объекты-матрицы, элементами которых являются целые числа, рациональные дроби (объекты класса «Дробь»), массивы и пр.

Глава 3. Об особенностях платформы .NET

В 2002 году компания Microsoft создала новую платформу разработки и выполнения программ, которая получила название *.NET Framework*. Это полностью объектно-ориентированная платформа, которая позволяет использовать уже имеющиеся и создавать собственные типы данных. В . *NET* под термином “тип” понимаются: классы, структуры, перечисления и иные формы данных. Платформа *.NET* позволяет разрабатывать компоненты (называемые сборками), которые предоставляют доступ к описанным в них типам другим компонентам (возможно написанным на других языках).

Основными целями платформы *.NET* являлось создание:

•нового формата выполняемых программных модулей – компонент (*EXE* и *DLL*), называемых сборками (*assembly*) или управляемыми модулями, основными особенностями которых является использование общего (независимого от исходного языка) промежуточного языка программирования и метаданных, описывающих все открытые типы данных, содержащиеся в них;

- специальной виртуальной машины (общеязыковой исполняющей среды, *common language runtime, CLR*), которая управляет компиляцией в инструкции процессора и выполнением модулей, составленных на промежуточном языке; *CLR* начинает работать при каждом запуске управляемых модулей на выполнение;
- общей библиотеки классов *.NET Framework* (*Framework Class Library, FCL*), которые помогают выполнить все базовую функциональность управляемых приложений (например, работа с коллекциями, файлами, сетями, графическим интерфейсом и т.п.).
- набора программных средств, помогающих разрабатывать управляемые модули, например, таких как компиляторы и отладчики; основным средством разработки является интегрированная среда разработки – Visual Studio, позволяющая автоматизировать разработку приложений на всех языках поддерживаемых платформой.

Платформа *.NET* активно развивается и с 2010 года уже используется версия *.NET Framework 4.0*.

Основой *.NET Framework* является общеязыковая среда выполнения программ (*CLR*). До разработки данной платформы приложения компилировались в инструкции конкретного процессора, выполнялись в процессах операционной системы *Windows* (ОС) и могли выполнять самостоятельно любые доступные им действия. С использованием платформы *.NET*, все созданные приложения компилируются в команды общего промежуточного языка (*common intermediate language, CIL*) и исполняются под управлением *CLR*. В связи с этим, они называются управляемыми приложениями (*managed application*). Среда *CLR* является виртуальной машиной, которая расположена поверх ОС (выполняется под управлением ОС и использует все ее возможности) и управляет выполнением приложений, разработанных для платформы *.NET*.

CLR управляет компиляцией программы с языка *CIL* в инструкции процессора по запросу (just-in-time) в период выполнения приложения. Обычно компиляция любого метода происходит лишь раз – при первом его вызове, и затем результат компиляции кэшируется в памяти, чтобы при повторном вызове он мог быть исполнен без задержки.

Платформа *.NET* является многоязыковой, т.е. поддерживает несколько языков – C++, C#, Visual Basic и F#. Независимо от языка, на котором написаны управляемые приложения, они используют один и тот же интерфейс прикладного программирования (*Application Program Interface, API*) – библиотеку классов *.NET Framework FCL*.

Microsoft поставляет компиляторы создающие модули на промежуточном языке *CIL* для всех поддерживаемых языков. После компиляции в *CIL* создаются управляемые модули – файлы с расширениями *EXE*, *DLL* или *NETMODULE*, которые выполняются под управлением *CLR*. В каждом управляемом модуле имеется раздел метаданных, в котором приводится описание типов данных данного модуля – методы, поля и свойства созданных структур и классов.

В качестве прикладного интерфейса *.NET Framework* использует библиотеку классов *.NET Framework*, которая содержит более 10 000 различных типов: классов, структур, интерфейсов, перечислений и делегатов. Бесспорным достоинством *FCL* является то, что она полностью объектно-ориентированная и используется всеми языками, которые работают с платформой *.NET*.

Физически библиотека *FCL* представляет собой набор *DLL* файлов (файлов в формате динамических библиотек). Каждый файл *DLL* – это сборка, загружаемая *CLR* по запросу. Встроенные типы данных, такие как целые, вещественные, логические, реализованы в модуле *Mscorlib.dll*, другие типы разнесены по разным *DLL* файлам библиотеки *FCL*.

Для облегчения использования *FCL*, все ее содержание хорошо структурировано в виде иерархически организованных групп типов. Каждая группа типов называется пространством имен. Всего в *FCL* около 100 таких пространств. В каждом из них содержатся классы и другие типы, имеющие некоторое общее назначение. Например, большая часть *API* для управления объектами графического интерфейса приложений Windows содержится в пространстве имен *System.Windows.Forms* (классы, представляющие окна, диалоги, меню и другие элементы). Другие примеры пространств имен и характеристика их содержимого приведены в Таблице 3.1.

Таблица 3.1. Основные пространства имен библиотеки FCL

Пространство имен	Содержимое
System	базовые типы данных и вспомогательные классы;
System.Collections	коллекции, словари, массивы переменной размерности и другие контейнеры;
System.Data и др.	классы ADO.NET для доступа к данным;
System.Drawing	классы для рисования в окне (GDI+);
System.IO	классы файлового и потокового ввода –вывода;
System.Net	классы для работы с сетевыми протоколами, например, http;
System.Windows. Forms	классы для реализации графического интерфейса пользователя;
System.Xml и др.	классы для чтения и вывода данных в формате XML.
и пр.	

Глава 4. Классы в C#. Основные понятия

Основным элементом программы, созданной в объектно-ориентированном стиле, является ***класс***. Класс определяет тип данных для описания множества объектов с одинаковым набором свойств и поведением. Переменные этого типа данных называются объектами. Разработка программы заключается в описании новых классов, использовании стандартных классов, создании объектов и определении их взаимодействия.

Для определения класса в C# используются следующий синтаксис:

```
[атрибуты] [модификаторы доступа]
class имя_класса [: базовый класс [,интерфейсы]]
{
    // поля класса
    .    .    .

    // методы класса
    .    .    .

    // свойства класса
    .    .    .

    // индексаторы класса
    .    .    .

    // события класса
    .    .    .

    // подклассы, делегаты
    .    .    .
}
```

Все составляющие класса называются его элементами или членами класса. Обращение к отдельному элементу класса имеет синтаксис:

```
имя_объекта.имя_элемента.
```

Заметим, что в качестве элемента может выступать отдельный класс, который может выполнять вспомогательную роль для основного класса.

4.1. Поля класса

Поля класса определяют структурные характеристики объектов класса и используются для хранения значений этих характеристик. Для определения поля класса используется синтаксис:

```
[атрибуты] [модификаторы] тип_данных имя_поля [= значение];
```

В качестве примера рассмотрим определение класса «Граф». Напомним, что граф – это пара множеств V и E, где V – непустое конечное множество точек, называемых вершинами, а E – множество ребер, соединяющих пары вершин.

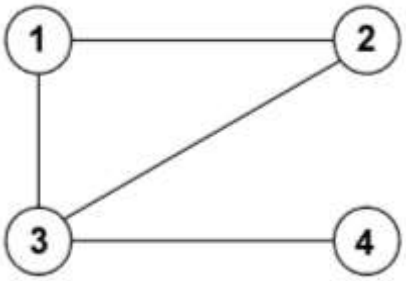

Рис. 4.1. Пример графа.

На Рис. 4.1 представлен граф с множеством вершин {«1», «2», «3», «4»} и множеством ребер {<1, 2>, <1, 3>, <2, 3>, <3, 4>}.

Часто для определения графа используется матрица смежности:

$$A(n, n)=\{ a_{ij} \},$$ где n – число вершин.

$$a_{ij} = \begin{cases} 1, \text{ вершина } i \text{ смежна вершине } j \\ 0, \text{ вершины } i \text{ и } j \text{ не смежны} \end{cases}$$

В случае взвешенного графа элементу a_{ij} вместо 1 присваивается значение соответствующего веса, а вместо 0 – достаточно большое число, обозначающее бесконечность. Эта матрица является

симметричной, а элементы, стоящие на главной диагонали, равны нулю. Так, матрица смежности для графа, изображенного на рис. 4.1, имеет вид:

$$A = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Исходя из вышесказанного, поля класса «Граф» должны содержать информацию о матрице смежности: количество вершин и двумерный массив ее элементов.

```
class Graph
{
   int n;              // количество вершин в графе
   double [,] graph;  // матрица смежности
   .   .    .
}
```

Для регулирования прав доступа к полям класса указываются модификаторы `public`, `private`, `protected` и `internal`. К `private`-полям можно обращаться только в методах самого класса. `Protected`-поля будут доступны для методов класса и классов, которые связаны с ним отношением «родитель-потомок». К `public`-полям можно обращаться из любого места программы. Модификатор доступа `internal` используется для полей, доступных всем классам, определенным в конкретной сборке. Если модификатор доступа не указан, по умолчанию поля считаются закрытыми (`private`).

Например, поля класса `Graph` с модификатором `public` будет таким:

```
class Graph
{
   // поля класса «Граф» являются общедоступными
   public int n;                  // количество вершин в графе
   public double [,] graph;       // матрица смежности
   .   .    .
}
```

В этом случае обращение к полям конкретного объекта класса `Graph` становится возможным из любого места программы.

Статические поля. Еще одним модификатором типа является `static`. Поле с модификатором `static` создается в единственном экземпляре для всех объектов данного класса. Обращение к этому полю вне класса, в котором оно определено, осуществляется по правилу `Имя_класса.Имя_статического_поля`, если, конечно, оно доступно. Внутри класса к статическим полям обращаются просто по имени. Самым простым примером использования статического поля класса является переменная, в которой хранится количество объектов данного класса.

Например, добавим статическое поле в класс `Graph`:

```
class Graph
{
   int n;              // количество вершин в графе
   double [,] graph;  // матрица смежности
   // статическое поле класса «Граф»
   static int count;  // количество объектов типа Graph
   .   .    .
}
```

В момент создания очередного объекта значение этого поля должно увеличиваться на 1, а при уничтожении объекта – уменьшиться на 1.

Константы. Модификатор `const` определяет поле, которое представляет собой обозначение некоторой константы. Инициализация константы происходит в момент ее объявления. Значением константы может быть любое константное выражение. Например, в классе «Круг» можно определить константу `PI`:

```
class Circle
{
   // объявление константы PI в классе «Граф»
   const double PI = 3.1415926;
   double radius;
   .   .    .
}
```

Далее `PI` можно использовать при вычислении площади круга или длины окружности, при этом изменять значение `PI` запрещено. Константное поле, содержащееся в классе, является статическим.

Переменные readonly. Модификатор `readonly` определяет поле, значение которого доступно только для чтения. В отличие от константы такие поля могут быть инициализированы как при их объявлении, так и в специальном методе класса (конструкторе), который создает объект класса. Вторым отличием является возможность присвоения этой переменной значения, полученного в результате вычисления некоторого выражения. Третье отличие заключается в том, что одно и то же `readonly`-поле в двух различных объектах может принимать различные значения (в константах же значение для всех объектов класса будет одинаковым).

Например, в следующем примере `readonly`-переменная `time` инициализируется в конструкторе класса значением текущего времени.

```
// пример использования readonly-поля для хранения даты
class Program
{
   readonly DateTime tt;

   Program()
   {
      // инициализация поля в конструкторе класса
      time = DateTime.Now;
   }

   static void Main(string[] args)
   {
      Program P = new Program();
      Console.WriteLine("time="+P.time);
   }
}
```

4.2. Методы класса

4.2.1. Основные понятия

Методы задают операции, которые могут выполнять объекты класса. Это самостоятельная программная единица, которая решает конкретную задачу. Метод можно рассматривать как преобразователь некоторых входных данных в результат (выходные данные метода). Неявным параметром является тот объект, который вызывает данный

метод. Доступ к нему можно получить с помощью ключевого слова this. Тип данных результата задает тип возвращаемого значения метода. Список формальных параметров метода – это список имен объектов с указанием их типов, которые являются входными данными метода.

Для определения метода класса используется синтаксис:

```
[атрибуты] [модификаторы доступа]
тип_возвр_знач имя_метода (список_форм_параметров)
{
   тело_метода
   .   .   .
   [return результат];
}
```

тип_возвр_значения (тип возвращаемого значения) – тип данных результата работы метода, список_форм_параметров (список формальных параметров) – список имен объектов с указанием их типов, которые представляют собой входные данные метода:

```
список_форм_параметров ::= Тип1 имяОб1,
                           Тип2 имяОб2, … ТипN имяОбN
```

Формальный параметр определяет локальную переменную метода, которая будет хранить значение какого-либо входного объекта.

Тело метода содержит программу, решающую конкретную задачу. Результат работы передается в операторе return.

Обращение к методу происходит посредством его вызова, при котором на место формальных параметров подставляются фактические параметры (аргументы). Типы аргументов должны совпадать с типами данных соответствующих формальным параметрам.

Например, для осуществления выбора минимального из двух чисел может быть написан метод, входными данными которого являются два исходных числа, а результатом – то из них, которое является минимальным.

```
// класс для выбора минимального из набора элементов
class Minimizer
{
   // метод поиска минимума из двух чисел
   double MinElement(double a, double b)
```

```
    {
        if (a<b) return a;
        else return b;
    }

    static public void Main(string [] args)
    {
        // создание объекта класса Minimizer
        Minimizer  ob = new Minimizer();
        Console.WriteLine("Введите число");
        double a1 = double.Parse(Console.ReadLine());
        Console.WriteLine("Введите еще одно число");
        double a2 = double.Parse(Console.ReadLine());

        // вызов метода с фактическими параметрами -
        // значениями переменных a1 и a2
        double res = ob.MinElement(a1, a2);
        Console.WriteLine("Минимальным из {0} и {1} является
                                {2}", a1, a2, res);
    }
}
```

К методам, как и к полям, применяются модификаторы доступа – `public`, `private`, `protected`, `internal`. `Private`-методы можно вызывать только в методах самого класса. `Protected`-методы будут доступны для методов класса и классов, которые связаны с исходным отношением «родитель-потомок». К `public`-методам можно обращаться из любого места программы. Модификатор доступа `internal` используется для методов, доступных всем классам, определенным в конкретной сборке. Если модификатор доступа не указан, по умолчанию методы считаются закрытыми (`private`).

К методам может также применяться модификатор `static`. Вызов таких методов осуществляется для класса в целом:

```
Имя_класса.Имя_метода(список_аргументов);
```

Нестатические поля в таких методах недоступны.

`Static`-методы часто применяются в стандартных библиотеках. Например, методы `ReadLine()` и `WriteLine()` являются статическими в классе `Console`, метод `Parse` – статический в классах `Int32`, `Float`, `Double`:

```
Console.WriteLine("Введите целое число:");
```

```
string s;
s = Console.ReadLine();
int i = Int32.Parse(s);
```

Вернемся к классу `Minimizer`. Часто бывает нужно выбирать минимум не из двух чисел, а из некоторого набора – из двух, трех, пяти, массива значений или из пустого списка параметров. В таких случаях говорят о ***методе с переменным числом параметров***, которые должны иметь один и тот же тип данных. При вызове такого метода из его фактических параметров формируется массив. Формальный параметр, задающий этот массив, предваряется ключевым словом `params` и должен располагаться в списке параметров метода последним. Например, добавим в класс `Minimizer` метод для поиска минимума в наборе элементов:

```
class Minimizer
{
   .   .   .
   // поиск минимального элемента в массиве или в наборе
   // целых чисел, перечисленных через запятую в параметрах
   double MinElement(params double [] a)
   {
      // для каждого типа данных определены константы,
      // задающие максимальное (MaxValue) и минимальное
      // (MinValue) значение из их диапазона
      double min = Double.MaxValue;
      foreach(double elem in a)
         if (elem < min)
            min = elem;
      return min;
   }

   static public void Main(string [] args)
   {
      Minimizer  ob = new Minimizer();
      // вызов метода для трех значений
      double res = ob.MinElement(5, 13, 7);
      Console.WriteLine("Минимальным из 5 и 13
                                    является {0}",  res);

      // вызов метода для массива
      double [] ar = new double[5];
      // заполнение массива случайными числами
      Random r = new Random();
      for(int i = 0; i < 5; i ++)
         ar[i] = r.next();
      res = ob.MinElement(ar);
```

```
        Console.WriteLine("Минимальным элементом массива
                                          является {0}", res);

        // вызов метода без параметров
        res = ob.MinElement();
        // в этом случае результат будет равен максимальному
        // из диапазона чисел типа double
        if (res == Double.MaxValue)
           Console.WriteLine("Данных в наборе не было");
        else
           Console.WriteLine("Минимальным элементом
                                        является {0}",  res);
    }
}
```

Напомним, что по определению метод может возвращать в качестве результата только один объект, тип которого определен как тип возвращаемого значения метода. Тем не менее, нередко возникают задачи, в которых результатом работы метода должны быть два или более значений. В этом случае в метод добавляются ***выходные параметры*** (`out`-параметры). Метод обязательно должен записать в них какое-то значение. В определении метода и при его вызове выходной параметр предваряется ключевым словом `out`.

Например, пусть требуется получить не только значение минимального элемента массива, но и количество таких элементов. Метод, решающий эту задачу, может возвращать значение минимального элемента, а количество вхождений будет возвращаться через выходной параметр.

```
// метод получения минимального элемента массива и
// количества его вхождений в массив
double MinElement(double [] a, out int count)
{
   double min = a[0];
   count = 1;
   for (int i = 1; i < a.Length; i++)
      if (a[i] < min)
      {
         min = a[i];
         count = 1;
      }
      else if (a[i] == min)
               count++;
   return min;
}
```

Вызов метода осуществляется как в следующем примере:

```
// вызов метода для массива
double [] ar = new double[5];

// заполнение массива случайными числами
Random r = new Random();
for(int i = 0; i < 5; i ++)
   ar[i] = r.Next();

// создание переменной для выходного параметра метода
int count_min_element;
double res = ob.MinElement(ar, out count_min_element);
Console.WriteLine("Минимальным элементом массива
                                        является {0}", res);
Console.WriteLine("Он встречается в массиве {0} раз",
                                        count_min_element);
```

Заметим, что поскольку `count_min_element` является `out`-параметром, то перед вызовом нет необходимости его инициализировать (значение обязательно будет присвоено в методе).

В зависимости от типа данных входные параметры передаются в метод по значению или по ссылке. Ссылочными типами данных являются классы и интерфейсы. Все остальные типы данных являются значимыми (структурными). При передаче параметра значимого типа в методе создается копия данного объекта, и все действия происходят именно с ней. При завершении работы метода эта копия уничтожается, а значение входного параметра остается неизменным.

Переменная ссылочного типа данных хранит адрес объекта в памяти. Создание объекта происходит с помощью операции `new`, которая возвращает адрес созданного объекта. Например,

```
// создание объекта класса Minimizer с помощью конструктора
Minimizer  ob = new Minimizer();
```

Переменная `ob` представляет собой ссылку на объект класса `Minimizer`.

Если объект ссылочного типа данных передается в метод, то через ссылку предоставляется доступ к полям этого объекта. При этом все

изменения, произведенные с полями объекта в методе, сохраняются и после завершения его работы, но значение самой ссылки в методе изменить нельзя.

Когда необходимо изменить значение некоторого параметра внутри метода, в том числе и ссылки на объект, его описывают как **ref-параметр**. В отличие от `out`-параметра он должен быть проинициализирован до его использования в качестве аргумента при вызове. Ключевое слово `ref` указывается для такого параметра при определении и при вызове метода. В следующем примере, переменная для хранения количества минимальных элементов создается и инициализируется значением 1 до вызова метода.

```
// метод получения минимального элемента массива и
// количества его вхождений в массив
double MinElement(double [] a, ref int count)
{
   double min = a[0];
   for (int i = 1; i < a.Length; i++)
      if (a[i] < min)
      {
         min = a[i];
         count = 1;
      }
      else if (a[i] == min)
         count++;
   return min;
}
```

Вызов метода:

```
// вызов метода для массива
double [] ar = new double[5];

// заполнение массива случайными числами
Random r = new Random();
for(int i = 0; i < 5; i ++)
   ar[i] = r.Next();

 // создание переменной для
 // параметра-ссылки
 int count_min_element = 1;
 double res = ob.MinElement(ar, ref count_min_element);
 Console.WriteLine("Минимальным элементом массива
                                    является {0}", res);
 Console.WriteLine("Он встречается в массиве
                          {0} раз",  count_min_element);
```

Иногда возникает необходимость производить одинаковую обработку для разных типов данных. В этом случае удобно определять несколько методов с одинаковым именем для разных типов данных. Такой прием называют перегрузкой функций (методов). Перегруженные методы должны иметь различные списки формальных параметров. Возвращаемые значения перегруженных функций также могут не совпадать, но это не должно быть единственным отличием.

Этот прием часто используется в библиотечных классах. Например, в классе `Console` для печати какого-либо сообщения существует 19 перегруженных методов `WriteLine()`:

- `WriteLine()` – записывает текущий признак конца строки в стандартный выходной поток;
- `WriteLine(double)` – записывает текстовое представление вещественного числа в стандартный выходной поток;
- `WriteLine(String)` – записывает заданную строку в стандартный выходной поток;
- `WriteLine(String, object[])` – записывает текстовые представления заданного массива объектов в стандартный выходной поток с использованием заданных сведений о форматировании и т.д.

4.2.2. Рекурсивные методы

Рекурсивные методы – это функции, в теле которых содержится вызов самих себя. Обычно их используют в ситуациях, когда легко свести исходную задачу к задаче того же вида, но с другими исходными данными, например, уменьшение размерности задачи, переход в новую точку и пр.

Каждый новый вызов рекурсивного метода предполагает сохранение состояния всех переменных в специальной области памяти. После выполнения рекурсивного метода, управление передается оператору, следующему за его вызовом. При этом состояние всех переменных восстанавливается. Поэтому применение таких методов считается очень затратным с точки зрения памяти и не

подходит для решения задач большой размерности. Тем не менее, существует множество задач, в которых рекурсивные алгоритмы более просты и дают компактные программы.

Следует помнить, что любой рекурсивный метод должен иметь обязательное условие выхода из рекурсии. При выполнении этого условия управление передается в место предыдущей итерации, откуда произошел рекурсивный вызов или в вызывающую функцию.

Далее приведем несколько примеров.

Пример 1 . Вычислить $n!$.

Когда рекурсия заключается в сведении к задаче меньшей размерности, алгоритм аналогичен обратному ходу метода математической индукции.

Очевидно, что $n!=n*(n-1)!$ Эта формула и определяет основное рекуррентное соотношение – $(n-1)!$ вычисляется с помощью вызова функции с фактическим параметром, принимающим значение $n-1$. Условие выхода из рекурсии – это сведение задачи к тривиальному случаю – 1!=1, 0!=1.

```
class Program
{
   // метод поиска факториала целого числа
   static uint Factorial(uint n)
   {
      // условие выхода из рекурсии
      if(n==1 || n==0)
         return 1;
      return n*Factorial(n-1);// рекурсивный вызов
   }

   static void Main(string[] args)
   {
      uint n;
      Console.WriteLine("Введите n:");
      while(true)
      {
         try
         {
            n=uint.Parse(Console.ReadLine());
            break;
         }
         catch (Exception ex)
         {
            Console.WriteLine("Введите корректное
                                                  число.");
         }
```

```
        }
        uint k=Factorial(n);        // первый вызов функции
        Console.WriteLine("{0}!={1}", n, k);
    }
}
```

На примере решения этой задачи рассмотрим, каким образом выполняется рекурсивный вызов метода. Допустим, пользователь ввел значение переменной *n*, равное 4.

Главная функция содержит вызов функции `Factorial(4)`. Для выполнения вызова метода в оперативной памяти выделяется некоторая область, необходимая для запоминания адреса возврата, создания копий параметров и копий локальных переменных.

Свободная область памяти в момент первого вызова `Factorial(4)` на Рис. 4.2 изображена в виде пустого прямоугольника. Внутри прямоугольника показано значение параметра *n*.

Вычисление факториала в методе `Factorial(4)` начинается с проверки условия выхода из рекурсии (`n==1` или `n==0`). Так как оно не выполняется, происходит второй вызов `Factorial(3)`, который является первым рекурсивным вызовом метода. Для второго вызова также выделяется некоторое количество памяти (Рис.4.3).

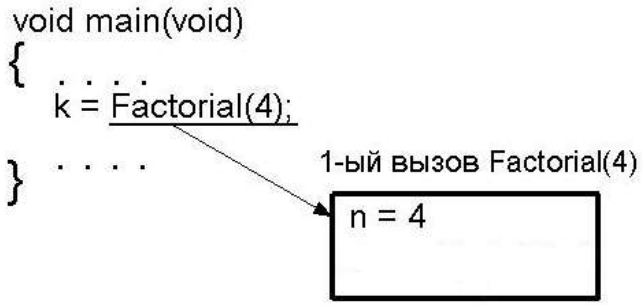

Рис. 4.2. Демонстрация первого вызова метода `Factorial(4)`.

Далее процесс повторяется, до тех пор, пока на четвертом вызове `Factorial(1)` не будет выполнено условие выхода из рекурсии (Рис. 4.4). Поэтому после четвертого вызова происходит возврат в вызывающую функцию (т. е. в третий экземпляр `Factorial(2)`, Рис. 4.5).

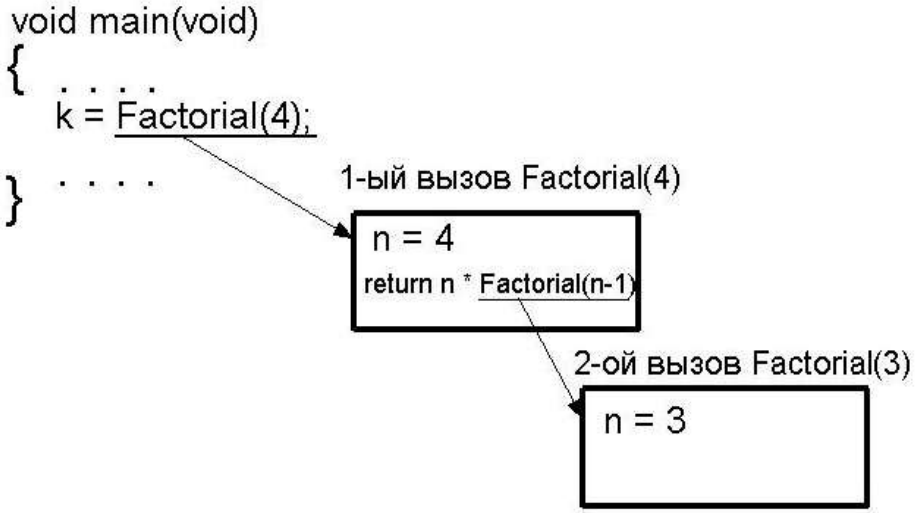

Рис. 4.3. Демонстрация первого рекурсивного вызова метода `Factorial(3)`.

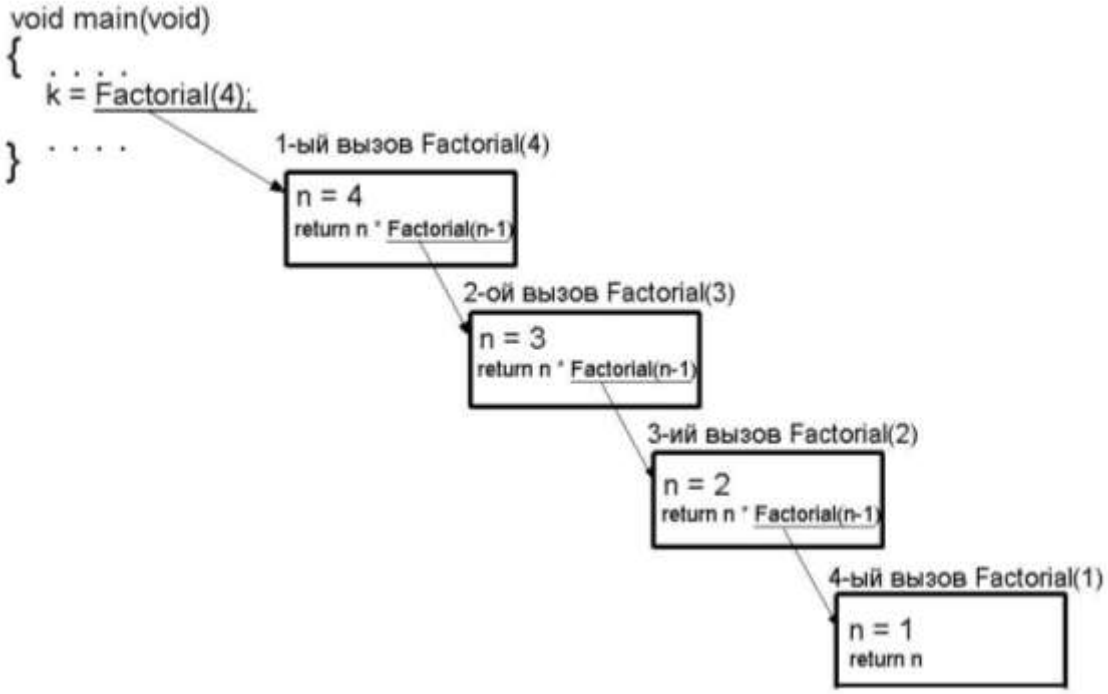

Рис. 4.4. Демонстрация рекурсивного вызова метода `Factorial(1)`.

Третий экземпляр `Factorial(2)` завершается, но перед завершением возвращает значение 2*1 (*n**результат 4-го вызова) и т. д. Первый экземпляр метода перед завершением работы возвратит значение 4*результат 2-го вызова (Рис. 4.6).

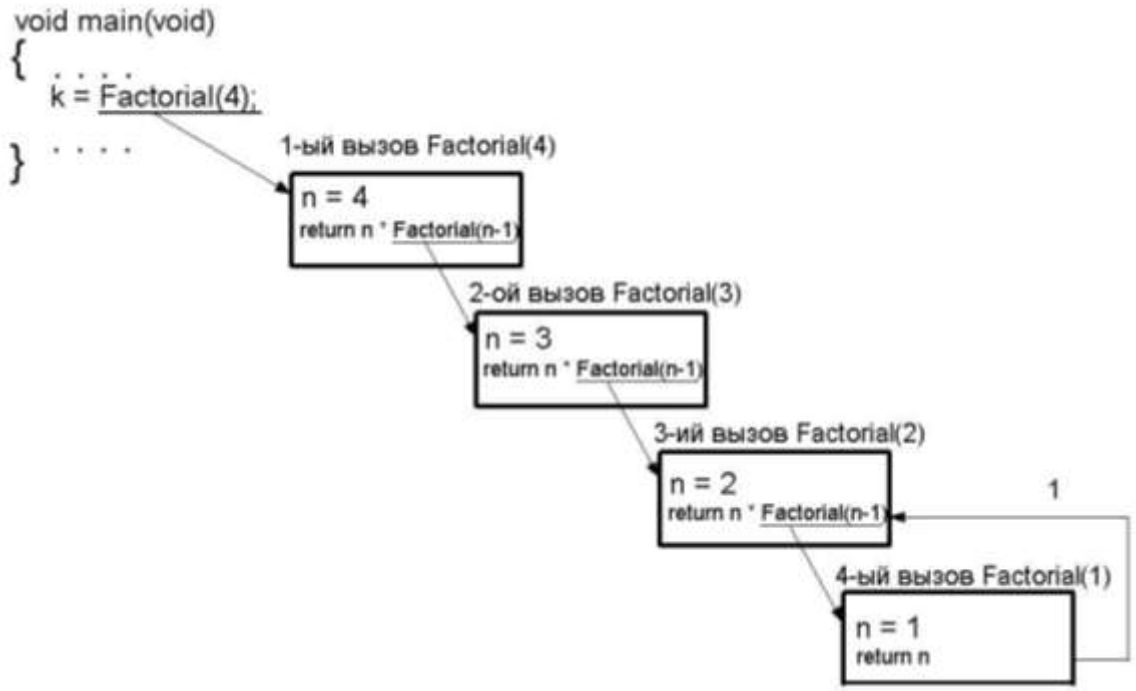

Рис. 4.5. Возврат из рекурсивного вызова метода `Factorial(1)`.

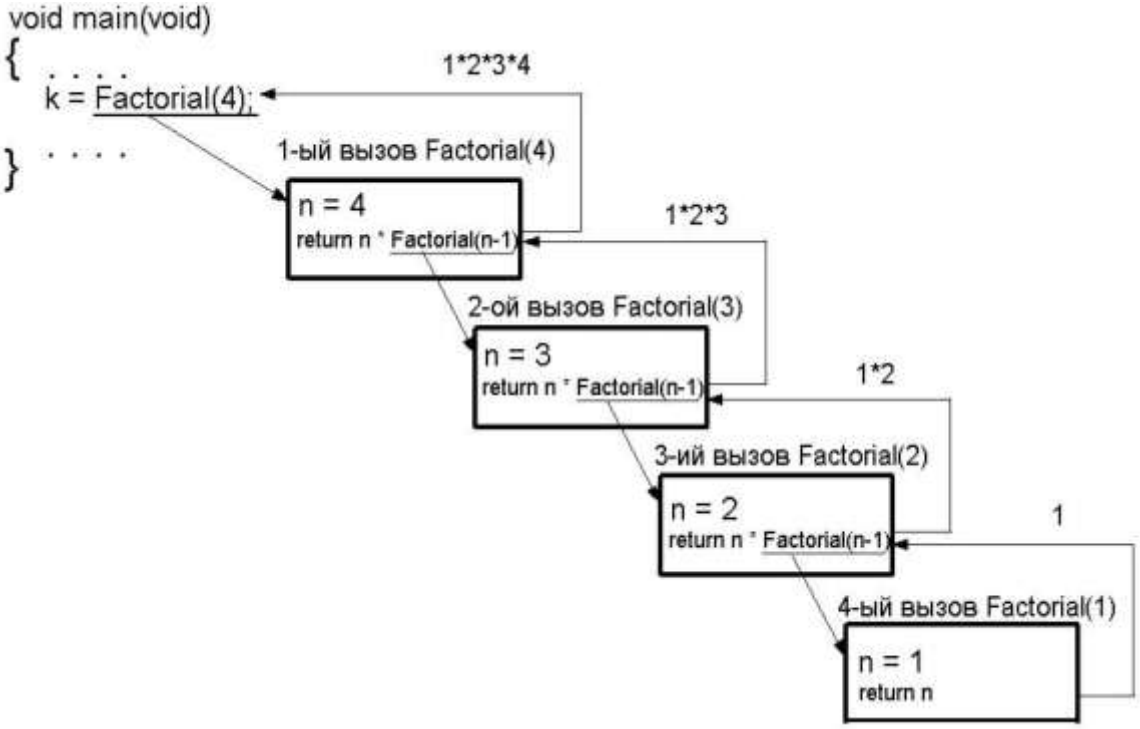

Рис. 4.6. Возврат из вызова метода `Factorial(4)`.

Пример 2. Вычислить *n*-ое число Фибоначчи. Известно, что F_1=1, F_2=1, $F_i = F_{i-1} + F_{i-2}$, т.е. каждый член последовательности чисел Фибоначчи, начиная с третьего, равен сумме двух предыдущих.

```
class Program
{
   // определение метода вычисления n-го числа Фибоначчи
   static uint Fibonachi(uint n)
   {
      // условие выхода из рекурсии
      if (n == 2 || n == 1)
         return 1;
      else
         // рекурсивный вызов функции
         return Fibonachi(n - 2) + Fibonachi(n - 1);
   }

   static void Main(string[] args)
   {
      uint n;
      Console.WriteLine("Введите n:");
      while(true)
      {
         try
         {
            n=uint.Parse(Console.ReadLine());
            break;
         }
         catch (Exception ex)
         {
            Console.WriteLine("Введите корректное
                                                  число.\n");
         }
      }
      uint k=Fibonachi(n);      // первый вызов функции
      Console.WriteLine("F({0})={1}",n,k);}
   }
}
```

Это пример задачи, в которой рекурсивный алгоритм очевиден, но его использование приводит к большим вычислительным расходам и излишним затратам памяти. В теле метода `Fibonachi()` несколько раз осуществляется рекурсивный вызов. Каждый вызов требует выделения памяти. Например, для вычисления F_{10} необходимо вызвать метод `Fibonachi` для вычисления F_9 и F_8. В свою очередь, для вычисления F_9 снова понадобится вызвать метод `Fibonachi()` для

вычисления F_8 и т.д. согласно приведенной ниже схеме.

Из схемы видно, что для некоторых аргументов приходится осуществлять повторные вызовы метода `Fibonachi()`, т. е. происходят лишние вычисления.

$$F_{10} = F_9 + F_8$$

$$F_9 = F_8 + F_7 \qquad\qquad F_8 = F_7 + F_6$$

$$F_8 = F_7 + F_6 \qquad F_7 = F_6 + F_5 \qquad F_7 = F_6 + F_5 \qquad F_6 = F_5 + F_4$$

. . . .

Отметим, что рекурсивные методы могут быть как статическими, так и методами объекта, т.е. вызываться для экземпляра класса.

4.2.3. Конструкторы и деструктор

Специальными методами класса являются конструкторы и деструктор.

Конструктор служит для создания и инициализации объектов класса. В отличие от других методов при описании конструктора не указывается тип возвращаемого значения, а его имя совпадает с именем класса.

```
[модификаторы] имя_класса (список_параметров)
{
   тело конструктора
}
```

В зависимости от вариантов инициализации объекта при создании можно определять несколько конструкторов, отличающихся списком параметров. В этом случае говорят о перегрузке конструкторов. Если в классе явно не определено ни одного конструктора, то он предоставляется по умолчанию. Этот конструктор присваивает нулевые значения всем полям класса.

В качестве модификаторов могут использоваться следующие:

- `static;`
- модификаторы доступа: `public, protected, private, internal.`

Статический конструктор служит для инициализации статических полей класса. Он вызывается один раз при создании первого объекта класса в программе.

Если в классе существуют несколько конструкторов, можно один конструктор вызывать из другого. Это делается с помощью ключевого слова `this` и указания всех параметров вызываемого конструктора. Например, пусть класс `MyClass` имеет два конструктора с прототипами:

```
MyClass (int a);
MyClass (int a, int b, double c);
```

Из второго конструктора можно вызвать первый следующим образом:

```
MyClass (int a, int b, double c):this(a)
{
   // тело конструктора
   .   .   .
}
```

Наличие конструкторов в классе позволяет создавать и инициализировать объекты класса. Объект создается с помощью оператора `new`:

```
MyClass ob  = new MyClass(list_of_params);
```

Поскольку класс относится к ссылочным типам данных, в переменной `ob` будет содержаться ссылка на объект класса `MyClass`. Если в программе производится присваивание

```
MyClass ob1 = ob;
```

то переменные `ob` и `ob1` будут ссылаться на один и тот же объект.

При использовании оператора `new` для создания объекта память для его хранения выделяется динамически. Если объект больше не нужен, память, занимаемая объектом, должна быть освобождена. C# обладает механизмом освобождения ресурсов памяти, называемым ***«сборщик мусора»***. Если к объекту не происходит обращение (в программе не осталось ссылок на объект), этот механизм освобождает память автоматически. Сборщик мусора вызывается периодически в процессе работы программы.

Когда объект использует внешние ресурсы (например, файлы, соединение с другими устройствами, подключение к базам данных и т.д.), для корректного освобождения этих ресурсов может использоваться специальный метод, который называется ***деструктором***. Он вызывается сборщиком мусора при уничтожении объекта. Деструктор в классе может быть только один. Его имя совпадает с именем класса, предваренное символом ‘~’. У деструктора нет возвращаемого значения и нет параметров. Он не имеет модификаторов.

```
~Имя_Класса()
{
    .   .   .
}
```

4.3. Перегрузка операций

При проектировании классов, характеризующих поведение математических объектов, удобно использовать традиционные математические знаки операций для выполнения соответствующих действий. Например, при сложении двух матриц было бы понятней использовать операцию "+", а не вызывать функцию `Summ()` (тем более, что другой программист может назвать эту функцию иным именем). Для таких ситуаций в C# есть очень удобное средство, называемое ***перегрузкой операций***. Фактически многие операции языка C# перегружены изначально. Например, операция "/", примененная к целым числам, является операцией целочисленного деления (25 / 10 = 2), а если хотя бы одно из чисел является вещественным, результатом деления будет вещественное число (25.0 / 10.0 = 2.5). Операция "–" также перегружена: унарный минус меняет

знак выражения на противоположный ($y = -x$), а бинарный выполняет операцию вычитания ($a = b - c$).

В языке C# у программиста имеется возможность задать для собственных типов данных свои методы обработки, закрепленные за обозначением той или иной операции.

При перегрузке операций необязательно придерживаться стандартных процедур выполнения операции. Например, операция `"*"`, обозначающая обычно умножение двух объектов (чисел, матриц, векторов и т.д.), может быть перегружена методом, который вовсе не осуществляет перемножение. Но лучше не изменять смысла перегружаемых операций.

Большинству операций языка C# соответствуют специальные операторные функции, имеющие следующий прототип:

```
[модификаторы]
тип_возвр_значения operator#(список_форм_параметров)
```

Здесь `"#"` – это знак операции C#. Не все операции можно перегружать. К запрещенным для перегрузки операциям относятся `"."`, `"()"`,`"[]"`,`"||"`,`"&&"`,`"?:"`,`"new"`, `"is"`, `"as"`, `"sizeof"`, `"typeof"` и некоторые другие. Существует несколько ограничений, которые следует учитывать при перегрузке операций. Во-первых, нельзя менять приоритет операций. Во-вторых, нельзя изменять число операндов операции. К примеру, операция `"!"` имеет только один операнд, поэтому и ее перегруженная реализация должна быть унарной. В остальном, правила перегрузки операций совпадают с правилами перегрузки функций. Перегруженные операции должны отличаться списками параметров. Например, операция `"*"` для матриц может быть перегружена как метод умножения двух матриц или метод умножения матрицы на число.

В C# операторные функции должны быть статическими, т.е. они являются методами класса, но не методами объекта. Поэтому для бинарных операторов список параметров должен состоять из двух элементов, для унарных – из одного, при этом хотя бы один параметр должен иметь тип того класса, в котором он определен. Также параметры не могут передаваться с модификаторами `ref` или `out`.

Приведем пример перегрузки операции "*" и "~" в классе Matrix:

```
using System;

namespace Matrix
{
   class Matrix
   {
      // элементы матрицы
      double[,] a;

      // конструктор матрицы заданных размеров
      public Matrix(int rows, int cols)
      {
         a = new double[rows, cols];
      }

      // метод заполнения матрицы случайными числами
      public void InputMatrix()
      {
         Random r=new Random();
         for (int i = 0; i < a.GetLength(0); i++)
            for (int j = 0; j < a.GetLength(1); j++)
               a[i, j] = (double)r.Next(100);
      }

      // метод преобразования матрицы в строку
      public override string ToString()
      {
         string s = null;
         for (int i = 0; i < a.GetLength(0); i++, s += "\n")
            for (int j = 0; j < a.GetLength(1); j++)
               s += a[i, j] + " ";
         return s;
      }

      // метод вывода матрицы
      public void OutputMatrix()
      {
         string s = this.ToString();
         Console.WriteLine(s);
      }

      // метод, перегружающий операцию
      // траспонирования матрицы
      static public Matrix operator ~(Matrix m)
      {
         Matrix newM = new Matrix( m.a.GetLength(1),
                                   m.a.GetLength(0));
         for (int i = 0; i < m.a.GetLength(0); i++)
```

```
            for (int j = 0; j < m.a.GetLength(1); j++)
                newM.a[j, i] = m.a[i, j];
        return newM;
    }

    // операция перемножения двух матриц
    static public Matrix operator *(Matrix m1, Matrix m2)
    {
        if(m1.a.GetLength(1) == m2.a.GetLength(0))
        {
            // создание матрицы-результата
            Matrix newM = new Matrix(m1.a.GetLength(0),
                                     m2.a.GetLength(1));

            // заполнение матрицы-результата
            for (int i = 0; i < m1.a.GetLength(0); i++)
                for (int j = 0; j < m2.a.GetLength(1); j++)
                    for(int k = 0; k < m1.a.GetLength(1); k++)
                        newM.a[i,j] += m1.a[i,k] * m2.a[k,j];
            return newM;
        }
        else
        {
            // количество столбцов первой матрицы должно
            // быть равно количеству строк второй матрицы
            throw new Exception("Матрицы таких размеров
                                     перемножать нельзя");
        }
    }

    // операция умножения матрицы на число
    static public Matrix operator *(Matrix m, double d)
    {
        // создание матрицы-результата
        Matrix newM = new Matrix(m.a.GetLength(0),
                                 m.a.GetLength(1));
        // заполнение матрицы-результата
        for (int i = 0; i < m.a.GetLength(0); i++)
            for (int j = 0; j < m.a.GetLength(1); j++)
                newM.a[i,j] = m.a[i,j] * d;
        return newM;
    }

    // операция умножения числа и матрицы
    static public Matrix operator *(double d, Matrix m)
    {
        // создание матрицы-результата
        Matrix newM = new Matrix(m.a.GetLength(0),
                                 m.a.GetLength(1));
        // заполнение матрицы-результата
        newM = m * d;
        return newM;
```

```
        }
        // другие операции и методы
    .     .     .
    }

    class Program
    {
        static void Main(string[] args)
        {
            Matrix x, y, z;
            x = new Matrix(2, 3);
            y = new Matrix(3, 4);

            x.InputMatrix();
            y.InputMatrix();

            x.OutputMatrix();
            y.OutputMatrix();

            z = ~x;          // траспонирование матрицы
            z.OutputMatrix();

            z = x * 5.0;     // умножение матрицы на число
            z.OutputMatrix();

            z = 2.5 * y;     // умножение числа на матрицу
            z.OutputMatrix();

            try
            {
                z = x * y;   // умножение матриц
                z.OutputMatrix();
            }
            catch (Exception ex)
            {
                // перехват исключения, связанного с
                // несоответствием размеров
                // перемножаемых матриц
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Подробнее разберем методы, перегружающие операторы. Оператор "~" будет использоваться для транспонирования матрицы. Этот оператор является унарным, т.е. имеет один операнд. Поэтому в списке параметров метода указывается объект класса `Matrix`.

Результатом является новая матрица, которая создается внутри метода и является транспонированной к исходной.

```
// метод, перегружающий операцию траспонирование матрицы
static public Matrix operator ~(Matrix m)
{
   Matrix newM = new Matrix(m.a.GetLength(1),
                            m.a.GetLength(0));
   for (int i = 0; i < m.a.GetLength(0); i++)
      for (int j = 0; j < m.a.GetLength(1); j++)
         newM.a[j, i] = m.a[i, j];
   return newM;
}
```

Оператор "*" может использоваться с операндами различных типов, например, можно матрицу умножать на матрицу, матрицу умножать на число или число на матрицу. Поэтому эту операцию можно перегрузить несколькими методами, которые будут отличаться списком параметров:

```
// операция перемножения двух матриц
static public Matrix operator *(Matrix m1, Matrix m2)
{
   if(m1.a.GetLength(1) == m2.a.GetLength(0))
   {
      // создание матрицы-результата
      Matrix newM = new Matrix(m1.a.GetLength(0),
                               m2.a.GetLength(1));

      // заполнение матрицы-результата
      for (int i = 0; i < m1.a.GetLength(0); i++)
         for (int j = 0; j < m2.a.GetLength(1); j++)
            for(int k = 0; k < m1.a.GetLength(1); k++)
               newM.a[i,j] += m1.a[i,k] * m2.a[k,j];
      return newM;
   }
   else
   {
      // количество столбцов первой матрицы должно быть
      // равно количеству строк второй матрицы
      throw new Exception("Матрицы таких размеров
                               перемножать нельзя");
   }
}

// операция умножения матрицы на число
static public Matrix operator *(Matrix m, double d)
{
   // создание матрицы-результата
```

```
        Matrix newM = new Matrix(m.a.GetLength(0),
                                 m.a.GetLength(1));
        // заполнение матрицы-результата
        for (int i = 0; i < m.a.GetLength(0); i++)
            for (int j = 0; j < m.a.GetLength(1); j++)
                newM.a[i,j] = m.a[i,j] * d;
        return newM;
    }

    // операция умножения числа и матрицы
    static public Matrix operator *(double d, Matrix m)
    {
        // создание матрицы-результата
        Matrix newM = new Matrix(m.a.GetLength(0),
                                 m.a.GetLength(1));
        // заполнение матрицы-результата
        newM = m * d;
        return newM;
    }
```

Для перегрузки отдельных видов операций существуют особенности, о которых требуется сказать отдельно.

При перегрузке операций сравнения обязательно в качестве типа возвращаемого значения указывается тип `bool`. Кроме того, такие операции перегружаются парно – операция "==" вместе с "!=", "<" вместе с ">", "<=" вместе с ">=".

Класс может содержать методы для осуществления операции ***преобразования*** в другие типы данных или из них. Они реализуются с помощью перегрузки специальных операторов.

Оператор преобразования к типу данных имеет вид:

```
operator имяТипа(Операнд);      // имяТипа – имя оператора
```

Этот оператор не имеет возвращаемого типа, поскольку он совпадает с именем оператора. Данная операция является унарной.

Операция преобразования может быть явной или неявной. Это указывается с помощью ключевых слов `explicit` или `implicit`. Класс может содержать только одно из этих двух преобразований. Например, определим в классе `Matrix` неявное преобразование целого числа в матрицу, результатом которого будет квадратная матрица, размер которой задан этим числом.

```
// операция неявного преобразования целого числа в матрицу
public static implicit operator Matrix(int n)
{
   Matrix m = new Matrix(n, n);
   m.InputMatrix();
   return m;
}
```

Вызов данного метода осуществляется так:

```
Matrix a = 5;     // создание матрицы размера 5 x 5
```

Также определим в классе `Matrix` явное преобразование матрицы в вещественное число, в результате которого будет возвращаться значение определителя матрицы.

```
// операция преобразования матрицы в вещественное число -
// ее определитель
public static explicit operator double(Matrix m)
{
   return m.Determinant();
}
```

Вызов данного метода осуществляется только при явном преобразовании типов:

```
Matrix a(2,2);
a.InputMatrix();
double det = (double)a;
```

Преобразование к логическому типу данных можно осуществить также с помощью перегрузки значений `true` и `false`, которые перегружаются обязательно вместе. Например, в классе `Matrix` эти операции могут определять, является ли квадратная матрица вырожденной (определитель матрицы равен нулю).

```
// операции проверки вырожденности матрицы
public static bool operator true(Matrix m)
{
   if ((double)m == 0.0)
      return false;
   return true;
}
```

```
public static bool operator false(Matrix m)
{
   if ((double)m == 0.0)
      return true;
   return false;
}
```

Имея данные операции, можно объекты класса `Matrix` применять в условных выражениях, т.е. в выражениях, которые принимают значения `true` или `false`:

```
Matrix a(2,2);
a.InputMatrix();
// использование объекта-матрицы в логическом выражении
if (z)
   Console.WriteLine("Матрица невырожденная");
else
   Console.WriteLine("Матрица вырожденная");
```

4.4. Свойства и индексаторы

Свойства и ***индексаторы*** – это методы специального вида, осуществляющие контролируемый доступ к данным. Контролируемый доступ предполагает проверку возможности доступа к значениям или возможности его изменения. Например, если свойство предназначено для управления полем класса, в котором хранится возраст человека, при установке нового значения этого свойства должна быть осуществлена проверка, не является ли новое значение отрицательным или очень большим.

Для определения свойства класса используется синтаксис:

```
[атрибуты] [модификаторы доступа]
тип_возвр_значения имя_свойства
{
   get
   {
      // тело функции получения свойства
      return результат;
   }

   set
   {
      // тело функции установки значения свойства
   }
}
```

Метод установки нового значения (`set`) неявно получает параметр, называемый `value`, в котором хранится это значение.

Для определения индексатора класса используется синтаксис:

```
[атрибуты] [модификаторы доступа]
тип_возвр_значения this[индексы]
{
   get
   {
      // тело функции получения значений по индексам
      return результат;
   }

   set
   {
      // тело функции присваивания значений по индексам
   }
}
```

Вызов индексатора осуществляется по правилу:

```
имя_объекта[индексы];
```

Для свойств и индексаторов обязательным является определение, по крайней мере, одной части – `get` или `set`.

Использование делегатов и событий будет рассмотрено позднее.

4.5. Разработка класса «Квадратное уравнение»

Пусть требуется создать класс для реализации методов решения квадратного уравнения $ax^2+bx+c=0$. Уравнение задается набором коэффициентов (от 1 до 3). Если при создании указывается иное количество коэффициентов, то квадратное уравнение определить нельзя, поэтому выдается предупреждение об ошибке. В классе должны быть предусмотрены средства для решения уравнений, в которых $a=0$, $b=0$ или $c=0$. Тогда уравнение может стать линейным ($0*x^2+5x+2=0$), обратиться в тождество (0 = 0) или стать неразрешимым (6 = 0).

1. Определение полей класса.

Поля класса определяют его структурные свойства, т.е. описывают состав объекта класса. Для определения квадратного уравнения требуется указать его коэффициенты: `a`, `b` и `c`. Также можно ввести в класс поля, характеризующие решение уравнения – количество корней (`count`) и сами корни (максимальное количество корней – 2) – `x1` и `x2`. Эти поля нужны, в частности, чтобы не осуществлять повторное решение уравнения.

```
class Equation
{
   // поля класса
   // константа для обозначения
   // бесконечного количества корней
   const int infinity = Int32.MaxValue;

   // переменные для хранения коэффициентов уравнения
   double a,b,c;

   // поле для хранения количества корней уравнения
   // (-1 означает, что вычисление корней
   // уравнения пока не осуществлялось)
   int count=-1;

   // переменные для хранения корней уравнения
   double x1,x2;
   .   .   .
}
```

Далее будет указываться только код отдельных элементов, которые добавляются к классу `Equation`.

2. Определение конструктора класса.

   Для создания объекта класса `Equation` требуется задать его коэффициенты. Поскольку количество коэффициентов может быть переменным (от 1 до 3), конструктор можно сделать методом с переменным числом параметров (определенным с ключевым словом `params`). Если будет задано недопустимое количество коэффициентов, сгенерируется соответствующее исключение:

```
// конструктор объекта-уравнения
public Equation(params double [] coef)
{
   // в зависимости от количества параметров (длины массива)
   // получаем различные виды уравнений
   switch (coef.Length)
   {
      case 3:
           // квадратное уравнение
           a = coef[0];
           b = coef[1];
           c = coef[2];
           break;
      case 2:
           // линейное уравнение
           a = 0.0;
           b = coef[0];
           c = coef[1];
           break;
      case 1:
           // тождество или неразрешимое уравнение
           a = 0.0;
           b = 0.0;
           c = coef[0];
           break;
      default:
           // генерация исключения при некорректном
           // наборе коэффициентов
           throw new Exception("Данный набор коэффициентов
                 не может определять квадратное уравнение");
   }
}
```

3. Определение методов решения уравнения.
   Для определения разрешимости уравнения сначала должен вызываться метод `Solve()`. В его задачу входит определение порядка уравнения (квадратное – уравнение 2-ого порядка, линейное – уравнение 1-ого порядка, тождество или неразрешимое – уравнение 0-ого порядка).

```
public void Solve()
{
   if (a == 0)
      //линейное уравнение
      if (b == 0)
         // уравнение нулевого порядка
         if (c == 0)
```

```
            {
                // тождество
                count = infinity;
            }
            else
            {
                // неразрешимое уравнение
                count = 0;
            }
        else
            // линейное уравнение
            LinSolve();
    else
        //квадратное уравнение
        QSolve();
}
```

Для решения линейных и квадратных уравнений дополнительно вызываются методы: `QSolve ()` – для уравнений 2-ого порядка, `LinSolve()` – для уравнений 1-ого порядка.

```
// метод решения квадратного уравнения
public void QSolve()
{
    // вычисление дискриминанта
    double disc = b * b - 4 * a * c;

    if (disc < 0.0)
        count = 0;      // вещественных корней нет
    else if (disc == 0.0)
          {
             // уравнение имеет один корень
             count = 1;
             x1 = x2 = -b / (2 * a);
          }
          else
          {
             // уравнение имеет два корня
             count = 2;
             x1 = (-b + Math.Sqrt(disc)) / (2 * a);
             x2 = (-b - Math.Sqrt(disc)) / (2 * a);
          }
}

// метод решения линейного уравнения
public void LinSolve()
{
    count = 1;
    x1 = -c / b;
}
```

4. Определение свойств для доступа к коэффициентам уравнения и получения количества его корней.

   Согласно принципу инкапсуляции данные класса являются закрытыми (`private`). К ним можно обращаться только из методов самого класса. В случае, когда требуется изменить или получить значения тех или иных данных класса следует определять специальные методы доступа к данным (обычно называются `get`- и `set`-методы). Как правило, эти методы оформляются в виде свойств класса.

   Приведем определения свойств для доступа к коэффициентам квадратного уравнения:

```
 // свойства для обеспечения доступа
 // к коэффициентам уравнения
public double A
{
    get { return a; }
    set { a = value; }
 }

 public double B
 {
    get { return b; }
    set { b = value; }
 }

 public double C
 {
    get { return c; }
    set { c = value; }
 }
```

   Для получения количества корней уравнения создадим также свойство `Count`. В отличие от свойств, осуществляющих доступ к коэффициентам уравнения, `Count` доступно только для чтения, так как количество корней должно вычисляться, а не определяться пользователем.

```
// свойство для получения количества корней уравнения
public int Count
{
   get { return count; }
}
```

5. Определение индексатора для получения корней уравнения по номеру.

   Отличием индексаторов от свойств является то, что индексатор осуществляет доступ к отдельному элементу из упорядоченного набора значений. Упорядочение в данном случае понимается как соответствие каждого элемента определенному набору индексов (индексов может быть несколько и они могут задаваться с помощью различных типов данных – целые числа, строки, символы, объекты какого-то класса и пр.).

   В класс `Equation` введем индексатор для получения корня уравнения по его номеру. Очевидно, что в качестве индекса здесь используется целое число (1 или 2), соответствующее номеру корня. В случаях, когда корня с заданным номером не существует или уравнение еще не решено, будет сгенерирована исключительная ситуация. Данный индексатор должен предоставлять доступ только для чтения, поскольку корни уравнения должны вычисляться.

```
// индексатор для получения корня уравнения по его номеру
public double this[int i]
{
   get
   {
      if (count == -1)
         throw new Exception("Уравнение еще не решено");
      if ((count == 1 || count==2) && i == 1)
         return x1;
      if (count == 2 && i == 2)
         return x2;
      throw new Exception("Уравнение не имеет
                                    корня с номером "+i);
   }
}
```

6. Метод распечатки корней уравнения.

   Для того чтобы распечатать корни уравнения, добавим в класс специальный метод:

```
// метод распечатки корней уравнения
public void PrintSolution()
{
    // сообщение определяется значением количества
    // корней уравнения
    switch(count)
    {
        case 2: Console.WriteLine("x1={0}, x2={1}",x1,x2);
                break;
        case 1: Console.WriteLine("x={0}",x1);
                break;
        case 0: Console.WriteLine("Корней нет");
                break;
        case Int32.MaxValue:
                Console.WriteLine("Любое x является решением");
                break;
        default:
                Console.WriteLine("Уравнение еще не решено");
                break;
    }
}
```

7. Пример использования класса `Equation`.
   Простая последовательность действий при создании и решении уравнения такова (Рис.4.7):

```
try
{
    // создание объекта-уравнения
    Equation e = new Equation(2.0,5.0,-9.0);
    // решение уравнения
    e.Solve();
    // распечатка полученного решения
    e.PrintSolution();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

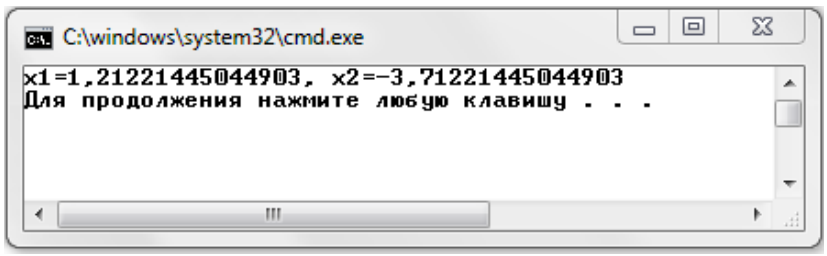

Рис.4.7. Результат решения квадратного уравнения

Сначала создается объект-уравнение, далее для него вызываются метод `Solve()` (решение уравнения) и метод `PrintSolution()` (печать корней). Поскольку в конструкторе может быть сгенерировано исключение, эти три команды помещаются в блок `try`. Соответствующий ему блок `catch` печатает сообщение об ошибке.

Если мы хотим изменить значение какого-то коэффициента уравнения, можно воспользоваться свойствами `A`, `B` или `C`. В следующем фрагменте кода после решения уравнения меняется его коэффициент `B` и уравнение решается заново (Рис.4.8):

```
try
{
   Equation e = new Equation(2.0,5.0,-9.0);
   e.Solve();
   e.PrintSolution();
   // изменение коэффициента B уравнения и его решение
   e.B = -3.0;
   e.Solve();
   e.PrintSolution();
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

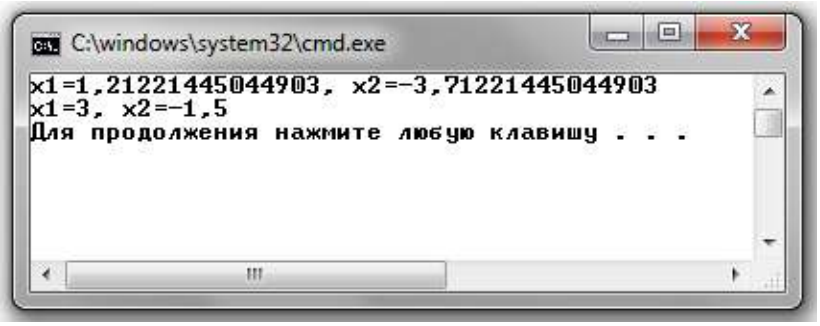

Рис.4.8. Результат решения квадратного уравнения до и после изменения его коэффициентов

Для демонстрации использования индексатора распечатаем корни уравнения без использования функции `PrintSolution()`.

```
Equation e = new Equation(2,5);
e.Solve();
```

```
try
{
   if (e.Count == Int32.MaxValue)
      Console.WriteLine("Корней много");
   // печать первого корня уравнения
   Console.WriteLine("x1 = {0}", e[1]);
   // печать второго корня уравнения
   Console.WriteLine("x2 = {0}", e[2]);
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Глава 5. Функциональные типы в C#

5.1. Делегаты

При выполнении программы машинный код откомпилированной функции загружается в оперативную память компьютера. Адрес, по которому функция будет расположена в памяти, можно использовать для ее вызова. Для этого в языке C# существует специальный тип данных – делегат, значением которого является адрес функции.

Делегат определяется типом возвращаемого значения и списком формальных параметров тех функций, на которые он будет ссылаться. Поэтому адреса различных функций, имеющих одинаковый прототип, могут быть присвоены одной переменной-делегату. Имя функции соответствует ее адресу.

Делегат определяется следующим образом:

```
delegate тип_функции имя_делегата(список_параметров);
```

где `тип_функции` определяет тип возвращаемого значения функции, `список_параметров` – список формальных параметров.

Покажем на примере способы работы с делегатами. В классе `Math` имеются статические методы, вычисляющие значения следующих математических функций:

```
double Sin(double);         // y = sin(x)
double Exp(double);         // y = exp(x)
double Atan(double);        // y = arctg(x)
// и  т.д.
```

Опишем делегат с именем `FunctionDelegate`, который определяет новый тип данных, указывающий на любой из таких методов.

```
delegate double FunctionDelegate (double x);
```

Теперь можно создавать переменные этого типа:

```
FunctionDelegate f;
```

Здесь определяется переменная f как указатель на функцию, которая возвращает значение типа double и имеет один аргумент типа double.

Присвоим переменной f адрес функции вычисления синуса:

```
f = Math.Sin;
```

Таким же образом можно было присвоить переменной f адрес любой из вышеперечисленных функций.

Далее вызывать функцию вычисления синуса можно через переменную f:

```
double y = f(3.1415926);    // вызов функции sin(3.1415926)
```

Делегаты могут выступать в качестве полей класса. Приведем пример использования делегата в классе Equation. В зависимости от типа уравнения мы вызываем соответствующую функцию: QSolve() или LinSolve(). Тип уравнения определяется его коэффициентами на этапе инициализации объекта. Поэтому в этот же момент в переменную-делегат можно записать адрес соответствующей функции решения уравнения.

Выделим три метода решения уравнений – QSolve() для квадратного уравнения, LinSolve() – для линейного уравнения и NullSolve() – для уравнения нулевого порядка. Для любого уравнения будет вызываться одна из этих функций. Функция, которая будет решать конкретное уравнение, будет определяться в конструкторе класса. Указатель на нее будет сохранен в экземпляре делегата с именем solve, который является полем класса Equation. Сам делегат как тип данных определен в пространстве имен приложения (можно определить и внутри класса). Вызов метода решения уравнения будет осуществляться с помощью делегата:

```
Equation e = new Equation(1,2,3);
e.solve();
e.PrintSolution();
```

Далее приведем полный код приложения для решения квадратного уравнения.

```
using System;

namespace DelegateApplication
{
    // определение делегата
    public delegate void Method();

    // класс «Квадратное уравнение»
    class Equation
    {
       const int infinity = Int32.MaxValue;

       double a, b, c;

       int count = -1;
       double x1, x2;

       // определение ссылки на экземпляр делегата
       // для определения способа решения уравнения
       public Method solve=null;

       // конструктор класса
       public Equation(params double[] coef)
       {
          switch (coef.Length)
          {
             case 3:
                  a = coef[0];
                  b = coef[1];
                  c = coef[2];
                  break;
             case 2:
                  a = 0.0;
                  b = coef[0];
                  c = coef[1];
                  break;
             case 1:
                  a = 0.0;
                  b = 0.0;
                  c = coef[0];
                  break;
             default:
                  throw new Exception("Данный набор
                            коэффициентов не может
                             определять
                            квадратное уравнение");
          }
```

```
   // инициализация делегата в зависимости от
   // коэффициентов уравнения
   if (a == 0)
      // линейное уравнение
      if (b == 0)
         solve = NullSolve;
      else
         solve = LinSolve;
   else
      // квадратное уравнение
      solve = QSolve;
}

// метод решения квадратного уравнения
public void QSolve()
{
   double disc = b * b - 4 * a * c;

   if (disc < 0.0)
      count = 0;
   else if (disc == 0.0)
        {
           count = 1;
           x1 = -b / (2 * a);
           x2 = x1;
        }
        else
        {
           count = 2;
           x1 = (-b + Math.Sqrt(disc)) / (2 * a);
           x2 = (-b - Math.Sqrt(disc)) / (2 * a);
        }
}

// метод решения линейного уравнения
public void LinSolve()
{
   count = 1;
   x1 = -c / b;
}

// метод решения уравнения 0-ого порядка
public void NullSolve()
{
   if (c == 0)
      count = infinity;
   else
      count = 0;
}
```

```
        // метод печати результата решения уравнения
        public void PrintSolution()
        {
            switch (count)
            {
                case 2:
                    Console.WriteLine("x1={0}, x2={1}", x1, x2);
                    break;
                case 1:
                    Console.WriteLine("x={0}", x1);
                    break;
                case 0:
                    Console.WriteLine("Корней нет");
                    break;
                case Int32.MaxValue:
                    Console.WriteLine("Любое x является
                                                    решением");
                    break;
                default:
                    Console.WriteLine("Уравнение еще не
                  решено");
                    break;
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Equation e = new Equation(1,2,3);
                // вызов метода решения уравнения через делегат
                e.solve();
                e.PrintSolution();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
```

Делегаты могут также использоваться и в качестве параметров других методов. Проиллюстрируем это на примере решения следующей задачи: найти корень уравнения $F(x)=0$ методом деления отрезка пополам, предусмотрев возможность использования в качестве $F(x)$ различных непрерывных математических функций.

Создадим класс `Roots` для вычисления корней уравнения. В него добавим статический метод `Root()`, который находит корень уравнения методом деления отрезка пополам. Параметрами метода являются концы отрезка, на котором осуществляется поиск корня, задаваемая точность решения и делегат, указывающий на метод вычисления $F(x)$.

```
// решение уравнения F(x) = 0 на отрезке [a; b]
// с точностью eps>0 методом деления отрезка пополам
public static double Root(double a, double b,
                          double eps, FunctionDelegate f)
{
   // корень гарантировано существует,
   // если на концах отрезка
   // функция принимает значения различных знаков.
   if(f(a) * f(b) > 0)
      throw new Exception("Возможно, корней на этом
                          отрезке не существует");
   // обеспечиваем выполнение условия:
   // a - левый конец отрезка, b - правый
   if(a > b)
   {
      double t = a;
      a = b;
      b = t;
   }
   // цикл метода деления отрезка пополам
   while(b - a > eps)
   {
      double c = (a + b) / 2;
      if(f(c) == 0)
         return c;
      if(f(a)*f(c) < 0)
         b = c;
      else
         a = c;
   }
   return (a + b) / 2;
}
```

Как и все в C#, делегаты являются классами. В пространстве имен `System` определены два класса: `Delegate` и `MulticastDelegate`. При определении делегата в качестве базового компилятором назначается один из этих классов в зависимости от типа возвращаемого значения: для типа `void` базовым является `MulticastDelegate`, для остальных типов –

Delegate. Существенным отличием этих двух типов делегатов является возможность сохранения в одном делегате типа MulticastDelegate указателей сразу на несколько функций. Эти указатели хранятся в виде списка, который называют ***списком вызова***. При обращении к такому делегату функции будут выполняться по очереди в порядке их помещения в список вызова.

Добавить функцию в список вызова можно с помощью статического метода Combine(), который сцепляет списки вызовов или массивы делегатов (метод получает в качестве параметра массив делегатов), или двух делегатов (метод получает в качестве параметров два делегата):

```
Delegate Combine(Delegate[]);
Delegate Combine(Delegate, Delegate);
```

Также для этих целей можно использовать перегруженную операцию "+=". Второй операнд операции представляет собой делегат или функцию, которая добавляется в список вызова исходного делегата.

Обратную операцию – удаление метода из списка вызова выполняют статический метод Remove() и операция "-=".

```
Delegate Remove(Delegate, Delegate);
```

Удаляется последнее вхождение метода или списка вызовов делегата из списка вызовов другого делегата.

Продемонстрируем использование комбинированных делегатов, модифицировав класс Equation. Напомним, что комбинированный делегат позволяет по цепочке вызывать несколько методов типа void с одинаковыми параметрами. Решение уравнения может быть представлено в виде последовательного выполнения трех действий – определения типа уравнения, собственно решения и печати результата. Каждое из этих действий оформляется в отдельный метод (DefineTypeEquation(), Solve(), PrintSolution()). Эти методы формируют список вызова комбинированного делегата SolveEquation.

Отметим, что указанные методы нецелесообразно использовать по отдельности. Например, метод `Solve()` не может выбрать конкретный метод решения, пока не определен тип решаемого уравнения, печать результата бессмысленна, если уравнение еще не решено. Поэтому решение осуществляется с помощью вызова делегата, который комбинирует эти три метода в нужной последовательности. Сами методы при этом могут быть закрыты для пользователя (`private`). Список вызова делегата формируется в конструкторе объекта класса `Equation`.

Метод `DefineTypeEquation()` в зависимости от коэффициентов уравнения формирует значение переменной `type`, которая определяет тип уравнения и добавляется в класс. Далее переменная `type` используется для определения способа решения уравнения в методе `Solve()`.

Далее приводим код программы.

```
using System;

namespace CombineDelegateApplication
{
   // комбинированный делегат
   public delegate void Method();

   class Equation
   {
      const int infinity = Int32.MaxValue;
      double a, b, c;

      int count = -1;
      double x1, x2;

      // тип уравнения -  type = 0 - уравнение 0-го порядка,
      //                  type = 1 - линейное уравнение
      //                  type = 2 - квадратное уравнение
      int type = -1;

      // комбинированный делегат для определения процесса
      // решения уравнения
      public Method SolveEquation = null;

      // конструктор класса
      public Equation(params double[] coef)
      {
         switch (coef.Length)
         {
```

```
            case 3:
                a = coef[0];
                b = coef[1];
                c = coef[2];
                break;
            case 2:
                a = 0.0;
                b = coef[0];
                c = coef[1];
                break;
            case 1:
                a = 0.0;
                b = 0.0;
                c = coef[0];
                break;
            default: throw new Exception("Данный набор
                      коэффициентов не может определять
                      квадратное уравнение");
        }


        // формирование списка вызова для решения уравнения
        // сначала определяется тип уравнения, далее
        // уравнение решается, в конце печатается результат
        SolveEquation += DefineTypeEquation;
        SolveEquation += Solve;
        SolveEquation += PrintSolution;
    }

    // метод определения типа уравнения
    void DefineTypeEquation()
    {
        if (a == 0)
            if (b == 0)
                // уравнение 0-ого порядка
                type = 0;
            else
                // линейное уравнение
                type = 1;
        else
            // квадратное уравнение
            type = 2;
    }

    // метод определения способа решения
    // в зависимости от типа уравнения
    void Solve()
    {
        switch (type)
        {
            case 0: NullSolve(); break;
            case 1: LinSolve(); break;
```

```
            case 2: QSolve(); break;
        }
    }

    // решение квадратного уравнения
    void QSolve()
    {
        double disc = b * b - 4 * a * c;

        if (disc < 0.0)
            count = 0;
        else if (disc == 0.0)
                {
                    count = 1;
                    x1 = -b / (2 * a);
                    x2 = x1;
                }
                else
                {
                    count = 2;
                    x1 = (-b + Math.Sqrt(disc)) / (2 * a);
                    x2 = (-b - Math.Sqrt(disc)) / (2 * a);
                }
    }

    // решение линейного уравнения
    void LinSolve()
    {
        count = 1;
        x1 = -c / b;
    }

    // решение уравнения 0-ого порядка
    void NullSolve()
    {
        if (c == 0)
            count = infinity;
        else
            count = 0;
    }

    // метод печати результата решения уравнения
    void PrintSolution()
    {
        switch (count)
        {
            case 2:
                Console.WriteLine("x1={0}, x2={1}", x1, x2);
                break;
            case 1:
                Console.WriteLine("x={0}", x1);
                break;
```

```
                case 0:
                    Console.WriteLine("Корней нет");
                    break;
                case Int32.MaxValue:
                    Console.WriteLine ("Любое x является
                                                решением");
                    break;
                default:
                    Console.WriteLine("Уравнение еще не решено");
                    break;
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Equation e = new Equation(1, -2, -1);
                // использование комбинированного делегата
                // для решения уравнения
                e.SolveEquation();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

5.2. События

Чаще всего, событие – это переход объекта из одного состояния в другое, т.е. изменение его структурных характеристик. В этих случаях бывает необходимо предусмотреть ту или иную реакцию объектов на произошедшие изменения. Это реализуется посредством определения ***обработчиков событий*** – специальных функций, которые вызываются автоматически при наступлении события. Функции-обработчики могут быть как элементами класса, в объекте которого произошло событие, так и методами других классов. Параметры функции-обработчика могут передавать информацию, которая характеризует наступившее событие, чтобы можно было использовать эти данные для корректной обработки.

В настоящее время все приложения, особенно построенные на основе графического пользовательского интерфейса, используют механизм обработки событий. Например, в текстовом редакторе Word необходимо сохранить созданный документ. Чтобы это выполнить, требуется нажать кнопку на панели инструментов или выбрать пункт меню «Файл –> Сохранить». Для кнопок панели инструментов и пунктов меню предусмотрены события нажатия кнопки и выбора пункта меню соответственно. Поскольку реакция программы на эти события должна быть одинаковой, обработчик для них будет единственный. Несмотря на то, что событие было вызвано объектом-кнопкой или пунктом меню, обработчик должен принадлежать объекту-документу, при этом имя файла, в который должно быть произведено сохранение, передается в функцию-обработчик как параметр.

Создание событий в C# происходит в следующем порядке. Некоторый класс объявляет событие, которое он может инициировать. Событие реализуется в виде делегата, который объявляется в классе с помощью ключевого слова `event`. Любые классы, в том числе и класс-инициатор, могут назначить обработчики на это событие. Назначение обработчика представляет собой добавление нужного метода в список вызова событийного делегата.

Покажем использование событий на примере класса `Equation`. Допустим, надо отслеживать событие изменения коэффициентов квадратного уравнения – при изменении любого коэффициента требуется решить уравнение заново. Для этого:

- в приложении объявляем делегат, определяющий вид обработчиков события:

```
public delegate void EquationDelegate(Equation e);
```

- в классе `Equation` объявляем событие `ReplaceCoefEvent` на основе делегата `EquationDelegate`:

```
public event EquationDelegate ReplaceCoefEvent;
```

- добавляем в класс `Program` метод обработки события `ReSolve()`. Его вид соответствует делегату `EquationDelegate`, объявленному в приложении. Задача этого метода – заново решить

уравнение, передаваемое в качестве параметра и распечатать новое решение:

```
static void ReSolve(Equation e)
{
   e.Solve();
   e.PrintSolution();
}
```

- при создании объекта класса `Equation` (например, в функции `Main()` класса `Program`) его событию `ReplaceCoefEvent` назначается обработчик:

```
Equation e = new Equation(2, 5);
e.ReplaceCoefEvent += ReSolve;
```

- в методы класса `Equation`, которые изменяют коэффициенты уравнения, добавим инициацию события (вызов обработчиков этого события через делегат):

```
// свойство доступа к коэффициенту A уравнения
public double A
{
   get
   {
      return a;
   }

   set
   {
      a = value;
      // вызов обработчиков события изменения коэффициентов
      if (ReplaceCoefEvent != null)
         ReplaceCoefEvent(this);
   }
}
```

Аналогичную инициацию события добавляем в свойства `B` и `C`.

Программный код приложения приводится далее:

```
using System;

namespace EventApplication
{
   // объявление событийного делегата
   public delegate void EquationDelegate(Equation e);

   public class Equation
   {
      const int infinity = Int32.MaxValue;

      double a, b, c;

      int count = -1;
      double x1, x2;

      // объявление события изменения
      // коэффициентов уравнения
      public event EquationDelegate ReplaceCoefEvent;

      // свойство доступа к коэффициенту A уравнения
      public double A
      {
         get
         {
            return a;
         }
         set
         {
            a = value;
            // инициация события - вызов его обработчиков
            if (ReplaceCoefEvent != null)
               ReplaceCoefEvent(this);
         }
      }

      // свойство доступа к коэффициенту B уравнения
      public double B
      {
         get
         {
            return b;
         }
         set
         {
            b = value;
            // инициация события - вызов его обработчиков
            if (ReplaceCoefEvent != null)
               ReplaceCoefEvent(this);
         }
      }
```

```
// свойство доступа к коэффициенту C уравнения
public double C
{
   get
   {
      return c;
   }
   set
   {
      c = value;
      // инициация события - вызов его обработчиков
      if (ReplaceCoefEvent != null)
         ReplaceCoefEvent(this);
   }
}

// конструктор класса
public Equation(params double[] coef)
{
   switch (coef.Length)
   {
      case 3:
         a = coef[0]; b = coef[1]; c = coef[2];
         break;
      case 2:
         a = 0.0; b = coef[0]; c = coef[1];
         break;
      case 1:
         a = 0.0; b = 0.0; c = coef[0];
         break;
      default:
         throw new Exception("Данный набор
               коэффициентов не может определять
               квадратное уравнение");
   }
}

// метод решения квадратного уравнения
public void QSolve()
{
   double disc = b * b - 4 * a * c;

   if (disc < 0.0)
      count = 0;
   else if (disc == 0.0)
        {
           count = 1;
           x1 = -b / (2 * a);
           x2 = x1;
        }
        else
        {
```

```
                count = 2;
                x1 = (-b + Math.Sqrt(disc)) / (2 * a);
                x2 = (-b - Math.Sqrt(disc)) / (2 * a);
            }
    }

    // метод решения линейного уравнения
    public void LinSolve()
    {
        count = 1;
        x1 = -c / b;
    }

    // метод определения типа уравнения и его решения
    public void Solve()
    {
        if (a == 0)
            if (b == 0)
                if (c == 0)
                    count = infinity;
                else
                    count = 0;
            else
                LinSolve();
        else
            QSolve();
    }

    // метод печати результата решения уравнения
    public void PrintSolution()
    {
        switch (count)
        {
            case 2:
                Console.WriteLine("x1={0}, x2={1}", x1, x2);
                break;
            case 1:
                Console.WriteLine("x={0}", x1);
                break;
            case 0:
                Console.WriteLine("Корней нет");
                break;
            case Int32.MaxValue:
                Console.WriteLine ("Любое x является
                                                    решением");
                break;
            default:
                Console.WriteLine("Уравнение еще не решено");
                break;
        }
    }
}
```

```
    class Program
    {
        // обработчик события изменения
        // коэффициентов уравнения
        static void ReSolve(Equation e)
        {
            e.Solve();
            e.PrintSolution();
        }

        static void Main(string[] args)
        {
            try
            {
                Equation e = new Equation(2, -6, 4);
                // назначение обработчика события уравнения e
                e.ReplaceCoefEvent += ReSolve;
                e.Solve();
                e.PrintSolution();
                // при изменении коэффициента возникает событие,
                // вызывается его обработчик,который решает
                // уравнение заново
                // и печатает его корни
                 e.B = -5;
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

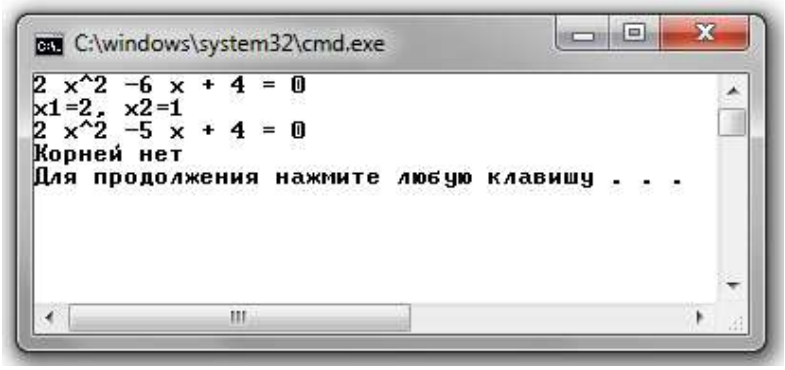

Рис. 5.1. Результаты работы программы
решения квадратного уравнения с применением событий

В среде *.NET Framework* определен универсальный вид описания обработчиков событий. Событийный делегат должен принимать два аргумента: объект-инициатор события и объект с дополнительной информацией о событии, который должен быть производным от класса

EventArgs (пространство имен System). Через второй параметр можно передать дополнительную информацию о возникшем событии.

Приведем изменения, которые следует внести в предыдущий пример, чтобы определение события соответствовало рекомендуемому стилю:

- создадим класс MessageEventArgs – наследник класса EventArgs:

```
// класс для передачи дополнительной информации о событии
public class MessageEventArgs : EventArgs
{
   public string message;
   public MessageEventArgs(string s)
   {
      message = s;
   }
}
```

Поле этого класса message будет содержать сообщение о возникшем событии.

- В приложении изменим объявление событийного делегата:

```
public delegate void EquationDelegate
                     (object src, MessageEventArgs args);
```

- В классе Program изменим обработчик события – метод ReSolve():

```
static void ReSolve(object src, MessageEventArgs args)
{
   // преобразование первого параметра к типу Equation
   Equation e = src as Equation;
   // вывод сообщения о возникшем событии
   Console.WriteLine(args.message);
   e.Solve();
   e.PrintSolution();
}
```

- В классе Equation изменяем инициацию события:

```
public double B
{
    get
    {
        return b;
    }
    set
    {
        b = value;
        if (ReplaceCoefEvent != null)
            ReplaceCoefEvent(this,
             new MessageEventArgs("Изменение коэффициента B"));
    }
}
```

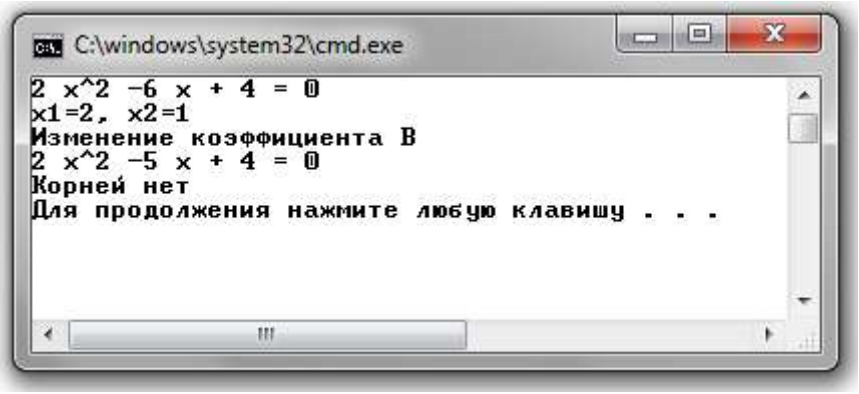

Рис. 5.2. Результаты работы программы решения квадратного уравнения с измененным способом обработки событий

Такой способ определения и обработки событий используется во всех библиотечных классах среды *.NET Framework*, особенно часто – при разработке Windows-приложений.

Глава 6. Наследование и полиморфизм позднего связывания

6.1. Наследование

Напомним, что ***наследование*** – это механизм, который позволяет создавать новые классы на основе существующих, используя их структурные и поведенческие свойства – поля, методы, свойства и пр.

Рассмотрим далее использование этого принципа на примере классов прямоугольной и квадратной матриц (`Matrix` – базовый класс, `QMatrix` – производный класс). Объект класса `Matrix` определяется двумерным массивом ее элементов. Поведенческие свойства класса определяются операциями матричного исчисления.

Так как квадратная матрица есть частный случай прямоугольной, структурные и поведенческие свойства класса «Квадратная матрица» (`QMatrix`) будут идентичны свойствам класса `Matrix`. Поэтому реализуем класс «Квадратная матрица» как наследник класса `Matrix`, добавив в него методы, специфичные для квадратной матрицы (вычисление определителя и получение обратной матрицы).

Приведем исходный код класса `Matrix`.

```
class Matrix
{
   // массив элементов матрицы
   double[,] a;

   // конструктор матрицы с указанием ее размеров
   public Matrix(int rows, int cols)
   {
      a = new double[rows, cols];
   }

   // метод заполнения матрицы случайными числами
   public void InputMatrix()
   {
      int rows = Rows, cols = Cols;
      Random r = new Random();
      for (int i = 0; i < rows; i++)
         for (int j = 0; j < cols; j++)
            a[i, j] = (double)r.Next(100);
```

```
}


// метод вывода матрицы
public void OutputMatrix()
{
   int rows = Rows, cols = Cols;
   for (int i = 0; i < rows; i++)
   {
      for (int j = 0; j < cols; j++)
         Console.Write("{0, 4:f2} ", a[i, j]);
      Console.WriteLine();
   }
}


// свойство получения количества строк матрицы
public int Rows
{
   get { return a.GetLength(0); }
}

// свойство получения количества столбцов матрицы
public int Cols
{
   get { return a.GetLength(1); }
}


// индексатор для доступа к элементам матрицы
public double this[int i, int j]
{
   get
   {
      int rows = Rows, cols = Cols;
      // проверяется корректность индексов
      if (i < 0 || i >= rows || j < 0 || j >= cols)
         throw new Exception("Индексы выходят из
                                                диапазона");
      return a[i, j];
   }
   set
   {
      int rows = Rows, cols = Cols;
      // проверяется корректность индексов
      if (i < 0 || i >= rows || j < 0 || j >= cols)
         throw new Exception("Индексы выходят из
                                                диапазона");
      a[i, j] = value;
   }
}
```

```
// операция сложения двух матриц
static public Matrix operator +(Matrix m1, Matrix m2)
{
   int rows1 = m1.Rows, cols1 = m1.Cols,
         rows2 = m2.Rows, cols2=m2.Cols;
   // размеры матриц должны совпадать
   if (rows1 != rows2 || cols1 != cols2)
      throw new Exception ("Матрицы таких размеров
                            складывать нельзя");
   Matrix newM = new Matrix(rows1, cols1);
   for (int i = 0; i < rows1; i++)
      for (int j = 0; j < cols1; j++)
         newM[j, i] = m1[i, j] + m2[i,j];
   return newM;
}

// метод, перегружающий операцию траспонирование матрицы
static public Matrix operator !(Matrix m)
{
   int rows = m.Rows, cols = m.Cols;
   Matrix newM = new Matrix(cols, rows);
   for (int i = 0; i < rows; i++)
      for (int j = 0; j < cols; j++)
         newM[j, i] = m[i, j];
   return newM;
}

// операция перемножения двух матриц
static public Matrix operator *(Matrix m1, Matrix m2)
{
   int rows1 = m1.Rows, cols1 = m1.Cols,
       rows2 = m2.Rows, cols2=m2.Cols;
   if (cols1 == rows2)
   {
      // создание матрицы-результата
      Matrix newM = new Matrix(rows1, cols2);
      // заполнение матрицы-результата
      for (int i = 0; i < rows1; i++)
         for (int j = 0; j < cols2; j++)
            for (int k = 0; k < cols1; k++)
               newM[i, j] += m1[i, k] * m2[k, j];
      return newM;
   }
   else
   {
      // количество столбцов первой матрицы должно
      // быть равно количеству строк второй матрицы
      throw new Exception ("Матрицы таких размеров
                            перемножать нельзя");
   }
}
```

```
    // операция перемножения матрицы на число
    static public Matrix operator *(Matrix m, double d)
    {
        // создание матрицы-результата
        int rows = m.Rows, cols = m.Cols;
        Matrix newM = new Matrix(rows, cols);
        // заполнение матрицы-результата
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                newM[i, j] = m[i, j] * d;
        return newM;
    }

    // операция перемножения числа на матрицу
    static public Matrix operator *(double d, Matrix m)
    {
        int rows = m.Rows, cols = m.Cols;
        // создание матрицы-результата
        Matrix newM = new Matrix(rows, cols);
        // заполнение матрицы-результата
        newM = m * d;
        return newM;
    }
}
```

В C# базовый класс при наследовании может быть только один. При определении производного класса он указывается после имени производного класса через ":":

```
[атрибуты] [модификаторы] class ChildClass: ParentClass
{
    .   .   .
}
```

Например, для класса «Квадратная матрица» определение выглядит так:

```
class QMatrix : Matrix
{
    .   .   .
}
```

С помощью принципа наследования в C# все классы образуют единую иерархию классов (типов объектов), вершиной которой является класс `object`, т.е. класс `object` является базовым по умолчанию для всех других классов.

Класс-наследник содержит все элементы базового класса. Доступ к ним из дочернего класса возможен, только если они определены с модификаторами `protected` и `public`. `Private`-элементы базового класса в классе-наследнике недоступны. Поэтому для обеспечения доступа к закрытым полям базового класса из производного, их объявление должно предваряться модификатором `protected`.

```
class Matrix
{
   protected double[,] a;
   .   .   .
}
```

Создание объекта дочернего класса производится с помощью конструктора, при этом сначала происходит создание его базовой части посредством вызова конструктора базового класса. Для такого вызова используется специальный синтаксис, называемый инициализацией в заголовке:

```
public ChildClass(список_параметров1):
                                    base(список_параметров2)
{
   .   .   .
}
```

где `base(список_параметров2)` – вызов конструктора базового класса.

Если вызов конструктора базового класса не указывается, автоматически осуществляется вызов конструктора базового класса, не имеющего параметров. Если ни одного конструктора в базовом классе нет, то конструктор без параметров будет предоставлен компилятором. Тогда вызов конструктора базового класса будет происходить автоматически. Если же в базовом классе имеются только конструкторы с параметрами, возникнет ошибка. Для ее исправления требуется определить в базовом классе конструктор без параметров.

Заметим, что конструктор производного класса должен инициализировать как базовую компоненту, так и собственную. Поэтому параметры конструктора производного класса

(список_параметров1) содержат данные для инициализации обеих компонент. Далее параметры, инициализирующие структурные свойства базового класса передаются его конструктору (список_параметров2), а конструктор производного класса инициализирует только собственную часть.

В конструкторе производного класса QMatrix осуществляется вызов конструктора класса Matrix, в который передается количество строк и столбцов создаваемой матрицы (в данном случае они совпадают):

```
class QMatrix : Matrix
{
   public QMatrix(int rows): base(rows, rows)
   {
        .    .    .
   }
   .    .    .
}
```

Класс QMatrix расширяет базовый класс путем добавления новых методов – вычисления определителя и обратной матрицы. Для этого требуется вычислить миноры исходной матрицы. Поэтому в класс QMatrix добавим метод формирования подматрицы, полученной из исходной вычеркиванием заданной строки и заданного столбца. Как известно, определитель такой подматрицы является одним из миноров исходной матрицы.

Согласно теореме Лапласа определитель матрицы Q ($\det(Q)$) равен сумме произведений элементов строки (столбца) на их алгебраические дополнения:

$$\det(Q) = q_{i1}A_{i1} + q_{i2}A_{i2} + \ldots + q_{in}A_{in},$$

где $A_{ij} = (-1)^{i+j} M_{ij}$, M_{ij} – дополнительный минор элемента q_{ij} (определитель матрицы, полученной вычеркиванием из исходной строки с номером i и столбца с номером j).

Согласно этой формуле для вычисления определителя n-ого порядка требуется вычислить n определителей $(n-1)$-ого порядка. Поэтому метод вычисления определителя матрицы будет рекурсивным.

Для вычисления обратной матрицы для матрицы Q требуется найти матрицу алгебраических дополнений (A_{ij}) элементов, транспонировать ее и разделить на значение $\det(Q)$. Если матрица является вырожденной (ее определитель равен нулю), обратной матрицы не существует. В этом случае генерируется исключение.

Таким образом, определение класса `QMatrix` станет таким:

```
class QMatrix : Matrix
{
    // конструктор производного класса
    public QMatrix(int rows): base(rows, rows)
    {
    }

    // метод получения подматрицы вычеркиванием заданных
    // строки и столбца
    QMatrix SubMatrix(int i1, int j1)
    {
        int rows = Rows;
        // матрица-результат имеет порядок
        // на 1 меньше исходной
        QMatrix temp = new QMatrix(rows - 1);
        // формируем новую матрицу, игнорируя
        // строку с номером i1 и столбец с номером j1
        for(int i = 0; i < i1; i++)
        {
            for(int j = 0;j < j1; j++)
                temp[i,j] = a[i,j];
            for (int j = j1 + 1; j < rows; j++)
                temp[i,j - 1] = a[i,j];
        }
        for (int i = i1 + 1; i < rows; i++)
        {
            for(int j = 0; j < j1; j++)
                temp[i - 1, j] = a[i, j];
            for (int j = j1 + 1; j < rows; j++)
                temp[i - 1, j - 1] = a[i,j];
        }
        return temp;
    }

    // метод вычисления определителя матрицы
    public double Determinant()
    {
        double det = 0;
        int rows = Rows;
        // определитель 1-ого порядка совпадает
        // с единственным элементом матрицы
```

```
        if(rows == 1)
            return a[0,0];
        QMatrix temp = new QMatrix(rows - 1);
        // раскладываем определитель по 0-ой строке
        for (int j = 0; j < rows; j++)
        {
            // получаем матрицу для вычисления
            // минора элемента a0j
            temp = SubMatrix(0, j);
            // добавляем очередное произведение элемента
            // на его алгебраическое дополнение
            if(j % 2 == 0)
                det += temp.Determinant() * a[0,j];
             else
                det -= temp.Determinant() * a[0,j];
        }
        return det;
    }

    // метод получения обратной матрицы
    public static QMatrix operator ~(QMatrix m)
    {
        int rows = m.Rows;
        QMatrix res= new QMatrix(rows);
        // вычисление определителя матрицы
        double det = m.Determinant();
        // если матрица вырожденная,
        // обратной матрицы не существует
        if(det == 0)
            throw new Exception("Матрица вырожденная");
        // вычисление транспонированной матрицы
        // алгебраических дополнений
        QMatrix temp = new QMatrix(rows - 1);
        int z;
        for(int i = 0; i < rows; i++)
        {
            z = i%2==0 ? 1 : -1;
            for(int j = 0; j < rows; j++)
            {
                temp = m.SubMatrix(i, j);
                res[j,i] = z * temp.Determinant() / det;
                z = -z;
            }
        }
        return res;
    }
}
```

В новых методах используются элементы базового класса – сама матрица (поле `a`), свойство `Rows` и индексатор доступа к элементам матрицы.

Продемонстрируем использование новых методов класса:

```
static void Main(string[] args)
{
    try
    {
        QMatrix x, y, z;
        x = new QMatrix(2);
        x.InputMatrix();
        x.OutputMatrix();
        // вычисление определителя квадратной матрицы
        Console.WriteLine("Определитель: " +
                                        x.Determinant());
        // получение обратной матрицы
        Console.WriteLine("Обратная матрица:");
        z = ~x;
        z.OutputMatrix();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

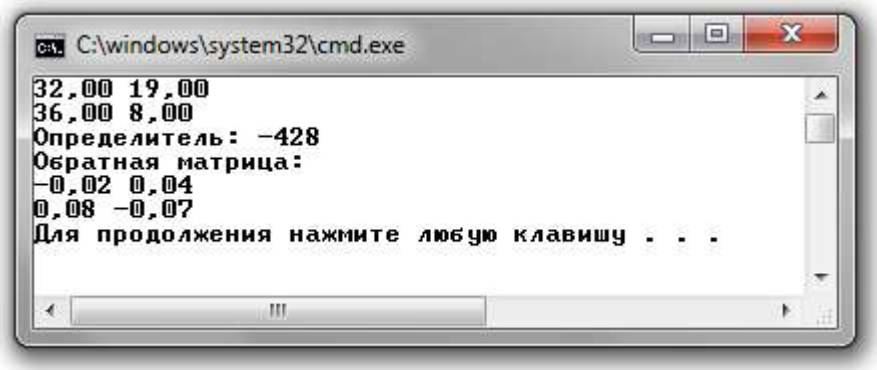

Рис.6.1. Результаты работы программы
демонстрации класса «Квадратная матрица»

Заметим, что в данном примере использовались и методы базового класса для объекта производного класса – методы `InputMatrix()` и `OutputMatrix()`.

Для проверки правильности вычисления обратной матрицы можно было бы воспользоваться операцией перемножения матриц, которая перегружена в базовом классе. В результате перемножения должна получаться единичная матрица.

```
  .    .    .
// получение обратной матрицы для x
z = ~x;
z.OutputMatrix();
// проверка, что z – обратна к x.
// Матрица y должна быть единичной
y = x * z;
y.OutputMatrix();
  .    .    .
```

Однако в этом случае при компиляции возникнет ошибка:

```
Cannot implicitly convert type 'Matrix' to 'QMatrix'. An
explicit conversion exists (are you missing a cast?)
```

Эта ошибка связана с невозможностью преобразования объекта базового класса к производному типу. При использовании операции "*" происходят два вида преобразования типов:

- При передаче параметров (операнды – квадратные матрицы `x` и `z`, преобразуются к типу базового класса). Это преобразование происходит без ошибок, поскольку объект производного класса принадлежит одновременно базовому классу. Это называют принципом подставимости.
- При возвращении операцией матрицы-результата и сохранении результата в объекте `QMatrix` должно производиться обратное преобразование – из `Matrix` в `QMatrix`. Это преобразование невозможно, поэтому и возникает ошибка.

Для исправления этой ошибки нужно добавить в базовый класс `Matrix` специальный метод `ToQMatrix()`, который осуществляет преобразование из типа `Matrix` в `QMatrix`:

```
// метод преобразования объекта класса Matrix в QMatrix
public QMatrix ToQMatrix()
{
   int rows = Rows;
   QMatrix qm = new QMatrix(rows);
   for (int i = 0; i < rows; i++)
      for (int j = 0; j < rows; j++)
         qm[i, j] = a[i, j];
   return qm;
}
```

В этом методе создается объект производного класса и его элементам присваиваются значения из элементов объекта базового класса.

```
  .    .    .
z = ~x;
z.OutputMatrix();

Console.WriteLine("Проверка правильности
                              обратной матрицы:");
y = (x * z).ToQMatrix();
y.OutputMatrix();
  .    .    .
```

После выполнения приведенного выше кода результат будет следующим:

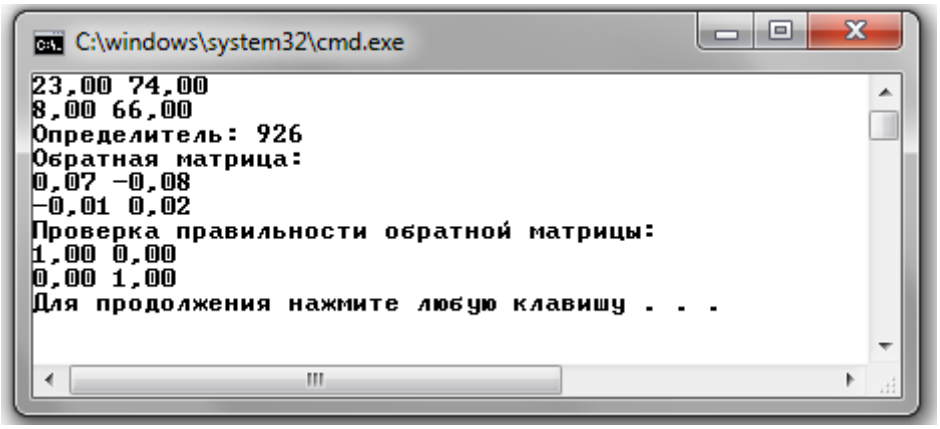

Рис.6.2. Результаты работы программы с проверкой правильности вычисления обратной матрицы

Кроме новых данных и методов, в производном классе могут быть переопределены унаследованные данные и методы. Переопределение означает, что в производный класс добавляются элементы, которые имеют имя, одинаковое с элементами базового класса. Таким образом, производный класс будет содержать два элемента с одним и тем же именем: унаследованный от базового класса и собственный, который скрывает базовый элемент. Объект производного класса будет обращаться к переопределенному элементу. Тем не менее, остается возможность обращения к элементу базового класса – его имя предваряется ключевым словом `base`:

```
base.имя_поля или base.имя_метода(параметры)
```

В C# существует возможность запрета наследовать класс. Это осуществляется с помощью указания модификатора `sealed` в списке модификаторов класса.

6.2. Виртуальные функции и абстрактные классы

Использование ***виртуальных функций*** в программах является одним из способов реализации принципа полиморфизма ООП, когда в дочернем классе переопределяется метод родительского класса. Этот механизм основывается на возможности хранения в переменной, являющейся ссылкой на базовый класс, адреса объекта производного класса. По умолчанию выбор вызываемой функции осуществляется в соответствии с типом ссылки. В случае вызова виртуальной функции через ссылку на базовый класс программа будет использовать метод, определенный для типа объекта, а не метод, определенный для типа ссылки (реализация принципа позднего связывания).

Объявление виртуального метода в базовом классе `BaseClass` осуществляется добавлением в начало объявления метода ключевого слова `virtual`, например:

```
virtual void Method()
{
        .    .    .
}
```

В производном классе `DerivedClass` этот метод может переопределяться. Для этого в определении метода указывается ключевое слово `override`:

```
override  void Method()
{
        .    .    .
}
```

Продемонстрируем работу виртуального метода на примере:

```
BaseClass bc = new BaseClass();
DerivedClass dc= new DerivedClass();
BaseClass  bcRef; // ссылка на базовый класс

bcRef = bc;       // ссылке на базовый класс присваивается
                  // адрес объекта класса BaseClass

bcRef.Method();   // вызов метода Method() класса BaseClass

bcRef = dc;       // ссылке на базовый класс
                  // присваивается адрес объекта
                  // класса DerivedClass

bcRef.Method();   // вызов метода Method()
                  // класса DerivedClass
```

В данном случае две одинаковые строки кода будут приводить к вызову двух разных методов: базового и производного классов.

Конструкторы классов не могут являться виртуальными функциями, поскольку производным классом не наследуется конструктор базового класса.

Виртуальные функции могут быть перекрыты в производных классах только виртуальными функциями. В этом случае метод производного класса не считается переопределенным и будет вызываться только для объекта производного класса.

Снова обратимся к классу «Квадратное уравнение». В приложении выделяются три вида уравнений – уравнения 0-ого, 1-ого и 2-ого порядков. Для решения каждого из них используется свой собственный метод решения. В случае расширения функциональности класса для решения уравнений более высоких порядков, появится необходимость внесения изменений в методы анализа и решения уравнения. Если типов уравнений будет много, то при добавлении нового типа уравнения методы необходимо изменять, при этом их код становится объемным. Еще одним недостатком является наличие неиспользуемых переменных класса `Equation`. Например, для задания уравнения 0-ого порядка достаточно хранить один коэффициент, а переменные для других коэффициентов и корней не нужны. Остальные элементы класса `Equation` будут использовать ресурсы, но их значения будут игнорироваться. Это говорит о неэффективном использовании памяти.

Определить уравнения можно, используя другую структуру классов, которая не приводит к изменению уже написанного кода при добавлении нового типа уравнений и хранит для каждого типа столько параметров, сколько необходимо для его задания.

Для каждого типа уравнения задается собственный класс, например, `Equation_0`, `Equation_1`, `Equation_2`. Все эти классы обладают одинаковым поведением – решается уравнение и печатается результат. Поэтому можно эти методы определить как виртуальные в отдельном родительском классе `Equation`. Поскольку родительский класс «не знает», каков тип уравнения, как его решать и как распечатать результат, методы родительского класса будут только информировать о том, что необходимо использовать объекты классов-наследников.

Базовый класс нашего примера теперь будет иметь вид:

```
// базовый класс для определения уравнений
public class Equation
{
   protected const int infinity = Int32.MaxValue;
   protected int count = -1;

   public Equation(){}

   // виртуальный метод решения уравнения
   public virtual void Solve()
   {
      PrintSolution();
   }

   // виртуальный метод печати результата решения уравнения
   public virtual void PrintSolution()
   {
      Console.WriteLine("Уравнение еще не решено,
                         поскольку не определен тип
                         уравнения!!!");
   }
}
```

Тогда класс-наследник, описывающий уравнения нулевого порядка, будет содержать методы, переопределяющие виртуальные методы базового класса:

```
// производный класс для определения уравнений 0-ого порядка
class Equation_0: Equation
{
   // уравнение 0-ого порядка определяется только
   // коэффициентом c
   protected double c;

   // конструктор
   public Equation_0(double c1)
   {
      c = c1;
   }

   // переопределенный виртуальный метод решения уравнения
   public override void Solve()
   {
      if (c == 0)
         count = infinity;
      else
         count = 0;
      PrintSolution();
   }

   // переопределенный виртуальный метод печати
   // результата решения уравнения
   override public void PrintSolution()
   {
      Console.WriteLine("{0} = 0", c);
      if (count == -1)
      {
         // вызов базовой реализации метода
         base.PrintSolution();
         return;
      }
      if (count==0)
         Console.WriteLine("Корней нет");
      else
         Console.WriteLine("Любое x является решением");
   }
}
```

В класс `Equation_0` также вводятся элементы, необходимые для задания уравнения нулевого порядка: коэффициент `c`, конструктор класса. Аналогичным образом определяются классы и для других типов уравнений. В нашем случае это классы `Equation_1`, `Equation_2`.

Для решения конкретных типов уравнений в базовом классе создадим специальный метод `CreateEquation()`, который

производит анализ типа уравнения по его коэффициентам и создает объект-уравнение нужного типа. Поскольку при создании уравнений нет необходимости создавать объект базового класса, этот метод определим как статический. Тип возвращаемого значения метода `CreateEquation` – ссылка на базовый класс, которая может хранить адрес объекта любого дочернего класса.

```
// метод базового класса, предназначенный для создания
// объектов-уравнений в зависимости от набора коэффициентов
public static Equation CreateEquation
                          (params double[] coef)
{
   double a, b, c;
   // инициализируем коэффициенты уравнения
   // в зависимости от длины массива коэффициентов
   switch (coef.Length)
   {
      case 3:
         a = coef[0];
         b = coef[1];
         c = coef[2];
         break;
      case 2:
         a = 0.0;
         b = coef[0];
         c = coef[1];
         break;
      case 1:
         a = 0.0;
         b = 0.0;
         c = coef[0];
         break;
      default: throw new Exception("Данный набор
                     коэффициентов не определяет
                     рассматриваемые типы уравнений");
   }
   // определение типа уравнения по коэффициентам
   if (a == 0)
      if (b == 0)
         // создается и возвращается
         // объект класса Equation_0
         return new Equation_0(c);
      else
         // создается и возвращается
         // объект класса Equation_1

         return new Equation_1(b, c);
    else
       // создается и возвращается
```

```
            // объект класса Equation_2
            return new Equation_2(a, b, c);
        }
    }
```

Классы для реализации уравнений будут иметь следующие определения:

```
    // базовый класс уравнения
    public class Equation
    {
        protected const int infinity = Int32.MaxValue;
        protected int count = -1;

        public Equation(){}

        // виртуальный метод решения уравнения
        public virtual void Solve()
        {
            PrintSolution();
        }

        // статический метод создания объекта-уравнения
        public static Equation CreateEquation
                                        (params double[] coef)
        {
            double a, b, c;
            switch (coef.Length)
            {
                case 3:
                    a = coef[0];
                    b = coef[1];
                    c = coef[2];
                    break;
                case 2:
                    a = 0.0;
                    b = coef[0];
                    c = coef[1];
                    break;
                case 1:
                    a = 0.0;
                    b = 0.0;
                    c = coef[0];
                    break;
                default:
                    throw new Exception("Данный набор
                              коэффициентов не определяет
                              рассматриваемые типы уравнений ");
            }
            if (a == 0)
```

```
            if (b == 0)
               return new Equation_0(c);
            else
               return new Equation_1(b, c);
         else
           return new Equation_2(a, b, c);
      }

      // виртуальный метод печати результата решения уравнения
      virtual public void PrintSolution()
      {
         Console.WriteLine("Уравнение еще не решено,
                            поскольку не определен тип
                            уравнения!!!");
      }
   }

   // производный класс уравнений 0-ого порядка
   class Equation_0: Equation
   {
      protected double c;

      // переопределенный виртуальный метод решения уравнения
      public override void Solve()
      {
         if (c == 0)
            count = infinity;
         else
            count = 0;
         PrintSolution();
      }

      // переопределенный виртуальный метод печати
      // результатов решения уравнения
      override public void PrintSolution()
      {
         Console.WriteLine("{0} = 0", c);
         if (count == -1)
         {
            base.PrintSolution();
            return;
         }
         if (count==0)
            Console.WriteLine("Корней нет");
         else
            Console.WriteLine("Любое x является решением");
      }

      // конструктор класса
      public Equation_0(double c1)
      {
         c = c1;
```

```
    }
}

// производный класс уравнения 1-ого порядка
// (линейного уравнения)
class Equation_1: Equation_0
{
    //добавляется коэффициент b
    protected double b;
    // уравнение имеет один корень
    protected double x1;

    // переопределенный виртуальный метод решения уравнения
    override public void Solve()
    {
        count = 1;
        x1 = -c / b;
        PrintSolution();
    }

    // переопределенный виртуальный метод печати
    // результатов решения уравнения
    override public void PrintSolution()
    {
        Console.WriteLine("{0} x + {1} = 0", b, c);
        if (count == -1)
        {
            base.PrintSolution();
            return;
        }
        Console.WriteLine("x={0}", x1);
    }

    // конструктор
    public Equation_1(double b1,double c1):base(c1)
    {
        b = b1;
    }
}

// производный класс уравнения 2-ого порядка
// (квадратного уравнения)
class Equation_2: Equation_1
{
    // по сравнению с линейным уравнением добавляется
    // коэффициент a
    protected double a;
    // уравнение может иметь два корня
    protected double x2;

    // переопределенный виртуальный метод решения уравнения
    public override void Solve()
```

```
    {
       double disc = b * b - 4 * a * c;

       if (disc < 0.0)
          count = 0;
       else if (disc == 0.0)
            {
               count = 1;
               x1 = -b / (2 * a);
               x2 = x1;
            }
            else
            {
               count = 2;
               x1 = (-b + Math.Sqrt(disc)) / (2 * a);
               x2 = (-b - Math.Sqrt(disc)) / (2 * a);
            }
       PrintSolution();
    }

    // переопределенный виртуальный метод печати
    // результатов решения уравнения
    override public void PrintSolution()
    {
        Console.WriteLine("{0} x^2 {1} x + {2} = 0",
                                                a, b, c);
        if (count == -1)
        {
           base.PrintSolution();
           return;
        }
        switch (count)
        {
           case 2:
              Console.WriteLine("x1={0}, x2={1}", x1, x2);
              break;
           case 1:
              Console.WriteLine("x={0}", x1);
              break;
           case 0:
              Console.WriteLine("Корней нет");
              break;
        }
     }

     // конструктор класса
     public Equation_2(double a1, double b1,
                                  double c1):base(b1,c1)
     {
        a = a1;
     }
}
```

Отметим, что при определении классов в примере, создана иерархия классов, которая имеет вид последовательности:

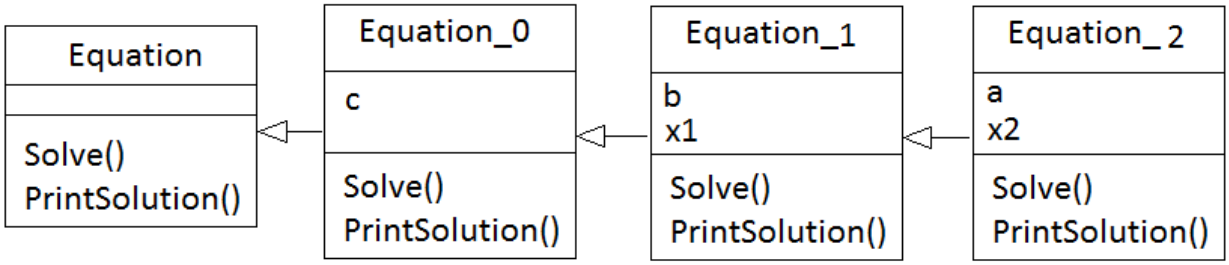

Рис.6.3. Иерархия классов-уравнений.

В данной иерархии в классах-наследниках заново переопределены виртуальные методы `Solve()`, `PrintSolution()` базового класса `Equation`, который является родителем для всех классов ирерархии.

Приведем пример использования базового класса для решения конкретного уравнения с применением виртуальных методов.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            // создание объекта базового класса
            Equation e = new Equation();
            // обращение к виртуальному методу базового класса
            e.Solve();
            // создание объекта класса Equation_2
            e = Equation.CreateEquation(1, -4, 4);
            // вызов вирутального метода решения
            // квадратного уравнения из класса Equation_2
            e.Solve();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

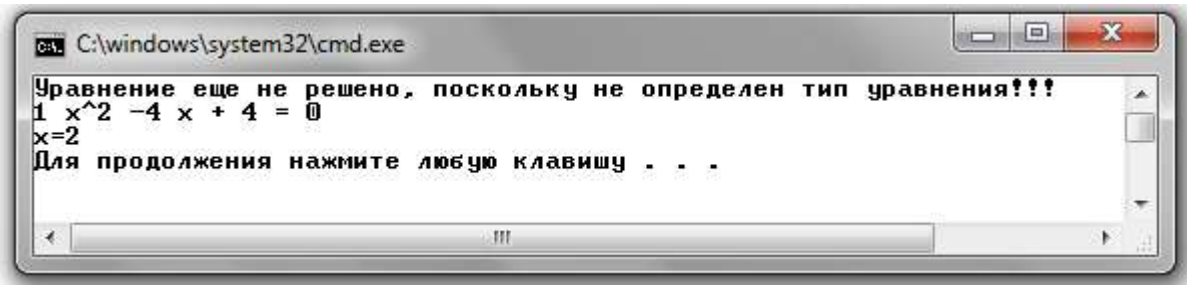

Рис.6.4. Результат работы программы решения уравнений
с созданием иерархии классов

В результате работы программного кода, приведенного в качестве примера, первое сообщение соответствует выполнению метода `Solve()` базового класса, поскольку объект для конкретного типа уравнения еще не создан и решать пока нечего. Далее, анализируя коэффициенты уравнения с помощью метода `CreateEquation()`, создается объект типа `Equation_2` и производится решение уравнения, после чего результат выводится на экран. В этом случае выполняются переопределенные методы `Solve()` и `PrintSolution()` класса `Equation_2`.

Приведенный пример показывает, что возникают случаи, когда использование объекта базового класса несущественно, но выделение общего поведения желательно. В этих ситуациях можно выделить базовый класс, который определяет некоторое абстрактное понятие с абстрактным поведением. Такие классы называют ***абстрактными***.

Поскольку поведение этих классов определить невозможно, виртуальные функции только объявляются в классе, но не определяются. Такие методы называются ***абстрактными методами***. При объявлении такого метода следует указывать ключевое слово `abstract`

```
abstract void Method();
```

Если в классе определен хотя бы один абстрактный метод, то класс является ***абстрактным*** и при его описании также указывается ключевое слово `abstract`. Абстрактный класс не может использоваться для создания объектов, поскольку не полностью определена его реализация. Его можно применять только в качестве базового класса, переопределив в его потомках абстрактные функции.

Если в производном классе не будет переопределена хотя бы одна абстрактная функция базового класса, то он также будет абстрактным. При его описании ключевое слово `abstract` обязательно.

В программах можно создавать ссылки на абстрактный класс для хранения адресов объектов классов-потомков.

Этот прием используем для иерархии классов различных типов уравнений. Базовый класс `Equation` будет являться абстрактным – в нем будет определен абстрактный метод `Solve()`, поскольку абстрактное уравнение решить нельзя. В дочерних классах этот метод определяется для решения конкретных видов уравнений. Приведем определение базового класса для этого случая.

```
namespace AbstarctClass
{
   // абстрактный базовый класс уравнения
   abstract public class Equation
   {

      protected const int infinity = Int32.MaxValue;

      protected int count = -1;

      // абстрактный метод решения уравнения
      public abstract void Solve();

      // метод создания объектов-наследников
      public static Equation CreateEquation
                           (params double[] coef)
      {
         double a, b, c;
         switch (coef.Length)
         {
            case 3:
               a = coef[0];
               b = coef[1];
               c = coef[2];
               break;
            case 2:
               a = 0.0;
               b = coef[0];
               c = coef[1];
               break;
            case 1:
               a = 0.0;
               b = 0.0;
               c = coef[0];
               break;
```

```
                default:
                    throw new Exception("Данный набор
                          коэффициентов не определяет
                          рассматриваемые типы уравнений ");
            }
            if (a == 0)
                if (b == 0)
                    return new Equation_0(c);
                else
                    return new Equation_1(b, c);
            else
                return new Equation_2(a, b, c);
        }

        // виртуальный метод печати результата
        // решения уравнения
        virtual public void PrintSolution()
        {
            Console.WriteLine("Уравнение еще не решено");
        }
    }
  .    .    .
}
```

Создание объектов абстрактного класса невозможно, поэтому для создания объектов – уравнений различных типов будем использовать метод `CreateEquation()`.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            // создание объекта класса Equation_2 и вызов
            // виртуального метода решения уравнения через
            // ссылку на базовый класс Equation
            Equation e = Equation.CreateEquation(1, -4, 4);
            e.Solve();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Поскольку свойства являются специальными видами методов класса, их также можно определить виртуальными и абстрактными:

```
abstract public тип_данных ИмяСвойства
{
   get;
   set;
}
```

Для этого в теле свойства не указывается реализация метода. Реализация самого свойства определяется в классе-наследнике:

```
public override тип_данных ИмяСвойства
{
   get { return выражение; }
   set { имяОбъекта = value; }
}
```

Абстрактные классы, которые содержат только абстрактные методы и свойства, называются ***интерфейсами***. Такие классы объявляются следующим образом:

```
[модификаторы] interface ИмяИнтерфеса
{
   // Объявление списка абстрактных методов
  .    .    .
}
```

Поскольку интерфейсы являются классами, то они могут наследовать другие интерфейсы.

С помощью интерфейсов в языке C# возможна реализация множественного наследования поведения. В этом случае наследование определяется так:

```
[модификаторы] class ИмяКласса: [ИмяБазовогоКласса,]
                                Список_интерфейсов
{
   .    .    .
}
```

6.3. Создание иерархии исключений

Обработка исключений является удобным способом предотвращения аварийного завершения программы в случае попытки выполнения каких-либо некорректных действий. Для этого используется класс `Exception` пространства имен `System`, с

помощью которого передается информация о возникшей проблеме. Когда же в программе требуется предусмотреть возможность появления различных по типу и обработке исключительных ситуаций, использовать один класс `Exception` и соответствующий ему обработчик `catch` становится неудобным, так как обработчик должен включать в себя все возможные обработки исключительных ситуаций.

Рассмотрим такую ситуацию на примере. При работе с матрицами исключительные ситуации могут возникнуть в случаях, когда:

- конструктор получает неположительные значения количества строк или столбцов;
- происходит обращение к элементу матрицы по некорректным индексам;
- при выполнении операции сложения размеры двух матриц не совпадают;
- при выполнении операции умножения количество столбцов первой матрицы не совпадает с количеством строк второй матрицы;
- при вычислении обратной матрицы исходная матрица оказывается вырожденной.

Если во всех этих случаях пользователю необходимо показать только сообщение об ошибке и прервать выполнение блока `try`, то достаточно использовать только стандартное исключение `Exception`, передав в его поле `Message` необходимое сообщение. Однако если требуется более сложная обработка ошибки, – например, корректировка данных, приведение матриц к определенному виду и повторное выполнение операции, вызвавшей исключение – единый обработчик будет сложным и неудобным:

```
try
{
   // код, в котором могут возникнуть
   // исключения с матрицами
}

catch (Exception e)
{
   if (e.Message.Equals("Summ"))
   {
      // действия по обработке ситуации, когда нельзя
      // сложить две матрицы
      .     .     .
   }
```

```
        if (e.Message.Equals("Product"))
        {
            // действия по обработке ситуации, когда нельзя
            // умножить две матрицы
            .     .     .
        }
        // обработка других исключительных ситуаций
      .     .     .
  }
```

Теперь продемонстрируем другой подход, основанный на создании иерархии классов исключений. При использовании такого подхода для каждого вида исключительных ситуаций создается свой класс, наследующий от класса `Exception`. Эти классы могут содержать свои данные и методы для корректной передачи информации и обработки соответствующей исключительной ситуации. Например, для обработки ситуации невозможности перемножения двух матриц создадим класс:

```
// класс-исключение о нарушении размеров матриц
// при их перемножении
class DimensionProductException : Exception
{
    Matrix ob1, ob2;   // ссылки на объекты-матрицы


    // конструктор класса-исключения
    public DimensionProductException(Matrix a, Matrix b)
    {
        ob1 = a;
        ob2 = b;
    }

    // метод обработки исключения - печати подробного
    // сообщения об ошибке
    public void ExceptionHandler()
    {
        Console.WriteLine("Сделана попытка
                                перемножения матриц:");
        ob1.OutputMatrix();
        Console.WriteLine();
        ob2.OutputMatrix();
        Console.WriteLine("Такие матрицы перемножить нельзя
                                из-за неправильных
                            размеров");
    }
}
```

В этот класс добавлены два объекта класса `Matrix`, перемножение которых вызвало ошибку. Далее, например, в методе `ExceptionHandler()` они используются для вывода подробной информации об ошибке:

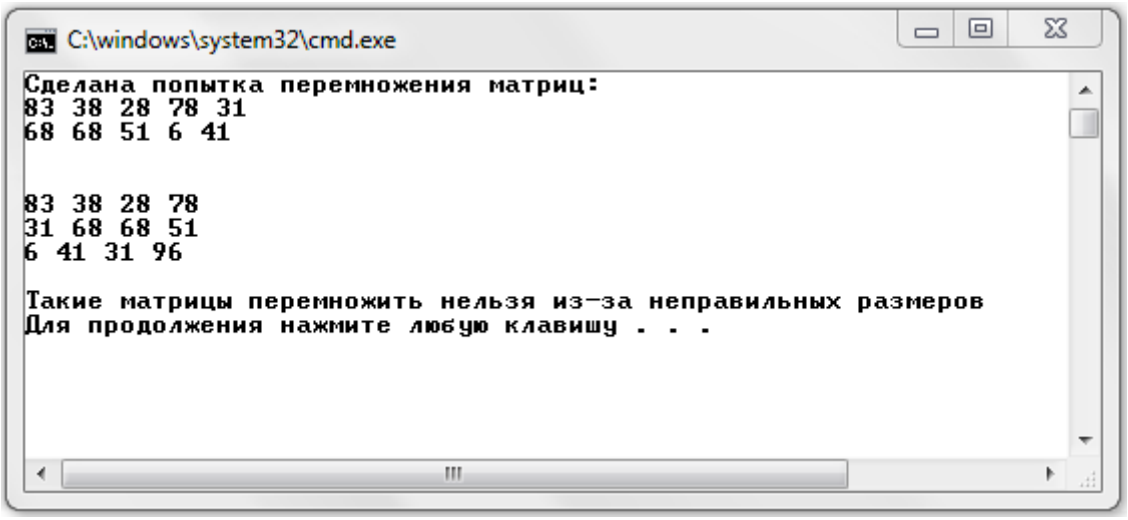

Рис.6.5. Демонстрация возникновения исключения при перемножении матриц

Аналогично можно создать собственный класс исключений для обработки попытки создания матрицы некорректных размеров:

```
// класс-исключение о некорректный размерах матрицы
class BadDimensionException : Exception
{
   // размеры, которые использовались при неудачном
   // создании матрицы
   int rows, cols;


   // конструктор класса-исключения
   public BadDimensionException(int m, int n)
   {
      rows = m; cols = n;
   }

   // метод обработки исключения - печати подробного
   // сообщения об ошибке
   public void ExceptionHandler()
   {
      Console.WriteLine("Попытка создания матрицы с
                         некорректными размерами");
```

```
        if (rows<=0)
            Console.WriteLine("Количество строк матрицы не
                                может быть равным {0}", rows);
        if (cols <= 0)
            Console.WriteLine("Количество столбцов матрицы
                          не может быть равным {0}", cols);
    }
}
```

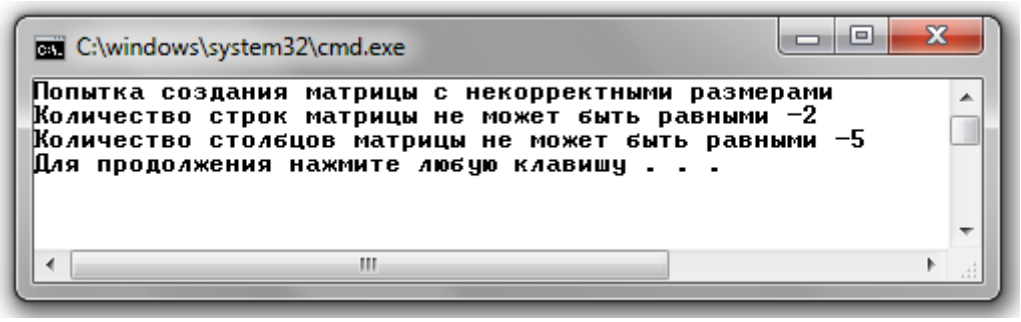

Рис.6.6. Демонстрация возникновения исключения при создании матриц

Имея несколько типов исключений, для каждого из них можно создать свой обработчик `catch` с соответствующим типом исключения в параметре. Обработчики разных исключений должны следовать друг за другом после контролируемого блока `try`, в котором эти исключения могут возникнуть.

```
  .     .     .
try
{
    // в этом фрагменте кода может возникнуть исключение
    // некорректных размеров при создании матрицы и при
    // перемножении двух матриц
    Matrix x, y, z;
    x = new Matrix(2, 5);
    y = new Matrix(3, 4);

    x.InputMatrix();
    y.InputMatrix();
    z = x * y;
    z.OutputMatrix();
}


catch (DimensionProductException ex)
{
    // вызов обработчика исключения, связанного
    // умножением матриц
```

```
        ex.ExceptionHandler();
    }
    catch (BadDimensionException ex)
    {
        // вызов обработчика исключения, связанного
        // созданием матрицы некорректного размера
        ex.ExceptionHandler();
    }
      .    .    .
```

Согласно принципу полиморфизма можно создавать один обработчик `catch` для нескольких видов исключений. Если имеется иерархия классов-исключений, достаточно создавать обработчик `catch` только для исключения базового типа. Универсальная обработка исключений-наследников производится с помощью переопределения виртуальных функций базового класса.

Например, создадим абстрактный базовый класс иерархии – класс `MatrixException`. В нем объявим абстрактный метод `ExceptionHandler()`, который предназначен для обработки ошибки. Все классы-исключения теперь должны наследовать от класса `MatrixException` и переопределять абстрактный метод `ExceptionHandler()` этого класса. Классы-наследники могут содержать и другие методы, которые позволяют, если не устранить ошибку, то хотя бы предотвратить ее влияние на последующий ход выполнения программы (корректное освобождение ресурсов, занимаемых объектом, присвоение переменным объекта корректных значений и пр.). Вызвать эти методы можно из метода `ExceptionHandler()` конкретного класса.

Таким образом, иерархия классов исключений может быть представлена в следующем виде:

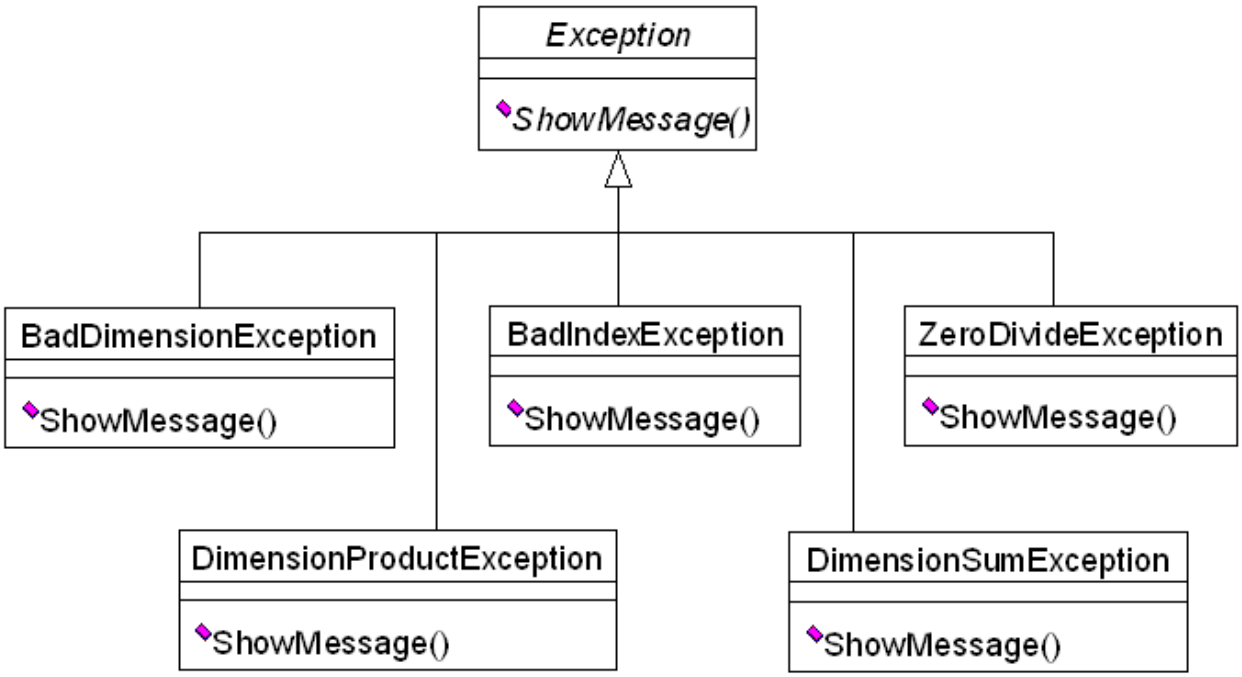

Рис.6.7. Иерархия исключений, возникающих при работе с матрицами

```
// абстрактный базовый класс иерархии исключений с
// единственным абстрактным методом обработки исключения
abstract class MatrixException : Exception
{
   abstract public void ExceptionHandler();
}
```

Приведем также вид одного из классов-наследников:

```
// класс-исключение о нарушении размеров матриц
// при их перемножении
class DimensionProductException : MatrixException
{
   Matrix ob1, ob2;

   // конструктор класса-исключения
   public DimensionProductException(Matrix a, Matrix b)
   {
      ob1 = a; ob2 = b;
   }

   // переопределенный абстрактный метод
   // обработки исключения
   public override void ExceptionHandler()
   {
      Console.WriteLine("Сделана попытка
                              перемножения матриц:");
      ob1.OutputMatrix();
```

```
            Console.WriteLine();
            ob2.OutputMatrix();
            Console.WriteLine("Такие матрицы перемножить
                              нельзя из-за неправильных размеров");
        }
    }
```

Используя иерархию классов-исключений, все виды ошибок можно обработать универсальным образом с помощью только одного обработчика `catch` типа ссылки на объект базового класса:

```
      .     .     .
    try
    {
        // в этом фрагменте кода может возникнуть исключение
        // некорректных размеров при создании матрицы и при
        // перемножении двух матриц
        Matrix x, y, z;
        x = new Matrix(2, 3);
        y = new Matrix(3, 4);

        x.InputMatrix();
        y.InputMatrix();
        z = x * y;
        z.OutputMatrix();
    }
    catch (MatrixException ex)
    {
        // общий обработчик исключений-наследников
        // класса MatrixException
        // вызов метода обработки исключения того класса,
        // которому принадлежит объект ex
        ex.ExceptionHandler();
    }
      .     .     .
```

В зависимости от возникающей ошибки согласно принципу полиморфизма ссылка на базовый класс будет указывать на объект производного класса, соответствующего типу ошибки. Тогда в обработчике `catch` будет вызываться метод `ExceptionHandler()` именно этого класса.

Глава 7. Обобщения

7.1. Обобщения. Основные понятия

Термин «обобщение», по существу, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых тип обрабатываемых данных указываются в виде параметра. С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для единообразной обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

Следует особо подчеркнуть, что в C# имеется возможность создавать обобщенный код, оперируя ссылками типа `object`. Класс `object` является базовым для всех остальных классов, таким образом, по ссылке типа `object` можно обращаться к объекту любого типа. Недостатком такого приема является несоблюдение типовой безопасности. Для преобразования типа `object` в конкретный тип данных необходимо приведение типов, которое может быть источником ошибок.

Применение обобщений, напротив, обеспечивает типовую безопасность и тем самым не требуют выполнения приведения типов для преобразования объекта или другого типа обрабатываемых данных.

Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

Рассмотрим пример использования обобщений при разработке класса для хранения множества элементов некоторого типа.

```
// обобщенный класс «Множество»
class Set<T>
{
   // массив элементов множества,
   // элементы имеют обобщенный тип T
```

```
    T[] elements;

    // конструктор класса
    public Set(T[] a)
    {
        if (a.Length == 0)
            elements = null;
        else
        {
            // копирование массива элементов множества
            elements = new T[a.Length];
            a.CopyTo(elements, 0);
        }
    }

    // индексатор доступа к элементам множества по номеру
    public T this[int index]
    {
        set
        {
            if (index < 0 || index >= elements.Length)
                throw new Exception("Индекс элемента выходит
                            за границы множества.");
            elements[index] = value;
        }

        get
        {
            if (elements.Length == 0)
                throw new Exception("Множество пустое.");
            if (index < 0 || index >= elements.Length)
                throw new Exception("Индекс элемента выходит
                            за границы множества.");
            return elements[index];
        }
    }

    // метод проверки наличия в множестве заданного элемента
    public bool Contains(T find)
    {
        foreach (T el in elements)
            if (el.Equals(find))
                return true;
        return false;
    }
}

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

```
        {
            // демонстрация создания объекта-множества
            // для целых чисел
            int[] a = { 1, 2, 3, 4, 5, 0 };
            Set<int> s1 = new Set<int>(a);
            Console.WriteLine(s1.Contains(4));
            // демонстрация создания объекта-множества
            // для вещественных чисел
            double[] a2 = { 11, 22, 33, 44, 55, 60 };
            Set<double> s2 = new Set<double>(a2);
            Console.WriteLine("" + s2.Contains(4));

            // демонстрация создания объекта-множества
            // для символьных строк
            string[] a3 = { "111", "222", "333", "444",
                            "555", "666" };
            Set<string> s3 = new Set<string>(a3);
            Console.WriteLine("" + s3.Contains("222"));
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Прокомментируем данный программный код. Обобщенный класс задается следующим образом:

```
class Set<T>
{
        .    .    .
}
```

где `T` – это имя обобщенного типа (параметр типа, параметризующий тип). Это имя указывает место подстановки конкретного типа, который указывается при создании объекта класса `Set`. Следовательно, имя `T` используется в классе `Set` всякий раз, когда требуется параметр типа. Имя `T` заключается в угловые скобки (< >). Обобщенный класс может быть построен на нескольких обобщенных типах. Тогда их имена указываются в угловых скобках через запятую.

Схема определения обобщенного класса имеет следующий вид:

```
class имя_обобщенного_класса<список_обобщенных_типов>
{
   // определение элементов класса
        .    .    .
}
```

список_обобщенных_типов := T1 [,T2, . . ., TN], где Ti – произвольный идентификатор обобщенного типа.

Далее имя T используется для объявления массива элементов elements, как показано в следующей строке кода:

```
T[] elements;    // объявить массив элементов типа T
```

Так как имя параметра типа T указывает на место подстановки конкретного типа при создании объекта класса Set, поэтому массив elements будет иметь тип, привязываемый к T при создании экземпляра объекта класса Set. Так, если вместо T указывается тип double, то в экземпляре данного объекта элементы массива elements будут иметь тип double. Подставляемые типы называются аргументами типа.

Рассмотрим конструктор класса Set.

```
// конструктор класса
public Set(T[] a)
{
   if (a.Length == 0)
      elements = null;
   else
   {
      elements = new T[a.Length];
      a.CopyTo(elements,0);
   }
}
```

Параметр a и массив elements относятся к одному и тому же обобщенному типу T. Конкретный тип обоих массивов определится в момент создания объекта класса Set.

С помощью параметра типа T можно также указывать тип, возвращаемый методом.

```
// индексатор для доступа к элементам множества
public T this[int index]
{
   set
   {
      if (index < 0 || index >= elements.Length)
         throw new Exception("Индекс элемента выходит
                              за границы множества.");
      elements[index] = value;
   }

   get
   {
      if (elements.Length == 0)
         throw new Exception("Множество пустое.");
      if (index < 0 || index >= elements.Length)
         throw new Exception("Индекс элемента выходит
                              за границы множества.");
      return elements[index];
   }
}
```

Элементы массива `elements` относятся к типу `T`, поэтому их тип совпадает с типом, возвращаемым индексатором `this[int]`.

В методе `Main(string [] args)` демонстрируется применение обобщенного класса `Set`.

```
// демонстрация создания объекта-множества
// для целых чисел
int[] a = { 1, 2, 3, 4, 5, 0 };
Set<int> s1 = new Set<int>(a);
Console.WriteLine(s1.Contains(4));

// демонстрация создания объекта-множества
// для вещественных чисел
double[] a2 = { 11, 22, 33, 44, 55, 60 };
Set<double> s2 = new Set<double>(a2);
Console.WriteLine("" + s2.Contains(4));

// демонстрация создания объекта-множества
// для символьных строк
string[] a3 = { "111", "222", "333", "444", "555", "666" };
Set<string> s3 = new Set<string>(a3);
Console.WriteLine("" + s3.Contains("222"));
```

В этом методе создаются три множества: для хранения целых чисел, вещественных чисел и символьных строк. Имя конкретного

типа данных элементов множества указывается вместо обобщенного типа `T` в угловых скобках при создании экземпляра класса `Set`. В каждом экземпляре тип `T` заменяется конкретным типом везде, где он встречается (в методах, в свойствах, в объявлении полей класса и т.д.).

Заметим, что два экземпляра класса `Set` для различных типов элементов, например, для `int` и для `double`, принадлежат разным типам данных и поэтому не приводятся друг к другу. Следующий код содержит ошибку, так как в нем делается попытка присвоить ссылке на тип `Set<int>` ссылку на объект типа `Set<double>`, создаваемый с помощью оператора `new`:

```
double [] a = { 1, 2, 3, 4, 5, 0 };
Set<int> s1 = new Set<double>(a);          // Ошибка
```

Аналогично, ошибки могут возникнуть и при обращении к методам объектов класса `Set`, в которых обобщенный тип используется при задании параметров методов или их типов возвращаемых значений. Так, например, в следующем фрагменте возникнет ошибка, поскольку результат функции присваивается переменной типа `double`.

```
int [] a = { 1, 2, 3, 4, 5, 0 };
Set<int> s1 = new Set<int>(a);
double d = s1[3];                      // Ошибка
```

Тип, образуемый вследствие подстановки аргументов типа вместо обобщенного, называется ***сконструированным типом***. В качестве параметра типа может быть подставлен как конкретный тип (например `int`, `string` или `MySomeStruct`), так и тип, сам по себе являющийся обобщенным параметром (или сконструированный с их использованием). В первом случае получаемый тип является так называемым ***закрытым типом***, а во втором - ***открытым***. Иногда закрытый тип называют специализацией, так как при этом порождается специализированная версия обобщенного типа. В приведенном выше примере `Set<int>` является сконструированным закрытым типом. Процесс создания сконструированного типа из обобщенного типа называется *generic type instantiation*.

Сконструированный тип создается, когда компилятор в первый раз встречает его упоминание. Например, это происходит при выполнении следующих строк кода:

```
Set<int> s1 = new Set<int>(a);
Set<double> s2 = new Set<double>(a2);
```

7.2. Уточнения, используемые в обобщениях

При определении обобщенного класса предполагается, что обобщенный тип может быть любым, т.е. компилятор «не знает», какие конкретные типы будут использоваться в качестве параметров обобщения. Поэтому параметры типа «по умолчанию» рассматриваются как `object`. Таким образом, при обращении к типу-параметру в обобщении программист может использовать только методы, унаследованные от класса `object` (`Equals()`, `GetType()`, `ToString()` и др.). Это сильно ограничивает выбор алгоритмов, которые могут использоваться в методах обобщения. Например, пусть в обобщение `Set` требуется добавить метод получения минимального элемента множества. Код данного метода мог бы выглядеть следующим образом:

```
// свойство для получения минимального элемента в множестве
public T Min
{
   get
   {
      if (elements.Length == 0)
         throw new Exception("Пустое множество");
      T min = elements[0];
      foreach (T el in elements)
         if (el<min)            // Ошибка!
            min = el;
      return min;
   }
}
```

Ошибка при компилировании этого свойства вызвана тем, что для класса `object` не определены операции сравнения объектов (в данном случае, операция "<"). Эту ошибку можно исправить, используя в свойстве операцию преобразования (к типу интерфейса

IComparable, в котором определен метод сравнения CompareTo()).

```
// свойство для получения минимального элемента в множестве
public T Min
{
   get
   {
      if (elements.Length == 0)
         throw new Exception("Пустое множество");
      T min = elements[0];
      foreach (T el in elements)
         // приведение элемента множества к типу IComparable
         // и сравнение его с помощью метода CompareTo()
         if (((IComparable)el).CompareTo(min)<0)
            min = el;
      return min;
   }
}
```

Этот вариант решения проблемы может привести к ошибкам времени выполнения программы. Класс, который используется в качестве T и объект которого преобразуется в свойстве к IComparable, обязан реализовывать этот интерфейс. Если это требование не выполняется, при преобразовании возникнет ошибка.

Более корректно эта проблема решается с помощью введения ***уточнений (constraints)*** в обобщения. Задать уточнение можно с помощью ключевого слова where сразу после списка параметров типа. Например, уточнение для обобщения Set о том, что обобщенный тип должен реализовывать интерфейс IComparable, записывается так:

```
// обобщенный класс «Множество» с уточнением о том,
// что элементы множества можно сравнивать
class Set<T> where T:IComparable
{
        .     .     .
     // свойство для получения
     // минимального элемента в множестве
     public T Min
     {
        get
        {
           if (elements.Length == 0)
```

```
                throw new Exception("Пустое множество");
            T min = elements[0];
            foreach (T el in elements)
                if (el.CompareTo(min)<0)
                    min = el;
                return min;
        }
    }
}
```

Заметим, что в этом случае указывать преобразование типа не требуется, поскольку уже на этапе компиляции будет отслеживаться, что аргумент типа, применяемого вместо обобщенного, раскрывает соответствующий интерфейс.

Для одного и того же обобщенного типа можно задать несколько уточнений – они перечисляются через запятую. В случае, когда обобщенных типов несколько, уточнения для каждого из них задаются с помощью собственной конструкции `where`:

```
// примеры возможных уточнений
class SomeGeneric<T1, T2, T3>
      where T1: IComparable
      where T2: SomeGenericClass<T1, T2>, IComparable, new()
      where T3: SomeClass, IEnumerable, IComparable
{
  .    .    .
}
```

В качестве уточнения может использоваться имя класса (при этом параметр типа должен быть унаследован от этого класса), имя интерфейса (который должен реализовать параметр типа) или `new()`. Класс и интерфейсы могут быть и обобщениями.

Имя класса в уточнении может быть задано только один раз и обязано быть первым в списке уточнений. Интерфейсы могут задаваться в любом количестве. Ограничение `new()` говорит о том, что параметр типа должен реализовывать `public`-конструктор, не имеющий параметров, что означает возможность создания экземпляра соответствующего типа с помощью оператора `new`. Ограничение `new()` должно быть последним в списке уточнений.

В .NET можно описывать не только обобщенные классы, но и обобщенные интерфейсы (`generic`-интерфейсы), обобщенные

делегаты (generic-делегаты), отдельные методы классов (generic-методы).

Обобщенные классы могут быть унаследованы от других классов, в том числе и от обобщенных. Также они могут реализовывать любое количество интерфейсов, как обычных, так и generic. Простые классы и интерфейсы также могут быть унаследованы от generic-классов, но при этом generic-классы и интерфейсы должны быть закрытыми типами. Обобщенные классы и интерфейсы не могут быть унаследованы от типов-параметров.

Приведем несколько примеров:

```
// Нельзя использовать параметр типа
// в качестве базового типа!
class Extend<V> : V {}              // Ошибка!

// Пусть определены обобщенный класс C
// и обобщенный интерфейс I1
class C<U, V> {}
interface I1<V> {}

//  Базовые типы закрытые.
class D : C<string, int>, I1<string> {}
//  Базовые типы конструируемые.
class E<T> : C<int, T>, I1<T> {}
class F : I1<T> {}          // Ошибка! Тип T не определен.
class G<T> : C<string, int> {}      // Базовый тип закрытый
```

7.3. Обобщенные интерфейсы

Обобщенный интерфейс задает абстрактное поведение (абстрактные методы), которое может быть использовано при различных типах параметров. Например, следующий обобщенный интерфейс задает все операции сравнения объектов обобщенного типа T:

```
// интерфейс, определяющий методы сравнения двух элементов
interface IMyComparer<T>
{
   bool Greater(T ob1, T ob2);
   bool Less(T ob1, T ob2);
```

```
    bool LessOrEqual(T ob1, T ob2);
    bool GreaterOrEqual(T ob1, T ob2);
    bool Equal(T ob1, T ob2);
    bool NotEqual(T ob1, T ob2);
}
```

Наследовать этот интерфейс могут как открытые (обобщенные) классы, так и закрытые (конкретные) классы. Во втором случае интерфейс `IMyComparer` используется для конкретного типа данных.

Создадим на основе этого интерфейса обобщенный класс, в котором определяются все методы сравнения.

```
// обобщенный класс, предназначенный для сравнения элементов
// раскрывает обобщенный интерфейс IMyComparer
class MyComparer<T> : IMyComparer<T> where T : IComparable
{
    // переопределяются различные методы сравнения
    // двух элементов, определенные интерфейсом IMyComparer
    public bool Greater(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2)>0;
    }

    public bool Less(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2) < 0;
    }

    public bool LessOrEqual(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2) <= 0;
    }

    public bool GreaterOrEqual(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2) >= 0;
    }

    public bool Equal(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2) == 0;
    }

    public bool NotEqual(T ob1, T ob2)
    {
        return ob1.CompareTo(ob2) != 0;
    }
}
```

Поскольку методы сравнения объектов используют функцию CompareTo, которую можно использовать только в случае, когда сравнимые объекты раскрывают интерфейс IComparable, на обобщенный тип T должно быть наложено ограничение:

```
class MyComparer<T> : IMyComparer<T> where T : IComparable
{
        .    .    .
}
```

7.4. Обобщенные методы

Необобщенный класс может содержать отдельные обобщенные методы. Обобщенные методы определяются следующим образом:

```
тип_функции имя_функции<список_обобщенных_типов>
                                    (список_параметров)
{
        .    .    .
}
```

список_обобщенных_типов := T1 [,T2, . . ., TN], Ti – произвольный идентификатор обобщенного типа;

тип_функции – тип возвращаемого значения, в качестве которого может быть указан как конкретный тип данных, так и один из списка обобщенных типов;

список_параметров – список формальных параметров обобщенной функции, которые могут быть описаны как с указанием конкретного типа данных, так и одного типа из списка обобщенных типов.

Например, определим класс, который содержит методы сортировки массивов различными способами. Для того чтобы можно было использовать эти методы для сортировки массивов с различными типами элементов, методы сортировки можно сделать обобщенными. В примере продемонстрирована реализация алгоритма быстрой сортировки.

```
// класс, предназначенный для сортировки массивов
class Sorter
{
   // оббобщенный метод быстрой сортировки массива
   public static void Sort<T>(IMyComparer<T> comparer,
                              T[] items, int left, int right)
   {
      int i = left;
      int j = right;
      T center = items[(left + right) / 2];
      // цикл упорядочивания элементов
      // относительно элемента center
      while (i <= j)
      {
         while (comparer.Greater(center, items[i]))
            i++;
         while (comparer.Greater(items[j], center))
            j--;

         if (i <= j)
         {
            T x = items[i];
            items[i] = items[j];
            items[j] = x;
            i++;
            j--;
         }
      }
     // вызов метода сортировки элементов,
     // расположенных слева от center

      if (left < j)
         Sort(comparer, items, left, j);
     // вызов метода сортировки элементов,
     // расположенных справа от center
      if (right > i)
         Sort(comparer, items, i, right);
   }
   // другие методы сортировки
     .  .    .
}
```

При описании обобщенного метода указывает параметр-тип `T`. Первым параметром является объект, тип которого раскрывает обобщенный интерфейс `IMyComparer`, предназначенный для сравнения объектов (его код представлен ранее), второй параметр метода – сортируемый массив элементов обобщенного типа:

```
public static void Sort<T>(IMyComparer<T> comparer,
                           T[] items, int left, int right)
{
  .    .    .
}
```

Вызов обобщенного метода класса осуществляется с указанием конкретного типа элементов массива. Например,

```
int[] a = {1, 2, 10, 4, -5, 0};

// создание объекта закрытого типа на основе
// обобщенного класса MyComparer и типа int
IMyComparer<int> intComparer = new MyComparer<int>();

// вызов обобщенного статического метода
// класса Sortirovschik
Sorter.Sort<int>(intComparer,a,0,a.Length-1);
```

7.5. Обобщенные делегаты

Поскольку делегаты являются классами, они также могут быть обобщенными. В этом случае можно создавать обобщенные алгоритмы, логику которых можно изменять передаваемыми в качестве параметров делегатами.

Определение обобщенного делегата имеет следующий вид:

```
delegate тип имя_делегата<список_обобщенных_типов>
                                    (список_параметров)
{
      .    .    .
}
```

список_обобщенных_типов := T1 [,T2, . . ., TN], Ti – произвольный идентификатор обобщенного типа;

тип – тип возвращаемого значения функций, на которые может ссылаться делегат. В качестве этого типа может быть указан как конкретный тип данных, так и один из списка обобщенных типов;

список_параметров – список формальных параметров функций, на которые ссылается делегат. Они также могут быть

описаны как с указанием конкретного типа данных, так и одного типа из списка обобщенных типов.

Например, для хранения ссылки на функции сравнения объектов произвольных типов, определим обобщенный делегат:

```
delegate bool CompareDelegate<T>(T ob1, T ob2);
```

Экземпляр этого делегата может указывать на любые функции различных классов, в том числе и конструированных на основе обобщенных, тип возвращаемого значения которых `bool` и формальными параметрами которых являются два объекта одинакового типа. Этому прототипу

```
bool SomeMethod(Type ob1, Type ob2)
```

соответствуют все функции сравнения объектов.

Тогда функция сортировки может быть определена следующим прототипом:

```
public static void Sort<T>(CompareDelegate <T> comparer,
                           T[] items, int left, int right)
```

где первым параметром является указатель на функцию сравнения, задающую порядок сортировки элементов массива (по возрастанию или по убыванию). Сам метод выглядит так:

```
public static void Sort<T>(CompareDelegate <T> comparer,
                           T[] items, int left, int right)
{
   int i = left;
   int j = right;
   T center = items[(left + right) / 2];

   while (i <= j)
   {
      // вызов операции сравнения через делегат
      while (comparer(center, items[i]))
         i++;
      while (comparer(items[j], center))
         j--;

      if (i <= j)
```

```
            {
                T x = items[i];
                items[i] = items[j];
                items[j] = x;
                i++;
                j--;
            }
        }

        if (left < j)
            Sort(comparer, items, left, j);
        if (right > i)
            Sort(comparer, items, i, right);
    }
```

Приведем примеры использования данной версии функции сортировки по возрастанию и по убывания массива целых чисел.

```
int[] a = {1, 2, 10, 4, -5, 0,-15,100,45,13,-250};
Console.WriteLine("Исходный массив");
foreach (int i in a)
   Console.Write(i.ToString() + " ");
Console.WriteLine();

// создание объекта, предназначенного
// для сравнения целых чисел
MyComparer<int> c = new MyComparer<int>();

// создание делегата, указывающего на функцию
// сравнения целых чисел
// Greater объекта c
CompareDelegate <int> g = new
                          CompareDelegate<int>(c.Greater);

// вызов функции сортировки массива по возрастанию
// с использованием делегата g
Sorter.Sort<int>(g, a, 0, a.Length - 1);
Console.WriteLine("Массив отсортирован
                           по возрастанию элементов");
foreach (int i in a)
   Console.Write(i.ToString() + " ");
Console.WriteLine();


// переназначение делегату g ссылки на функцию сравнения
// целых чисел Less объекта c
g = c.Less;

// вызов функции сортировки массива по убыванию
// с использованием делегата g
Sorter.Sort<int>(g, a, 0, a.Length - 1);
```

```
Console.WriteLine("Массив отсортирован
                                  по убыванию элементов");
foreach (int i in a)
   Console.Write("" + i.ToString() + " ");
Console.WriteLine();
```

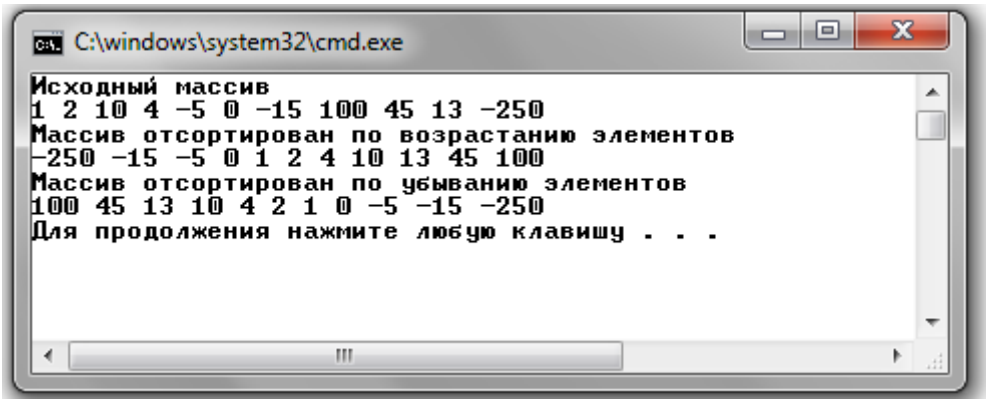

Рис.7.1. Результат выполнения программы сортировки массива целых чисел

Предметный указатель

Литература

1. Рамбо, Дж. UML 2.0. Объектно-ориентированное моделирование и разработка [Текст]: пер. с англ. / Джеймс Рамбо, Мартин Блаха. – СПб: Питер, 2007. – 544 с.

2. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ [Текст]: пер. с англ. / Гради Буч. – М: Бином, СПб: Невский диалект, 1999. – 560 с.

3. Пышкин, Е.В. Основные концепции и механизмы объектно-ориентированного программирования [Текст]/ Е.В.Пышкин. – СПб: БХВ-Петербург, 2005. – 640 с.

4. Шилдт, Г.. C# 4.0: полное руководство [Текст]: Пер. с англ. / Герберт Шилдт. – М.: ООО "И.Д. Вильямс", 2011. – 1056 с.

5. Дейтел, Х. C# в подлиннике. Наиболее полное руководство [Текст]: пер. с англ./ Харви Дейтел, Пол Дейтел. – СПб: БХВ-Петербург, 2006 г.. – 1056 с.

6. Троелсен, Э. Язык программирования C# 2010 и платформа .NET 4 [Текст]: Пер. с англ. / Эндрю Троелсен. – М.: ООО "И.Д. Вильямс", 2011. – 1392 с.

7. Уотсон, К. Visual C# 2010: полный курс [Текст]: Пер. с англ./ Карли Уотсон, Кристиан Нейгел, Якоб Хаммер Педерсен, Джон Д. Рид, Морган Скиннер. – М.: Диалектика, 2010. – 960 с.

8. Трей, Нэш C# 2010: ускоренный курс для профессионалов курс [Текст]: Пер. с англ./ Нэш Трей. - М.: ООО "И.Д. Вильямс", 2011. – 592 с.

9. Кубенский, А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++ [Текст] / А.А. Кубенский. – СПб: БХВ-Петербург, 2004. – 464 с.

10. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.

11. Сайт центра разработки на Visual C# [Интернет-ресурс]. URL: http://msdn.microsoft.com/ru-ru/vcsharp/. Дата обращения: 10.11.2011.