1. Лекция: Visual Studio .Net, Framework .Net

Имя .Net

Имена нынешнего поколения продуктов от Microsoft сопровождаются окончанием .Net (читается Dot Net), отражающим видение Microsoft современного коммуникативного мира. Компьютерные сети объединяют людей и технику. Человек, работающий с компьютером или использующий мобильный телефон, естественным образом становится частью локальной или глобальной сети. В этой сети используются различные специальные устройства, начиная от космических станций и кончая датчиками, расположенными, например, в гостиницах и посылающими информацию об объекте всем мобильным устройствам в их окрестности. В глобальном информационном мире коммуникативная составляющая любых программных продуктов начинает играть определяющую роль.

В программных продуктах .Net за этим именем стоит вполне конкретное содержание, которое предполагает, в частности, наличие открытых стандартов коммуникации, переход от создания монолитных приложений к созданию компонентов, допускающих повторное использование в разных средах и приложениях. Возможность повторного использования уже созданных компонентов и легкость расширения их функциональности - все это непременные атрибуты новых технологий. Важную роль в этих технологиях играет язык XML, ставший стандартом обмена сообщениями в сети.

Не пытаясь охватить все многообразие сетевого взаимодействия, рассмотрим реализацию новых идей на примере *Visual Studio .Net* - продукта, важного для разработчиков.

Visual Studio .Net - открытая среда разработки

Среда разработки *Visual Studio .Net* - это уже проверенный временем программный продукт, являющийся седьмой версией Студии. Но новинки этой версии, связанные с идеей .Net, позволяют считать ее принципиально новой разработкой, определяющей новый этап в создании программных продуктов. Выделю две важнейшие, на мой взгляд, идеи:

- *открытость* **для языков программирования;**
- принципиально новый подход к построению **каркаса среды - Framework .Net**.

Открытость

Среда разработки теперь является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой Microsoft - **Visual C++ .Net** (с управляемыми расширениями), **Visual C# .Net**, **J# .Net**, **Visual Basic .Net**, - в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами-производителями. Таких расширений среды **Visual Studio** сделано уже достаточно много, практически они существуют для всех известных языков - **Fortran** и **Cobol**, **RPG** и **Component Pascal**, **Oberon** и **SmallTalk**. Я у себя на компьютере включил в среду компилятор одного из лучших объектных языков - языка **Eiffel**.

Открытость среды не означает полной свободы. Все разработчики компиляторов при включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение, которое можно считать и главным достоинством, состоит в том, что все языки, включаемые в среду разработки *Visual Studio .Net*, должны использовать единый *каркас - Framework .Net*. Благодаря этому достигаются многие желательные свойства: легкость использования компонентов, разработанных на различных языках; возможность разработки нескольких частей одного приложения на

разных языках; возможность бесшовной отладки такого приложения; возможность написать класс на одном языке, а его потомков - на других языках. Единый каркас приводит к сближению языков программирования, позволяя вместе с тем сохранять их индивидуальность и имеющиеся у них достоинства. Преодоление языкового барьера - одна из важнейших задач современного мира. Благодаря единому каркасу, *Visual Studio .Net* в определенной мере решает эту задачу в мире программистов.

Framework .Net - единый каркас среды разработки

В *каркасе Framework .Net* можно выделить два основных компонента:

- статический - ***FCL (Framework Class Library)*** - *библиотеку классов* каркаса;
- динамический - ***CLR (Common Language Runtime)*** - общеязыковую исполнительную среду.

Библиотека классов FCL - статический компонент каркаса

Понятие каркаса приложений - Framework Applications - появилось достаточно давно; по крайней мере оно широко использовалось еще в четвертой версии Visual Studio. Десять лет назад, когда я с Ильмиром писал книгу [В.А. Биллиг, И.Х. Мусикаев "Visual C++, 4-я версия. Книга для программистов"], для нас это было еще новое понятие. Мы подробно обсуждали роль библиотеки классов MFC (Microsoft Foundation Classes) как каркаса приложений Visual C. Несмотря на то, что каркас был представлен только статическим компонентом, уже тогда была очевидна его роль в построении приложений. Уже в то время важнейшее значение в библиотеке классов MFC имели классы, задающие архитектуру строящихся приложений. Когда разработчик выбирал один из возможных типов приложения, например, архитектуру Document-View, то в его приложение автоматически встраивались класс Document, задающий структуру документа, и класс View, задающий его визуальное представление. Класс Form и классы, задающие элементы управления, обеспечивали единый интерфейс приложений. Выбирая тип приложения, разработчик изначально получал нужную ему функциональность, поддерживаемую классами каркаса. Библиотека классов поддерживала и более традиционные для программистов классы, задающие расширенную систему типов данных, в частности, динамические типы данных - списки, деревья, коллекции, шаблоны.

За прошедшие 10 лет роль каркаса в построении приложений существенно возросла - прежде всего, за счет появления его динамического компонента, о котором чуть позже поговорим подробнее. Что же касается статического компонента - библиотеки классов, то и здесь за десять лет появился ряд важных нововведений.

Единство каркаса

Каркас стал единым для всех языков среды. Поэтому, на каком бы языке программирования не велась разработка, она использует классы одной и той же библиотеки. Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными хранилищами данных и прочая универсальность.

Встроенные примитивные типы

Важной частью *библиотеки FCL* стали классы, задающие ***примитивные типы*** - те типы, которые считаются *встроенными* в язык программирования. Типы каркаса покрывают все множество *встроенных типов*, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic - Integer, а в языке C# - int,

проецируется на один и тот же тип каркаса System.Int32. В каждом языке программирования, наряду с "родными" для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе. Поэтому, по сути, все языки среды разработки могут пользоваться единой системой *встроенных типов*, что, конечно, способствует облегчению взаимодействия компонентов, написанных на разных языках.

Структурные типы

Частью библиотеки стали не только простые *встроенные типы*, но и *структурные типы*, задающие организацию данных - строки, массивы, перечисления, структуры (записи). Это также способствует унификации и реальному сближению языков программирования.

Архитектура приложений

Существенно расширился набор возможных архитектурных типов построения приложений. Помимо традиционных Windows- и консольных приложений, появилась возможность построения Web-приложений. Большое внимание уделяется возможности создания повторно используемых компонентов - разрешается строить библиотеки классов, библиотеки элементов управления и библиотеки Web-элементов управления. Популярным архитектурным типом являются Web-службы, ставшие сегодня благодаря открытому стандарту одним из основных видов повторно используемых компонентов. Для языков C#, J#, Visual Basic, поддерживаемых Microsoft, предлагается одинаковый набор из 12 архитектурных типов приложений. Несколько особняком стоит Visual C++, сохраняющий возможность работы не только с *библиотекой FCL*, но и с библиотеками MFC и ATL, и с построением соответствующих MFC и ATL-проектов. Компиляторы языков, поставляемых другими фирмами, создают проекты, которые удовлетворяют общим требованиям среды, сохраняя свою индивидуальность. Так, например, компилятор Eiffel допускает создание проектов, использующих как *библиотеку FCL*, так и собственную библиотеку классов.

Модульность

Число классов *библиотеки FCL* велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединяются в группы, называемые пространством имен (Namespace). Для динамического компонента *CLR* физической единицей, объединяющей классы и другие ресурсы, является сборка (assembly).

Основным пространством имен *библиотеки FCL* является пространство System, содержащее как классы, так и другие вложенные пространства имен. Так, уже упоминавшийся *примитивный тип* Int32 непосредственно вложен в пространство имен System и его полное имя, включающее имя пространства - `System.Int32`.

В пространство `System` вложен целый ряд других пространств имен. Например, в пространстве `System.Collections` находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов - списками, очередями, словарями. В пространство `System.Collections`, в свою очередь, вложено пространство имен `Specialized`, содержащие классы со специализацией, например, коллекции, элементами которых являются только строки. Пространство `System.Windows.Forms` содержит классы, используемые при создании Windows-приложений. Класс `Form` из этого пространства задает форму - окно, заполняемое элементами управления, графикой, обеспечивающее интерактивное взаимодействие с пользователем.

По ходу курса мы будем знакомиться со многими классами, принадлежащими различным пространствам имен *библиотеки FCL*.

Общеязыковая исполнительная среда CLR - динамический компонент каркаса

Наиболее революционным изобретением *Framework .Net* явилось создание **исполнительной среды** *CLR*. С ее появлением процесс написания и выполнения приложений становится принципиально другим. Но обо всем по порядку.

Двухэтапная компиляция. Управляемый модуль и управляемый код

Компиляторы языков программирования, включенные в *Visual Studio .Net*, создают модули на промежуточном языке ***MSIL (Microsoft Intermediate Language)***, называемом далее просто - IL. Фактически компиляторы создают так называемый ***управляемый модуль*** - переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и ***метаданные*** - всю необходимую информацию как для *CLR*, так и конечных пользователей, работающих с приложением. О *метаданных* - важной новинке *Framework .Net* - мы еще будем говорить неоднократно. В зависимости от выбранного типа проекта, PE-файл может иметь уточнения exe, dll, mod или mdl.

Заметьте, PE-файл, имеющий уточнение exe, хотя и является exe-файлом, но это не совсем обычный, исполняемый Windows, файл. При его запуске он распознается как специальный PE-файл и передается *CLR* для обработки. Исполнительная среда начинает работать с кодом, в котором специфика исходного языка программирования исчезла. Код на IL начинает выполняться под управлением *CLR* (по этой причине **код** называется ***управляемым***). Исполнительную среду можно рассматривать как своеобразную виртуальную IL-машину. Эта машина транслирует "на лету" требуемые для исполнения участки кода в команды реального процессора, который в действительности и выполняет код.

Виртуальная машина

Отделение каркаса от студии явилось естественным шагом. *Каркас Framework .Net* перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание PE-модулей на IL отделено от выполнения, и эти процессы могут быть реализованы на разных платформах. В состав *CLR* входят трансляторы JIT (Just In Time Compiler), которые и выполняют трансляцию IL в командный код той машины, где установлена и функционирует исполнительная среда *CLR*. Конечно, в первую очередь Microsoft реализовала *CLR* и *FCL* для различных версий Windows, включая Windows 98/Me/NT 4/2000, 32 и 64-разрядные версии Windows XP и семейство .Net Server. Для операционных систем Windows CE и Palm разработана облегченная версия *Framework .Net*.

В 2001 году ECMA (Европейская ассоциация производителей компьютеров) приняла язык программирования C#, *CLR* и *FCL* в качестве стандарта, так что *Framework .Net* уже функционирует на многих платформах, отличных от Windows. Он становится свободно распространяемой виртуальной машиной. Это существенно расширяет сферу его применения. Производители различных компиляторов и сред разработки программных продуктов предпочитают теперь также транслировать свой код в IL, создавая модули в соответствии со спецификациями *CLR*. Это обеспечивает возможность выполнения их кода на разных платформах.

Microsoft использовала получивший широкое признание опыт виртуальной машины Java, улучшив процесс за счет того, что, в отличие от Java, промежуточный код не интерпретируется исполнительной средой, а компилируется с учетом всех особенностей текущей платформы. Благодаря этому создаются высокопроизводительные приложения.

Следует отметить, что *CLR*, работая с IL-кодом, выполняет достаточно эффективную оптимизацию и, что не менее важно, защиту кода. Зачастую нецелесообразно выполнять оптимизацию на уровне создания IL-кода - она иногда может не улучшить, а ухудшить ситуацию, не давая *CLR* провести оптимизацию на нижнем уровне, где можно учесть даже особенности процессора.

Дизассемблер и ассемблер

Если у вас есть готовый PE-файл, то иногда полезно анализировать его IL-код и связанные с ним *метаданные*. В состав Framework SDK входит *дизассемблер* - ildasm, выполняющий дизассемблирование PE-файла и показывающий *метаданные*, а также IL-код с комментариями в наглядной форме. Мы иногда будем пользоваться результатами дизассемблирования. У меня на компьютере кнопка, вызывающая *дизассемблер*, находится на панели, где собраны наиболее часто используемые мной приложения. Вот путь к папке, в которой обычно находится *дизассемблер*:

```
C:\Program Files\Microsoft Visual Studio .Net\ FrameworkSDK\Bin\ildasm.exe
```

Профессионалы, предпочитающие работать на низком уровне, могут программировать на языке *ассемблера* IL. В этом случае в их распоряжении будет вся мощь *библиотеки FCL* и все возможности *CLR*. У меня на компьютере путь к папке, где находится *ассемблер*, следующий:

```
C:\WINDOWS\Microsoft.Net\Framework\v1.1.4322\ilasm.exe
```

В этом курсе к *ассемблеру* мы обращаться не будем - я упоминаю о нем для полноты картины.

Метаданные

Переносимый исполняемый PE-файл является самодокументируемым файлом и, как уже говорилось, содержит и код, и *метаданные*, описывающие код. Файл начинается с манифеста и включает в себя описание всех классов, хранимых в PE-файле, их свойств, методов, всех аргументов этих методов - всю необходимую *CLR* информацию. Поэтому помимо PE-файла не требуется никаких дополнительных файлов и записей в реестр - вся нужная информация извлекается из самого файла. Среди классов *библиотеки FCL* имеется класс `Reflection`, методы которого позволяют извлекать необходимую информацию. Введение *метаданных* не только важная техническая часть *CLR*, но это также часть новой идеологии разработки программных продуктов. Мы увидим, что и на уровне языка C# самодокументированию уделяется большое внимание.

Мы увидим также, что при проектировании класса программист может создавать собственные атрибуты, добавляемые к *метаданным* PE-файла. Клиенты этого класса могут, используя класс `Reflection`, получать эту дополнительную информацию, и на ее основании принимать соответствующие решения.

На рис. 1.1 показаны результаты дизассемблирования PE-файла простого консольного приложения с именем `Account`, включающего три класса: `Account`, `Testing` и `Class1`. *Дизассемблер* структурирует информацию, хранимую в *метаданных*, и показывает ее в типичном формате дерева. Как обычно, это дерево можно сжимать или раскрывать, демонстрируя детали класса. Значки, приписываемые каждому узлу дерева, характеризуют тип узла - класс, свойство, метод, описание. Двойной щелчок кнопки мыши на этом узле позволяет раскрыть его. При раскрытии метода можно получить его код. На рис. 1.1 показан код метода `add` из класса `Account`.

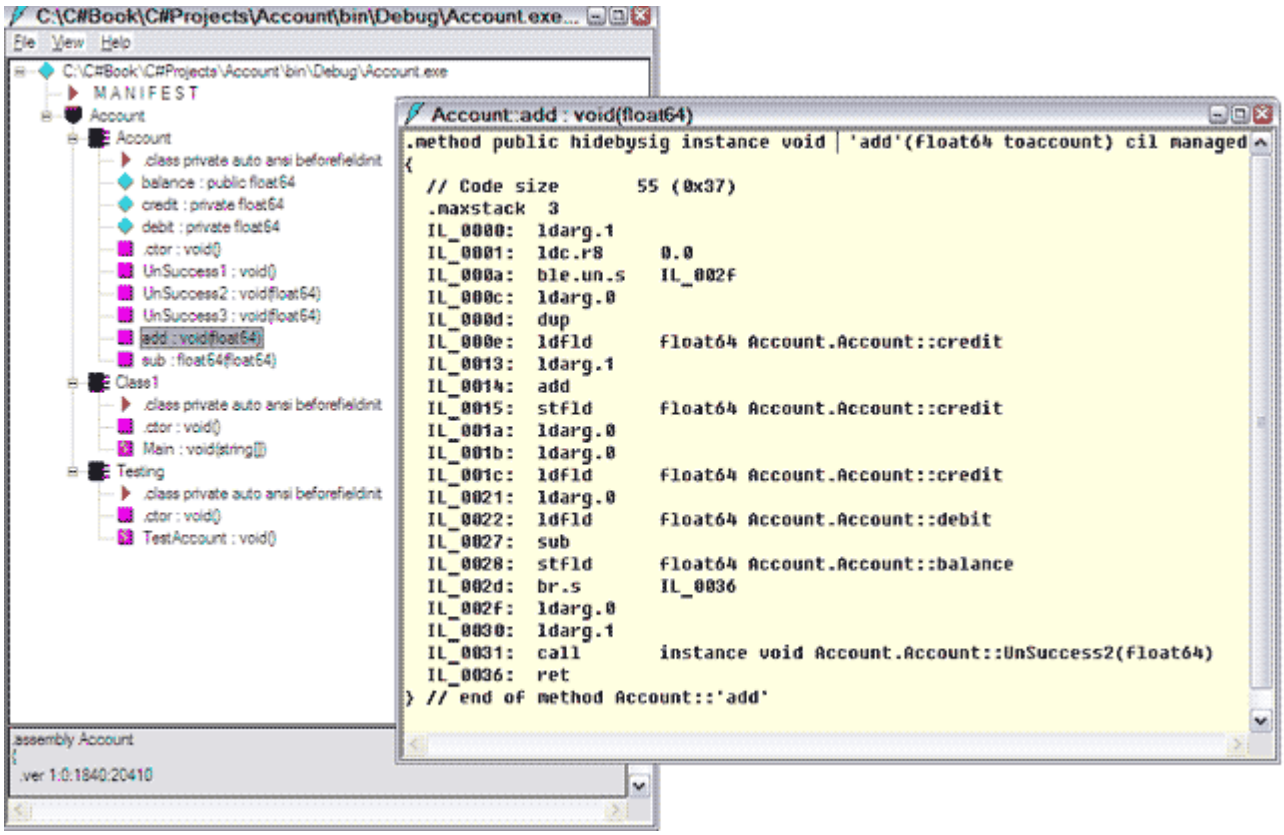

Рис. 1.1. Пример структуры PE-файла, задающего сборку

Сборщик мусора - Garbage Collector - и управление памятью

Еще одной важной особенностью построения *CLR* является то, что исполнительная среда берет на себя часть функций, традиционно входящих в ведение разработчиков трансляторов, и облегчает тем самым их работу. Один из таких наиболее значимых компонентов *CLR* - *сборщик мусора (Garbage Collector)*. Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения. В ряде языков программирования (классическим примером является язык C/C++) память освобождает сам программист, в явной форме отдавая команды как на создание, так и на удаление объекта. В этом есть своя логика - "я тебя породил, я тебя и убью". Однако можно и нужно освободить человека от этой работы. Неизбежные ошибки программиста при работе с памятью тяжелы по последствиям, и их крайне тяжело обнаружить. Как правило, объект удаляется в одном модуле, а необходимость в нем обнаруживается в другом, далеком модуле. Обоснование того, что программист не должен заниматься удалением объектов, а сборка мусора должна стать частью исполнительной среды, дано достаточно давно. Наиболее полно оно обосновано в работах Бертрана Мейера и в его книге "Object-Oriented Construction Software", первое издание которой появилось еще в 1988 году.

В *CLR* эта идея реализована в полной мере. Задача сборки мусора снята не только с программистов, но и с разработчиков трансляторов, она решается в нужное время и в нужном месте - исполнительной средой, ответственной за выполнение вычислений. Здесь же решаются и многие другие вопросы, связанные с использованием памяти, в частности, проверяются возможные нарушения использования "чужой" памяти и другие нарушения, например, с использованием нетипизированных указателей. Данные, удовлетворяющие требованиям *CLR* и допускающие сборку мусора, называются *управляемыми данными*.

Но, как же, спросите вы, быть с языком C++ и другими языками, где есть нетипизированные указатели, адресная арифметика, возможности удаления объектов

программистом? Такие возможности сохранены и в языке C#. Ответ следующий - *CLR* позволяет работать как с *управляемыми*, так и с *неуправляемыми данными*. Однако использование *неуправляемых данных* регламентируется и не поощряется. Так, в C# модуль, использующий *неуправляемые данные* (указатели, адресную арифметику), должен быть помечен как небезопасный (unsafe), и эти данные должны быть четко зафиксированы. Об этом мы еще будем говорить при рассмотрении языка C# в последующих лекциях. Исполнительная среда, не ограничивая возможности языка и программистов, вводит определенную дисциплину в применении потенциально опасных средств языков программирования.

Исключительные ситуации

Что происходит, когда при вызове некоторой функции (процедуры) обнаруживается, что она не может нормальным образом выполнить свою работу? Возможны разные варианты обработки такой ситуации. Функция может возвращать код ошибки или специальное значение типа `HResult`, может **выбрасывать исключение**, тип которого характеризует возникшую ошибку. В *CLR* принято во всех таких ситуациях выбрасывать исключение. Косвенно это влияет и на язык программирования. Выбрасывание исключений наилучшим образом согласуется с исполнительной средой. В языке C# выбрасывание исключений, их дальнейший перехват и обработка - основной рекомендуемый способ обработки *исключительных ситуаций*.

События

У *CLR* есть свое видение того, что представляет собой тип. Есть формальное описание *общей системы типов CTS* - **Common Type System**.В соответствии с этим описанием, каждый тип, помимо полей, методов и свойств, может содержать и *события*. При возникновении *событий* в процессе работы с тем или иным объектом данного типа посылаются сообщения, которые могут получать другие объекты. Механизм обмена сообщениями основан на *делегатах* - функциональном типе. Надо ли говорить, что в язык C# встроен механизм *событий*, полностью согласованный с возможностями *CLR*. Мы подробно изучим все эти механизмы, рассматривая их на уровне языка.

Исполнительная среда *CLR* обладает мощными динамическими механизмами - сборки мусора, динамического связывания, обработки *исключительных ситуаций* и *событий*. Все эти механизмы и их реализация в *CLR* созданы на основании практики существующих языков программирования. Но уже созданная исполнительная среда, в свою очередь, влияет на языки, ориентированные на использование *CLR*. Поскольку язык C# создавался одновременно с созданием *CLR*, то, естественно, он стал языком, наиболее согласованным с исполнительной средой, и средства языка напрямую отображаются в средства исполнительной среды.

Общие спецификации и совместимые модули

Уже говорилось, что *каркас Framework .Net* облегчает межъязыковое взаимодействие. Для того чтобы классы, разработанные на разных языках, мирно уживались в рамках одного приложения, для их бесшовной отладки и возможности построения разноязычных потомков они должны удовлетворять некоторым ограничениям. Эти ограничения задаются *набором общеязыковых спецификаций - CLS (Common Language Specification)*. Класс, удовлетворяющий *спецификациям CLS*, называется *CLS-совместимым*. Он доступен для использования в других языках, классы которых могут быть клиентами или наследниками совместимого класса.

Спецификации CLS точно определяют, каким набором *встроенных типов* можно пользоваться в *совместимых модулях*. Понятно, что эти типы должны быть общедоступными для всех языков, использующих *Framework .Net*. В *совместимых модулях* должны использоваться *управляемые данные* и выполняться некоторые другие ограничения. Заметьте, ограничения касаются только интерфейсной части класса, его

открытых свойств и методов. Закрытая часть класса может и не удовлетворять *CLS*. Классы, от которых не требуется совместимость, могут использовать специфические особенности языка программирования.

На этом я закончу обзорное рассмотрение *Visual Studio .Net* и ее *каркаса Framework .Net*. Одной из лучших книг, подробно освещающих эту тему, является книга Джеффри Рихтера, переведенная на русский язык: "Программирование на платформе .Net Framework". Крайне интересно, что для Рихтера языки являются лишь надстройкой над каркасом, поэтому он говорит о программировании, использующем возможности исполнительной среды *CLR* и *библиотеки FCL*.

2. Лекция: Язык C# и первые проекты

Создание C#

Язык C# является наиболее известной новинкой в области создания языков программирования. В отличие от 60-х годов XX века - периода бурного языкотворчества - в нынешнее время языки создаются крайне редко. За последние 15 лет большое влияние на теорию и практику программирования оказали лишь два языка: Eiffel, лучший, по моему мнению, объектно-ориентированный язык, и Java, ставший популярным во многом благодаря технологии его использования в Интернете и появления такого понятия как виртуальная Java-машина. Чтобы новый язык получил признание, он должен действительно обладать принципиально новыми качествами. Языку C# повезло с родителями. Явившись на свет в недрах Microsoft, будучи наследником C++, он с первых своих шагов получил мощную поддержку. Однако этого явно недостаточно для настоящего признания достоинств языка. Попробуем разобраться, имеет ли он большое будущее?

Создателем языка является сотрудник Microsoft Андреас Хейлсберг. Он стал известным в мире программистов задолго до того, как пришел в Microsoft. Хейлсберг входил в число ведущих разработчиков одной из самых популярных сред разработки - Delphi. В Microsoft он участвовал в создании версии Java - J++, так что опыта в написании языков и сред программирования ему не занимать. Как отмечал сам Андреас Хейлсберг, C# создавался как язык компонентного программирования, и в этом одно из главных достоинств языка, направленное на возможность повторного использования созданных компонентов. Из других объективных факторов отметим следующие:

- C# создавался параллельно с каркасом Framework .Net и в полной мере учитывает все его возможности - как FCL, так и CLR;
- C# является полностью объектно-ориентированным языком, где даже типы, встроенные в язык, представлены классами;
- C# является мощным объектным языком с возможностями наследования и универсализации;
- C# является наследником языков C/C++, сохраняя лучшие черты этих популярных языков программирования. Общий с этими языками синтаксис, знакомые операторы языка облегчают переход программистов от C++ к C#;
- сохранив основные черты своего великого родителя, язык стал проще и надежнее. Простота и надежность, главным образом, связаны с тем, что на C# хотя и допускаются, но не поощряются такие опасные свойства C++ как указатели, адресация, разыменование, адресная арифметика;
- благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты C# получают те же преимущества работы с виртуальной машиной, что и программисты Java. Эффективность кода даже повышается, поскольку исполнительная среда CLR предоставляет собой компилятор промежуточного языка, в то время как виртуальная Java-машина является интерпретатором байт-кода;
- мощная библиотека каркаса поддерживает удобство построения различных типов приложений на C#, позволяя легко строить Web-службы, другие виды компонентов, достаточно просто сохранять и получать информацию из базы данных и других хранилищ данных;
- реализация, сочетающая построение надежного и эффективного кода, является немаловажным фактором, способствующим успеху C#.

Виды проектов

Как уже отмечалось, Visual Studio .Net для языков C#, Visual Basic и J# предлагает 12 возможных видов *проектов*. Среди них есть пустой *проект*, в котором изначально не

содержится никакой функциональности; есть также *проект*, ориентированный на создание Web-служб. В этой книге, направленной, прежде всего, на изучение языка C#, основным видом используемых *проектов* будут обычные Windows-приложения. На начальных этапах, чтобы не усложнять задачу проблемами пользовательского интерфейса, будем рассматривать также *консольные приложения*.

Давайте разберемся, как создаются *проекты* и что они изначально собой представляют. Поговорим также о сопряженных понятиях: *решение (solution)*, *проект (project)*, *пространство имен (namespace)*, *сборка (assembly)*. Рассмотрим результаты работы компилятора Visual Studio с позиций программиста, работающего над *проектом*, и с позиций CLR, компилирующей PE-файл в исходный код процессора.

С точки зрения программиста, компилятор создает *решение*, с точки зрения CLR - *сборку*, содержащую PE-файл. Программист работает с *решением*, CLR - со *сборкой*.

Решение содержит один или несколько *проектов*, ресурсы, необходимые этим *проектам*, возможно, дополнительные файлы, не входящие в *проекты*. Один из *проектов решения* должен быть выделен и назначен *стартовым проектом*. Выполнение *решения* начинается со *стартового проекта*. *Проекты* одного *решения* могут быть *зависимыми* или *независимыми*. Например, все *проекты* одной лекции данной книги могут быть для удобства собраны в одном *решении* и иметь общие свойства . Изменяя *стартовый проект*, получаем возможность перехода к нужному примеру. Заметьте, *стартовый проект* должен иметь точку входа - класс, содержащий статическую процедуру с именем Main, которой автоматически передается управление в момент запуска *решения* на выполнение. В уже имеющееся *решение* можно добавлять как новые, так и существующие *проекты*. Один и тот же *проект* может входить в несколько *решений*.

Проект состоит из классов, собранных в одном или нескольких *пространствах имен*. *Пространства имен* позволяют **структурировать** *проекты*, содержащие большое число классов, объединяя в одну группу близкие классы. Если над *проектом* работает несколько исполнителей, то, как правило, каждый из них создает свое *пространство имен*. Помимо структуризации, это дает возможность присваивать классам имена, не задумываясь об их уникальности. В разных *пространствах имен* могут существовать одноименные классы. *Проект* - это основная единица, с которой работает программист. Он выбирает тип *проекта* , а Visual Studio создает скелет *проекта* в соответствии с выбранным типом.

Дальнейшие объяснения лучше сочетать с реальной работой над *проектами*. Поэтому во всей этой книге я буду вкратце описывать свои действия по реализации тех или иных *проектов*, надеясь, что их повторение читателем будет способствовать пониманию текста и сути изучаемых вопросов.

Консольный проект

У себя на компьютере я открыл установленную лицензионную версию Visual Studio .Net 2003, выбрал из предложенного меню - создание нового *проекта* на C#, установил вид *проекта* - *консольное приложение*, дал имя *проекту* - `ConsoleHello`, указал, где будет храниться *проект*. Как выглядит задание этих установок, показано на рис. 2.1.

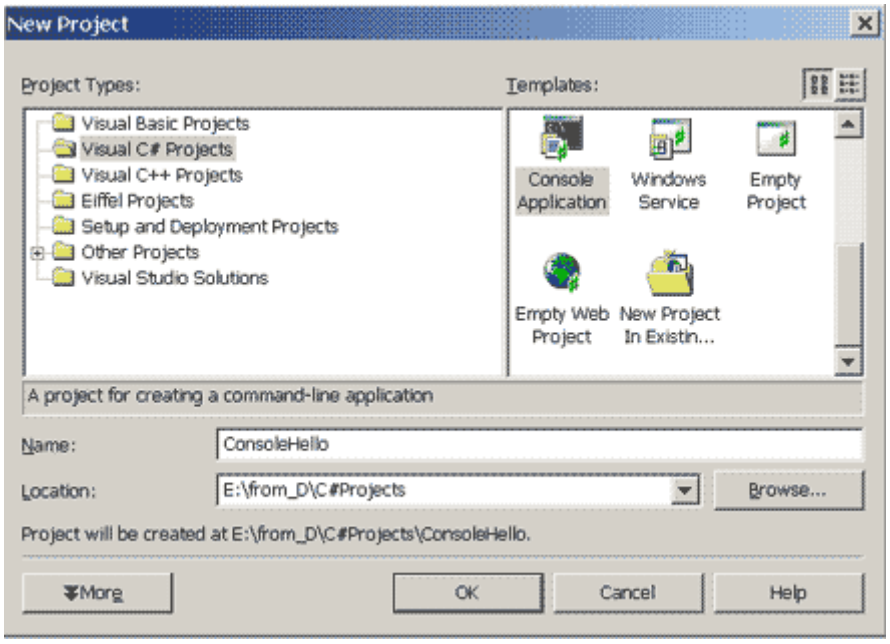

Рис. 2.1. Окно создания нового проекта

Если принять эти установки, то компилятор создаст *решение*, имя которого совпадает с именем *проекта*. На рис. 2.2 показано, как выглядит это *решение* в среде разработки:

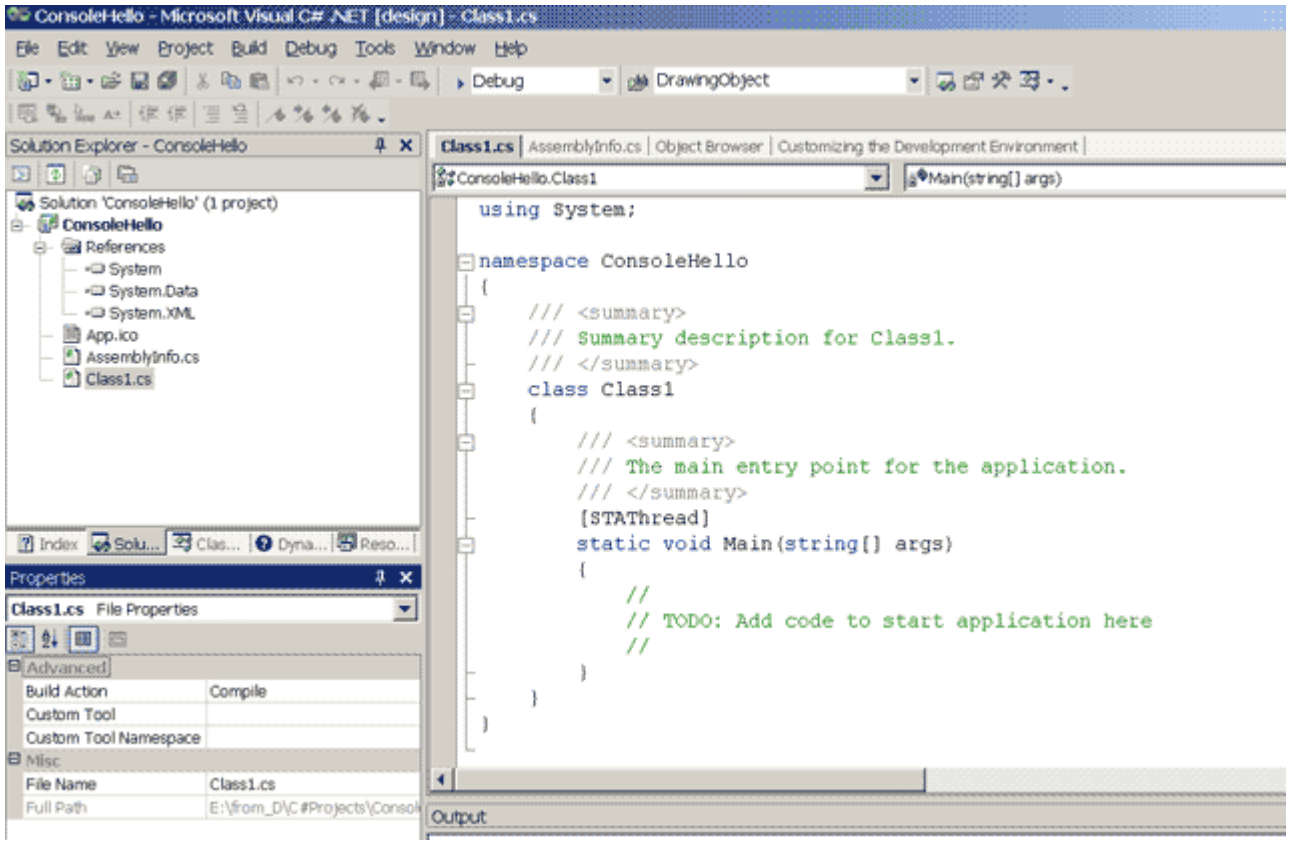

Рис. 2.2. Среда разработки и консольное приложение, построенное по умолчанию

Интегрированная среда разработки IDE (Integrated Development Envirionment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей. Внешний вид ее достаточно традиционен, хотя здесь есть новые возможности; я не буду заниматься ее описанием, полагаясь на опыт читателя и справочную систему. Обращаю внимание лишь на три окна, из тех, что показаны на рис. 2.2. В окне Solution Explorer представлена структура построенного *решения*. В

окне Properties можно увидеть свойства выбранного элемента *решения*. В окне документов отображается выбранный документ, в данном случае, программный код класса *проекта* - `ConsoleHello.Class1`. Заметьте, в этом окне можно отображать и другие документы, список которых показан в верхней части окна.

Построенное *решение* содержит, естественно, единственный заданный нами *проект* - `ConsoleHello`. Наш *проект*, как показано на рис. 2.2, включает в себя папку со ссылками на системные *пространства имен* из библиотеки FCL, файл со значком приложения и два файла с уточнением `cs`. Файл `AssemblyInfo` содержит информацию, используемую в *сборке*, а файл со стандартным именем `Class1` является построенным по умолчанию классом, который задает точку входа - процедуру `Main`, содержащую для данного типа *проекта* только *комментарий*.

Заметьте, класс *проекта* погружен в *пространство имен*, имеющее по умолчанию то же имя, что и *решение*, и *проект*. Итак, при создании нового *проекта* автоматически создается достаточно сложная вложенная структура - *решение*, содержащее *проект*, содержащий *пространство имен*, содержащее класс, содержащий точку входа. Для простых *решений* такая структурированность представляется избыточной, но для сложных - она осмысленна и полезна.

Помимо понимания структуры *решения*, стоит также разобраться в трех важных элементах, включенных в начальный *проект* - предложение using, тэги документации в *комментариях* и *атрибуты*.

Пространству имен может предшествовать одно или несколько предложений using, где после ключевого слова следует название *пространства имен* - из библиотеки FCL или из *проектов*, связанных с текущим *проектом*. В данном случае задается *пространство имен* `System` - основное *пространство имен* библиотеки FCL. Предложение using NameA облегчает запись при использовании классов, входящих в *пространство* `NameA`, поскольку в этом случае не требуется каждый раз задавать полное имя класса с указанием имени *пространства*, содержащего этот класс. Чуть позже мы увидим это на примере работы с классом `Console` пространства `System`. Заметьте, полное имя может потребоваться, если в нескольких используемых *пространствах имен* имеются классы с одинаковыми именами.

Все языки допускают *комментарии*. В C#, как и в C++, допускаются однострочные и многострочные *комментарии*. Первые начинаются с двух символов косой черты. Весь текст до конца строки, следующий за этой парой символов, (например, `"//любой текст"`) воспринимается как *комментарий*, не влияющий на выполнение программного кода. Началом многострочного *комментария* является пара символов `/*`, а концом - `*/`. Заметьте, тело процедуры `Main` содержит три однострочных *комментария*.

Здесь же, в *проекте*, построенном по умолчанию, мы встречаемся с еще одной весьма важной новинкой C# - **XML-тегами**, формально являющимися частью *комментария*. Отметим, что описанию класса `Class1` и описанию метода `Main` предшествует заданный в строчном *комментарии* XML-тег `<summary>`. Этот тэг распознается специальным инструментарием, строящим XML-отчет *проекта*. Идея самодокументируемых программных *проектов*, у которых документация является неотъемлемой частью, является важной составляющей стиля компонентного надежного программирования на C#. Мы рассмотрим реализацию этой идеи в свое время более подробно, но уже с первых шагов будем использовать *теги* документирования и строить XML-отчеты. Заметьте, кроме *тега* `<summary>` возможны и другие тэги, включаемые в отчеты. Некоторые *теги* добавляются почти автоматически. Если в нужном месте (перед объявлением класса, метода) набрать подряд три символа косой черты, то автоматически вставится тэг

документирования, так что останется только дополнить его соответствующей информацией.

Еще одна новинка C#, встречающаяся в начальном *проекте*, это *атрибут* `[STAThread]`, который предшествует описанию процедуры `Main`. Так же, как и тэги документирования, *атрибуты* распознаются специальным инструментарием и становятся частью метаданных. *Атрибуты* могут быть как стандартными, так и заданными пользователем. Стандартные *атрибуты* используются CLR и влияют на то, как она будет выполнять *проект*. В данном случае *атрибут* `[STAThread]` (Single Thread Apartment) задает однопоточную модель выполнения. Об *атрибутах* и метаданных мы еще будем говорить подробно. Заметьте, если вы нечетко представляете, каков смысл однопоточной модели, и не хотите, чтобы в вашем тексте присутствовали непонятные для вас указания, то этот *атрибут* можно удалить из текста, что не отразится на выполнении.

Скажем еще несколько слов о точке входа - процедуре `Main`. Ее заголовок можно безболезненно упростить, удалив аргументы, которые, как правило, не задаются. Они имеют смысл, когда *проект* вызывается из командной строки, позволяя с помощью параметров задать нужную стратегию выполнения *проекта*.

Таков консольный *проект*, построенный по умолчанию. Функциональности у него немного. Его можно скомпилировать, выбрав соответствующий пункт из меню **build**. Если компиляция прошла без ошибок, то в результате будет произведена *сборка* и появится PE-файл в соответствующей папке `Debug` нашего *проекта*. Приложение можно запустить нажатием соответствующих клавиш (например, CTRL+F5) или выбором соответствующего пункта из меню **Debug**. Приложение будет выполнено под управлением CLR. В результате выполнения появится консольное окно с предложением нажать любую клавишу для закрытия окна.

Слегка изменим *проект*, построенный по умолчанию, добавим в него свой код и превратим его в классический *проект* приветствия. Нам понадобятся средства для работы с консолью - чтения с консоли (клавиатуры) и вывода на консоль (дисплей) строки текста. Библиотека FCL предоставляет для этих целей класс `Console`, среди многочисленных методов которого есть методы `ReadLine` и `WriteLine` с очевидной семантикой. Вот код *проекта*, полученный в результате моих корректировок:

```
using System;
namespace ConsoleHello
{
   /// <summary>
   /// Первый консольный проект - Приветствие
   /// </summary>
   class Class1
   {
      /// <summary>
      /// Точка входа. Запрашивает имя и выдает приветствие
      /// </summary>
      static void Main()
      {
         Console.WriteLine("Введите Ваше имя");
         string name;
         name = Console.ReadLine();
         if (name=="")
            Console.WriteLine ("Здравствуй, мир!");
         else
            Console.WriteLine("Здравствуй, " + name + "!");
      }
   }
}
```

Я изменил текст в *тегах* `<summary>`, удалил *атрибут* и аргументы процедуры `Main`, добавил в ее тело операторы ввода-вывода. Благодаря предложению `using`, мне не требуется при вызове методов класса `Console` каждый раз писать `System.Console`. Надеюсь, что программный текст понятен без дальнейших пояснений.

В завершение первого *проекта* построим его XML-отчет. Для этого в свойствах *проекта* необходимо указать имя файла, в котором будет храниться отчет. Установка этого свойства *проекта*, так же как и других свойств, делается в окне Property Pages, открыть которое можно по-разному. Я обычно делаю это так: в окне Solution Explorer выделяю строку с именем *проекта*, а затем в окне Properties нажимаю имеющуюся там кнопку Property Pages. Затем в открывшемся окне свойств, показанном на рис. 2.3, устанавливается нужное свойство. В данном случае я задал имя файла отчета `hello.xml`.

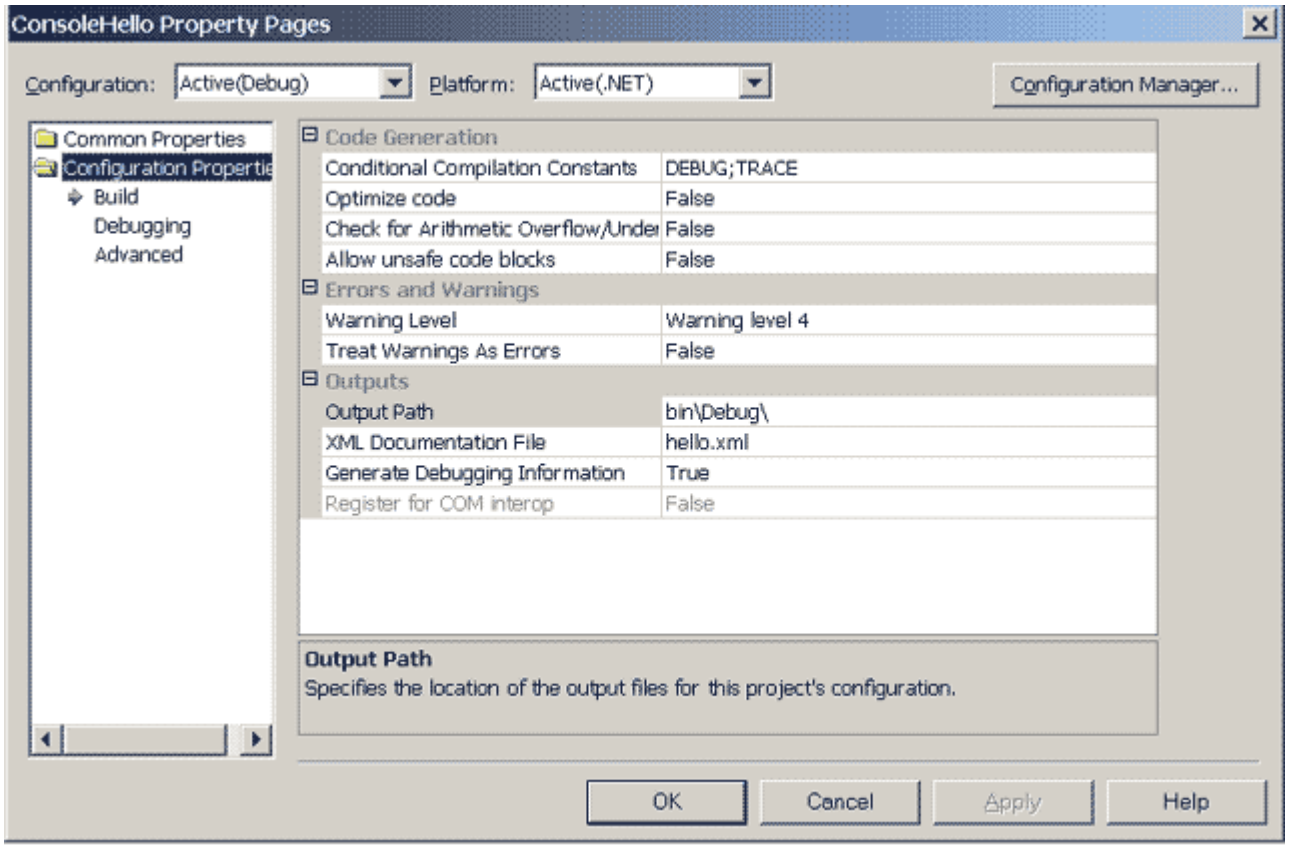

Рис. 2.3. Окно Property Pages проекта и задание имени XML-отчета

После перестройки *проекта* можно открыть этот файл с документацией. Если соблюдать дисциплину, то в нем будет задана спецификация *проекта*, с описанием всех классов, их свойств и методов. Вот как выглядит этот отчет в данном примере:

```
<?xml version="1.0"?>
<doc>
<assembly>
   <name>ConsoleHello</name>
</assembly>
<members>
   <member name="T:ConsoleHello.Class1">
      <summary>
         Первый консольный проект - Приветствие
      </summary>
   </member>
   <member name="M:ConsoleHello.Class1.Main">
      <summary>
```

```
            Точка входа. Запрашивает имя и_выдает приветствие
        </summary>
    </member>
</members>
</doc>
```

Как видите, отчет описывает наш *проект*, точнее, *сборку*. Пользователь, пожелавший воспользоваться этой *сборкой*, из отчета поймет, что она содержит один класс, назначение которого указано в *теге* `<summary>`. Класс содержит лишь один элемент - точку входа `Main` с заданной спецификацией в *теге* `<summary>`.

Windows-проект

Проделаем аналогичную работу: построим Windows-*проект*, рассмотрим, как он выглядит по умолчанию, а затем дополним его до *проекта* `"Приветствие"`. Повторяя уже описанные действия, в окне нового *проекта* (см. рис. 2.1) я выбрал тип *проекта* `Windows Application`, дав *проекту* имя `WindowsHello`.

Как и в консольном случае, по умолчанию строится *решение*, содержащее единственный *проект*, содержащий единственное *пространство имен* (все три конструкции имеют совпадающие имена). В *пространство имен* вложен единственный класс `Form1`, но это уже далеко не столь простой класс, как ранее. Вначале приведу его код, а потом уже дам необходимые пояснения:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
namespace WindowsHello
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;
        public Form1()
        {
            // Required for Windows Form Designer support
            InitializeComponent();
            // TODO: Add any constructor code after
            // InitializeComponent call
        }
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
```

```
        }
        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new
                System.ComponentModel.Container();
            this.Size = new System.Drawing.Size(300,300);
            this.Text = "Form1";
        }
        #endregion
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

Начну с того, что теперь *пространству имен* предшествует 6 предложений using; это означает, что используются не менее 6-ти классов, находящихся в разных *пространствах имен* библиотеки FCL. Одним из таких используемых классов является класс `Form` из глубоко вложенного *пространства имен* `System.Windows.Forms`. Построенный по умолчанию класс `Form1` является наследником класса `Form` и автоматически наследует его функциональность - свойства, методы, события. При создании объекта этого класса, характеризующего форму, одновременно Visual Studio создает визуальный образ объекта - окно, которое можно заселять элементами управления. В режиме проектирования эти операции можно выполнять вручную, при этом автоматически происходит изменение программного кода класса. Появление в *проекте* формы, открывающейся по умолчанию при запуске *проекта*, означает переход к визуальному, управляемому событиями программированию. Сегодня такой стиль является общепризнанным, а стиль *консольного приложения* следует считать анахронизмом, правда, весьма полезным при изучении свойств языка.

В класс `Form1` встроено закрытое (`private`) свойство - объект `components` класса `Container`. В классе есть конструктор, вызывающий закрытый метод класса `InitializeComponent`. В классе есть деструктор, освобождающий занятые ресурсы, которые могут появляться при добавлении элементов в контейнер `components`. Наконец, в классе есть точка входа - процедура `Main` с непустым телом.

Начало начал - точка "большого взрыва"

Основной операцией, инициирующей вычисления в объектно-ориентированных приложениях, является вызов метода `F` некоторого класса `x`, имеющий вид:

```
x.F(arg1, arg2, ..., argN);
```

В этом вызове x называется целью вызова, и здесь возможны три ситуации:

- `x` - имя класса. В этом случае метод `F` должен быть статическим методом класса, объявленным с *атрибутом* `static`, как это имеет место, например, для точки вызова - процедуры `Main`;

- `x` - имя объекта или объектное выражение. В этом случае `F` должен быть обычным, не статическим методом. Иногда такой метод называют ***экземплярным***, подчеркивая тот факт, что метод вызывается экземпляром класса - некоторым объектом;
- `x` - не указывается при вызове. Такой вызов называется неквалифицированным, в отличие от двух первых случаев. Заметьте, неквалифицированный вызов вовсе не означает, что цель вызова отсутствует, - она просто задана по умолчанию. Целью является текущий объект (текущий класс для статических методов). Текущий объект имеет зарезервированное имя `this`. Применяя это имя, любой неквалифицированный вызов можно превратить в квалифицированный вызов. Иногда без этого имени просто не обойтись.

Но как появляются объекты? Как они становятся текущими? Как реализуется самый первый вызов метода, другими словами, кто и где вызывает точку входа - метод `Main`? С чего все начинается?

Когда CLR получает *сборку* для выполнения, то в *решении*, входящем в *сборку*, отмечен *стартовый проект*, содержащий класс с точкой входа - статическим методом (процедурой) `Main`. Некоторый объект исполнительной среды CLR и вызывает этот метод, так что первоначальный вызов метода осуществляется извне приложения. Это и есть точка "большого взрыва" - начало зарождения мира объектов и объектных вычислений. Дальнейший сценарий зависит от содержимого точки входа. Как правило, в ней создаются один или несколько объектов, а затем вызываются методы и/или обработчики событий, происходящих с созданными объектами. В этих методах и обработчиках событий могут создаваться новые объекты, вызываться новые методы и новые обработчики. Так, начиная с одной точки, разворачивается целый мир объектов приложения.

Выполнение проекта по умолчанию после "большого взрыва"

Давайте посмотрим, что происходит в *проекте*, создаваемом по умолчанию, когда произошел "большой взрыв", вселенная создана и процедура `Main` начала работать. Процедура `Main` содержит всего одну строчку:

```
Application.Run(new Form1());
```

Прокомментируем этот квалифицированный вызов. Целью здесь является класс `Application` из *пространства имен* `System.Windows.Forms`. Класс вызывает статический метод `Run`, которому в качестве фактического аргумента передается объектное выражение `new Form1()`. При вычислении этого выражения создается первый объект - экземпляр класса `Form1`. Этот объект становится текущим. Для создания объекта вызывается конструктор класса. В процессе работы конструктора осуществляется неквалифицированный вызов метода `InitializeComponent()`. Целью этого вызова является текущий объект - уже созданный объект класса `Form1`. Ни в конструкторе, ни в вызванном методе новые объекты не создаются. По завершении работы конструктора объект класса `Form1` передается методу `Run` в качестве аргумента.

Метод `Run` класса `Application` - это знаменитый метод. Во-первых, он открывает форму - видимый образ объекта класса `Form1`, с которой теперь может работать пользователь. Но главная его работа состоит в том, что он создает настоящее Windows-приложение, запуская цикл обработки сообщений о происходящих событиях. Поступающие сообщения обрабатываются операционной системой согласно очереди и приоритетам, вызывая обработчики соответствующих событий. Поскольку наша форма по умолчанию не заселена никакими элементами управления, то поступающих сообщений немного. Все, что может делать пользователь с формой, так это перетаскивать ее по экрану, свертывать и изменять размеры. Конечно, он может еще закрыть форму. Это приведет к

завершению цикла обработки сообщений, к завершению работы метода `Run`, к завершению работы метода `Main`, к завершению работы приложения.

Проект WindowsHello

Давайте расширим приложение по умолчанию до традиционного приветствия в Windows-стиле, добавив окошки для ввода и вывода информации. Как уже говорилось, при создании Windows-приложения по умолчанию создается не только объект класса `Form1` - потомка класса `Form`, но и его видимый образ - форма, с которой можно работать в режиме проектирования, населяя ее элементами управления. Добавим в форму следующие элементы управления:

- текстовое окно и метку. По умолчанию они получат имена `textBox1` и `label1`. Текстовое окно предназначается для ввода имени пользователя, метка, визуально связанная с окном, позволит указать назначение текстового окна. Я установил свойство `Multiline` для текстового окна как `true`, свойство `Text` у метки - Ваше Имя;
- аналогичная пара элементов управления - `textBox2` и `label2` - предназначены для вывода приветствия. Поскольку окно `textBox2` предназначено для вывода, то я включил его свойство ReadOnly;
- я посадил на форму командную кнопку, обработчик события `Click` которой и будет организовывать чтение имени пользователя из окна `textBox1` и вывод приветствия в окно `textBox2`.

На рис. 2.4 показано, как выглядит наша форма в результате проектирования.

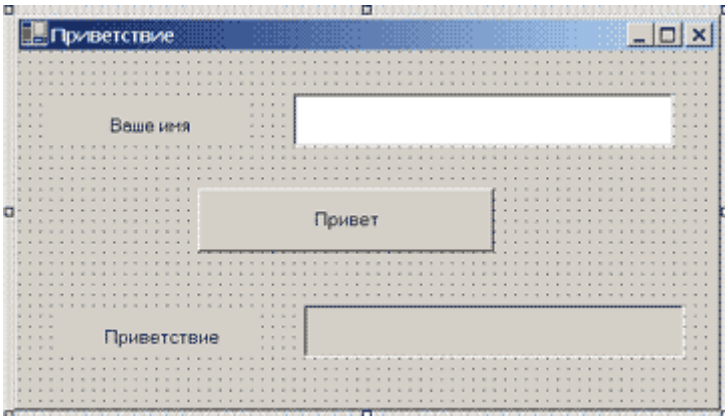

Рис. 2.4. Форма "Приветствие"

Я не буду далее столь же подробно описывать действия по проектированию интерфейса форм, полагая, что все это интуитивно ясно и большинству хорошо знакомо. Более важно понимать то, что все действия по проектированию интерфейса незамедлительно транслируются в программный код, добавляемый в класс **Form1**. Мы вручную сажаем элемент управления на форму, тут же в классе появляется закрытое свойство, задающее этот элемент, а в процедуре **InitailizeComponent** выполняется его инициализация. Мы меняем некоторое свойство элемента управления, это незамедлительно находит отражение в программном коде указанной процедуры.

Вот как выглядит автоматически добавленное в класс описание элементов управления:

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.Label label2;
```

А вот фрагмент текста процедуры InitailizeComponent:

```
#region Windows Form Designer generated code
   /// <summary>
   /// Required method for Designer support - do not
   /// modify the contents of this method with the code
   /// editor.
   /// </summary>
   private void InitializeComponent()
   {
      this.label1 = new System.Windows.Forms.Label();
      this.textBox1 = new System.Windows.Forms.TextBox();
      this.button1 = new System.Windows.Forms.Button();
      this.textBox2 = new System.Windows.Forms.TextBox();
      this.label2 = new System.Windows.Forms.Label();
      this.SuspendLayout();
      // label1
      this.label1.Location = new System.Drawing.Point(24, 40);
      this.label1.Name = "label1";
      this.label1.Size = new System.Drawing.Size(152, 32);
      this.label1.TabIndex = 0;
      this.label1.Text = "Ваше имя";
      this.label1.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
... аналогично задаются описания свойств всех элементов управления ...
... далее задаются свойства самой формы ...
         // Form1
         //
         this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);
         this.ClientSize = new System.Drawing.Size(528, 268);
         this.Controls.AddRange(new
            System.Windows.Forms.Control[]
            {
               this.textBox2,
               this.label2,
               this.button1,
               this.textBox1,
               this.label1
            });
         this.Name = "Form1";
         this.Text = "Приветствие";
         this.Load += new System.EventHandler(this.Form1_Load);
         this.ResumeLayout(false);
      }
      #endregion
```

Заметьте, в *теге* `<summary>` нас предупреждают, что этот метод требуется специальному инструментарию - Дизайнеру формы - и он не предназначен для редактирования пользователем; добавление и удаление кода этого метода производится автоматически. Обращаю внимание, что после заполнения свойств элементов управления заключительным шагом является их добавление в коллекцию `Controls`, хранящую все элементы управления. Здесь используется метод `AddRange`, позволяющий добавить в коллекцию одним махом целый массив элементов управления. Метод `Add` позволяет добавлять в коллекцию по одному элементу. Позже нам придется добавлять элементы управления в форму программно, динамически изменяя интерфейс формы. Для этого мы будем выполнять те же операции: объявить элемент управления, создать его, используя конструкцию new, задать нужные свойства и добавить в коллекцию `Controls`.

В заключение приведу текст обработчика событий командной кнопки. Как задается обработчик того или иного события для элементов управления? Это можно делать по-разному. Есть стандартный способ включения событий. Достаточно выделить нужный

элемент в форме, в окне свойств нажать кнопку событий (со значком молнии) и из списка событий выбрать нужное событие и щелкнуть по нему. В данной ситуации все можно сделать проще - двойной щелчок по кнопке включает событие, и автоматически строится заготовка обработчика события с нужным именем и параметрами. Вот как она выглядит:

```
private void button1_Click(object sender,System.EventArgs e)
   {

   }
```

Нам остается добавить свой текст. Я добавил следующие строки:

```
      string temp;
        temp = textBox1.Text;
      if( temp == "")
         textBox2.Text = "Здравствуй, мир!";
      else
         textBox2.Text = "Здравствуй, " + temp + " !";
```

И вот как это работает.

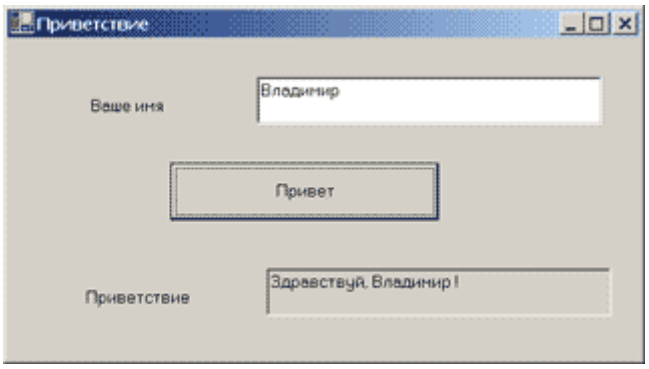

Рис. 2.5. Форма "Приветствие" в процессе работы

На этом мы закончим первое знакомство с *проектами* на C# и в последующих лекциях приступим к систематическому изучению возможностей языка.

3. Лекция: Система типов языка C#

Общий взгляд

Знакомство с новым языком программирования разумно начинать с изучения *системы типов* этого языка. Как в нем устроена *система типов* данных? Какие есть *простые типы*, как создаются *сложные*, структурные типы, как определяются собственные типы, *динамические типы*, как определяются *классы*?

В первых языках программирования понятие *класса* отсутствовало - рассматривались только типы данных. При определении типа явно задавалось только множество возможных значений, которые могут принимать переменные этого типа. Например, тип `integer` задает целые числа в некотором диапазоне. Неявно с типом всегда связывался и набор разрешенных операций. В типизированных языках, к которым относится большинство языков программирования, понятие переменной естественным образом связывалось с типом. Если есть тип `T` и переменная `x` типа `T`, то это означало, что переменная может принимать значения из множества, заданного типом, и к ней применимы операции, разрешенные типом.

Классы и объекты впервые появились в программировании в языке Симула 67. Произошло это спустя 10 лет после появления первого алгоритмического языка Фортран. Определение *класса* наряду с описанием данных содержало четкое определение операций или *методов*, применимых к данным. Объекты - экземпляры *класса* являются обобщением понятия переменной. Сегодня определение *класса* в C# и других объектных языках, аналогично определению типа в CTS, содержит:

- данные, задающие *свойства* объектов *класса*;
- *методы*, определяющие поведение объектов *класса*;
- события, которые могут происходить с объектами *класса*.

Так есть ли различие между этими двумя основополагающими понятиями - типом и *классом*? На первых порах можно считать, что *класс* - это хорошо определенный тип данных, объект - хорошо определенная переменная. Понятия фактически являются синонимами, какое из них употреблять лишь дело вкуса. *Встроенные типы*, такие как `integer` или `string`, предпочитают называть по-прежнему типами, а их экземпляры - переменными. Что же касается абстракции данных, описывающей служащих и названной, например, Employee, то естественнее называть ее *классом*, а ее экземпляры - объектами. Такой взгляд на типы и *классы* довольно полезен, но он не является полным. Позже при обсуждении *классов* и наследования постараемся более четко определить принципиальные различия в этих понятиях.

Объектно-ориентированное программирование, доминирующее сегодня, построено на *классах* и объектах. Тем не менее, понятия типа и переменной все еще остаются центральными при описании языков программирования, что характерно и для языка C#. Заметьте, что и в Framework .Net предпочитают говорить о *системе типов*, хотя все типы *библиотеки FCL* являются *классами*.

Типы данных принято разделять на *простые* и *сложные* в зависимости от того, как устроены их данные. У *простых (скалярных) типов* возможные значения данных едины и неделимы. *Сложные типы* характеризуются способом структуризации данных - одно значение *сложного типа* состоит из множества значений данных, организующих *сложный тип*.

Есть и другие критерии классификации типов. Так, типы разделяются на *встроенные типы* и *типы, определенные программистом (пользователем)*. ***Встроенные типы*** изначально принадлежат языку программирования и составляют его базис. В основе *системы типов* любого языка программирования всегда лежит базисная *система типов*,

встроенных в язык. На их основе программист может строить собственные, им самим определенные типы данных. Но способы (правила) создания таких типов являются базисными, встроенными в язык.

Типы данных разделяются также на *статические* и *динамические*. Для данных ***статического типа*** память отводится в момент объявления, требуемый размер данных (памяти) известен при их объявлении. Для данных ***динамического типа*** размер данных в момент объявления обычно неизвестен и память им выделяется динамически по запросу в процессе выполнения программы.

Еще одна важная классификация типов - это их деление на *значимые* и *ссылочные*. Для ***значимых типов*** значение переменной (объекта) является неотъемлемой собственностью переменной (точнее, собственностью является память, отводимая значению, а само значение может изменяться). Для ***ссылочных типов*** значением служит *ссылка* на некоторый объект в памяти, расположенный обычно в динамической памяти - "куче". Объект, на который указывает *ссылка*, может быть разделяемым. Это означает, что несколько ссылочных переменных могут указывать на один и тот же объект и разделять его значения. *Значимый тип* принято называть развернутым, подчеркивая тем самым, что значение объекта развернуто непосредственно в памяти, отводимой объекту. О *ссылочных* и *значимых типах* еще предстоит обстоятельный разговор.

Для большинства процедурных языков, реально используемых программистами - Паскаль, C++, Java, Visual Basic, C#, - система *встроенных типов* более или менее одинакова. Всегда в языке присутствуют арифметический, *логический (булев)*, *символьный типы*. Арифметический тип всегда разбивается на подтипы. Всегда допускается организация данных в виде *массивов* и записей (*структур*). Внутри арифметического типа всегда допускаются **преобразования**, всегда есть функции, преобразующие строку в число и обратно. Так что, мой читатель, Ваше знание, по крайней мере, одного из процедурных языков позволяет построить общую картину *системы типов* и для языка C#. Отличия будут в нюансах, которые и придают аромат и неповторимость языку.

Поскольку язык C# является непосредственным *потомком* языка C++, то и *системы типов* этих двух языков близки и совпадают вплоть до названия типов и областей их определения. Но отличия, в том числе принципиального характера, есть и здесь.

Система типов

Давайте рассмотрим, как устроена *система типов* в языке C#, но вначале для сравнения приведу классификацию типов в стандарте языка C++.

Стандарт языка C++ включает следующий набор **фундаментальных** типов.

1. *Логический* тип (`bool`).
2. *Символьный* тип (`char`).
3. *Целые* типы. *Целые типы* могут быть одного из трех размеров - `short`, `int`, `long`, сопровождаемые описателем `signed` или `unsigned`, который указывает, как интерпретируется значение, - со знаком или без оного.
4. Типы с *плавающей точкой*. Эти типы также могут быть одного из трех размеров - `float`, `double`, `long double`.

   Кроме того, в языке есть

5. Тип *void*, используемый для указания на отсутствие информации.

   Язык позволяет конструировать типы.

6. *Указатели* (например, `int*` - типизированный *указатель* на переменную типа `int`).
7. *Ссылки* (например, `double&` - типизированная *ссылка* на переменную типа `double`).
8. *Массивы* (например, `char[]` - *массив* элементов типа `char`).

   Язык позволяет конструировать пользовательские типы

9. **Перечислимые** типы (`enum`) для представления значений из конкретного множества.
10. *Структуры* (`struct`).
11. *Классы*.

Первые три вида типов называются интегральными или счетными. Значения их перечислимы и упорядочены. *Целые типы* и *типы с плавающей точкой* относятся к арифметическому типу. Типы подразделяются также на *встроенные* и *типы, определенные пользователем*.

Эта схема типов сохранена и в языке C#. Однако здесь на верхнем уровне используется и другая классификация, носящая для C# принципиальный характер. Согласно этой классификации все типы можно разделить на четыре категории:

1. *Типы-значения* (`value`), или *значимые типы*.
2. *Ссылочные* (`reference`).
3. *Указатели* (`pointer`).
4. Тип *void*.

Эта классификация основана на том, где и как хранятся значения типов. Для *ссылочного типа* значение задает *ссылку* на область памяти в "куче", где расположен соответствующий объект. Для *значимого типа* используется прямая адресация, значение хранит собственно данные, и память для них отводится, как правило, в стеке.

В отдельную категорию выделены *указатели*, что подчеркивает их особую роль в языке. *Указатели* имеют ограниченную область действия и могут использоваться только в небезопасных блоках, помеченных как `unsafe`.

Особый статус имеет и тип *void*, указывающий на отсутствие какого-либо значения.

В языке C# жестко определено, какие типы относятся к *ссылочным*, а какие - к *значимым*. К *значимым типам* относятся: *логический*, арифметический, *структуры*, *перечисление*. *Массивы*, **строки** и *классы* относятся к *ссылочным типам*. На первый взгляд, такая классификация может вызывать некоторое недоумение, почему это *структуры*, которые в C++ близки к *классам*, относятся к *значимым типам*, а *массивы* и строки - к *ссылочным*. Однако ничего удивительного здесь нет. В C# *массивы* рассматриваются как динамические, их размер может определяться на этапе вычислений, а не в момент трансляции. Строки в C# также рассматриваются как динамические переменные, длина которых может изменяться. Поэтому строки и *массивы* относятся к *ссылочным типам*, требующим распределения памяти в "куче".

Со *структурами* дело сложнее. *Структуры* C# представляют частный случай *класса*. Определив свой *класс* как *структуру*, программист получает возможность отнести *класс* к *значимым типам*, что иногда бывает крайне полезно. Замечу, что в хорошем объектном языке Eiffel программист может любой *класс* объявить развернутым (`expanded`), что эквивалентно отнесению к *значимому типу*. У программиста C# только благодаря *структурам* появляется возможность управлять отнесением *класса* к *значимым* или *ссылочным типам*. Правда, это неполноценное средство, поскольку на

структуры накладываются дополнительные ограничения по сравнению с обычными *классами*.

Рассмотрим классификацию, согласно которой все типы делятся на *встроенные* и *определенные пользователем*. Все *встроенные типы* C# однозначно отображаются, а фактически совпадают с системными типами каркаса Net Framework, размещенными в пространстве имен `System`. Поэтому всюду, где можно использовать имя типа, например, - `int`, с тем же успехом можно использовать и имя `System.Int32`.

Замечание:

Следует понимать тесную связь и идентичность *встроенных типов* языка C# и типов каркаса. Какими именами типов следует пользоваться в программных текстах - это спорный вопрос. Джеффри Рихтер в своей известной книге "Программирование на платформе Framework .Net" рекомендует использовать системные имена. Другие авторы считают, что следует пользоваться именами типов, принятыми в языке. Возможно, в модулях, предназначенных для межъязыкового взаимодействия, разумны системные имена, а в остальных случаях - имена конкретного языка программирования.

В заключение этого раздела приведу таблицу (3.1), содержащую описание всех *встроенных типов* языка C# и их основные характеристики.

Логический тип			
Имя типа	**Системный тип**	**Значения**	**Размер**
`Bool`	`System.Boolean`	`true, false`	8 бит
Арифметические целочисленные типы			
Имя типа	**Системный тип**	**Диапазон**	**Размер**
`Sbyte`	`System.SByte`	-128 — 127	Знаковое, 8 Бит
`Byte`	`System.Byte`	0 — 255	Беззнаковое, 8 Бит
`Short`	`System.Short`	-32768 —32767	Знаковое, 16 Бит
`Ushort`	`System.UShort`	0 — 65535	Беззнаковое, 16 Бит
`Int`	`System.Int32`	≈(-2*10^9 — 2*10^9)	Знаковое, 32 Бит
`Uint`	`System.UInt32`	≈(0 — 4*10^9)	Беззнаковое, 32 Бит
`Long`	`System.Int64`	≈(-9*10^18 — 9*10^18)	Знаковое, 64 Бит
`Ulong`	`System.UInt64`	≈(0— 18*10^18)	Беззнаковое, 64 Бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
`Float`	`System.Single`	+1.5*10^-45 - +3.4*10^38	7 цифр
`Double`	`System.Double`	+5.0*10^-324 - +1.7*10^308	15-16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
`Decimal`	`System.Decimal`	+1.0*10^-28 - +7.9*10^28	28-29 значащих цифр
Символьные типы			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
`Char`	`System.Char`	U+0000 - U+ffff	16 бит Unicode символ
`String`	`System.String`	Строка из символов Unicode	
Объектный тип			
Имя типа	**Системный тип**	**Примечание**	
Object	`System.Object`	Прародитель всех *встроенных* и пользовательских типов	

Система *встроенных типов* языка C# не только содержит практически все *встроенные типы* (за исключением `long double`) стандарта языка C++, но и перекрывает его разумным образом. В частности тип `string` является *встроенным* в язык, что вполне естественно. В области совпадения сохранены имена типов, принятые в C++, что облегчает жизнь тем, кто привык работать на C++, но собирается по тем или иным причинам перейти на язык C#.

Типы или классы? И типы, и классы

Язык C# в большей степени, чем язык C++, является языком объектного программирования. В чем это выражается? В языке C# сглажено различие между типом и *классом*. Все типы - *встроенные* и пользовательские - одновременно являются *классами*, связанными *отношением наследования*. *Родительским*, *базовым классом* является *класс Object*. Все остальные типы или, точнее, *классы* являются его *потомками*, наследуя *методы* этого *класса*. У *класса Object* есть четыре наследуемых *метода*:

1. `bool Equals (object obj)` - проверяет эквивалентность текущего объекта и объекта, переданного в качестве аргумента;
2. `System.Type GetType ()` - возвращает системный тип текущего объекта;
3. `string ToString ()` - возвращает строку, связанную с объектом. Для арифметических типов возвращается значение, преобразованное в строку;
4. `int GetHashCode()` - служит как хэш-функция в соответствующих алгоритмах поиска по ключу при хранении данных в хэш-таблицах.

Естественно, что все *встроенные типы* нужным образом переопределяют *методы* родителя и добавляют собственные *методы* и *свойства*. Учитывая, что и типы, создаваемые пользователем, также являются *потомками класса Object*, то для них необходимо переопределить *методы* родителя, если предполагается использование этих *методов*; реализация родителя, предоставляемая по умолчанию, не обеспечивает нужного эффекта.

Перейдем теперь к примерам, на которых будем объяснять дальнейшие вопросы, связанные с типами и *классами*, переменными и объектами. Начнем с вполне корректного в языке C# примера объявления переменных и присваивания им значений:

```
int x=11;
int v = new Int32();
v = 007;
string s1 = "Agent";
s1 = s1 + v.ToString() +x.ToString();
```

В этом примере переменная `x` объявляется как обычная переменная типа `int`. В то же время для объявления переменной `v` того же типа `int` используется стиль, принятый для объектов. В объявлении применяется конструкция `new` и вызов конструктора *класса*. В операторе присваивания, записанном в последней строке фрагмента, для обеих переменных вызывается *метод* `ToString`, как это делается при работе с объектами. Этот *метод*, наследуемый от *родительского класса Object*, переопределенный в *классе* `int`, возвращает строку с записью целого. Сообщу еще, что *класс* `int` не только наследует *методы* родителя - *класса Object*, - но и дополнительно определяет *метод* `CompareTo`, выполняющий сравнение целых, и *метод* `GetTypeCode`, возвращающий системный код типа. Для *класса* `Int` определены также статические *методы* и поля, о которых расскажу чуть позже.

Так что же такое после этого `int`, спросите Вы: тип или *класс*? Ведь ранее говорилось, что `int` относится к value-типам, следовательно, он хранит в стеке значения своих

переменных, в то время как объекты должны задаваться *ссылками*. С другой стороны, создание экземпляра с помощью конструктора, вызов *методов*, наконец, существование *родительского класса Object*, - все это указывает на то, что `int` - это настоящий *класс*. Правильный ответ состоит в том, что `int` - это и тип, и *класс*. В зависимости от контекста `x` может восприниматься как переменная типа `int` или как объект *класса* `int`. Это же верно и для всех остальных value- типов. Замечу еще, что все *значимые типы* фактически реализованы как *структуры*, представляющие частный случай *класса*.

Остается понять, для чего в языке C# введена такая двойственность. Для `int` и других *значимых типов* сохранена концепция типа не только из-за ностальгических воспоминаний о типах. Дело в том, что *значимые типы* эффективнее в реализации, им проще отводить память, так что именно соображения эффективности реализации заставили авторов языка сохранить *значимые типы*. Более важно, что зачастую необходимо оперировать значениями, а не *ссылками* на них, хотя бы из-за различий в семантике присваивания для переменных *ссылочных* и *значимых типов*.

С другой стороны, в определенном контексте крайне полезно рассматривать переменные типа `int` как настоящие объекты и обращаться с ними как с объектами. В частности, полезно иметь возможность создавать и работать со списками, чьи элементы являются разнородными объектами, в том числе и принадлежащими к *значимым типам*.

Дальнейшие примеры работы с типами и проект Types

Обсуждение особенностей тех или иных конструкций языка невозможно без приведения примеров. Для каждой лекции я строю один или несколько проектов, сохраняя по возможности одну и ту же схему и реально выполняя проекты в среде Visual Studio .Net. Для работы с примерами данной лекции построен консольный проект с именем `Types`, содержащий два *класса*: `Class1` и `Testing`. Расскажу чуть подробнее о той схеме, по которой выстраиваются проекты. *Класс* `Class1` строится автоматически при начальном создании проекта. Он содержит процедуру `Main` - точку входа в проект. В процедуре `Main` создается объект *класса* `Testing` и вызываются *методы* этого *класса*, тестирующие те или иные ситуации. Для решения специальных задач, помимо всегда создаваемого *класса* `Testing`, создаются один или несколько *классов*. Добавление нового *класса* в проект я осуществляю выбором пункта меню **Project/Add Class**. В этом случае автоматически строится заготовка для нового *класса*, содержащая конструктор без параметров. Дальнейшая работа над *классом* ведется над этой заготовкой. Создаваемые таким образом *классы* хранятся в проекте в отдельных файлах. Это особенно удобно, если *классы* используются в разных проектах. Функционально связанную группу *классов* удобнее хранить в одном файле, что не возбраняется.

Все проекты в книге являются самодокументируемыми. *Классы* и их *методы* сопровождаются *тегами* <*summary*>. В результате появляются подсказки при вызове *методов* и возможность построения *XML-отчета*, играющего роль спецификации проекта.

Приведу текст *класса* `Class1`:

```
using System;
namespace Types
{
   /// <summary>
   /// Проект Types содержит примеры, иллюстрирующие работу
   /// со встроенными скалярными типами языка C#.
   /// Проект содержит классы: Testing, Class1.
   ///
   /// </summary>
      class Class1
```

```
    {
        /// <summary>
        /// Точка входа проекта.
        /// В ней создается объект класса Testing
        /// и вызываются его методы.
                  /// </summary>
        [STAThread]
        static void Main()
        {
            Testing tm = new Testing();
            Console.WriteLine("Testing.Who Test");
            tm.WhoTest();
            Console.WriteLine("Testing.Back Test");
            tm.BackTest();
            Console.WriteLine("Testing.OLoad Test");
            tm.OLoadTest();
            Console.WriteLine("Testing.ToString Test");
            tm.ToStringTest();
            Console.WriteLine("Testing.FromString Test");
            tm.FromStringTest();
            Console.WriteLine("Testing.CheckUncheck Test");
            tm.CheckUncheckTest();
        }
    }
}
```

Класс `Class1` содержит точку входа `Main` и ничего более. В процедуре `Main` создается объект `tm` *класса* `Testing`, затем поочередно вызываются семь *методов* этого *класса*. Каждому вызову предшествует выдача соответствующего сообщения на консоль. Каждый *метод* - это отдельный пример, подлежащий обсуждению.

Семантика присваивания

Рассмотрим присваивание:

```
x = e.
```

Чтобы присваивание было допустимым, типы переменной `x` и выражения `e` должны быть согласованными. Пусть сущность `x` согласно объявлению принадлежит *классу* `T`. Будем говорить, что тип `T` основан на *классе* `T` и является базовым типом `x`, так что базовый тип определяется *классом* объявления. Пусть теперь в рассматриваемом нами присваивании выражение e связано с объектом типа `T1`.

Определение: тип `T1` ***согласован по присваиванию*** с базовым типом `T` переменной `x`, если *класс* `T1` является *потомком класса* `T`.

Присваивание допустимо, если и только если имеет место *согласование типов*. Так как все *классы* в языке C# - встроенные и определенные пользователем - по определению являются *потомками класса Object*, то отсюда и следует наш частный случай - переменным *класса Object* можно присваивать выражения любого типа.

Несмотря на то, что обстоятельный разговор о наследовании, родителях и *потомках* нам еще предстоит, лучше с самого начала понимать отношения между *родительским классом* и классом-потомком, отношения между объектами этих *классов*. Класс-потомок при создании **наследует** все *свойства* и *методы* родителя. *Родительский класс* не имеет возможности наследовать *свойства* и *методы*, создаваемые его *потомками*. Наследование - это односторонняя операция от родителя к *потомку*. Ситуация с присваиванием симметричная. Объекту *родительского класса* **присваивается** объект класса-потомка. Объекту класса-потомка не может быть присвоен объект *родительского*

класса. Присваивание - это односторонняя операция от *потомка* к родителю. Одностороннее присваивание реально означает, что ссылочная переменная *родительского класса* может быть связана с любыми объектами, имеющими тип *потомков родительского класса*.

Например, пусть задан некоторый *класс* `Parent`, а *класс* `Child` - его *потомок*, объявленный следующим образом:

```
class Child:Parent {...}
```

Пусть теперь в некотором *классе*, являющемся клиентом *классов* `Parent` и `Child`, объявлены переменные этих *классов* и созданы связанные с ними объекты:

```
Parent p1 = new Parent(), p2 = new Parent();
Child ch1 = new Child(), ch2 = new Child();
```

Тогда допустимы присваивания:

```
p1 = p2; p2= p1; ch1=ch2; ch2 = ch1; p1 = ch1; p1 = ch2;
```

Но недопустимы присваивания:

```
ch1 = p1; ch2 = p1; ch2 = p2;
```

Заметьте, ситуация не столь удручающая - сын может вернуть себе переданный родителю объект, задав *явное преобразование*. Так что следующие присваивания допустимы:

```
p1 = ch1; ... ch1 = (Child)p1;
```

Семантика присваивания справедлива и для другого важного случая - при рассмотрении соответствия между формальными и фактическими аргументами процедур и функций. Если формальный аргумент согласно объявлению имеет тип `T`, а выражение, задающее фактический аргумент, имеет тип `T1`, то имеет место *согласование типов* **формального и фактического аргумента**, если и только если *класс* `T1` является *потомком класса* `T`. Отсюда незамедлительно следует, что если формальный параметр процедуры принадлежит *классу Object*, то фактический аргумент может быть выражением любого типа.

Преобразование к типу object

Рассмотрим частный случай присваивания `x = e;` когда `x` имеет тип `object`. В этом случае гарантируется полная *согласованность по присваиванию* - выражение `e` может иметь любой тип. В результате присваивания значением переменной `x` становится *ссылка* на объект, заданный выражением `e`. Заметьте, текущим типом `x` становится тип объекта, заданного выражением `e`. Уже здесь проявляется одно из важных различий между *классом* и типом. Переменная, лучше сказать сущность `x`, согласно объявлению принадлежит *классу Object*, но ее тип - тип того объекта, с которым она связана в текущий момент, - может динамически изменяться.

Примеры преобразований

Перейдем к примерам. *Класс* `Testing`, содержащий примеры, представляет собой набор данных разного типа, над которыми выполняются операции, иллюстрирующие преобразования типов. Вот описание *класса* `Testing`:

```
using System;
namespace Types
{
   /// <summary>
   /// Класс Testing включает данные разных типов. Каждый его
   /// открытый метод описывает некоторый пример,
   /// демонстрирующий работу с типами.
   /// Открытые методы могут вызывать закрытые методы класса.
   /// </summary>
   public class Testing
   {
      /// <summary>
      /// набор скалярных данных разного типа.
      /// </summary>
      byte b = 255;
      int x = 11;
      uint ux = 1111;
      float y = 5.5f;
      double dy = 5.55;
      string s = "Hello!";
      string s1 = "25";
      object obj = new Object();
      // Далее идут методы класса, приводимые по ходу
      // описания примеров
   }
```

В набор данных *класса* входят скалярные данные арифметического типа, относящиеся к *значимым типам*, переменные строкового типа и типа `object`, принадлежащие *ссылочным типам*. Рассмотрим закрытый (`private`) *метод* этого *класса* - процедуру `WhoIsWho` с формальным аргументом *класса Object*. Процедура выводит на консоль переданное ей имя аргумента, его тип и значение. Вот ее текст:

```
/// <summary>
/// Метод выводит на консоль информацию о типе и
/// значении фактического аргумента. Формальный
/// аргумент имеет тип object. Фактический аргумент
/// может иметь любой тип, поскольку всегда
/// допустимо неявное преобразование в тип object.
/// </summary>
/// <param name="name"> - Имя второго аргумента</param>
/// <param name="any"> - Допустим аргумент любого типа</param>
void WhoIsWho(string name, object any)
{
   Console.WriteLine("type {0} is {1} , value is {2}",
      name, any.GetType(), any.ToString());
}
```

Вот открытый (`public`) *метод класса* `Testing`, в котором многократно вызывается *метод* `WhoIsWho` с аргументами разного типа:

```
/// <summary>
/// получаем информацию о типе и значении
/// переданного аргумента - переменной или выражения
/// </summary>
public void WhoTest()
{
   WhoIsWho("x",x);
   WhoIsWho("ux",ux);
   WhoIsWho("y",y);
   WhoIsWho("dy",dy);
   WhoIsWho("s",s);
   WhoIsWho("11 + 5.55 + 5.5f",11 + 5.55 + 5.5f);
```

```
    obj = 11 + 5.55 + 5.5f;
    WhoIsWho("obj",obj);
}
```

Заметьте, сущность `any` - формальный аргумент *класса Object* при каждом вызове - динамически изменяет тип, связываясь с объектом, заданным фактическим аргументом. Поэтому тип аргумента, выдаваемый на консоль, - это тип фактического аргумента. Заметьте также, что наследуемый от *класса Object метод* `GetType` возвращает тип *FCL*, то есть тот тип, на который отражается тип языка и с которым реально идет работа при выполнении модуля. В большинстве вызовов фактическим аргументом является переменная - соответствующее *свойство класса* `Testing`, но в одном случае передается обычное арифметическое выражение, автоматически преобразуемое в объект. Аналогичная ситуация имеет место и при выполнении присваивания в рассматриваемой процедуре.

На рис. 3.1 показаны результаты вывода на консоль, полученные при вызове *метода* `WhoTest` в приведенной выше процедуре `Main` *класса* `Class1`.

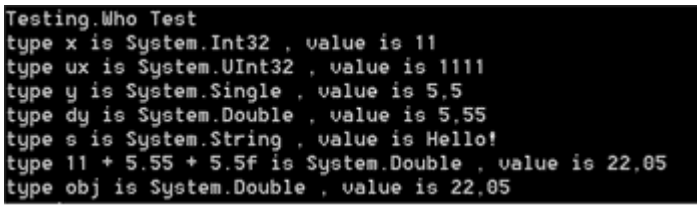

Рис. 3.1. Вывод на печать результатов теста WhoTest

Семантика присваивания. Преобразования между ссылочными и значимыми типами

Рассматривая семантику присваивания и передачи аргументов, мы обошли молчанием один важный вопрос. Будем называть *целью* левую часть оператора присваивания, а также формальный аргумент при передаче аргументов в процедуру или функцию. Будем называть *источником* правую часть оператора присваивания, а также фактический аргумент при передаче аргументов в процедуру или функцию. Поскольку *источник* и *цель* могут быть как *значимого*, так и *ссылочного типа*, то возможны четыре различные комбинации. Рассмотрим их подробнее.

- *Цель* **и** *источник значимого типа*. Здесь наличествует семантика значимого присваивания. В этом случае *источник* и *цель* имеют собственную память для хранения значений. Значения *источника* заменяют значения соответствующих полей *цели*. *Источник* и *цель* после этого продолжают жить независимо. У них своя память, хранящая после присваивания одинаковые значения.
- *Цель* **и** *источник ссылочного типа*. Здесь имеет место семантика ссылочного присваивания. В этом случае значениями *источника* и *цели* являются *ссылки* на объекты, хранящиеся в памяти ("куче"). При ссылочном присваивании *цель* разрывает связь с тем объектом, на который она ссылалась до присваивания, и становится *ссылкой* на объект, связанный с *источником*. Результат ссылочного присваивания двоякий. Объект, на который ссылалась *цель*, теряет одну из своих ссылок и может стать висячим, так что его дальнейшую судьбу определит сборщик мусора. С объектом в памяти, на который ссылался *источник*, теперь связываются, по меньшей мере, две *ссылки*, рассматриваемые как различные имена одного объекта. Ссылочное присваивание приводит к созданию псевдонимов - к появлению разных имен у одного объекта. Особо следует учитывать ситуацию, когда *цель* и/или *источник* имеет значение *void*. Если такое значение имеет *источник*, то в результате присваивания *цель* получает это значение и более не ссылается ни на какой объект. Если же *цель* имела

значение `void`, а *источник* - нет, то в результате присваивания ранее "висячая" *цель* становится *ссылкой* на объект, связанный с *источником*.

- *Цель ссылочного типа*, *источник значимого типа*. В этом случае "на лету" *значимый тип* преобразуется в *ссылочный*. Как обеспечивается двойственность существования *значимого* и *ссылочного типа* - переменной и объекта? Ответ прост: за счет специальных, эффективно реализованных операций, преобразующих переменную *значимого типа* в объект и обратно. Операция "***упаковать***" (boxing) выполняется автоматически и неявно в тот момент, когда по контексту требуется объект, а не переменная. Например, при вызове процедуры `WhoIsWho` требуется, чтобы аргумент `any` был объектом. Если фактический аргумент является переменной *значимого типа*, то автоматически выполняется операция "*упаковать*". При ее выполнении создается настоящий объект, хранящий значение переменной. Можно считать, что происходит упаковка переменной в объект. Необходимость в упаковке возникает достаточно часто. Примером может служить и процедура консольного вывода `WriteLine` *класса* `Console`, которой требуются объекты, а передаются зачастую переменные *значимого типа*.
- *Цель значимого типа*, *источник ссылочного типа*. В этом случае "на лету" *ссылочный тип* преобразуется в *значимый*. Операция "***распаковать***" (unboxing) выполняет обратную операцию, - она "сдирает" объектную упаковку и извлекает хранимое значение. Заметьте, операция "*распаковать*" не является обратной к операции "*упаковать*" в строгом смысле этого слова. Оператор `obj = x` корректен, но выполняемый следом оператор `x = obj` приведет к ошибке. Недостаточно, чтобы хранимое значение в упакованном объекте точно совпадало по типу с переменной, которой присваивается объект. Необходимо *явно заданное преобразование* к нужному типу.

Операции "упаковать" и "распаковать" (boxing и unboxing).

Примеры

В нашем следующем примере демонстрируется применение обеих операций - упаковки и распаковки. Поскольку формальный аргумент процедуры `Back` принадлежит *классу Object*, то при передаче фактического аргумента *значимого типа* происходит упаковка значения в объект. Этот объект и возвращается процедурой. Его *динамический тип* определяется тем объектом памяти, на который указывает *ссылка*. Когда возвращаемый результат присваивается переменной *значимого типа*, то, несмотря на совпадение типа переменной с *динамическим типом* объекта, необходимо выполнить распаковку, "содрать" объектную упаковку и вернуть непосредственное значение. Вот как выглядит процедура `Back` и тестирующая ее процедура `BackTest` из *класса* `Testing`:

```
/// <summary>
/// Возвращает переданный ему аргумент.
/// Фактический аргумент может иметь произвольный тип.
/// Возвращается всегда объект класса object.
/// Клиент, вызывающий метод, должен при необходимости
/// задать явное преобразование получаемого результата
/// </summary>
/// <param name="any"> Допустим любой аргумент</param>
/// <returns></returns>
object Back(object any)
{
   return(any);
}
/// <summary>
/// Неявное преобразование аргумента в тип object
/// Явное приведение типа результата.
/// </summary>
public void BackTest()
```

```
{
   ux = (uint)Back(ux);
   WhoIsWho("ux",ux);
   s1 = (string)Back(s);
   WhoIsWho("s1",s1);
   x =(int)(uint)Back(ux);
   WhoIsWho("x",x);
   y = (float)(double)Back(11 + 5.55 + 5.5f);
   WhoIsWho("y",y);
}
```

Обратите внимание, что если *значимый тип* в левой части оператора присваивания не совпадает с *динамическим типом* объекта, то могут потребоваться две операции приведения. Вначале нужно распаковать значение, а затем привести его к нужному типу, что и происходит в двух последних операторах присваивания. Приведу результаты вывода на консоль, полученные при вызове процедуры `BackTest` в процедуре `Main`.

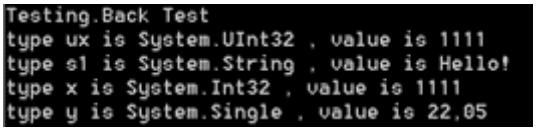

Рис. 3.2. Вывод на печать результатов теста BackTest

Две двойственные операции "*упаковать*" и "*распаковать*" позволяют, в зависимости от контекста, рассматривать *значимые типы* как *ссылочные*, переменные как объекты, и наоборот.

4. Лекция: Преобразования типов

Где, как и когда выполняются преобразования типов?

Необходимость в преобразовании типов возникает в **выражениях**, **присваиваниях**, **замене формальных аргументов** *метода* **фактическими**.

Если при вычислении выражения операнды операции имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа `byte` должны быть приведены к типу `int`, поскольку сложение не определено над байтами. При выполнении присваивания `x=e` тип *источника* `e` и тип *цели* `x` должны быть *согласованы*. Аналогично, при вызове *метода* также должны быть согласованы типы *источника* и *цели* - фактического и формального аргументов.

Преобразования ссылочных типов

Поскольку операции над ссылочными типами не определены (*исключением* являются строки, но операции над ними, в том числе и присваивание, выполняются как над значимыми типами), то необходимость в них возникает только при присваиваниях и вызовах *методов*. Семантика таких преобразований рассмотрена в предыдущей лекции 3, где подробно обсуждалась семантика присваивания и совпадающая с ней семантика замены формальных аргументов фактическими. Там же много говорилось о преобразованиях между ссылочными и значимыми типами, выполняемых при этом операциях упаковки значений в объекты и обратной их распаковки.

Коротко повторю основные положения, связанные с преобразованиями ссылочных типов. При присваиваниях (замене аргументов) тип *источника* должен быть *согласован* с типом *цели*, то есть объект, связанный с *источником*, должен принадлежать классу, являющемуся *потомком* класса *цели*. В случае согласования типов, ссылочная переменная *цели* связывается с объектом *источника* и ее тип динамически изменяется, становясь типом *источника*. Это преобразование выполняется автоматически и *неявно*, не требуя от программиста никаких дополнительных указаний. Если же тип *цели* является *потомком* типа *источника*, то *неявное преобразование* отсутствует, даже если объект, связанный с *источником*, принадлежит типу *цели*. *Явное преобразование*, заданное программистом, позволяет справиться с этим случаем. Ответственность за корректность *явных преобразований* лежит на программисте, так что может возникнуть ошибка на этапе выполнения, если связываемый объект реально не является объектом класса *цели*. За примерами следует обратиться к лекции 3, еще раз обратив внимание на присваивания объектов классов `Parent` и `Child`.

Преобразования типов в выражениях

Продолжая тему преобразований типов, рассмотрим привычные для программистов *преобразования между значимыми типами* и, прежде всего, преобразования внутри арифметического типа.

В C# такие преобразования делятся на *неявные* и *явные*. К ***неявным*** относятся те преобразования, результат выполнения которых всегда успешен и не приводит к потере точности данных. *Неявные преобразования* выполняются автоматически. Для арифметических данных это означает, что в *неявных преобразованиях* диапазон типа назначения содержит в себе диапазон исходного типа. Например, преобразование из типа `byte` в тип `int` относится к *неявным*, поскольку диапазон типа `byte` является подмножеством диапазона `int`. Это преобразование всегда успешно и не может приводить к потере точности. Заметьте, преобразования из целочисленных типов к типам с плавающей точкой относятся к *неявным*. Хотя здесь и может происходить

некоторое искажение значения, но точность представления значения сохраняется, например, при преобразовании из `long` в `double` порядок значения остается неизменным.

К ***явным*** относятся разрешенные преобразования, успех выполнения которых не гарантируется или может приводить к потере точности. Такие потенциально опасные преобразования должны быть явно заданы программистом. Преобразование из типа `int` в тип `byte` относится к *явным*, поскольку оно небезопасно и может приводить к потере значащих цифр. Заметьте, не для всех типов существуют *явные преобразования*. В этом случае требуются другие механизмы преобразования типов, которые будут рассмотрены позже.

Преобразования внутри арифметического типа

Арифметический тип, как показано в таблице 3.1, распадается на 11 подтипов. На рис. 4.1 показана схема преобразований внутри арифметического типа.

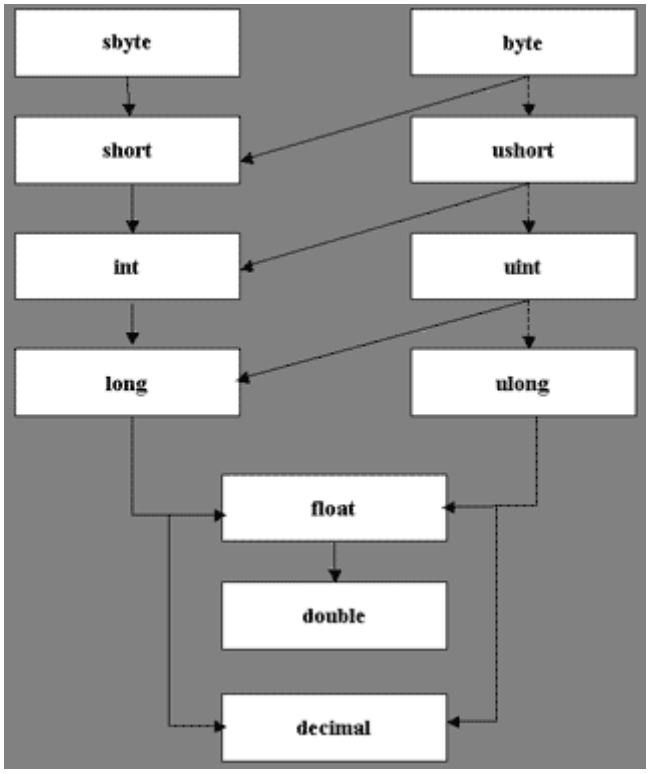

Рис. 4.1. Иерархия преобразований внутри арифметического типа

Диаграмма, приведенная на рисунке, позволяет ответить на ряд важных вопросов, связанных с существованием преобразований между типами. Если на диаграмме задан путь (стрелками) от типа `A` к типу `B`, то это означает существование *неявного преобразования* из типа `A` в тип `B`. Все остальные преобразования между подтипами арифметического типа существуют, но являются *явными*. Заметьте, что циклов на диаграмме нет, все стрелки односторонние, так что преобразование, обратное к *неявному*, всегда должно быть задано явным образом.

Путь, указанный на диаграмме, может быть достаточно длинным, но это вовсе не означает, что выполняется вся последовательность преобразований на данном пути. Наличие пути говорит лишь о существовании *неявного преобразования*, а само преобразование выполняется только один раз, - из типа *источника* `A` в тип назначения `B`.

Иногда возникает ситуация, при которой для одного типа *источника* может одновременно существовать несколько типов назначений и необходимо осуществить выбор *цели* - типа назначения. Такие проблемы выбора возникают, например, при работе с перегруженными *методами* в классах.

Понятие перегрузки *методов* и операций подробно будет рассмотрено в последующих лекциях (см. лекцию 8).

Диаграмма, приведенная на рис. 4.1, и в этом случае помогает понять, как делается выбор. Пусть существует две или более реализации перегруженного *метода*, отличающиеся типом формального аргумента. Тогда при вызове этого *метода* с аргументом типа `T` может возникнуть проблема, какую реализацию выбрать, поскольку для нескольких реализаций может быть допустимым преобразование аргумента типа `T` в тип, заданный формальным аргументом данной реализации *метода*. Правило выбора реализации при вызове *метода* таково: выбирается та реализация, для которой путь преобразований, заданный на диаграмме, короче. Если есть точное соответствие параметров по типу (путь длины 0), то, естественно, именно эта реализация и будет выбрана.

Давайте рассмотрим еще один тестовый пример. В класс `Testing` включена группа перегруженных *методов* `OLoad` с одним и двумя аргументами. Вот эти *методы*:

```
/// <summary>
/// Группа перегруженных методов OLoad
/// с одним или двумя аргументами арифметического типа.
/// Если фактический аргумент один, то будет вызван один из
/// методов, наиболее близко подходящий по типу аргумента.
/// При вызове метода с двумя аргументами возможен
/// конфликт выбора подходящего метода, приводящий
/// к ошибке периода компиляции.
/// </summary>
void OLoad(float par)
{
   Console.WriteLine("float value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа long
/// </summary>
/// <param name="par"></param>
void OLoad(long par)
{
   Console.WriteLine("long value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа ulong
/// </summary>
/// <param name="par"></param>
void OLoad(ulong par)
{
   Console.WriteLine("ulong value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с одним параметром типа double
/// </summary>
/// <param name="par"></param>
void OLoad(double par)
{
   Console.WriteLine("double value {0}", par);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа long и long
```

```
/// </summary>
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(long par1, long par2)
{
   Console.WriteLine("long par1 {0}, long par2 {1}", par1, par2);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа
/// double и double
/// </summary>
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(double par1, double par2)
{
   Console.WriteLine("double par1 {0}, double par2 {1}",par1, par2);
}
/// <summary>
/// Перегруженный метод OLoad с двумя параметрами типа
/// int и float
/// </summary>
/// <param name="par1"></param>
/// <param name="par2"></param>
void OLoad(int par1, float par2)
{
   Console.WriteLine("int par1 {0}, float par2 {1}",par1, par2);
}
```

Все эти *методы* устроены достаточно просто. Они сообщают информацию о типе и значении переданных аргументов. Вот тестирующая процедура, вызывающая *метод* `OLoad` с разным числом и типами аргументов:

```
/// <summary>
/// Вызов перегруженного метода OLoad. В зависимости от
/// типа и числа аргументов вызывается один из методов группы.
/// </summary>
public void OLoadTest()
{
   OLoad(x); OLoad(ux);
   OLoad(y); OLoad(dy);
   // OLoad(x,ux);
   // conflict: (int, float) и (long,long)
   OLoad(x,(float)ux);
   OLoad(y,dy); OLoad(x,dy);
}
```

Заметьте, один из вызовов закомментирован, так как он приводит к конфликту на этапе трансляции. Для устранения конфликта при вызове *метода* пришлось задать *явное преобразование* аргумента, что показано в строке, следующей за строкой-комментарием.

```
Testing.OLoad Test
long value 1111
long value 1111
float value 22,05
double value 5,55
int par1 1111, float par2 1111
double par1 22,0499992370605, double par2 5,55
double par1 1111, double par2 5,55
```

Рис. 4.2. Вывод на печать результатов теста OLoadTest

Прежде чем посмотреть на результаты работы тестирующей процедуры, попробуйте понять, какой из перегруженных *методов* вызывается для каждого из вызовов. В случае каких-либо сомнений используйте схему, приведенную на 4.1.

Приведу все-таки некоторые комментарии. При первом вызове *метода* тип *источника* - `int`, а тип аргумента у четырех возможных реализаций соответственно `float`, `long`, `ulong`, `double`. Явного соответствия нет, поэтому нужно искать самый короткий путь на схеме. Так как не существует *неявного преобразования* из типа `int` в тип `ulong` (на диаграмме нет пути), то остаются возможными три реализации. Но путь из `int` в `long` короче, чем остальные пути, поэтому будет выбрана long-реализация *метода*.

Следующий вызов демонстрирует еще одну возможную ситуацию. Для типа *источника* `uint` существуют две возможные реализации, и пути преобразований для них имеют одинаковую длину. В этом случае выбирается та реализация, для которой на диаграмме путь показан сплошной, а не пунктирной стрелкой, потому будет выбрана реализация с параметром `long`.

Рассмотрим еще ситуацию, приводящую к конфликту. Первый аргумент в соответствии с правилами требует вызова одной реализации, а второй аргумент будет настаивать на вызове другой реализации. Возникнет коллизия, не разрешимая правилами C# и приводящая к ошибке периода компиляции. Коллизию требуется устранить, например, как это сделано в примере. Обратите внимание - обе реализации допустимы, и существуй даже только одна из них, ошибки бы не возникало.

Явные преобразования

Как уже говорилось, *явные преобразования* могут быть опасными из-за потери точности. Поэтому они выполняются по указанию программиста, - на нем лежит вся ответственность за результаты.

Преобразования строкового типа

Важным классом преобразований являются **преобразования в строковый тип** и наоборот. Преобразования в строковый тип всегда определены, поскольку, напомню, все типы являются *потомками базового класса Object*, а, следовательно, обладают *методом* `ToString()`. Для встроенных типов определена подходящая реализация этого *метода*. В частности, для всех подтипов арифметического типа *метод* `ToString()` возвращает в подходящей форме строку, задающую соответствующее значение арифметического типа. Заметьте, *метод* `ToString` можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда по контексту требуется преобразование к строковому типу. Вот соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования в строку данных различного типа.
/// </summary>
public void ToStringTest()
{
   s ="Владимир Петров ";
   s1 =" Возраст: "; ux = 27;
   s = s + s1 + ux.ToString();
   s1 =" Зарплата: "; dy = 2700.50;
   s = s + s1 + dy;
   WhoIsWho("s",s);
}
```

```
Testing.ToString Test
type s is System.String , value is Владимир Петров  Возраст: 27 Зарплата: 2700.5
```

Рис. 4.3. Вывод на печать результатов теста ToStringTest

Здесь для переменной `ux` *метод* был вызван явно, а для переменной `dy` он вызывается автоматически. Результат работы этой процедуры показан на рис. 4.3.

Преобразования из строкового типа в другие типы, например, в арифметический, должны выполняться явно. Но *явных преобразований* между арифметикой и строками не существуют. Необходимы другие механизмы, и они в C# имеются. Для этой *цели* можно использовать соответствующие *методы класса Convert библиотеки FCL*, встроенного в пространство имен `System`. Приведу соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования строки в данные различного типа.
/// </summary>
public void FromStringTest()
{
   s ="Введите возраст ";
   Console.WriteLine(s);
   s1 = Console.ReadLine();
   ux = Convert.ToUInt32(s1);
   WhoIsWho("Возраст: ",ux);
   s ="Введите зарплату ";
   Console.WriteLine(s);
   s1 = Console.ReadLine();
   dy = Convert.ToDouble(s1);
   WhoIsWho("Зарплата: ",dy);
}
```

Этот пример демонстрирует ввод с консоли данных разных типов. Данные, читаемые с консоли *методом* `ReadLine` или `Read`, всегда представляют собой строку, которую затем необходимо преобразовать в нужный тип. Тут-то и вызываются соответствующие *методы класса Convert*. Естественно, для успеха преобразования строка должна содержать значение в формате, допускающем подобное преобразование. Заметьте, например, что при записи значения числа для выделения дробной части должна использоваться запятая, а не точка; в противном случае возникнет ошибка периода выполнения.

На рис. 4.4 показаны результаты вывода и ввода данных с консоли при работе этой процедуры.

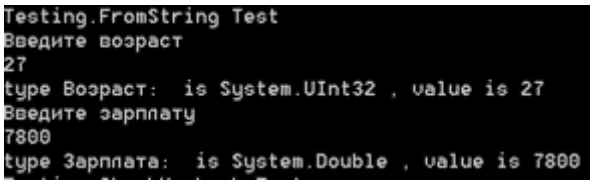

Рис. 4.4. Вывод на печать результатов теста FromStringTest

Преобразования и класс Convert

Класс Convert, определенный в пространстве имен `System`, играет важную роль, обеспечивая необходимые преобразования между различными типами. Напомню, что внутри арифметического типа можно использовать более простой, скобочный способ приведения к нужному типу. Но таким способом нельзя привести, например, переменную типа `string` к типу `int`, оператор присваивания: `ux = (int)s1;` приведет к ошибке периода компиляции. Здесь необходим вызов *метода* `ToInt32` *класса Convert*, как это сделано в последнем примере предыдущего раздела.

Методы класса Convert поддерживают общий способ выполнения преобразований между типами. *Класс Convert* содержит 15 статических *методов* вида `To <Type> (ToBoolean(),...ToUInt64())`, где `Type` может принимать значения от `Boolean` до

UInt64 для всех встроенных типов, перечисленных в таблице 3.1. Единственным *исключением* является тип object, - *метода* ToObject нет по понятным причинам, поскольку для всех типов существует *неявное преобразование* к типу object. Среди других *методов* отмечу общий статический *метод* ChangeType, позволяющий преобразование объекта к некоторому заданному типу.

Существует возможность преобразования к системному типу DateTime, который хотя и не является встроенным типом языка C#, но допустим в программах, как и любой другой системный тип. Приведу простейший пример работы с этим типом:

```
// System type: DateTime
System.DateTime dat = Convert.ToDateTime("15.03.2003");
Console.WriteLine("Date = {0}", dat);
```

Результатом вывода будет строка:

```
Date = 15.03.2003      0:00:00
```

Все *методы* To <Type> *класса Convert* перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти *методы* задают все возможные преобразования между всеми встроенными типами языка C#.

Кроме *методов*, задающих преобразования типов, в *классе Convert* имеются и другие *методы*, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования значений объектов и другие *методы*. Подробности можно посмотреть в справочной системе.

Проверяемые преобразования

Уже упоминалось о том, что при выполнении *явных преобразований* могут возникать нежелательные явления, например, потеря точности. Я говорил, что вся ответственность за это ложится на программиста, и легче ему от этого не становится. А какую часть этого бремени может взять на себя язык программирования? Что можно предусмотреть для обнаружения ситуаций, когда такие явления все-таки возникают? В языке C# имеются необходимые для этого средства.

Язык C# позволяет создать *проверяемый блок*, в котором будет осуществляться проверка результата вычисления арифметических выражений. Если результат вычисления значения *источника* выходит за диапазон возможных значений целевой переменной, то возникнет *исключение* (говорят также: "будет выброшено *исключение*") соответствующего типа. Если предусмотрена обработка *исключения*, то дальнейшее зависит от обработчика *исключения*. В лучшем случае, программа сможет продолжить корректное выполнение. В худшем, - она остановится и выдаст информацию об ошибке. Заметьте, не произойдет самого опасного - продолжения работы программы с неверными данными.

Синтаксически *проверяемый блок* предваряется ключевым словом checked. В теле такого блока арифметические преобразования проверяются на допустимость. Естественно, подобная проверка требует дополнительных временных затрат. Если группа операторов в теле такого блока нам кажется безопасной, то их можно выделить в *непроверяемый блок*, используя ключевое слово unchecked. Замечу еще, что и в *непроверяемом блоке* при работе *методов Convert* все опасные преобразования проверяются и приводят к выбрасыванию *исключений*. Приведу пример, демонстрирующий все описанные ситуации:

```
/// <summary>
```

```
/// Демонстрация проверяемых и непроверяемых преобразований.
/// Опасные проверяемые преобразования приводят к исключениям.
/// Опасные непроверяемые преобразования приводят к неверным
/// результатам, что совсем плохо.
/// </summary>
public void CheckUncheckTest()
{
   x = -25^2;
   WhoIsWho ("x", x);
   b= 255;
   WhoIsWho("b",b);
   // Проверяемые опасные преобразования.
   // Возникают исключения, перехватываемые catch-блоком.
   checked
   {
      try
      {
         b += 1;
      }
      catch (Exception e)
      {
         Console.WriteLine("Переполнение при вычислении b");
         Console.WriteLine(e);
      }
      try
      {
         b = (byte)x;
      }
      catch (Exception e)
      {
         Console.WriteLine("Переполнение при преобразовании к byte");
         Console.WriteLine(e);
      }
      // непроверяемые опасные преобразования
      unchecked
      {
         try
         {
            b +=1;
                              WhoIsWho ("b", b);
            b = (byte)x;
            WhoIsWho ("b", b);
            ux= (uint)x;
            WhoIsWho ("ux", ux);
            Console.WriteLine("Исключений нет, но результаты не верны!");
         }
         catch (Exception e)
         {
            Console.WriteLine("Этот текст не должен появляться");
            Console.WriteLine(e);
         }
         // автоматическая проверка преобразований в Convert
         // исключения возникают, несмотря на unchecked
         try
         {
            b = Convert.ToByte(x);
         }
         catch (Exception e)
         {
            Console.WriteLine("Переполнение при
               преобразовании к byte!");
            Console.WriteLine(e);
         }
         try
```

```
            {
               ux= Convert.ToUInt32(x);
            }
            catch (Exception e)
            {
               Console.WriteLine("Потеря знака при
                  преобразовании к uint!");
               Console.WriteLine(e);
            }
         }
      }
   }
```

Исключения и охраняемые блоки. Первое знакомство

В этом примере мы впервые встречаемся с *охраняемыми* try-блоками. *Исключениям* и способам их обработки посвящена отдельная лекция, но не стоит откладывать надолго знакомство со столь важным механизмом. Как показывает практика программирования, любая вызываемая программа не гарантирует, что в процессе ее работы не возникнут какие-либо неполадки, в результате которых она не сможет выполнить свою часть контракта. *Исключения* являются нормальным способом уведомления об ошибках в работе программы. Возникновение ошибки в работе программы должно приводить к выбрасыванию *исключения* соответствующего типа, следствием чего является прерывание нормального хода выполнения программы и передача управления обработчику *исключения* - стандартному или предусмотренному самой программой.

Заметьте, рекомендуемый стиль программирования в C# отличается от стиля, принятого в языках C/C++, где функция, в которой возникла ошибка, завершается нормальным образом, уведомляя об ошибке в возвращаемом значении результата. Вызывающая программа должна анализировать результат, чтобы понять, была ли ошибка в работе вызванной функции и какова ее природа. При программировании в стиле C# ответственность за обнаружение ошибок лежит на вызванной программе. Она должна не только обнаружить ошибку, но и явно сообщить о ней, выбрасывая *исключение* соответствующего типа. Вызываемая программа должна попытаться исправить последствия ошибки в обработчике *исключения*. Подробности смотри в лекции про *исключения*.

В состав *библиотеки FCL* входит *класс Exception*, *свойства* и *методы* которого позволяют работать с *исключениями* как с объектами, получать нужную информацию, дополнять объект собственной информацией. У *класса Exception* - большое число *потомков*, каждый из которых описывает определенный тип *исключения*. При проектировании собственных классов можно параллельно проектировать и классы, задающие собственный тип *исключений*, который может выбрасываться в случае ошибок при работе *методов* класса. Создаваемый класс *исключений* должен быть *потомком класса Exception*.

Если в некотором модуле предполагается возможность появления *исключений*, то разумно предусмотреть и их обработку. В этом случае в модуле создается *охраняемый* **try-блок**, предваряемый ключевым словом `try`. Вслед за этим блоком следуют один или несколько блоков, обрабатывающих *исключения*, - catch-блоков. Каждый catch-блок имеет формальный параметр *класса Exception* или одного из его *потомков*. Если в try-блоке возникает *исключение* типа `T`, то catch-блоки начинают конкурировать в борьбе за перехват *исключения*. Первый по порядку catch-блок, тип формального аргумента которого согласован с типом `T` - совпадает с ним или является его *потомком* - захватывает *исключение* и начинает выполняться; поэтому порядок написания catch-блоков небезразличен. Вначале должны идти специализированные обработчики. Универсальным обработчиком является catch-блок с формальным параметром родового *класса Exception*, согласованным с *исключением* любого типа `T`. Универсальный

обработчик, если он есть, стоит последним, поскольку захватывает *исключение* любого типа.

Конечно, плохо, когда в процессе работы той или иной процедуры возникает *исключение*. Однако его появление еще не означает, что процедура не сможет выполнить свой контракт. *Исключение* может быть нужным образом обработано, после чего продолжится нормальный ход вычислений приложения. Гораздо хуже, когда возникают ошибки в работе процедуры, не приводящие к *исключениям*. Тогда работа продолжается с неверными данными без исправления ситуации и даже без уведомления о возникновении ошибки. Наш пример показывает, что вычисления в C# могут быть небезопасными и следует применять специальные средства языка, такие как, например, checked-блоки, чтобы избежать появления подобных ситуаций.

Вернемся к обсуждению нашего примера. Здесь как в *проверяемых*, так и в *непроверяемых блоках* находятся *охраняемые блоки* с соответствующими обработчиками исключительных ситуаций. Во всех случаях применяется универсальный обработчик, захватывающий любое *исключение* в случае его возникновения в try-блоке. Сами обработчики являются простыми уведомителями, они лишь сообщают об ошибочной ситуации, не пытаясь исправить ее.

Опасные вычисления в охраняемых проверяемых блоках

Такая ситуация возникает в первых двух try-блоках нашего примера. Эти блоки встроены в *проверяемый* checked-блок. В каждом из них используются опасные вычисления, приводящие к неверным результатам. Так, при присваивании невинного выражения `b+1` из-за переполнения переменная `b` получает значение `0`, а не `256`. Поскольку вычисление находится в *проверяемом блоке*, то ошибка обнаруживается и результатом является вызов *исключения*. Далее, поскольку все это происходит в *охраняемом блоке*, то управление перехватывается и обрабатывается в соответствующем catch-блоке. Эту ситуацию следует отнести к нормальному, разумно построенному процессу вычислений.

Опасные вычисления в охраняемых непроверяемых блоках

Такую ситуацию демонстрирует третий try-блок нашего примера, встроенный в *непроверяемый* unchecked-блок. Здесь участвуют те же самые опасные вычисления, но теперь их корректность не проверяется, они не вызывают *исключений*, и как следствие, соответствующий catch-блок не вызывается. Результаты вычислений при этом неверны, но никаких уведомлений об этом нет. Это самая плохая ситуация, которая может случиться при работе наших программ.

Заметьте, проверку переполнения в арифметических вычислениях можно включить не только с помощью создания checked-блоков, но и задав *свойство* checked проекта (по умолчанию, оно выключено). Как правило, это *свойство* проекта всегда включается в процессе разработки и отладки. В законченной версии проекта *свойство* вновь отключается, поскольку полная проверка всех преобразований требует определенных накладных расходов, увеличивая время работы; а *проверяемые блоки* остаются лишь там, где такой контроль действительно необходим.

Область действия проверки или ее отключения можно распространить и на отдельное выражение. В этом случае спецификаторы `checked` и `unchecked` предшествуют выражению, заключенному в круглые скобки. Такое выражение называется *проверяемым (непроверяемым)* **выражением**, а `checked` и `unchecked` рассматриваются как операции, допустимые в выражениях.

Опасные преобразования и методы класса Convert

Явно выполняемые преобразования по определению относятся к опасным. *Явные преобразования* можно выполнять по-разному. Синтаксически наиболее просто выполнить приведение типа - кастинг, явно указав тип приведения, как это сделано в только что рассмотренном примере. Но если это делается в *непроверяемом блоке*, последствия могут быть самыми печальными. Поэтому такой способ приведения типов следует применять с большой осторожностью. Надежнее выполнять преобразования типов более универсальным способом, используя стандартный встроенный *класс Convert*, специально спроектированный для этих *целей*.

В нашем примере четвертый и пятый try-блоки встроены в *непроверяемый* unchecked-блок. Но опасные преобразования реализуются *методами класса Convert*, которые сами проводят проверку и при необходимости выбрасывают *исключения*, что и происходит в нашем случае.

На рис. 4.5 показаны результаты работы процедуры `CheckUncheckTest`. Их анализ способствует лучшему пониманию рассмотренных нами ситуаций.

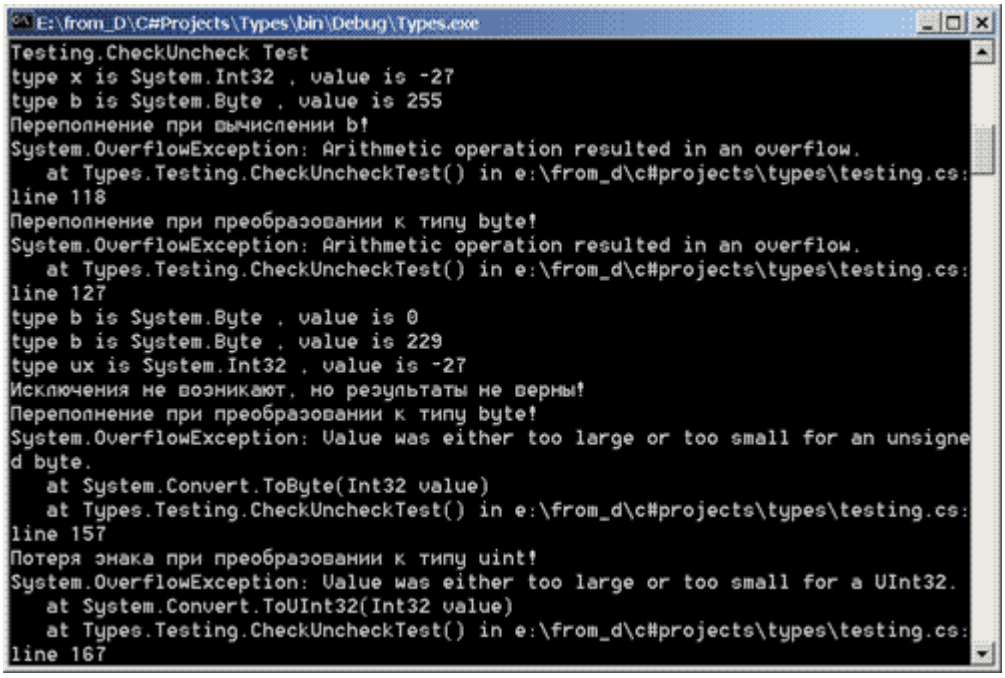

Рис. 4.5. Вывод на печать результатов теста CheckUncheckTest

На этом, пожалуй, пора поставить точку в обсуждении системы типов языка C#. За получением тех или иных подробностей, как всегда, следует обращаться к справочной системе.

5. Лекция: Переменные и выражения

Объявление переменных

В лекции 4 рассматривались типы языка C#. Естественным продолжением этой темы является рассмотрение *переменных* языка. *Переменные* и типы - тесно связанные понятия. С объектной точки зрения *переменная* - это экземпляр типа. Скалярную *переменную* можно рассматривать как сущность, обладающую именем, значением и типом. Имя и тип задаются при объявлении *переменной* и остаются неизменными на все *время ее жизни*. Значение *переменной* может меняться в ходе вычислений, эта возможность вариации значений и дало имя понятию *переменная* **(Variable)** в математике и программировании. Получение начального значения *переменной* называется ее *инициализацией*. Важной новинкой языка C# является требование обязательной *инициализации переменной* до начала ее использования. Попытка использовать неинициализированную *переменную* приводит к ошибкам, обнаруживаемым еще на этапе компиляции. *Инициализация переменных*, как правило, выполняется в момент объявления, хотя и может быть отложена.

Тесная связь типов и классов в языке C# обсуждалась в предыдущей лекции. Не менее тесная связь существует между *переменными* и объектами. Так что, когда речь идет о *переменной* значимого типа, то во многих ситуациях она может играть роль объекта некоторого класса. В этой лекции обсуждение будет связано со скалярными *переменными* встроенных типов. Все *переменные*, прежде чем появиться в вычислениях, должны быть объявлены. Давайте рассмотрим, как это делается в C#.

Проект Variables

Как обычно, для рассмотрения примеров построен специальный проект. В данной лекции это консольный проект с именем `Variables`. Построенный по умолчанию класс `Class1` содержит точку входа `Main`. Добавленный в проект класс `Testing` содержит набор скалярных *переменных* и методов, тестирующих разные аспекты работы со скалярными *переменными* в C#. В процедуре `Main` создается объект класса `Testing` и поочередно вызываются его методы, каждый из которых призван проиллюстрировать те или иные моменты работы.

Синтаксис объявления

Общий *синтаксис объявления* сущностей в C# похож на *синтаксис объявления* в C++, хотя и имеет ряд отличий. Вот какова общая структура объявления:

```
[<атрибуты>] [<модификаторы>] <тип> <объявители>;
```

Об атрибутах - этой новинке языка C# - уже шла речь, о них будем говорить и в последующих лекциях курса. Модификаторы будут появляться по мере необходимости. При объявлении *переменных* чаще всего задаются модификаторы доступа - `public`, `private` и другие. Если атрибуты и модификаторы могут и не указываться в объявлении, то задание типа необходимо всегда. Ограничимся пока рассмотрением уже изученных встроенных типов. Когда в роли типа выступают имена типов из таблицы 3.1, это означает, что объявляются простые скалярные *переменные*. Структурные типы - массивы, перечисления, структуры и другие пользовательские типы - будут изучаться в последующих лекциях.

При объявлении простых *переменных* указывается их тип и список *объявителей*, где *объявитель* - это имя или имя с *инициализацией*. Список *объявителей* позволяет в одном объявлении задать несколько *переменных* одного типа. Если *объявитель* задается именем *переменной*, то имеет место *объявление с отложенной инициализацией*. Хороший стиль программирования предполагает задание

инициализации переменной в момент ее объявления. *Инициализацию* можно осуществлять двояко - обычным присваиванием или в объектной манере. Во втором случае для *переменной* используется конструкция `new` и вызывается конструктор по умолчанию. Процедура `SimpleVars` класса `Testing` иллюстрирует различные способы объявления *переменных* и простейшие вычисления над ними:

```
public void SimpleVars()
{
   //Объявления локальных переменных
   int x, s; //без инициализации
   int y =0, u = 77; //обычный способ инициализации
   //допустимая инициализация
   float w1=0f, w2 = 5.5f, w3 =w1+ w2 + 125.25f;
   //допустимая инициализация в объектном стиле
   int z= new int();
   //Недопустимая инициализация.
   //Конструктор с параметрами не определен
   //int v = new int(77);
      x=u+y; //теперь x инициализирована
   if(x> 5) s = 4;
   for (x=1; x<5; x++)s=5;
   //Инициализация в if и for не рассматривается,
   //поэтому s считается неинициализированной переменной
   //Ошибка компиляции:использование неинициализированной переменной
   //Console.WriteLine("s= {0}",s);
}  //SimpleVars
```

В первой строке объявляются *переменные* `x` и `s` с отложенной *инициализацией*. Заметьте (и это важно!), что всякая попытка использовать еще не инициализированную *переменную* в правых частях операторов присваивания, в вызовах функций, вообще в вычислениях приводит к ошибке уже на этапе компиляции.

Последующие объявления *переменных* эквивалентны по сути, но демонстрируют два стиля *инициализации* - обычный и объектный. Обычная форма *инициализации* предпочтительнее не только в силу своей естественности, но она и более эффективна, поскольку в этом случае инициализирующее выражение может быть достаточно сложным, с *переменными* и функциями. На практике объектный стиль для скалярных *переменных* используется редко. Вместе с тем полезно понимать, что объявление с *инициализацией* `int y =0` можно рассматривать как создание нового объекта (`new`) и вызова для него конструктора по умолчанию. При *инициализации* в объектной форме может быть вызван только конструктор по умолчанию, другие конструкторы с параметрами для встроенных типов не определены. В примере закомментировано объявление *переменной* `v` с *инициализацией* в объектном стиле, приводящее к ошибке, где делается попытка дать *переменной* значение, передавая его конструктору в качестве параметра.

Откладывать *инициализацию* не стоит, как показывает пример с *переменной* `s`, *объявленной с отложенной инициализацией*. В вычислениях она дважды получает значение: один раз в операторе `if`, другой - в операторе цикла `for`. Тем не менее, при компиляции возникнет ошибка, утверждающая, что в процедуре `WriteLine` делается попытка использовать неинициализированную *переменную* `s`. Связано это с тем, что для операторов `if` и `for` на этапе компиляции не вычисляются условия, зависящие от *переменных*. Поэтому компилятор предполагает худшее - условия ложны, *инициализация* `s` в этих операторах не происходит. А за *инициализацией* наш компилятор следит строго, ты так и знай!

Время жизни и область видимости переменных

Давайте рассмотрим такие важные характеристики *переменных*, как *время их жизни* и *область видимости*. Зададимся вопросом, как долго живут объявленные *переменные* и в какой области программы видимы их имена? Ответ зависит от того, где и как, в каком контексте объявлены *переменные*. В языке C# не так уж много возможностей для объявления *переменных*, пожалуй, меньше, чем в любом другом языке. Открою "страшную" тайну, - здесь вообще нет настоящих *глобальных переменных*. Их отсутствие не следует считать некоторым недостатком C#, это достоинство языка. Но обо всем по порядку.

Поля

Первая важнейшая роль *переменных*, - они задают свойства структур, интерфейсов, классов. В языке C# такие *переменные* называются ***полями (fields)***. Структуры, интерфейсы, классы, *поля* - рассмотрению этих понятий будет посвящена большая часть этой книги, а сейчас сообщу лишь некоторые минимальные сведения, связанные с рассматриваемой темой. *Поля* объявляются при описании класса (и его частных случаев - интерфейса, структуры). Когда конструктор класса создает очередной объект - экземпляр класса, то он в динамической памяти создает набор *полей*, определяемых классом, и записывает в них значения, характеризующие свойства данного конкретного экземпляра. Так что каждый объект в памяти можно рассматривать как набор соответствующих *полей* класса со своими значениями. Время существования и *область видимости полей* определяется объектом, которому они принадлежат. Объекты в динамической памяти, с которыми не связана ни одна ссылочная *переменная*, становятся недоступными. Реально они оканчивают свое существование, когда сборщик мусора (garbage collector) выполнит чистку "кучи". Для значимых типов, к которым принадлежат экземпляры структур, жизнь оканчивается при завершении блока, в котором они объявлены.

Есть одно важное исключение. Некоторые *поля* могут жить дольше. Если при объявлении класса *поле* объявлено с модификатором `static`, то такое *поле* является частью класса и единственным на все его экземпляры. Поэтому `static`-*поля* живут так же долго, как и сам класс. Более подробно эти вопросы будут обсуждаться при рассмотрении классов, структур, интерфейсов.

Глобальные переменные уровня модуля. Существуют ли они в C#?

Где еще могут объявляться *переменные*? Во многих языках программирования *переменные* могут объявляться на уровне модуля. Такие *переменные* называются ***глобальными***. Их область действия распространяется, по крайней мере, на весь модуль. *Глобальные переменные* играют важную роль, поскольку они обеспечивают весьма эффективный способ обмена информацией между различными частями модуля. Обратная сторона эффективности аппарата *глобальных переменных*, - их опасность. Если какая-либо процедура, в которой доступна *глобальная переменная*, некорректно изменит ее значение, то ошибка может проявиться в другой процедуре, использующей эту *переменную*. Найти причину ошибки бывает чрезвычайно трудно. В таких ситуациях приходится проверять работу многих компонентов модуля.

В языке C# роль модуля играют классы, пространства имен, проекты, решения. *Поля* классов, о которых шла речь выше, могут рассматриваться как *глобальные переменные* класса. Но здесь у них особая роль. Данные (*поля*) являются тем центром, вокруг которого вращается мир класса. Заметьте, каждый экземпляр класса - это отдельный мир. *Поля* экземпляра (открытые и закрытые) - это глобальная информация, которая доступна всем методам класса, играющим второстепенную роль - они обрабатывают данные.

Статические *поля* класса хранят информацию, общую для всех экземпляров класса. Они представляют определенную опасность, поскольку каждый экземпляр способен менять их значения.

В других видах модуля - пространствах имен, проектах, решениях - нельзя объявлять *переменные*. В пространствах имен в языке C# разрешено только объявление классов и их частных случаев: структур, интерфейсов, делегатов, перечислений. Поэтому *глобальных переменных* уровня модуля, в привычном для других языков программирования смысле, в языке C# нет. Классы не могут обмениваться информацией, используя *глобальные переменные*. Все взаимодействие между ними обеспечивается способами, стандартными для объектного подхода. Между классами могут существовать два типа отношений - клиентские и наследования, а основной способ инициации вычислений - это вызов метода для объекта-цели или вызов обработчика события. *Поля* класса и аргументы метода позволяют передавать и получать нужную информацию. Устранение *глобальных переменных* как источника опасных, трудно находимых ошибок существенно повышает надежность создаваемых на языке C# программных продуктов.

Введем в класс `Testing` нашего примера три закрытых *поля* и добавим конструктор с параметрами, инициализирующий значения *полей* при создании экземпляра класса:

```
//fields
      int x,y; //координаты точки
      string name; //имя точки
      //конструктор с параметрами
      public Testing(int x, int y, string name)
      {
         this.x = x; this.y = y; this.name = name;
      }
```

В процедуре `Main` первым делом создается экземпляр класса `Testing`, а затем вызываются методы класса, тестирующие различные ситуации:

```
Testing ts = new Testing(5,10,"Точка1");
         ts.SimpleVars();
```

Локальные переменные

Перейдем теперь к рассмотрению *локальных переменных*. Этому важнейшему виду *переменных* будет посвящена вся оставшаяся часть данной лекции. Во всех языках программирования, в том числе и в C#, основной контекст, в котором появляются *переменные*, - это процедуры. *Переменные*, объявленные на уровне процедуры, называются *локальными*, - они локализованы в процедуре.

В некоторых языках, например в Паскале, *локальные переменные* должны быть объявлены в вершине процедурного блока. Иногда это правило заменяется менее жестким, но, по сути, аналогичным правилом, - где бы внутри процедурного блока ни была объявлена *переменная*, она считается объявленной в вершине блока, и ее *область видимости* распространяется на весь процедурный блок. В C#, также как и в языке C++, принята другая стратегия. *Переменную* можно объявлять в любой точке процедурного блока. *Область ее видимости* распространяется от точки объявления до конца процедурного блока.

На самом деле, ситуация с процедурным блоком в C# не так проста. Процедурный блок имеет сложную структуру; в него могут быть вложены другие блоки, связанные с операторами выбора, цикла и так далее. В каждом таком блоке, в свою очередь, допустимы вложения блоков. В каждом внутреннем блоке допустимы объявления *переменных*. *Переменные*, объявленные во внутренних блоках, локализованы именно в

этих блоках, их *область видимости* и *время жизни* определяются этими блоками. *Локальные переменные* начинают существовать при достижении вычислений в блоке точки объявления и перестают существовать, когда процесс вычисления завершает выполнение операторов блока. Можно полагать, что для каждого такого блока выполняется так называемый пролог и эпилог. В прологе *локальным переменным* отводится память, в эпилоге память освобождается. Фактически ситуация сложнее, поскольку выделение памяти, а следовательно, и начало жизни *переменной*, объявленной в блоке, происходит не в момент входа в блок, а лишь тогда, когда достигается точка объявления *локальной переменной*.

Давайте обратимся к примеру. В класс `Testing` добавлен метод с именем `ScopeVar`, вызываемый в процедуре `Main`. Вот код этого метода:

```
/// <summary>
/// Анализ области видимости переменных
/// </summary>
/// <param name="x"></param>
public void ScopeVar(int x)
{
   //int x=0;
   int y =77; string s = name;
   if (s=="Точка1")
   {
         int u = 5; int v = u+y; x +=1;
         Console.WriteLine("y= {0}; u={1};
            v={2}; x={3}", y,u,v,x);
   }
   else
   {
         int u= 7; int v= u+y;
         Console.WriteLine("y= {0}; u={1}; v={2}", y,u,v);
   }
   //Console.WriteLine("y= {0}; u={1}; v={2}",y,u,v);
   //Локальные переменные не могут быть статическими.
   //static int Count = 1;
   //Ошибка: использование sum до объявления
   //Console.WriteLine("x= {0}; sum ={1}", x,sum);
   int i;long sum =0;
   for(i=1; i<x; i++)
   {
      //ошибка: коллизия имен: y
      //float y = 7.7f;
      sum +=i;
   }
   Console.WriteLine("x= {0}; sum ={1}", x,sum);
}//ScopeVar
```

Заметьте, в теле метода встречаются имена *полей*, аргументов и *локальных переменных*. Эти имена могут совпадать. Например, имя `x` имеет *поле* класса и формальный аргумент метода. Это допустимая ситуация. В языке C# разрешено иметь *локальные переменные* с именами, совпадающими с именами *полей* класса, - в нашем примере таким является имя `y`; однако, запрещено иметь *локальные переменные*, имена которых совпадают с именами формальных аргументов. Этот запрет распространяется не только на внешний уровень процедурного блока, что вполне естественно, но и на все внутренние блоки.

В процедурный блок вложены два блока, порожденные оператором `if`. В каждом из них объявлены *переменные* с одинаковыми именами `u` и `v`. Это корректные объявления, поскольку время существования и *области видимости* этих *переменных* не пересекаются. Итак, для невложенных блоков разрешено объявление *локальных переменных* с одинаковыми именами. Заметьте также, что *переменные* `u` и `v` перестают

существовать после выхода из блока, так что операторы печати, расположенные внутри блока, работают корректно, а оператор печати вне блока приводит к ошибке, - `u` и `v` здесь не видимы, кончилось *время их жизни*. По этой причине оператор закомментирован.

Выражение, проверяемое в операторе `if`, зависит от значения *поля* `name`. Значение *поля* глобально для метода и доступно всюду, если только не перекрывается именем аргумента (как в случае с *полем* `x`) или *локальной переменной* (как в случае с *полем* `y`).

Во многих языках программирования разрешено иметь *локальные* статические *переменные*, у которых *область видимости* определяется блоком, но *время их жизни* совпадает со *временем жизни* проекта. При каждом повторном входе в блок такие *переменные* восстанавливают значение, полученное при предыдущем выходе из блока. В языке C# статическими могут быть только *поля*, но не *локальные переменные*. Незаконная попытка объявления `static` *переменной* в процедуре `ScopeVar` закомментирована. Попытка использовать имя *переменной* в точке, предшествующей ее объявлению, также незаконна и закомментирована.

Глобальные переменные уровня процедуры. Существуют ли?

Поскольку процедурный блок имеет сложную структуру с вложенными внутренними блоками, то и здесь возникает тема *глобальных переменных*. *Переменная*, объявленная во внешнем блоке, рассматривается как глобальная по отношению к внутренним блокам. Во всех известных мне языках программирования во внутренних блоках разрешается объявлять *переменные* с именем, совпадающим с именем *глобальной переменной*. *Конфликт имен* снимается за счет того, что локальное внутреннее определение сильнее внешнего. Поэтому *область видимости* внешней *глобальной переменной* сужается и не распространяется на те внутренние блоки, где объявлена *переменная* с подобным именем. Внутри блока действует локальное объявление этого блока, при выходе восстанавливается область действия внешнего имени. В языке C# этот гордиев узел *конфликтующих имен* разрублен, - во внутренних блоках запрещено использование имен, совпадающих с именем, использованным во внешнем блоке. В нашем примере незаконная попытка объявить во внутреннем блоке уже объявленное имя y закомментирована.

Обратите внимание, что подобные решения, принятые создателями языка C#, не только упрощают жизнь разработчикам транслятора. Они способствуют повышению эффективности программ, а самое главное, повышают надежность программирования на C#.

Отвечая на вопрос, вынесенный в заголовок, следует сказать, что *глобальные переменные* на уровне процедуры в языке C#, конечно же, есть, но нет *конфликта имен* между *глобальными* и *локальными переменными* на этом уровне. *Область видимости глобальных переменных* процедурного блока распространяется на весь блок, в котором они объявлены, начиная от точки объявления, и не зависит от существования внутренних блоков. Когда говорят, что в C# нет *глобальных переменных*, то, прежде всего, имеют в виду их отсутствие на уровне модуля. Уже во вторую очередь речь идет об отсутствии *конфликтов имен* на процедурном уровне.

Константы

Константы C# могут появляться, как обычно, в виде литералов и именованных *констант*. Вот пример *константы*, заданной литералом и стоящей в правой части оператора присваивания:

```
y = 7.7f;
```

Значение *константы* "7.7f" является одновременно ее именем, оно же позволяет однозначно определить тип *константы*. Заметьте, иногда, как в данном случае, приходится добавлять к значению специальные символы для точного указания типа. Я не буду останавливаться на этих подробностях. Если возникает необходимость уточнить, как записываются литералы, то достаточно получить справку по этой теме. Делается все так же, как и в языке C++.

Всюду, где можно объявить *переменную*, можно объявить и именованную *константу*. *Синтаксис объявления* схож. В объявление добавляется модификатор `const`, *инициализация констант* обязательна и не может быть отложена. Инициализирующее выражение может быть сложным, важно, чтобы оно было вычислимым в момент его определения. Вот пример объявления *констант*:

```
/// <summary>
/// Константы
/// </summary>
public void Constants()
{
   const int SmallSize = 38, LargeSize =58;
   const int MidSize = (SmallSize + LargeSize)/2;
   const double pi = 3.141593;
   //LargeSize = 60; //Значение константы нельзя изменить.
   Console.WriteLine("MidSize= {0}; pi={1}",
      MidSize, pi);
}//Constants
```

6. Лекция: Выражения. Операции в выражениях

Выражения

Выражения строятся из операндов - констант, переменных, функций, - объединенных знаками операций и скобками. При вычислении *выражения* определяется его значение и тип. Эти характеристики однозначно задаются значениями и типами операндов, входящих в *выражение*, и правилами вычисления *выражения*. Правила также задают:

- *приоритет операций*;
- для операций одного *приоритета* **порядок применения** - слева направо или справа налево;
- преобразование типов операндов и выбор реализации для перегруженных операций;
- тип и значение результата выполнения операции над заданными значениями операндов определенного типа.

Программист, записывающий *выражение*, должен знать, по каким правилам оно будет вычисляться. Сложность в том, что эти правила, начиная с *приоритета операций*, варьируются от языка к языку. Давайте посмотрим, как это делается в C#.

Приоритет и порядок выполнения операций

Большинство операций в языке C#, их *приоритет* и порядок наследованы из языка C++. Однако имеются и различия: например, нет операции " , ", позволяющей вычислять список *выражений*; добавлены уже упоминавшиеся операции `checked` и `unchecked`, применимые к *выражениям*.

Как это обычно делается, приведем *таблицу приоритетов* операций, в каждой строке которой собраны операции одного *приоритета*, а строки следуют в порядке *приоритетов*, от высшего к низшему.

Приоритет	**Категория**	**Операции**	**Порядок**
0	Первичные	`(expr) x.y f(x) a[x] x++ x-- new sizeof(t) typeof(t) checked(expr) unchecked(expr)`	Слева направо
1	Унарные	`+ - ! ~ ++x --x (T)x`	Слева направо
2	Мультипликативные (Умножение)	`- * / %`	Слева направо
3	Аддитивные (Сложение)	`+ -`	Слева направо
4	*Сдвиг*	`<< >>`	Слева направо
5	*Отношения*, проверка типов	`< > <= >= is as`	Слева направо
6	Эквивалентность	`== !=`	Слева направо
7	Логическое	`&`	Слева направо
8	Логическое исключающее ИЛИ (XOR)	`^`	Слева направо

9	Логическое ИЛИ (OR)	`|`	Слева направо
10	Условное И	`&&`	Слева направо
11	Условное ИЛИ	`||`	Слева направо
12	Условное выражение	`? :`	Справа налево
13	Присваивание	`= *= /= %= += -= <<= >>= &= ^= |=`	Справа налево

Перегрузка операций

Под ***перегрузкой операции*** понимается существование нескольких реализаций одной и той же операции. Большинство операций языка C# перегружены - одна и та же операция может применяться к операндам различных типов. Поэтому перед выполнением операции идет поиск реализации, подходящей для данных типов операндов. Замечу, что операции, как правило, выполняются над операндами одного типа. Если же операнды разных типов, то предварительно происходит неявное преобразование типа операнда. Оба операнда могут быть одного типа, но преобразование типов может все равно происходить - по той причине, что для заданных типов нет соответствующей перегруженной операции. Такая ситуация достаточно часто возникает на практике, поскольку, например, операция сложения не определена для младших подтипов арифметического типа. Приведу начальный фрагмент процедуры `Express`, предназначенной для анализа *выражений*:

```
/// <summary>
/// Анализ выражений
/// </summary>
public void Express()
{
   //перегрузка операций
   byte b1 =1, b2 =2, b3;
   short sh1;
   int in1;
   //b3 = b1 + b2;      //ошибка: результат типа int
   b3 = (byte)(b1+b2);
   //sh1 = b1 + b2;     //ошибка: результат типа int
   sh1 = (short)(b1+b2);
   in1 = b1+ b2 + sh1;
   Console.WriteLine("b3= " + b3 + " sh1= "+ sh1 +" in1= " + in1);
}//Express
```

Разберем этот фрагмент. Начнем с первого закомментированного оператора присваивания `b3 = b1+b2;`. *Выражение* здесь простейшее, включает одну бинарную операцию сложения. Оба операнда имеют тип `byte`, казалось бы, и результат должен быть типа `byte` и без помех присвоен переменной `b3`. Однако это не так. Для данных типа `byte` нет перегруженной реализации сложения. Ближайшей операцией является сложение целых типа `int`. Поэтому оба операнда преобразуются к типу `int`, выполняется операция сложения, результат имеет тип `int` и не может быть неявно преобразован в тип `byte`, - возникает ошибка еще на этапе компиляции. Корректная запись показана в следующем операторе. Аналогичная ситуация возникает, когда в левой части оператора стоит переменная типа `short`, - и здесь необходимо явное *приведение к типу*. Этого *приведения* не требуется, когда в левой части стоит переменная типа `int`.

Давайте разберем, как в данном примере организован вывод в методе `WriteLine`. До сих пор при вызове метода задавалось несколько параметров и использовалась форма вывода данных с подстановкой значений параметров в строку, заданную первым параметром. Здесь же есть только один параметр - это строка, заданная сложным *выражением*. Операция, многократно применяемая в этом *выражении*, это сложение " `+` ". Операнды сложения имеют разный тип: левый операнд имеет тип `string`, правый - арифметический (`byte`, `short`, `int`). В этом случае арифметический тип преобразуется к типу `string` и выполняется сложение строк (конкатенация). Напомню, при преобразовании арифметического типа к типу `string` вызывается метод `ToString()`, определенный для всех встроенных типов. Результатом этого *выражения* является строка, она и будет результатом вывода метода `WriteLine`.

Полагаю, что разбор данного примера и материалы предыдущей лекции, где приводилась иерархия преобразований внутри арифметического типа и обсуждались вопросы выбора реализации перегруженного метода, дают необходимое представление о том, как работает *перегрузка операций* при вычислении *выражений*. В деталях, как всегда, может помочь справочная система.

С чего начинается выполнение выражения

Вычисление *выражения* начинается с выполнения операций высшего *приоритета*. Первым делом вычисляются *выражения* в круглых скобках - `(expr)`, определяются значения полей объекта - `x.y`, вычисляются функции - `f(x)`, переменные с индексами - `a[i]`. Выполнение этих операций достаточно понятно и не нуждается в комментировании. Операции `checked` и `unchecked` включают и выключают проверку преобразований арифметического типа в *выражениях*, которым они предшествуют. О других операциях этой категории скажу чуть подробнее.

Операции "увеличить" и "уменьшить" (increment, decrement)

Операции *"увеличить на единицу"* и "*уменьшить на единицу*" могут быть префиксными и постфиксными. К высшему *приоритету* относятся постфиксные операции `x++` и `x--`. Префиксные операции имеют на единицу меньший *приоритет*. Главной особенностью как префиксных, так и постфиксных операций является побочный эффект, в результате которого значение `x` увеличивается (`++`) или уменьшается (`--`) на единицу. Для префиксных (`++x`, `--x`) операций результатом их выполнения является измененное значение `x`, постфиксные операции возвращают в качестве результата значение `x` до изменения. Приведу пример применения этих операций, дополнив метод `Express` новым фрагментом:

```
//операции increment и decrement
//Следующее выражение допустимо, но писать подобное никогда не следует
in1 = ++in1 +in1+ in1++;
//in2 = ++in1 + in1++ + in1;
Console.WriteLine(" in1= " + in1);
```

Обратите внимание, что хотя у постфиксной операции высший *приоритет*, это вовсе не означает, что при вычислении *выражения* вначале выполнится операция `in1++`, затем `++in1`, и только потом будет проводиться сложение. Нет, вычисления проводятся в том порядке, в котором они написаны. Поскольку на входе значение `in1` было равно `6`, то *выражение* будет вычисляться следующим образом:

```
7(7) + 7 + 7(8),
```

где в скобках записан побочный эффект операции. Так что консольный вывод даст следующий результат:

```
in1 = 21
```

Операциями "*увеличить*" и "*уменьшить*" не следует злоупотреблять. Уже оператор, приведенный в нашем фрагменте, сложен для понимания из-за побочного эффекта. Понимаете ли вы, что при изменении порядка записи слагаемых, как это сделано в закомментированном операторе, результат вычисления *выражения* будет уже не 21, а 22?

Разный *приоритет* префиксных и постфиксных операций носит условный характер. Эти операции применимы только к переменным, свойствам и индексаторам класса, то есть к *выражениям*, которым отведена область памяти. В языках C++ и C# такие *выражения* называются **l-value**, поскольку они могут встречаться в левых частях оператора присваивания. Как следствие, запись в C# *выражения* `< --x++ >` приведет к ошибке. Едва лишь к `x` слева или справа приписана одна из операций, *выражение* перестает принадлежать к классу **l-value** *выражений*, и вторую операцию приписать уже нельзя.

Операции sizeof и typeof

Операция sizeof возвращает размер значимых типов, заданный в байтах. Пояснения требуют некоторые особенности ее применения. Она должна выполняться только в небезопасных блоках. Поэтому проект, в котором она может использоваться, должен быть скомпилирован с включенным свойством `/unsafe`. На рис. 6.1 показано, как на странице свойств проекта можно включить это свойство:

Далее необходимо создать небезопасный блок, например, метод класса, помеченный как `unsafe`, в котором уже можно вызывать эту функцию (операцию). Приведу пример такого метода, созданного в классе `Testing`:

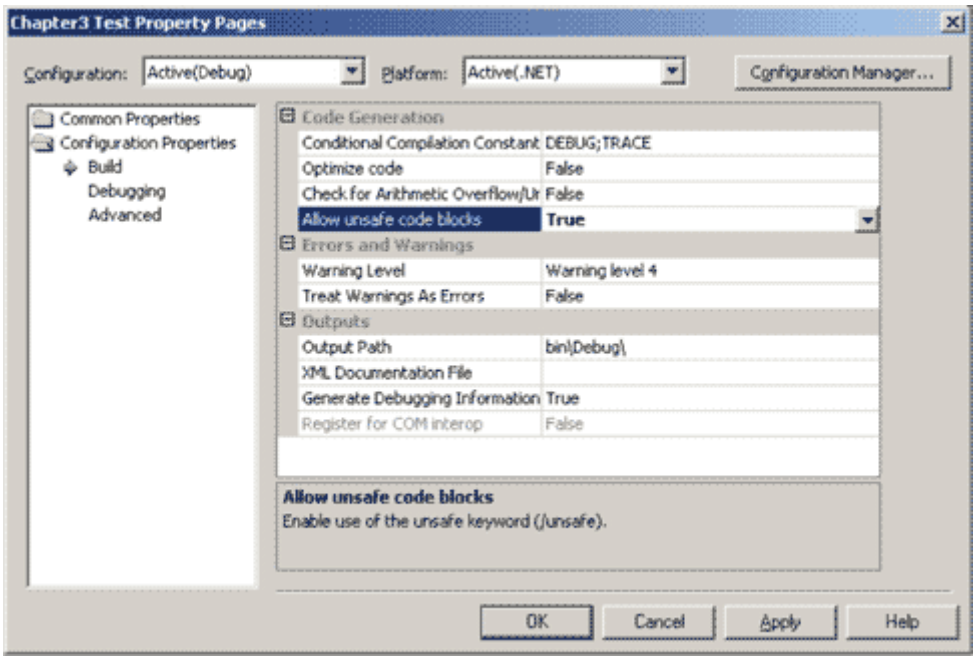

Рис. 6.1. Включение свойства /unsafe

```
/// <summary>
/// определение размеров и типов
/// </summary>
unsafe public static void SizeMethod()
{
   Console.WriteLine("Размер типа Boolean = " + sizeof(bool));
   Console.WriteLine("Размер типа double = " + sizeof(double));
   Console.WriteLine("Размер типа char = " + sizeof(System.Char));
```

```
   int b1=1;
   Type t = b1.GetType();
   Console.WriteLine("Тип переменной b1: {0}", t);
   //Console.WriteLine("Размер переменной b1: {0}", sizeof(t));
}//SizeMethod
```

В этом примере операция применяется к трем встроенным типам - `bool`, `double`, `char`. Консольный вывод дает в качестве результата значения: 1, 8 и 2. Обращаю внимание на то, что аргументом операции может быть только имя типа. Попытка применить эту операцию к переменной `t` типа `Type`, имеющей значение `System.Int32`, приводит к ошибке компиляции.

Операция typeof, примененная к своему аргументу, возвращает его тип. И здесь в роли аргумента может выступать имя класса, как встроенного, так и созданного пользователем. Возвращаемый операцией результат имеет тип `Type`. К экземпляру класса применять операцию нельзя, но зато для экземпляра можно вызвать метод `GetType`, наследуемый всеми классами, и получить тот же результат, что дает *typeof* с именем данного класса. Такой альтернативный способ получения типа по экземпляру класса `int` показан в приведенном выше программном фрагменте. А сейчас я приведу фрагмент, где используется вызов *операции typeof*:

```
t = typeof(Class1);
Console.WriteLine("Тип класса Class1: {0}", t);
t = typeof(Testing);
Console.WriteLine("Тип класса Testing: {0}", t);
```

Как получить подробную информацию о классе?

Пожалуй, следует рассказать не только о том, как можно получить переменную типа `Type`, а и о том, что можно с этой переменной делать.

Этот и последующий раздел прерывают последовательное рассмотрение темы операций языка C#. Полагаю, понимание того, с какой целью выполняются те или иные операции, не менее важно, чем знание самой операции, И я не стал откладывать изложение этого материала на последующие лекции.

Можно ли, зная тип (класс), получить подробную информацию обо всех методах и полях класса? Ясно, что такая информация может быть весьма полезной, если класс поставлен сторонней фирмой. Оказывается, это сделать нетрудно. Вся необходимая информация содержится в метаданных, поставляемых вместе с классом. Процесс получения метаданных называется отражением (**reflection**). Об отражении и метаданных уже говорилось в первой вводной лекции, и эта тема будет обсуждаться и далее. А сейчас я приведу пример, демонстрирующий получение подробной информации о методах и полях класса. Первым делом следует упростить в проекте использование классов пространства имен `Reflection`, добавив в начало проекта предложение

```
using System.Reflection.
```

В класс `Testing` я добавил существенно расширенный вариант метода `WhoIsWho`, который уже появлялся в наших примерах. Вот текст новой версии этой процедуры:

```
/// <summary>
/// Подробная информация о классе объекта, его значении,
/// методах класса, всех членов класса
/// </summary>
/// <param name="name">имя объекта</param>
/// <param name="any">объект любого типа</param>
```

```
public void WhoIsWho(string name,object any)
{
   Type t = any.GetType();
   Console.WriteLine("Тип {0}: {1} , значение: {2}",
      name, any.GetType(), any.ToString());
   Console.WriteLine("Методы класса:");
   MethodInfo[] ClassMethods = t.GetMethods();
   foreach (MethodInfo curMethod in ClassMethods)
   {
      Console.WriteLine(curMethod);
   }
   Console.WriteLine("Все члены класса:");
   MemberInfo[] ClassMembers = t.GetMembers();
   foreach (MemberInfo curMember in ClassMembers)
   {
      Console.WriteLine(curMember.ToString());
   }
}//WhoIsWho
```

Коротко прокомментирую эту процедуру. Вначале создается переменная `t` типа `Type`. Значением этой переменной будет тип аргумента, переданного в процедуру в качестве значения параметра `any`. Напомню, `any` имеет базовый тип `object` и потому метод может быть вызван с аргументом, роль которого может играть *выражение* любого типа. Далее вызываются методы переменной `t` - `GetMethods()` и `GetMembers()`. Эти методы соответственно возвращают в качестве значений массивы элементов классов `MethodInfo` и `MemberInfo`. Эти классы содержатся в пространстве имен `Reflection`, они хранят информацию в первом случае о методах класса, во втором - о полях и методах класса, заданного переменной `t`. В пространстве имен `Reflection` имеются и другие классы, чьи методы и свойства полезны для получения дополнительной информации об исследуемом классе. Но я не буду сейчас столь подробно развивать эту тему.

В процедуре `Main` дважды вызывается процедура `WhoIsWho`. В первом вызове ее аргументом является *выражение* типа `double`, во втором - сам объект `ts`, вызывающий метод:

```
ts.WhoIsWho("2+2.5", 2+2.5);
ts.WhoIsWho("ts", ts);
```

И класс `double`, и созданный в этом проекте класс `Testing` имеют довольно много методов. Имеют они и свойства. Процедура `WhoIsWho` выдаст подробную информацию обо всех элементах этих классов. Результаты консольного вывода, полученного при двух вызовах этой процедуры, показаны на рис. 6.2.

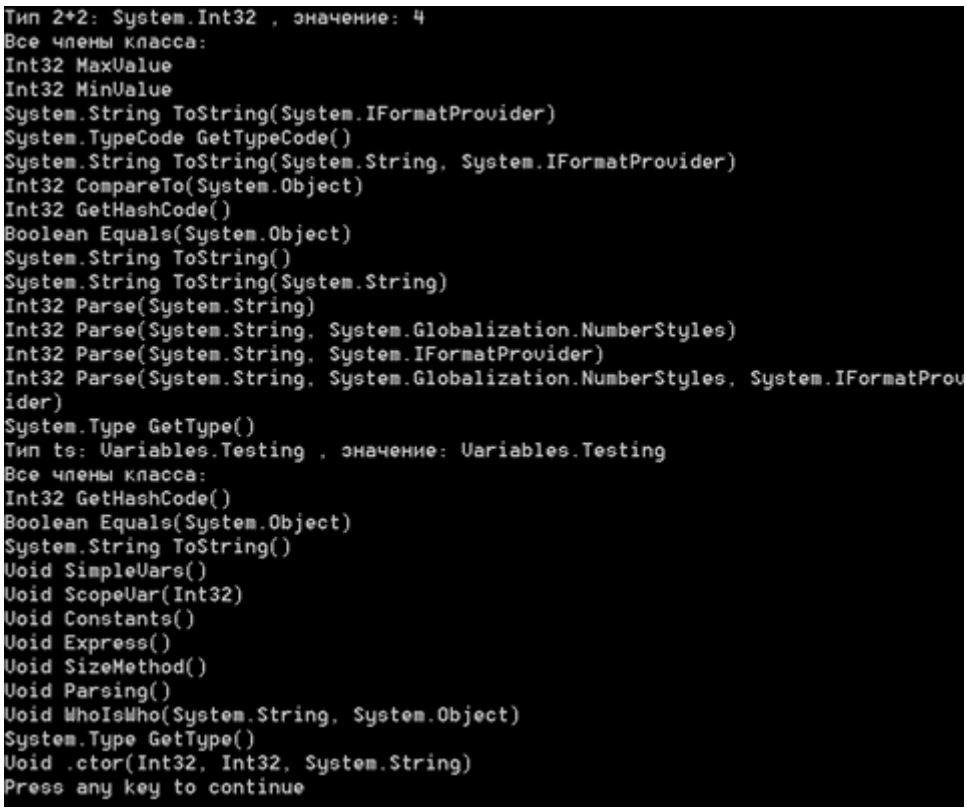

Рис. 6.2. Информация о классах int и Testing, полученная в процедуре WhoIsWho

Рассмотрим выводимую информацию о классах. Для созданного в проекте класса `Testing` отображается информация о полях и методах как собственных, так и наследуемых от общего родителя - класса `Object`. Заметьте, отображается информация только об открытых полях и методах класса, а поскольку поля нашего класса закрыты, то и информации о них нет.

Класс `Int` подробно обсуждался в предыдущей и в этой лекции. Все методы, которые могут вызывать переменные (объекты) класса `Int`, были уже рассмотрены. Тем не менее, из выводимой информации можно узнать и нечто новое, поскольку выдаются сведения и о статических полях и методах класса.

Статические поля и методы арифметических классов

Все арифметические классы, в том числе класс `Int`, обладают двумя полезными полями (свойствами) - `MinValue` и `MaxValue`. Эти поля возвращают минимальное и максимальное значение, которое могут иметь экземпляры класса. Поля являются статическими и потому недоступны для экземпляров класса и могут быть вызваны только при указании имени класса. Разумно привести пример вызова этих полей для класса `Int` и, например, для класса `Double`:

```
//Min и Max значения типов
Console.WriteLine("Class int");
Console.WriteLine("Мин. значение int = " + int.MinValue);
Console.WriteLine("Макс. значение int = " + int.MaxValue);
Console.WriteLine("Class double");
Console.WriteLine("Мин. значение double = " + double.MinValue);
Console.WriteLine("Макс. значение double = " + double.MaxValue);
```

Все арифметические классы, в том числе класс `Int`, обладают перегруженным статическим *методом Parse*, у которого первым обязательным параметром является строка, задающая значение соответствующего арифметического типа в привычной для данного региона (локализованной) форме. Форматом строки и стилем ее представления

можно управлять с помощью других параметров *метода Parse*. Вот пример вызова этого метода для классов `Int` и `Double`:

```
/// <summary>
/// Преобразования типа с использованием метода Parse
/// </summary>
public void Parsing()
{
   //method Parse
   Console.WriteLine("Введите целое");
   string strdata = Console.ReadLine();
   int intdata = int.Parse(strdata);
   Console.WriteLine("Введите число с дробной частью и порядком");
   strdata = Console.ReadLine();
   double doubdata = double.Parse(strdata);
   Console.WriteLine("intdata = {0}; doubdata = {1}",
      intdata, doubdata);
}
//Parsing
```

Как видите, *метод Parse* с успехом заменяет соответствующий метод класса `Convert`.

На рис. 6.3 можно увидеть консольный вывод, полученный в результате работы процедуры `Parsing`.

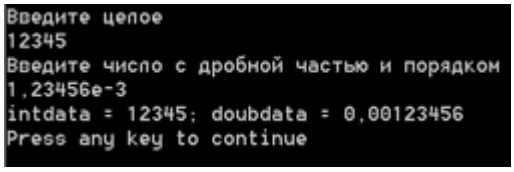

Рис. 6.3. Результаты работы процедуры Parsing

Операция new

Пора вернуться к основной теме - операциям, допустимым в языке C#. Последней из еще не рассмотренных операций высшего уровня *приоритета* является *операция new*. Ключевое слово `new` используется в двух контекстах - как модификатор и как операция в инициализирующих *выражениях* объявителя. Во втором случае результатом выполнения *операции new* является создание нового объекта и вызов соответствующего конструктора. Примеров подобного использования *операции new* было приведено достаточно много, в том числе и в этой лекции.

Арифметические операции

В языке C# имеются обычные для всех языков *арифметические операции* - "`+`, `-`, `*`, `/`, `%`". Все они перегружены. Операции "`+`" и "`-`" могут быть унарными и бинарными. Операция деления "`/`" над целыми типами осуществляет деление нацело, для типов с плавающей и фиксированной точкой - обычное деление. Операция "`%`" определена над всеми арифметическими типами и возвращает остаток от деления нацело. Тип результата зависит от типов операндов. Приведу пример вычислений с различными арифметическими типами:

```
/// <summary>
/// Арифметические операции
/// </summary>
public void Ariphmetica()
{
   int n = 7,m =3, p,q;
   p= n/m; q= p*m + n%m;
```

```
   if (q==n) Console.WriteLine("q=n");
   else Console.WriteLine("q!=n");
   double x=7, y =3, u,v,w;
   u = x/y; v= u*y;
   w= x%y;
   if (v==x) Console.WriteLine("v=x");
   else Console.WriteLine("v!=x");
   decimal d1=7, d2 =3, d3,d4,d5;
   d3 = d1/d2; d4= d3*d2;
   d5= d1%d2;
   if (d4==d1) Console.WriteLine("d4=d1");
   else Console.WriteLine("d4!=d1");
}//Ariphmetica
```

При проведении вычислений в двух первых случаях проверяемое условие оказалось истинным, в третьем - ложным. Для целых типов можно исходить из того, что равенство `n = n/m*m + n%m` истинно. Для типов с плавающей точкой выполнение точного равенства `x = x/y*y` следует считать скорее случайным, а не закономерным событием. Законно невыполнение этого равенства, как это имеет место при вычислениях с фиксированной точкой.

Операции отношения

Операции отношения можно просто перечислить - в объяснениях они не нуждаются. Всего операций 6 (`==`, `!=`, `<`, `>`, `<=`, `>=`). Для тех, кто не привык работать с языком C++, стоит обратить внимание на запись операций "равно" и "не равно".

Операции проверки типов

Операции проверки типов `is` и `as` будут рассмотрены в последующих лекциях. (см. лекцию 19).

Операции сдвига

Операции сдвига вправо "`>>`" и сдвига влево "`<<`" в обычных вычислениях применяются редко. Они особенно полезны, если данные рассматриваются как строка битов. Результатом операции является сдвиг строки битов влево или вправо на `K` разрядов. В применении к обычным целым положительным числам сдвиг вправо равносилен делению нацело на 2^K, а сдвиг влево - умножению на 2^K. Для отрицательных чисел сдвиг влево и деление дают разные результаты, отличающиеся на единицу. В языке C# *операции сдвига* определены только для некоторых целочисленных типов - `int`, `uint`, `long`, `ulong`. Величина сдвига должна иметь тип `int`. Вот пример применения этих операций:

```
/// <summary>
/// операции сдвига
/// </summary>
public void Shift()
{
   int n = 17,m =3, p,q;
   p= n>>2; q = m<<2;
   Console.WriteLine("n= " + n + "; m= " + m + "; p=n>>2 = "+p
      + "; q=m<<2 " + q);
   long x=-75, y =-333, u,v,w;
   u = x>>2; v = y<<2; w = x/4;
   Console.WriteLine("x= " + x + "; y= " + y + "; u=x>>2 = "+u
      + "; v=y<<2 " + v + "; w = x/4 = " + w);
}//Shift
```

Логические операции

Начну с предупреждения тем, кто привык к языку C++. Правила работы с *логическими выражениями* в языках C# и C++ имеют принципиальные различия. В языке C++ практически для всех типов существует неявное преобразование в логический тип. Правило преобразования простое, - ненулевые значения трактуются как истина, нулевое - как ложь. В языке C# неявных преобразований к логическому типу нет даже для целых арифметических типов. Поэтому вполне корректная в языке C++ запись:

```
int k1 = 7;
if (k1) Console.WriteLine("ok!");
```

незаконна в программах на C#. На этапе трансляции возникнет ошибка, поскольку вычисляемое условие имеет тип `int`, а неявное преобразование этого типа к типу `bool` отсутствует.

В языке C# более строгие правила действуют и для *логических операций*. Так, запись

```
if(k1 && (x>y)),
```

корректная в языке C++, приводит к ошибке в программах на C#, поскольку операция `&&` определена только для операндов типа `bool`, а в данном *выражении* один из операндов имеет тип `int`. В языке C# в данных ситуациях следует использовать записи:

```
if(k1>0)
if((k1>0) && (x>y))
```

После этого важного предупреждения перейду к более систематическому изложению некоторых особенностей выполнения *логических операций*. Так же, как и в языке C++, *логические операции* делятся на две категории: одни выполняются над логическими значениями операндов, другие осуществляют выполнение *логической операции* над битами операндов. По этой причине в C# существуют две унарные операции отрицания - логическое отрицание, заданное операцией "`!`", и побитовое отрицание, заданное операцией "`~`". Первая из них определена над операндом типа `bool`, вторая - над операндом целочисленного типа, начиная с типа `int` и выше (`int`, `uint`, `long`, `ulong`). Результатом операции во втором случае является операнд, в котором каждый бит заменен его дополнением. Приведу пример:

```
/// <summary>
/// Логические выражения
/// </summary>
public void Logic()
{
   //операции отрицания ~,!
   bool b1,b2;
   b1 = 2*2==4; b2 =!b1;
   //b2= ~b1;
   uint j1 =7, j2;
   j2= ~j1;
   //j2 = !j1;
   int j4 = 7, j5;
   j5 = ~j4;
   Console.WriteLine("uint j2 = " + j2 +" int j5 = " + j5);
}//Logic
```

В этом фрагменте закомментированы операторы, приводящие к ошибкам. В первом случае была сделана попытка применения операции побитового отрицания к *выражению* типа `bool`, во втором - логическое отрицание применялось к

целочисленным данным. И то, и другое в C# незаконно. Обратите внимание на разную интерпретацию побитового отрицания для беззнаковых и знаковых целочисленных типов. Для переменных `j5` и `j2` строка битов, задающая значение - одна и та же, но интерпретируется по-разному. Соответствующий вывод таков:

```
uint j2 = 4294967288 int j5 = -8
```

Бинарные *логические операции* "`&&` - условное **И**" и "`||` - условное **ИЛИ**" определены только над данными типа `bool`. Операции называются условными или краткими, поскольку, будет ли вычисляться второй операнд, зависит от уже вычисленного значения первого операнда. В операции "`&&`", если первый операнд равен значению `false`, то второй операнд не вычисляется и результат операции равен `false`. Аналогично, в операции "`||`", если первый операнд равен значению `true`, то второй операнд не вычисляется и результат операции равен `true`. Ценность *условных логических операций* заключается не в их эффективности по времени выполнения. Часто они позволяют вычислить *логическое выражение*, имеющее смысл, но в котором второй операнд не определен. Приведу в качестве примера классическую задачу поиска по образцу в массиве, когда разыскивается элемент с заданным значением (образец). Такой элемент в массиве может быть, а может и не быть. Вот типичное решение этой задачи в виде упрощенном, но передающем суть дела:

```
//Условное And - &&
int[] ar= {1,2,3};
int search = 7; int i=0;
while ((i < ar.Length) && (ar[i]!= search)) i++;
if(i<ar.Length) Console.WriteLine("Образец найден");
else Console.WriteLine("Образец не найден");
```

Если значение переменной `search` (образца) не совпадает ни с одним из значений элементов массива `ar`, то последняя проверка условия цикла `while` будет выполняться при значении `i`, равном `ar.Length`. В этом случае первый операнд получит значение `false`, и, хотя второй операнд при этом не определен, цикл нормально завершит свою работу. Второй операнд не определен в последней проверке, поскольку индекс элемента массива выходит за допустимые пределы (в C# индексация элементов начинается с нуля). Заметьте, что "нормальная" конъюнкция требует вычисления обеих операндов, поэтому ее применение в данной программе приводило бы к выбросу исключения в случае, когда образца нет в массиве.

Три бинарные побитовые операции - "`&` - **AND** " , "`|` - **OR** ", "`^` - **XOR**" используются двояко. Они определены как над целыми типами выше `int`, так и над булевыми типами. В первом случае они используются как побитовые операции, во втором - как обычные *логические операции*. Иногда необходимо, чтобы оба операнда вычислялись в любом случае, тогда без этих операций не обойтись. Вот пример первого их использования:

```
//Логические побитовые операции And, Or, XOR (&,|,^)
int k2 = 7, k3 = 5, k4, k5, k6;
k4 = k2 & k3; k5 = k2| k3; k6 = k2^k3;
Console.WriteLine("k4 = " + k4 + " k5 = " + k5 + " k6 = " + k6);
```

Приведу результаты вывода:

```
k4 = 5 k5 = 7 k6 =2
```

Приведу пример поиска по образцу с использованием логического **AND**:

```
i=0; search = ar[ar.Length - 1];
while ((i < ar.Length) & (ar[i]!= search)) i++;
if(i<ar.Length) Console.WriteLine("Образец найден");
```

```
else Console.WriteLine("Образец не найден");
```

В данном фрагменте гарантируется наличие образца поиска в массиве, и фрагмент будет успешно выполнен. В тех же случаях, когда массив не содержит элемента `search`, будет выброшено исключение. Содержательный смысл такой процедуры - появление исключения - может быть признаком ошибки в данных, что требует специальной обработки ситуации.

Условное выражение

В C#, как и в C++, разрешены *условные выражения*. Конечно, без них можно обойтись, заменив их условным оператором. Вот простой пример их использования, поясняющий синтаксис их записи:

```
//Условное выражение
int a = 7, b= 9, max;
max= (a>b) ? a:b;
Console.WriteLine("a = " + a + "; b= " + b +
   "; max(a,b) = " + max);
```

Условное выражение начинается с условия, заключенного в круглые скобки, после которого следует знак вопроса и пара *выражений*, разделенных двоеточием " `:` ". Условием является *выражение* типа `bool`. Если оно истинно, то из пары *выражений* выбирается первое, в противном случае результатом является значение второго *выражения*. В данном примере переменная `max` получит значение `9`.

Операция приведения к типу

Осталось рассмотреть еще одну операцию - ***приведение к типу***. Эта операция первого *приоритета* имеет следующий синтаксис:

```
(type) <унарное выражение>
```

Она задает явное преобразование типа, определенного *выражением*, к типу, указанному в скобках. Чтобы операция была успешной, необходимо, чтобы такое явное преобразование существовало. Напомню, существуют явные преобразования внутри арифметического типа, но не существует, например, явного преобразования арифметического типа в тип `bool`. При определении пользовательских типов для них могут быть заданы явные преобразования в другие, в том числе встроенные, типы. О явных преобразованиях говорилось достаточно много, приводились и примеры. Поэтому ограничусь совсем простым примером:

```
//cast
int p;
p = (int)x;
//b = (bool)x;
```

В данном примере явное преобразование из типа `double` в тип `int` выполняется, а преобразование `double` в тип `bool` приводит к ошибке, потому и закомментировано.

7. Лекция: Присваивание и встроенные функции

Присваивание

В большинстве языков программирования *присваивание* - это оператор, а не операция. В языке C# *присваивание* унаследовало многие особенности *присваивания* языка C++. В C# оно толкуется как операция, используемая в выражениях. Однако в большинстве случаев *присваивание* следует рассматривать и использовать как обычный оператор.

Возьмем полезный случай реального использования *присваивания* как операции. В ситуации, называемой множественным *присваиванием*, списку переменных присваивается одно и тоже значение. Вот пример:

```
/// <summary>
/// анализ присваивания
/// </summary>
public void Assign()
{
   double x,y,z,w =1, u =7, v= 5;
   x = y = z = w =(u+v+w)/(u-v-w);
}//Assign
```

По мере изложения в метод `Assign` будут добавляться фрагменты кода, связанные с рассматриваемой темой *присваивания*.

О семантике *присваивания* говорилось уже достаточно много. Но следует внести еще некоторые уточнения. Правильно построенное выражение *присваивания* состоит из левой и правой части. Левая часть - это список переменных, в котором знак равенства выступает в качестве разделителя. Правая часть - это выражение. Выражение правой части вычисляется, при необходимости приводится к типу переменных левой части, после чего все переменные левой части получают значение вычисленного выражения. Последние действия можно рассматривать как побочный эффект операции *присваивания*. Заметьте, все переменные в списке левой части должны иметь один тип или неявно приводиться к одному типу. Операция *присваивания* выполняется справа налево, поэтому вначале значение выражения получит самая правая переменная списка левой части, при этом значение самого выражения не меняется. Затем значение получает следующая справа по списку переменная - и так до тех пор, пока не будет достигнут конец списка. Так что реально можно говорить об одновременном *присваивании*, в котором все переменные списка получают одно и то же значение. В нашем примере, несмотря на то, что переменная `w` первой получит значение, а выражение в правой части зависит от `w`, все переменные будут иметь значение `13.0`. Рассмотрим еще один фрагмент кода:

```
bool b;
x=5; y=6;
//b= x=y;
//if (x=y) z=1;else z=-1;
```

В программе на языке C++ можно было снять комментарии с операторов, и этот фрагмент кода компилировался и выполнялся бы без ошибок. Другое дело, что результат мог быть некорректен, поскольку, вероятнее всего, операция *присваивания* "`x=y`" написана по ошибке и ее следует заменить операцией эквивалентности "`x==y`". В языке C# оба закомментированных оператора, к счастью, приведут к ошибке трансляции, поскольку результат *присваивания* имеет тип `double`, для которого нет неявного преобразования в тип `bool`. На C# такая программа будет выполняться, только если `x` и `y` будут иметь тип `bool`, но в этом случае, возможно, применение операции *присваивания* имеет смысл. С типами `double` корректная программа на C# может быть такой:

```
x =y;
b= (y!=0);
if(y!=0) z=1; else z = -1;
```

В программе появился лишний оператор, но исчезла двусмысленность, порождаемая операцией *присваивания*.

Специальные случаи присваивания

В языке C++ для двух частных случаев *присваивания* предложен отдельный синтаксис. Язык C# наследовал эти полезные свойства. Для *присваиваний* вида "`x=x+1`", в которых переменная увеличивается или уменьшается на единицу, используются специальные префиксные и постфиксные операции "`++`" и "`--`". Другой важный частный случай - это *присваивания* вида:

```
X = X <operator> (expression)
```

Для таких *присваиваний* используется краткая форма записи:

```
X <operator>= expression
```

В качестве операции разрешается использовать арифметические, логические (побитовые) операции и операции сдвига языка C#. Семантика такого *присваивания* достаточно очевидна, и я ограничусь простым примером:

```
x += u+v; y /=(u-v);
b &= (x<y);
```

Однако и здесь есть один подводный камень, когда `x= x+a` не эквивалентно `x +=a`. Рассмотрим следующий пример:

```
byte b3 = 21;
b3 +=1; //Это допустимо
//b3 = b3+1; //А это недопустимо:результат типа int
```

Закомментированный оператор приведет к ошибке компиляции, поскольку правая часть имеет тип `int`, а неявное преобразование к типу `byte` отсутствует. Следует понимать, что преимущество первой формы записи - только кажущееся: если при инициализации переменная `b` получит допустимое значение `255`, то следующий оператор *присваивания* в краткой форме не выдаст ошибки, но даст неверный результат, а это - самое худшее, что может случиться в программе. Так что надежнее пользоваться полной формой записи *присваивания*, не экономя на паре символов.

Определенное присваивание

Присваивание в языке C# называется ***определенным присваиванием (definite assignment)***. В этом термине отражен тот уже обсуждавшийся факт, что все используемые в выражениях переменные должны быть ранее инициализированы и иметь определенные значения. Единственное, за чем компилятор не следит, так это за инициализацией переменных массива. Для них используется инициализация элементов, задаваемая по умолчанию. Приведу пример:

```
//определенное присваивание
int an =0 ; //переменные должны быть инициализированы
for (int i= 0;i<5;i++)
{an =i+1;}
x+=an; z+=an; y = an;
string[] ars = new string[3];
```

```
double[] ard = new double[3];
for (int i= 0;i<3;i++)
{
   //массивы могут быть без инициализации
   ard[i] += i+1;
   ars[i] += i.ToString()+1;
   Console.WriteLine("ard[" +i + "]=" +ard[i] +
      ";  ars[" +i + "]=" +ars[i]);
}
```

Заметьте, в этом фрагменте переменная an обязана быть инициализированной, а массивы `ard` и `ars` не инициализируются и спокойно участвуют в вычислениях.

Еще раз о семантике присваивания

Подводя итоги рассмотрения *присваивания* `x=e`, следует отметить, что семантика *присваивания* далеко не столь проста, как может показаться с первого взгляда. Напомню, что деление типов на значимые и ссылочные приводит к двум семантикам *присваивания*. Будет ли семантика значимой или ссылочной - определяется типом левой части *присваивания*. Переменные значимых типов являются единоличными владельцами памяти, в которой хранятся их значения. При значимом *присваивании* память для хранения значений остается той же - меняются лишь сами значения, хранимые в ней. Переменные ссылочных типов (объекты) являются ссылками на реальные объекты динамической памяти. Ссылки могут разделять одну и ту же область памяти - ссылаться на один и тот же объект. Ссылочное *присваивание* - это операция над ссылками. В результате ссылочного *присваивания* ссылка начинает указывать на другой объект.

Рассмотрим объявления:

```
int x=3, y=5;
object obj1, obj2;
```

Здесь объявлены четыре сущности: две переменные значимого типа и две - объектного. Значимые переменные `x` и `y` проинициализированы и имеют значения, объектные переменные `obj1` и `obj2` являются пустыми ссылками со значением `void`. Рассмотрим *присваивания*:

```
obj1 = x; obj2 = y;
```

Эти *присваивания* ссылочные (из-за типа левой части), поэтому правая часть приводится к ссылочному типу. В результате неявного преобразования - операции `boxing` - в динамической памяти создаются два объекта, обертывающие соответственно значения переменных `x` и `y`. Сущности `obj1` и `obj2` получают значения ссылок на эти объекты.

Класс Math и его функции

Кроме переменных и констант, первичным материалом для построения выражений являются функции. Большинство их в проекте будут созданы самим программистом, но не обойтись и без *встроенных функций*. Умение работать в среде Visual Studio .Net предполагает знание встроенных возможностей этой среды, знание возможностей каркаса Framework .Net, пространств имен, доступных при программировании на языке C#, а также соответствующих *встроенных классов* и функций этих классов. Продолжим знакомство с возможностями, предоставляемыми пространством имен `System`. Мы уже познакомились с классом `Convert` этого пространства и частично с классом `Console`. Давайте рассмотрим еще один класс - *класс Math*, содержащий стандартные

математические функции, без которых трудно обойтись при построении многих выражений. Этот класс содержит два статических поля, задающих константы `E` и `PI`, а также 23 статических метода. Методы задают:

- тригонометрические функции - `Sin`, `Cos`, `Tan`;
- обратные тригонометрические функции - `ASin`, `ACos`, `ATan`, `ATan2 (sinx, cosx)`;
- гиперболические функции - `Tanh`, `Sinh`, `Cosh`;
- экспоненту и логарифмические функции - `Exp`, `Log`, `Log10`;
- модуль, корень, знак - `Abs`, `Sqrt`, `Sign`;
- функции округления - `Ceiling`, `Floor`, `Round`;
- минимум, максимум, степень, остаток - `Min`, `Max`, `Pow`, `IEEEReminder`.

В особых пояснениях эти функции не нуждаются. Приведу пример:

```
/// <summary>
/// работа с функциями класса Math
/// </summary>
public  void MathFunctions()
{
   double a, b,t,t0,dt,y;
   string NameFunction;
   Console.WriteLine("Введите имя F(t)исследуемой функции
      a*F(b*t)" + " (sin, cos, tan, cotan)");
   NameFunction = Console.ReadLine();
   Console.WriteLine("Введите параметр a (double)");
   a= double.Parse(Console.ReadLine());
   Console.WriteLine("Введите параметр b (double)");
   b= double.Parse(Console.ReadLine());
   Console.WriteLine("Введите начальное время t0(double)");
   t0= double.Parse(Console.ReadLine());
   const int points = 10;
   dt = 0.2;
   for(int i = 1; i<=points; i++)
   {
      t = t0 + (i-1)* dt;
      switch (NameFunction)
      {
         case ("sin"):
            y = a*Math.Sin(b*t);
            break;
         case ("cos"):
            y = a*Math.Cos(b*t);
            break;
         case ("tan"):
            y = a*Math.Tan(b*t);
            break;
         case ("cotan"):
            y = a/Math.Tan(b*t);
            break;
         case ("ln"):
            y = a*Math.Log(b*t);
            break;
         case ("tanh"):
            y = a*Math.Tanh(b*t);
            break;
         default:
            y=1;
            break;
      }//switch
      Console.WriteLine ("t = " + t + "; " + a +"*" +
         NameFunction +"(" + b + "*t)= " + y + ";");
   }//for
```

```
    double u = 2.5, v = 1.5, p,w;
    p= Math.Pow(u,v);
    w = Math.IEEERemainder(u,v);
    Console.WriteLine ("u = " + u + "; v= " + v +
       "; power(u,v)= " + p + "; reminder(u,v)= " + w);
}//MathFunctions
```

Заметьте, в примерах программного кода я постепенно расширяю диапазон используемых средств. Часть из этих средств уже описана, а часть (например, оператор цикла `for` и оператор выбора `switch`) будут описаны позже. Те, у кого чтение примеров вызывает затруднение, смогут вернуться к ним при повторном чтении книги.

Коротко прокомментирую этот код. В данном примере пользователь определяет, какую функцию он хочет вычислить и при каких значениях ее параметров. Некоторые параметры задаются константами и инициализированными переменными, но для большинства их значения вводятся пользователем. Одна из целей этого фрагмента состоит в демонстрации консольного ввода данных разного типа, при котором используется описанный ранее метод `Parse`.

Функция, заданная пользователем, вычисляется в операторе `switch`. Здесь реализован выбор из 6 стандартных функций, входящих в джентльменский набор *класса Math*.

Вызов еще двух функций из *класса Math* содержится в двух последних строчках этой процедуры. На рис. 7.1 можно видеть результаты ее работы.

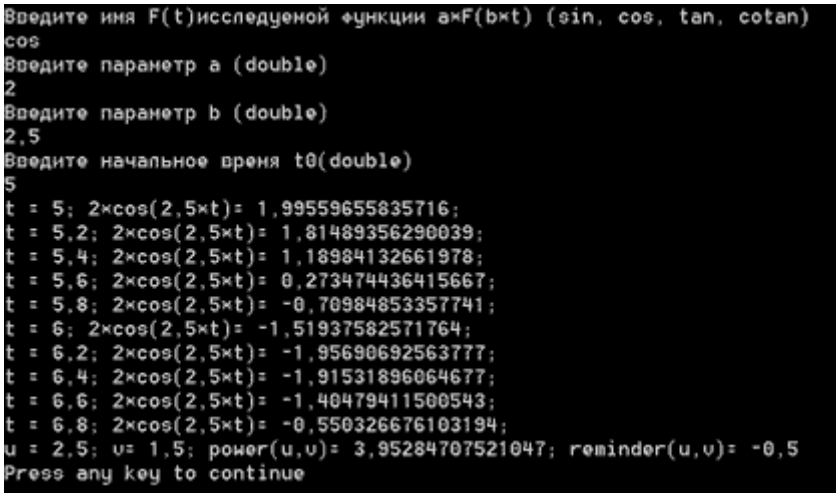

Рис. 7.1. Результаты работы процедуры MathFunctions

Класс Random и его функции

Умение генерировать случайные числа требуется во многих приложениях. ***Класс Random*** содержит все необходимые для этого средства. *Класс Random* имеет конструктор класса: для того, чтобы вызывать методы класса, нужно вначале создавать экземпляр класса. Этим *Random* отличается от *класса Math*, у которого все поля и методы - статические, что позволяет обойтись без создания экземпляров *класса Math*.

Как и всякий "настоящий" класс, *класс Random* является наследником класса `Object`, а, следовательно, имеет в своем составе и методы родителя. Рассмотрим только оригинальные методы *класса Random* со статусом `public`, необходимые для генерирования последовательностей случайных чисел. Класс имеет защищенные методы, знание которых полезно при необходимости создания собственных потомков *класса Random*, но этим мы заниматься не будем.

Начнем рассмотрение с конструктора класса. Он перегружен и имеет две реализации. Одна из них позволяет генерировать неповторяющиеся при каждом запуске серии случайных чисел. Начальный элемент такой серии строится на основе текущей даты и времени, что гарантирует уникальность серии. Этот конструктор вызывается без параметров. Он описан как `public Random()`. Другой конструктор с параметром - `public Random (int)` обеспечивает важную возможность генерирования повторяющейся серии случайных чисел. Параметр конструктора используется для построения начального элемента серии, поэтому при задании одного и того же значения параметра серия будет повторяться.

Перегруженный метод `public int Next()` при каждом вызове возвращает положительное целое, равномерно распределенное в некотором диапазоне. Диапазон задается параметрами метода. Три реализации метода отличаются набором параметров:

- `public int Next ()` - метод без параметров выдает целые положительные числа во всем положительном диапазоне типа `int`;
- `public int Next (int max)` - выдает целые положительные числа в диапазоне `[0,max]`;
- `public int Next (int min, int max)` - выдает целые положительные числа в диапазоне `[min,max]`.

Метод `public double NextDouble ()` имеет одну реализацию. При каждом вызове этого метода выдается новое случайное число, равномерно распределенное в интервале `[0-1)`.

Еще один полезный метод *класса Random* позволяет при одном обращении получать целую серию случайных чисел. Метод имеет параметр - массив, который и будет заполнен случайными числами. Метод описан как `public void NextBytes (byte[] buffer)`. Так как параметр `buffer` представляет массив байтов, то, естественно, генерированные случайные числа находятся в диапазоне `[0, 255]`.

Приведу теперь пример работы со случайными числами. Как обычно, для проведения экспериментов по генерации случайных чисел я создал метод `Rand` в классе `Testing`. Вот программный код этого метода:

```
/// <summary>
/// Эксперименты с классом Random
/// </summary>
public  void Rand()
{
   const int  initRnd = 77;
   Random realRnd = new Random();
   Random repeatRnd = new Random(initRnd);
   // случайные числа в диапазоне [0,1)
   Console.WriteLine("случайные числа в диапазоне[0,1)");
   for(int i =1; i <= 5; i++)
   {
      Console.WriteLine("Число " + i + "= "
         + realRnd.NextDouble() );
   }
   // случайные числа в диапазоне[min,max]
   int min = -100, max=-10;
   Console.WriteLine("случайные числа в диапазоне [" +
      min +"," + max + "]");
   for(int i =1; i <= 5; i++)
   {
      Console.WriteLine("Число " + i + "= "
         + realRnd.Next(min,max) );
   }
```

```
    // случайный массив байтов
    byte[] bar = new byte[10];
    repeatRnd.NextBytes(bar);
    Console.WriteLine("Массив случайных чисел в диапазоне
       [0, 255]");
    for(int i =0; i < 10; i++)
    {
       Console.WriteLine("Число " + i + "= " +bar[i]);
    }
}//Rand
```

Приведу краткий комментарий к тексту программы. Вначале создаются два объекта *класса Random*. У этих объектов разные конструкторы. Объект с именем `realRnd` позволяет генерировать неповторяющиеся серии случайных чисел. Объект `repeatRnd` дает возможность повторить при необходимости серию. Метод `NextDouble` создает серию случайных чисел в диапазоне `[0, 1)`. Вызываемый в цикле метод `Next` с двумя параметрами создает серию случайных отрицательных целых, равномерно распределенных в диапазоне `[-100, -10]`. Метод `NextBytes` объекта `repeatRnd` позволяет получить при одном вызове массив случайных чисел из диапазона `[0, 255]`. Результаты вывода можно увидеть на рис. 7.2.

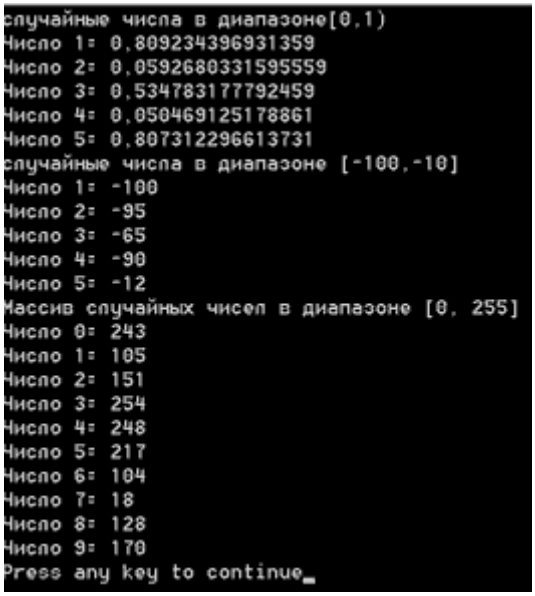

Рис. 7.2. Генерирование последовательностей случайных чисел в процедуре Rand

На этом заканчивается рассмотрение темы выражений языка C#.

8. Лекция: Операторы языка C#

Операторы языка C#

Состав операторов языка C#, их синтаксис и семантика унаследованы от языка C++. Как и положено, потомок частично дополнил состав, переопределил синтаксис и семантику отдельных операторов, постарался улучшить характеристики языка во благо программиста. Посмотрим, насколько это удалось языку C#.

Оператор присваивания

Как в языке C++, так и в C# *присваивание* формально считается операцией. Вместе с тем запись:

```
X= expr;
```

следует считать настоящим *оператором присваивания*, так же, как и одновременное *присваивание* со списком переменных в левой части:

```
X1 = X2 = ... = Xk = expr;
```

В отличие от языка C++ появление *присваивания* в выражениях C# хотя и допустимо, но практически не встречается. Например, запись:

```
if(x = expr)...
```

часто используемая в C++, в языке C# в большинстве случаев будет воспринята как ошибка еще на этапе компиляции.

В предыдущих лекциях семантика *присваивания* разбиралась достаточно подробно, поэтому сейчас я на этом останавливаться не буду.

Блок или составной оператор

С помощью фигурных скобок несколько операторов языка (возможно, перемежаемых объявлениями) можно объединить в единую синтаксическую конструкцию, называемую ***блоком*** или ***составным оператором***:

```
{
   оператор_1
   ...
   оператор_N
}
```

В языках программирования нет общепринятой нормы для использования символа точки с запятой при записи последовательности операторов. Есть три различных подхода и их вариации. Категорические противники точек с запятой считают, что каждый оператор должен записываться на отдельной строке (для длинных операторов определяются правила переноса). В этом случае точки с запятой (или другие аналогичные разделители) не нужны. Горячие поклонники точек с запятой (к ним относятся языки C++ и C#) считают, что точкой с запятой должен оканчиваться каждый оператор. В результате в *операторе if* перед `else` появляется точка с запятой. Третьи полагают, что точка с запятой играет роль разделителя операторов, поэтому перед `else` ее не должно быть. В приведенной выше записи *блока*, следуя синтаксису C#, каждый из операторов заканчивается символом "точка с запятой". Но, заметьте, *блок* не заканчивается этим символом!

Синтаксически *блок* воспринимается как единичный оператор и может использоваться всюду в конструкциях, где синтаксис требует одного оператора. Тело цикла, ветви *оператора if*, как правило, представляются *блоком*. Приведу достаточно формальный и слегка запутанный пример, где тело процедуры представлено *блоком*, в котором есть встроенные *блоки*, задающие тело *оператора цикла for* и тела ветвей *оператора if*:

```
/// <summary>
/// демонстрация блоков (составных операторов)
/// </summary>
public void Block()
{
   int limit = 100;
   int x = 120, y = 50;
   int sum1 =0, sum2=0;
   for (int i = 0; i< 11; i++)
   {
      int step = Math.Abs(limit -x)/10;
      if (x > limit)
         {x -= step; y += step;}
      else
         {x += step; y -= step;}
      sum1 += x; sum2 +=y;
   }
   //limit = step;     //переменная step перестала существовать
   //limit = i;// переменная i перестала существовать
   Console.WriteLine("x= {0}, y= {1}, sum1 ={2}, sum2 = {3}",
      x,y,sum1,sum2);
}
```

Заметьте, здесь в тело основного *блока* вложен *блок*, задающий тело цикла, в котором объявлены две локальные переменные - `i` и `step`.

В свою очередь, в тело цикла вложены *блоки*, связанные с ветвями `then` и `else` *оператора if*. Закомментированные операторы, стоящие сразу за окончанием цикла, напоминают, что соответствующие локальные переменные, определенные в *блоке*, перестают существовать по его завершении.

Приведенная процедура `Block` является методом класса `Testing`, который входит в проект `Statements`, созданный для работы с примерами этой лекции. Вот описание полей и конструктора класса `Testing`:

```
/// <summary>
/// Класс Testing - тестирующий класс. Представляет набор
/// скалярных переменных и методов тестирующих работу
/// с операторами, процедурами и функциями C#.
/// </summary>
public class Testing
{
   public Testing(string name, int age)
   {
      this.age = age;
      this.name = name;
   }
   //поля класса
   public string name;
   public int age;

   private int period;
   private string status;
```

Пустой оператор

Пустой оператор - это "пусто", завершаемое точкой с запятой. Иногда полезно рассматривать отсутствие операторов как существующий *пустой оператор*. Синтаксически допустимо ставить лишние точки с запятой, полагая, что вставляются *пустые операторы*. Например, синтаксически допустима следующая конструкция:

```
for (int j=1; j<5; j++)
        {;;;};
```

Она может рассматриваться как задержка по времени, работа на холостом ходе.

Операторы выбора

Как в C++ и других языках программирования, в языке C# для выбора одной из нескольких возможностей используются две конструкции - *if* и *switch*. Первую из них обычно называют альтернативным выбором, вторую - разбором случаев.

Оператор if

Начнем с синтаксиса *оператора if*:

```
if(выражение_1) оператор_1
else if(выражение_2) оператор_2
...
else if(выражение_K) оператор_K
else оператор_N
```

Какие особенности синтаксиса следует отметить? Выражения *if* должны заключаться в круглые скобки и быть булевого типа. Точнее, выражения должны давать значения `true` или `false`. Напомню, арифметический тип не имеет явных или неявных преобразований к булевому типу. По правилам синтаксиса языка C++, `then`-ветвь оператора следует сразу за круглой скобкой без ключевого слова `then`, типичного для большинства языков программирования. Каждый из операторов может быть *блоком* - в частности, *if*-оператором. Поэтому возможна и такая конструкция:

```
if(выражение1) if(выражение2) if(выражение3) ...
```

Ветви `else` и *if*, позволяющие организовать выбор из многих возможностей, могут отсутствовать. Может быть опущена и заключительная `else`-ветвь. В этом случае краткая форма *оператора if* задает альтернативный выбор - делать или не делать - выполнять или не выполнять `then`-оператор.

Семантика *оператора if* проста и понятна. Выражения *if* проверяются в порядке их написания. Как только получено значение `true`, проверка прекращается и выполняется оператор (это может быть *блок*), который следует за выражением, получившим значение `true`. С завершением этого оператора завершается и *оператор if*. Ветвь `else`, если она есть, относится к ближайшему открытому *if*.

Оператор switch

Частным, но важным случаем выбора из нескольких вариантов является ситуация, при которой выбор варианта определяется значениями некоторого выражения. Соответствующий оператор C#, унаследованный от C++, но с небольшими изменениями в синтаксисе, называется *оператором switch*. Вот его синтаксис:

```
switch(выражение)
{
   case константное_выражение_1: [операторы_1 оператор_перехода_1]
```

```
    ...
    case константное_выражение_K: [операторы_K оператор_перехода_K]
    [default: операторы_N оператор_перехода_N]
}
```

Ветвь `default` может отсутствовать. Заметьте, по синтаксису допустимо, чтобы после двоеточия следовала пустая последовательность операторов, а не последовательность, заканчивающаяся *оператором перехода*. Константные выражения в case должны иметь тот же тип, что и *switch*-выражение.

Семантика *оператора switch* чуть запутана. Вначале вычисляется значение `switch`-выражения. Затем оно поочередно в порядке следования case сравнивается на совпадение с константными выражениями. Как только достигнуто совпадение, выполняется соответствующая последовательность операторов `case`-ветви. Поскольку последний оператор этой последовательности является *оператором перехода* (чаще всего это оператор `break`), то обычно он завершает выполнение *оператора switch*. Использование *операторов перехода* - это плохая идея. Таким оператором может быть *оператор goto*, передающий управление другой `case`-ветви, которая, в свою очередь, может передать управление еще куда-нибудь, получая блюдо "спагетти" вместо хорошо структурированной последовательности операторов. Семантика осложняется еще и тем, что `case`-ветвь может быть пустой последовательностью операторов. Тогда в случае совпадения константного выражения этой ветви со значением `switch`-выражения будет выполняться первая непустая последовательность очередной `case`-ветви. Если значение `switch`-выражения не совпадает ни с одним константным выражением, то выполняется последовательность операторов ветви `default`, если же таковой ветви нет, то *оператор switch* эквивалентен *пустому оператору*.

Полагаю, что *оператор switch* - это самый неудачный оператор языка C# как с точки зрения синтаксиса, так и семантики. Неудачный синтаксис порождает запутанную семантику, являющуюся источником плохого стиля программирования. Понять, почему авторов постигла неудача, можно, оправдать - нет. Дело в том, что оператор унаследован от C++, где его семантика и синтаксис еще хуже. В языке C# синтаксически каждая `case`-ветвь должна заканчиваться *оператором перехода* (забудем на минуту о пустой последовательности), иначе возникнет ошибка периода компиляции. В языке C++ это правило не является синтаксически обязательным, хотя на практике применяется та же конструкция с конечным оператором `break`. При его отсутствии управление "проваливается" в следующую `case`-ветвь. Конечно, профессионал может с успехом использовать этот трюк, но в целом ни к чему хорошему это не приводит. Борясь с этим, в C# потребовали обязательного включения *оператора перехода*, завершающего ветвь. Гораздо лучше было бы, если бы последним оператором мог быть только оператор `break`, писать его было бы не нужно и семантика стала бы прозрачной - при совпадении значений двух выражений выполняются операторы соответствующей `case`-ветви, при завершении которой завершается и *оператор switch*.

Еще одна неудача в синтаксической конструкции *switch* связана с существенным ограничением, накладываемым на `case`-выражения, которые могут быть только константным выражением. Уж если изменять оператор, то гораздо лучше было бы использовать синтаксис и семантику Visual Basic, где в `case`-выражениях допускается список, каждое из выражений которого может задавать диапазон значений.

Разбор случаев - это часто встречающаяся ситуация в самых разных задачах. Применяя *оператор switch*, помните о недостатках его синтаксиса, используйте его в правильном стиле. Заканчивайте каждую `case`-ветвь оператором `break`, но не применяйте *goto*.

Когда разбор случаев предполагает проверку попадания в некоторый диапазон значений, приходится прибегать к *оператору if* для формирования специальной

переменной. Этот прием демонстрируется в следующем примере, где идет работа над данными нашего класса `Testing`:

```
/// <summary>
/// Определяет период в зависимости от возраста - age
/// Использование ветвящегося оператора if
/// </summary>
public void SetPeriod()
{
   if ((age > 0)&& (age <7))period=1;
   else if ((age >= 7)&& (age <17))period=2;
   else if ((age >= 17)&& (age <22))period=3;
   else if ((age >= 22)&& (age <27))period=4;
   else if ((age >= 27)&& (age <37))period=5;
   else period =6;
}
```

Этот пример демонстрирует применение ветвящегося *оператора if*. С содержательной точки зрения он интересен тем, что в поля класса пришлось ввести специальную переменную `period`, позволяющую в дальнейшем использовать разбор случаев в зависимости от периода жизни:

```
/// <summary>
/// Определяет статус в зависимости от периода - period
/// Использование разбора случаев - оператора Switch
/// </summary>
public void SetStatus()
{
   switch (period)
   {
      case 1:
         status = "child";
         break;
      case 2:
         status = "schoolboy";
         break;
      case 3:
         status = "student";
         break;
      case 4:
         status = "junior researcher";
         break;
      case 5:
         status = "senior researcher";
         break;
      case 6:
         status = "professor";
         break;
      default :
         status = "не определен";
         break;
   }
   Console.WriteLine("Имя = {0}, Возраст = {1}, Статус = {2}",
      name, age, status);
}//SetStatus
```

Этот пример демонстрирует корректный стиль использования *оператора switch*. В следующем примере показана роль пустых последовательностей операторов `case`-**ветвей** для организации списка выражений одного варианта:

```
/// <summary>
/// Разбор случаев с использованием списков выражений
```

```
/// </summary>
/// <param name="operation">операция над аргументами</param>
/// <param name="arg1">первый аргумент бинарной операции</param>
/// <param name="arg2">второй аргумент бинарной операции</param>
/// <param name="result">результат бинарной операции</param>
public void ExprResult(string operation,int arg1, int arg2,
   ref int result)
{
   switch (operation)
   {
      case "+":
      case "Plus":
      case "Плюс":
         result = arg1 + arg2;
         break;
      case "-":
      case "Minus":
      case "Минус":
         result = arg1 - arg2;
         break;
      case "*":
      case "Mult":
      case "Умножить":
         result = arg1 * arg2;
         break;
      case "/":
      case "Divide":
      case "Div":
      case "разделить":
      case "Делить":
         result = arg1 / arg2;
         break;
      default:
         result = 0;
         Console.WriteLine("Операция не определена");
         break;
   }
   Console.WriteLine ("{0} ({1}, {2}) = {3}",
      operation, arg1, arg2, result);
}//ExprResult
```

Операторы перехода

Операторов перехода, позволяющих прервать естественный порядок выполнения операторов *блока*, в языке C# несколько.

Оператор goto

Оператор goto имеет простой синтаксис и семантику:

```
goto [метка|case константное_выражение|default];
```

Все операторы языка C# могут иметь метку - уникальный идентификатор, предшествующий оператору и отделенный от него символом двоеточия. Передача управления помеченному оператору - это классическое использование *оператора goto*. Два других способа использования *goto* (передача управления в case или `default`-**ветвь**) используются в *операторе switch*, о чем шла речь выше.

"О вреде *оператора goto*" и о том, как можно обойтись без него, писал еще Эдгар Дейкстра при обосновании принципов структурного программирования.

Я уже многие годы не применяю этот оператор и считаю, что хороший стиль программирования не предполагает использования этого оператора в C# ни в каком из вариантов - ни в *операторе switch*, ни для организации безусловных переходов.

Операторы break и continue

В структурном программировании признаются полезными "переходы вперед" (но не назад), позволяющие при выполнении некоторого условия выйти из цикла, из *оператора выбора*, из *блока*. Для этой цели можно использовать *оператор goto*, но лучше применять специально предназначенные для этих целей операторы `break` и `continue`.

Оператор break может стоять в теле цикла или завершать `case`-**ветвь** в *операторе switch*. Пример его использования в *операторе switch* уже демонстрировался. При выполнении оператора `break` в теле цикла завершается выполнение самого внутреннего цикла. В теле цикла, чаще всего, оператор `break` помещается в одну из ветвей *оператора if*, проверяющего условие преждевременного завершения цикла:

```
public void Jumps()
{
   int i = 1, j=1;
   for(i =1; i<100; i++)
   {
      for(j = 1; j<10;j++)
      {
         if (j>=3) break;
      }
      Console.WriteLine("Выход из цикла j при j = {0}", j);
      if (i>=3) break;
   }
   Console.WriteLine("Выход из цикла i при i= {0}", i);
}//Jumps
```

Оператор `continue` используется только в теле цикла. В отличие от оператора `break`, завершающего внутренний цикл, `continue` осуществляет переход к следующей итерации этого цикла.

Оператор return

Еще одним оператором, относящимся к группе *операторов перехода*, является оператор `return`, позволяющий завершить выполнение процедуры или функции. Его синтаксис:

```
return [выражение];
```

Для функций его присутствие и аргумент обязательны, поскольку выражение в операторе `return` задает значение, возвращаемое функцией.

Операторы цикла

Без циклов жить нельзя в программах, нет.

Оператор for

Наследованный от C++ весьма удобный *оператор цикла for* обобщает известную конструкцию цикла типа арифметической прогрессии. Его синтаксис:

```
for(инициализаторы; условие; список_выражений) оператор
```

Оператор, стоящий после закрывающей скобки, задает тело цикла. В большинстве случаев телом цикла является *блок*. Сколько раз будет выполняться тело цикла, зависит от трех управляющих элементов, заданных в скобках. **Инициализаторы** задают начальное значение одной или нескольких переменных, часто называемых счетчиками или просто переменными цикла. В большинстве случаев цикл *for* имеет один счетчик, но часто полезно иметь несколько счетчиков, что и будет продемонстрировано в следующем примере. `Условие` задает условие окончания цикла, соответствующее выражение при вычислении должно получать значение `true` или `false`. **Список выражений**, записанный через запятую, показывает, как меняются счетчики цикла на каждом шаге выполнения. Если условие цикла истинно, то выполняется тело цикла, затем изменяются значения счетчиков и снова проверяется условие. Как только условие становится ложным, цикл завершает свою работу. В цикле *for* тело цикла может ни разу не выполняться, если условие цикла ложно после инициализации, а может происходить зацикливание, если условие всегда остается истинным. В нормальной ситуации тело цикла выполняется конечное число раз.

Счетчики цикла зачастую объявляются непосредственно в инициализаторе и соответственно являются переменными, локализованными в цикле, так что после завершения цикла они перестают существовать.

В тех случаях, когда предусматривается возможность преждевременного завершения цикла с помощью одного из *операторов перехода*, счетчики объявляются до цикла, что позволяет анализировать их значения при выходе из цикла.

В качестве примера рассмотрим классическую задачу: является ли строка текста палиндромом. Напомню, палиндромом называется симметричная строка текста, читающаяся одинаково слева направо и справа налево. Для ее решения цикл *for* подходит наилучшим образом: здесь используются два счетчика - один возрастающий, другой убывающий. Вот текст соответствующей процедуры:

```
/// <summary>
/// Определение палиндромов.Демонстрация цикла for
/// </summary>
/// <param name="str">текст</param>
/// <returns>true - если текст является палиндромом</returns>
public bool Palindrom(string str)
{
   for (int i =0,j =str.Length-1; i<j; i++,j--)
      if(str[i]!=str[j]) return(false);
   return(true);
}//Palindrom
```

Циклы While

Цикл while (выражение) является универсальным видом цикла, включаемым во все языки программирования. Тело цикла выполняется до тех пор, пока остается истинным выражение `while`. В языке C# у этого вида цикла две модификации - с проверкой условия в начале и в конце цикла. Первая модификация имеет следующий синтаксис:

```
while(выражение) оператор
```

Эта модификация соответствует стратегии: "сначала проверь, а потом делай". В результате проверки может оказаться, что и делать ничего не нужно. Тело такого цикла может ни разу не выполняться. Конечно же, возможно и зацикливание. В нормальной ситуации каждое выполнение тела цикла - это очередной шаг к завершению цикла.

Цикл, проверяющий условие завершения в конце, соответствует стратегии: "сначала делай, а потом проверь". Тело такого цикла выполняется, по меньшей мере, один раз. Вот синтаксис этой модификации:

```
do
   оператор
while(выражение);
```

Приведу пример, в котором участвуют обе модификации *цикла while*. Во внешнем цикле проверка выполняется в конце, во внутреннем - в начале. Внешний цикл представляет собой типичный образец организации учебных программ, когда в диалоге с пользователем многократно решается некоторая задача. На каждом шаге пользователь вводит новые данные, решает задачу и анализирует полученные данные. В его власти, продолжить вычисления или нет, но хотя бы один раз решить задачу ему приходится. Внутренний цикл `do while` используется для решения уже известной задачи с палиндромами. Вот текст соответствующей процедуры:

```
/// <summary>
/// Два цикла: с проверкой в конце и в начале.
/// Внешний цикл - образец многократно решаемой задачи.
/// Завершение цикла определяется в диалоге
/// с пользователем.
/// </summary>
public void Loop()
{
   string answer, text;
   do
   {
      Console.WriteLine("Введите слово");
      text = Console.ReadLine();
      int i =0, j = text.Length-1;
      while ((i<j) && (text[i] == text[j]))
         {i++; j--;}
      if (text[i] == text[j])
         Console.WriteLine(text +" - это палиндром!");
      else
         Console.WriteLine(text +" - это не палиндром!");
      Console.WriteLine("Продолжим? (yes/no)");
      answer = Console.ReadLine();
   }
   while(answer =="yes");
}//Loop
```

Цикл foreach

Новым видом цикла, не унаследованным от C++, является *цикл foreach*, удобный при работе с массивами, коллекциями и другими подобными контейнерами данных. Его синтаксис:

```
foreach(тип идентификатор in контейнер) оператор
```

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

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

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

```
/// <summary>
/// Демонстрация цикла foreach. Вычисление суммы,
/// максимального и минимального элементов
/// трехмерного массива, заполненного случайными числами.
/// </summary>
public void SumMinMax()
{
   int [,,] arr3d = new int[10,10,10];
   Random rnd = new Random();
   for (int i =0; i<10; i++)
      for (int j =0; j<10; j++)
         for (int k =0; k<10; k++)
            arr3d[i,j,k]= rnd.Next(100);
   long sum =0; int min=arr3d[0,0,0], max=arr3d[0,0,0];
   foreach(int item in arr3d)
   {
      sum +=item;
      if (item > max) max = item;
      else if (item < min) min = item;
   }
   Console.WriteLine("sum = {0}, min = {1}, max = {2}",
      sum, min, max);
}//SumMinMax
```

9. Лекция: Процедуры и функции - методы класса

Процедуры и функции - функциональные модули

Первыми формами модульности, появившимися в языках программирования, были *процедуры* и *функции*. Они позволяли задавать определенную функциональность и многократно выполнять один и тот же параметризованный программный код при различных значениях параметров. Поскольку *функции* в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов *процедуры* и *функции* позволяли решать одну из важнейших задач, стоящих перед программистами, - задачу повторного использования программного кода. Встроенные в язык *функции* давали возможность существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек *процедур* и *функций*, доступных из используемого языка.

Процедуры и функции - методы класса

Долгое время *процедуры* и *функции* играли не только функциональную, но и архитектурную роль. Весьма популярным при построении программных систем был метод функциональной декомпозиции "сверху вниз", и сегодня еще играющий важную роль. Но с появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ОО-языков, к которым относится и язык C#, в роли архитектурного модуля выступает *класс*. Программная система строится из модулей, роль которых играют *классы*, но каждый из этих модулей имеют содержательную начинку, задавая некоторую абстракцию данных.

Процедуры и *функции* связываются теперь с *классом*, они обеспечивают функциональность данных *класса* и называются методами *класса*. Главную роль в программной системе играют данные, а *функции* лишь служат данным. Напомню здесь, что в C# *процедуры* и *функции* существуют только как методы некоторого *класса*, они не существуют вне *класса*.

В данном контексте понятие *класс* распространяется и на все его частные случаи - структуры, интерфейсы, делегаты.

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

Прежнюю роль библиотек *процедур* и *функций* теперь играют *библиотеки классов*. *Библиотека классов* FCL, доступная в языке C#, существенно расширяет возможности языка. Знание *классов* этой *библиотеки* и методов этих *классов* совершенно необходимо для практического программирования на C# с использованием всей его мощи.

Уже в лекции 1 мы говорили о роли *библиотеки FCL* - статическом компоненте Framework .Net. В лекции 4 рассматривались возможности *класса* `Convert` этой *библиотеки*, а в лекции 7 - *классы* `Math` и `Random`. Изучение *классов* FCL будет постоянно сопровождать наш курс.

Процедуры и функции. Отличия

Функция отличается от *процедуры* двумя особенностями:

- всегда вычисляет некоторое значение, возвращаемое в качестве результата *функции*;
- вызывается в выражениях.

Процедура C# имеет свои особенности:

- возвращает формальный результат `void`, указывающий на *отсутствие результата*;
- вызов *процедуры* является оператором языка;
- имеет *входные* и *выходные аргументы*, причем *выходных аргументов* - ее результатов - может быть достаточно много.

Хорошо известно, что одновременное существование в языке *процедур* и *функций* в каком-то смысле избыточно. Добавив еще один *выходной аргумент*, любую *функцию* можно записать в виде *процедуры*. Справедливо и обратное. Если допускать *функции с побочным эффектом*, то любую *процедуру* можно записать в виде *функции*. В языке C - дедушке C# - так и сделали, оставив только *функции*. Однако значительно удобнее иметь обе формы реализации метода: и *процедуры*, и *функции*. Обычно метод предпочитают реализовать в виде *функции* тогда, когда он имеет один *выходной аргумент*, рассматриваемый как результат вычисления значения *функции*. Возможность вызова *функций* в выражениях также влияет на выбор в пользу реализации метода в виде *функции*. В других случаях метод реализуют в виде *процедуры*.

Описание методов (процедур и функций). Синтаксис

Синтаксически в описании метода различают две части - *описание заголовка* и *описание тела* метода:

```
заголовок_метода
тело_метода
```

Рассмотрим синтаксис заголовка метода:

```
[атрибуты][модификаторы]{void| тип_результата_функции}
имя_метода([список_формальных_аргументов])
```

Имя метода и список формальных аргументов составляют *сигнатуру метода*. Заметьте, в *сигнатуру* не входят имена формальных аргументов - здесь важны типы аргументов. В *сигнатуру* не входит и тип возвращаемого результата.

Квадратные скобки (метасимволы синтаксической формулы) показывают, что атрибуты и модификаторы могут быть опущены при описании метода. Подробное их рассмотрение будет дано в лекциях, посвященных описанию *классов*. Сейчас же упомяну только об одном из модификаторов - модификаторе доступа. У него четыре возможных значения, из которых пока рассмотрим только два - `public` и `private`. Модификатор `public` показывает, что ***метод открыт*** и доступен для вызова клиентами и потомками *класса*. Модификатор `private` говорит, что метод предназначен для внутреннего использования в *классе* и доступен для вызова только в теле методов самого *класса*. Заметьте, если модификатор доступа опущен, то по умолчанию предполагается, что он имеет значение `private` и метод является ***закрытым*** для клиентов и потомков *класса*.

Обязательным при *описании заголовка* является указание типа результата, имени метода и круглых скобок, наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует. Формально тип результата метода указывается всегда, но значение `void` однозначно определяет, что метод реализуется *процедурой*.

Тип результата, отличный от `void`, указывает на *функцию*. Вот несколько простейших примеров описания методов:

```
void A() {...};
int B(){...);
public void C(){...};
```

Методы `A` и `B` являются *закрытыми*, а метод `C` - *открыт*. Методы `A` и `C` реализованы *процедурами*, а метод `B` - *функцией*, возвращающей целое значение.

Список формальных аргументов

Как уже отмечалось, список формальных аргументов метода может быть пустым, и это довольно типичная ситуация для методов *класса*. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Рассмотрим теперь синтаксис объявления формального аргумента:

```
[ref|out|params]тип_аргумента имя_аргумента
```

Обязательным является указание типа и имени аргумента. Заметьте, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, *классом*, структурой, интерфейсом, перечислением, функциональным типом.

Несмотря на фиксированное число формальных аргументов, есть возможность при *вызове метода* передавать ему *произвольное число фактических аргументов*. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово *params*. Оно задается один раз и указывается только для последнего аргумента списка, объявляемого как массив произвольного типа. При *вызове метода* этому формальному аргументу соответствует *произвольное число фактических аргументов*.

Содержательно, все аргументы метода разделяются на три группы: *входные*, *выходные* и **обновляемые**. Аргументы первой группы передают информацию методу, их значения в теле метода только читаются. Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода. Аргументы третьей группы выполняют обе *функции*. Их значения используются в ходе вычислений и обновляются в результате работы метода. *Выходные аргументы* всегда должны сопровождаться ключевым словом *out*, обновляемые - *ref*. Что же касается *входных аргументов*, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром *ref*, о чем подробнее скажу чуть позже. Заметьте, если аргумент объявлен как *выходной* с ключевым словом *out*, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

Для иллюстрации давайте рассмотрим группу методов *класса* `Testing` из проекта `ProcAndFun`, сопровождающего эту лекцию:

```
/// <summary>
/// Группа перегруженных методов A()
/// первый аргумент представляет сумму кубов
/// произвольного числа оставшихся аргументов
/// Аргументы могут быть разного типа.
/// </summary>
void A(out long p2, int p1)
{
      p2 =(long) Math.Pow(p1,3);
```

```
   Console.WriteLine("Метод A-1");
}
void A(out long p2, params int[] p)
{
   p2=0; for(int i=0; i <p.Length; i++) p2 += (long)Math.Pow(p[i],3);
   Console.WriteLine("Метод A-2");
}
void A(out double p2, double p1)
{
   p2 = Math.Pow(p1,3);
   Console.WriteLine("Метод A-3");
}
void A(out double p2, params double[] p)
{
   p2=0; for(int i=0; i <p.Length; i++) p2 += Math.Pow(p[i],3);
   Console.WriteLine("Метод A-4");
}
/// <summary>
/// Функция с побочным эффектом
/// </summary>
/// <param name="a">Увеличивается на 1</param>
/// <returns>значение a на входе</returns>
int f(ref int a)
{
   return(a++);
}
```

Четыре перегруженных метода с именем `A` и метод `f` будут использоваться при объяснении *перегрузки* и побочного эффекта. Сейчас проанализируем только их заголовки. Все *методы закрыты*, поскольку объявлены без модификатора доступа. Перегруженные методы с именем `A` являются *процедурами*, метод `f` - *функцией*. Все четыре перегруженных метода имеют разную сигнатуру. Хотя имена и число аргументов у всех методов одинаковы, но типы и ключевые слова, предшествующие аргументам, различны. Первый аргумент у всех четырех перегруженных методов - *выходной* и сопровождается ключевым словом *out*, в теле метода этому аргументу присваивается значение. Аргумент *функции* `f` - обновляемый, он снабжен ключевым словом *ref*, в теле *функции* используется его значение для получения результата *функции*, но и само значение аргумента изменяется в теле *функции*. Два метода из группы перегруженных методов используют ключевое слово *params* для своего последнего аргумента. Позже мы увидим, что при *вызове этих методов* указанному аргументу будет соответствовать несколько фактических аргументов, число которых может быть произвольным.

Тело метода

Синтаксически тело метода является *блоком*, который представляет собой последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле *функции*, то в *блоке* должен быть хотя бы один оператор перехода, возвращающий значение *функции* в форме *return (выражение)*.

```
Оператор return описан в лекции 8.
```

Переменные, описанные в *блоке*, считаются локализованными в этом *блоке*. В записи операторов *блока* участвуют имена локальных переменных *блока*, имена полей класса и имена аргументов метода.

Область видимости, время жизни переменных, конфликты имен рассмотрены в лекции 5, семантика операторов - в лекции 8. Дополнительные сведения о семантике выполнения метода будут даны в этой лекции.

Знания семантики описаний и операторов достаточно для понимания семантики *блока*. Необходимые уточнения будут даны чуть позже.

Вызов метода. Синтаксис

Как уже отмечалось, метод может вызываться в выражениях или быть вызван как оператор. В качестве оператора может использоваться любой метод - как *процедура*, так и *функция*. Конечно, *функцию* разумно вызывать как оператор, только если она обладает побочным эффектом. В последнем случае она вызывается ради своего побочного эффекта, а возвращаемое значение никак не используется. Подобную роль играет использование некоторых выражений с побочным эффектом в роли оператора, классическим примером является оператор `x++`;.

Если же попытаться вызвать *процедуру* в выражении, то это приведет к ошибке еще на этапе компиляции. Возвращаемое *процедурой* значение `void` несовместимо с выражениями. Так что в выражениях могут быть вызваны только *функции*.

Сам *вызов метода*, независимо от того, *процедура* это или *функция*, имеет один и тот же синтаксис:

```
имя_метода([список_фактических_аргументов])
```

Если это оператор, то вызов завершается точкой с запятой. Формальный аргумент, задаваемый при описании метода - это всегда имя аргумента (идентификатор). Фактический аргумент - это выражение, значительно более сложная синтаксическая конструкция. Вот точный синтаксис фактического аргумента:

```
[ref|out]выражение
```

О соответствии списков формальных и фактических аргументов

Между списком формальных и списком фактических аргументов должно выполняться определенное *соответствие* по числу, порядку следования, типу и статусу аргументов. Если в первом списке `n` формальных аргументов, то фактических аргументов должно быть не меньше `n` (*соответствие* по числу). Каждому `i`-му формальному аргументу (для всех `i` от `1` до `n-1`) ставится в *соответствие* `i`-й фактический аргумент. Последнему формальному аргументу, при условии, что он объявлен с ключевым словом *params*, ставятся в *соответствие* все оставшиеся фактические аргументы (*соответствие* по порядку). Если формальный аргумент объявлен с ключевым словом *ref* или *out*, то фактический аргумент должен сопровождаться таким же ключевым словом в точке вызова (*соответствие* по статусу).

Появление ключевых слов при *вызове методов* - это особенность языка C#, отличающая его от большинства других языков. Такой синтаксис следует приветствовать, поскольку он направлен на повышение надежности программной системы, напоминая программисту о том, что данный фактический аргумент является *выходным* и значение его наверняка изменится после *вызова метода*. Однако из-за непривычности синтаксиса при *вызове методов* эти слова часто забывают писать, что приводит к появлению синтаксических ошибок.

Если формальный аргумент объявлен с типом `T`, то выражение, задающее фактический аргумент, должно быть согласовано по типу с типом `T`: допускает преобразование к типу `T`, совпадает с типом `T` или является его потомком (*соответствие* по типу).

Если формальный аргумент является *выходным* - объявлен с ключевым словом *ref* или *out*, - то соответствующий фактический аргумент не может быть выражением,

поскольку используется в левой части оператора присваивания; следовательно, он должен быть именем, которому можно присвоить значение.

Вызов метода. Семантика

Что происходит в момент *вызова метода*? Выполнение начинается с вычисления фактических аргументов, которые, как мы знаем, являются выражениями. Вычисление этих выражений может приводить, в свою очередь, к вызову других методов, так что этот первый этап может быть довольно сложным и требовать больших временных затрат. В чисто функциональном программировании все вычисление по программе сводится к вызову одной *функции*, фактическими аргументами которой являются вызовы *функций* и так далее и так далее.

Для простоты понимания семантики вызова можно полагать, что в точке вызова создается *блок*, соответствующий телу метода (в реальности все значительно эффективнее). В этом *блоке* происходит замена имен формальных аргументов фактическими аргументами. Для *выходных аргументов*, для которых фактические аргументы также являются именами, эта замена или передача аргументов осуществляется `по ссылке`, то есть заменяет формальный аргумент ссылкой на реально существующий объект, заданный фактическим аргументом. Чуть более сложную семантику имеет *вызов по значению*, применяемый к формальным аргументам, которые объявлены без ключевых слов *ref* и *out*. При вычислении выражений, заданных такими фактическими аргументами, их значения присваиваются специально создаваемым переменным, локализованным в теле исполняемого *блока*. Имена этих локализованных переменных и подставляются вместо имен формальных аргументов. Понятно, что тип локализованных переменных определяется типом соответствующего формального аргумента. Понятно также, что семантика замены формальных аргументов фактическими - это, по сути, семантика оператора присваивания.

Семантика присваивания рассматривалась в лекциях 3, 6 и 7.

Каково следствие семантики *вызова по значению*? Если вы забыли указать ключевое слово *ref* или *out* для аргумента, фактически являющегося *выходным*, то к нему будет применяться *вызов по значению*. Даже если в теле метода происходит изменение значения этого аргумента, то оно действует только на время выполнения тела метода. Как только метод заканчивает свою работу (завершается *блок*), все локальные переменные (в том числе, созданные для замены формальных аргументов) оканчивают свое существование, так что изменения не затронут фактических аргументов и они сохранят свои значения, бывшие у них до вызова. Отсюда вывод: все *выходные аргументы*, значения которых предполагается изменить в процессе работы, должны иметь ключевое слово *ref* или *out*. Еще один важный вывод: ключевым словом *ref* полезно иногда снабжать и *входные аргументы*. Если известно, что фактический аргумент будет всегда представлен именем, а не сложным выражением, то в целях экономии памяти разумно для таких аргументов применять семантику *вызова по ссылке*. В этом случае не будет создаваться копия аргумента - это экономит память и время, что может быть важно при работе со сложными структурами.

Говоря о семантике *вызова по ссылке* и по значению, следует сделать одно важное уточнение. В объектном программировании, каковым является и программирование на C#, основную роль играют ссылочные типы - мы работаем с *классами* и объектами. Когда методу передается объект ссылочного типа, то все поля этого объекта могут меняться в методе самым беззастенчивым образом. И это несмотря на то, что объект формально не является выходным, не имеет ключевых слов *ref* или *out*, использует семантику *вызова по значению*. Сама ссылка на объект, как и положено, остается неизменной, но состояние объекта, его поля могут полностью обновиться. Такая ситуация типична и представляет один из основных способов изменения состояния объектов. Именно поэтому *ref* или *out* не часто появляются при описании аргументов метода.

Что нужно знать о методах?

Знания формального синтаксиса и семантики недостаточно, чтобы эффективно работать с методами. Рассмотрим сейчас несколько важных вопросов, касающихся различных сторон работы с методами *класса*.

Почему у методов мало аргументов?

Методы *класса* имеют значительно меньше аргументов, чем *процедуры* и *функции* в классическом процедурном стиле программирования, когда не используется концепция *классов*. За счет чего происходит уменьшение числа аргументов у методов? Ведь аргументы играют важную роль: они передают методу информацию, нужную ему для работы, и возвращают информацию - результаты работы метода - программе, вызвавшей его.

Все дело в том, что методы *класса* - это не просто *процедуры*; это *процедуры*, обслуживающие данные. Все поля доступны любому методу по определению. Нужно четко понимать, что в момент выполнения программной системы работа идет не с *классом*, а с объектами - экземплярами *класса*. Из полей соответствующего объекта - цели вызова - извлекается информация, нужная методу в момент вызова, а работа метода чаще всего сводится к обновлению значений полей этого объекта. Поэтому очевидно, что методу не нужно через *входные аргументы* передавать информацию, содержащуюся в полях. Если в результате работы метода обновляется значение некоторого поля, то, опять-таки, не нужен никакой *выходной аргумент*.

Поля класса или функции без аргументов?

Поля хранят информацию о состоянии объектов *класса*. Состояние объекта динамически изменяется в ходе вычислений - обновляются значения полей. Часто возникающая дилемма при проектировании *класса*: что лучше - создать ли поле, хранящее информацию, или создать *функцию* без аргументов, вычисляющую значение этого поля всякий раз, когда это значение понадобится. Решение дилеммы - это вечный для программистов выбор между памятью и временем. Если предпочесть поле, то это приводит к дополнительным расходам памяти. Они могут быть значительными, когда создается большое число объектов - ведь свое поле должен иметь каждый объект. Если предпочесть *функцию*, то это потребует временных затрат на вычисление значения, и затраты могут быть значительными в сравнении с выбором текущего значения поля.

Если бы синтаксис описания метода допускал отсутствие скобок у *функции* (метода), в случае, когда список аргументов отсутствует, то клиент *класса* мог бы и не знать, обращается ли он к полю или к методу. Такой синтаксис принят, например, в языке Eiffel. Преимущество этого подхода в том, что изменение реализации никак не сказывается на клиентах *класса*. В языке C# это не так. Когда мы хотим получить длину строки, то пишем `s.Length`, точно зная, что `Length` - это поле, а не метод *класса* `String`. Если бы по каким-либо причинам разработчики *класса* `String` решили изменить реализацию и заменить поле `Length` соответствующей *функцией*, то ее вызов имел бы вид `s.Length()`.

Пример: две версии класса Account

Проиллюстрируем рассмотренные выше вопросы на примере проектирования *классов* `Account` и `Account1`, описывающих такую абстракцию данных, как банковский счет. Определим на этих данных две основные операции - занесение денег на счет и снятие денег. В первом варианте - *классе* `Account` - будем активно использовать поля *класса*. Помимо двух основных полей `credit` и `debit`, хранящих приход и расход счета, введем поле `balance`, которое задает текущее состояние счета, и два поля, связанных с последней выполняемой операцией. Поле `sum` будет хранить сумму денег текущей

операции, а поле `result` - результат выполнения операции. Полей у *класса* много, и как следствие, у методов *класса* аргументов будет немного. Вот описание нашего *класса*:

```
/// <summary>
/// Класс Account определяет банковский счет. Простейший
/// вариант с возможностью трех операций: положить деньги
/// на счет, снять со счета, узнать баланс.Вариант с полями
/// </summary>
public class Account
{
   //закрытые поля класса
   int debit=0, credit=0, balance =0;
   int sum =0, result=0;
   /// <summary>
   /// Зачисление на счет с проверкой
   /// </summary>
   /// <param name="sum">зачисляемая сумма</param>
   public void putMoney(int sum)
   {
      this.sum = sum;
      if (sum >0)
      {
         credit += sum; balance = credit - debit; result =1;
      }
      else result = -1;
      Mes();
   }//putMoney
   /// <summary>
   /// Снятие со счета с проверкой
   /// </summary>
   /// <param name="sum"> снимаемая сумма</param>
   public void getMoney(int sum)
   {
      this.sum = sum;
      if(sum <= balance)
      {
         debit += sum; balance = credit - debit; result =2;
      }
      else result = -2;
      Mes();
   }//getMoney
   /// <summary>
   /// Уведомление о выполнении операции
   /// </summary>
   void Mes()
   {
      switch (result)
      {
         case 1:
            Console.WriteLine("Операция зачисления денег
               прошла успешно!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}",sum, balance);
      break;
         case 2:
            Console.WriteLine("Операция снятия денег
               прошла успешно!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}", sum,balance);
            break;
         case -1:
            Console.WriteLine("Операция зачисления денег
```

```
                не выполнена!");
             Console.WriteLine("Сумма должна быть больше нуля!");
             Console.WriteLine("Сумма={0},
                Ваш текущий баланс={1}", sum,balance);
             break;
          case -2:
             Console.WriteLine("Операция снятия денег
                не выполнена!");
             Console.WriteLine("Сумма должна быть
                не больше баланса!");
             Console.WriteLine("Сумма={0},
                Ваш текущий баланс={1}", sum,balance);
             break;
          default:
             Console.WriteLine("Неизвестная операция!");
             break;
       }
    }
}//Account
```

Как можно видеть, только у методов `getMoney` и `putMoney` имеется один *входной аргумент*. Это тот аргумент, который нужен по сути дела, поскольку только клиент может решить, какую сумму он хочет снять или положить на счет. Других аргументов у методов *класса* нет - вся информация передается через поля *класса*. Уменьшение числа аргументов приводит к повышению эффективности работы с методами, так как исчезают затраты на передачу фактических аргументов. Но за все надо платить. В данном случае, усложняются сами операции работы со вкладом, поскольку нужно в момент выполнения операции обновлять значения многих полей *класса*. *Закрытый метод* `Mes` вызывается после выполнения каждой операции, сообщая о том, как прошла операция, и информируя клиента о текущем состоянии его баланса.

А теперь спроектируем аналогичный *класс* `Account1`, отличающийся только тем, что у него будет меньше полей. Вместо поля `balance` в *классе* появится соответствующая *функция* с этим же именем, вместо полей `sum` и `result` появятся аргументы у методов, обеспечивающие необходимую передачу информации. Вот как выглядит этот *класс*:

```
/// <summary>
/// Класс Account1 определяет банковский счет.
/// Вариант с аргументами и функциями
/// </summary>
public class Account1
{
   //закрытые поля класса
   int debit=0, credit=0;
   /// <summary>
   /// Зачисление на счет с проверкой
   /// </summary>
   /// <param name="sum">зачисляемая сумма</param>
   public void putMoney(int sum)
   {
      int res =1;
      if (sum >0)credit += sum;
      else res = -1;
      Mes(res,sum);
   }//putMoney
   /// <summary>
   /// Снятие со счета с проверкой
   /// </summary>
   /// <param name="sum"> снимаемая сумма</param>
   public void getMoney(int sum)
   {
      int res=2;
```

```
      if(sum <= balance())debit += sum;
      else res = -2;
      balance();
      Mes(res, sum);
   }//getMoney
/// <summary>
/// вычисление баланса
/// </summary>
/// <returns>текущий баланс</returns>
   int balance()
   {
      return(credit - debit);
   }
   /// <summary>
   /// Уведомление о выполнении операции
   /// </summary>
   void Mes(int result, int sum)
   {
      switch (result)
      {
         case 1:
            Console.WriteLine("Операция зачисления денег прошла успешно!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}", sum,balance());
            break;
         case 2:
            Console.WriteLine("Операция снятия денег
               прошла успешно!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}", sum,balance());
            break;
         case -1:
            Console.WriteLine("Операция зачисления денег
               не выполнена!");
            Console.WriteLine("Сумма должна быть больше нуля!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}", sum,balance());
            break;
         case -2:
            Console.WriteLine("Операция снятия денег не выполнена!");
            Console.WriteLine("Сумма должна быть
               не больше баланса!");
            Console.WriteLine("Сумма={0},
               Ваш текущий баланс={1}", sum,balance());
            break;
         default:
            Console.WriteLine("Неизвестная операция!");
            break;
      }
   }
}//Account1
```

Сравнивая этот *класс* с *классом* `Account`, можно видеть, что число полей сократилось с пяти до двух, упростились основные методы `getMoney` и `putMoney`. Но, в качестве платы, у *класса* появился дополнительный метод `balance()`, многократно вызываемый, и у метода `Mes` теперь появились два аргумента. Какой *класс* лучше? Однозначно сказать нельзя, все зависит от контекста, от приоритетов, заданных при создании конкретной системы.

Приведу *процедуру класса* `Testing`, тестирующую работу с *классами* `Account` и `Account1`:

```
public void TestAccounts()
{
   Account myAccount = new Account();
   myAccount.putMoney(6000);
   myAccount.getMoney(2500);
   myAccount.putMoney(1000);
   myAccount.getMoney(4000);
   myAccount.getMoney(1000);
   //Аналогичная работа с классом Account1
   Console.WriteLine("Новый класс и новый счет!");
   Account1 myAccount1 = new Account1();
   myAccount1.putMoney(6000);
   myAccount1.getMoney(2500);
   myAccount1.putMoney(1000);
   myAccount1.getMoney(4000);
   myAccount1.getMoney(1000);
}
```

На рис. 9.1 показаны результаты работы этой *процедуры*.

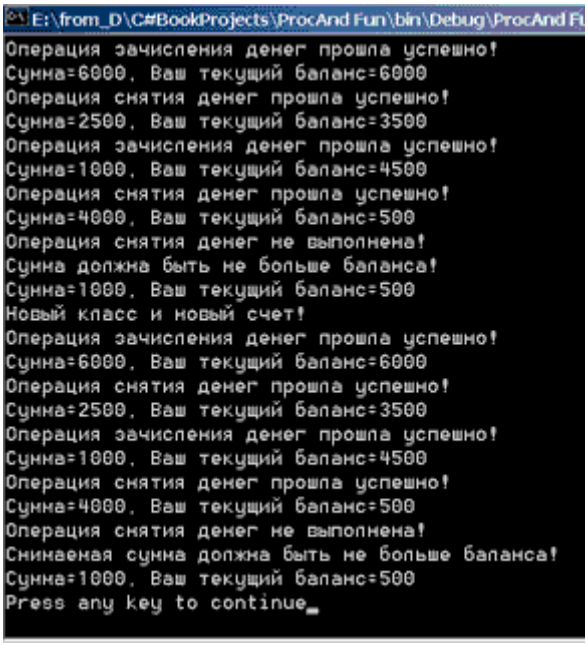

Рис. 9.1. Тестирование классов Account и Account1

Функции с побочным эффектом

Функция называется ***функцией с побочным эффектом***, если помимо результата, вычисляемого *функцией* и возвращаемого ей в операторе `return`, она имеет *выходные аргументы* с ключевыми словами *ref* и *out*. В языках C/C++ *функции с побочным эффектом* применяются сплошь и рядом. Хороший стиль ОО-программирования не рекомендует использование таких *функций*. Выражения, использующие *функции с побочным эффектом*, могут потерять свои прекрасные свойства, присущие им в математике. Если `f(a)` - *функция с побочным эффектом*, то `a+f(a)` может быть не равно `f(a) +a`, так что теряется коммутативность операции сложения.

Примером такой *функции* является *функция* `f`, приведенная выше. Вот тест, демонстрирующий потерю коммутативности сложения при работе с этой *функцией*:

```
/// <summary>
/// тестирование побочного эффекта
/// </summary>
public void TestSideEffect()
```

```
{
   int a = 0, b=0, c=0;
   a =1; b = a + f(ref a);
   a =1; c =  f(ref a)+ a;
   Console.WriteLine("a={0}, b={1}, c={2}",a,b,c);
}
```

На рис. 9.2 показаны результаты работы этого метода.

Рис. 9.2. Демонстрация вызова функции с побочным эффектом

Обратите внимание на полезность указания ключевого слова *ref* в момент вызова. Его появление хоть как-то оправдывает некоммутативность сложения.

Методы. Перегрузка

Должно ли быть уникальным имя метода в *классе*? Нет, этого не требуется. Более того, проектирование методов с одним и тем же именем является частью стиля программирования на C++ и стиля C#. Существование в *классе* методов с одним и тем же именем называется *перегрузкой*, а сами одноименные методы называются **перегруженными**.

Перегрузка методов полезна, когда требуется решать подобные задачи с разным набором аргументов. Типичный пример - это нахождение площади треугольника. Площадь можно вычислить по трем сторонам, по двум углам и стороне, по двум сторонам и углу между ними и при многих других наборах аргументов. Считается удобным во всех случаях иметь для метода одно имя, например `Square`, и всегда, когда нужно вычислить площадь, не задумываясь, вызывать метод `Square`, передавая ему известные в данный момент аргументы.

Перегрузка характерна и для знаков операций. В зависимости от типов аргументов, один и тот же знак может выполнять фактически разные операции. Классическим примером является знак операции сложения `+`, который играет роль операции сложения не только для арифметических данных разных типов, но и выполняет конкатенацию строк.

О *перегрузке* операций при определении *класса* будет подробно сказано в лекции, посвященной *классам*.

Перегрузка требует уточнения семантики *вызова метода*. Когда встречается вызов неперегруженного метода, то имя метода в вызове однозначно определяет, тело какого метода должно выполняться в точке вызова. Когда же метод перегружен, то знания имени недостаточно - оно не уникально. Уникальной характеристикой перегруженных методов является их сигнатура. Перегруженные методы, имея одинаковое имя, должны отличаться либо числом аргументов, либо их типами, либо ключевыми словами (заметьте: с точки зрения сигнатуры, ключевые слова *ref* и *out* не отличаются). Уникальность сигнатуры позволяет вызвать требуемый перегруженный метод.

Выше уже были приведены четыре перегруженных метода с именем `A`, различающиеся по сигнатуре. Эти методы отличаются типами аргументов и ключевым словом *params*. Когда вызывается метод `A` с двумя аргументами, то, в зависимости от типа, будет вызываться реализация без ключевого *params*. Когда же число аргументов больше двух, то работает реализация, позволяющая справиться с заранее не фиксированным числом аргументов. Заметьте, эта реализация может прекрасно работать и для случая двух аргументов, но полезно иметь частные случаи для фиксированного набора

аргументов. При поиске подходящего перегруженного метода частные случаи получают предпочтение в сравнении с общим случаем.

Тема поиска подходящего перегруженного метода уже рассматривалась в лекции 3, где шла речь о преобразованиях арифметического типа. Стоит вернуться к примеру, который был рассмотрен в этом разделе и демонстрировал возможность возникновения конфликта: один фактический аргумент требует выбора некоей реализации, для другого - предпочтительнее реализация иная. Для устранения таких конфликтов требуется вмешательство программиста.

Насколько полезна *перегрузка* методов? Здесь нет экономии кода, поскольку каждую реализацию нужно задавать явно; нет выигрыша по времени - напротив, требуются определенные затраты на поиск подходящей реализации, который может приводить к конфликтам, - к счастью, обнаруживаемым на этапе компиляции. В нашем примере вполне разумно иметь четыре метода с разными именами и осознанно вызывать метод, применимый к данным аргументам. Все-таки есть ситуации, где *перегрузка* полезна, недаром она широко используется при построении *библиотеки* FCL. Возьмем, например, *класс* `Convert`, у которого 16 методов с разными именами, зависящими от целевого типа преобразования. Каждый из этих 16 методов перегружен, и в свою очередь, имеет 16 реализаций в зависимости от типа источника. Согласитесь, что неразумно было бы иметь в *классе* `Convert` 256 методов вместо 16-ти перегруженных методов. Впрочем, также неразумно было бы пользоваться одним перегруженным методом, имеющим 256 реализаций. *Перегрузка* - это инструмент, который следует использовать с осторожностью и обоснованно.

В заключение этой темы посмотрим, как проводилось тестирование работы с перегруженными методами:

```
/// <summary>
/// Тестирование перегруженных методов A()
/// </summary>
public void TestLoadMethods()
{
   long u=0; double v =0;
   A(out u, 7); A(out v, 7.5);
   Console.WriteLine ("u= {0}, v= {1}", u,v);
   A(out v,7);
   Console.WriteLine("v= {0}",v);
   A(out u, 7,11,13);
   A(out v, 7.5, Math.Sin(11.5)+Math.Cos(13.5), 15.5);
   Console.WriteLine ("u= {0}, v= {1}", u,v);
}//TestLoadMethods
```

На рис. 9.3 показаны результаты этого тестирования.

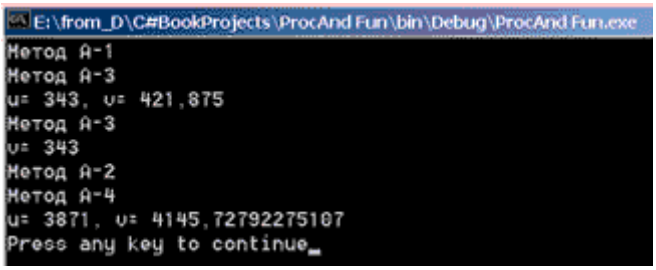

Рис. 9.3. Тестирование перегрузки методов

10. Лекция: Корректность методов. Рекурсия

Корректность методов

Написать метод, задающий ту или иную функциональность, нетрудно. Это может сделать каждый. Значительно сложнее написать метод, корректно решающий поставленную задачу. *Корректность* метода - это не внутреннее понятие, подлежащее определению в терминах самого метода. *Корректность* определяется по отношению к внешним спецификациям метода. Если нет спецификаций, то говорить о *корректности* "некорректно".

Спецификации можно задавать по-разному. Мы определим их здесь через понятия предусловий и постусловий метода, используя символику триад Хоара, введенных Чарльзом Энтони Хоаром - выдающимся программистом и ученым, одну из знаменитых программ которого приведем чуть позже в этой лекции.

Пусть `P(x,z)` - программа `P` с входными аргументами `x` и выходными `z`. Пусть `Q(y)` - некоторое логическое условие (предикат) над переменными программы `y`. Язык для записи предикатов `Q(y)` формализовать не будем. Отметим только, что он может быть шире языка, на котором записываются условия в программах, и включать, например, кванторы. Предусловием программы `P(x,z)` будем называть предикат `Pre(x)`, заданный на входах программы. Постусловием программы `P(x,z)` будем называть предикат `Post(x,z)`, связывающий входы и выходы программы. Для простоты будем полагать, что программа `P` не изменяет своих входов `x` в процессе своей работы. Теперь несколько определений:

Определение 1 (***частичной корректности***): Программа `P(x,z)` корректна (частично, или условно) по отношению к предусловию `Pre(x)` и постусловию `Post(x,z)`, если из истинности предиката `Pre(x)` следует, что для программы `P(x,z)`, запущенной на входе `x`, гарантируется выполнение предиката `Post(x,z)` при условии завершения программы.

Условие *частичной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

```
[Pre(x)]P(x,z)[Post(x,z)]
```

Определение 2 (***полной корректности***): Программа `P(x,z)` корректна (полностью, или тотально) по отношению к предусловию `Pre(x)` и постусловию `Post(x,z)`, если из истинности предиката `Pre(x)` следует, что для программы `P(x,z)`, запущенной на входе `x`, гарантируется ее завершение и выполнение предиката `Post(x,z)`.

Условие *полной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

```
{Pre(x)}P(x,z){Post(x,z)}
```

Доказательство полной корректности обычно состоит из двух независимых этапов - *доказательства частичной корректности* и доказательства завершаемости программы. Заметьте, *полностью корректная* программа, которая запущена на входе, не удовлетворяющем ее предусловию, вправе зацикливаться, а также возвращать любой результат. Любая программа корректна по отношению к предусловию, заданному тождественно ложным предикатом `False`. Любая завершающаяся программа корректна по отношению к постусловию, заданному тождественно истинным предикатом `True`.

Корректная программа говорит своим клиентам: если вы хотите вызвать меня и ждете гарантии выполнения постусловия после моего завершения, то будьте добры гарантировать выполнение предусловия на входе. Задание предусловий и постусловий методов - это такая же важная часть работы программиста, как и написание самого метода. На языке C# пред- и постусловия обычно задаются в теге `<summary>`, предшествующем методу, и являются частью XML-отчета. К сожалению, технология работы в Visual Studio не предусматривает возможности автоматической проверки предусловия перед вызовом метода и проверки постусловия после его завершения с выбрасыванием исключений в случае их невыполнения. Программисты, для которых требование *корректности* является важнейшим условием качества их работы, сами встраивают такую проверку в свои программы. Как правило, подобная проверка обязательна на этапе отладки и может быть отключена в готовой системе, в *корректности* которой программист уже уверен. А вот проверку предусловий важно оставлять и в готовой системе, поскольку истинность предусловий должен гарантировать не разработчик метода, а клиент, вызывающий метод. Клиентам же свойственно ошибаться и вызывать метод в неподходящих условиях.

Формальное *доказательство корректности* метода - задача ничуть не проще, чем написание корректной программы. Но вот парадокс. Чем сложнее метод, его алгоритм, а следовательно, и само *доказательство*, тем важнее использовать понятия предусловий и постусловий, понятия *инвариантов* циклов в процессе разработки метода. Рассмотрение этих понятий параллельно с разработкой метода может существенно облегчить построение корректного метода. Этот подход будет продемонстрирован в нашей лекции при рассмотрении метода QuickSort - быстрой сортировки массива.

Инварианты и варианты цикла

Циклы, как правило, являются наиболее сложной частью метода - большинство ошибок связано именно с ними. При написании корректно работающих циклов крайне важно понимать и использовать понятия *инварианта* и *варианта цикла*. Без этих понятий не обходится и формальное *доказательство корректности* циклов. Ограничимся рассмотрением цикла в следующей форме:

```
Init(x,z); while(B)S(x,z);
```

Здесь `B` - условие цикла `while`, `S` - его тело, а `Init` - группа предшествующих операторов, задающая инициализацию цикла. Реально ни один цикл не обходится без инициализирующей части. Синтаксически было бы правильно, чтобы `Init` являлся бы формальной частью оператора цикла. В операторе `for` эта частично сделано - инициализация счетчиков является частью цикла.

Определение 3 (***инварианта цикла***): предикат `Inv(x, z)` называется *инвариантом* цикла `while`, если истинна следующая триада Хоара:

```
{Inv(x, z)& B}S(x,z){Inv(x,z)}
```

Содержательно это означает, что из истинности *инварианта* цикла до начала выполнения тела цикла и из истинности условия цикла, гарантирующего выполнение тела, следует истинность *инварианта* после выполнения тела цикла. Сколько бы раз ни выполнялось тело цикла, его *инвариант* остается истинным.

Для любого цикла можно написать сколь угодно много *инвариантов*. Любое тождественное условие (`2*2 =4`) является *инвариантом* любого цикла. Поэтому среди *инвариантов* выделяются так называемые подходящие *инварианты* цикла. Они называются подходящими, поскольку позволяют доказать *корректность цикла* по

отношению к его пред- и постусловиям. Как доказать *корректность цикла*? Рассмотрим соответствующую триаду:

```
{Pre(x)} Init(x,z); while(B)S(x,z);{Post(x,z)}
```

Доказательство разбивается на три этапа. Вначале доказываем истинность триады:

```
(*)     {Pre(x)} Init(x,z){RealInv(x,z)}
```

Содержательно это означает, что предикат `RealInv` становится истинным после выполнения инициализирующей части. Далее доказывается, что `RealInv` является *инвариантом* цикла:

```
(**)    {RealInv(x, z)& B} S(x,z){RealInv(x,z)}
```

На последнем шаге доказывается, что наш *инвариант* обеспечивает решение задачи после завершения цикла:

```
(***)   ~B & RealInv(x, z) -> Post(x,z)
```

Это означает, что из истинности *инварианта* и условия завершения цикла следует требуемое постусловие.

Определение 4 (подходящего *инварианта)*: предикат `RealInv`, удовлетворяющий условиям (`*`), (`**`), (`***`) называется подходящим *инвариантом* цикла.

С циклом связано еще одно важное понятие - *варианта* цикла, используемое для доказательства завершаемости цикла.

Определение 5 (***варианта цикла***): целочисленное неотрицательное выражение `Var(x, z)` называется *вариантом цикла*, если выполняется следующая триада:

```
{(Var(x,z)= n) & B} S(x,z){(Var(x,z)= m) & (m < n)}
```

Содержательно это означает, что каждое выполнение тела цикла приводит к уменьшению значения его *варианта*. После конечного числа шагов *вариант* достигает своей нижней границы, и цикл завершается. Простейшим примером *варианта* цикла является выражение `n-i` для цикла:

```
for(i=1; i<=n; i++) S(x, z);
```

Пользоваться *инвариантами* и *вариантами* цикла нужно не только и не столько для того, чтобы проводить формальное *доказательство корректности* циклов. Они способствуют написанию корректных циклов. Правило корректного программирования гласит: "При написании каждого цикла программист должен определить его походящий *инвариант* и *вариант*". Задание предусловий, постусловий, *вариантов* и *инвариантов* циклов является такой же частью процесса разработки корректного метода, как и написание самого кода.

Рекурсия

Рекурсия является одним из наиболее мощных средств в арсенале программиста. Рекурсивные структуры данных и *рекурсивные методы* широко используются при построении программных систем. *Рекурсивные методы*, как правило, наиболее всего удобны при работе с рекурсивными структурами данных - списками, деревьями. *Рекурсивные методы* обхода деревьев служат классическим примером.

Определение 6 (***рекурсивного метода***): метод `P` (процедура или функция) называется *рекурсивным*, если при выполнении тела метода происходит вызов метода `P`.

Рекурсия может быть прямой, если вызов `P` происходит непосредственно в теле метода `P`. *Рекурсия* может быть косвенной, если в теле `P` вызывается метод `Q` (эта цепочка может быть продолжена), в теле которого вызывается метод `P`. Определения методов `P` и `Q` взаимно рекурсивны, если в теле метода `Q` вызывается метод `P`, вызывающий, в свою очередь, метод `Q`.

Для того чтобы *рекурсия* не приводила к зацикливанию, в тело нормального *рекурсивного метода* всегда встраивается оператор выбора, одна из ветвей которого не содержит рекурсивных вызовов. Если в теле *рекурсивного метода* рекурсивный вызов встречается только один раз, значит, что *рекурсию* можно заменить обычным циклом, что приводит к более эффективной программе, поскольку реализация *рекурсии* требует временных затрат и работы со стековой памятью. Приведу вначале простейший пример рекурсивного определения функции, вычисляющей факториал целого числа:

```
public long factorial(int n)
{
   if (n<=1) return(1);
   else return(n*factorial(n-1));
}//factorial
```

Функция `factorial` является примером прямого рекурсивного определения - в ее теле она сама себя вызывает. Здесь, как и положено, есть нерекурсивная ветвь, завершающая вычисления, когда `n` становится равным единице. Это пример так называемой "хвостовой" *рекурсии*, когда в теле встречается ровно один рекурсивный вызов, стоящий в конце соответствующего выражения. Хвостовую *рекурсию* намного проще записать в виде обычного цикла. Вот циклическое определение той же функции:

```
public long fact(int n)
{
   long res =1;
   for(int i = 2; i <=n; i++) res*=i;
   return(res);
}//fact
```

Конечно, циклическое определение проще, понятнее и эффективнее, и применять *рекурсию* в подобных ситуациях не следует. Интересно сравнить время вычислений, дающее некоторое представление о том, насколько эффективно реализуется *рекурсия*. Вот соответствующий тест, решающий эту задачу:

```
public void TestTailRec()
{
   Hanoi han = new Hanoi(5);
   long time1, time2;
   long f=0;
   time1 = getTimeInMilliseconds();
   for(int i = 1; i <1000000; i++)f =han.fact(15);
   time2 =getTimeInMilliseconds();
   Console.WriteLine(" f= {0}, " + "Время работы
      циклической процедуры: {1}",f,time2 -time1);
   time1 = getTimeInMilliseconds();
   for(int i = 1; i <1000000; i++)f =han.factorial(15);
   time2 =getTimeInMilliseconds();
   Console.WriteLine(" f= {0}, " + "Время работы
      рекурсивной процедуры: {1}",f,time2 -time1);
}
```

Каждая из функций вызывается в цикле, работающем 1000000 раз. До начала цикла и после его окончания вычисляется текущее время. Разность этих времен и дает оценку времени работы функций. Обе функции вычисляют факториал числа 15.

Проводить сравнение эффективности работы различных *вариантов* - это частый прием, используемый при разработке программ. И я им буду пользоваться неоднократно. Встроенный тип DateTime обеспечивает необходимую поддержку для получения текущего времени. Он совершенно необходим, когда приходится работать с датами. Я не буду подробно описывать его многочисленные статические и динамические методы и свойства. Ограничусь лишь приведением функции, которую я написал для получения текущего времени, измеряемого в миллисекундах. Статический метод Now класса DateTime возвращает объект этого класса, соответствующий дате и времени в момент создания объекта. Многочисленные свойства этого объекта позволяют извлечь требуемые характеристики. Приведу текст функции getTimeInMilliseconds:

```
long getTimeInMilliseconds()
{
   DateTime time = DateTime.Now;
   return(((time.Hour*60 + time.Minute)*60 + time.Second)*1000
      + time.Millisecond);
}
```

Результаты измерений времени работы рекурсивного и циклического *вариантов* функций слегка отличаются от запуска к запуску, но порядок остается одним и тем же. Эти результаты показаны на рис. 10.1.

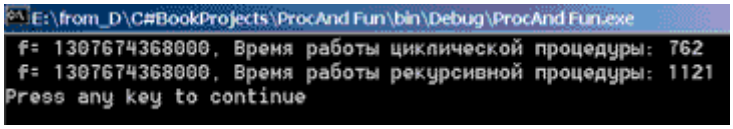

Рис. 10.1. Сравнение времени работы циклической и рекурсивной функций

Вовсе не обязательно, что *рекурсивные методы* будут работать медленнее нерекурсивных. Классическим примером являются методы сортировки. Известно, что время работы нерекурсивной пузырьковой сортировки имеет порядок c*n2, где c - некоторая константа. Для рекурсивной процедуры сортировки слиянием время работы - q*n*log(n), где q - константа. Понятно, что для больших n сортировка слиянием работает быстрее, независимо от соотношения значений констант. Сортировка слиянием - хороший пример применения *рекурсивных методов*. Она демонстрирует известный прием, называемый "разделяй и властвуй". Его суть в том, что исходная задача разбивается на подзадачи меньшей размерности, допускающие решение тем же алгоритмом. Решения отдельных подзадач затем объединяются, давая решение исходной задачи. В задаче сортировки исходный массив размерности n можно разбить на два массива размерности n/2, для каждого из которых рекурсивно вызывается метод сортировки слиянием. Полученные отсортированные массивы сливаются в единый массив с сохранением упорядоченности.

На примере сортировки слиянием покажем, как можно оценить время работы рекурсивной процедуры. Обозначим через T(n) время работы процедуры на массиве размерности n. Учитывая, что слияние можно выполнить за линейное время, справедливо следующее соотношение:

```
T(n) = 2T(n/2) + cn
```

Предположим для простоты, что n задается степенью числа 2, то есть n = 2k. Тогда наше соотношение имеет вид:

```
T(2k) = 2T(2k-1) + c2k
```

Полагая, что `T(1) =c`, путем несложных преобразований, используя индукцию, можно получить окончательный результат:

```
T(2k) = c*k*2k = c*n*log(n)
```

Известно, что это - лучшее по порядку время решения задачи сортировки. Когда исходную задачу удается разделить на подзадачи одинаковой размерности, то, при условии существования линейного алгоритма слияния, рекурсивный алгоритм имеет аналогичный порядок сложности. К сожалению, не всегда удается исходную задачу разбить на `k` подзадач одинаковой размерности `n/k`. Часто такое разбиение не представляется возможным.

Рекурсивное решение задачи "Ханойские башни"

Рассмотрим известную задачу о конце света - "Ханойские башни". Ее содержательная постановка такова. В одном из буддийских монастырей монахи уже тысячу лет занимаются перекладыванием колец. Они располагают тремя пирамидами, на которых надеты кольца разных размеров.

В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру. Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие - кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды. Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света.

Беспокоиться о близком конце света не стоит. Задача эта не под силу и современным компьютерам. Число ходов в ней равно 2^{64}, а это, как известно, большое число, и компьютер, работающий в сотню миллионов раз быстрее монахов, не справится с этой задачей в ближайшие тысячелетия.

Рассмотрим эту задачу в компьютерной постановке. Я спроектировал класс Hanoi, в котором роль пирамид играют три массива, а числа играют роль колец. Вот описание данных этого класса и некоторых его методов:

```
public class Hanoi
{
   int size,moves;
   int[] tower1, tower2,tower3;
   int top1,top2,top3;
   Random rnd = new Random();
   public Hanoi(int size)
   {
      this.size = size;
      tower1 = new int[size];
      tower2 = new int[size];
      tower3 = new int[size];
      top1 = size; top2=top3=moves =0;
   }
   public void Fill()
   {
      for(int i =0; i< size; i++)
         tower1[i]=size-i;
   }
}//Hanoi
```

Массивы `tower` играют роль ханойских башен, связанные с ними переменные `top` задают вершину - первую свободную ячейку при перекладывании колец (чисел). Переменная `size` задает размер массивов (число колец), а переменная `moves`

используется для подсчета числа ходов. Для дальнейших экспериментов нам понадобится генерирование случайных чисел, поэтому в классе определен объект уже известного нам класса `Random` (см. лекцию 7). Конструктор класса инициализирует поля класса, а метод `Fill` формирует начальное состояние, задавая для первой пирамиды числа, идущие в порядке убывания к ее вершине (`top`).

Займемся теперь непосредственно методом, реализующим нашу игру и перекладывающим кольца в соответствии с правилами игры. Заметьте, написать нерекурсивный *вариант* ханойских башен совсем не просто. Можно, конечно, написать цикл, завершающийся по достижению требуемой конфигурации, на каждом шаге которого выполняется очередной ход. Но даже первый ход не тривиален. Поскольку фиксирована пирамида, где должны быть собраны кольца, то неясно, куда нужно переложить первое кольцо - на вторую или третью пирамиду?

Рекурсивный *вариант* решения задачи прозрачен, хотя и напоминает некоторый род фокуса, что характерно для рекурсивного стиля мышления. Базис *рекурсии* прост. Для перекладывания одного кольца задумываться о решении не нужно - оно делается в один ход. Если есть базисное решение, то оставшаяся часть также очевидна. Нужно применить рекурсивно алгоритм, переложив `n-1` кольцо с первой пирамиды на третью пирамиду. Затем сделать очевидный ход, переложив последнее самое большое кольцо с первой пирамиды на вторую. Затем снова применить *рекурсию*, переложив `n-1` кольцо с третьей пирамиды на вторую пирамиду. Задача решена. Столь же проста ее запись на языке программирования:

```
public void HanoiTowers()
{
   HT(ref tower1,ref tower2, ref tower3,
      ref top1, ref top2, ref top3,size);
   Console.WriteLine("\nВсего ходов 2^n -1 = {0}",moves);
}
```

Как обычно в таких случаях, вначале пишется нерекурсивная процедура, вызывающая рекурсивный *вариант* с аргументами. В качестве фактических аргументов процедуре `HT` передаются поля класса, обновляемые в процессе многочисленных рекурсивных вызовов и потому снабженные ключевым словом `ref`. Рекурсивный *вариант* реализует описанную выше идею алгоритма:

```
/// <summary>
/// Перенос count колец с tower1 на tower2, соблюдая
/// правила и используя tower3. Свободные вершины
/// башен - top1, top2, top3
/// </summary>
void HT(ref int[] t1, ref int[] t2, ref int[] t3,
   ref int top1, ref int top2, ref int top3, int count)
{
   if (count == 1)Move(ref t1,ref t2, ref top1,ref top2);
   else
   {
      HT(ref t1,ref t3,ref t2,ref top1,ref top3, ref top2,count-1);
      Move(ref t1,ref t2,ref top1, ref top2);
      HT(ref t3,ref t2,ref t1,ref top3,ref top2, ref top1,count-1);
   }
}//HT
```

Процедура `Move` описывает очередной ход. Ее аргументы однозначно задают, с какой и на какую пирамиду нужно перенести кольцо. Никаких сложностей в ее реализации нет:

```
void Move(ref int[]t1, ref int[] t2, ref int top1, ref int top2)
{
```

```
   t2[top2] = t1[top1-1];
   top1--; top2++; moves++;
   //PrintTowers();
}//Move
```

Метод `PrintTowers` позволяет проследить за ходом переноса. Приведу еще метод класса `Testing`, тестирующий работу по переносу колец:

```
public void TestHanoiTowers()
{
   Hanoi han = new Hanoi(10);
   Console.WriteLine("Ханойские башни");
   han.Fill();
   han.PrintTowers();
   han.HanoiTowers();
   han.PrintTowers();
}
```

На рис. 10.2 показаны результаты работы с включенной печатью каждого хода для случая переноса трех колец.

Рис. 10.2. "Ханойские башни"

В рекурсивном *варианте* исчезли все трудности, связанные с выбором хода и соблюдением правил. Выбор выполняется почти автоматически, поскольку слияние частных решений не нарушает правил. В этом еще одна мощь *рекурсии*.

Решение исходной задачи свелось к решению двух подзадач и одному ходу. В отличие от задачи сортировки слиянием, обе подзадачи имеют не половинный размер, а размер, лишь на единицу меньший исходного. Это, казалось бы, незначительное изменение приводит к серьезным потерям эффективности вычислений. Если сложность в первом случае имела порядок `n*log(n)`, то теперь она становится экспоненциальной. Давайте проведем анализ временных затрат для ханойских башен (и всех задач, сводящихся к решению двух подзадач размерности `n-1`). Подсчитаем требуемое число ходов `T(n)`. С учетом структуры решения:

```
T(n) = 2T(n-1) +1
```

Простое доказательство по индукции дает:

```
T(n) = 2n-1 + 2n-2 + ... + 2 +1 = 2n -1
```

Можно показать, что последовательность ходов, реализуемая рекурсивным алгоритмом, является оптимальной, так что никакой другой алгоритм не может решить задачу за меньшее число ходов.

Быстрая сортировка Хоара

Продолжая тему *рекурсии*, познакомимся с реализацией на C# еще одного известного рекурсивного алгоритма, применяемого при сортировке массивов. Описанный ранее рекурсивный алгоритм сортировки слиянием имеет один существенный недостаток - для слияния двух упорядоченных массивов за линейное время необходима дополнительная память. Разработанный Ч. Хоаром метод сортировки, получивший название быстрого метода сортировки - `QuickSort`, не требует дополнительной памяти. Хотя этот метод и не является самым быстрым во всех случаях, но на практике он обеспечивает хорошие результаты. Нужно отметить, что именно этот метод сортировки встроен в класс `System.Array`.

Идея алгоритма быстрой сортировки состоит в том, чтобы выбрать в исходном массиве некоторый элемент `M`, затем в начальной части массива собрать все элементы, меньшие `M`. Так появляются две подзадачи размерности - `k` и `n-k`, к которым рекурсивно применяется алгоритм. Если в качестве элемента `M` выбирать медиану сортируемой части массива, то обе подзадачи имели бы одинаковый размер и алгоритм быстрой сортировки был бы оптимальным по времени работы. Но расчет медианы требует своих затрат времени и усложняет алгоритм. Поэтому обычно элемент `M` выбирается случайным образом. В этом случае быстрая сортировка оптимальна лишь в среднем, а для плохих *вариантов* (когда в качестве `M` всякий раз выбирается минимальный элемент) имеет порядок `n2`.

Несмотря на простоту идеи, алгоритм сложен в своей реализации, поскольку весь построен на циклах и операторах выбора. Я проводил построение алгоритма параллельно с обоснованием его *корректности*, введя *инварианты* соответствующих циклов. Текст обоснования встроен в текст метода. Приведу его, а затем дам некоторые объяснения. Вначале, как обычно, приведу нерекурсивную процедуру, вызывающую *рекурсивный метод*:

```
/// <summary>
/// Вызывает рекурсивную процедуру QSort,
/// передавая ей границы сортируемого массива.
/// Сортируемый массив tower1 задается
/// соответствующим полем класса.
public void QuickSort()
{
   QSort(0,size-1);
}
```

Вот чистый текст рекурсивной процедуры быстрой сортировки Хоара:

```
void QSort(int start, int finish)
{
   if(start != finish)
   {
      int ind = rnd.Next(start,finish);
      int item = tower1[ind];
      int ind1 = start, ind2 = finish;
      int temp;
      while (ind1 <=ind2)
      {
         while((ind1 <=ind2)&& (tower1[ind1] < item)) ind1++;
         while ((ind1 <=ind2)&&(tower1[ind2] >= item)) ind2--;
         if (ind1 < ind2)
         {
            temp = tower1[ind1]; tower1[ind1] = tower1[ind2];
            tower1[ind2] = temp;       ind1++; ind2--;
         }
      }
      if (ind1 == start)
      {
```

```
            temp = tower1[start]; tower1[start] = item;
            tower1[ind] = temp;
            QSort(start+1,finish);
         }
         else
         {
            QSort(start,ind1-1);
            QSort(ind2+1, finish);
         }
      }
}// QuickSort
```

Проведите эксперимент - закройте книгу и попробуйте написать эту процедуру самостоятельно. Если вам удастся сделать это без ошибок и она пройдет у вас с первого раза, то вы - блестящий программист и вам нужно читать другие книги. Я полагаю, что в таких процедурах ошибки неизбежны и для их исправления требуется серьезная отладка. Полагаю также, что помимо обычного тестирования полезно применять обоснование *корректности*, основанное на предусловиях и постусловиях, *инвариантах* цикла. Проектируя эту процедуру, я параллельно встраивал обоснование ее *корректности*. Это не строгое доказательство, но, дополняя тестирование, оно достаточно, чтобы автор поверил в *корректность* процедуры и представил ее на суд зрителей, как это сделал я.

```
/// <summary>
/// Небольшая по размеру процедура содержит три
/// вложенных цикла while, два оператора if и рекурсивные
/// вызовы. Для таких процедур задание инвариантов и
/// обоснование корректности облегчает отладку.
/// </summary>
/// <param name="start">начальный индекс сортируемой части
/// массива tower</param>
/// <param name="finish">конечный индекс сортируемой части
/// массива tower</param>
/// Предусловие: (start <= finish)
/// Постусловие: массив tower отсортирован по возрастанию
void QSort(int start, int finish)
{
   if(start != finish)
   //если (start = finish), то процедура ничего не делает,
   //но постусловие выполняется, поскольку массив из одного
   //элемента отсортирован по определению. Докажем истинность
   //постусловия для массива с числом элементов >1.
   {
      int ind = rnd.Next(start,finish);
      int item = tower1[ind];
      int ind1 = start, ind2 = finish;
      int temp;
   /// Введем три непересекающихся множества:
   /// S1: {tower1(i), start <= i =< ind1-1}
   /// S2: {tower1(i), ind1 <= i =< ind2}
   /// S1: {tower1(i), ind2+1 <= i =< finish}
   /// Введем следующие логические условия,
   /// играющие роль инвариантов циклов нашей программы:
   /// P1: объединение S1, S2, S3 = tower1
   /// P2: (S1(i) < item) Для всех элементов S1
   /// P3: (S3(i) >= item) Для всех элементов S3
   /// P4: item - случайно выбранный элемент tower1
   /// Нетрудно видеть, что все условия становятся
   /// истинными после завершения инициализатора цикла.
   /// Для пустых множеств S1 и S3 условия P2 и P3
   /// считаются истинными по определению.
   /// Inv = P1 & P2 & P3 & P4
      while (ind1 <=ind2)
```

```
      {
         while((ind1 <=ind2)&& (tower1[ind1] < item)) ind1++;
         //(Inv == true) & ~B1 (B1 - условие цикла while)
         while ((ind1 <=ind2)&&(tower1[ind2] >= item)) ind2--;
         //(Inv == true) & ~B2 (B2 - условие цикла while)
         if (ind1 < ind2)
         //Из Inv & ~B1 & ~B2 & B3 следует истинность:
         //((tower1[ind1] >= item)&&(tower1[ind2]<item))==true.
         //Это условие гарантирует, что последующий обмен
         //элементов обеспечит выполнение инварианта Inv
         {
            temp = tower1[ind1]; tower1[ind1] = tower1[ind2];
            tower1[ind2] = temp;
            ind1++; ind2--;
         }
         //(Inv ==true)
      }
      //из условия окончания цикла следует: (S2 - пустое множество)
      if (ind1 == start)
      //В этой точке S1 и S2 - это пустые множества, -> //(S3 = tower1)
      // Нетрудно доказать, что отсюда следует истинность:
      //(item = min)
      // Как следствие, можно минимальный элемент сделать первым,
      // а к оставшемуся множеству применить рекурсивный вызов.
      {
         temp = tower1[start]; tower1[start] = item;
         tower1[ind] = temp;
         QSort(start+1,finish);
      }
      else
      // Здесь оба множества S1 и S3 не пусты.
      // К ним применим рекурсивный вызов.
      {
         QSort(start,ind1-1);
         QSort(ind2+1, finish);
      }
      //Индукция по размеру массива и истинность инварианта
      //доказывает истинность постусловия в общем случае.
   }
}// QuickSort
```

Приведу некоторые пояснения к этому доказательству. Задание предусловия и постусловия процедуры `QSort` достаточно очевидно - сортируемый массив должен быть не пустым, а после работы метода должен быть отсортированным. Важной частью обоснования является четкое введение трех множеств - `S1`, `S2`, `S3` - и условий, накладываемых на их элементы. Эти условия и становятся частью *инварианта*, сохраняющегося при работе различных циклов нашего метода. Вначале множества `S1` и `S3` пусты, в ходе вычислений пустым становится множество `S2`. Так происходит формирование подзадач, к которым рекурсивно применяется алгоритм. Особым представляется случай, когда множество `S1` тоже пусто. Нетрудно показать, что эта ситуация возможна только в том случае, если случайно выбранный элемент множества, служащий критерием разбиения исходного множества на два подмножества, является минимальным элементом.

Почему обоснование полезно практически? Дело в том, что в данном алгоритме приходится следить за границами множеств (чтобы они не пересекались), за пустотой множеств (служащих условием окончания циклов), за выполнением условий, накладываемых на элементы множеств. Если явно не ввести эти понятия, то вероятность ошибки существенно возрастает. В заключение следует все-таки привести результат сортировки хотя бы одного массива.

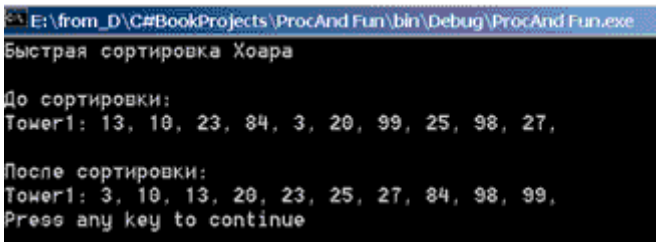

Рис. 10.3. Результаты быстрой сортировки массива

11. Лекция: Массивы языка C#

Общий взгляд

Массив задает способ организации данных. ***Массивом*** называют упорядоченную совокупность элементов одного типа. Каждый элемент *массива* имеет индексы, определяющие порядок элементов. Число индексов характеризует ***размерность массива***. Каждый индекс изменяется в некотором диапазоне `[a,b]`. В языке C#, как и во многих других языках, индексы задаются целочисленным типом. В других языках, например, в языке Паскаль, индексы могут принадлежать счетному конечному множеству, на котором определены функции, задающие следующий и предыдущий элемент. Диапазон `[a,b]` называется ***граничной парой***, `a` - ***нижней границей***, `b` - ***верхней границей*** индекса. При объявлении *массива* границы задаются выражениями. Если все границы заданы константными выражениями, то число элементов *массива* известно в момент его объявления и ему может быть выделена память еще на этапе трансляции. Такие *массивы* называются ***статическими***. Если же выражения, задающие границы, зависят от переменных, то такие *массивы* называются ***динамическими***, поскольку память им может быть отведена только динамически в процессе выполнения программы, когда становятся известными значения соответствующих переменных. *Массиву*, как правило, выделяется непрерывная область памяти.

В языке C++ все *массивы* являются *статическими*; более того, все *массивы* являются ***0-базируемыми***. Это означает, что *нижняя граница* всех индексов *массива* фиксирована и равна нулю. Введение такого ограничения имеет свою логику, поскольку здесь широко используется адресная арифметика. Так, несколько странное выражение `mas + i` , где `mas` - это имя *массива*, а `i` - индексное выражение, имеет вполне определенный смысл для C++ программистов. Имя *массива* интерпретируется как адрес первого элемента *массива*, к этому адресу прибавляется число, равное произведению `i` на размер памяти, необходимой для одного элемента *массива*. В результате сложения в такой адресной арифметике эффективно вычисляется адрес элемента `mas[i]`.

В языке C# снято существенное ограничение языка C++ на статичность *массивов*. *Массивы* в языке C# являются настоящими *динамическими массивами*. Как следствие этого, напомню, *массивы* относятся к ссылочным типам, память им отводится динамически в "куче". К сожалению, не снято ограничение *0-базируемости*, хотя, на мой взгляд, в таком ограничении уже нет логики из-за отсутствия в C# адресной арифметики. Было бы гораздо удобнее во многих задачах иметь возможность работать с *массивами*, у которых *нижняя граница* не равна нулю.

В языке C++ "классических" *многомерных массивов* нет. Здесь введены одномерные *массивы* и *массивы массивов*. Последние являются более общей структурой данных и позволяют задать не только многомерный куб, но и изрезанную, ступенчатую структуру. Однако использование *массива массивов* менее удобно, и, например, классик и автор языка C++ Бьерн Страуструп в своей книге "Основы языка C++" пишет: "Встроенные *массивы* являются главным источником ошибок - особенно когда они используются для построения *многомерных массивов*. Для новичков они также являются главным источником смущения и непонимания. По возможности пользуйтесь шаблонами `vector`, `valarray` и т.п.".

Шаблоны, определенные в стандартных библиотеках, конечно, стоит использовать, но все-таки странной является рекомендация не пользоваться структурами, встроенными непосредственно в язык. Замечу, что в других языках *массивы* являются одной из любимых структур данных, используемых программистами.

В языке C#, соблюдая преемственность, сохранены одномерные *массивы* и *массивы массивов*. В дополнение к ним в язык добавлены *многомерные массивы*. Динамические

многомерные массивы языка C# являются весьма мощной, надежной, понятной и удобной структурой данных, которую смело можно рекомендовать к применению не только профессионалам, но и новичкам, программирующим на C#. После этого краткого обзора давайте перейдем к более систематическому изучению деталей работы с *массивами* в C#.

Объявление массивов

Рассмотрим, как объявляются одномерные *массивы*, *массивы массивов* и *многомерные массивы*.

Объявление одномерных массивов

Напомню общую структуру объявления:

```
[<атрибуты>] [<модификаторы>] <тип> <объявители>;
```

Забудем пока об атрибутах и модификаторах. Объявление одномерного *массива* выглядит следующим образом:

```
<тип>[] <объявители>;
```

Заметьте, в отличие от языка C++ квадратные скобки приписаны не к имени переменной, а к типу. Они являются неотъемлемой частью определения класса, так что запись `T[]` следует понимать как класс **одномерный** *массив* **с элементами типа** `T`.

Что же касается границ изменения индексов, то эта характеристика к классу не относится, она является характеристикой переменных - экземпляров, каждый из которых является одномерным *массивом* со своим числом элементов, задаваемых в объявителе переменной.

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

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

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

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

Следуя синтаксу, элементы константного *массива* следует заключать в фигурные скобки.

Во втором случае создание и инициализация *массива* выполняется в объектном стиле с вызовом конструктора *массива*. И это наиболее распространенная практика объявления *массивов*. Приведу пример:

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

Итак, если *массив* объявляется без инициализации, то создается только висячая ссылка со значением `void`. Если инициализация выполняется конструктором, то в динамической памяти создается сам *массив*, элементы которого инициализируются константами соответствующего типа (ноль для арифметики, пустая строка для строковых *массивов*), и ссылка связывается с этим *массивом*. Если *массив* инициализируется константным *массивом*, то в памяти создается константный *массив*, с которым и связывается ссылка.

Как обычно задаются элементы *массива*, если они не заданы при инициализации? Они либо вычисляются, либо вводятся пользователем. Давайте рассмотрим первый пример работы с *массивами* из проекта с именем `Arrays`, поддерживающего эту лекцию:

```
public void TestDeclaration()
{
   //объявляются три одномерных массива A,B,C
   int[] A = new int[5], B= new int[5], C= new int[5];
   Arrs.CreateOneDimAr(A);
   Arrs.CreateOneDimAr(B);
   for(int i = 0; i<5; i++)
      C[i] = A[i] + B[i];
   //объявление массива с явной инициализацией
   int[] x ={5,5,6,6,7,7};
   //объявление массивов с отложенной инициализацией
   int[] u,v;
   u = new int[3];
   for(int i=0; i<3; i++) u[i] =i+1;
   //v= {1,2,3}; //присваивание константного массива
   //недопустимо
   v = new int[4];
   v=u; //допустимое присваивание
   Arrs.PrintAr1("A", A); Arrs.PrintAr1("B", B);
   Arrs.PrintAr1("C", C); Arrs.PrintAr1("X", x);
   Arrs.PrintAr1("U", u); Arrs.PrintAr1("V", v);
}
```

На что следует обратить внимание, анализируя этот текст:

- В процедуре показаны разные способы объявления *массивов*. Вначале объявляются одномерные *массивы* `A`, `B` и `C`, создаваемые конструктором. Значения элементов этих трех *массивов* имеют один и тот же тип `int`. То, что они имеют одинаковое число элементов, произошло по воле программиста, а не диктовалось требованиями языка. Заметьте, что после такого объявления с инициализацией конструктором, все элементы имеют значение, в данном случае - ноль, и могут участвовать в вычислениях.
- *Массив* `x` объявлен с явной инициализацией. Число и значения его элементов определяется константным *массивом*.
- *Массивы* `u` и `v` объявлены с отложенной инициализацией. В последующих операторах *массив* `u` инициализируется в объектном стиле - элементы получают его в цикле значения.
- Обратите внимание на закомментированный оператор присваивания. В отличие от инициализации, использовать константный *массив* в правой части оператора присваивания недопустимо. Эта попытка приводит к ошибке, поскольку `v` - это ссылка, которой можно присвоить ссылку, но нельзя присвоить константный *массив*. Ссылку присвоить можно. Что происходит в операторе присваивания `v = u`? Это корректное ссылочное присваивание: хотя `u` и `v` имеют разное число элементов, но они являются объектами одного класса. В результате присваивания память, отведенная *массиву* `v`, освободится ею займется теперь сборщик мусора. Обе ссылки `u` и `v` будут теперь указывать на один и тот же

массив, так что изменение элемента одного *массива* немедленно отразится на другом *массиве*.

- Далее определяется двумерный *массив* `w` и делается попытка выполнить оператор присваивания `v=w`. Это ссылочное присваивание некорректно, поскольку объекты `w` и `v` - разных классов и для них не выполняется требуемое для присваивания согласование по типу.
- Для поддержки работы с *массивами* создан специальный класс `Arrs`, статические методы которого выполняют различные операции над *массивами*. В частности, в примере использованы два метода этого класса, один из которых заполняет *массив* случайными числами, второй - выводит *массив* на печать. Вот текст первого из этих методов:

```
public static void CreateOneDimAr(int[] A)
{
   for(int i = 0; i<A.GetLength(0);i++)
      A[i] = rnd.Next(1,100);
}//CreateOneDimAr
```

Здесь `rnd` - это статическое поле класса `Arrs`, объявленное следующим образом:

```
private  static Random rnd = new Random();
```

Процедура печати *массива* с именем `name` выглядит так:

```
public static void PrintAr1(string name,int[] A)
{
   Console.WriteLine(name);
   for(int i = 0; i<A.GetLength(0);i++)
      Console.Write("\t" + name + "[{0}]={1}", i, A[i]);
   Console.WriteLine();
}//PrintAr1
```

На рис. 11.1 показан консольный вывод результатов работы процедуры `TestDeclarations`.

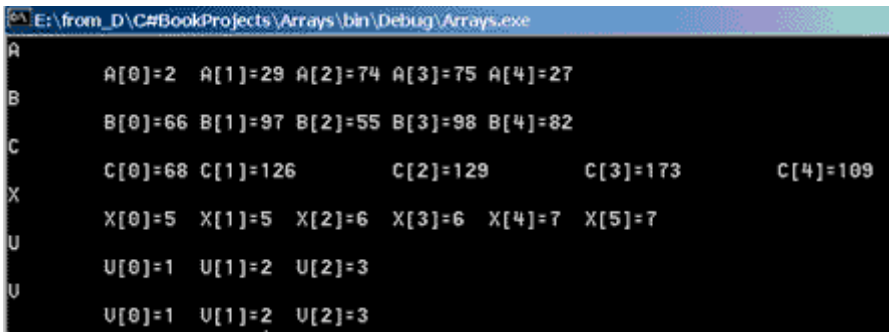

Рис. 11.1. Результаты объявления и создания массивов

Особое внимание обратите на вывод, связанный с *массивами* `u` и `v`.

Динамические массивы

Во всех вышеприведенных примерах объявлялись *статические массивы*, поскольку *нижняя граница* равна нулю по определению, а *верхняя* всегда задавалась в этих примерах константой. Напомню, что в C# все *массивы*, независимо от того, каким выражением описывается граница, рассматриваются как *динамические*, и память для них распределяется в "куче". Полагаю, что это отражение разумной точки зрения: ведь *статические массивы*, скорее исключение, а правилом является использование

динамических массивов. В действительности реальные потребности в размере *массива*, скорее всего, выясняются в процессе работы в диалоге с пользователем.

Чисто синтаксически нет существенной разницы в объявлении *статических* и *динамических массивов*. Выражение, задающее границу изменения индексов, в динамическом случае содержит переменные. Единственное требование - значения переменных должны быть определены в момент объявления. Это ограничение в C# выполняется автоматически, поскольку хорошо известно, сколь требовательно C# контролирует инициализацию переменных.

Приведу пример, в котором описана работа с *динамическим массивом*:

```
public void TestDynAr()
{
   //объявление динамического массива A1
   Console.WriteLine("Введите число элементов массива A1");
   int size = int.Parse(Console.ReadLine());
   int[] A1 = new int[size];
   Arrs.CreateOneDimAr(A1);
   Arrs.PrintAr1("A1",A1);
}//TestDynAr
```

В особых комментариях эта процедура не нуждается. Здесь *верхняя граница массива* определяется пользователем.

Многомерные массивы

Уже объяснялось, что разделение *массивов* на одномерные и *многомерные* носит исторический характер. Никакой принципиальной разницы между ними нет. Одномерные *массивы* -это частный случай *многомерных*. Можно говорить и по-другому: *многомерные массивы* являются естественным обобщением одномерных. Одномерные *массивы* позволяют задавать такие математические структуры как векторы, двумерные - матрицы, трехмерные - кубы данных, *массивы* большей *размерности* - многомерные кубы данных. Замечу, что при работе с базами данных многомерные кубы, так называемые кубы OLAP, встречаются сплошь и рядом.

В чем особенность объявления *многомерного массива*? Как в типе указать *размерность массива*? Это делается достаточно просто, за счет использования запятых. Вот как выглядит объявление *многомерного массива* в общем случае:

```
<тип>[, ... ,] <объявители>;
```

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

```
public void TestMultiArr()
{
   int[,]matrix = {{1,2},{3,4}};
   Arrs.PrintAr2("matrix", matrix);
}//TestMultiArr
```

Давайте рассмотрим классическую задачу умножения прямоугольных матриц. Нам понадобится три *динамических массива* для представления матриц и три процедуры,

одна из которых будет заполнять исходные матрицы случайными числами, другая - выполнять умножение матриц, третья - печатать сами матрицы. Вот тестовый пример:

```
public void TestMultiMatr()
{
   int n1, m1, n2, m2,n3, m3;
   Arrs.GetSizes("MatrA",out n1,out m1);
   Arrs.GetSizes("MatrB",out n2,out m2);
   Arrs.GetSizes("MatrC",out n3,out m3);
   int[,]MatrA = new int[n1,m1], MatrB = new int[n2,m2];
   int[,]MatrC = new int[n3,m3];
   Arrs.CreateTwoDimAr(MatrA);Arrs.CreateTwoDimAr(MatrB);
   Arrs.MultMatr(MatrA, MatrB, MatrC);
   Arrs.PrintAr2("MatrA",MatrA); Arrs.PrintAr2("MatrB",MatrB);
   Arrs.PrintAr2("MatrC",MatrC);
}//TestMultiMatr
```

Три матрицы - `MatrA`, `MatrB` и `MatrC` - имеют произвольные размеры, выясняемые в диалоге с пользователем, и использование для их описания *динамических массивов* представляется совершенно естественным. Метод `CreateTwoDimAr` заполняет случайными числами элементы матрицы, переданной ему в качестве аргумента, метод `PrintAr2` выводит матрицу на печать. Я не буду приводить их код, похожий на код их одномерных аналогов.

Метод `MultMatr` выполняет умножение прямоугольных матриц. Это классическая задача из набора задач, решаемых на первом курсе. Вот текст этого метода:

```
public void MultMatr(int[,]A, int[,]B, int[,]C)
{
   if (A.GetLength(1) != B.GetLength(0))
      Console.WriteLine("MultMatr: ошибка размерности!");
   else
      for(int i = 0; i < A.GetLength(0); i++)
         for(int j = 0; j < B.GetLength(1); j++)
         {
            int s=0;
            for(int k = 0; k < A.GetLength(1); k++)
               s+= A[i,k]*B[k,j];
            C[i,j] = s;
         }
}//MultMatr
```

В особых комментариях эта процедура не нуждается. Замечу лишь, что прежде чем проводить вычисления, производится проверка корректности размерностей исходных матриц при их перемножении, - число столбцов первой матрицы должно быть равно числу строк второй матрицы.

Обратите внимание, как выглядят результаты консольного вывода на данном этапе работы (рис. 11.2).

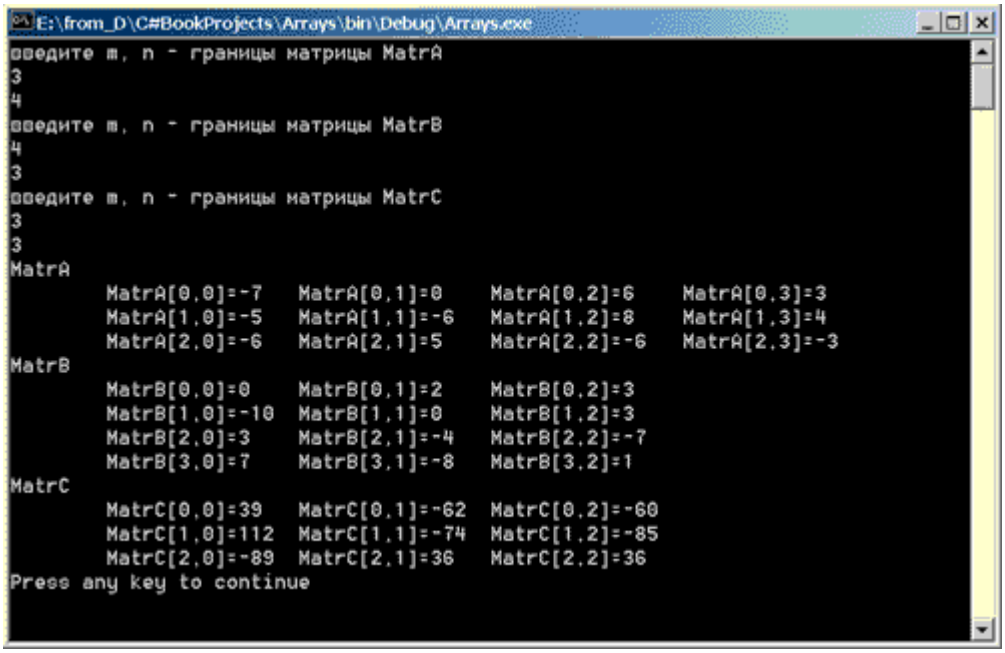

Рис. 11.2. Умножение матриц

Массивы массивов

Еще одним видом *массивов* C# являются ***массивы массивов***, называемые также ***изрезанными массивами (jagged arrays)***. Такой *массив массивов* можно рассматривать как одномерный *массив*, элементы которого являются *массивами*, элементы которых, в свою очередь, снова могут быть *массивами*, и так может продолжаться до некоторого уровня вложенности.

В каких ситуациях может возникать необходимость в таких структурах данных? Эти *массивы* могут применяться для представления деревьев, у которых узлы могут иметь произвольное число потомков. Таковым может быть, например, генеалогическое дерево. Вершины первого уровня - `Fathers`, представляющие отцов, могут задаваться одномерным *массивом*, так что `Fathers[i]` - это `i`-й отец. Вершины второго уровня представляются *массивом массивов* - `Children`, так что `Children[i]` - это *массив* детей `i`-го отца, а `Children[i][j]` - это `j`-й ребенок `i`-го отца. Для представления внуков понадобится третий уровень, так что `GrandChildren [i][j][k]` будет представлять `k`-го внука `j`-го ребенка `i`-го отца.

Есть некоторые особенности в объявлении и инициализации таких *массивов*. Если при объявлении типа *многомерных массивов* для указания *размерности* использовались запятые, то для *изрезанных массивов* применяется более ясная символика - совокупности пар квадратных скобок; например, `int[][]` задает *массив*, элементы которого - одномерные *массивы* элементов типа `int`.

Сложнее с созданием самих *массивов* и их инициализацией. Здесь нельзя вызвать конструктор `new int[3][5]`, поскольку он не задает *изрезанный массив*. Фактически нужно вызывать конструктор для каждого *массива* на самом нижнем уровне. В этом и состоит сложность объявления таких *массивов*. Начну с формального примера:

```
//массив массивов - формальный пример
//объявление и инициализация
int[][] jagger = new int[3][]
{
   new int[] {5,7,9,11},
   new int[] {2,8},
   new int[] {6,12,4}
```

```
};
```

Массив `jagger` имеет всего два уровня. Можно считать, что у него три элемента, каждый из которых является *массивом*. Для каждого такого *массива* необходимо вызвать конструктор `new`, чтобы создать внутренний *массив*. В данном примере элементы внутренних *массивов* получают значение, будучи явно инициализированы константными *массивами*. Конечно, допустимо и такое объявление:

```
int[][] jagger1 = new int[3][]
{
   new int[4],
   new int[2],
   new int[3]
};
```

В этом случае элементы *массива* получат при инициализации нулевые значения. Реальную инициализацию нужно будет выполнять программным путем. Стоит заметить, что в конструкторе верхнего уровня константу 3 можно опустить и писать просто `new int[][]`. Самое забавное, что вызов этого конструктора можно вообще опустить - он будет подразумеваться:

```
int[][] jagger2 =
{
   new int[4],
   new int[2],
   new int[3]
};
```

А вот конструкторы нижнего уровня необходимы. Еще одно важное замечание - *динамические массивы* возможны и здесь. В общем случае, границы на любом уровне могут быть выражениями, зависящими от переменных. Более того, допустимо, чтобы *массивы* на нижнем уровне были *многомерными*. Но это уже "от лукавого" - вряд ли стоит пользоваться такими сложными структурами данных, ведь с ними предстоит еще и работать.

Приведу теперь чуть более реальный пример, описывающий простое генеалогическое дерево, которое условно назову "отцы и дети":

```
//массив массивов -"Отцы и дети"
int Fcount =3;
string[] Fathers = new string[Fcount];
Fathers[0] ="Николай"; Fathers[1] = "Сергей";
   Fathers[2] = "Петр";
string[][] Children = new string[Fcount][];
Children[0] = new string[] {"Ольга", "Федор"};
Children[1] = new string[]
   {"Сергей","Валентина","Ира","Дмитрий"};
Children[2] = new string[]{"Мария","Ирина","Надежда"};
myar.PrintAr3(Fathers,Children);
```

Здесь отцов описывает обычный *динамический* одномерный *массив* `Fathers`. Для описания детей этих отцов необходим уже *массив массивов*, который также является динамическим на верхнем уровне, поскольку число его элементов совпадает с числом элементов *массива* `Fathers`. Здесь показан еще один способ создания таких *массивов*. Вначале конструируется *массив* верхнего уровня, содержащий ссылки со значением `void`. А затем на нижнем уровне конструктор создает настоящие *массивы* в динамической памяти, с которыми и связываются ссылки.

Я не буду демонстрировать работу с генеалогическим деревом, ограничусь лишь печатью этого *массива*. Здесь есть несколько поучительных моментов. В классе Arrs для печати *массива* создан специальный метод `PrintAr3`, которому в качестве аргументов передаются *массивы* `Fathers` и `Children`. Вот текст данной процедуры:

```
public void PrintAr3(string [] Fathers, string[][] Children)
{
   for(int i = 0; i < Fathers.Length; i++)
   {
      Console.WriteLine("Отец : {0}; Его дети:", Fathers[i]);
      for(int j = 0; j < Children[i].Length; j++)
         Console.Write( Children[i][j] + " ");
      Console.WriteLine();
   }
}//PrintAr3
```

Приведу некоторые комментарии к этой процедуре:

- Внешний цикл по `i` организован по числу элементов *массива* `Fathers`. Заметьте, здесь используется свойство `Length`, в отличие от ранее применяемого метода `GetLength`.
- В этом цикле с тем же успехом можно было бы использовать и имя *массива* `Children`. Свойство `Length` для него возвращает число элементов верхнего уровня, совпадающее, как уже говорилось, с числом элементов *массива* `Fathers`.
- Во внутреннем цикле свойство `Length` вызывается для каждого элемента `Children[i]`, который является *массивом*.
- Остальные детали, надеюсь, понятны.

Приведу вывод, полученный в результате работы процедуры `PrintAr3`.

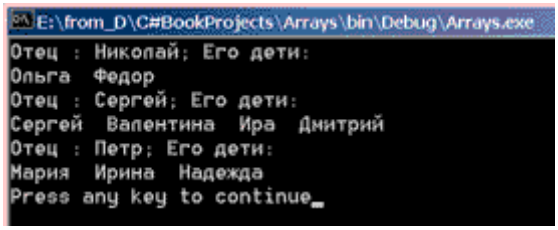

Рис. 11.3. Дерево "Отцы и дети"

Процедуры и массивы

В наших примерах *массивы* неоднократно передавались процедурам в качестве входных аргументов и возвращались в качестве результатов.

В лекции 9 подробно описывались особенности передачи аргументов в процедуру. Остается подчеркнуть только некоторые детали:

- В процедуру достаточно передавать только сам объект - *массив*. Все его характеристики (*размерность*, границы) можно определить, используя свойства и методы этого объекта.
- Когда *массив* является выходным аргументом процедуры, как аргумент `C` в процедуре `MultMatr`, выходной аргумент совсем не обязательно снабжать ключевым словом `ref` или `out` (хотя и допустимо). Передача аргумента по значению в таких ситуациях так же хороша, как и передача по ссылке. В результате вычислений меняется сам *массив* в динамической памяти, а ссылка на него остается постоянной. Процедура и ее вызов без ключевых слов выглядит проще, поэтому обычно они опускаются. Заметьте, в процедуре `GetSizes`, где

определялись границы *массива*, ключевое слово `out`, сопровождающее аргументы, совершенно необходимо.

- Может ли процедура-функция возвращать *массив* в качестве результата? В C# ответ на этот вопрос положителен. В следующей лекции будет приведен пример подобной функции.

12. Лекция: Класс Array и новые возможности массивов

Класс Array

Нельзя понять многие детали работы с массивами в C#, если не знать устройство класса `Array` из библиотеки FCL, потомками которого являются все классы-массивы. Рассмотрим следующие объявления:

```
//Класс Array
         int[] ar1 = new int[5];
         double[] ar2 ={5.5, 6.6, 7.7};
         int[,] ar3 = new Int32[3,4];
```

Зададимся естественным вопросом: к какому или к каким классам принадлежат объекты `ar1`, `ar2` и `ar3`? Ответ прост: все они принадлежат к разным классам. Переменная `ar1` принадлежит к классу `int[]` - одномерному массиву значений типа `int`, `ar2` - `double[]` - одномерному массиву значений типа `double`, `ar3` - двумерному массиву значений типа `int`. Следующий закономерный вопрос: а что общего есть у этих трех объектов? Прежде всего, все три класса этих объектов, как и другие классы, являются потомками класса `Object`, а потому имеют общие методы, наследованные от класса `Object` и доступные объектам этих классов.

У всех классов, являющихся массивами, много общего, поскольку все они являются потомками *класса System.Array*. *Класс System.Array* наследует ряд **интерфейсов**: *ICloneable*, *IList*, *ICollection*, *IEnumerable*, а, следовательно, обязан реализовать все их методы и свойства. Помимо наследования свойств и методов класса `Object` и вышеперечисленных интерфейсов, класс `Array` имеет довольно большое число собственных методов и свойств. Взгляните, как выглядит *отношение наследования* на семействе классов, определяющих массивы.

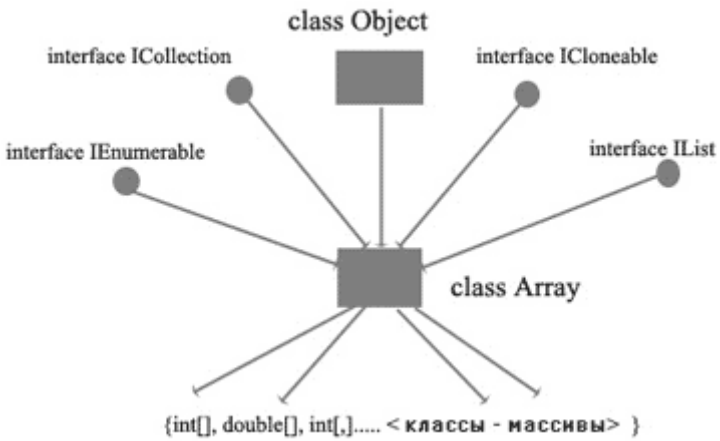

Рис. 12.1. Отношение наследования на классах-массивах

Благодаря такому мощному родителю, над массивами определены самые разнообразные операции - копирование, *поиск*, обращение, *сортировка*, получение различных характеристик. Массивы можно рассматривать как *коллекции* и устраивать циклы `For Each` для перебора всех элементов. Важно и то, что когда у семейства классов есть общий родитель, то можно иметь общие процедуры обработки различных потомков этого родителя. Для общих процедур работы с массивами характерно, что один или несколько формальных аргументов имеют родительский тип `Array`. Естественно, внутри такой процедуры может понадобиться анализ - какой реальный тип массива передан в процедуру.

Рассмотрим пример подобной процедуры. Ранее я для печати элементов массива использовал различные процедуры `PrintAr1`, `PrintAr2` и так далее, по одной для каждого класса массива. Теперь я приведу общую процедуру, формальный аргумент которой будет принадлежать родителю всех классов-массивов, что позволит передавать массив любого класса в качестве фактического аргумента:

```
public static void PrintAr(string name, Array A)
{
   Console.WriteLine(name);
   switch (A.Rank)
   {
      case 1:
         for(int i = 0; i<A.GetLength(0);i++)
            Console.Write("\t" + name + "[{0}]={1}",
               i, A.GetValue(i));
         Console.WriteLine();
         break;
      case 2:
         for(int i = 0; i<A.GetLength(0);i++)
         {
            for(int j = 0; j<A.GetLength(1);j++)
               Console.Write("\t" + name + "[{0},{1}]={2}",
                  i,j, A.GetValue(i,j));
            Console.WriteLine();
         }
         break;
      default: break;
   }
}//PrintAr
```

Вот как выглядит создание массивов и вызов процедуры печати:

```
public void TestCommonPrint()
{
   //Класс Array
   int[] ar1 = new int[5];
   double[] ar2 ={5.5, 6.6, 7.7};
   int[,] ar3 = new Int32[3,4];
   Arrs.CreateOneDimAr(ar1);Arrs.PrintAr("ar1", ar1);
   Arrs.PrintAr("ar2", ar2);
   Arrs.CreateTwoDimAr(ar3);Arrs.PrintAr("ar3", ar3);
}//TestCommonPrint
```

Вот результаты вывода массивов `ar1`, `ar2` и `ar3`.

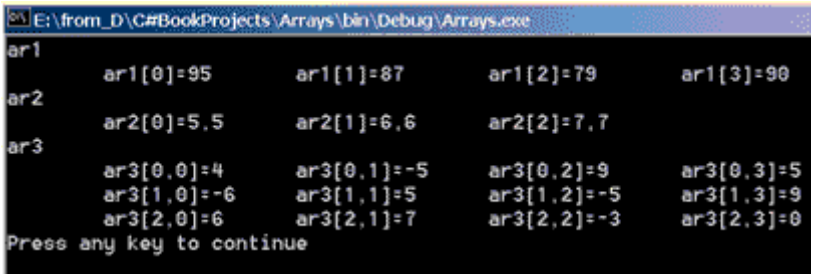

Рис. 12.2. Печать массивов. Результаты работы процедуры PrintAr

Приведу некоторые комментарии.

Первое, на что следует обратить внимание: формальный аргумент процедуры принадлежит базовому классу `Array`, наследниками которого являются все массивы в CLR и, естественно, все массивы C#.

Для того чтобы сохранить возможность работы с индексами, как в одномерном, так и в двумерном случае, пришлось организовать разбор случаев. Свойство `Rank`, возвращающее размерность массива, используется в этом разборе.

К элементам массива `A`, имеющего класс `Array`, нет возможности прямого доступа в обычной манере - `A [<индексы>]`, но зато есть специальные методы `GetValue (<индексы>)` и `SetValue (<индексы>)`.

Естественно, разбор случаев можно продолжить, придав процедуре большую функциональность.

Заметьте, если разбор случаев вообще не делать, а использовать `PrintAr` только для печати одномерных массивов, то она будет столь же проста, как и процедура `PrintAr1`, но сможет печатать любые одномерные массивы, независимо от типа их элементов.

Массивы как коллекции

В ряде задач массивы C# целесообразно рассматривать как *коллекции*, не используя систему индексов для *поиска* элементов. Это, например, задачи, требующие однократного или многократного прохода по всему массиву - нахождение суммы элементов, нахождение максимального элемента, печать элементов. В таких задачах вместо циклов типа `For` по каждому измерению достаточно рассмотреть единый цикл `For Each` по всей *коллекции*. Эта возможность обеспечивается тем, что класс `Array` наследует интерфейс *IEnumerable*. Обратите внимание, этот интерфейс обеспечивает только возможность чтения элементов *коллекции* (массива), не допуская их изменения. Применим эту стратегию и построим еще одну версию процедуры печати. Эта версия будет самой короткой и самой универсальной, поскольку подходит для печати массива, независимо от его размерности и типа элементов. Вот ее код:

```
public static void PrintCollection(string name,Array A)
{
   Console.WriteLine(name);
   foreach (object item in A )
      Console.Write("\t {0}", item);
   Console.WriteLine();
}//PrintCollection
```

Конечно, за все нужно платить. Платой за универсальность процедуры печати является то, что многомерный массив печатается как одномерный без разделения элементов на строки.

К сожалению, ситуация с чтением и записью элементов массива не симметрична. Приведу вариант процедуры `CreateCollection`:

```
public static void CreateCollection(Array A)
{
   int i=0;
   foreach (object item in A )
      //item = rnd.Next(1,10); //item read only
      A.SetValue(rnd.Next(1,10), i++);
}//CreateCollection
```

Заметьте, эту процедуру сделать универсальной не удается, поскольку невозможно модифицировать элементы *коллекции*. Поэтому цикл `For Each` здесь ничего не дает, и разумнее использовать обычный цикл. Данная процедура не универсальна и позволяет создавать элементы только для одномерных массивов.

Сортировка и поиск. Статические методы класса Array

Статические методы класса `Array` позволяют решать самые разнообразные задачи:

1. ***Copy*** - позволяет копировать весь массив или его часть в другой массив.
2. ***IndexOf***, ***LastIndexOf*** - определяют индексы первого и последнего вхождения образца в массив, возвращая -1, если такового вхождения не обнаружено.
3. ***Reverse*** - выполняет обращение массива, переставляя элементы в обратном порядке.
4. ***Sort*** - осуществляет *сортировку* массива.
5. ***BinarySearch*** - определяет индекс первого вхождения образца в отсортированный массив, используя алгоритм двоичного *поиска*.

Все методы перегружены и имеют ряд модификаций. Большинство из этих методов применимо только к одномерным массивам. Приведу примеры различных операций, доступных при работе с массивами, благодаря наследованию от класса `Array`:

```
public void TestCollection()
{
   //операции над массивами
   int nc = 7;
   int[] col1 = new int[nc], col2 = new int[nc];
   double[] col3 = new double[nc];
   int[,] col4 = new int[2,2];
   Arrs.CreateCollection(col1);
   Arrs.PrintCollection("col1",col1);
   Arrs.CreateCollection(col2);
   Arrs.PrintCollection("col2",col2);
   Arrs.CreateCollection(col3);
   Arrs.PrintCollection("col3",col3);
   Arrs.CreateTwoDimAr(col4);
   Arrs.PrintCollection("col4",col4);
   //сортировка, поиск, копирование
   // поиск элемента
   int first = Array.IndexOf(col1, 2);
   int last =  Array.LastIndexOf(col1,2);
   if (first == -1)
      Console.WriteLine("Нет вхождений 2 в массив col1");
   else if (first ==last)
      Console.WriteLine("Одно вхождение 2 в массив col1");
   else
      Console.WriteLine("Несколько вхождений 2 в массив col1");
   //first = Array.IndexOf(col4, 4);
   //только одномерный массив
   Array.Reverse(col1);
   Console.WriteLine("Обращение массива col1:");
   Arrs.PrintCollection("col1",col1);
   //Копирование
   Array.Copy(col1, col3, col1.Length);
   Console.WriteLine(" Массив col3 после копирования массива col1:");
   Arrs.PrintCollection("col3",col3);
   Array.Copy(col1,1,col2,1,2);
   Console.WriteLine("копирование двух элементов col1 в col2:");
   Arrs.PrintCollection("col1",col1);
   Arrs.PrintCollection("col2",col2);
   //быстрая сортировка Хоара
   Array.Sort(col1);
   Console.WriteLine("Отсортированный массив col1:");
   Arrs.PrintCollection("col1",col1);
   first = Array.BinarySearch(col1, 2);
   Console.WriteLine("Индекс вхождения 2 в col1: {0}",first);
   //Создание экземпляра (массива)
   Array my2Dar = Array.CreateInstance(typeof(double), 2,3);
   Arrs.PrintCollection("my2Dar",my2Dar);
```

```
    //клонирование
    my2Dar = (Array)col4.Clone();
    Console.WriteLine("Массив my2Dar после клонирования
       col4:");
    Arrs.PrintCollection("my2Dar",my2Dar);
    //копирование CopyTo
    col1.CopyTo(col2,0);
    Console.WriteLine("Массив col2 после копирования col1:");
    Arrs.PrintCollection("col2",col2);
}
```

В этой процедуре продемонстрированы вызовы различных статических методов класса Array. Для метода *Copy* показан вызов двух реализаций этого метода, когда копируется весь массив и часть массива. Закомментированный оператор вызова метода *IndexOf* напоминает о невозможности использования методов *поиска* при работе с многомерными массивами. Приведу результаты вывода, порожденные этим кодом.

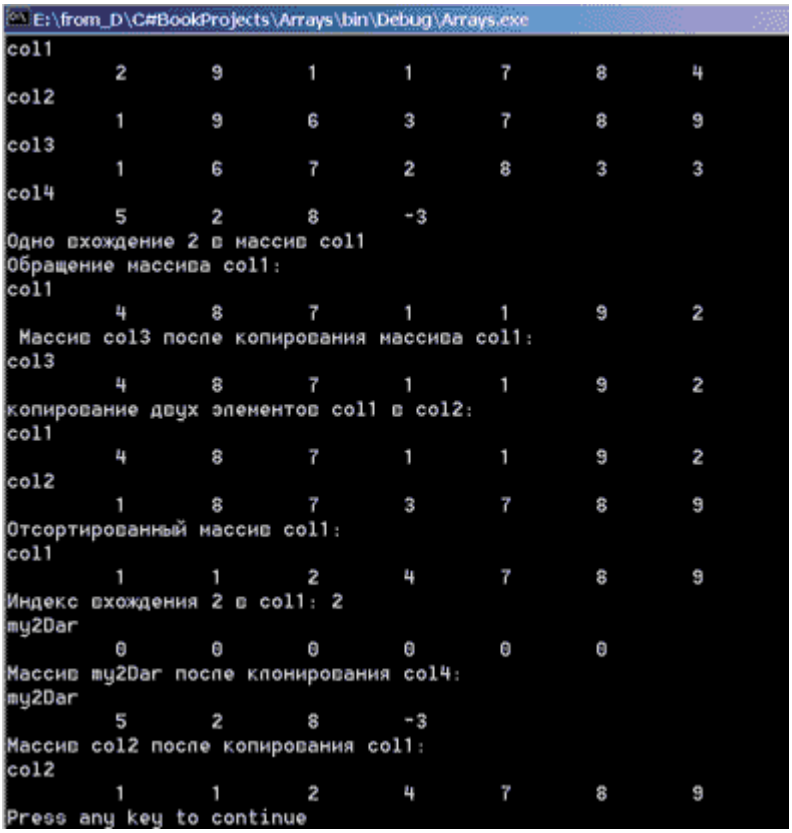

Рис. 12.3. Результаты применения статических методов класса Array

Таблица 12.1. Свойства класса Array		
Свойство	**Родитель**	**Описание**
IsFixedSize	Интерфейс *IList*	`True`, если массив статический
IsReadOnly	Интерфейс *IList*	Для всех массивов имеет значение `false`
IsSynchronized	Интерфейс *ICollection*	`True` или `False`, в зависимости от того, установлена ли синхронизация доступа для массива
SyncRoot	Интерфейс *ICollection*	Собственный метод синхронизации доступа к массиву. При работе с массивом его можно закрыть на время обработки, что запрещает его модификацию каким-либо потоком: `Array myCol = new int[];`

		`lock( myCol.SyncRoot ) {`
		`foreach ( Object item in myCol )`
		`{`
		`// безопасная обработка массива }`

Length		Число элементов массива
Rank		Размерность массива

Таблица 12.2. Статические методы класса Array

Метод	Описание
BinarySearch	Двоичный *поиск*. Описание и примеры даны в тексте
`Clear`	Выполняет начальную инициализацию элементов. В зависимости от типа элементов устанавливает значение `0` для арифметического типа, `false` - для логического типа, `Null` для ссылок, "" - для строк.
Copy	Копирование части или всего массива в другой массив. Описание и примеры даны в тексте
`CreateInstance`	Класс `Array`, в отличие от многих классов, может создавать свои экземпляры не только с помощью конструктора `new`, но и при вызове метода `CreateInstance`: `Array my2Dar = Array.CreateInstance(typeof(double), 2,2)`
IndexOf	Индекс первого вхождения образца в массив. Описание и примеры даны в тексте
LastIndexOf	Индекс последнего вхождения образца в массив. Описание и примеры даны в тексте
Reverse	Обращение одномерного массива. Описание и примеры даны в тексте
Sort	*Сортировка* массива. Описание и примеры даны в тексте

Сводка свойств и методов класса Array

Многие возможности, которыми можно пользоваться при работе с массивами, уже обсуждены. В завершение этой темы в таблицах 12.1-12.3 приведем сводку всех свойств и методов класса `Array`.

Таблица 12.3. Динамические методы класса Array

Метод	Родитель	Описание
`Equals`	Класс `Object`	Описание и примеры даны в предыдущих главах.
`GetHashCode`	Класс `Object`	Описание и примеры даны в предыдущих главах.
`GetType`	Класс `Object`	Описание и примеры даны в предыдущих главах.
`ToString`	Класс `Object`	Описание и примеры даны в предыдущих главах.
`Clone`	Интерфейс *ICloneable*	Позволяет создать плоскую или глубокую копию массива. В первом случае создаются только элементы первого уровня, а ссылки указывают на те же самые объекты. Во втором случае копируются объекты на всех уровнях. Для массивов создается только плоская копия.
`CopyTo`	Интерфейс *ICollection*	Копируются все элементы одномерного массива в другой одномерный массив, начиная с заданного индекса: `col1.CopyTo(col2,0);`
`GetEnumerator`	Интерфейс *IEnumerable*	Стоит за спиной цикла `ForEach`
`GetLength`		Возвращает число элементов массива по указанному измерению. Описание и примеры даны в тексте главы.

`GetLowerBound`, `GetUpperBound`		Возвращает нижнюю и верхнюю границу по указанному измерению. Для массивов нижняя граница всегда равна нулю.
`GetValue`, `SetValue`		Возвращает или устанавливает значение элемента массива с указанными индексами.
`Initialize`		Может быть применен только к массивам значимого типа. Инициализирует элементы, вызывая соответствующий конструктор. Как правило, не используется в обычных программах.

Класс Object и массивы

Давайте обсудим допустимость *преобразований* между классами-массивами и классом `Object`. Понятно, что существует неявное *преобразование* объекта любого класса в объект класса `Object`, так что переменной типа `object` всегда можно присвоить переменную типа массив. Обратное такое *преобразование* также существует, но оно должно быть явным. Как всегда, при проведении явных *преобразований* не гарантируется успешность их выполнения.

В этой лекции и ранее обсуждался вопрос о создании универсальных процедур, которые могли бы работать с данными разных типов. Серьезный разговор об универсализации классов еще предстоит, сейчас же лишь напомню, что уже рассматривался такой прием, как перегрузка метода. У клиента, использующего перегруженный метод, создается впечатление, что он вызывает универсальный метод, работающий с аргументами разного типа. Создатель перегруженного метода должен, конечно, написать множество реализаций для поддержки такой универсальности. Другой уже обсуждавшийся прием состоит в том, что формальный аргумент метода принадлежит родительскому классу, тогда методу при вызове может быть передан аргумент любого из потомков.

Приведу в качестве примера многострадальную процедуру печати объектов, многократные варианты которой уже были рассмотрены. На этот раз формальный аргумент процедуры будет иметь тип `object` - прародителя всех классов. Разберем, как можно выяснить, что в процедуру передается массив, как определить его тип и работать с ним уже как с массивом, а не как с переменной класса `Object`. Вот текст этой процедуры, названной `PrintObject`:

```
public static void PrintObj(object A)
{
   Console.WriteLine("A.GetType()={0})", A.GetType());
   if (A.GetType()==typeof(System.Int32[]))
   {
      int[] temp;
      temp = (int[])A;
      for(int i = 0; i<temp.GetLength(0);i++)
         Console.Write("\t temp[{0}]={1}", i,temp[i]);
      Console.WriteLine();
   }
   else
      Console.WriteLine("Аргумент не является массивом целых");
}//PrintObject
```

Несколько замечаний к реализации.

Метод `GetType`, примененный к аргументу, возвращает не тип `Object`, а реальный тип фактического аргумента. Поэтому можно проанализировать, какому классу принадлежит объект, переданный в процедуру.

На каждой ветви разбора можно создать временный объект нужного типа и скопировать в него переданный аргумент. В данном примере рассматривается только одна ветвь, в которой создается целочисленный одномерный массив `temp`.

Заметьте, при присваивании значения переменной `temp` выполняется явное *преобразование* из класса `Object` в класс `Int[]`.

При наличии переменной `temp`, выполнение нужных действий над массивом не представляет никаких трудностей.

Приведу два примера вызова этой процедуры:

```
//работа с процедурой PrintObject
//Корректный и некорректный вызовы
Arrs.PrintObj(col1);
Arrs.PrintObj(col3);
```

Вот какой вывод порождается этим фрагментом кода:

Рис. 12.4. Результаты работы процедуры PrintObj

Массивы объектов

Во всех рассмотренных примерах этой главы нам встречались массивы, элементы которых имели только простые значимые типы. В реальных программах *массивы объектов* и других ссылочных типов встречаются не менее часто. Каков бы ни был тип элементов, большой разницы при работе с массивами нет. Но один важный нюанс все же есть, и его стоит отметить. Он связан с инициализацией элементов по умолчанию. Уже говорилось о том, что компилятор не следит за инициализацией элементов массива и доверяет инициализации, выполненной конструктором массива по умолчанию. Но для массивов ссылочного типа инициализация по умолчанию присваивает ссылкам значение `Null`. Это означает, что создаются только ссылки, но не сами объекты. По этой причине, пока не будет проведена настоящая инициализация с созданием объектов и заданием ссылок на конкретные объекты, работать с массивом ссылочного типа будет невозможно.

Рассмотрим детали этой проблемы на примере. Определим достаточно простой и интуитивно понятный класс, названный `Winners`, свойства которого задают имя победителя и его премию, а методы позволяют установить размер премии для каждого победителя и распечатать его свойства. Приведу код, описывающий этот класс:

```
/// <summary>
/// Класс победителей с именем и премией
/// </summary>
public class Winners
{
   //поля класса
   string name;
   int price;
   //статическое или динамическое поле rnd?
   //static Random rnd = new Random();
   Random rnd = new Random();
   // динамические методы
   public void SetVals(string name)
```

```
   {
      this.name = name;
      this.price = rnd.Next(5,10)* 1000;
   }//SetVals
   public void PrintWinner(Winners win)
   {
      Console.WriteLine("Имя победителя: {0}," +
         " его премия - {1}", win.name, win.price);
   }//PrintWinner
}//class Winners
```

Коротко прокомментирую этот текст.

1. Свойство `name` описывает имя победителя, а свойство `price` - величину его премии.
2. Свойство `rnd` необходимо при работе со случайными числами.
3. Метод `SetVals` выполняет инициализацию. Он присваивает полю name значение, переданное в качестве аргумента, и полю `price` - случайное значение.
4. Метод `PrintWinner` - метод печати свойств класса. Без подобного метода не обходится ни один класс.
5. В классе появится еще один статический метод `InitAr`, но о нем скажу чуть позже.

Пусть теперь в одном из методов нашего тестирующего класса `Testing` предполагается работа с классом `Winners`, начинающаяся с описания победителей. Естественно, задается массив, элементы которого имеют тип `Winners`. Приведу начало тестирующего метода, в котором дано соответствующее объявление:

```
public void TestWinners()
{
   //массивы объектов
   int nwin = 3;
   Winners[] wins = new Winners[nwin];
   string[] winames = {"Т. Хоар", "Н. Вирт", "Э. Дейкстра"};
```

В результате создан массив `wins`, состоящий из объектов класса `Winners`. Что произойдет, если попытаться задать значения полей объектов, вызвав специально созданный для этих целей метод `SetVals`? Рассмотрим фрагмент кода, осуществляющий этот вызов:

```
//создание значений элементов массива
         for(int i=0; i < wins.Length; i++)
            wins[i].SetVals(winames[i]);
```

На этапе выполнения будет сгенерировано исключение - нулевая ссылка. Причина понятна: хотя массив `wins` и создан, но это массив ссылок, имеющих значение `null`. Сами объекты, на которые должны указывать ссылки, не создаются в момент объявления массива ссылочного типа. Их нужно создавать явно. Ситуация аналогична объявлению массива массивов. И там необходим явный вызов конструктора для создания каждого массива на внутреннем уровне.

Как же создавать эти объекты? Конечно, можно возложить эту обязанность на пользователя, объявившего массив `wins`, - пусть он и создаст экземпляры для каждого элемента массива. Правильнее все-таки иметь в классе соответствующий метод. Метод должен быть статическим, чтобы его можно было вызывать еще до того, как созданы экземпляры класса, поскольку метод предназначен для создания этих самых экземпляров. Так в нашем классе появился статический метод `InitAr`:

```
//статический метод
public static Winners[] InitAr(Winners[] Winar)
{
   for(int i=0; i < Winar.Length; i++)
      Winar[i] = new Winners();
   return(Winar);
}//InitAr
```

Методу передается массив объектов, возможно, с нулевыми ссылками. Он возвращает тот же массив, но уже с явно определенными ссылками на реально созданные объекты. Теперь достаточно вызвать этот метод, после чего можно спокойно вызывать и метод `SetVals`. Вот как выглядит правильная последовательность вызовов методов класса `Winners`:

```
Winners.InitAr(wins);
      //создание значений элементов массива
      for(int i=0; i < wins.Length; i++)
         wins[i].SetVals(winames[i]);
      //печать значений элементов массива
      for(int i=0; i < wins.Length; i++)
         wins[i].PrintWinner(wins[i]);
   }//TestWinners
```

Теперь все корректно, массивы создаются, элементы заполняются нужными значениями, их можно распечатать:

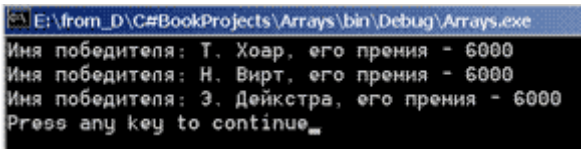

Рис. 12.5. Печать элементов массива wins

Обратите внимание, что всем победителям назначена одна и та же премия. Хотя понятно, что дело в программной ошибке, но в ней можно видеть и знак свыше. Коль скоро для победителей выбраны такие имена, почитаемые всеми программистами, то негоже пытаться расставить их по ранжиру даже в примере.

Что же касается ошибки, то она связана с тем, что в данном случае свойство rnd следует сделать статическим, чтобы оно было одно на все экземпляры класса. В тексте описания варианта класса приведены оба варианта объявления свойства, один из которых закомментирован.

Массивы. Семантика присваивания

Преобразования между классами массивов и родительскими классами `Array` и `Object` уже рассматривались. А существуют ли другие *преобразования* между классами массивов? Что происходит при присваивании `x=e`; (передаче аргументов в процедуру), если `x` и `e` - это массивы разных классов? Возможно ли присваивание? Ответ на этот вопрос положительный, хотя накладываются довольно жесткие ограничения на условия, когда такие *преобразования* допустимы. Известно, например, что между классами `Int` и `Object` существуют взаимные *преобразования* - в одну сторону явное, в другую неявное. А вот между классами `Int[]` и `Object[]` нет ни явных, ни неявных *преобразований*. С другой стороны, такое *преобразование* существует между классами `String[]` и `Object[]`. В чем же тут дело, и где логика? Запомните, главное ограничение на возможность таких *преобразований* состоит в том, что элементы массивов должны иметь ссылочный тип. А теперь притянем сюда логику. Крайне желательно обеспечить возможность проведения *преобразований* между массивами, элементы которых

принадлежат одному семейству классов, связанных *отношением наследования*. Такая возможность и была реализована. А вот для массивов с элементами значимых типов подобную же возможность не захотели или не смогли реализовать.

Сформулируем теперь точные правила, справедливые для присваивания и передачи аргументов в процедуру. Для того, чтобы было возможным неявное *преобразование* массива с элементами класса S в массив с элементами класса `T`, необходимо выполнение следующих условий:

- классы `S` и `T` должны быть ссылочного типа;
- размерности массивов должны совпадать;
- должно существовать неявное *преобразование* элементов класса `S` в элементы класса `T`.

Заметьте, если `S` - это родительский класс, а `T` - его потомок, то для массивов одной размерности остальные условия выполняются. Вернемся теперь к примеру с классами `Int[]`, `String[]` и `Object[]`. Класс `Int` не относится к ссылочным классам, и потому *преобразования* класса `Int[]` в `Object[]` не существует. Класс `string` является ссылочным классом и потомком класса `Object`, а потому существует неявное *преобразование* между классами `String[]` и `Object[]`.

Правило для явного *преобразования* можно сформулировать, например, так. Если существует неявное *преобразование* массива с элементами класса `S` в массив с элементами класса `T`, то существует явное *преобразование* массива с элементами класса `T` в массив с элементами класса `S`.

Для демонстрации *преобразований* между массивами написана еще одна процедура печати. Вот ее текст:

```
public static void PrintArObj(string name,object[] A)
{
   Console.WriteLine(name);
   foreach (object item in A )
      Console.Write("\t {0}", item);
   Console.WriteLine();
}//PrintArObj
```

Как видите, формальный аргумент этой процедуры принадлежит классу `Object[]`. При ее вызове фактическими аргументами могут быть массивы, удовлетворяющие выше указанным условиям. Вот пример кода, в котором вызывается эта процедура. В этом же фрагменте показаны и присваивания массива одного класса другому, где выполняются явные и неявные *преобразования* массивов.

```
public void TestMas()
{
   string[] winames = {"T. Хоар", "Н. Вирт", "Э. Дейкстра"};
   Arrs.PrintArObj("winames", winames);
   object[] cur = new object[5];
   cur = winames;
   Arrs.PrintArObj("cur", cur);
   winames = (string[])cur;
   Arrs.PrintArObj("winames", winames);
}//TestMas
```

Взгляните на результаты работы этой процедуры.

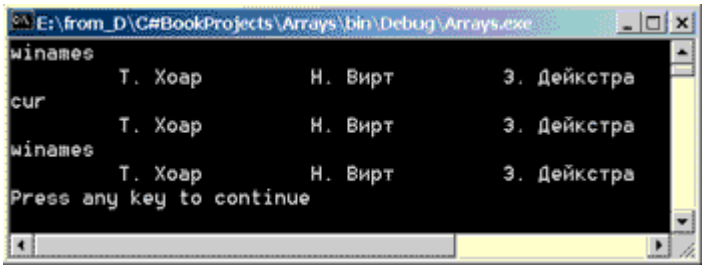

Рис. 12.6. Семантика присваивания и преобразования массивов

Приступая к описаниям массивов, я полагал, что 10 страниц одной лекции будет вполне достаточно. Оказалось, что массивы C# более интересны. Надеюсь, с этим согласятся и читатели.

13. Лекция: Символы и строки постоянной длины в C#

Общий взгляд

Строкам не повезло. По понятным причинам в первых языках программирования строковому типу уделялось гораздо меньше внимания, чем арифметическому типу или массивам. Поэтому в разных языках строки представлены по-разному и стандарт на строковый тип сложился относительно недавно. Когда говорят о строковом типе, то обычно различают тип, представляющий:

- отдельные символы, чаще всего, его называют типом `char`;
- *строки постоянной длины*, часто они представляются массивом символов;
- строки переменной длины - это, как правило, тип `string`, соответствующий современному представлению о строковом типе.

Символьный тип `char`, представляющий частный случай строк длиной `1`, полезен во многих задачах. Основные операции над строками - это разбор и сборка. При их выполнении приходится, чаще всего, доходить до каждого символа строки. В языке Паскаль, где был введен тип `char`, сам строковый тип рассматривался, как `char[]`-массив символов. При таком подходе получение `i`-го символа строки становится такой же простой операцией, как и получение `i`-го элемента массива. Следовательно, эффективно реализуются обычные операции над строками - определение вхождения одной строки в другую, выделение подстроки, замена символов строки. Однако заметьте, представление строки массивом символов хорошо только для *строк постоянной длины*. Массив не приспособлен к изменению его размеров, вставки или удалению символов (подстрок).

Наиболее часто используемым строковым типом является тип, обычно называемый `string`, который задает строки переменной длины. Над этим типом допускаются операции поиска вхождения одной строки в другую, операции вставки, замены и удаления подстрок.

Строки C++

В языке C++ есть все виды строк. Символьный тип **char** используется для задания отдельных символов. Для *строк постоянной длины* можно использовать массив символов - `char[]`. Особенностью, характерной для языка C++, точнее для языка C, является завершение строки символом с нулевым кодом. Строки, завершаемые нулем, называются обычно ***строками C***. Массив `char[]` задает *строку C* и потому должен иметь размер, по крайней мере, на единицу больше фактического размера строки. Вот пример объявления подобных строк в C++:

```
//Массивы и строки
   char strM1[] = "Hello, World!";
   char strM2[20] = "Yes";
```

Массив `strM1` состоит из 14 символов, массив `strM2` - из 20, но его четвертый символ имеет код `0`, сигнализирующий о фактическом конце строки.

Другой способ задания *строк C*, заканчивающихся нулем, состоит в использовании типизированного указателя - *char**.

```
//Строки, заданные указателем char*
   char* strPM1 ="Hello, World!";
   char* strPM2;
```

Два типа, `char[]` и *char**, допускают взаимные преобразования.

Не могу удержаться, чтобы не привести процедуру копирования строк, соответствующую духу и стилю C++:

```
void mycpy(char* p, const char* q)
{
   while(*p++ = *q++);
}
```

Эта процедура копирует содержимое строки `q` в строку `p`. В этой короткой программе, в которой, кроме условия цикла `while`, ничего больше нет, фактически используются многие средства языка C++ - разыменование указателей, адресная арифметика, присваивание как операция, завершение строки нулем, логическая интерпретация значений. Раз уж я привел эту программу, то поясню, как она работает. Вначале указатель `q` задает адрес начала строки, поэтому разыменование `*q` задает первый символ копируемой строки. Это значение присваивается первому символу строки `p`. Суффиксные операции `p++` и `q++` увеличивают значение указателей на единицу, но поскольку используется адресная арифметика, то в результате вычисляется адрес, задающий следующий символ соответствующих строк, и процесс копирования продолжается. При достижении последнего символа строки `q` - символа с кодом нуль - он также будет скопирован в строку `p`. Но в этот момент выражение присваивание впервые вернет в качестве значения результат `0`, который будет проинтерпретирован в условии цикла `while` как `false`, и цикл завершит свою работу. Строка будет скопирована.

Можно восхищаться этой короткой и эффективной программой, можно ругать ее за сложность восприятия. Трудно назвать ее интуитивно понятной. Но во многом все определяется вкусом и привычкой.

Тип string не является частью языка C++, но входит в библиотеку, определяемую стандартом языка. Стандартные библиотеки, по сути, являются продолжением языка. Тип (класс) `string` обеспечивает работу со строками переменной длины и поддерживает многие полезные операции над строками.

Строки C#

Давайте разберемся, как устроены строки C# и что взято из языка C++.

Класс char

В C# есть *символьный класс Char*, основанный на классе `System.Char` и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы - символьные литералы. Константу можно задавать:

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

Вот несколько примеров объявления символьных переменных и работы с ними:

```
public void TestChar()
{
   char ch1='A', ch2 ='\x5A', ch3='\u0058';
   char ch = new Char();
   int code; string s;
   ch = ch1;
```

```
   //преобразование символьного типа в тип int
   code = ch; ch1=(char) (code +1);
   //преобразование символьного типа в строку
   //s = ch;
   s = ch1.ToString()+ch2.ToString()+ch3.ToString();
   Console.WriteLine("s= {0}, ch= {1}, code = {2}",
      s, ch, code);
}//TestChar
```

Три символьные переменные инициализированы константами, значения которых заданы тремя разными способами. Переменная `ch` объявляется в объектном стиле, используя new и вызов конструктора класса. Тип `char`, как и все типы C#, является классом. Этот класс наследует свойства и методы класса `Object` и имеет большое число собственных методов.

Существуют ли преобразования между *классом char* и другими классами? Явные или неявные преобразования между *классами char* и `string` отсутствуют, но, благодаря методу `ToString`, переменные типа `char` стандартным образом преобразуются в тип `string`. Как отмечалось в лекции 3, существуют неявные преобразования типа `char` в целочисленные типы, начиная с типа `ushort`. Обратные преобразования целочисленных типов в тип `char` также существуют, но они уже явные.

В результате работы процедуры `TestChar` строка `s`, полученная сцеплением трех символов, преобразованных в строки, имеет значение `BZX`, переменная `ch` равна `A`, а ее код - переменная `code` - `65`.

Не раз отмечалось, что семантика присваивания справедлива при вызове методов и замене формальных аргументов на фактические. Приведу две процедуры, выполняющие взаимно-обратные операции - получение по коду символа и получение символа по его коду:

```
public int SayCode(char sym)
{
   return (sym);
}//SayCode

public char SaySym(object code)
{
   return ((char)((int)code));
}// SaySym
```

Как видите, в первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное. Ради универсальности она слегка усложнена. Формальный параметр имеет тип `Object`, что позволяет передавать ей в качестве аргумента код, заданный любым целочисленным типом. Платой за это является необходимость выполнять два явных преобразования.

Таблица 13.1. Статические методы и свойства класса Char

Метод	Описание
`GetNumericValue`	Возвращает численное значение символа, если он является цифрой, и (`-1`) в противном случае
`GetUnicodeCategory`	Все символы разделены на категории. Метод возвращает Unicode категорию символа. Ниже приведен пример
`IsControl`	Возвращает `true`, если символ является управляющим
`IsDigit`	Возвращает `true`, если символ является десятичной цифрой
`IsLetter`	Возвращает `true`, если символ является буквой

IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичной или шестнадцатиричной цифрой)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsSurrogate	Некоторые символы Unicode с кодом в интервале [0x1000, 0x10FFF] представляются двумя 16-битными "суррогатными" символами. Метод возвращает true, если символ является суррогатным
IsUpper	Возвращает true, если символ задан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
Parse	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру
MaxValue, MinValue	Свойства, возвращающие символы с максимальным и минимальным кодом. Возвращаемые символы не имеют видимого образа

Класс Char, как и все классы в C#, наследует свойства и методы родительского класса Object. Но у него есть и собственные методы и свойства, и их немало. Сводка этих методов приведена в таблице 13.1.

Большинство статических методов перегружены. Они могут применяться как к отдельному символу, так и к строке, для которой указывается номер символа для применения метода. Основную группу составляют методы Is, крайне полезные при разборе строки. Приведу примеры, в которых используются многие из перечисленных методов:

```
public void TestCharMethods()
{
   Console.WriteLine("Статические методы класса char:");
   char ch='a', ch1='1', lim =';', chc='\xA';
   double d1, d2;
   d1=char.GetNumericValue(ch); d2=char.GetNumericValue(ch1);
   Console.WriteLine("Метод GetNumericValue:");
   Console.WriteLine("sym 'a' - value {0}", d1);
   Console.WriteLine("sym '1' - value {0}", d2);
   System.Globalization.UnicodeCategory cat1, cat2;
      cat1 =char.GetUnicodeCategory(ch1);
      cat2 =char.GetUnicodeCategory(lim);
   Console.WriteLine("Метод GetUnicodeCategory:");
   Console.WriteLine("sym '1' - category {0}", cat1);
   Console.WriteLine("sym ';' - category {0}", cat2);
   Console.WriteLine("Метод IsControl:");
   Console.WriteLine("sym '\xA' - IsControl - {0}",
      char.IsControl(chc));
   Console.WriteLine("sym ';' - IsControl - {0}",
      char.IsControl(lim));
   Console.WriteLine("Метод IsSeparator:");
   Console.WriteLine("sym ' ' - IsSeparator - {0}",
      char.IsSeparator(' '));
   Console.WriteLine("sym ';' - IsSeparator - {0}",
```

```
      char.IsSeparator(lim));
   Console.WriteLine("Метод IsSurrogate:");
   Console.WriteLine("sym '\u10FF' - IsSurrogate - {0}",
      char.IsSurrogate('\u10FF'));
   Console.WriteLine("sym '\\' - IsSurrogate - {0}",
      char.IsSurrogate('\\'));
   string str = "\U00010F00";
   //Символы Unicode в интервале [0x10000,0x10FFF]
   //представляются двумя 16-битными суррогатными символами
   Console.WriteLine("str = {0}, str[0] = {1}", str, str[0]);
   Console.WriteLine("str[0] IsSurrogate - {0}",
      char.IsSurrogate(str, 0));
   Console.WriteLine("Метод IsWhiteSpace:");
   str ="пробелы, пробелы!" + "\xD" + "\xA" + "Всюду пробелы!";
   Console.WriteLine("sym '\xD ' - IsWhiteSpace - {0}",
      char.IsWhiteSpace('\xD'));
   Console.WriteLine("str: {0}", str);
   Console.WriteLine("и ее пробелы - символ 8 {0},символ 17 {1}",
      char.IsWhiteSpace(str,8), char.IsWhiteSpace(str,17));
   Console.WriteLine("Метод Parse:");
   str="A";
   ch = char.Parse(str);
   Console.WriteLine("str:{0}  char: {1}",str, ch);
   Console.WriteLine("Минимальное и максимальное значение:{0}, {1}",
      char.MinValue.ToString(), char.MaxValue.ToString());
   Console.WriteLine("Их коды: {0}, {1}",
      SayCode(char.MinValue), SayCode(char.MaxValue));
}//TestCharMethods
```

Результаты консольного вывода, порожденного выполнением метода, изображены на рис. 13.1.

Рис. 13.1. Вызовы статических методов класса char

Кроме статических методов, у *класса Char* есть и динамические. Большинство из них - это методы родительского класса `Object`, унаследованные и переопределенные в *классе Char*. Из собственных динамических методов стоит отметить метод `CompareTo`, позволяющий проводить сравнение символов. Он отличается от метода `Equal` тем, что

для несовпадающих символов выдает "расстояние" между символами в соответствии с их упорядоченностью в кодировке Unicode. Приведу пример:

```
public void testCompareChars()
{
   char ch1, ch2;
   int dif;
   Console.WriteLine("Метод CompareTo");
   ch1='A'; ch2= 'Z';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
   ch1='a'; ch2= 'A';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
   ch1='Я'; ch2= 'A';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
   ch1='A'; ch2= 'A';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
   ch1='A'; ch2= 'A';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
   ch1='Ё'; ch2= 'A';
   dif = ch1.CompareTo(ch2);
   Console.WriteLine("Расстояние между символами {0},
      {1} = {2}", ch1, ch2, dif);
}//TestCompareChars
```

Результаты сравнения изображены на рис. 13.2.

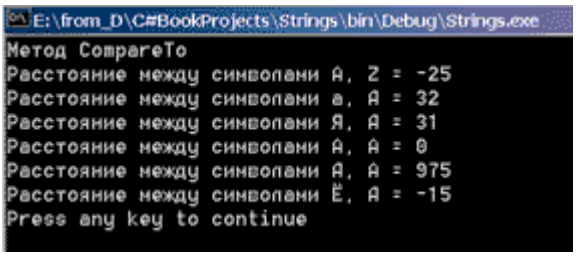

Рис. 13.2. Сравнение символов

Анализируя эти результаты, можно понять, что в кодировке Unicode как латиница, так и кириллица плотно упакованы. Исключение составляет буква Ё - заглавная и малая - они выпадают из плотной кодировки. Малые буквы в кодировке непосредственно следуют за заглавными буквами. Расстояние между алфавитами в кодировке довольно большое - русская буква А на 975 символов правее в кодировке, чем соответствующая буква в латинском алфавите.

Класс char[] - массив символов

В языке C# определен *класс Char[]*, и его можно использовать для представления *строк постоянной длины*, как это делается в C++. Более того, поскольку массивы в C# динамические, то расширяется класс задач, в которых можно использовать массивы символов для представления строк. Так что имеет смысл разобраться, насколько хорошо C# поддерживает работу с таким представлением строк.

Прежде всего, ответим на вопрос, задает ли массив символов C# *строку C*, заканчивающуюся нулем? Ответ: нет, не задает. Массив `char[]` - это обычный массив. Более того, его нельзя инициализировать строкой символов, как это разрешается в C++. Константа, задающая строку символов, принадлежит классу `String`, а в C# не определены взаимные преобразования между классами `String` и *Char[]*, даже явные. У класса `String` есть, правда, динамический метод `ToCharArray`, задающий подобное преобразование. Возможно также посимвольно передать содержимое переменной `string` в массив символов. Приведу пример:

```
public void TestCharArAndString()
{
   //массивы символов
   //char[] strM1 = "Hello, World!";
   //ошибка: нет преобразования класса string в класс char[]
   string hello = "Здравствуй, Мир!";
   char[] strM1 = hello.ToCharArray();
   PrintCharAr("strM1",strM1);
   //копирование подстроки
   char[] World = new char[3];
   Array.Copy(strM1,12,World,0,3);
   PrintCharAr("World",World);
   Console.WriteLine(CharArrayToString(World));
}//TestCharArAndString
```

Закомментированные операторы в начале этой процедуры показывают, что прямое присваивание строки массиву символов недопустимо. Однако метод `ToCharArray`, которым обладают строки, позволяет легко преодолеть эту трудность. Еще одну возможность преобразования строки в массив символов предоставляет статический метод `Copy` класса `Array`.

В нашем примере часть строки `strM1` копируется в массив `World`. По ходу дела в методе вызывается процедура `PrintCharAr` класса `Testing`, печатающая массив символов как строку. Вот ее текст:

```
void PrintCharAr(string name,char[] ar)
{
   Console.WriteLine(name);
   for(int i=0; i < ar.Length; i++)
      Console.Write(ar[i]);
   Console.WriteLine();
}//PrintCharAr
```

Метод `ToCharArray` позволяет преобразовать строку в массив символов. К сожалению, обратная операция не определена, поскольку метод `ToString`, которым, конечно же, обладают все объекты *класса Char[]*, печатает информацию о классе, а не содержимое массива. Ситуацию легко исправить, написав подходящую процедуру. Вот текст этой процедуры `CharArrayToString`, вызываемой в нашем тестирующем примере:

```
string CharArrayToString(char[] ar)
{
   string result="";
   for(int i = 0; i< ar.Length; i++) result += ar[i];
   return(result);
}//CharArrayToString
```

Класс Char[], как и всякий класс-массив в C#, является наследником не только класса `Object`, но и класса `Array`, и, следовательно, обладает всеми методами родительских классов, подробно рассмотренных в предыдущей главе. А есть ли у него специфические методы, которые позволяют выполнять операции над строками,

представленными массивами символов? Таких специальных операций нет. Но некоторые перегруженные методы класса `Array` можно рассматривать как операции над строками. Например, метод `Copy` дает возможность выделять и заменять подстроку в теле строки. Методы `IndexOf`, `LastIndexOf` позволяют определить индексы первого и последнего вхождения в строку некоторого символа. К сожалению, их нельзя использовать для более интересной операции - нахождения индекса вхождения подстроки в строку. При необходимости такую процедуру можно написать самому. Вот как она выглядит:

```
int IndexOfStr( char[]s1, char[] s2)
{
   //возвращает индекс первого вхождения подстроки s2 в
   //строку s1
   int i =0, j=0, n=s1.Length-s2.Length; bool found = false;
   while( (i<=n) && !found)
   {
      j = Array.IndexOf(s1,s2[0],i);
      if (j <= n)
      {
         found=true; int k = 0;
         while ((k < s2.Length)&& found)
         {
            found =char.Equals(s1[k+j],s2[k]); k++;
         }
      }
      i=j+1;
   }
   if(found) return(j); else return(-1);
}//IndexOfStr
```

В реализации используется метод `IndexOf` класса `Array`, позволяющий найти начало совпадения строк, после чего проверяется совпадение остальных символов. Реализованный здесь алгоритм является самым очевидным, но далеко не самым эффективным.

А теперь рассмотрим процедуру, в которой определяются индексы вхождения символов и подстрок в строку:

```
public void TestIndexSym()
{
   char[] str1, str2;
   str1 = "рококо".ToCharArray();
   //определение вхождения символа
   int find, lind;
   find= Array.IndexOf(str1,'о');
   lind = Array.LastIndexOf(str1,'о');
   Console.WriteLine("Индексы вхождения о в рококо:{0},{1};
      ", find, lind);
   //определение вхождения подстроки
   str2 = "рок".ToCharArray();
   find = IndexOfStr(str1,str2);
   Console.WriteLine("Индекс первого вхождения рок в
      рококо:{0}", find);
   str2 = "око".ToCharArray();
   find = IndexOfStr(str1,str2);
   Console.WriteLine("Индекс первого вхождения око в
      рококо:{0}", find);
}//TestIndexSym
```

В этой процедуре вначале используются стандартные методы класса Array для определения индексов вхождения символа в строку, а затем созданный метод

`IndexOfStr` для определения индекса первого вхождения подстроки. Корректность работы метода проверяется на разных строках. Вот результаты ее работы.

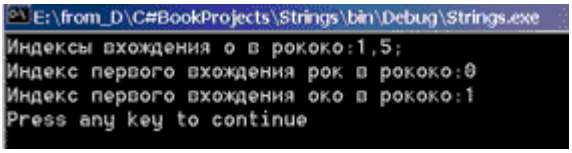

Рис. 13.3. Индексы вхождения подстроки в строку

Существует ли в C# тип char*

В языке C# указатели допускаются в блоках, отмеченных как небезопасные. Теоретически в таких блоках можно объявить переменную *типа Char**, но все равно не удастся написать столь же короткую, как в C++, процедуру копирования строк. Правильно считать, что в C# строки *типа char** использовать не рекомендуется.

14. Лекция: Строки C#. Классы String и StringBuilder

Класс String

В предыдущей лекции мы говорили о символьном типе char и строках постоянной длины, задаваемых массивом символов. Основным типом при работе со строками является тип *string*, задающий строки переменной длины. *Класс String* в языке C# относится к ссылочным типам. Над строками - объектами этого класса - определен широкий набор операций, соответствующий современному представлению о том, как должен быть устроен строковый тип.

Объявление строк. Конструкторы класса string

Объекты *класса String* объявляются как все прочие объекты простых типов - с явной или отложенной инициализацией, с явным или неявным вызовом конструктора класса. Чаще всего, при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковой константой. Но у класса `Sring` достаточно много конструкторов. Они позволяют сконструировать строку из:

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

Некоторым конструкторам в качестве параметра инициализации можно передать строку, заданную типом `char*`. Но все это небезопасно, и подобные примеры приводиться и обсуждаться не будут. Приведу примеры объявления строк с вызовом разных конструкторов:

```
public void TestDeclStrings()
{
   //конструкторы
   string world = "Мир";
   //string s1 = new string("s1");
   //string s2 = new string();
   string sssss = new string('s',5);
   char[] yes = "Yes".ToCharArray();
   string stryes = new string(yes);
   string strye = new string(yes,0,2);
   Console.WriteLine("world = {0}; sssss={1}; stryes={2};"+
      " strye= {3}", world, sssss, stryes, strye);
}
```

Объект `world` создан без явного вызова конструктора, а объекты `sssss`, `stryes`, `strye` созданы разными конструкторами *класса String*.

Заметьте, не допускается явный вызов конструктора по умолчанию - конструктора без параметров. Нет также конструктора, которому в качестве аргумента можно передать обычную строковую константу. Соответствующие операторы в тексте закомментированы.

Операции над строками

Над строками определены следующие операции:

- присваивание (`=`);
- две операции проверки эквивалентности (`= =`) и (`!=`);
- конкатенация или сцепление строк (`+`);

- взятие индекса ([]).

Начну с присваивания, имеющего важную особенность. Поскольку string - это ссылочный тип, то в результате присваивания создается ссылка на константную строку, хранимую в "куче". С одной и той же строковой константой в "куче" может быть связано несколько переменных строкового типа. Но эти переменные не являются псевдонимами - разными именами одного и того же объекта. Дело в том, что строковые константы в "куче" не изменяются (о неизменяемости строкового типа будем далее говорить подробно), поэтому когда одна из переменных получает новое значение, она связывается с новым константным объектом в "куче". Остальные переменные сохраняют свои связи. Для программиста это означает, что семантика присваивания строк аналогична семантике значимого присваивания.

В отличие от других ссылочных типов операции, проверяющие эквивалентность, сравнивают значения строк, а не ссылки. Эти операции выполняются как над значимыми типами.

Бинарная операция "+" сцепляет две строки, приписывая вторую строку к хвосту первой.

Возможность взятия индекса при работе со строками отражает тот приятный факт, что строку можно рассматривать как массив и получать без труда каждый ее символ. Каждый символ строки имеет тип char, доступный только для чтения, но не для записи.

Вот пример, в котором над строками выполняются данные операции:

```
public void TestOpers()
{
   //операции над строками
   string s1 ="ABC", s2 ="CDE";
   string s3 = s1+s2;
   bool b1 = (s1==s2);
   char ch1 = s1[0], ch2=s2[0];
   Console.WriteLine("s1={0}, s2={1}, b1={2}," +
      "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
   s2 = s1;
   b1 = (s1!=s2);
   ch2 = s2[0];
   Console.WriteLine("s1={0}, s2={1}, b1={2}," +
      "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
   //Неизменяемые значения
   s1= "Zenon";
   //s1[0]='L';
}
```

Строковые константы

Без констант не обойтись. В C# существуют два вида строковых констант:

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

В обычных константах некоторые символы интерпретируются особым образом. Связано это прежде всего с тем, что необходимо уметь задавать в строке непечатаемые символы, такие, как, например, символ табуляции. Возникает необходимость задавать символы их кодом - в виде escape-последовательностей. Для всех этих целей используется комбинация символов, начинающаяся символом "\" - обратная косая

черта. Так, пары символов: "\n", "\t", "\\", "\"" задают соответственно символ перехода на новую строку, символ табуляции, сам символ обратной косой черты, символ кавычки, вставляемый в строку, но не сигнализирующий о ее окончании. Комбинация "\xNNNN" задает символ, определяемый шестнадцатеричным кодом NNNN. Хотя такое решение возникающих проблем совершенно естественно, иногда возникают неудобства: например, при задании констант, определяющих путь к файлу, приходится каждый раз удваивать символ обратной косой черты. Это одна из причин, по которой появились *@-константы*.

В *@-константах* все символы трактуются в полном соответствии с их изображением. Поэтому путь к файлу лучше задавать *@-константой*. Единственная проблема в таких случаях: как задать символ кавычки, чтобы он не воспринимался как конец самой константы. Решением является удвоение символа. Вот соответствующие примеры:

```
//Два вида констант
s1= "\x50";
s2=@"\x50""";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1 = "c:\\c#book\\ch5\\chapter5.doc";
s2 = @"c:\c#book\ch5\chapter5.doc";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1= "\"A\"";
s2=@"""A""";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
```

Взгляните на результаты работы приведенных фрагментов кода, полученные при вызове процедур TestDeclStrings и TestOpers.

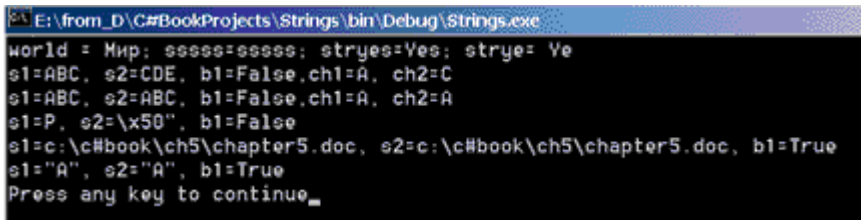

Рис. 14.1. Объявления, константы и операции над объектами string

Неизменяемый класс string

В языке C# существует понятие ***неизменяемый (immutable) класс***. Для такого класса невозможно изменить значение объекта при вызове его методов. Динамические методы могут создавать новый объект, но не могут изменить значение существующего объекта.

К таким *неизменяемым классам* относится и *класс String*. Ни один из методов этого класса не меняет значения существующих объектов. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Невозможность изменять значения строк касается не только методов. Аналогично, при работе со строкой как с массивом разрешено только чтение отдельных символов, но не их замена. Оператор присваивания из нашего последнего примера, в котором делается попытка изменить первый символ строки, недопустим, а потому закомментирован.

```
//Неизменяемые значения
        s1= "Zenon"; ch1 = s1[0];
//s1[0]='L';
```

Статические свойства и методы класса String

Таблица 14.1. Статические методы и свойства класса String	
Метод	**Описание**
`Empty`	Возвращается пустая строка. Свойство со статусом `read only`
`Compare`	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать или не учитывать регистр, особенности национального форматирования дат, чисел и т.д.
`CompareOrdinal`	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
`Concat`	Конкатенация строк. Метод перегружен, допускает сцепление произвольного числа строк
`Copy`	Создается копия строки
Format	Выполняет форматирование в соответствии с заданными спецификациями *формата*. Ниже приведено более полное описание метода
`Intern`, `IsIntern`	Отыскивается и возвращается ссылка на строку, если таковая уже хранится во внутреннем пуле данных. Если же строки нет, то первый из методов добавляет строку во внутренний пул, второй - возвращает `null`. Методы применяются обычно тогда, когда строка создается с использованием построителя строк - класса *StringBuilder*
Join	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом *Join*, является обратной к операции, заданной методом *Split*. Последний является динамическим методом и, используя разделители, осуществляет разделение строки на элементы

Метод Format

Метод Format в наших примерах встречался многократно. Всякий раз, когда выполнялся вывод результатов на консоль, неявно вызывался и *метод Format*. Рассмотрим оператор печати:

```
Console.WriteLine("s1={0}, s2={1}", s1,s2);
```

Здесь строка, задающая первый аргумент метода, помимо обычных символов, содержит *форматы*, заключенные в фигурные скобки. В данном примере используется простейший вид *формата* - он определяет объект, который должен быть подставлен в участок строки, занятый данным *форматом*. Помимо неявных вызовов, нередко возникает необходимость явного форматирования строки.

Давайте рассмотрим общий синтаксис как самого *метода Format*, так и используемых в нем *форматов*. *Метод Format*, как и большинство методов, является перегруженным и может вызываться с разным числом параметров. Первый необязательный параметр метода задает провайдера, определяющего национальные особенности, которые используются в процессе форматирования. В качестве такого параметра должен быть задан объект, реализующий интерфейс `System.IFormatProvider`. Если этот параметр не задан, то используется культура, заданная по умолчанию. Вот примеры двух реализаций этого метода:

```
public static string Format(string, object);
public static string Format(IFormatProvider, string, params object[]);
```

Параметр типа string задает форматируемую строку. Заданная строка содержит один или несколько *форматов*, они распознаются за счет окружающих *формат* фигурных скобок. Число *форматов*, вставленных в строку, определяет и число объектов, передаваемых при вызове *метода Format*. Каждый *формат* определяет форматирование объекта, на который он ссылается и который, после преобразования его в строку, будет подставлен в результирующую строку вместо *формата*. *Метод Format* в качестве результата возвращает переданную ему строку, где все спецификации *формата* заменены строками, полученными в результате форматирования объектов.

Общий синтаксис, специфицирующий *формат*, таков:

```
{N [,M [:<коды_форматирования>]]}
```

Обязательный параметр `N` задает индекс объекта, заменяющего *формат*. Можно считать, что методу всегда передается массив объектов, даже если фактически передан один объект. Индексация объектов начинается с нуля, как это принято в массивах. Второй параметр `M`, если он задан, определяет минимальную ширину поля, которое отводится строке, вставляемой вместо *формата*. Третий необязательный параметр задает коды форматирования, указывающие, как следует форматировать объект. Например, код `C` (`Currency`) говорит о том, что параметр должен форматироваться как валюта с учетом национальных особенностей представления. Код `P` (`Percent`) задает форматирование в виде процентов с точностью до сотой доли.

Все становится ясным, когда появляются соответствующие примеры. Вот они:

```
public void TestFormat()
{
   //метод Format
   int x=77;
   string s= string.Format("x={0}",x);
   Console.WriteLine(s + "\tx={0}",x);
   s= string.Format("Итого:{0,10} рублей",x);
   Console.WriteLine(s);
   s= string.Format("Итого:{0,6:######} рублей",x);
   Console.WriteLine(s);
   s= string.Format("Итого:{0:P} ",0.77);
   Console.WriteLine(s);
   s= string.Format("Итого:{0,4:C} ",77.77);
   Console.WriteLine(s);
   //Национальные особенности
   System.Globalization.CultureInfo ci =
      new System.Globalization.CultureInfo("en-US");
   s= string.Format(ci,"Итого:{0,4:C} ",77.77);
   Console.WriteLine(s);
}//TestFormat
```

Приведу некоторые комментарии к этой процедуре. Вначале демонстрируется, что и явный, и неявный вызовы *метода Format* дают один и тот же результат. В дальнейших примерах показано использование различных спецификаций *формата* с разным числом параметров и разными кодами форматирования. В частности, показан вывод процентов и валют. В последнем примере с валютами демонстрируется задание провайдером национальных особенностей. С этой целью создается объект класса `CultureInfo`, инициализированный так, чтобы он задавал особенности *форматирования*, принятые в США. Заметьте, класс `CultureInfo` наследует интерфейс `IFormatProvider`. Российские национальные особенности форматирования установлены по умолчанию. При необходимости их можно установить таким же образом, как это сделано для США, задав соответственно константу "`ru-RU`". Результаты работы метода показаны на рис. 14.2.

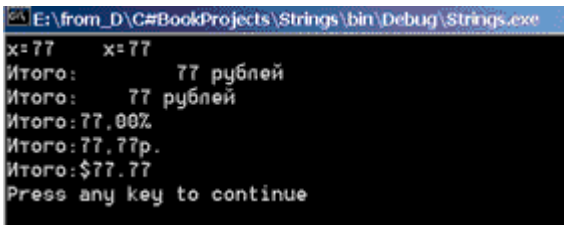

Рис. 14.2. Результаты работы метода Format

Методы Join и Split

Методы *Join* и *Split* выполняют над строкой текста взаимно обратные преобразования. *Динамический метод Split* позволяет осуществить разбор текста на элементы. *Статический метод Join* выполняет обратную операцию, собирая строку из элементов.

Заданный строкой текст зачастую представляет собой совокупность структурированных элементов - абзацев, предложений, слов, скобочных выражений и т.д. При работе с таким текстом необходимо разделить его на элементы, пользуясь специальными разделителями элементов, - это могут быть пробелы, скобки, знаки препинания. Практически подобные задачи возникают постоянно при работе со структурированными текстами. Методы *Split* и *Join* облегчают решение этих задач.

Динамический метод Split, как обычно, перегружен. Наиболее часто используемая реализация имеет следующий синтаксис:

```
public string[] Split(params char[])
```

На вход методу *Split* передается один или несколько символов, интерпретируемых как разделители. Объект `string`, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода. Другая реализация позволяет ограничить число элементов возвращаемого массива.

Синтаксис *статического метода Join* таков:

```
public static string Join(string delimiters, string[] items )
```

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива `items`, между которыми вставляется строка разделителей `delimiters`. Как правило, строка `delimiters` состоит из одного символа, который и разделяет в результирующей строке элементы массива `items`; но в отдельных случаях ограничителем может быть строка из нескольких символов.

Рассмотрим примеры применения этих методов. В первом из них строка представляет сложноподчиненное предложение, которое разбивается на простые предложения. Во втором предложение разделяется на слова. Затем производится обратная сборка разобранного текста. Вот код соответствующей процедуры:

```
public void TestSplitAndJoin()
{
   string txt = "А это пшеница, которая в темном чулане
      хранится," +" в доме, который построил Джек!";
   Console.WriteLine("txt={0}", txt);
   Console.WriteLine("Разделение текста на простые предложения:");
   string[] SimpleSentences, Words;
   //размерность массивов SimpleSentences и Words
   //устанавливается автоматически в соответствии с
   //размерностью массива, возвращаемого методом Split
```

```
   SimpleSentences = txt.Split(',');
   for(int i=0;i< SimpleSentences.Length; i++)
      Console.WriteLine("SimpleSentences[{0}]= {1}",
         i, SimpleSentences[i]);
   string txtjoin = string.Join(",",SimpleSentences);
   Console.WriteLine("txtjoin={0}", txtjoin);
   Words = txt.Split(',', ' ');
   for(int i=0;i< Words.Length; i++)
      Console.WriteLine("Words[{0}]= {1}",i, Words[i]);
   txtjoin = string.Join(" ",Words);
   Console.WriteLine("txtjoin={0}", txtjoin);
}//TestSplitAndJoin
```

Результаты выполнения этой процедуры показаны на рис. 14.3.

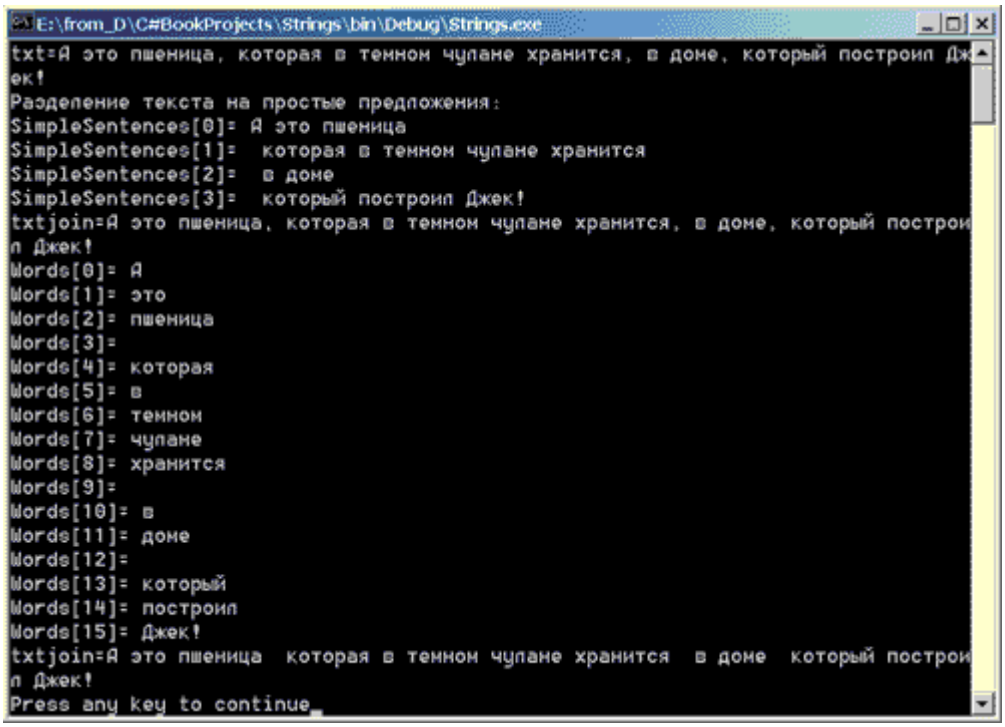

Рис. 14.3. Разбор и сборка строки текста

Обратите внимание, что методы *Split* и *Join* хорошо работают, когда при разборе используется только один разделитель. В этом случае сборка действительно является обратной операцией и позволяет восстановить исходную строку. Если же при разборе задается некоторое множество разделителей, то возникают две проблемы:

- невозможно при сборке восстановить строку в прежнем виде, поскольку не сохраняется информация о том, какой из разделителей был использован при разборе строки. Поэтому при сборке между элементами вставляется один разделитель, возможно, состоящий из нескольких символов;
- при разборе двух подряд идущих разделителей предполагается, что между ними находится пустое слово. Обратите внимание в тексте нашего примера, как и положено, после запятой следует пробел. При разборе текста на слова в качестве разделителей указаны символы пробела и запятой. По этой причине в массиве слов, полученном в результате разбора, имеются пустые слова.

Если при разборе предложения на слова использовать в качестве разделителя только пробел, то пустые слова не появятся, но запятая будет являться частью некоторых слов.

Как всегда, есть несколько способов справиться с проблемой. Один из них состоит в том, чтобы написать собственную реализацию этих функций, другой - в корректировке полученных результатов, третий - в использовании более мощного аппарата регулярных выражений, и о нем мы поговорим чуть позже.

Динамические методы класса String

Операции, разрешенные над строками в C#, разнообразны. Методы этого класса позволяют выполнять вставку, удаление, замену, поиск вхождения подстроки в строку. *Класс String* наследует методы класса `Object`, частично их переопределяя. *Класс String* наследует и, следовательно, реализует методы четырех интерфейсов: `IComparable`, `ICloneable`, `IConvertible`, `IEnumerable`. Три из них уже рассматривались при описании классов-массивов.

Рассмотрим наиболее характерные методы при работе со строками.

Сводка методов, приведенная в таблице 14.2, дает достаточно полную картину широких возможностей, имеющихся при работе со строками в C#. Следует помнить, что *класс string* является *неизменяемым*. Поэтому `Replace`, `Insert` и другие методы представляют собой функции, возвращающие новую строку в качестве результата и не изменяющие строку, вызвавшую метод.

Таблица 14.2. Динамические методы и свойства класса String

Метод	Описание
`Insert`	Вставляет подстроку в заданную позицию
`Remove`	Удаляет подстроку в заданной позиции
`Replace`	Заменяет подстроку в заданной позиции на новую подстроку
`Substring`	Выделяет подстроку в заданной позиции
`IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny`	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
`StartsWith, EndsWith`	Возвращается `true` или `false`, в зависимости от того, начинается или заканчивается строка заданной подстрокой
`PadLeft, PadRight`	Выполняет набивку нужным числом пробелов в начале и в конце строки
`Trim, TrimStart, TrimEnd`	Обратные операции к методам `Pad`. Удаляются пробелы в начале и в конце строки, или только с одного ее конца
`ToCharArray`	Преобразование строки в массив символов

Класс StringBuilder - построитель строк

Класс string не разрешает изменять существующие объекты. *Строковый класс StringBuilder* позволяет компенсировать этот недостаток. Этот класс принадлежит к изменяемым классам и его можно найти в пространстве имен `System.Text`. Рассмотрим класс *StringBuilder* подробнее.

Объявление строк. Конструкторы класса StringBuilder

Объекты этого класса объявляются с явным вызовом конструктора класса. Поскольку специальных констант этого типа не существует, то вызов конструктора для

инициализации объекта просто необходим. Конструктор класса перегружен, и наряду с конструктором без параметров, создающим пустую строку, имеется набор конструкторов, которым можно передать две группы параметров. Первая группа позволяет задать строку или подстроку, значением которой будет инициализироваться создаваемый объект класса *StringBuilder*. Вторая группа параметров позволяет задать ***емкость объекта*** - объем памяти, отводимой данному экземпляру класса *StringBuilder*. Каждая из этих групп не является обязательной и может быть опущена. Примером может служить конструктор без параметров, который создает объект, инициализированный пустой строкой, и с некоторой *емкостью* , заданной по умолчанию, значение которой зависит от реализации. Приведу в качестве примера синтаксис трех конструкторов:

- `public StringBuilder (string str, int cap)`. Параметр `str` задает строку инициализации, `cap` - *емкость объекта*;
- `public StringBuilder (int curcap, int maxcap)`. Параметры `curcap` и `maxcap` задают начальную и максимальную *емкость объекта*;
- `public StringBuilder (string str, int start, int len, int cap)`. Параметры `str`, start, len задают строку инициализации, cap - *емкость объекта*.

Операции над строками

Над строками этого класса определены практически те же операции с той же семантикой, что и над строками *класса String*:

- присваивание (`=`);
- две операции проверки эквивалентности (`= =`) и (`!=`);
- взятие индекса (`[]`).

Операция конкатенации (`+`) не определена над строками класса *StringBuilder*, ее роль играет метод `Append`, дописывающий новую строку в хвост уже существующей.

Со строкой этого класса можно работать как с массивом, но, в отличие от *класса String*, здесь уже все делается как надо: допускается не только чтение отдельного символа, но и его изменение. Рассмотрим с небольшими модификациями наш старый пример:

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

Этот пример демонстрирует возможность выполнения над строками класса *StringBuilder* тех же операций, что и над строками *класса String*. В результате присваивания создается дополнительная ссылка на объект, операции проверки на эквивалентность работают со значениями строк, а не со ссылками на них. Конкатенацию можно заменить вызовом метода `Append`. Появляется новая возможность - изменять отдельные символы строки. (Для того чтобы имя класса *StringBuilder* стало доступным, в проект добавлено предложение `using System.Text`, ссылающееся на соответствующее пространство имен.)

Основные методы

У класса *StringBuilder* методов значительно меньше, чем у *класса String*. Это и понятно - класс создавался с целью дать возможность изменять значение строки. По этой причине у класса есть основные методы, позволяющие выполнять такие операции над строкой как вставка, удаление и замена подстрок, но нет методов, подобных поиску вхождения, которые можно выполнять над обычными строками. Технология работы обычно такова: конструируется строка класса *StringBuilder*; выполняются операции, требующие изменение значения; полученная строка преобразуется в строку *класса String*; над этой строкой выполняются операции, не требующие изменения значения строки. Давайте чуть более подробно рассмотрим основные методы класса *StringBuilder*:

- `public StringBuilder Append (<объект>)`. К строке, вызвавшей метод, присоединяется строка, полученная из объекта, который передан методу в качестве параметра. Метод перегружен и может принимать на входе объекты всех простых типов, начиная от `char` и `bool` до `string` и `long`. Поскольку объекты всех этих типов имеют метод `ToString`, всегда есть возможность преобразовать объект в строку, которая и присоединяется к исходной строке. В качестве результата возвращается ссылка на объект, вызвавший метод. Поскольку возвращаемую ссылку ничему присваивать не нужно, то правильнее считать, что метод изменяет значение строки;
- `public StringBuilder Insert (int location,<объект>)`. Метод вставляет строку, полученную из объекта, в позицию, указанную параметром `location`. Метод `Append` является частным случаем метода `Insert`;
- `public StringBuilder Remove (int start, int len)`. Метод удаляет подстроку длины `len`, начинающуюся с позиции `start`;
- `public StringBuilder Replace (string str1,string str2)`. Все вхождения подстроки `str1` заменяются на строку `str2`;
- `public StringBuilder AppendFormat (<строка форматов>, <объекты>)`. Метод является комбинацией *метода Format класса String* и метода `Append`. Строка *форматов*, переданная методу, содержит только спецификации *форматов*. В соответствии с этими спецификациями находятся и форматируются объекты. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода `Remove`, все рассмотренные методы являются перегруженными. В их описании дана схема вызова метода, а не точный синтаксис перегруженных реализаций. Приведу примеры, чтобы продемонстрировать, как вызываются и как работают эти методы:

```
//Методы Insert, Append, AppendFormat
StringBuilder strbuild = new StringBuilder();
string str = "это  это не ";
strbuild.Append(str); strbuild.Append(true);
strbuild.Insert(4,false); strbuild.Insert(0,"2*2=5 - ");
Console.WriteLine(strbuild);
string txt = "А это пшеница, которая в темном чулане
   хранится," +" в доме, который построил Джек!";
```

```
StringBuilder txtbuild = new StringBuilder();
int num =1;
foreach(string sub in txt.Split(','))
{
   txtbuild.AppendFormat(" {0}: {1} ", num++,sub);
}
str = txtbuild.ToString();
Console.WriteLine(str);
```

В этом фрагменте кода конструируются две строки. Первая из них создается из строк и булевых значений `true` и `false`. Для конструирования используются методы `Insert` и `Append`. Вторая строка конструируется в цикле с применением метода `AppendFormat`. Результатом этого конструирования является строка, в которой простые предложения исходного текста пронумерованы.

Обратите внимание, что сконструированная вторая строка передается в обычную строку *класса String*. Никаких проблем преобразования строк одного класса в другой класс не возникает, поскольку все объекты, в том числе, объекты класса *StringBuilder*, обладают по определению методом `ToString`.

Обратите внимание, как выглядят результаты работы.

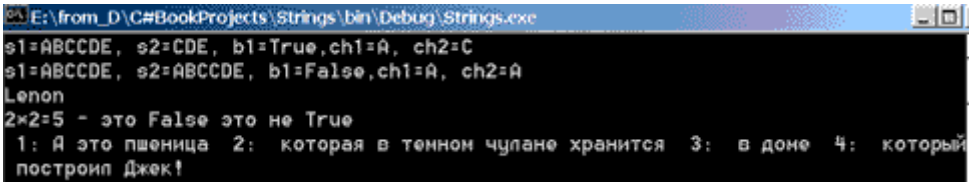

Рис. 14.4. Операции и методы класса StringBuilder

Емкость буфера

Каждый экземпляр строки класса *StringBuilder* имеет буфер, в котором хранится строка. Объем буфера - его емкость - может меняться в процессе работы со строкой. Объекты класса имеют две характеристики емкости - текущую и максимальную. В процессе работы текущая емкость изменяется, естественно, в пределах максимальной емкости, которая реально достаточно высока. Если размер строки увеличивается, то соответственно автоматически растет и текущая емкость. Если же размер строки уменьшается, то емкость буфера остается на том же уровне. По этой причине иногда разумно уменьшать емкость. Следует помнить, что попытка уменьшить емкость до величины, меньшей длины строки, приведет к ошибке.

У класса *StringBuilder* имеется 2 свойства и один метод, позволяющие анализировать и управлять емкостными свойствами буфера. Напомню, что этими характеристиками можно управлять также еще на этапе создания объекта, - для этого имеется соответствующий конструктор. Рассмотрим свойства и метод класса, связанные с емкостью буфера:

- свойство `Capacity` - возвращает или устанавливает текущую емкость буфера;
- свойство `MaxCapacity` - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса;
- метод `int EnsureCapacity (int capacity)` - позволяет уменьшить емкость буфера. Метод пытается вначале установить емкость, заданную параметром `capacity`; если это значение меньше размера хранимой строки, то емкость устанавливается такой, чтобы гарантировать размещение строки. Это число и возвращается в качестве результата работы метода.

Приведу код, в котором проводятся различные эксперименты с емкостью буфера:

```
//Емкость буфера
int curvol1 = txtbuild.Capacity;
int curvol2 = strbuild.Capacity;
int maxvol1 = txtbuild.MaxCapacity;
int maxvol2 = strbuild.MaxCapacity;
Console.WriteLine("curvol1= {0}",curvol1);
Console.WriteLine("curvol2= {0}",curvol2);
Console.WriteLine("maxvol1= {0}",maxvol1);
Console.WriteLine("maxvol2= {0}",maxvol2);
int sure1 = txtbuild.EnsureCapacity(100);
int sure2 = strbuild.EnsureCapacity(100);
Console.WriteLine("sure1= {0}",sure1);
Console.WriteLine("sure2= {0}",sure2);
curvol2 = strbuild.Capacity;
Console.WriteLine("curvol2= {0}",curvol2);
//ошибка! попытка установить емкость меньше длины строки
//strbuild.Capacity = 25;
strbuild.Capacity = 256; //так можно!
curvol2 = strbuild.Capacity;
Console.WriteLine("curvol2= {0}",curvol2);
//увеличим строку - емкость увеличится
int len = txtbuild.Length;
txtbuild.Append(txtbuild.ToString());
curvol1 = txtbuild.Capacity;
Console.WriteLine("curvol1= {0}",curvol1);
//уменьшим строку
txtbuild.Remove(len, len);
curvol1 = txtbuild.Capacity;
Console.WriteLine("curvol1= {0}",curvol1);
```

В этом фрагменте кода анализируются и изменятся емкостные свойства буфера двух объектов. Демонстрируется, как меняется емкость при увеличении и уменьшении размера строки. Результаты работы этого фрагмента кода показаны на рис. 14.5.

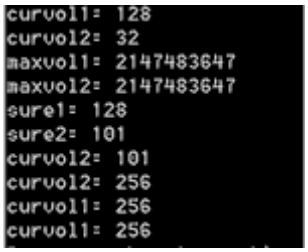

Рис. 14.5. Анализ емкостных свойств буфера

15. Лекция: Регулярные выражения

Пространство имен RegularExpression и классы регулярных выражений

Стандартный класс `String` позволяет выполнять над строками различные операции, в том числе поиск, замену, вставку и удаление подстрок. Существуют специальные операции, такие как `Join`, `Split`, которые облегчают разбор строки на элементы. Тем не менее, есть классы задач по обработке символьной информации, где стандартных возможностей явно не хватает. Чтобы облегчить решение подобных задач, в Net Framework встроен более мощный аппарат работы со строками, основанный на *регулярных выражениях*. Специальное *пространство имен RegularExpression*, содержит набор классов, обеспечивающих работу с *регулярными выражениями*. Все классы этого пространства доступны для C# и всех языков, использующих каркас Net Framework. В основе *регулярных выражений* лежит хорошая теория и хорошая практика их применения. Полное описание, как теоретических основ, так и практических особенностей применения этого аппарата в C#, требует отдельной книги. Придется ограничиться введением в эту интересную область работы со строками, не рассматривая подробно все классы, входящие в *пространство имен RegularExpression*.

Немного теории

Пусть `T = {a1, a2, ....an}` - алфавит символов. Словом в алфавите `T` называется последовательность записанных подряд символов, а длиной слова - число его символов. Пустое слово, не содержащее символов, обычно обозначается как `e`. Алфавит `T` можно рассматривать как множество всех слов длины `1`. Рассмотрим операцию конкатенации над множествами, так, что конкатенация алфавита `T` с самим собой дает множество всех слов длины `2`. Обозначается конкатенация `TT` как `T2`. Множество всех слов длины `k` обозначается - `Tk`, его можно рассматривать как `k`-кратную конкатенацию алфавита `T`. Множество всех непустых слов произвольной длины, полученное объединением всех множеств `Tk`, обозначается `T+`, а объединение этого множества с пустым словом называется *итерацией языка* и обозначается `T*`. *Итерация* описывает все возможные слова, которые можно построить в данном алфавите. Любое подмножество слов `L(T)`, содержащееся в `T*`, называется языком в алфавите `T`.

Определим класс языков, задаваемых *регулярными множествами*. *Регулярное множество* определяется рекурсивно следующими правилами:

- пустое множество, а также множество, содержащее пустое слово, и одноэлементные множества, содержащие символы алфавита, являются *регулярными* базисными множествами;
- если множества `P` и `Q` являются регулярными, то множества, построенные применением операций объединения, конкатенации и *итерации* - `P>>Q`, `PQ`, `P*`, `Q*` - тоже являются регулярными.

Регулярные выражения представляют удобный способ задания *регулярных множеств*. Аналогично множествам, они определяются рекурсивно:

- *регулярные* базисные выражения задаются символами и определяют соответствующие *регулярные* базисные множества, например, выражение `f` задает одноэлементное множество `{f}` при условии, что `f` - символ алфавита `T`;
- если `p` и `q` - *регулярные выражения*, то операции объединения, конкатенации и *итерации* - `p+q`, `pq`, `p*`, `q*` - являются *регулярными выражениями*, определяющими соответствующие *регулярные множества*.

По сути, *регулярные выражения* - это более простой и удобный способ записи *регулярных множеств* в виде обычной строки. Каждое *регулярное множество*, а, следовательно, и каждое *регулярное выражение* задает некоторый язык `L(T)` в алфавите `T`. Этот класс языков - достаточно мощный, с его помощью можно описать интересные языки, но устроены они довольно просто - их можно определить также с помощью простых грамматик, например, правосторонних грамматик. Более важно, что для любого *регулярного выражения* можно построить конечный автомат, который распознает, принадлежит ли заданное слово языку, порожденному *регулярным выражением*. На этом основана практическая ценность *регулярных выражений*.

С точки зрения практика *регулярное выражение* задает образец поиска. После чего можно проверить, удовлетворяет ли заданная строка или ее подстрока данному образцу. В языках программирования синтаксис *регулярного выражения* существенно обогащается, что дает возможность более просто задавать сложные образцы поиска. Такие синтаксические надстройки, хотя и не меняют сути *регулярных выражений*, крайне полезны для практиков, избавляя программиста от ненужных сложностей. (В Net Framework эти усложнения, на мой взгляд, чрезмерны. Выигрывая в мощности языка, проигрываем в простоте записи его выражений.)

Синтаксис регулярных выражений

Регулярное выражение на C# задается строковой константой. Это может быть обычная или `@`-константа. Чаще всего, следует использовать именно `@`-константу. Дело в том, что символ "`\`" широко применяется в *регулярных выражениях* как для записи escape-последовательностей, так и в других ситуациях. Обычные константы в таких случаях будут выдавать синтаксическую ошибку, а `@`-константы не выдают ошибок и корректно интерпретируют запись *регулярного выражения*.

Синтаксис *регулярного выражения* простой формулой не описать, здесь используются набор разнообразных средств:

- символы и escape-последовательности;
- символы операций и символы, обозначающие специальные классы множеств;
- имена *групп* и обратные ссылки;
- символы утверждений и другие средства.

Конечно, *регулярное выражение* может быть совсем простым, например, строка "`abc`" задает образец поиска, так что при вызове соответствующего метода будут разыскиваться одно или все вхождения подстроки "`abc`" в искомую строку. Но могут существовать и очень сложно устроенные *регулярные выражения*. Приведу таблицу, (15.1) в которой дается интерпретация символов в соответствии с их делением на *группы*. Таблица не полна, в ней отражаются не все *группы*, а описание *группы* не содержит всех символов. Она позволяет дать общее представление о синтаксисе, которое будет дополнено большим числом примеров. За деталями придется обращаться к справочной системе, которая, к сожалению, далеко не идеальна для данного раздела.

Повторяю, данная таблица не полна. В ней не отражены, например, такие категории, как подстановки, обратные ссылки, утверждения.

Для приведенных категорий также не дан полный список возможных символов.

Таблица 15.1. Символы, используемые в регулярных выражениях

Символ	Интерпретация
	Категория: escape-последовательности
`\b`	При использовании его в квадратных скобках соответствует имволу "обратная косая черта" с кодом - `\u0008`

\t	Соответствует символу табуляции \u0009
\r	Соответствует символу возврата каретки \u000D
\n	Соответствует символу новой строки \u000A
\e	Соответствует символу escape \u001B
\040	Соответствует символу ASCII, заданному кодом до трех цифр в восьмеричной системе
\x20	Соответствует символу ASCII, заданному кодом из двух цифр в шестнадцатиричной системе
\u0020	Соответствует символу Unicode, заданному кодом из четырех цифр в шестнадцатиричной системе
	Категория: подмножества (классы) символов
.	Соответствует любому символу, за исключением символа конца строки
[aeiou]	Соответствует любому символу из множества, заданного в квадратных скобках
[^aeiou]	Отрицание. Соответствует любому символу, за исключением символов, заданных в квадратных скобках
[0-9a-fA-F]	Задание диапазона символов, упорядоченных по коду. Так, 0-9 задает любую цифру
\p{name}	Соответствует любому символу, заданному множеству с именем name, например, имя Ll задает множество букв латиницы в нижнем регистре. Поскольку все символы разбиты на подмножества, задаваемые категорией Unicode, то в качестве имени можно задавать имя категории
\P{name}	Отрицание. Большая буква всегда задает отрицание множества, заданного малой буквой
\w	Множество символов, используемых при задании идентификаторов - большие и малые символы латиницы, цифры и знак подчеркивания
\s	Соответствует символам белого пробела
\d	Соответствует любому символу из множества цифр
	Категория: Операции (модификаторы)
*	*Итерация*. Задает ноль или более соответствий; например, \w* или
(abc)*.	Аналогично, {0,}
+	Положительная *итерация*. Задает одно или более соответствий; например, \w+ или (abc)+. Аналогично, {1,}
?	Задает ноль или одно соответствие; например, \w? или (abc)?. Аналогично, {0,1}
{n}	Задает в точности n соответствий; например, \w{2}
{n,}	Задает, по меньшей мере, n соответствий; например, (abc){2,}
{n,m}	Задает, по меньшей мере, n, но не более m соответствий; например, (abc){2,5}
	Категория: Группирование
(?<Name>)	При обнаружении соответствия выражению, заданному в круглых скобках, создается именованная *группа*, которой дается имя Name. Например, (?<tel>\d{7}). При обнаружении последовательности из семи цифр будет создана *группа* с именем tel
()	Круглые скобки разбивают *регулярное выражение* на *группы*. Для каждого подвыражения, заключенного в круглые скобки, создается *группа*, автоматически получающая номер. Номера следуют в обратном порядке, поэтому полному *регулярному выражению* соответствует *группа* с номером 0
(?imnsx)	Включает или выключает в *группе* любую из пяти возможных опций. Для выключения опции перед ней ставится знак минус. Например, (?i-s:)

	включает опцию `i`, задающую нечувствительность к регистру, и выключает опцию `s` - статус single-line

Знакомство с классами пространства RegularExpressions

В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.

Класс Regex

Это основной класс, всегда создаваемый при работе с *регулярными выражениями*. Объекты этого класса определяют *регулярные выражения*. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая *регулярное выражение*. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению `RegexOptions` и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать *регулярное выражение*. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Рассмотрим четыре основных метода *класса Regex*.

Метод Match запускает поиск соответствия. В качестве параметра методу передается строка поиска, где разыскивается первая подстрока, которая удовлетворяет образцу, заданному *регулярным выражением*.В качестве результата метод возвращает объект *класса Match*, описывающий результат поиска. При успешном поиске свойства объекта будут содержать информацию о найденной подстроке.

Метод `Matches` позволяет разыскать все вхождения, то есть все подстроки, удовлетворяющие образцу. У алгоритма поиска есть важная особенность - разыскиваются непересекающиеся вхождения подстрок. Можно считать, что метод `Matches` многократно запускает *метод Match*, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск. В качестве результата возвращается объект *MatchCollection*, представляющий коллекцию объектов *Match*.

Метод `NextMatch` запускает новый поиск, начиная с того места, на котором остановился предыдущий поиск.

Метод `Split` является обобщением метода `Split` класса `String`. Он позволяет, используя образец, разделить искомую строку на элементы. Поскольку образец может быть устроен сложнее, чем простое множество разделителей, то метод `Split` *класса Regex* эффективнее, чем его аналог класса `String`.

Классы Match и MatchCollection

Как уже говорилось, объекты этих классов создаются автоматически при вызове *методов Match* и `Matches`. *Коллекция MatchCollection*, как и все коллекции, позволяет получить доступ к каждому ее элементу - объекту *Match*. Можно, конечно, организовать цикл `for each` для последовательного доступа ко всем элементам коллекции.

Класс Match является непосредственным наследником *класса Group*, который, в свою очередь, является наследником класса `Capture`. При работе с объектами *класса Match* наибольший интерес представляют не столько методы класса, сколько его свойства, большая часть которых унаследована от родительских классов. Рассмотрим основные свойства:

- свойства `Index`, `Length` и `Value` наследованы от прародителя `Capture`. Они описывают найденную подстроку- индекс начала подстроки в искомой строке, длину подстроки и ее значение;
- свойство `Groups` *класса Match* возвращает коллекцию *групп* - объект `GroupCollection`, который позволяет работать с *группами*, созданными в процессе поиска соответствия;
- свойство `Captures`, наследованное от объекта `Group`, возвращает коллекцию `CaptureCollection`. Как видите, при работе с *регулярными выражениями* реально приходится создавать один объект *класса Regex*, объекты других классов автоматически появляются в процессе работы с объектами *Regex*.

Классы Group и GroupCollection

Коллекция `GroupCollection` возвращается при вызове свойства `Group объекта Match`. Имея эту коллекцию, можно добраться до каждого объекта `Group`, в нее входящего. *Класс Group* является наследником класса `Capture` и, одновременно, родителем *класса Match*. От своего родителя он наследует свойства `Index`, `Length` и `Value`, которые и передает своему потомку.

Давайте рассмотрим чуть более подробно, когда и как создаются *группы* в процессе поиска соответствия. Если внимательно проанализировать предыдущую таблицу, которая описывает символы, используемые в *регулярных выражениях*, в частности символы группирования, то можно понять несколько важных фактов, связанных с *группами*:

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

В заключение отмечу, что создание именованных *групп* крайне полезно при разборе строк, содержащих разнородную информацию. Примеры разбора подобных текстов будут даны.

Классы Capture и CaptureCollection

Коллекция `CaptureCollection` возвращается при вызове свойства `Captures` объектов *класса Group* и *Match*. *Класс Match* наследует это свойство у своего родителя - *класса Group*. Каждый объект `Capture`, входящий в коллекцию, характеризует соответствие, захваченное в процессе поиска, - соответствующую подстроку. Но поскольку свойства объекта `Capture` передаются по наследству его потомкам, то можно избежать непосредственной работы с объектами `Capture`. По крайней мере, в моих примерах не встретится работа с этим объектом, хотя "за кулисами" он непременно присутствует.

Перечисление RegexOptions

Объекты этого перечисления описывают опции, влияющие на то, как устанавливается соответствие. Обычно такой объект создается первым и передается конструктору объекта *класса Regex*. В вышеприведенной таблице, в разделе, посвященном символам группирования, говорится о том, что опции можно включать и выключать, распространяя, тем самым, их действие на участок шаблона, заданный соответствующей *группой*. Об одной из этих опций - `Compiled`, влияющей на

эффективность работы *регулярных выражений*, уже упоминалось. Об остальных говорить не буду - при необходимости можно посмотреть справку.

Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска *регулярные выражения*. В этом случае необходимо будет создать объект класса `RegexCompilationInfo` и передать ему информацию о *регулярных выражениях*, подлежащих компиляции, и о том, куда поместить оттранслированную программу. Дополнительно в таких ситуациях следует включить опцию `Compiled`. К сожалению, соответствующих примеров на эту тему не будет.

Примеры работы с регулярными выражениями

Полагаю, что примеры дополнят краткое описание возможностей *регулярных выражений* и позволят лучше понять, как с ними работать. Начну с функции `FindMatch`, которая производит поиск первого вхождения подстроки, соответствующей образцу:

```
string FindMatch(string str, string strpat)
{
   Regex pat = new Regex(strpat);
   Match match =pat.Match(str);
   string found = "";
   if (match.Success)
   {
      found =match.Value;
      Console.WriteLine("Строка ={0}\tОбразец={1}\
         tНайдено={2}", str,strpat,found);
   }
   return(found);
}//FindMatch
```

В качестве входных аргументов функции передается строка `str`, в которой ищется вхождение, и строка `strpat`, задающая образец - *регулярное выражение*. Функция возвращает найденную в результате поиска подстроку. Если соответствия нет, то возвращается пустая строка. Функция начинает свою работу с создания объекта `pat` *класса Regex*, конструктору которого передается образец поиска. Затем вызывается *метод Match* этого объекта, создающий объект `match` *класса Match*. Далее анализируются свойства этого объекта. Если соответствие обнаружено, то найденная подстрока возвращается в качестве результата, а соответствующая информация выводится на печать. (Чтобы спокойно работать с классами *регулярных выражений*, я не забыл добавить в начало проекта предложение: `using System.Text.RegularExpressions`.)

Поскольку запись *регулярных выражений* - вещь, привычная не для всех программистов, я приведу достаточно много примеров:

```
public void TestSinglePat()
{
   //поиск по образцу первого вхождения
   string str,strpat,found;
   Console.WriteLine("Поиск по образцу");
   //образец задает подстроку, начинающуюся с символа a,
   //далее идут буквы или цифры.
   str ="start"; strpat =@"a\w+";
   found = FindMatch(str,strpat);
   str ="fab77cd efg";
   found = FindMatch(str,strpat);
```

```
   //образец задает подстроку,начинающуюся с символа a,
   //заканчивающуюся f с возможными символами b и d в середине
   strpat = "a(b|d)*f";  str = "fabadddbdf";
   found = FindMatch(str,strpat);
   //диапазоны и escape-символы
   strpat = "[X-Z]+"; str = "aXYb";
   found = FindMatch(str,strpat);
   strpat = @"\u0058Y\x5A"; str = "aXYZb";
   found = FindMatch(str,strpat);
}//TestSinglePat
```

Некоторые комментарии к этой процедуре.

Регулярные выражения задаются `@`-константами, описанными в лекции 14. Здесь они как нельзя кстати.

В первом образце используется последовательность символов `\w+`, обозначающая, как следует из таблицы 15.1, непустую последовательность латиницы и цифр. В совокупности образец задает подстроку, начинающуюся символом `a`, за которым следуют буквы или цифры (хотя бы одна). Этот образец применяется к двум различным строкам.

В следующем образце используется символ `*` для обозначения *итерации*. В целом *регулярное выражение* задает строки, начинающиеся с символа a и заканчивающиеся символом `f`, между которыми находится возможно пустая последовательность символов из `b` и `d`.

Последующие два образца демонстрируют использование диапазонов и escape-последовательностей для представления символов, заданных кодами (в Unicode и шестнадцатиричной кодировке).

Взгляните на результаты, полученные при работе этой процедуры.

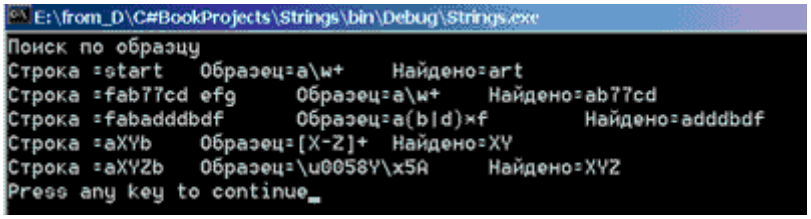

Рис. 15.1. Регулярные выражения. Поиск по образцу

Пример "чет и нечет"

Не всякий класс языков можно описать с помощью *регулярных выражений*. И даже тогда, когда такая возможность есть, могут потребоваться определенные усилия для корректной записи соответствующего *регулярного выражения*. Рассмотрим, например, язык `L1` в алфавите `T={0,1}`, которому принадлежат пустое слово и слова, содержащие четное число нулей и четное число единиц. В качестве другого примера рассмотрим язык `L2`, отличающийся от первого тем, что в нем число единиц нечетно. Оба языка можно задать *регулярными выражениями*, но корректная запись непроста и требует определенного навыка. Давайте запишем *регулярные выражения*, определяющие эти языки, и покажем, что C# справляется с проблемой их распознавания. Вот *регулярное выражение*, описывающее первый язык:

```
(00|11)*((01|10)(00|11)*(01|10)(00|11)*)*
```

Дадим содержательное описание этого языка. Слова языка представляют возможно пустую последовательность из пар одинаковых символов. Далее может идти последовательность, начинающаяся и заканчивающаяся парами различающихся символов, между которыми может стоять произвольное число пар одинаковых символов. Такая *группа* может повторяться многократно. *Регулярное выражение* короче и точнее передает описываемую структуру слов языка `L1`.

Язык `L2` описать теперь совсем просто. Его слова представляют собой единицу, окаймленную словами языка `L1`.

Прежде чем перейти к примеру распознавания слов языков `L1` и `L2`, приведу процедуру `FindMatches`, позволяющую найти все вхождения образца в заданный текст:

```
void FindMatches(string str, string strpat)
{
   Regex pat = new Regex(strpat);
   MatchCollection matchcol =pat.Matches(str);
   Console.WriteLine("Строка ={0}\tОбразец={1}",str,strpat);
   Console.WriteLine("Число совпадений ={0}",matchcol.Count);
   foreach(Match match in matchcol)
      Console.WriteLine("Index = {0} Value = {1}, Length ={2}",
         match.Index,match.Value, match.Length);
}//FindMatches
```

Входные аргументы у процедуры те же, что и у функции `FindMatch`, ищущей первое вхождение. Я не стал задавать выходных аргументов процедуры, ограничившись тем, что все результаты непосредственно выводятся на печать в самой процедуре. Выполнение процедуры, так же, как и в `FindMatch`, начинается с создания объекта `pat` *класса Regex*, конструктору которого передается *регулярное выражение*. Замечу, что *класс Regex*, так же, как и класс `String`, относится к неизменяемым (immutable) классам, поэтому для каждого нового образца нужно создавать новый объект `pat`.

В отличие от `FindMatch`, объект pat вызывает метод `Matches`, который определяет все вхождения подстрок, удовлетворяющих образцу, в заданный текст. Результатом выполнения метода `Matches` является автоматически создаваемый объект класса *MatchCollection*, хранящий коллекцию объектов уже известного нам *класса Match*, каждый из которых задает очередное вхождение. В процедуре используются свойства коллекции и ее элементов для получения в цикле по элементам коллекции нужных свойств - индекса очередного вхождения подстроки в строку, ее длины и значения.

Вот процедура, в которой многократно вызывается `FindMatches` для различных строк и образцов поиска:

```
public void TestMultiPat()
{
   //поиск по образцу всех вхождений
   string str,strpat,found;
   Console.WriteLine("Распознавание языков: чет и нечет");
   //четное число нулей и единиц
   strpat ="((00|11)*((01|10)(00|11)*(01|10)(00|11)*)*)";
   str = "0110111101101";
   FindMatches(str, strpat);
   //четное число нулей и нечетное единиц
   string strodd = strpat + "1" + strpat;
   FindMatches(str, strodd);
}//TestMultiPat
```

Коротко прокомментирую работу этой процедуры. Первые два примера связаны с распознаванием языков `L1` и `L2` (чет и нечет) - языков с четным числом единиц и нулей

в первом случае и нечетным числом единиц во втором. *Регулярные выражения*, описывающие эти языки, подробно рассматривались. В полном соответствии с теорией, константы задают эти выражения. На вход для распознавания подается строка из нулей и единиц. Для языка `L1` метод находит три соответствия. Первое из них задает максимально длинную подстроку, содержащую четное число нулей и единиц, и две пустые подстроки, по определению принадлежащие языку `L1`. Для языка `L2` находится одно соответствие - это сама входная строка. Взгляните на результаты распознавания.

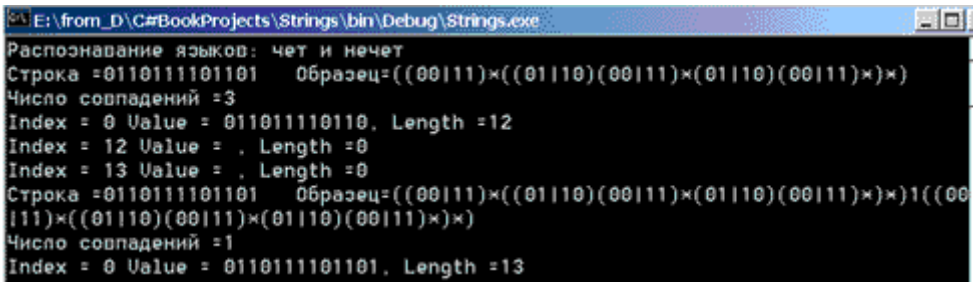

Рис. 15.2. Регулярные выражения. Пример "чет и нечет"

Пример "око и рококо"

Следующий образец в нашем примере позволяет прояснить некоторые особенности работы метода `Matches`. Сколько раз строка "око" входит в строку "рококо" - один или два? Все зависит от того, как считать. С точки зрения метода `Matches`, - один раз, поскольку он разыскивает непересекающиеся вхождения, начиная очередной поиск вхождения подстроки с того места, где закончилось предыдущее вхождение. Еще один пример на эту же тему работает с числовыми строками.

```
Console.WriteLine("око и рококо");
strpat="око"; str = "рококо";
FindMatches(str, strpat);
strpat="123";
str= "01234512361237812 39";
FindMatches(str, strpat);
```

На рис. 15.3 показаны результаты поисков.

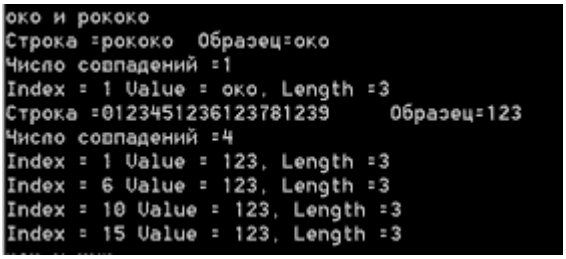

Рис. 15.3. Регулярные выражения. Пример "око и рококо"

Пример "кок и кук"

Этот пример на поиск множественных соответствий навеян словами песни Высоцкого, где говорится, что дикари не смогли распознать, где кок, а где Кук. Наше *регулярное выражение* также не распознает эти слова. Обратите внимание на точку в *регулярном выражении*, которая соответствует любому символу, за исключением символа конца строки. Все слова в строке поиска - кок, кук, кот и другие - будут удовлетворять шаблону, так что в результате поиска найдется множество соответствий.

```
Console.WriteLine("кок и кук");
strpat="(т|к).(т|к)";
str="кок тот кук тут как кот";
```

```
FindMatches(str, strpat);
```

Вот результаты работы этого фрагмента кода.

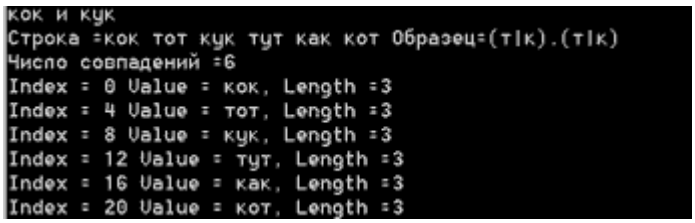

Рис. 15.4. Регулярные выражения. Пример "кок и кук"

Пример "обратные ссылки"

В этом примере рассматривается ранее упоминавшаяся, но не описанная возможность задания в *регулярном выражении* обратных ссылок. Можно ли описать с помощью *регулярных выражений* язык, в котором встречаются две подряд идущие одинаковые подстроки? Ответ на это вопрос отрицательный, поскольку грамматика такого языка должна быть контекстно-зависимой, и нужна память, чтобы хранить уже распознанные части строки. Аппарат *регулярных выражений*, предоставляемый классами *пространства RegularExpression*, тем не менее, позволяет решить эту задачу. Причина в том, что расширение стандартных *регулярных выражений* в Net Framework является не только синтаксическим. Содержательные расширения связаны с введением понятия *группы*, которой отводится память и дается имя. Это и дает возможность ссылаться на уже созданные *группы*, что и делает грамматику языка контекстно-зависимой. Ссылка на ранее полученную *группу* называется обратной ссылкой. Признаком обратной ссылки является пара символов "\k", после которой идет имя *группы*. Приведу пример:

```
Console.WriteLine("Ссылка назад - второе вхождение слова");
strpat = @"\s(?<word>\w+)\s\k'word'";
str = "I know know that, You know that!";
FindMatches(str, strpat);
```

Рассмотрим более подробно *регулярное выражение*, заданное строкой strpat. В *группе*, заданной скобочным выражением, после знака вопроса идет имя *группы* "word", взятое в угловые скобки. После имени *группы* идет шаблон, описывающий данную *группу*, в нашем примере шаблон задается произвольным идентификатором "\w+"(? так? кто кого задает?). В дальнейшем описании шаблона задается ссылка на *группу* с именем "word". Здесь имя *группы* заключено в одинарные кавычки. Поиск успешно справился с поставленной задачей, подтверждением чему являются результаты работы этого фрагмента кода.

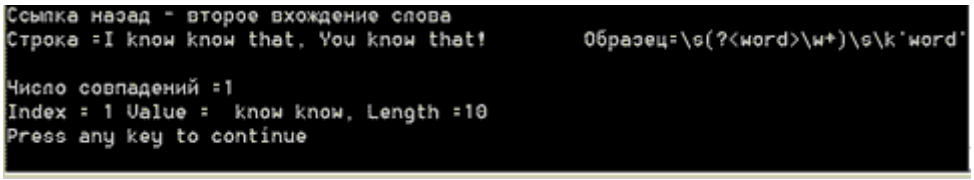

Рис. 15.5. Регулярные выражения. Пример "обратные ссылки"

Пример "Дом Джека"

Давайте вернемся к задаче разбора предложения на элементы. В классе string для этого имеется метод Split, который и решает поставленную задачу. Однако у этого метода есть существенный недостаток, - он не справляется с идущими подряд разделителями и создает для таких пар пустые слова. Метод Split *класса Regex* лишен этих недостатков, в качестве разделителей можно задавать любую пару символов,

произвольное число пробелов и другие комбинации символов. Повторим наш прежний пример:

```
public void TestParsing()
{
   string str,strpat;
   //разбор предложения - создание массива слов
   str = "А это пшеница, которая в темном чулане
      хранится," +" в доме, который    построил Джек!";
   strpat =" +|, ";
   Regex pat = new Regex(strpat);
   string[] words;
   words = pat.Split(str);
   int i=1;
   foreach(string word in words)
      Console.WriteLine("{0}: {1}",i++,word);
}//TestParsing
```

Регулярное выражение, заданное строкой `strpat`, определяет множество разделителей. Заметьте, в качестве разделителя задан пробел, повторенный сколь угодно много раз, либо пара символов - запятая и пробел. Разделители задаются *регулярными выражениями*. Метод `Split` применяется к объекту `pat` *класса Regex*. В качестве аргумента методу передается текст, подлежащий расщеплению. Вот как выглядит массив слов после применения метода `Split`.

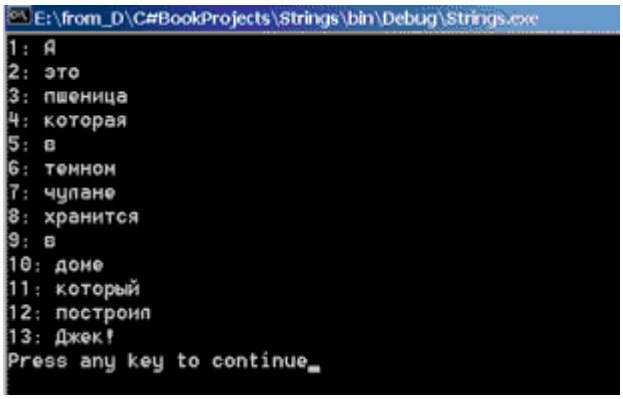

Рис. 15.6. Регулярные выражения. Пример "Дом Джека"

Пример "Атрибуты"

Как уже говорилось, *регулярные выражения* особенно хороши при разборе сложных текстов. Примерами таковых могут быть различные справочники, различные текстовые базы данных, весьма популярные теперь XML-документы, разбором которых приходится заниматься. В качестве заключительного примера рассмотрим структурированный документ, строки которого содержат некоторые атрибуты, например, телефон, адрес и e-mail. Структуру документа можно задавать по-разному; будем предполагать, что каждый атрибут задается парой "имя: Значение" Наша задача состоит в том, чтобы выделить из строки соответствующие атрибуты. В таких ситуациях *регулярное выражение* удобно задавать в виде *групп*, где каждая *группа* соответствует одному атрибуту. Приведу начальный фрагмент кода очередной тестирующей процедуры, в котором описываются строки текста и образцы поиска:

```
public void TestAttributes()
{
   string s1 = "tel: (831-2) 94-20-55 ";
   string s2 = "Адрес: 117926, Москва, 5-й Донской проезд,
      стр.10,кв.7";
   string s3 = "e-mail: Valentin.Berestov@tverorg.ru ";
   string s4 = s1+ s2 + s3;
```

```
   string s5 = s2 + s1 + s3;
   string pat1 = @"tel:\s(?<tel>\((\d|-)*\)\s(\d|-)+)\s";
   string pat2= @"Адрес:\s(?<addr>[0-9А-Яа-я \-\,\.]+)\s";
   string pat3 =@"e-mail:\s(?<em>[a-zA-Z.@]+)\s";
   string compat = pat1+pat2+pat3;
   string tel="", addr = "", em = "";
```

Строки `s4` и `s5` представляют строку разбираемого документа. Их две, для того чтобы можно было проводить эксперименты, когда атрибуты в документе представлены в произвольном порядке. Каждая из строк `pat1`, `pat2`, `pat3` задает одну именованную *группу* в *регулярном выражении*, имена *групп* - `tel`, `Адрес`, `e-mail` - даются в соответствии со смыслом атрибутов. Сами шаблоны подробно описывать не буду - сделаю лишь одно замечание. Например, шаблон телефона исходит из того, что номеру предшествует код, заключенный в круглые скобки. Поскольку сами скобки играют особую роль, то для задания скобки как символа используется пара - "`\(`". Это же касается и многих других символов, используемых в шаблонах, - точки, дефиса и т.п. Строка `compat` представляет составное *регулярное выражение*, содержащее все три *группы*. Строки `tel`, `addr` и `em` нам понадобятся для размещения в них результатов разбора. Применим вначале к строкам `s4` и `s5` каждый из шаблонов `pat1`, `pat2`, `pat3` в отдельности и выделим соответствующий атрибут из строки. Вот код, выполняющий эти операции:

```
Regex reg1 = new Regex(pat1);
      Match match1= reg1.Match(s4);
      Console.WriteLine("Value =" + match1.Value);
      tel= match1.Groups["tel"].Value;
      Console.WriteLine(tel);
      Regex reg2 = new Regex(pat2);
      Match match2= reg2.Match(s5);
      Console.WriteLine("Value =" + match2.Value);
      addr= match2.Groups["addr"].Value;
      Console.WriteLine(addr);
      Regex reg3 = new Regex(pat3);
      Match match3= reg3.Match(s5);
      Console.WriteLine("Value =" + match3.Value);
      em= match3.Groups["em"].Value;
      Console.WriteLine(em);
```

Все выполняется нужным образом - создаются именованные *группы*, к ним можно получить доступ и извлечь найденный значения атрибутов. А теперь попробуем решить ту же задачу одним махом, используя составной шаблон `compat`:

```
Regex comreg = new Regex(compat);
      Match commatch= comreg.Match(s4);
      tel= commatch.Groups["tel"].Value;
      Console.WriteLine(tel);
      addr= commatch.Groups["addr"].Value;
      Console.WriteLine(addr);
      em= commatch.Groups["em"].Value;
      Console.WriteLine(em);
   }//  TestAttributes
```

И эта задача успешно решается. Взгляните на результаты разбора текста.

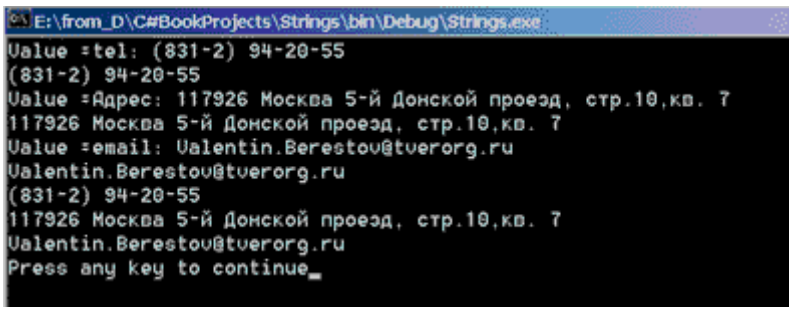

Рис. 15.7. Регулярные выражения. Пример "Атрибуты"

На этом и завершим рассмотрение *регулярных выражений* а также лекции, посвященные работе с текстами в C#.

16. Лекция: Классы

Классы и ООП

Объектно-ориентированное программирование и проектирование построено на классах. Любую программную систему, выстроенную в объектном стиле, можно рассматривать как совокупность классов, возможно, объединенных в проекты, пространства имен, решения, как это делается при программировании в Visual Studio .Net.

Две роли классов

У класса две различные роли: модуля и типа данных. *Класс - это модуль*, архитектурная единица построения программной системы. Модульность построения - основное свойство программных систем. В ООП программная система, строящаяся по модульному принципу, состоит из классов, являющихся основным видом модуля. Модуль может не представлять собой содержательную единицу - его размер и содержание определяется архитектурными соображениями, а не семантическими. Ничто не мешает построить монолитную систему, состоящую из одного модуля - она может решать ту же задачу, что и система, состоящая из многих модулей.

Вторая роль класса не менее важна. *Класс - это тип данных*, задающий реализацию некоторой абстракции данных, характерной для задачи, в интересах которой создается программная система. С этих позиций классы - не просто кирпичики, из которых строится система. Каждый кирпичик теперь имеет важную содержательную начинку. Представьте себе современный дом, построенный из кирпичей, и дом будущего, где каждый кирпич выполняет определенную функцию: один следит за температурой, другой - за составом воздуха в доме. ОО-программная система напоминает дом будущего.

Состав класса, его размер определяется не архитектурными соображениями, а той абстракцией данных, которую должен реализовать класс. Если вы создаете класс `Account`, реализующий такую абстракцию как банковский счет, то в этот класс нельзя добавить *поля из класса* `Car`, задающего автомобиль.

Объектно-ориентированная разработка программной системы основана на стиле, называемом ***проектированием от данных***. Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи. Каждая из таких абстракций реализуется в виде класса, которые и становятся модулями - архитектурными единицами построения нашей системы. В основе класса лежит абстрактный тип данных.

В хорошо спроектированной ОО-системе каждый класс играет обе роли, так что каждый модуль нашей системы имеет вполне определенную смысловую нагрузку. Типичная ошибка - рассматривать класс только как архитектурную единицу, объединяя под обложкой класса разнородные *поля* и функции, после чего становится неясным, какой же тип данных задает этот класс.

Синтаксис класса

Ни одна из предыдущих лекций не обходилась без появления классов и обсуждения многих деталей, связанных с ними. Сейчас попробуем сделать некоторые уточнения, подвести итоги и с новых позиций взглянуть на уже знакомые вещи. Начнем с *синтаксиса описания класса*:

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

Атрибутам будет посвящена отдельная лекция. Возможными модификаторами в объявлении класса могут быть модификаторы `new`, `abstract`, `sealed`, о которых подробно будет говориться при рассмотрении наследования, и четыре *модификатора доступа*, два из которых - `private` и `protected` - могут быть заданы только для вложенных классов. Обычно класс имеет атрибут доступа `public`, являющийся значением по умолчанию. Так что в простых случаях объявление класса выглядит так:

```
public class Rational {тело_класса}
```

В теле класса могут быть объявлены:

- *константы*;
- *поля*;
- *конструкторы* и *деструкторы*;
- *методы*;
- события;
- делегаты;
- классы (структуры, интерфейсы, перечисления).

О событиях и делегатах предстоит подробный разговор в последующих лекциях. Из синтаксиса следует, что классы могут быть вложенными. Такая ситуация - довольно редкая. Ее стоит использовать, когда некоторый класс носит вспомогательный характер, разрабатывается в интересах другого класса, и есть полная уверенность, что внутренний класс никому не понадобится, кроме класса, в который он вложен. Как уже упоминалось, внутренние классы обычно имеют *модификатор доступа*, отличный от `public`. Основу любого класса составляют его *конструкторы*, *поля* и *методы*.

Поля класса

Поля класса синтаксически являются обычными переменными (объектами) языка. Их описание удовлетворяет обычным правилам объявления переменных, о чем подробно говорилось в лекции 5. Содержательно *поля* задают представление той самой абстракции данных, которую реализует класс. *Поля* характеризуют *свойства объектов класса*. Напомню, что, когда создается новый объект класса (в динамической памяти или в стеке), то этот объект представляет собой набор *полей класса*. Два объекта одного класса имеют один и тот же набор *полей*, но разнятся значениями, хранимыми в этих *полях*. Все объекты класса `Person` могут иметь *поле*, характеризующее рост персоны, но один объект может быть высокого роста, другой - низкого, а третий - среднего роста.

Доступ к полям

Каждое *поле* имеет *модификатор доступа*, принимающий одно из четырех значений: `public`, `private`, `protected`, `internal`. Атрибутом доступа по умолчанию является атрибут `private`. Независимо от значения атрибута доступа, все *поля* доступны для всех *методов класса*. Они являются для *методов класса* глобальной информацией, с которой работают все *методы*, извлекая из *полей* нужные им данные и изменяя их значения в ходе работы. Если *поля* доступны только для *методов класса*, то они имеют атрибут доступа `private`, который можно опускать. Такие *поля* считаются закрытыми, но часто желательно, чтобы некоторые из них были доступны в более широком контексте. Если некоторые *поля класса* `A` должны быть доступны для *методов класса* `B`, являющегося потомком класса `A`, то эти *поля* следует снабдить атрибутом `protected`. Такие *поля* называются защищенными. Если некоторые *поля* должны быть доступны для *методов классов* `B1`, `B2` и так далее, дружественных по отношению к классу `A`, то эти *поля* следует снабдить атрибутом `internal`, а все дружественные классы B поместить в один проект (assembly). Такие *поля* называются дружественными. Наконец, если некоторые *поля* должны быть доступны для *методов* любого класса `B`,

которому доступен сам класс A, то эти *поля* следует снабдить атрибутом public. Такие *поля* называются общедоступными или открытыми.

Методы класса

Методы класса синтаксически являются обычными процедурами и функциями языка. Их описание удовлетворяет обычным правилам объявления процедур и функций, о чем подробно говорилось в лекции 9. Содержательно *методы* определяют ту самую абстракцию данных, которую реализует класс. *Методы* содержат описания *операций*, доступных над объектами класса. Два объекта одного класса имеют один и тот же набор *методов*.

Доступ к методам

Каждый *метод* имеет *модификатор доступа*, принимающий одно из четырех значений: public, private, protected, internal. Атрибутом доступа по умолчанию является атрибут private. Независимо от значения атрибута доступа, все *методы* доступны для вызова при выполнении *метода класса*. Если *методы* имеют атрибут доступа private, возможно, опущенный, то тогда они доступны только для вызова и только внутри *методов* самого класса. Такие *методы* считаются *закрытыми*. Понятно, что класс, у которого все *методы закрыты*, абсурден, поскольку никто не смог бы вызвать ни один из его *методов*. Как правило, у класса есть открытые *методы*, задающие интерфейс класса, и *закрытые методы*. Интерфейс - это лицо класса и именно он определяет, чем класс интересен своим клиентам, что он может делать, какие сервисы предоставляет клиентам. *Закрытые методы* составляют важную часть класса, позволяя клиентам не вникать во многие детали реализации. Эти *методы* клиентам класса недоступны, они о них могут ничего не знать, и, самое главное, изменения в *закрытых методах* никак не отражаются на клиентах класса при условии корректной работы открытых *методов*.

Если некоторые *методы класса* A должны быть доступны для вызовов в *методах класса* B, являющегося потомком класса A, то такие *методы* следует снабдить атрибутом protected. Если некоторые *методы* должны быть доступны только для *методов классов* B1, B2 и так далее, дружественных по отношению к классу A, то такие *методы* следует снабдить атрибутом internal, а все дружественные классы B поместить в один проект. Наконец, если некоторые *методы* должны быть доступны для *методов* любого класса B, которому доступен сам класс A, то такие *методы* снабжаются атрибутом public.

Методы-свойства

Методы, называемые **свойствами (Properties)**, представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами. При работе со *свойствами объекта* (*полями*) часто нужно решить, какой *модификатор доступа* использовать, чтобы реализовать нужную стратегию доступа к *полю* класса. Перечислю пять наиболее употребительных стратегий:

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

Открытость свойств (атрибут public) позволяет реализовать только первую стратегию. В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через *методы*. Для эффективности этого процесса и введены специальные *методы-свойства*.

Приведу вначале пример, а потом уточню синтаксис этих *методов*. Рассмотрим класс `Person`, у которого пять *полей*: `fam`, `status`, `salary`, `age`, `health`, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны. Для каждого из этих *полей* может быть разумной своя стратегия доступа. Возраст доступен для чтения и записи, фамилию можно задать только один раз, статус можно только читать, зарплата недоступна для чтения, а здоровье закрыто для доступа и только специальные *методы* класса могут сообщать некоторую информацию о здоровье персоны. Вот как на C# можно обеспечить эти стратегии доступа к закрытым *полям класса*:

```
public class Person
{
   //поля (все закрыты)
   string fam="", status="", health="";
   int age=0, salary=0;
   //методы - свойства
   /// <summary>
   ///стратегия: Read,Write-once (Чтение, запись при
   ///первом обращении)
   /// </summary>
   public string Fam
   {
      set {if (fam == "") fam = value;}
      get {return(fam);}
   }
   /// <summary>
   ///стратегия: Read-only(Только чтение)
   /// </summary>
   public string Status
   {
      get {return(status);}
   }
   /// <summary>
   ///стратегия: Read,Write (Чтение, запись)
   /// </summary>
   public int Age
   {
      set
      {
         age = value;
         if(age < 7)status ="ребенок";
         else if(age <17)status ="школьник";
         else if (age < 22)status = "студент";
         else status = "служащий";
      }
      get {return(age);}
   }
   /// <summary>
   ///стратегия: Write-only (Только запись)
   /// </summary>
   public int Salary
   {
      set {salary = value;}
   }
}
```

Рассмотрим теперь общий *синтаксис методов-свойств*. Пусть `name` - это закрытое свойство. Тогда для него можно определить открытый *метод-свойство* (функцию), возвращающую тот же тип, что и *поле* `name`. Имя *метода* обычно близко к имени *поля* (например, `Name`). Тело свойства содержит два *метода* - `get` и `set`, один из которых может быть опущен. *Метод* `get` возвращает значение закрытого *поля*, *метод* `set` - устанавливает значение, используя передаваемое ему значение в момент вызова,

хранящееся в служебной переменной со стандартным именем `value`. Поскольку `get` и `set` - это обычные процедуры языка, то программно можно реализовать сколь угодно сложные стратегии доступа. В нашем примере фамилия меняется, только если ее значение равно пустой строке и это означает, что фамилия персоны ни разу еще не задавалась. Статус персоны пересчитывается автоматически при всяком изменении возраста, явно изменять его нельзя. Вот пример, показывающий, как некоторый клиент создает и работает с *полями* персоны:

```
public void TestPersonProps()
{
   Person pers1 = new Person();
   pers1.Fam = "Петров";
   pers1.Age = 21;
   pers1.Salary = 1000;
   Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
      pers1.Fam, pers1.Age, pers1.Status);
   pers1.Fam = "Иванов"; pers1.Age += 1;
   Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
      pers1.Fam, pers1.Age, pers1.Status);
}//TestPersonProps
```

Заметьте, клиент работает с *методами-свойствами* так, словно они являются настоящими *полями*, вызывая их как в правой, так и в левой части оператора присваивания. Заметьте также, что с каждым *полем* можно работать только в *полном* соответствии с той стратегией, которую реализует данное свойство. Попытка изменения фамилии не принесет успеха, а изменение возраста приведет и к одновременному изменению статуса. На рис. 16.1 показаны результаты работы этой процедуры.

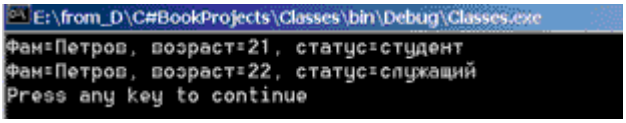

Рис. 16.1. Методы-свойства и стратегии доступа к полям

Индексаторы

Свойства являются частным случаем *метода класса* с особым синтаксисом. Еще одним частным случаем является *индексатор*. Метод-индексатор является обобщением *метода-свойства*. Он обеспечивает доступ к закрытому *полю*, представляющему массив. Объекты класса индексируются по этому *полю*.

Синтаксически объявление *индексатора* - такое же, как и в случае свойств, но *методы* `get` и `set` приобретают аргументы по числу размерности массива, задающего индексы элемента, значение которого читается или обновляется. Важным ограничением является то, что у класса может быть только один *индексатор* и у этого *индексатора* стандартное имя `this`. Так что если среди *полей* класса есть несколько массивов, то индексация объектов может быть выполнена только по одному из них.

Добавим в класс `Person` свойство `children`, задающее детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит *индексатор*:

```
const int Child_Max = 20; //максимальное число детей
Person[] children = new Person[Child_Max];
int count_children=0; //число детей
public Person this[int i] //индексатор
{
   get {if (i>=0 && i< count_children)return(children[i]);
         else return(children[0]);}
   set
   {
```

```
      if (i==count_children && i< Child_Max)
       {children[i] = value; count_children++;}
   }
}
```

Имя у *индексатора* - `this`, в квадратных скобках в заголовке перечисляются индексы. В *методах* `get` и `set`, обеспечивающих доступ к массиву `children`, по которому ведется индексирование, анализируется корректность задания индекса. Закрытое *поле* `count_children`, хранящее текущее число детей, доступно только для чтения благодаря добавлению соответствующего *метода-свойства*. Запись в это *поле* происходит в *методе* `set` *индексатора*, когда к массиву `children` добавляется новый элемент.

Протестируем процесс добавления детей персоны и работу *индексатора*:

```
public void TestPersonChildren()
{
   Person pers1 = new Person(), pers2 = new Person();
   pers1.Fam = "Петров"; pers1.Age = 42;
      pers1.Salary = 10000;
   pers1[pers1.Count_children] = pers2;
   pers2.Fam ="Петров"; pers2.Age = 21; pers2.Salary = 1000;
   Person pers3= new Person("Петрова");
   pers1[pers1.Count_children] = pers3;
   pers3.Fam ="Петрова"; pers3.Age = 5;
   Console.WriteLine ("Фам={0}, возраст={1}, статус={2}",
      pers1.Fam, pers1.Age, pers1.Status);
   Console.WriteLine ("Сын={0}, возраст={1}, статус={2}",
      pers1[0].Fam, pers1[0].Age, pers1[0].Status);
   Console.WriteLine ("Дочь={0}, возраст={1}, статус={2}",
      pers1[1].Fam, pers1[1].Age, pers1[1].Status);
}
```

Заметьте, *индексатор* создает из объекта как бы массив объектов, индексированный по соответствующему *полю*, в данном случае по *полю* `children`. На рис. 16.2 показаны результаты вывода.

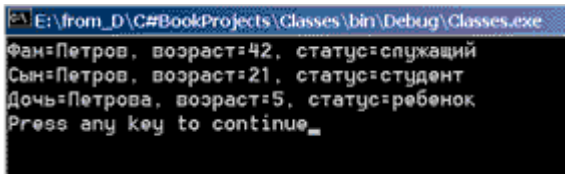

Рис. 16.2. Работа с индексатором класса

Операции

Еще одним частным случаем являются *методы*, задающие над объектами-классами *бинарную или унарную операцию*. Введение в класс таких *методов* позволяет строить выражения, аналогичные арифметическим и булевым выражениям с обычно применяемыми знаками *операций* и сохранением приоритетов *операций*. Синтаксис задания таких *методов* и детали применения опишу чуть позже при проектировании класса рациональных чисел `Rational`, где введение *операций* вполне оправдано.

Статические поля и методы класса

Ранее говорилось, что когда *конструктор класса* создает новый объект, то в памяти создается структура данных с *полями*, определяемыми классом. Уточним теперь это описание. Не все *поля* отражаются в структуре объекта. У класса могут быть *поля*, связанные не с объектами, а с самим классом. Эти *поля* объявляются как *статические* с модификатором `static`. ***Статические поля*** доступны всем *методам класса*. Независимо

от того, какой объект вызвал *метод*, используются одни и те же *статические поля*, позволяя *методу* использовать информацию, созданную другими объектами класса. *Статические поля* представляют общий информационный пул для всех объектов классов, позволяя извлекать и создавать общую информацию. Например, у класса `Person` может быть *статическое поле* `message`, в котором каждый объект может оставить сообщение для других объектов класса.

Аналогично *полям*, у класса могут быть и *статические методы*, объявленные с модификатором `static`. Такие *методы* не используют информацию о ***свойствах конкретных объектов класса*** - они обрабатывают общую для класса информацию, хранящуюся в его *статических полях*. Например, в классе `Person` может быть *статический метод*, обрабатывающий данные из *статического поля* `message`. Другим частым случаем применения *статических методов* является ситуация, когда класс предоставляет свои сервисы объектам других классов. Таковым является класс `Math` из библиотеки FCL, который не имеет собственных *полей* - все его *статические методы* работают с объектами арифметических классов.

Как вызываются *статические поля* и *методы*? Напомню, что всякий вызов *метода* в объектных вычислениях имеет вид `x.F(...);` где `x` - это цель вызова. Обычно целью вызова является объект, который вызывает *методы класса*, не являющиеся *статическими* (динамическими или экземплярными). В этом случае *поля* целевого объекта доступны *методу* и служат глобальным источником информации. Если же необходимо вызвать *статический метод* (поле), то целью должен быть сам класс. Можно полагать, что для каждого класса автоматически создается статический объект с именем класса, содержащий *статические поля* и обладающий *статическими методами*. Этот объект и его *методы* доступны и тогда, когда ни один другой динамический объект класса еще не создан. (Полагаю, что разумно обратиться к лекции 2, а именно, к разделу, описывающему точку большого взрыва, процесс вычислений в ОО-системах, вызовы *статических методов*.)

Константы

В классе могут быть объявлены *константы*. Синтаксис объявления приводился в лекции 5. *Константы* фактически являются *статическими полями*, доступными только для чтения, значения которых задаются при инициализации. Однако задавать модификатор `static` для *констант* не только не нужно, но и запрещено. В нашем классе `Person` была задана *константа* `Child_Max`, характеризующая максимальное число детей у персоны.

Никаких проблем не возникает, когда речь идет о *константах* встроенных типов, таких, как `Child_Max`. Однако совсем не просто определить в языке C# *константы* собственного класса. Этот вопрос будет подробно рассматриваться чуть позже, когда речь пойдет о проектировании класса `Rational`.

Конструкторы класса

Конструктор - неотъемлемый компонент класса. Нет классов без *конструкторов*. ***Конструктор*** представляет собой специальный *метод класса*, позволяющий создавать объекты класса. Одна из синтаксических особенностей этого *метода* в том, что его имя должно совпадать с именем класса. Если программист не определяет *конструктор класса*, то к классу автоматически добавляется *конструктор* по умолчанию - *конструктор* без аргументов. Заметьте, что если программист сам создает один или несколько *конструкторов*, то автоматического добавления *конструктора* без аргументов не происходит.

Как и когда происходит создание объектов? Чаще всего, при объявлении сущности в момент ее инициализации. Давайте обратимся к нашему последнему примеру и рассмотрим создание трех объектов класса Person:

```
Person pers1 = new Person(), pers2 = new Person();
Person pers3= new Person("Петрова");
```

Сущности `pers1`, `pers2` и `pers3` класса `Person` объявляются с инициализацией, задаваемой унарной *операцией* `new`, которой в качестве аргумента передается *конструктор класса* `Person`. У класса может быть несколько *конструкторов* - это типичная практика, - отличающихся сигнатурой. В данном примере в первой строке вызывается *конструктор* без аргументов, во второй строке для сущности `pers3` вызывается *конструктор* с одним аргументом типа `string`. Разберем в деталях процесс создания:

- первым делом для сущности pers создается ссылка, пока висячая, со значением `null`;
- затем в динамической памяти создается объект - структура данных с *полями*, определяемыми классом `Person`. *Поля* объекта инициализируются значениями по умолчанию: ссылочные *поля* - значением `null`, арифметические - нулями, строковые - пустой строкой. Эту работу выполняет *конструктор* по умолчанию, который, можно считать, всегда вызывается в начале процесса создания. Заметьте, если инициализируется переменная значимого типа, то все происходит аналогичным образом, за исключением того, что объект создается в стеке;
- если *поля класса* проинициализированы, как в нашем примере, то выполняется инициализация *полей* заданными значениями;
- если вызван *конструктор* с аргументами, то начинает выполняться тело этого *конструктора*. Как правило, при этом происходит инициализация отдельных *полей* класса значениями, переданными *конструктору*. Так, *поле* `fam` объекта `pers3` получает значение "Петрова";
- На заключительном этапе ссылка связывается с созданным объектом.

Процесс создания объектов становится сложнее, когда речь идет об объектах, являющихся потомками некоторого класса. В этом случае, прежде чем создать сам объект, нужно вызвать *конструктор*, создающий родительский объект. Но об этом мы еще поговорим при изучении наследования. (Ключевое слово `new` используется в языке для двух разных целей. Во-первых, это имя *операции*, запускающей только что описанный процесс создания объекта. Во-вторых, это модификатор класса или *метода*. Роль `new` как модификатора будет выяснена при рассмотрении наследования.)

Зачем классу нужно несколько *конструкторов*? Дело в том, что, в зависимости от контекста и создаваемого объекта, может требоваться различная инициализация его *полей*. Перегрузка *конструкторов* и обеспечивает решение этой задачи.

Немного экзотики, связанной с *конструкторами*. *Конструктор* может быть объявлен с атрибутом `private`. Понятно, что в этом случае внешний пользователь не может воспользоваться им для создания объектов. Но это могут делать *методы класса*, создавая объекты для собственных нужд со специальной инициализацией. Пример такого *конструктора* будет дан позже.

В классе можно объявить статический *конструктор* с атрибутом `static`. Он вызывается автоматически - его не нужно вызывать стандартным образом. Точный момент вызова не определен, но гарантируется, что вызов произойдет до создания первого объекта класса. Такой *конструктор* может выполнять некоторую предварительную работу, которую нужно выполнить один раз, например, связаться с базой данных, заполнить значения *статических полей* класса, создать *константы* класса, выполнить другие подобные действия. Статический *конструктор*, вызываемый автоматически, не должен иметь *модификаторов доступа*. Вот пример объявления такого *конструктора* в классе `Person`:

```
static Person()
{
```

```
   Console.WriteLine("Выполняется статический конструктор!");
}
```

В нашей тестирующей процедуре, работающей с объектами класса `Person`, этот *конструктор* вызывается первым, и первым появляется сообщение этого *конструктора*.

Подводя итоги, можно отметить, что объекты создаются динамически в процессе выполнения программы - для создания объекта всегда вызывается тот или иной *конструктор класса*.

Деструкторы класса

Если задача создания объектов полностью возлагается на программиста, то задача удаления объектов, после того, как они стали не нужными, в Visual Studio .Net снята с программиста и возложена на соответствующий инструментарий - сборщик мусора. В классическом варианте языка C++ *деструктор* так же необходим классу, как и *конструктор*. В языке C# у класса может быть *деструктор*, но он не занимается удалением объектов и не вызывается нормальным образом в ходе выполнения программы. Так же, как и статический *конструктор*, *деструктор класса*, если он есть, вызывается автоматически в процессе сборки мусора. Его роль - в освобождении ресурсов, например, файлов, открытых объектом. *Деструктор* C# фактически является финализатором (finalizer), с которыми мы еще встретимся при обсуждении исключительных ситуаций. Приведу формальное описание *деструктора класса* `Person`:

```
~Person()
{
   //Код деструктора
}
```

Имя *деструктора* строится из имени класса с предшествующим ему символом `~` (тильда). Как и у статического *конструктора*, у *деструктора* не указывается *модификатор доступа*.

Проектирование класса Rational

В заключение этой лекции займемся проектированием класса `Rational`, описывающего известный в математике тип данных - рациональные числа. По ходу проектирования будут вводиться новые детали, связанные с описанием класса. Начнем проектирование, как обычно, с задания тега `<summary>`, описывающего назначение класса, его свойства и поведение. Вот этот текст:

```
/// <summary>
/// Класс Rational
/// определяет новый тип данных - рациональные числа и
/// основные операции над ними - сложение, умножение,
/// вычитание и деление. Рациональное число задается парой
/// целых чисел (m,n) и изображается обычно в виде дроби m/n.
/// Число m называется числителем,n - знаменателем. Для
/// каждого рационального числа существует множество его
/// представлений, например, 1/2, 2/4, 3/6, 6/12 - задают
/// одно и тоже рациональное число. Среди всех представлений
/// можно выделить то, в котором числитель и знаменатель
/// взаимно несократимы. Такой представитель будет храниться
/// в полях класса. Операции над рациональными числами
/// определяются естественным для математики образом
/// </summary>
public class Rational
{
   // Описание тела класса Rational
```

```
}//Rational
```

Свойства класса Rational

Два целых числа - `m` и `n` представляют рациональное число. Они и становятся *полями* класса. Совершенно естественно сделать эти *поля* закрытыми. Разумная стратегия доступа к ним - "ни чтения, ни записи", поскольку пользователь не должен знать, как представлено рациональное число в классе, и не должен иметь доступа к составляющим рационального числа. Поэтому для таких закрытых *полей* не будут определяться *методы-свойства*. Вот объявление *полей* класса:

```
//Поля класса. Числитель и знаменатель рационального числа.
      int m,n;
```

Конструкторы класса Rational

Инициализация *полей конструктором* по умолчанию никак не может нас устраивать, поскольку нулевой знаменатель - это нонсенс. Поэтому определим *конструктор* с аргументами, которому будут передаваться два целых: числитель и знаменатель создаваемого числа. Кажется, что это единственный разумный *конструктор*, который может понадобиться нашему классу. Однако чуть позже мы добавим в класс закрытый *конструктор* и статический *конструктор*, позволяющий создать *константы* нашего класса. Вот определение *конструктора*:

```
/// <summary>
/// Конструктор класса. Создает рациональное число
/// m/n, эквивалентное a/b, но со взаимно несократимыми
/// числителем и знаменателем. Если b=0, то результатом
/// является рациональное число 0 -пара (0,1).
/// </summary>
/// <param name="a">числитель</param>
/// <param name="b">знаменатель</param>
public Rational(int a, int b)
{
   if(b==0) {m=0; n=1;}
   else
   {
      //приведение знака
      if( b<0) {b=-b; a=-a;}
      //приведение к несократимой дроби
      int d = nod(a,b);
      m=a/d; n=b/d;
   }
}
```

Как видите, *конструктор* класса может быть довольно сложным.

В нем, как в нашем случае, может проверяться корректность задаваемых аргументов. Для рациональных чисел мы полагаем, что задание нулевого знаменателя означает задание рационального числа `0`, и этоэквивалентно заданию пары `(0, 1)`. В остальных случаях выполняется приведение заданной пары чисел к эквивалентному рациональному числу с несократимыми числителем и знаменателем. По ходу дела вызывается *закрытый метод* класса, вычисляющий значение `НОД(a,b)` - наибольшего общего делителя чисел `a` и `b`.

Методы класса Rational

Если *поля класса* почти всегда закрываются, чтобы скрыть от пользователя представление данных класса, то *методы класса* всегда имеют открытую часть - те

сервисы (службы), которые класс предоставляет своим клиентам и наследникам. Но не все *методы* открываются. Большая часть *методов* класса может быть закрытой, скрывая от клиентов детали реализации, необходимые для внутреннего использования. Заметьте, сокрытие представления и реализации делается не по соображениям утаивания того, как реализована система. Чаще всего, ничто не мешает клиентам ознакомиться с полным текстом класса. Сокрытие делается в интересах самих клиентов. При сопровождении программной системы изменения в ней неизбежны. Клиенты не почувствуют на себе негативные последствия изменений, если они делаются в закрытой части класса. Чем больше закрытая часть класса, тем меньше влияние изменений на клиентов класса.

Закрытый метод НОД

Метод, вычисляющий наибольший общий делитель пары чисел, понадобится не только *конструктору класса*, но и всем *операциям* над рациональными числами. Алгоритм нахождения общего делителя хорошо известен со времен Эвклида. Я приведу программный код *метода* без особых пояснений:

```
/// <summary>
/// Закрытый метод класса.
/// Возвращает наибольший общий делитель чисел a,b
/// </summary>
/// <param name="a">первое число</param>
/// <param name="b">второе число, положительное</param>
/// <returns>НОД(a,b)</returns>
int nod(int m, int n)
{
   int p=0;
   m=Math.Abs(m); n =Math.Abs(n);
   if(n>m){p=m; m=n; n=p;}
   do
   {
   p = m%n; m=n; n=p;
   }while (n!=0);
   return(m);
}//nod
```

Печать рациональных чисел

Почти любой класс содержит один или несколько *методов*, позволяющих выводить на печать данные о классе. Такой *метод* имеется и в классе `Rational`. Вот его текст:

```
public void PrintRational(string name)
{
   Console.WriteLine(" {0} = {1}/{2}",name,m,n);
}
```

Метод печатает имя и значение рационального числа в форме `m/n`.

Тестирование создания рациональных чисел

В классе `Testing`, предназначенном для тестирования нашей работы и являющегося клиентом класса `Rational`, создадим процедуру, позволяющую проверить корректность создания рациональных чисел. Вот эта процедура:

```
public void TestCreateRational()
{
   Rational r1=new Rational(0,0), r2 = new Rational(1,1);
   Rational r3=new Rational(10,8), r4 = new Rational(2,6);
```

```
   Rational r5=new Rational(4,-12), r6 = new Rational (-12,-14);
   r1.PrintRational("r1:(0,0)");
   r2.PrintRational("r2:(1,1)");
   r3.PrintRational("r3:(10,8)");
   r4.PrintRational("r4:(2,6)");
   r5.PrintRational("r5: (4,-12)");
   r6.PrintRational("r6: (-12,-14)");
}
```

Она создает и печатает шесть рациональных чисел. Вот как выглядят результаты ее работы.

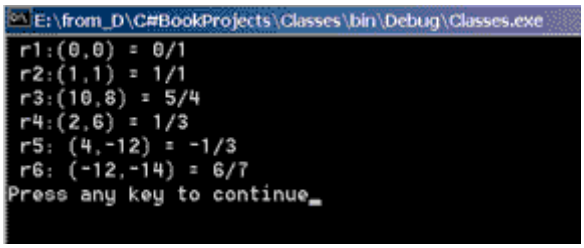

Рис. 16.3. Создание и печать рациональных чисел

Операции над рациональными числами

Определим над рациональными числами стандартный набор *операций* - сложение и вычитание, умножение и деление. Реализуем эти *операции методами* с именами `Plus`, `Minus`, `Mult`, `Divide` соответственно. Поскольку рациональные числа - это прежде всего именно числа, то для выполнения *операций* над ними часто удобнее пользоваться привычными знаками *операций* (`+`, `-`, `*`, `/`). Язык C# допускает определение *операций*, заданных указанными символами. Этот процесс называется перегрузкой *операций*, и мы рассмотрим сейчас, как это делается. Конечно, можно было бы обойтись только перегруженными *операциями*, но мы приведем оба способа. Пользователь сам будет решать, какой из способов применять в конкретной ситуации - вызывать *метод* или *операцию*.

Покажем вначале реализацию *метода* `Plus` и *операции* `+`:

```
public Rational Plus(Rational a)
{
   int u,v;
   u = m*a.n +n*a.m; v= n*a.n;
   return( new Rational(u, v));
}//Plus
public static Rational operator +(Rational r1, Rational r2)
{
   return (r1.Plus(r2));
}
```

Метод `Plus` реализуется просто. По правилам сложения дробей вычисляется числитель и знаменатель результата, и эти данные становятся аргументами *конструктора*, создающего требуемое рациональное число, которое удовлетворяет правилам класса.

Обратите внимание на то, как определяется *операция класса*. Именем соответствующего *метода* является сам знак *операции*, которому предшествует ключевое слово `operator`. Важно также помнить, что *операция* является *статическим методом* класса с атрибутом `static`.

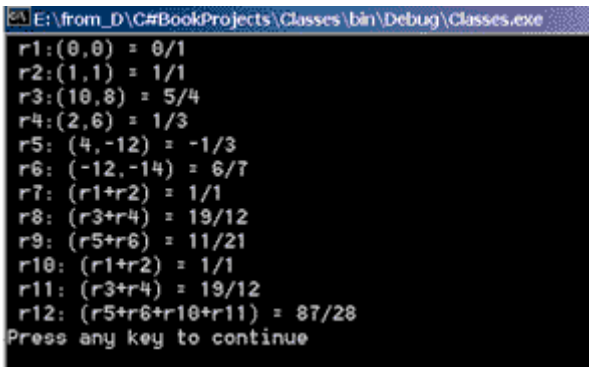

Рис. 16.4. Сложение рациональных чисел

В данном конкретном случае *операция* реализуется вызовом *метода* `Plus`. Как теперь все это работает? Вот пример:

```
public void TestPlusRational()
{
   Rational r1=new Rational(0,0), r2 = new Rational(1,1);
   Rational r3=new Rational(10,8), r4 = new Rational(2,6);
   Rational r5=new Rational(4,-12), r6 = new Rational
      (-12,-14);
   Rational r7,r8, r9,r10, r11, r12;
   r7 = r1.Plus(r2); r8 = r3.Plus(r4); r9 = r5.Plus(r6);
   r10 = r1+r2; r11 = r3+r4; r12 = r5+r6+r10+r11;
   r1.PrintRational("r1:(0,0)"); r2.PrintRational("r2:(1,1)");
   r3.PrintRational("r3:(10,8)"); r4.PrintRational("r4:(2,6)");
   r5.PrintRational("r5: (4,-12)"); r6.PrintRational
      ("r6: (-12,-14)");
   r7.PrintRational("r7: (r1+r2)"); r8.PrintRational
      ("r8: (r3+r4)");
   r9.PrintRational("r9: (r5+r6)"); r10.PrintRational
      ("r10: (r1+r2)");
   r11.PrintRational("r11: (r3+r4)");
   r12.PrintRational("r12: (r5+r6+r10+r11)");
}
```

Обратите внимание на вычисление `r12`: здесь ощутимо видно преимущество *операций*, позволяющих записывать сложные выражения в простой форме. Результаты вычислений показаны на рис. 16.4.

Аналогичным образом определим остальные *операции* над рациональными числами:

```
public Rational Minus(Rational a)
{
   int u,v;
   u = m*a.n - n*a.m; v= n*a.n;
   return( new Rational(u, v));
}//Minus
public static Rational operator -(Rational r1, Rational r2)
{
   return (r1.Minus(r2));
}
public Rational Mult(Rational a)
{
   int u,v;
   u = m*a.m; v= n*a.n;
   return( new Rational(u, v));
}//Mult
public static Rational operator *(Rational r1, Rational r2)
{
```

```
   return (r1.Mult(r2));
}
public Rational Divide(Rational a)
{
   int u,v;
   u = m*a.n; v= n*a.m;
   return( new Rational(u, v));
}//Divide
public static Rational operator /(Rational r1, Rational r2)
{
   return (r1.Divide(r2));
}
```

Вот тест, проверяющий работу этих *операций*:

```
public void TestOperRational()
{
   Rational r1=new Rational(1,2), r2 = new Rational(1,3);
   Rational r3, r4, r5, r6 ;
   r3 = r1- r2; r4 = r1*r2; r5 = r1/r2; r6 = r3+r4*r5;
   r1.PrintRational("r1: (1,2)"); r2.PrintRational("r2: (1,3)");
   r3.PrintRational("r3: (r1-r2)"); r4.PrintRational("r4: (r1*r2)");
   r5.PrintRational("r5: (r1/r2)");
   r6.PrintRational("r6: (r3+r4*r5)");
}
```

Результаты работы этого теста показаны на рис. 16.5. Обратите внимание: при перегрузке *операций* сохраняется общепринятый приоритет *операций*. Поэтому при вычислении выражения `r3+r4*r5` вначале будет выполняться умножение рациональных чисел, а потом уже сложение.

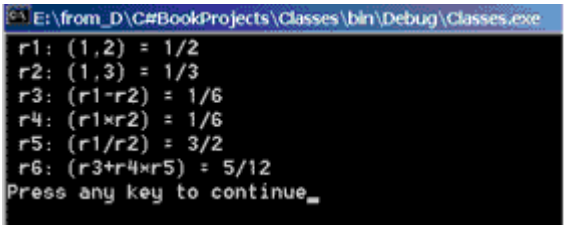

Рис. 16.5. Операции и выражения над рациональными числами

Константы класса Rational

Рассмотрим важную проблему определения *констант* в собственном классе. Определим две *константы* `0` и `1` класса `Rational`. Кажется, что сделать это невозможно из-за ограничений, накладываемых на объявление *констант*. Напомню, *константы* должны быть инициализированы в момент объявления, и их значения должны быть заданы *константными* выражениями, известными в момент компиляции. Но в момент компиляции у класса `Rational` нет никаких известных *константных* выражений. Как же быть? Справиться с проблемой поможет статический *конструктор*, созданный для решения подобных задач. Роль *констант* класса будут играть *статические поля*, объявленные с *атрибутом readonly*, то есть доступные только для чтения. Нам также будет полезен закрытый *конструктор класса*. Еще укажем, что введение *констант* класса требует использования экзотических средств языка C#. Вначале определим закрытый *конструктор*:

```
private Rational(int a, int b, string t)
{
   m = a; n = b;
}
```

Не забудем, что при перегрузке *методов* (в данном случае *конструкторов*) сигнатуры должны различаться, и поэтому пришлось ввести дополнительный аргумент `t` для избежания конфликтов. Поскольку *конструктор* закрытый, то гарантируется корректное задание аргументов при его вызове. Определим теперь *константы* класса, которые, как я уже говорил, задаются *статическими полями* с *атрибутом readonly*:

```
//Константы класса 0 и 1 - Zero и One
      public static readonly Rational Zero, One;
```

А теперь зададим статический *конструктор*, в котором определяются значения *констант*:

```
static Rational()
{
   Console.WriteLine("static constructor Rational");
   Zero = new Rational(0, 1, "private");
   One = new Rational (1, 1, "private");
}//Статический конструктор
```

Как это все работает? Статический *конструктор* вызывается автоматически один раз до начала работы с объектами класса. Он и задаст значения *статических полей* `Zero`, `One`, представляющих рациональные числа с заданным значением. Поскольку эти *поля* имеют атрибут `static` и *readonly*, то они доступны для всех объектов класса и не изменяются в ходе вычислений, являясь настоящими *константами* класса. Прежде чем привести пример работы с *константами*, давайте добавим в наш класс важные булевы *операции* над рациональными числами - равенство и неравенство, больше и меньше. При этом две последние *операции* сделаем перегруженными, позволяя сравнивать рациональные числа с числами типа `double`:

```
public static bool operator ==(Rational r1, Rational r2)
{
   return((r1.m ==r2.m)&& (r1.n ==r2.n));
}
   public static bool operator !=(Rational r1, Rational r2)
{
   return((r1.m !=r2.m)|| (r1.n !=r2.n));
}
   public static bool operator <(Rational r1, Rational r2)
{
   return(r1.m * r2.n < r2.m* r1.n);
}
   public static bool operator >(Rational r1, Rational r2)
{
   return(r1.m * r2.n > r2.m* r1.n);
}
public static bool operator <(Rational r1, double r2)
{
   return((double)r1.m / (double)r1.n < r2);
}
public static bool operator >(Rational r1, double r2)
{
   return((double)r1.m / (double)r1.n > r2);
}
```

Наш последний пример демонстрирует работу с *константами*, булевыми и арифметическими выражениями над рациональными числами:

```
public void TestRationalConst()
{
   Rational r1 = new Rational(2,8), r2 =new Rational(2,5);
   Rational r3 = new Rational(4, 10), r4 = new Rational(3,7);
```

```
   Rational r5 = Rational.Zero, r6 = Rational.Zero;
   if ((r1 != Rational.Zero) && (r2 == r3))r5 =
      (r3+Rational.One)*r4;
   r6 = Rational.One + Rational.One;
   r1.PrintRational("r1: (2,8)");
   r2.PrintRational ("r2: (2,5)");
   r3.PrintRational("r3: (4,10)");
   r4.PrintRational("r4: (3,7)");
   r5.PrintRational("r5: ((r3 +1)*r4)");
   r6.PrintRational("r6: (1+1)");
}
```

Результаты работы этого примера показаны на рис. 16.6.

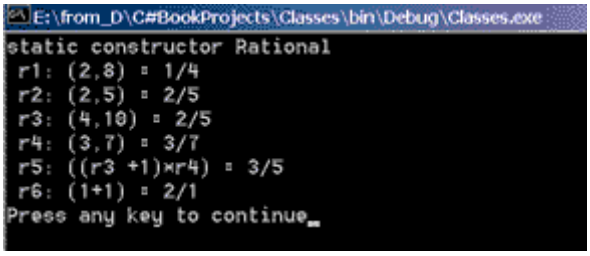

Рис. 16.6. Константы и выражения типа Rational

17. Лекция: Структуры и перечисления

Развернутые и ссылочные типы

Рассмотрим объявление объекта класса `T` с инициализацией:

```
T x = new T();
```

Напомню, как выполняется этот оператор. В памяти создается объект типа `T`, основанного на классе `T`, и сущность `x` связывается с этим объектом. Сущность, не прошедшая инициализацию (явную или неявную), не связана ни с одним объектом, а потому не может использоваться в вычислениях - у нее нет полей, хранящих значения, она не может вызывать методы класса. Объектам нужна память, чтобы с ними можно было работать. Есть две классические стратегии выделения памяти и связывания объекта, создаваемого в памяти, и сущности, объявленной в тексте.

Определение 1. Класс `T` относится к ***развернутому типу***, если память отводится сущности `x`; объект разворачивается на памяти, жестко связанной с сущностью.

Определение 2. Класс `T` относится к ***ссылочному типу***, если память отводится объекту; сущность `x` является ссылкой на объект.

Для *развернутого типа* характерно то, что каждая сущность ни с кем не разделяет свою память; сущность жестко связывается со своим объектом. В этом случае сущность и объект можно и не различать, они становятся неделимым понятием. Для *ссылочных типов* ситуация иная - несколько сущностей могут ссылаться на один и тот же объект. Такие сущности разделяют память и являются разными именами одного объекта. Полезно понимать разницу между сущностью, заданной ссылкой, и объектом, на который в текущий момент указывает ссылка.

Развернутые и *ссылочные типы* порождают две различные семантики присваивания - развернутое присваивание и ссылочное присваивание. Рассмотрим присваивание:

```
y = x;
```

Когда сущность `y` и выражение `x` принадлежат *развернутому типу*, то при присваивании изменяется объект. Значения полей объекта, связанного с сущностью `y`, изменяются, получая значения полей объекта, связанного с `x`. Когда сущность `y` и выражение `x` принадлежат *ссылочному типу*, то изменяется ссылка, но не объект. Ссылка `y` получает значение ссылки `x`, и обе они после присваивания указывают на один и тот же объект.

Язык программирования должен позволять программисту в момент определения класса указать, к *развернутому* или *ссылочному типу* относится класс. К сожалению, язык C# не позволяет этого сделать напрямую - в нем у класса нет модификатора, позволяющего задать *развернутый* или *ссылочный тип*. Какие же средства языка позволяют частично решить эту важную задачу? В лекции 3, где рассматривалась система типов языка C#, отмечалось, что все типы языка делятся на ссылочные и значимые. Термин "значимый" является синонимом термина "развернутый". Беда только в том, что деление на значимые и *ссылочные типы* предопределено языком и не управляется программистом. Напомню, к значимым типам относятся все встроенные арифметические типы, булев тип, *структуры*; к *ссылочным типам* - массивы, строки, классы. Так можно ли в C# спроектировать свой собственный класс так, чтобы он относился к значимым типам? Ответ на это вопрос положительный, хотя и с рядом оговорок. Для того чтобы класс отнести к значимым типам, его нужно реализовать как структуру.

Классы и структуры

Структура - это частный случай класса. Исторически *структуры* используются в языках программирования раньше классов. В языках PL/1, C и Pascal они представляли собой только совокупность данных (полей класса), но не включали ни методов, ни событий. В языке C++ возможности *структур* были существенно расширены и они стали настоящими классами, хотя и с некоторыми *ограничениями*. В языке C# - наследнике C++ - сохранен именно такой подход к *структурам*.

Чем следует руководствоваться, делая выбор между *структурой* и классом? Полагаю, можно пользоваться следующими правилами:

- если необходимо отнести класс к *развернутому типу*, делайте его *структурой*;
- если у класса число полей относительно невелико, а число возможных объектов относительно велико, делайте его *структурой*. В этом случае память объектам будет отводиться в стеке, не будут создаваться лишние ссылки, что позволит повысить эффективность работы;
- в остальных случаях проектируйте настоящие классы.

Поскольку на *структуры* накладываются дополнительные *ограничения*, то может возникнуть необходимость в компромиссе - согласиться с *ограничениями* и использовать *структуру* либо пожертвовать развернутостью и эффективностью и работать с настоящим классом. Стоит отметить: когда говорится, что все встроенные типы - `int` и другие - представляют собой классы, то, на самом деле, речь идет о классах, реализованных в виде *структур*.

Структуры

Рассмотрим теперь более подробно вопросы описания *структур*, их *синтаксиса*, семантики и тех особенностей, что отличают их от классов.

Синтаксис структур

Синтаксис объявления структуры аналогичен синтаксису объявления класса:

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

Какие изменения произошли в *синтаксисе* в сравнении с синтаксисом класса, описанным в лекции 16? Их немного. Перечислим их:

- ключевое слово `class` изменено на слово `struct`;
- список родителей, который для классов, наряду с именами интерфейсов, мог включать имя родительского класса, заменен списком интерфейсов. Для *структур* не может быть задан родитель (класс или *структура*). Заметьте, *структура* может наследовать интерфейсы;
- для *структур* неприменимы модификаторы `abstract` и `sealed`. Причиной является отсутствие механизма наследования.

Все, что может быть вложено в тело класса, может быть вложено и в тело *структуры*: поля, методы, конструкторы и прочее, включая классы и интерфейсы.

Аналогично классу, *структура* может иметь статические и не статические поля и методы, может иметь несколько конструкторов, в том числе статические и закрытые конструкторы. Для *структур* можно создавать собственные константы, используя поля с атрибутом `readonly` и статический конструктор. *Структуры* похожи на классы по своему

описанию и ведут себя сходным образом, хотя и имеют существенные различия в семантике присваивания.

Перечислим *ограничения, накладываемые на структуры*.

- Самое серьезное ограничение связано с ограничением наследования. У *структуры* не может быть наследников. У *структуры* не может быть задан родительский класс или родительская *структура*. Конечно, всякая *структура*, как и любой класс в C#, является наследником класса `Object`, наследуя все свойства и методы этого класса. *Структура* может быть наследником одного или нескольких интерфейсов, реализуя методы этих интерфейсов.
- Второе серьезное ограничение связано с процессом создания объектов. Пусть `T` - *структура*, и дано объявление без инициализации - `T x`. Это объявление корректно, в результате будет создан объект без явного вызова операции `new`. Сущности `x` будет отведена память, и на этой памяти будет развернут объект. Но поля объекта не будут инициализированы и, следовательно, не будут доступны для использования в вычислениях. Об этих особенностях подробно говорилось при рассмотрении значимых типов. В этом отношении все, что верно для типа `int`, верно и для всех *структур*.
- Если при объявлении класса его поля можно инициализировать, что найдет отражение при работе конструктора класса, то поля *структуры* не могут быть инициализированы.
- Конструктор по умолчанию у *структур* имеется, при его вызове поля инициализируются значениями по умолчанию. Этот конструктор нельзя заменить, создав собственный конструктор без аргументов.
- В конструкторе нельзя вызывать методы класса. Поля *структуры* должны быть проинициализированы до вызова методов.

Класс Rational или структура Rational

Вернемся к классу `Rational`, спроектированному в предыдущей лекции. Очевидно, что его вполне разумно представить в виде *структуры*. Наследование ему не нужно. Семантика присваивания *развернутого типа* больше подходит для рациональных чисел, чем ссылочная семантика, ведь рациональные числа - это еще один подкласс арифметического класса. В общем, класс `Rational` - прямой кандидат в *структуры*. Зададимся вопросом, насколько просто объявление класса превратить в объявление *структуры*? Достаточно ли заменить слово `class` словом `struct`? В данном случае одним словом не обойтись. Есть одно мешающее *ограничение на структуры*. В конструкторе класса `Rational` вызывается метод `nod`, а вызов методов в конструкторе запрещен. Нетрудно обойти это ограничение, изменив конструктор, то есть явно задав вычисление общего делителя в его теле. Приведу текст этого конструктора:

```
public struct Rational
{
   public Rational(int a, int b)
   {
      if(b==0) {m=0; n=1;}
      else
      {
         //приведение знака
         if( b<0) {b=-b; a=-a;}
         //приведение к несократимой дроби
         int p = 1, m1=a, n1 =b;
         m1=Math.Abs(m1); n1 =Math.Abs(n1);
         if(n1>m1){p=m1; m1=n1; n1=p;}
         do
         {
            p = m1%n1; m1=n1; n1=p;
         }while (n1!=0);
```

```
            p=m1;
            m=a/p; n=b/p;
        }
    }//Конструктор
//поля и методы класса
}
```

Все остальное остается без изменения. Приведу пример работы с рациональными числами, представленными *структурой*:

```
public void TwoSemantics()
{
   Rational r1 = new Rational(1,3), r2 = new Rational(3,5);
   Rational r3, r4;
   r3 = r1+r2; r4 = r3;
   if(r3 >1) r3 = r1+r3 + Rational.One;
   else r3 = r2+r3 - Rational.One;
   r3.PrintRational("r3"); r4.PrintRational("r4");
}
```

В этом примере используются константы, работает статический конструктор, закрытый конструктор, перегруженные операции сравнения, арифметические выражения над рациональными числами. В результате вычислений `r3` получит значение `8/15`, `r4-14/15`. Заметьте, аналогичный пример для класса `Rational` даст те же результаты. Для класса `Rational` и *структуры* `Rational` нельзя обнаружить разницу между ссылочным и развернутым присваиванием. Это связано с особенностью класса `Rational` - он по построению относится к неизменяемым (**immutable**) классам, аналогично классу `String`. Операции этого класса не изменяют поля объекта, а каждый раз создают новый объект. В этом случае можно считать, что объекты класса обладают присваиванием *развернутого типа*.

Встроенные структуры

Как уже говорилось, все значимые типы языка реализованы *структурами*. В библиотеке FCL имеются и другие *встроенные структуры*. Рассмотрим в качестве примера *структуры* `Point`, `PointF`, `Size`, `SizeF` и `Rectangle`, находящиеся в пространстве имен `System.Drawing` и активно используемые при работе с графическими объектами. Первые четыре *структуры* имеют два открытых поля `X` и `Y` (`Height` и `Width`), задающие для точек - *структур* `Point` и `PointF` - координаты, целочисленные или в форме с плавающей точкой. Для размеров - *структур* `Size` и `SizeF` - они задают высоту и ширину, целочисленными значениями или в форме с плавающей точкой. *Структуры* `Point` и `Size` позволяют задать прямоугольную область - *структуру* `Rectangle`. Конструктору прямоугольника можно передать в качестве аргументов две *структуры* - точку, задающую координаты левого верхнего угла прямоугольника, и размер - высоту и ширину прямоугольника.

Между четырьмя *структурами* определены взаимные преобразования: точки можно преобразовать в размеры и наоборот, сложение и вычитание определено над точками и размерами, но не над точками, плавающий тип которых разными способами можно привести к целому. Ряд операций над этими *структурами* продемонстрирован в следующем примере:

```
public void TestPointAndSize()
{
   Point pt1 = new Point(3,5), pt2 = new Point(7,10), pt3;
   PointF pt4 = new PointF(4.55f,6.75f);
   Size sz1 = new Size(10,20), sz2;
   SizeF sz3 = new SizeF(10.3f, 20.7f);
   pt3 = Point.Round(pt4);
```

```
   sz2 = new Size(pt1);
   Console.WriteLine ("pt1: " + pt1);
   Console.WriteLine ("sz2 =new Size(pt1): " + sz2);
   Console.WriteLine ("pt4: " + pt4);
   Console.WriteLine ("pt3 =Point.Round(pt4): " + pt3);
   pt1.Offset(5,7);
   Console.WriteLine ("pt1.Offset(5,7): " + pt1);
   Console.WriteLine ("pt2: " + pt2);
   pt2 = pt2+ sz2;
   Console.WriteLine ("pt2= pt2+ sz2: " + pt2);
}//TestPointAndSize
```

Результаты его выполнения показаны на рис. 17.1

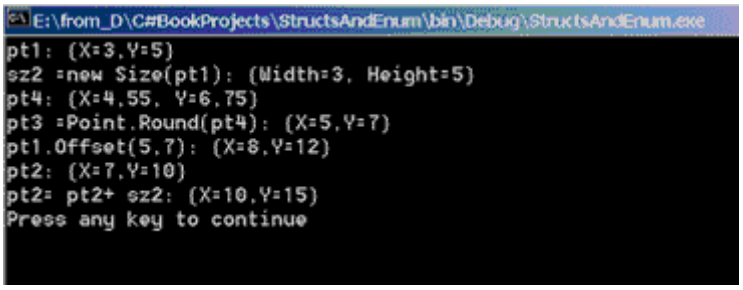

Рис. 17.1. Операции над точками и размерами

Отметим, что метод `ToString`, определенный для этих *структур*, выдает строку со значениями полей в приемлемой для восприятия форме.

Еще раз о двух семантиках присваивания

В заключение разговора о *ссылочных* и *развернутых типах* построим класс `CPoint`, являющийся полным аналогом *структуры* `Point`. Не буду приводить описание этого класса - надеюсь, оно достаточно понятно. Ограничусь примером, в котором аналогичные действия выполняются над объектами, принадлежащими *структуре* `Point` и классу `CPoint`:

```
public void TestTwoSemantics()
{
   Console.WriteLine("Структуры: присваивание
      развернутого типа!");
   Point pt1 = new Point(3,5), pt2;
   pt2 = pt1;
   Console.WriteLine ("pt1: " + pt1);
   Console.WriteLine ("pt2=pt1: " + pt2);
   pt1.X +=10;
   Console.WriteLine ("pt1.X =pt1.X +10: " + pt1);
   Console.WriteLine ("pt2: " + pt2);
   Console.WriteLine("Классы: присваивание ссылочного типа!");
   CPoint cpt1 = new CPoint(3,5), cpt2;
   cpt2 = cpt1;
   Console.WriteLine ("cpt1: " + cpt1);
   Console.WriteLine ("cpt2=cpt1: " + cpt2);
   cpt1.X +=10;
   Console.WriteLine ("cpt1.X =cpt1.X +10: " + cpt1);
   Console.WriteLine ("cpt2: " + cpt2);
}
```

Результаты вычислений показаны на рис. 17.2.

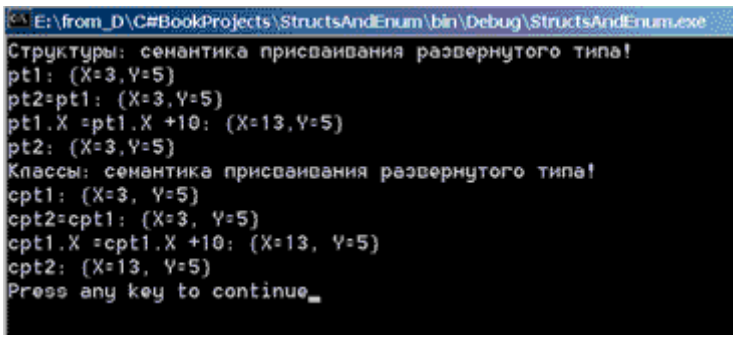

Рис. 17.2. Две семантики присваивания

Действия над объектами `Point` и `CPoint` выполняются аналогичные а результаты получаются разные: в конце вычислений `pt1` и `pt2` различны, а `cpt1` и `cpt2` совпадают.

Перечисления

Перечисление - это частный случай класса, класс, заданный без собственных методов. *Перечисление* задает конечное множество возможных значений, которые могут получать объекты класса *перечисление*. Поскольку у *перечислений* нет собственных методов, то синтаксис объявления этого класса упрощается - остается обычный заголовок и тело класса, содержащее список возможных значений. Вот формальное определение синтаксиса *перечислений*:

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

Описание атрибутов отложим на последующие лекции. Модификаторами могут быть четыре известных модификатора доступа и модификатор new. Ключевое слов enum говорит, что определяется частный случай *класса* - *перечисление*. Список возможных значений задает те значения, которые могут получать объекты этого класса. Возможные значения должны быть идентификаторами; но допускаются в их написании и буквы русского алфавита. Можно указать также базовый для *перечисления* класс.

Дело в том, что значения, заданные списком, проецируются на плотное подмножество базового класса. Реально значения объектов *перечисления* в памяти задаются значениями базового класса, так же, как значения класса `bool` реально представлены в памяти нулем и единицей, а не константами `true` и `false`, удобными для их использования программистами в тексте программ. По умолчанию, базовым классом является класс `int`, а подмножество проекции начинается с нуля. Но при желании можно изменить интервал представления и сам базовый класс. Естественно, на базовый класс накладывается ограничение. Он должен быть одним из встроенных классов, задающих счетное множество (`int`, `byte`, `long`, другие счетные типы). Единственное исключение из этого правила - нельзя выбирать класс `char` в качестве базового класса. Как правило, принятый по умолчанию выбор базового класса и его подмножества вполне приемлем в большинстве ситуаций.

Приведу примеры объявлений *классов-перечислений*:

```
public enum Profession{teacher, engineer, businessman};
public enum MyColors {red, blue, yellow, black, white};
public enum TwoColors {black, white};
public enum Rainbow {красный, оранжевый, желтый, зеленый, голубой, синий,
фиолетовый};
public enum Sex: byte {man=1, woman};
public enum Days:long {Sun,Mon,Tue,Wed,Thu, Fri, Sat};
```

Вот несколько моментов, на которые следует обратить внимание при объявлении *перечислений*:

- как и другие классы, *перечисления* могут быть объявлены непосредственно в пространстве имен проекта или могут быть вложены в описание класса. Последний вариант часто применяется, когда *перечисление* используется в одном классе и имеет атрибут доступа `private`;
- константы разных *перечислений* могут совпадать, как в *перечислениях* `MyColors` и `TwoColors`. Имя константы всегда уточняется именем *перечисления*;
- константы могут задаваться словами русского языка, как в *перечислении* `Rainbow`;
- разрешается задавать базовый класс *перечисления*. Для *перечисления* `Days` базовым классом задан класс `long`;
- разрешается задавать не только базовый класс, но и указывать начальный элемент подмножества, на которое проецируется множество значений *перечисления*. Для *перечисления* `Sex` в качестве базового класса выбран класс `byte`, а подмножество значений начинается с 1, так что хранимым значением константы `man` является 1, а `woman` - 2.

Рассмотрим теперь пример *работы с объектами - экземплярами различных перечислений*:

```
public void TestEnum()
{
   //MyColors color1 = new MyColors(MyColors.blue);
   MyColors color1= MyColors.white;
   TwoColors color2;
   color2 = TwoColors.white;
   //if(color1 != color2) color2 = color1;
   if(color1.ToString() != color2.ToString())
      Console.WriteLine ("Цвета разные: {0}, {1}",
         color1, color2);
   else Console.WriteLine("Цвета одинаковые: {0},
      {1}",color1, color2);
   Rainbow color3;
   color3 = (Rainbow)3;
   if (color3 != Rainbow.красный)color3 =Rainbow.красный;
   int num = (int)color3;
   Sex who = Sex.man;
   Days first_work_day = (Days)(long)1;
   Console.WriteLine ("color1={0}, color2={1},
      color3={2}",color1, color2, color3);
   Console.WriteLine ("who={0}, first_work_day={1}",
      who,first_work_day);
}
```

Данный пример иллюстрирует следующие **особенности** *работы с объектами перечислений*:

- объекты *перечислений* нельзя создавать в объектном стиле с использованием операции new, поскольку *перечисления* не имеют конструкторов;
- объекты можно объявлять с явной инициализацией, как `color1`, или с отложенной инициализацией, как `color2`. При объявлении без явной инициализации объект получает значение первой константы *перечисления*, так что `color2` в момент объявления получает значение `black`;
- объекту можно присвоить значение, которое задается константой *перечисления*, уточненной именем *перечисления*, как для `color1` и `color2`. Можно также задать значение базового типа, приведенное к типу *перечисления*, как для `color3`;

- нельзя сравнивать объекты разных *перечислений*, например `color1` и `color2`, но можно сравнивать строки, возвращаемые методом `ToString`, например `color1.ToSting()` и `color2.Tostring()`;
- существуют явные взаимно обратные преобразования констант базового типа и констант *перечисления*;
- Метод `ToString`, наследованный от класса `Object`, возвращает строку, задающую константу *перечисления*.

Персоны и профессии

Рассмотрим еще один пример *работы с перечислениями*, приближенный к реальности. Добавим в класс `Person`, рассмотренный в предыдущей лекции 16, поле, определяющее профессию персоны. Вполне разумно иметь *перечисление*, например, `Profession`, задающее список возможных профессий. Сделаем это поле, как обычно, закрытым, а доступ к нему обеспечим соответствующим свойством:

```
Profession prof;
public Profession Prof
{
   get {return (prof);}
   set {prof = value;}
}
```

Добавим еще в класс `Person` метод `Analysis`, анализирующий профессию, организуя традиционный разбор случаев и принимая решение на каждой ветви, в данном примере - выводя соответствующий текст:

```
public void Analysis()
{
   switch (prof)
      {
      case Profession.businessman:
         Console.WriteLine ("профессия: бизнесмен");
         break;
      case Profession.teacher:
         Console.WriteLine ("профессия: учитель");
         break;
      case Profession.engineer:
         Console.WriteLine ("профессия: инженер");
         break
      default:
         Console.WriteLine ("профессия: неизвестна");
         break;
      }
}
```

Приведу простой тестирующий пример работы с объектом `Person` и его профессией:

```
public void TestProfession()
{
   Person pers1 = new Person ("Петров");
   pers1.Prof = Profession.teacher;
   pers1.Analysis();
}
```

Результаты *работы с объектами перечислений*, полученные при вызове тестов `TestEnum` и `TestProfession`, показаны на рис. 17.3.

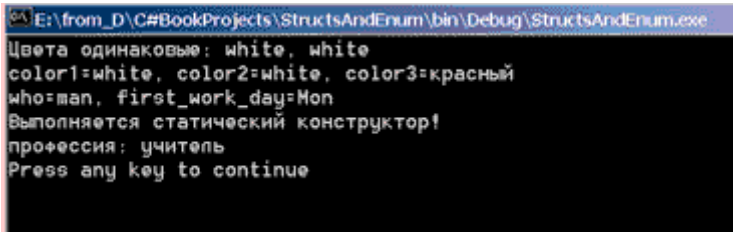

Рис. 17.3. Результаты работы с перечислениями

18. Лекция: Отношения между классами. Клиенты и наследники

Отношения между классами

Каждый класс, как не раз отмечалось, играет две роли: он является модулем - архитектурной единицей, и он имеет содержательный смысл, определяя некоторый тип данных. Но классы программной системы - это ансамбль, в котором классы, играя свои роли, не являются независимыми - все они находятся в определенных отношениях друг с другом. Два основных типа отношений между классами определены в ОО-системах. Первое отношение *"клиенты и поставщики"*, называется часто *клиентским отношением* или *отношением вложенности (встраивания)*. Второе отношение *"родители и наследники"* называется *отношением наследования*.

Определение 1. Классы `A` и `B` находятся в отношении "***клиент-поставщик***", если одним из полей класса `B` является объект класса `A`. Класс `A` называется поставщиком класса `B`, класс `B` называется клиентом класса `A`.

Определение 2. Классы `A` и `B` находятся в отношении "***родитель - наследник***", если при объявлении класса `B` класс `A` указан в качестве родительского класса. Класс `A` называется родителем класса `B`, класс `B` называется наследником класса `A`.

Оба отношения - *наследования* и *вложенности* - являются транзитивными. Если `B` - клиент `A` и `C` - клиент `B`, то отсюда следует, что `C` - клиент `A`. Если `B` - наследник `A` и `C` - наследник `B`, то отсюда следует, что `C` - наследник `A`.

Определения 1 и 2 задают прямых или непосредственных клиентов и поставщиков, прямых родителей и наследников. Вследствие транзитивности необходимо ввести понятие уровня. Прямые клиенты и поставщики, прямые родители и наследники относятся к соответствующему уровню 1 (клиенты уровня 1, поставщики уровня 1 и так далее). Затем следует рекурсивное определение: прямой клиент клиента уровня `k` относится к уровню `k+1`.

Для *отношения наследования* используется терминология, заимствованная из естественного языка. Прямые классы-наследники часто называются ***сыновними или дочерними классами***. Непрямые родители называются предками, а их непрямые наследники - *потомками*.

Замечу, что цепочки *вложенности* и *наследования* могут быть достаточно длинными. На практике вполне могут встречаться цепочки длины 10. Например, библиотечные классы, составляющие систему Microsoft Office, полностью построены на *отношении вложенности*. При программной работе с объектами Word можно начать с объекта, задающего приложение Word, и добраться до объекта, задающего отдельный символ в некотором слове некоторого предложения одного из открытых документов Word. Для выбора нужного объекта можно задать такую цепочку: приложение Word - коллекция документов - документ - область документа - коллекция абзацев - абзац - коллекция предложений - предложение - коллекция слов - слово - коллекция символов - символ. В этой цепочке каждому понятию соответствует класс библиотеки Microsoft Office, где каждая пара соседствующих классов связана отношением "*поставщик-клиент*".

Классы библиотеки FCL связаны как *отношением вложенности*, так и *отношением наследования*. Длинные цепочки *наследования* достаточно характерны для классов этой библиотеки.

Отношения "является" и "имеет"

При проектировании классов часто возникает вопрос, какое же отношение между классами нужно построить. Рассмотрим совсем простой пример двух классов - `Square` и `Rectangle`, описывающих квадраты и прямоугольники. Наверное, понятно, что эти классы следует связать скорее *отношением наследования*, чем *вложенности*; менее понятным остается вопрос, а какой из этих двух классов следует сделать родительским. Еще один пример двух классов - `Car` и `Person`, описывающих автомобиль и персону. Какими отношениями с этими классами должен быть связан класс `Person_of_Car`, описывающий владельца машины? Может ли он быть наследником обоих классов? Найти правильные ответы на эти вопросы проектирования классов помогает понимание того, что отношение "*клиент-поставщик*" задает *отношение "имеет" ("has")*, а *отношение наследования* задает *отношение "является" ("is a")*. В случае классов `Square` и `Rectangle` понятно, что каждый объект квадрат "является" прямоугольником, поэтому между этими классами имеет место *отношение наследования*, и родительским классом является класс `Rectangle`, а класс `Square` является его *потомком*.

В случае автомобилей, персон и владельцев авто также понятно, что владелец "имеет" автомобиль и "является" персоной. Поэтому класс `Person_of_Car` является клиентом класса `Car` и наследником класса `Person`.

Отношение вложенности

Рассмотрим два класса `A` и `B`, связанных *отношением вложенности*. Оба класса применяются для демонстрации идей и потому устроены просто, не неся особой смысловой нагрузки. Пусть класс-поставщик `A` уже построен. У класса два поля, конструктор, один статический и один динамический метод. Вот его текст:

```
public class ClassA
{
   public ClassA(string f1, int f2)
   {
      fieldA1 = f1; fieldA2 = f2;
   }
   public string fieldA1;
   public int fieldA2;
   public void MethodA()
   {
      Console.WriteLine( "Это класс A");
      Console.WriteLine ("поле1 = {0}, поле2 = {1}",
         fieldA1, fieldA2);
   }
   public static void StatMethodA()
   {
      string s1 = "Статический метод класса A";
      string s2 = "Помните: 2*2 = 4";
      Console.WriteLine(s1 + " ***** " + s2);
   }
}
```

Построим теперь класс `B` - клиента класса `A`. Класс будет устроен похожим образом, но в дополнение будет иметь одним из своих полей объект `inner` класса `A`:

```
public class ClassB
{
   public ClassB(string f1A, int f2A, string f1B, int f2B)
   {
      inner = new ClassA(f1A, f2A);
      fieldB1 = f1B; fieldB2 = f2B;
   }
   ClassA inner;
```

```
   public string fieldB1;
   public int fieldB2;
   public void MethodB1()
   {
      inner.MethodA();
      Console.WriteLine( "Это класс B");
      Console.WriteLine ("поле1 = {0}, поле2 = {1}",
         fieldB1, fieldB2);
   }
```

Обратите внимание: конструктор клиента (класса `B`) отвечает за инициализацию полей класса, поэтому он должен создать объект поставщика (класса `A`), вызывая, как правило, конструктор поставщика. Если для создания объектов поставщика требуются аргументы, то они должны передаваться конструктору клиента, как это сделано в нашем примере.

После того как конструктор создал поле - объект поставщика - методы класса могут использовать этот объект, вызывая доступные клиенту методы и поля класса поставщика. Метод класса `B` - `MethodB1` начинает свою работу с вызова: `inner.MethodA`, используя сервис, поставляемый методом класса `A`.

Расширение определения клиента класса

До сих пор мы говорили, что клиент содержит поле, представляющее объект класса поставщика. Это частая, но не единственная ситуация, когда класс является клиентом другого класса. Возможна ситуация, когда метод клиентского класса локально создает объект поставщика, вызывает его методы в собственных целях, но по завершении метода локальный объект заканчивает свою жизнь. Еще одна возможная ситуация - когда объекты поставщика вообще не создаются ни конструктором, ни методами класса клиента, но клиент вызывает статические методы класса поставщика. Оба эти варианта демонстрируют следующие два метода класса `B`:

```
public void MethodB2()
{
   ClassA loc = new ClassA("локальный объект A",77);
   loc.MethodA();
}
public void MethodB3()
{
   ClassA.StatMethodA();
}
```

Дадим теперь расширенное определение клиента.

Определение 3. Класс `B` называется клиентом класса `A`, если в классе `B` создаются объекты класса `A` - поля или локальные переменные - или вызываются статические поля или методы класса `A`.

Отношения между клиентами и поставщиками

Что могут делать клиенты и что могут делать поставщики? Класс-поставщик создает свойства (поля) и сервисы (методы), предоставляемые своим клиентам. Клиенты создают объекты поставщика. Вызывая доступные им методы и поля объектов, они управляют работой созданных объектов поставщика. Клиенты не могут ни повлиять на поведение методов поставщика, ни изменить состав предоставляемых им полей и методов, они не могут вызывать закрытые поставщиком поля и методы класса.

Класс-поставщик интересен клиентам своей открытой частью, составляющей интерфейс класса. Но большая часть класса может быть закрыта для клиентов - им незачем вникать в детали представления и в детали реализации. Сокрытие информации вовсе не означает, что разработчики класса не должны быть знакомы с тем, как все реализовано, хотя иногда и такая цель преследуется. В общем случае сокрытие означает, что классы-клиенты строят свою реализацию, основываясь только на интерфейсной части класса-поставщика. Поставщик закрывает поля и часть методов класса от клиентов, задавая для них атрибут доступа `private` или `protected`. Он может некоторые классы считать привилегированными, предоставляя им методы и поля, недоступные другим классам. В этом случае поля и методы, предназначенные для таких vip-персон, снабжаются атрибутом доступа `internal`, а классы с привилегиями должны принадлежать одной сборке.

В заключение построим тест, проверяющий работу с объектами классов `A` и `B`:

```
public void TestClientSupplier()
{
   ClassB objB = new ClassB("AA",22, "BB",33);
   objB.MethodB1();
   objB.MethodB2();
   objB.MethodB3();
}
```

Результаты работы этого теста показаны на рис. 18.1.

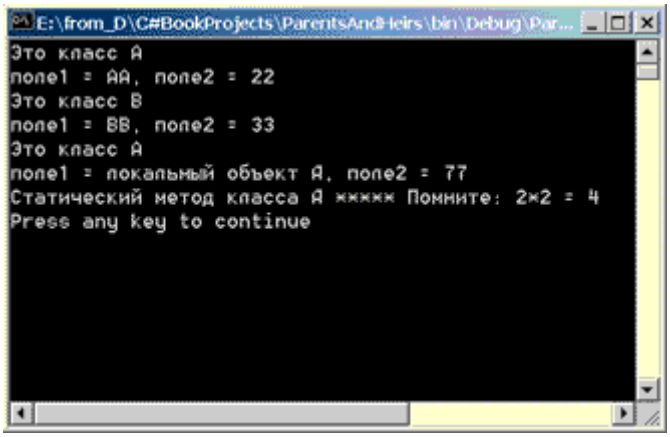

Рис. 18.1. Клиенты и поставщики

Сам себе клиент

Зададимся вопросом, может ли класс быть *сам себе клиентом*, другими словами, может ли поле класса быть объектом описываемого класса? Другой, не менее интересный вопрос - могут ли два класса быть одновременно клиентами и поставщиками друг для друга? Ответы на оба вопросы положительны, и подобные ситуации типичны и не являются какой-либо экзотикой.

Первая ситуация характерна для динамических структур данных. Элемент односвязного списка имеет поле, представляющее элемент односвязного списка; элемент двусвязного списка имеет два таких поля; узел двоичного дерева имеет два поля, представляющих узлы двоичного дерева. Эта ситуация характерна не только для рекурсивно определяемых структур данных. Вот еще один типичный пример. В классе `Person` могут быть заданы два поля - `Father` и `Mother`, задающие родителей персоны, и массив `Children`. Понятно, что все эти объекты могут быть того же класса `Person`.

Не менее часто встречается ситуация, когда классы имеют поля, взаимно ссылающиеся друг на друга. Типичным примером могут служить классы `Man` и `Woman`, первый из которых имеет поле `wife` класса `Woman`, а второй - поле `husband` класса `Man`.

Заметьте, классы устроены довольно просто - их тексты понятны, отношения между классами очевидны. А вот динамический мир объектов этих классов может быть довольно сложным, отношения между объектами могут быть запутанными; для объектов характерны не только любовные треугольники, но и куда более сложные фигуры.

Наследование

Мощь ООП основана на *наследовании*. Когда построен полезный класс, то он может многократно использоваться. Повторное использование - это одна из главных целей ООП. Но и для хороших классов неизбежно наступает момент, когда необходимо расширить возможности класса, придать ему новую функциональность, изменить интерфейс. Всякая попытка изменять сам работающий класс чревата большими неприятностями - могут перестать работать прекрасно работавшие программы, многим клиентам класса вовсе не нужен новый интерфейс и новые возможности. Здесь-то и приходит на выручку *наследование*. Существующий класс не меняется, но создается его *потомок*, продолжающий дело отца, только уже на новом уровне.

Класс-*потомок* наследует все возможности родительского класса - все поля и все методы, открытую и закрытую часть класса. Правда, не ко всем полям и методам класса возможен прямой доступ *потомка*. Поля и методы родительского класса, снабженные атрибутом `private`, хотя и наследуются, но по-прежнему остаются закрытыми, и методы, создаваемые *потомком*, не могут к ним обращаться напрямую, а только через методы, *наследованные* от родителя. Единственное, что не наследует *потомок* - это конструкторы родительского класса. Конструкторы *потомок* должен создавать сам. В этом есть некоторая разумная идея, и я позже поясню ее суть.

Рассмотрим класс, названный `Found`, играющий роль родительского класса. У него есть обычные поля, конструкторы и методы, один из которых снабжен новым модификатором `virtual`, ранее не появлявшимся в классах, другой - модификатором `override`:

```
public class Found
{
   //поля
   protected string name;
   protected int credit;
   public Found()
   {
   }
   public Found(string name, int sum)
   {
      this.name = name; credit = sum;
   }
   public virtual void VirtMethod()
   {
      Console.WriteLine ("Отец: " + this.ToString() );
   }
   public override string ToString()
   {
      return(String.Format("поля: name = {0}, credit = {1}",
         name, credit));
   }
   public void NonVirtMethod()
   {
      Console.WriteLine ("Мать: " + this.ToString() );
```

```
    }
    public void Analysis()
    {
        Console.WriteLine ("Простой анализ");
    }
    public void Work()
    {
        VirtMethod();
        NonVirtMethod();
        Analysis();
    }
}
```

Заметьте, класс `Found`, как и все классы, по умолчанию является наследником класса `Object`, его *потомки* наследуют методы этого класса уже не напрямую, а через методы родителя, который мог переопределить методы класса `Object`. В частности, класс `Found` переопределил метод `ToString`, задав собственную реализацию возвращаемой методом строки, которая связывается с объектами класса. Как часто делается, в этой строке отображаются значения полей объекта данного класса. На переопределение родительского метода `ToString` указывает модификатор метода `override`.

Класс `Found` закрыл свои поля для клиентов, но открыл для *потомков*, снабдив их модификатором доступа `protected`.

Создадим теперь класс `Derived` - *потомка* класса `Found`. В простейшем случае объявление класса может выглядеть так:

```
public class Derived:Found
{
}
```

Тело класса `Derived` пусто, но это вовсе не значит, что объекты этого класса не имеют полей и методов: они "являются" объектами класса `Found`, наследуя все его поля и методы (кроме конструктора) и поэтому могут делать все, что могут делать объекты родительского класса.

Вот пример работы с объектами родительского и производного класса:

```
public void TestFoundDerived()
{
    Found bs = new Found ("father", 777);
    Console.WriteLine("Объект bs вызывает методы базового
        класса");
    bs.VirtMethod();
    bs.NonVirtMethod();
    bs.Analysis();
    bs.Work();
    Derived der = new Derived();
    Console.WriteLine("Объект der вызывает методы класса
        потомка");
    der.VirtMethod();
    der.NonVirtMethod();
    der.Analysis();
    der.Work();
}
```

Результаты работы этой процедуры показаны на рис. 18.2.

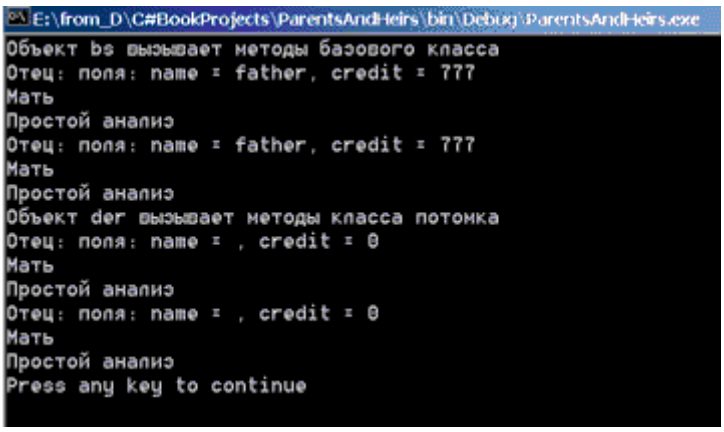

Рис. 18.2. Объект потомка наследует поля и методы родителя

В чем отличие работы объектов `bs` и `der`? Поскольку класс-потомок `Derived` ничего самостоятельно не определял, то он наследовал все поля и методы у своего родителя, за исключением конструкторов. У этого класса имеется собственный конструктор без аргументов, задаваемый по умолчанию. При создании объекта `der` вызывался его собственный конструктор по умолчанию, инициализирующий поля класса значениями по умолчанию. Об особенностях работы конструкторов *потомков* скажу чуть позже, сейчас же упомяну лишь, что конструктор по умолчанию *потомка* вызывает конструктор без аргументов своего родителя, поэтому для успеха работы родитель должен иметь такой конструктор. Заметьте, поскольку родитель не знает, какие у него могут быть *потомки*, то желательно конструктор без аргументов включать в число конструкторов класса, как это сделано для класса `Found`.

Добавление полей потомком

Ничего не делающий самостоятельно *потомок* не эффективен, от него мало проку. Что же может делать *потомок*? Прежде всего, он может добавить новые свойства - поля класса. Заметьте, *потомок* не может ни отменить, ни изменить модификаторы или типы полей, наследованных от родителя - он может только добавить собственные поля.

Модифицируем наш класс `Derived`. Пусть он добавляет новое поле класса, закрытое для клиентов этого класса, но открытое для его *потомков*:

```
protected int debet;
```

Напомню, хорошей стратегией является стратегия "ничего не скрывать от *потомков*". Какой родитель знает, что именно из сделанного им может понадобиться *потомкам*?

Конструкторы родителей и потомков

Каждый класс должен позаботиться о *создании собственных конструкторов*. Он не может в этом вопросе полагаться на родителя, поскольку, как правило, добавляет собственные поля, о которых родитель ничего не может знать. Конечно, если не задать конструкторов класса, то будет добавлен конструктор по умолчанию, инициализирующий все поля значениями по умолчанию, как это мы видели в предыдущем примере. Но это редкая ситуация. Чаще всего, класс создает собственные конструкторы и, как правило, не один, задавая разные варианты инициализации полей.

При *создании конструкторов* классов *потомков* есть одна важная особенность. Всякий конструктор создает объект класса - структуру, содержащую поля класса. Но *потомок*, прежде чем создать собственный объект, вызывает конструктор родителя, создавая родительский объект, который затем будет дополнен полями *потомка*. Ввиду транзитивности этого процесса, конструктор родителя вызывает конструктор своего

родителя, и этот процесс продолжается, пока первым делом не будет создан объект прародителя.

Вызов конструктора родителя происходит не в теле конструктора, а в заголовке, пока еще не создан объект класса. Для вызова конструктора используется ключевое слово base, именующее родительский класс. Как это делается, покажу на примере конструкторов класса `Derived`:

```
public Derived() {}
public Derived(string name, int cred, int deb):base (name,cred)
{
   debet = deb;
}
```

Для конструктора без аргументов вызов аналогичного конструктора родителя подразумевается по умолчанию. Для конструкторов с аргументами вызов конструктора с аргументами родительского класса должен быть явным. Этот вызов синтаксически следует сразу за списком аргументов конструктора, будучи отделен от этого списка символом двоеточия. Конструктору *потомка* передаются все аргументы, необходимые для инициализации полей, часть из которых передаются конструктору родителя для инициализации родительских полей.

Итак, вызов конструктора - *потомка* приводит к цепочке вызовов конструкторов - предков, заканчивающейся вызовом конструктора прародителя. Затем в обратном порядке создаются объекты, начиная с объекта прародителя, и выполняются тела соответствующих конструкторов, инициализирующие поля и выполняющие другую работу этих конструкторов. Последним создается объект *потомка* и выполняется тело конструктора *потомка*.

Добавление методов и изменение методов родителя

Потомок может создать новый *собственный метод* с именем, отличным от имен наследуемых методов. В этом случае никаких особенностей нет. Вот пример такого метода, создаваемого в классе `Derived`:

```
public void DerivedMethod()
{
   Console.WriteLine("Это метод класса Derived");
}
```

В отличие от неизменяемых полей классов - предков, класс - *потомок* может изменять наследуемые им методы. Если *потомок* создает метод с именем, совпадающим с именем метода предков, то возможны три ситуации:

- перегрузка метода. Она возникает, когда сигнатура создаваемого метода отличается от сигнатуры наследуемых методов предков. В этом случае в классе *потомка* будет несколько перегруженных методов с одним именем, и вызов нужного метода определяется обычными правилами перегрузки методов;
- переопределение метода. Метод родителя в этом случае должен иметь модификатор `virtual` или `abstract`. Эта наиболее интересная ситуация подробно будет рассмотрена в следующих разделах нашей лекции. При переопределении сохраняется сигнатура и модификаторы доступа наследуемого метода;
- скрытие метода. Если родительский метод не является виртуальным или *абстрактным*, то *потомок* может создать новый метод с тем же именем и той же сигнатурой, скрыв родительский метод в данном контексте. При вызове метода предпочтение будет отдаваться методу *потомка*, а имя наследуемого метода будет скрыто. Это не означает, что оно становится недоступным. Скрытый

родительский метод всегда может быть вызван, если при вызове уточнить ключевым словом `base` имя метода.

Метод потомка, скрывающий метод родителя, следует сопровождать модификатором new, указывающим на новый метод. Если этот модификатор опущен, но из контекста ясно, что речь идет о новом методе, то выдается предупреждающее сообщение при компиляции проекта.

Вернемся к нашему примеру. Класс `Found` имел в своем составе метод `Analysis`. Его *потомок* класс `Derived` создает свой *собственный метод* анализа, скрывая метод родителя:

```
new public void Analysis()
{
   base.Analysis();
   Console.WriteLine("Сложный анализ");
}
```

Если модификатор new опустить, он добавится по умолчанию с выдачей предупреждающего сообщения о скрытии метода родителя. Как компилятор узнает, что в этой ситуации речь идет о новом методе? Причины понятны. С одной стороны, родительский метод не имеет модификаторов `virtual` или `abstract`, поэтому речь не идет о переопределении метода. С другой стороны, в родительском классе уже есть метод с данным именем и сигнатурой, и поскольку в классе не могут существовать два метода с одинаковой сигнатурой, то речь может идти только о новом методе класса, скрывающем родительский метод. Несмотря на "интеллект" транслятора, хороший стиль программирования требует явного указания модификатора `new` в подобных ситуациях.

Заметьте, *потомок* строит свой анализ на основе метода, наследованного от родителя, вызывая первым делом скрытый родительский метод.

Рассмотрим случай, когда *потомок* добавляет перегруженный метод. Вот пример, когда *потомок* класса `Derived` - класс `ChildDerived` создает свой метод анализа, изменяя сигнатуру метода `Analysis`:

```
public void Analysis(int level)
{
   base.Analysis();
   Console.WriteLine("Анализ глубины {0}", level);
}
```

Большой ошибки не будет, если указать модификатор `new` и в этом случае, но будет выдано предупреждающее сообщение, что модификатор может быть опущен, поскольку сокрытия родительского метода не происходит.

Статический контроль типов и динамическое связывание

Рассмотрим семейство классов `A1, A2, ... An`, связанных *отношением наследования*. Класс `Ak+1` является прямым *потомком* класса `Ak`. Пусть создана последовательность объектов `x1, x2, ... xn`, где `xk` - это объект класса `Ak`. Пусть в классе `A1` создан метод `M` с модификатором `virtual`, переопределяемый всеми *потомками*, так что в рамках семейства классов метод `M` существует в `n`-формах, каждая из которых задает реализацию метода, выбранную соответствующим *потомком*. Рассмотрим основную операцию, инициирующую объектные вычисления - вызов объектом метода класса:

```
x1.M(arg1, arg2, ... argN)
```

Контролем типов называется проверка каждого вызова, удостоверяющая, что:

- в классе `A1` объекта `x1` действительно имеется метод `M`;
- список фактических аргументов в точке вызова соответствует по числу и типам списку формальных аргументов метода `M`, заданного в классе `A1`.

Язык C#, как и большинство других языков программирования, позволяет выполнить эту проверку еще на этапе компиляции и в случае нарушений выдать сообщение об ошибке периода компиляции. *Контроль типов*, выполняемый на этапе компиляции, называется ***статическим контролем типов***. Некоторые языки, например Smalltalk, производят этот контроль динамически - непосредственно перед выполнением метода. Понятно, что ошибки, обнаруживаемые при динамическом *контроле типов*, трудно исправимы и *потому* приводят к более тяжелым последствиям. В таких случаях остается уповать на то, что система тщательно отлажена, иначе непонятно, что будет делать конечный пользователь, получивший сообщение о том, что вызываемого метода вообще нет в классе данного объекта.

Перейдем к рассмотрению связывания. Напомним, что в рассматриваемом семействе классов метод `M` полиморфен: имея одно и то же имя и сигнатуру, он существует в разных формах - для каждого класса задана собственная реализация метода. С другой стороны, из-за возможностей, предоставляемых односторонним присваиванием, в точке вызова неясно, с объектом какого класса семейства в данный момент связана сущность `x1` (вызову мог предшествовать такой оператор присваивания `if(B) x1 = xk;`).

Статическим связыванием называется связывание цели вызова и вызываемого метода на этапе компиляции, когда с сущностью связывается метод класса, заданного при объявлении сущности.

Динамическим связыванием называется связывание цели вызова и вызываемого метода на этапе выполнения, когда с сущностью связывается метод класса объекта, связанного с сущностью в момент выполнения.

При *статическом связывании* метод выбирается из класса сущности, при динамическом - из класса объекта, связанного с сущностью. Понятно, что на этапе компиляции возможно только *статическое связывание*, поскольку только в период выполнения можно определить, с объектом какого класса связана данная сущность. Это может быть класс любого из *потомков* класса сущности.

Какой же из видов связывания следует применять? *Статическое связывание* более эффективно в реализации, поскольку может быть сделано на этапе компиляции, так что при выполнении не потребуется никаких проверок. *Динамическое связывание* требует накладных расходов в период выполнения. Однако во многих случаях преимущества *динамического связывания* столь значительны, что о затратах не стоит и беспокоиться.

Уже достаточно давно разработан эффективный механизм реализации *динамического связывания*. Еще на этапе компиляции подготавливается так называемая таблица виртуальных методов, содержащая их адреса. Связывание объекта `xk` с принадлежащим ему методом `Mk` производится выбором соответствующего элемента из этой таблицы и выполняется ненамного сложнее, чем получение по индексу соответствующего элемента массива.

В языке C# принята следующая стратегия связывания. По умолчанию предполагается *статическое связывание*. Для того чтобы выполнялось *динамическое связывание*, метод родительского класса должен снабжаться модификатором `virtual` или `abstract`, а его *потомки* должны иметь модификатор `override`.

Три механизма, обеспечивающие полиморфизм

Под ***полиморфизмом*** в ООП понимают способность одного и того же программного текста `x.M` выполняться по-разному, в зависимости от того, с каким объектом связана сущность `x`. *Полиморфизм* гарантирует, что вызываемый метод M будет принадлежать классу объекта, связанному с сущностью `x`. В основе *полиморфизма*, характерного для семейства классов, лежат три механизма:

- одностороннее присваивание объектов внутри семейства классов; сущность, базовым классом которой является класс предка, можно связать с объектом любого из *потомков*. Другими словами, для введенной нами последовательности объектов `xk` присваивание `xi = xj` допустимо для всех `j >=i;`
- переопределение *потомком* метода, наследованного от родителя. Благодаря переопределению, в семействе классов существует совокупность *полиморфных методов* с одним именем и сигнатурой;
- *динамическое связывание*, позволяющее в момент выполнения вызывать метод, который принадлежит целевому объекту.

В совокупности это и называется *полиморфизмом семейства классов*. Целевую сущность часто называют ***полиморфной сущностью***, вызываемый метод - *полиморфным методом*, сам вызов - **полиморфным вызовом**.

Вернемся к нашему примеру с классами `Found`, `Derived`, `ChildDerived`. Напомню, в родительском классе определен виртуальный метод `VirtMethod` и переопределен виртуальный метод `ToString` родительского класса `object`. *Потомок* класса `Found` - класс `Derived` переопределяет эти методы:

```
public override void VirtMethod()
{
   Console.WriteLine("Сын: " + this.ToString());
}
public override string ToString()
{
   return(String.Format("поля: name = {0},
      credit = {1},debet ={2}",name, credit, debet));
}
```

Потомок класса `Derived` - класс `ChildDerived` не создает новых полей. Поэтому он использует во многом методы родителя. Его конструктор состоит из вызова конструктора родителя:

```
public ChildDerived(string name, int cred, int deb):base (name,cred, deb)
      {}
```

Нет и переопределения метода `Tostring`, поскольку используется реализация родителя. А вот метод `VirtMethod` переопределяется:

```
public override void VirtMethod()
{
   Console.WriteLine("внук: " + this.ToString());
}
```

В классе `Found` определены два невиртуальных метода `NonVirtmethod` и `Work`, наследуемые *потомками* `Derived` и `ChildDerived` без всяких переопределений. Вы ошибаетесь, если думаете, что работа этих методов полностью определяется базовым классом `Found`. *Полиморфизм* делает их работу куда более интересной. Давайте рассмотрим в деталях работу метода `Work`:

```
public void Work()
{
   VirtMethod();
   NonVirtMethod();
   Analysis();
}
```

При компиляции метода `Work` будет обнаружено, что вызываемый метод `VirtMethod` является виртуальным, поэтому для него будет применяться *динамическое связывание*. Это означает, что вопрос о вызове метода откладывается до момента, когда метод `Work` будет вызван объектом, связанным с `x`. Объект может принадлежать как классу `Found`, так и классам `Derived` и `ChildDerived`, в зависимости от класса объекта и будет вызван метод этого класса.

Для не виртуальных методов `NonVirtMethod` и `Analysis` будет применено *статическое связывание*, так что `Work` всегда будет вызывать методы, принадлежащие классу `Found`. Однако и здесь не все просто. Метод `NonVirtMethod`

```
public void NonVirtMethod()
{
   Console.WriteLine ("Мать: "+ this.ToString());
}
```

в процессе своей работы вызывает виртуальный метод `ToString`. Опять-таки, для метода `ToString` будет применяться *динамическое связывание*, и в момент выполнения будет вызываться метод класса объекта.

Что же касается метода `Analysis`, определенного в каждом классе, то всегда в процессе работы `Work` будет вызываться только родительский метод анализа из-за стратегии *статического связывания*.

Хочу обратить внимание на важный принципиальный момент. Вполне понятно, когда *потомки* вызывают методы родительского класса. *Потомкам* все известно о своих предках. Но благодаря *полиморфизму* методы родительского класса, в свою очередь, могут вызывать методы своих *потомков*, которых они совсем не знают и которые обычно и не написаны в момент создания родительского класса. Достигается это за счет того, что между родителями и *потомками* заключается жесткий контракт. *Потомок*, переопределяющий виртуальный метод, сохраняет сигнатуру метода, сохраняет атрибуты доступа, изменяя реализацию метода, но не форму его вызова.

Класс `Found`, создающий метод `Work`, говорит примерно следующее: "Я предоставляю этот метод своим *потомкам*. *Потомок*, вызвавший этот метод, должен иметь `VirtMethod`, выполняющий специфическую для *потомка* часть работы; конечно, *потомок* может воспользоваться и моей реализацией, но допустима и его собственная реализация. Затем часть работы выполняю я сам, но выдача информации об объекте определяется самим объектом. Заключительную часть работы, связанную с анализом, я *потомкам* не доверяю и делаю ее сам".

Пример работы с полиморфным семейством классов

Классы семейства с *полиморфными методами* уже созданы. Давайте теперь в клиентском классе `Testing` напишем метод, создающий объекты наших классов и вызывающий методы классов для объектов семейства:

```
public void TestFoundDerivedReal()
{
   Found bs = new Found ("father", 777);
   Console.WriteLine("Объект bs вызывает методы класса Found");
```

```
    bs.VirtMethod();
    bs.NonVirtMethod();
    bs.Analysis();
    bs.Work();
    Derived der = new Derived("child", 888, 555);
    Console.WriteLine("Объект der вызывает методы класса Derived");
    der.DerivedMethod();
    der.VirtMethod();
    der.NonVirtMethod();
    der.Analysis();
    der.Work();
    ChildDerived chider = new ChildDerived("grandchild", 999, 444);
    Console.WriteLine("Объект chider вызывает методы ChildDerived");
    chider.VirtMethod();
    chider.NonVirtMethod();
    chider.Analysis(5);
    chider.Work();
}
```

Результаты работы этого метода изображены на рис. 18.3.

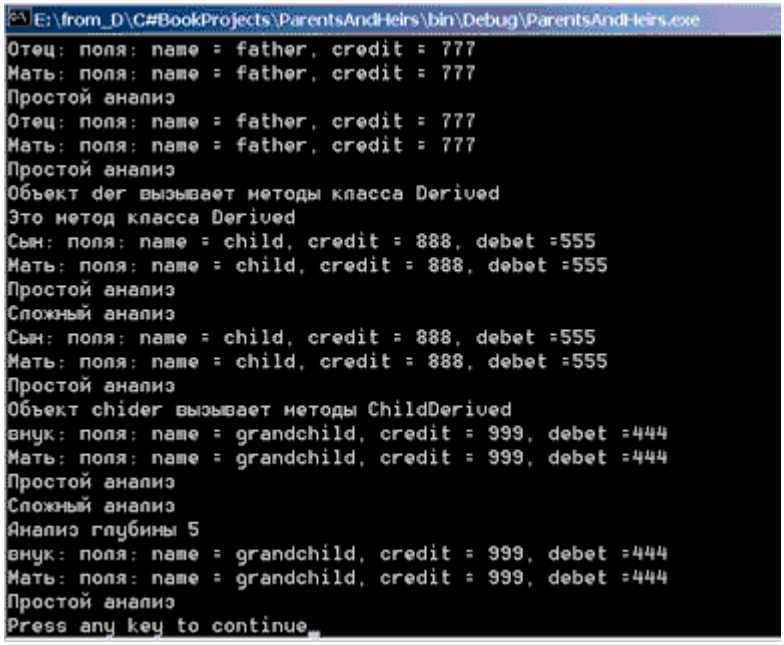

Рис. 18.3. Полиморфизм семейства классов

В последующих лекциях нам неоднократно встретятся более содержательные семейства классов с *полиморфизмом*, так что мы сумеем еще оценить мощь этого механизма ООП.

Абстрактные классы

С *наследованием* тесно связан еще один важный механизм проектирования семейства классов - *механизм абстрактных классов*. Начну с определений.

Класс называется *абстрактным*, если он имеет хотя бы один *абстрактный метод*.

Метод называется *абстрактным*, если при определении метода задана его сигнатура, но не задана реализация метода.

Объявление *абстрактных методов* и *абстрактных классов* должно сопровождаться модификатором `abstract`. Поскольку *абстрактные классы* не являются полностью определенными классами, то нельзя создавать объекты *абстрактных классов*.

Абстрактные классы могут иметь *потомков*, частично или полностью реализующих *абстрактные методы* родительского класса. *Абстрактный метод* чаще всего рассматривается как виртуальный метод, переопределяемый *потомком*, поэтому к ним применяется стратегия *динамического связывания*.

Абстрактные классы являются одним из важнейших инструментов объектно-ориентированного проектирования классов. К сожалению, я не могу входить в детали рассмотрения этой важной темы и ограничусь лишь рассмотрением самой идеи применения *абстрактного класса*. В основе любого класса лежит абстракция данных. *Абстрактный класс* описывает эту абстракцию, не входя в детали реализации, ограничиваясь описанием тех операций, которые можно выполнять над данными класса. Так, проектирование *абстрактного класса* `Stack`, описывающего стек, может состоять из рассмотрения основных операций над стеком и не определять, как будет реализован стек - списком или массивом. Два *потомка абстрактного класса* - `ArrayStack` и `ListStack` могут быть уже конкретными классами, основанными на различных представлениях стека.

Вот описание полностью *абстрактного класса* `Stack`:

```
public abstract class Stack
{
   public Stack()
   {}
   /// <summary>
   /// втолкнуть элемент item  в стек
   /// </summary>
   /// <param name="item"></param>
   public abstract void put(int item);
   /// <summary>
   /// удалить элемент в вершине стека
   /// </summary>
   public abstract void remove();
   /// <summary>
   /// прочитать элемент в вершине стека
   /// </summary>
   public abstract int item();
   /// <summary>
   /// определить, пуст ли стек
   /// </summary>
   /// <returns></returns>
   public abstract bool IsEmpty();
}
```

Описание класса содержит только сигнатуры методов класса и их спецификацию, заданную тегами `<summary>`. Построим теперь одного из *потомков* этого класса, реализация которого основана на списковом представлении. Класс `ListStack` будет *потомком абстрактного класса* `Stack` и клиентом класса `Linkable`, задающего элементы списка. Класс `Linkable` выглядит совсем просто:

```
public class Linkable
{
   public Linkable()
   { }
   public int info;
   public Linkable next;
}
```

В нем - два поля и конструктор по умолчанию. Построим теперь класс `ListStack`:

```
public class ListStack: Stack
```

```
{
   public ListStack()
   {
      top = new Linkable();
   }
   Linkable top;
   /// <summary>
   /// втолкнуть элемент item  в стек
   /// </summary>
   /// <param name="item"></param>
   public override void put(int item)
   {
      Linkable newitem = new Linkable();
      newitem.info = item;
      newitem.next = top;
      top = newitem;
   }
   /// <summary>
   /// удалить элемент в вершине стека
   /// </summary>
   public override void remove()
   {
      top = top.next;
   }
   /// <summary>
   /// прочитать элемент в вершине стека
   /// </summary>
   public override int item()
   {
      return(top.info);
   }
   /// <summary>
   /// определить, пуст ли стек
   /// </summary>
   /// <returns></returns>
   public override bool IsEmpty()
   {
      return(top.next == null);
   }
}
```

Класс имеет одно поле `top` класса `Linkable` и методы, наследованные от *абстрактного класса* `Stack`. Теперь, когда задано представление данных, нетрудно написать реализацию операций. Реализация операций традиционна для стеков и, надеюсь, не требует пояснений.

Приведу пример работы со стеком:

```
public void TestStack()
{
   ListStack stack = new ListStack();
   stack.put(7); stack.put(9);
   Console.WriteLine(stack.item());
   stack.remove();     Console.WriteLine(stack.item());
   stack.put(11); stack.put(13);
   Console.WriteLine(stack.item());
   stack.remove();     Console.WriteLine(stack.item());
   if(!stack.IsEmpty()) stack.remove();
   Console.WriteLine(stack.item());
}
```

В результате работы этого теста будет напечатана следующая последовательность целых: 9, 7, 13, 11, 7.

Классы без потомков

Экзотическим, но иногда полезным видом классов являются классы, для которых запрещается строить классы-потомки путем *наследования*. При создании такого класса нет необходимости в выполнении над классом каких-либо болезненных операций. Вполне достаточно приписать классу *модификатор sealed* - он и запрещает построение *потомков*.

19. Лекция: Интерфейсы. Множественное наследование

Интерфейсы

Слово "*интерфейс*" многозначное и в разных контекстах оно имеет различный смысл. В данной лекции речь идет о понятии *интерфейса*, стоящем за ключевым словом interface. В таком понимании *интерфейс* - это частный случай класса. ***Интерфейс*** представляет собой полностью абстрактный класс, все методы которого абстрактны. От абстрактного класса *интерфейс* отличается некоторыми деталями в синтаксисе и поведении. Синтаксическое отличие состоит в том, что методы *интерфейса* объявляются без указания модификатора доступа. Отличие в поведении заключается в более жестких требованиях к потомкам. Класс, наследующий *интерфейс*, обязан полностью реализовать все методы *интерфейса*. В этом - отличие от класса, наследующего абстрактный класс, где потомок может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом. Но, конечно, не ради этих отличий были введены *интерфейсы* в язык C#. У них значительно более важная роль.

Введение в язык частных случаев усложняет его и свидетельствует о некоторых изъянах, для преодоления которых и вводятся частные случаи. Например, введение структур в язык C# позволило определять классы как развернутые типы. Конечно, проще было бы ввести в объявление класса соответствующий модификатор, позволяющий любой класс объявлять развернутым. Но этого сделано не было, а, следуя традиции языка C++, были введены структуры как частный случай классов.

Подробнее о развернутых и ссылочных типах см. лекцию 17.

Интерфейсы позволяют частично справиться с таким существенным недостатком языка, как отсутствие *множественного наследования* классов. Хотя реализация *множественного наследования* встречается с рядом проблем, его отсутствие существенно снижает выразительную мощь языка. В языке C# полного *множественного наследования* классов нет. Чтобы частично сгладить этот пробел, допускается *множественное наследование интерфейсов*. Обеспечить возможность классу иметь несколько родителей - один полноценный класс, а остальные в виде *интерфейсов*, - в этом и состоит основное *назначение интерфейсов*.

Отметим одно важное *назначение интерфейсов*. *Интерфейс* позволяет описывать некоторые желательные свойства, которыми могут обладать объекты разных классов. В библиотеке FCL имеется большое число подобных *интерфейсов*, с некоторыми из них мы познакомимся в этой лекции. Все классы, допускающие сравнение своих объектов, обычно наследуют *интерфейс* `IComparable`, реализация которого позволяет сравнивать объекты не только на равенство, но и на "больше", "меньше".

Две стратегии реализации интерфейса

Давайте опишем некоторый *интерфейс*, задающий дополнительные свойства объектов класса:

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

У этого *интерфейса* два метода, которые и должны будут реализовать все классы - наследники *интерфейса*. Заметьте, у методов нет модификаторов доступа.

Класс, наследующий *интерфейс* и реализующий его методы, может реализовать их явно, объявляя соответствующие методы класса открытыми. Вот пример:

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

Класс реализует методы *интерфейса*, делая их открытыми для клиентов класса и наследников. Другая стратегия реализации состоит в том, чтобы все или некоторые методы *интерфейса* сделать закрытыми. Для реализации этой стратегии класс, наследующий *интерфейс*, объявляет методы без модификатора доступа, что по умолчанию соответствует модификатору `private`, и уточняет имя метода именем *интерфейса*. Вот соответствующий пример:

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

Класс `ClainP` реализовал методы *интерфейса* `IProps`, но сделал их закрытыми и недоступными для вызова клиентов и наследников класса. Как же получить доступ к закрытым методам? Есть два способа решения этой проблемы:

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

В чем главное достоинство *обертывания*? Оно позволяет переименовывать методы *интерфейса*. Метод *интерфейса* со своим именем закрывается, а потом открывается под тем именем, которое класс выбрал для него. Вот пример *обертывания* закрытых методов в классе `ClainP`:

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

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

Преобразование к классу интерфейса

Создать объект класса *интерфейса* обычным путем с использованием конструктора и операции `new` нельзя. Тем не менее, можно объявить объект *интерфейсного* класса и связать его с настоящим объектом путем приведения (*кастинга*) объекта наследника к классу *интерфейса*. Это преобразование задается явно. Имея объект, можно вызывать методы *интерфейса* - даже если они закрыты в классе, для интерфейсных объектов они являются открытыми. Приведу соответствующий пример, в котором идет работа как с объектами классов `Clain`, `ClainP`, так и с объектами интерфейсного класса `IProps`:

```
public void TestClainIProps()
{
   Console.WriteLine("Объект класса Clain вызывает
      открытые методы!");
   Clain clain = new Clain();
   clain.Prop1(" свойство 1 объекта");
   clain.Prop2("Владимир", 44);
   Console.WriteLine("Объект класса IProps вызывает
      открытые методы!");
   IProps ip = (IProps)clain;
   ip.Prop1("интерфейс: свойство");
   ip.Prop2 ("интерфейс: свойство",77);
   Console.WriteLine("Объект класса ClainP вызывает
      открытые методы!");
   ClainP clainp = new ClainP();
   clainp.MyProp1(" свойство 1 объекта");
   clainp.MyProp2("Владимир", 44);
   Console.WriteLine("Объект класса IProps вызывает
      закрытые методы!");
   IProps ipp = (IProps)clainp;
   ipp.Prop1("интерфейс: свойство");
   ipp.Prop2 ("интерфейс: свойство",77);
}
```

Этот пример демонстрирует работу с классом, где все наследуемые методы *интерфейса* открыты, и с классом, закрывающим наследуемые методы *интерфейса*. Показано, как *обертывание* и *кастинг* позволяют добраться до закрытых методов класса. Результаты выполнения этой тестирующей процедуры приведены на рис. 19.1.

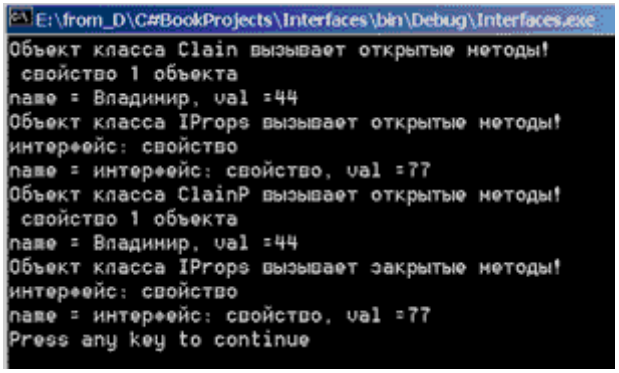

Рис. 19.1. Наследование интерфейса. Две стратегии

Проблемы множественного наследования

При *множественном наследовании* классов возникает ряд проблем. Они остаются и при *множественном наследовании интерфейсов*, хотя становятся проще. Рассмотрим две основные проблемы - *коллизию имен* и *наследование от общего предка*.

Коллизия имен

Проблема *коллизии имен* возникает, когда два или более *интерфейса* имеют методы с одинаковыми именами и сигнатурой. Сразу же заметим, что если имена методов совпадают, но сигнатуры разные, то это не приводит к конфликтам - при реализации у класса наследника просто появляются перегруженные методы. Но что следует делать классу-наследнику в тех случаях, когда сигнатуры методов совпадают? И здесь возможны *две стратегии* - *склеивание методов* и *переименование*.

Стратегия *склеивания* применяется тогда, когда класс - наследник *интерфейсов* - полагает, что разные *интерфейсы* задают один и тот же метод, единая реализация которого и должна быть обеспечена наследником. В этом случае наследник строит единственную общедоступную реализацию, соответствующую методам всех *интерфейсов*, которые имеют единую сигнатуру.

Другая стратегия исходит из того, что, несмотря на единую сигнатуру, методы разных *интерфейсов* должны быть реализованы по-разному. В этом случае необходимо переименовать конфликтующие методы. Конечно, *переименование* можно сделать в самих *интерфейсах*, но это неправильный путь: наследники не должны требовать изменений своих родителей - они сами должны меняться. *Переименование* методов *интерфейсов* иногда невозможно чисто технически, если *интерфейсы* являются *встроенными* или поставляются сторонними фирмами. К счастью, мы знаем, как производить *переименование* метода *интерфейса* в самом классе наследника. Напомню, для этого достаточно реализовать методы разных *интерфейсов* как закрытые, а затем открыть их с *переименованием*.

Итак, *коллизия имен* при *множественном наследовании интерфейсов* хотя и возможна, но легко разрешается. Разработчик класса может выбрать любую из двух возможных *стратегий*, наиболее подходящую для данного конкретного случая. Приведу пример двух *интерфейсов*, имеющих методы с одинаковой сигнатурой, и класса - наследника этих *интерфейсов*, применяющего разные стратегии реализации для конфликтующих методов.

```
public interface IProps
{
   void Prop1(string s);
   void Prop2 (string name, int val);
   void Prop3();
}
public interface IPropsOne
{
   void Prop1(string s);
   void Prop2 (int val);
   void Prop3();
}
```

У двух *интерфейсов* - по три метода с совпадающими именами, сигнатуры двух методов совпадают, а в одном случае различаются. Вот класс, наследующий оба *интерфейса*:

```
public class ClainTwo:IProps,IPropsOne
{
   /// <summary>
   /// склеивание методов двух интерфейсов
   /// </summary>
   /// <param name="s"></param>
```

```
    public void Prop1 (string s)
    {
       Console.WriteLine(s);
    }
    /// <summary>
    /// перегрузка методов двух интерфейсов
    /// </summary>
    /// <param name="s"></param>
    /// <param name="x"></param>
    public void Prop2(string s, int x)
    {
       Console.WriteLine(s + "; " + x);
    }
    public void Prop2 (int x)
    {
       Console.WriteLine(x);
    }
    /// <summary>
    /// переименование методов двух интерфейсов
    /// </summary>
    void IProps.Prop3()
    {
       Console.WriteLine("Свойство 3 интерфейса 1");
    }
    void IPropsOne.Prop3()
    {
       Console.WriteLine("Свойство 3 интерфейса 2");
    }
    public void Prop3FromInterface1()
    {
       ((IProps)this).Prop3();
    }
    public void Prop3FromInterface2()
    {
       ((IPropsOne)this).Prop3();
    }
}
```

Для первого из методов с совпадающей сигнатурой выбрана стратегия *склеивания*, так что в классе есть только один метод, реализующий методы двух *интерфейсов*. Методы с разной сигнатурой реализованы двумя перегруженными методами класса. Для следующей пары методов с совпадающей сигнатурой выбрана стратегия *переименования*. Методы *интерфейсов* реализованы как закрытые методы, а затем в классе объявлены два новых метода с разными именами, являющиеся обертками закрытых методов класса.

Приведу пример работы с объектами класса и интерфейсными объектами:

```
public void TestCliTwoInterfaces()
{
   Console.WriteLine("Объект ClainTwo вызывает методы двух интерфейсов!");
   Cli.ClainTwo claintwo = new Cli.ClainTwo();
   claintwo.Prop1("Склейка свойства двух интерфейсов");
   claintwo.Prop2("перегрузка ::: ",99);
   claintwo.Prop2(9999);
   claintwo.Prop3FromInterface1();
   claintwo.Prop3FromInterface2();
   Console.WriteLine("Интерфейсный объект вызывает методы 1-го
      интерфейса!");
   Cli.IProps ip1 = (Cli.IProps)claintwo;
   ip1.Prop1("интерфейс IProps: свойство 1");
   ip1.Prop2("интерфейс 1 ", 88);
   ip1.Prop3();
```

```
    Console.WriteLine("Интерфейсный объект вызывает методы 2-го
       интерфейса!");
    Cli.IPropsOne ip2 = (Cli.IPropsOne)claintwo;
    ip2.Prop1("интерфейс IPropsOne: свойство1");
    ip2.Prop2(7777);
    ip2.Prop3();
}
```

Результаты работы тестирующей процедуры показаны на рис. 19.2.

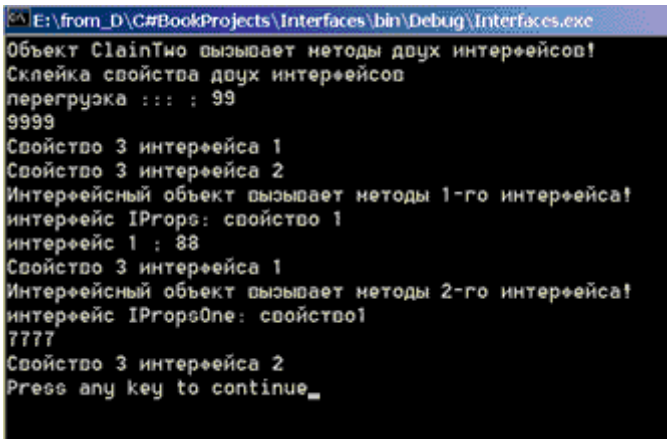

Рис. 19.2. Решение проблемы коллизии имен

Наследование от общего предка

Проблема *наследования от общего предка* характерна, в первую очередь, для *множественного наследования* классов. Если класс `C` является наследником классов `A` и `B`, а те, в свой черед, являются наследниками класса `Parent`, то класс наследует свойства и методы своего предка `Parent` дважды, один раз получая их от класса `A`, другой от - `B`. Это явление называется еще дублирующим наследованием. Для классов ситуация осложняется тем, что классы `A` и `B` могли по-разному переопределить методы родителя и для потомков предстоит сложный выбор реализации.

Для *интерфейсов* сама ситуация дублирующего наследования маловероятна, но возможна, поскольку *интерфейс*, как и любой класс, может быть наследником другого *интерфейса*. Поскольку у *интерфейсов* наследуются только сигнатуры, а не реализации, как в случае классов, то проблема дублирующего наследования сводится к проблеме *коллизии имен*. По-видимому, естественным решением этой проблемы в данной ситуации является *склеивание*, когда методам, пришедшим разными путями от одного родителя, будет соответствовать единая реализация.

Начнем наш пример с наследования *интерфейсов*:

```
public interface IParent
{
   void ParentMethod();
}
public interface ISon1:IParent
{
   void Son1Method();
}
public interface ISon2:IParent
{
   void Son2Method();
}
```

Два сыновних *интерфейса* наследуют метод своего родителя. А теперь рассмотрим класс, наследующий оба *интерфейса*:

```
public class Pars:ISon1, ISon2
{
   public void ParentMethod()
   {
      Console.WriteLine("Это метод родителя!");
   }
   public void Son1Method()
    {
       Console.WriteLine("Это метод старшего сына!");
    }
   public void Son2Method()
   {
      Console.WriteLine("Это метод младшего сына!");
   }
}//class Pars
```

Класс обязан реализовать метод `ParentMethod`, приходящий от обоих *интерфейсов*. Понимая, что речь идет о дублировании метода общего родителя - *интерфейса* `IParent`, лучшей стратегией реализации является *склеивание методов* в одной реализации, что и было сделано. Приведу тестирующую процедуру, где создается объект класса и три объекта интерфейсных классов, каждый из которых может вызывать только методы своего *интерфейса*:

```
public void TestIParsons()
{
   Console.WriteLine("Объект класса вызывает методы трех
      интерфейсов!");
   Cli.Pars ct = new Cli.Pars();
   ct.ParentMethod();
   ct.Son1Method();
   ct.Son2Method();
   Console.WriteLine("Интерфейсный объект 1 вызывает свои
      методы!");
   Cli.IParent ip = (IParent)ct;
   ip.ParentMethod();
   Console.WriteLine("Интерфейсный объект 2 вызывает свои
      методы!");
   Cli.ISon1 ip1 = (ISon1)ct;
   ip1.ParentMethod();
   ip1.Son1Method();
   Console.WriteLine("Интерфейсный объект 3 вызывает свои
      методы!");
   Cli.ISon2 ip2 = (ISon2)ct;
   ip2.ParentMethod();
   ip2.Son2Method();
}
```

Результаты работы тестирующей процедуры показаны на рис. 19.3.

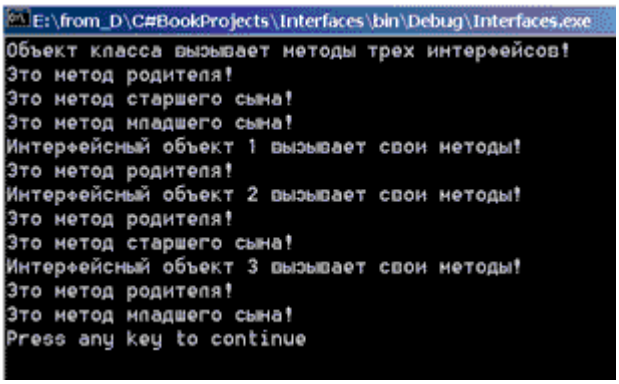

Рис. 19.3. Дублирующее наследование интерфейсов

Встроенные интерфейсы

Рассмотрим несколько *встроенных интерфейсов*, являющихся частью библиотеки FCL. Они используются многими классами-библиотеками так же, как и классами, создаваемыми пользователем.

Упорядоченность объектов и интерфейс IComparable

Часто, когда создается класс, желательно задать *отношение порядка на его объектах*. Такой класс следует объявить наследником *интерфейса* `IComparable`. Этот *интерфейс* имеет всего один метод `CompareTo (object obj)`, возвращающий целочисленное значение, положительное, отрицательное или равное нулю, в зависимости от выполнения отношения "больше", "меньше" или "равно".

Как правило, в классе вначале определяют метод `CompareTo`, а после этого вводят перегруженные операции, чтобы выполнять сравнение объектов привычным образом с использованием знаков операций отношения.

Давайте введем *отношение порядка* на классе `Person`, рассмотренном в лекции 16, сделав этот класс наследником *интерфейса* `IComparable`. Реализуем в этом классе метод *интерфейса* `CompareTo`:

```
public class Person:IComparable
{
   public int CompareTo( object pers)
   {
      const string s = "Сравниваемый объект не принадлежит
         классу Person";
      Person p = pers as Person;
      if (!p.Equals(null))
         return (fam.CompareTo(p.fam));
      throw new ArgumentException (s);
   }
   // другие компоненты класса
}
```

Поскольку аргумент метода должен иметь универсальный тип `object`, то перед выполнением сравнения его нужно привести к типу `Person`. Это *приведение* использует операцию `as`, позволяющую проверить корректность выполнения *приведения*.

При *приведении типов* часто используются операции `is` и `as`. Логическое выражение `(obj is T)` истинно, если объект `obj` имеет тип `T`. Оператор присваивания (`obj = P as T;`) присваивает объекту `obj` объект `P`, приведенный к типу `T`, если такое *приведение*

возможно, иначе объекту присваивается значение `null`. Семантику `as` можно выразить следующим условным выражением: `(P is T) ? (T)P : (T)null`.

Заметьте также, что при проверке на значение `null` используется отношение `Equals`, а не обычное равенство, которое будет переопределено.

Отношение порядка на объектах класса `Person` задается как отношение порядка на фамилиях персон. Так как строки наследуют *интерфейс* `IComparable`, то для фамилий персон вызывается метод `CompareTo`, его результат и возвращается в качестве результата метода `CompareTo` для персон. Если аргумент метода не будет соответствовать нужному типу, то выбрасывается исключение со специальным уведомлением.

Конечно, сравнение персон может выполняться по разным критериям: возрасту, росту, зарплате. Общий подход к сравнению персон будет рассмотрен в следующей лекции 20.

Введем теперь в нашем классе `Person` перегрузку операций отношения:

```
public static bool operator <(Person p1, Person p2)
{
   return (p1.CompareTo(p2) < 0);
}
public static bool operator >(Person p1, Person p2)
{
   return (p1.CompareTo(p2) > 0);
}
public static bool operator <=(Person p1, Person p2)
{
   return (p1.CompareTo(p2) <= 0);
}
public static bool operator >=(Person p1, Person p2)
{
   return (p1.CompareTo(p2) >=0);
}
public static bool operator ==(Person p1, Person p2)
{
   return (p1.CompareTo(p2) == 0);
}
public static bool operator !=(Person p1, Person p2)
{
   return (p1.CompareTo(p2) != 0);
}
```

Как обычно, приведу тестовый пример, проверяющий работу с введенными методами:

```
public void TestCompare()
{
   Person poet1 = new Person("Пушкин");
   Person poet2 = new Person("Лермонтов");
   Person poet3 = new Person("Пастернак");
   Person poet4 = new Person("Мандельштам");
   Person poet5 = new Person("Ахматова");
   Person poet6 = new Person("Цветаева");
   Console.WriteLine("{0} > {1} = {2}", poet1.Fam,
      poet2.Fam, (poet1 > poet2));
   Console.WriteLine("{0} >= {1} = {2}", poet3.Fam,
      poet4.Fam, (poet3 >= poet4));
   Console.WriteLine("{0} != {1} = {2}", poet5.Fam,
      poet6.Fam, (poet5 != poet6));
```

```
}
```

Вот результаты работы этого теста.

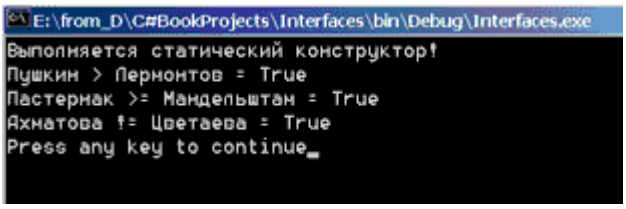

Рис. 19.4. Сравнение персон

Конечно, заданный нами порядок не имеет никакого отношения к поэтическому дару, а лишь говорит об относительном расположении фамилий поэтов в словарях.

Клонирование и интерфейс ICloneable

Клонированием называется процесс создания копии объекта, а копия объекта называется его ***клоном***. Различают два типа *клонирования*: *поверхностное (shallow)* и *глубокое (deep)*. При ***поверхностном клонировании*** копируется сам объект. Все значимые поля *клона* получают значения, совпадающие со значениями полей объекта; все ссылочные поля *клона* являются ссылками на те же объекты, на которые ссылается и сам объект. При ***глубоком клонировании*** копируется вся совокупность объектов, связанных взаимными ссылками. Представьте себе мир объектов, описывающих людей. У этих объектов могут быть ссылки на детей и родителей, учителей и учеников, друзей и родственников. В текущий момент может существовать большое число таких объектов, связанных ссылками. Достаточно выбрать один из них в качестве корня, и при его *клонировании* воссоздастся копия существующей структуры объектов.

Глубокое клонирование требует рекурсивной процедуры обхода существующей структуры объектов, тщательно отработанной во избежание проблемы зацикливания. В общем случае, когда есть несколько классов, являющихся взаимными клиентами, *глубокое клонирование* требует наличия в каждом классе рекурсивной процедуры. Эти процедуры взаимно вызывают друг друга. Я не буду в этой лекции приводить примеры, хотя среди проектов, поддерживающих книгу, есть и проект, реализующий *глубокое клонирование*, где объекты разных классов связаны взаимными ссылками.

Поверхностное клонирование можно выполнить достаточно просто. Наиболее простой путь - *клонирование* путем вызова метода `MemberwiseClone`, наследуемого от прародителя `object`. Единственное, что нужно помнить: этот метод защищен, он не может быть вызван у клиента класса. Поэтому *клонирование* нужно выполнять в исходном классе, используя прием *обертывания* метода.

Давайте обеспечим эту возможность для класса `Person`, создав в нем соответствующий метод:

```
public Person StandartClone()
{
   Person p = (Person)this.MemberwiseClone();
   return(p);
}
```

Теперь клиенты класса могут легко создавать поверхностные *клоны*. Вот пример:

```
public void TestStandartClone()
{
   Person mother = new Person("Петрова Анна");
   Person daughter = new Person("Петрова Ольга");
```

```
    Person son = new Person("Петров Игорь");
    mother[0] = daughter;
    mother[1] = son;
    Person mother_clone = mother.StandartClone();
    Console.WriteLine("Дети матери: {0}",mother.Fam);
    Console.WriteLine (mother[0].Fam);
    Console.WriteLine (mother[1].Fam);
    Console.WriteLine("Дети клона: {0}",mother_clone.Fam);
    Console.WriteLine (mother_clone[0].Fam);
    Console.WriteLine (mother_clone[1].Fam);
}
```

При создании *клона* будет создана копия только одного объекта `mother`. Обратите внимание: при работе с полем `children`, задающим детей, используется индексатор класса `Person`, выполняющий индексацию по этому полю. Вот как выглядят результаты работы теста.

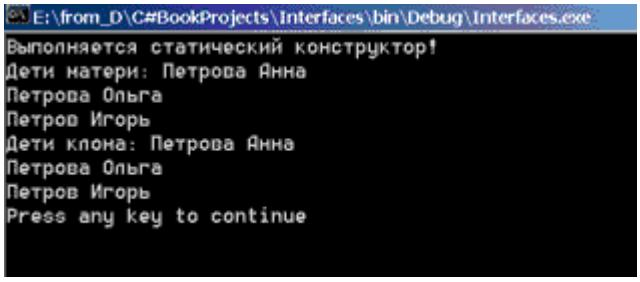

Рис. 19.5. Поверхностное клонирование

Если стандартное *поверхностное клонирование* нас не устраивает, то класс можно объявить наследником *интерфейса* `ICloneable` и реализовать метод `Clone` - единственный метод этого *интерфейса*. В нем можно реализовать полное *глубокое клонирование* или подходящую для данного случая модификацию.

Давайте расширим наш класс, придав ему родительский *интерфейс* `ICloneable`. Реализация метода `Clone` будет отличаться от стандартной реализации тем, что к имени объекта - полю `Fam` - будет приписываться слово "clone". Вот как выглядит этот метод:

```
public object Clone()
{
    Person p = new Person(this.fam + "_clone");
    //копирование полей
    p.age = this.age; p.children = this.children;
    p.count_children = this.count_children;
    p.health = this.health; p.salary = this.salary;
    p.status = this.status;
    return (p);
}
```

Эта реализация является слегка модифицированной версией стандартного *поверхностного клонирования*. Я добавил несколько строчек в тестирующую процедуру для проверки работы этой версии *клона*:

```
    Person mother_clone2 = (Person)mother.Clone();
    Console.WriteLine("Дети клона_2: {0}",mother_clone2.Fam);
    Console.WriteLine (mother_clone2[0].Fam);
    Console.WriteLine (mother_clone2[1].Fam);
```

Все работает должным образом.

Сериализация объектов

При работе с программной системой зачастую возникает необходимость в *сериализации объектов*. Под ***сериализацией*** понимают процесс сохранения объектов в долговременной памяти (файлах) в период выполнения системы. Под *десериализацией* понимают обратный процесс - восстановление состояния объектов, хранимых в долговременной памяти. Механизмы *сериализации* C# и Framework .Net поддерживают два формата сохранения данных - в бинарном файле и XML-файле. В первом случае данные при *сериализации* преобразуются в бинарный поток символов, который при *десериализации* автоматическипреобразуется в нужное состояние объектов. Другой возможный преобразователь (SOAP formatter) запоминает состояние объекта в формате xml.

Сериализация позволяет запомнить рубежные состояния системы объектов с возможностью последующего возвращения к этим состояниям. Она необходима, когда завершение сеанса работы не означает завершение вычислений. В этом случае очередной сеанс работы начинается с восстановления состояния, сохраненного в конце предыдущего сеанса работы. Альтернативой *сериализации* является работа с обычной файловой системой, с базами данных и другими хранилищами данных. Поскольку механизмы *сериализации*, предоставляемые языком C#, эффективно поддерживаются .Net Framework, то при необходимости сохранения данных значительно проще и эффективнее пользоваться *сериализацией*, чем самому организовывать их хранение и восстановление.

Еще одно важное применение *сериализации* - это обмен данными удаленных систем. При удаленном обмене данными предпочтительнее формат xml из-за открытого стандарта передачи данных в Интернете по soap-протоколу, из-за открытого стандарта на структуру xml-документов. Обмен становится достаточно простым даже для систем, построенных на разных платформах и в разных средах разработки.

Так же, как и *клонирование*, *сериализация* может быть поверхностной, когда сериализуется на одном шаге единственный объект, и глубокой, когда, начиная с корневого объекта, сериализуется совокупность объектов, связанных взаимными ссылками (граф объектов). Глубокую *сериализацию*, часто обязательную, самому организовать непросто, так как она требует, как правило, рекурсивного обхода структуры объектов.

Если класс объявить с *атрибутом [Serializable]*, то в него встраивается стандартный механизм *сериализации*, поддерживающий, что крайне приятно, глубокую *сериализацию*. Если по каким-либо причинам стандартная *сериализация* нас не устраивает, то класс следует объявить наследником *интерфейса* `ISerialzable`, реализация методов которого позволит управлять процессом *сериализации*. Мы рассмотрим обе эти возможности

Класс с атрибутом сериализации

Класс, объекты которого предполагается сериализовать стандартным образом, должен при объявлении сопровождаться *атрибутом [Serializable]*. Стандартная *сериализация* предполагает два способа сохранения объекта: в виде бинарного потока символов и в виде xml-документа. В бинарном потоке сохраняются все поля объекта, как открытые, так и закрытые. Процессом этим можно управлять, помечая некоторые поля класса атрибутом `[NonSerialized]` - эти поля сохраняться не будут:

```
[Serializable]
public class Test
{
   public string name;
   [NonSerialazed] int id;
   int age;
   //другие поля и методы класса
}
```

В класс `Test` встроен стандартный механизм *сериализации* его объектов. При *сериализации* поля name и age будут сохраняться, поле `id` - нет.

Для запуска механизма необходимо создать объект, называемый форматером и выполняющий *сериализацию* и *десериализацию* данных с подходящим их форматированием. Библиотека FCL предоставляет два класса форматеров. Бинарный форматер, направляющий данные в бинарный поток, принадлежит классу `BinaryFormatter`. Этот класс находится в пространстве имен библиотеки FCL:

```
System.Runtime.Serialization.Formatters.Binary
```

Давайте разберемся, как устроен этот класс. Он является наследником двух *интерфейсов*: `IFormatter` и `IRemotingFormatter`. *Интерфейс* `IFormatter` имеет два открытых метода: `Serialize` и `Deserialize`, позволяющих сохранять и восстанавливать всю совокупность связанных объектов с заданным объектом в качестве корня. *Интерфейс* `IRemotingFormatter` имеет те же открытые методы: `Serialize` и `Deserialize`, позволяющие выполнять глубокую *сериализацию*, но в режиме удаленного вызова. Поскольку сигнатуры одноименных методов *интерфейсов* отличаются, то конфликта имен при наследовании не происходит - в классе `BinaryFormatter` методы `Serialize` и `Deserialize` перегружены. Для удаленного вызова задается дополнительный параметр, что и позволяет различать, локально или удаленно выполняются процессы обмена данными.

В пространстве имен библиотеки FCL:

```
System.Runtime.Serialization.Formatters.Soap
```

находится класс `SoapFormatter`. Он является наследником тех же *интерфейсов* `IFormatter` и `IRemotingFormatter` и реализует их методы `Serialize` и `Deserialize`, позволяющие выполнять глубокую *сериализацию* и *десериализацию* при сохранении данных в формате xml. Помимо методов класса `SoapFormatter`, xml-сериализацию можно выполнять средствами другого класса -- `XmlSerializer`.

Из новых средств, еще не рассматривавшихся в наших лекциях, для организации *сериализации* понадобятся файлы. Пространство имен `IO` библиотеки FCL предоставляет классы, поддерживающие ввод-вывод данных. В частности, в этом пространстве есть абстрактный класс `Stream` для работы с потоками данных. С одним из его потомков - классом `FileStream` - мы и будем работать в нашем примере.

В качестве примера промоделируем сказку Пушкина "О рыбаке и рыбке". Как вы помните, жадная старуха богатела, богатела, но после очередного желания оказалась у разбитого корыта, вернувшись в начальное состояние. *Сериализация* позволит нам запомнить начальное состояние, меняющееся по мере выполнения рыбкой первых пожеланий рыбака и его старухи. *Десериализация* вернет все в начальное состояние. Опишу класс, задающий героев пушкинской сказки:

```
[Serializable]
public class Personage
{
   public Personage(string name, int age)
   {
      this.name = name; this.age = age;
   }
   //поля класса
   static int wishes;
   public string name, status, wealth;
   int age;
   public Personage couple;
```

```
   //методы класса
}
```

Герои сказки - объекты этого класса обладают свойствами, задающими имя, возраст, статус, имущество и супруга. Имя и возраст задаются в конструкторе класса, а остальные свойства задаются в следующем методе:

```
public void marry (Personage couple)
{
   this.couple = couple;
   couple.couple = this;
   this.status ="крестьянин";
   this.wealth ="рыбацкая сеть";
   this.couple.status = "крестьянка";
   this.couple.wealth = "корыто";
   SaveState();
}
```

Предусловие метода предполагает, что метод вызывается один раз главным героем (рыбаком). В методе устанавливаются взаимные ссылки между героями сказки, их начальное состояние. Завершается метод сохранением состояния объектов, выполняемого при вызове метода `SaveState`:

```
void SaveState()
{
   BinaryFormatter bf = new BinaryFormatter();
   FileStream fs = new FileStream
      ("State.bin",FileMode.Create, FileAccess.Write);
   bf.Serialize(fs,this);
   fs.Close();
}
```

Здесь и выполняется *сериализация* графа объектов. Как видите, все просто. Вначале создается форматер - объект `bf` класса `BinaryFormatter`. Затем определяется файл, в котором будет сохраняться состояние объектов, - объект `fs` класса `FileStream`. Заметьте, в конструкторе файла, кроме имени файла, указываются его характеристики: статус, режим доступа. На деталях введения файлов я останавливаться не буду. Теперь, когда основные объекты определены, остается вызвать метод `Serialize` объекта `bf`, которому в качестве аргументов передается объект `fs` и текущий объект, представляющий корневой объект графа объектов, которые подлежат *сериализации*. Глубокая *сериализация*, реализуемая в данном случае, не потребовала от нас никаких усилий.

Нам понадобится еще метод, описывающий жизнь героев сказки:

```
public Personage AskGoldFish()
{
   Personage fisher = this;
   if (fisher.name == "рыбак")
   {
      wishes++;
      switch (wishes)
      {
         case 1: ChangeStateOne();break;
         case 2: ChangeStateTwo();break;
         case 3: ChangeStateThree();break;
         default: BackState(ref fisher);break;
      }
   }
   return(fisher);
```

```
}//AskGoldFish
```

Метод реализует анализ желаний героини сказки. Первые три желания исполняются, и состояние героев меняется:

```
void ChangeStateOne()
{
   this.status = "муж дворянки";
   this.couple.status = "дворянка";
   this.couple.wealth = "имение";
}
void ChangeStateTwo()
{
   this.status = "муж боярыни";
   this.couple.status = "боярыня";
   this.couple.wealth = "много поместий";
}
void ChangeStateThree()
{
   this.status = "муж государыни";
   this.couple.status = "государыня";
   this.couple.wealth = "страна";
}
```

Начиная с четвертого желания, все возвращается в начальное состояние - выполняется *десериализация* графа объектов:

```
void BackState(ref Personage fisher)
{
   BinaryFormatter bf = new BinaryFormatter();
   FileStream fs = new FileStream
      ("State.bin",FileMode.Open, FileAccess.Read);
   fisher = (Personage)bf.Deserialize(fs);
   fs.Close();
}
```

Обратите внимание, что у метода есть аргумент, передаваемый по ссылке. Этот аргумент получает значение - ссылается на объект, создаваемый методом `Deserialize`. Без аргумента метода не обойтись, поскольку возвращаемый методом объект нельзя присвоить текущему объекту `this`. Важно также отметить, что метод `Deserialize` восстанавливает весь граф объектов, возвращая в качестве результата корень графа.

В классе определен еще один метод, сообщающий о текущем состоянии объектов:

```
public void About()
{
   Console.WriteLine("имя = {0}, возраст = {1},"+
   "статус = {2}, состояние ={3}",name,age,status, wealth);
   Console.WriteLine("имя = {0}, возраст = {1}," +
   "статус = {2}, состояние ={3}",    this.couple.name,
   this.couple.age,this.couple.status, this.couple.wealth);
}
```

Для завершения сказки нам нужно в клиентском классе создать ее героев:

```
public void TestGoldFish()
{
   Personage fisher = new Personage("рыбак", 70);
   Personage wife = new Personage("старуха", 70);
   fisher.marry(wife);
   Console.WriteLine("До золотой рыбки"); fisher.About();
```

```
   fisher = fisher.AskGoldFish();
   Console.WriteLine("Первое желание"); fisher.About();
    fisher = fisher.AskGoldFish();
   Console.WriteLine("Второе желание"); fisher.About();
    fisher = fisher.AskGoldFish();
   Console.WriteLine("Третье желание"); fisher.About();
    fisher = fisher.AskGoldFish();
   Console.WriteLine("Еще хочу");      fisher.About();
   fisher = fisher.AskGoldFish();
   Console.WriteLine("Хочу, но уже поздно");  fisher.About();
}
```

На рис. 19.6 показаны результаты исполнения сказки.

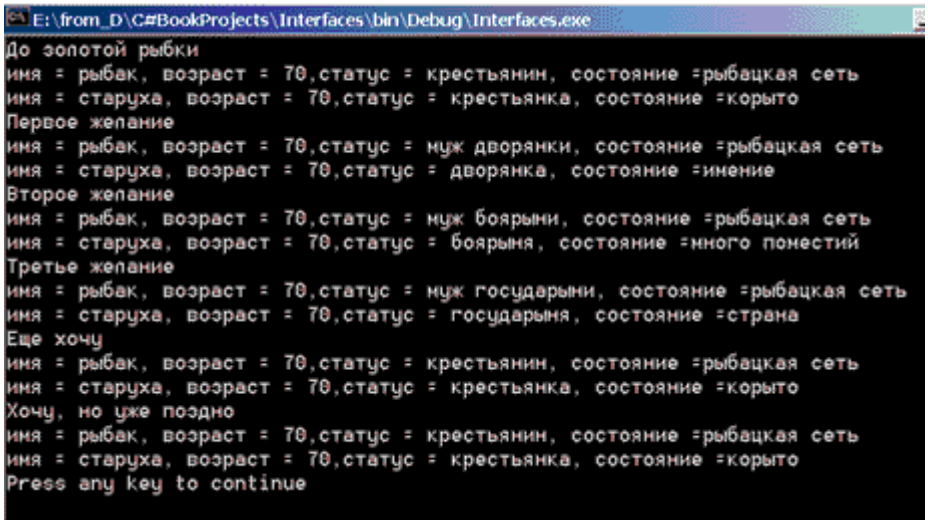

Рис. 19.6. Сказка о рыбаке и рыбке

Что изменится, если перейти к сохранению данных в xml-формате? немногое. Нужно лишь заменить объявление форматера:

```
void SaveStateXML()
{
   SoapFormatter sf = new SoapFormatter();
   FileStream fs = new FileStream
      ("State.xml",FileMode.Create, FileAccess.Write);
   sf.Serialize(fs,this);
   fs.Close();
}
void BackStateXML(ref Personage fisher)
{
   SoapFormatter sf = new SoapFormatter();
   FileStream fs = new FileStream
("State.xml",FileMode.Open, FileAccess.Read);
   fisher = (Personage)sf.Deserialize(fs);
   fs.Close();
}
```

Клиент, работающий с объектами класса, этих изменений и не почувствует. Результаты вычислений останутся теми же, что и в предыдущем случае. Правда, файл, сохраняющий данные, теперь выглядит совсем по-другому. Это обычный xml-документ, который мог быть создан в любом из приложений. Вот как выглядит этот документ, открытый в браузере Internet Explorer.

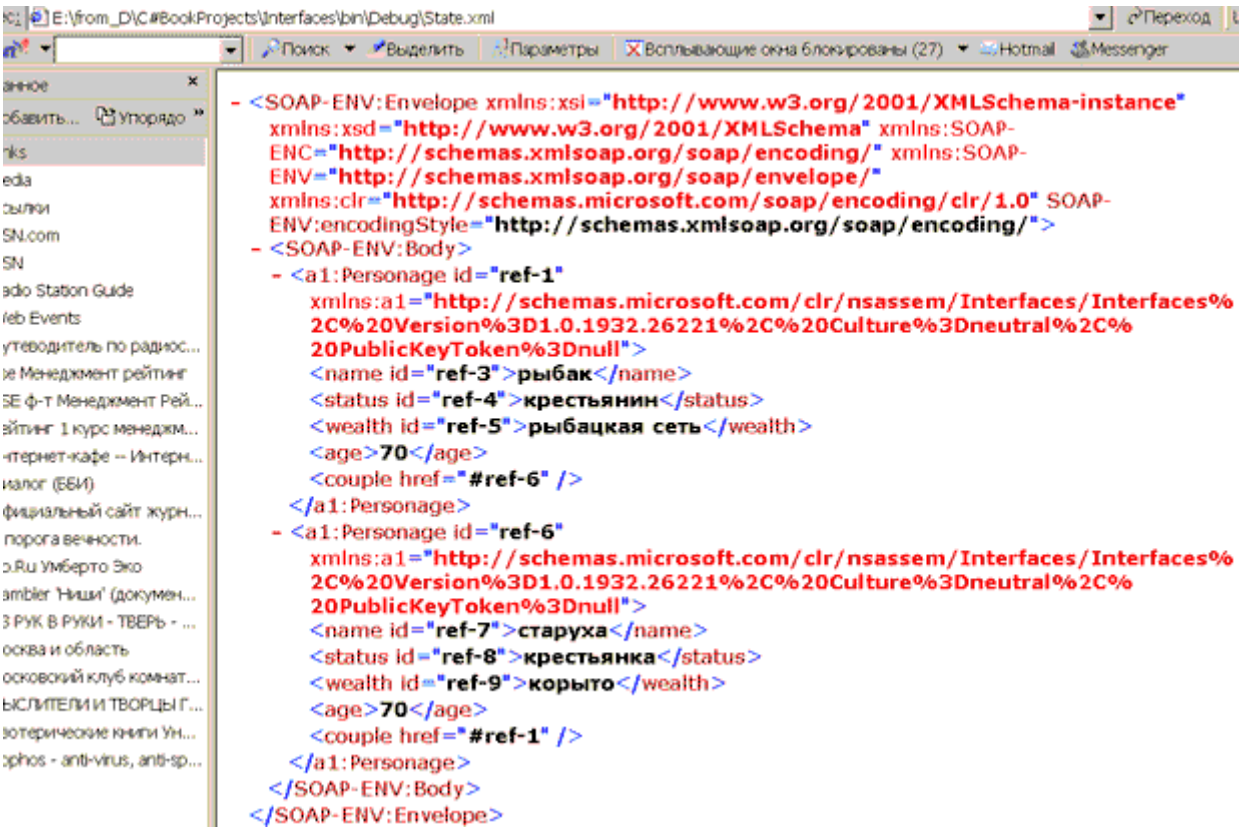

Рис. 19.7. XML-документ, сохраняющий состояние объектов

Интерфейс ISerializable

При необходимости можно самому управлять процессом *сериализации*. В этом случае наш класс должен быть наследником *интерфейса* `ISerializable`. Класс, наследующий этот *интерфейс*, должен реализовать единственный метод этого *интерфейса* `GetObjectData` и добавить защищенный конструктор. Схема *сериализации* и *десериализации* остается и в этом случае той же самой. Можно использовать как бинарный форматер, так и soap-форматер. Но теперь метод `Serialize` использует не стандартную реализацию, а вызывает метод `GetObjectData`, управляющий записью данных. Метод `Deserialize`, в свою очередь, вызывает защищенный конструктор, создающий объект и заполняющий его поля сохраненными значениями.

Конечно, возможность управлять сохранением и восстановлением данных дает большую гибкость и позволяет, в конечном счете, уменьшить размер файла, хранящего данные, что может быть крайне важно, особенно если речь идет об обмене данными с удаленным приложением. Если речь идет о поверхностной *сериализации*, то атрибут `NonSerialized`, которым можно помечать поля, не требующие *сериализации*, как правило, достаточен для управления эффективным сохранением данных. Так что управлять имеет смысл только глубокой *сериализацией*, когда сохраняется и восстанавливается граф объектов. Но, как уже говорилось, это может быть довольно сложным занятием, что будет видно и для нашего простого примера с рыбаком и рыбкой.

Рассмотрим, как устроен метод GetObjectData, управляющий сохранением данных. У этого метода два аргумента:

```
GetObjectData(SerializedInfo info, StreamingContext context)
```

Поскольку самому вызывать этот метод не приходится - он вызывается автоматически методом `Serialize`, то можно не особенно задумываться о том, как создавать аргументы метода. Более важно понимать, как их следует использовать. Чаще всего

используется только аргумент `info` и его метод `AddValue (key, field)`. Данные сохраняются вместе с ключом, используемым позже при чтении данных. Аргумент `key`, который может быть произвольной строкой, задает ключ, а аргумент `field` - поле объекта. Например, для сохранения полей `name` и `age` можно задать следующие операторы:

```
info.AddValue("name",name); info.AddValue("age", age);
```

Поскольку имена полей уникальны, то их разумно использовать в качестве ключей.

Если поле `son` класса `Father` является объектом класса `Child` и этот класс сериализуем, то для сохранения объекта `son` следует вызвать метод:

```
son.GetObjectData(info, context)
```

Если не возникает циклов, причиной которых являются взаимные ссылки, то особых сложностей с *сериализацией* и *десериализацией* не возникает. Взаимные ссылки осложняют картину и требуют индивидуального подхода к решению. На последующем примере мы покажем, как можно справиться с этой проблемой в конкретном случае.

Перейдем теперь к рассмотрению специального конструктора класса. Он может быть объявлен с атрибутом доступа `private`, но лучше, как и во многих других случаях, применять атрибут `protected`, что позволит использовать этот конструктор потомками класса, осуществляющими собственную *сериализацию*. У конструктора те же аргументы, что и у метода `GetObjectData`. Опять-таки, в основном используется аргумент `info` и его метод `GetValue(key, type)`, который выполняет операцию, обратную к операции метода `AddValue`. По ключу `key` находится хранимое значение, а аргумент `type` позволяет привести его к нужному типу. У метода `GetValue` имеется множество типизированных версий, позволяющих не задавать тип. Так что восстановление полей `name` и `age` можно выполнить следующими операторами:

```
name = info.GetString("name"); age = info.GetInt32("age");
```

Восстановление поля `son`, являющегося ссылочным типом, выполняется вызовом его специального конструктора:

```
son = new Child(info, context);
```

А теперь вернемся к нашему примеру со стариком, старухой и золотой рыбкой. Заменим стандартную *сериализацию* собственной. Для этого, оставив атрибут *сериализации* у класса `Personage`, сделаем класс наследником *интерфейса* `ISerializable`:

```
[Serializable]
   public class Personage :ISerializable
   {...}
```

Добавим в наш класс специальный метод, вызываемый при *сериализации* - метод сохранения данных:

```
//Специальный метод сериализации
public void GetObjectData(SerializationInfo info,
   StreamingContext context)
      {
         info.AddValue("name",name); info.AddValue("age", age);
         info.AddValue("status",status);
         info.AddValue("wealth", wealth);
         info.AddValue("couplename",couple.name);
```

```
            info.AddValue("coupleage", couple.age);
            info.AddValue("couplestatus",couple.status);
            info.AddValue("couplewealth", couple.wealth);
        }
```

В трех первых строках сохраняются значимые поля объекта и тут все ясно. Но вот запомнить поле, хранящее объект `couple` класса `Personage`, напрямую не удается. Попытка рекурсивного вызова

```
couple.GetObjectData(info,context);
```

привела бы к зацикливанию, если бы раньше из-за повторяющегося ключа не возникала исключительная ситуация в момент записи поля name объекта `couple`. Поэтому приходится явно сохранять поля этого объекта уже с другими ключами. Понятно, что с ростом сложности структуры графа объектов задача существенно осложняется.

Добавим в наш класс специальный конструктор, вызываемый при *десериализации* - конструктор восстановления состояния:

```
//Специальный конструктор сериализации
protected Personage(SerializationInfo info,
        StreamingContext context)
    {
        name = info.GetString("name"); age = info.GetInt32("age");
        status = info.GetString("status");
        wealth = info.GetString("wealth");
        couple = new Personage(info.GetString("couplename"),
                  info.GetInt32("coupleage"));
        couple.status = info.GetString("couplestatus");
        couple.wealth = info.GetString("couplewealth");
        this.couple = couple;    couple.couple = this;
    }
```

Опять первые строки восстановления значимых полей объекта прозрачно ясны. А с полем `couple` приходится повозиться. Вначале создается новый объект обычным конструктором, аргументы которого читаются из сохраняемой памяти. Затем восстанавливаются значения других полей этого объекта, а затем уже происходит взаимное связывание двух объектов.

Кроме введения конструктора класса и метода `GetObjectData`, никаких других изменений в проекте не понадобилось - ни в методах класса, ни на стороне клиента. Внешне проект работал совершенно идентично ситуации, когда не вводилось наследование *интерфейса сериализации*. Но с внутренних позиций изменения произошли: методы форматеров `Serialize` и `Deserialize` в процессе своей работы теперь вызывали созданный нами метод и конструктор класса. Небольшие изменения произошли и в файлах, хранящих данные.

Мораль: должны быть веские основания для отказа от стандартно реализованной *сериализации*. Повторюсь, такими основаниями можгут служить необходимость в уменьшении объема файла, хранящего данные, и в сокращении времени передачи данных.

Когда в нашем примере вводилось собственное управление *сериализацией*, то не ставилась цель минимизации объема хранимых данных, в обоих случаях сохранялись одни и те же данные. Тем не менее представляет интерес взглянуть на таблицу, хранящую объемы создаваемых файлов.

Таблица 19.1. Размеры файлов при различных случаях сериализации		
Формат	**Сериализация**	**Размер файла**
Бинарный поток	Стандартная	355 байтов
Бинарный поток	Управляемая	355 байтов
XML-документ	Стандартная	1, 14 Кб.
XML-документ	Управляемая	974 байта

Преимуществами XML-документа являются его читабельность и хорошо развитые средства разбора, но зато бинарное представление выигрывает в объеме и скорости передачи тех же данных.

20. Лекция: Функциональный тип в C#. Делегаты

Как определяется функциональный тип и как появляются его экземпляры

Слово *делегат (delegate)* используется в C# для обозначения хорошо известного понятия. ***Делегат*** задает определение функционального типа (класса) данных. Экземплярами класса являются функции. Описание *делегата* в языке C# представляет собой описание еще одного частного случая класса. Каждый *делегат* описывает множество функций с заданной сигнатурой. Каждая функция (метод), сигнатура которого совпадает с сигнатурой *делегата*, может рассматриваться как экземпляр класса, заданного *делегатом*. Синтаксис объявления *делегата* имеет следующий вид:

```
[<спецификатор доступа>] delegate <тип результата > <имя класса> (<список аргументов>);
```

Этим объявлением класса задается функциональный тип - множество функций с заданной сигнатурой, у которых аргументы определяются списком, заданным в объявлении *делегата*, и тип возвращаемого значения определяется типом результата *делегата*.

Спецификатор доступа может быть, как обычно, опущен. Где следует размещать объявление *делегата*? Как и у всякого класса, есть две возможности:

- непосредственно в пространстве имен, наряду с объявлениями других классов, структур, интерфейсов;
- внутри другого класса, наряду с объявлениями методов и свойств. Такое объявление рассматривается как объявление вложенного класса.

Так же, как и интерфейсы C#, *делегаты* не задают реализации. Фактически между некоторыми классами и *делегатом* заключается *контракт на реализацию делегата*. Классы, согласные с *контрактом*, должны объявить у себя статические или динамические функции, сигнатура которых совпадает с сигнатурой *делегата*. Если *контракт* выполняется, то можно создать *экземпляры делегата*, присвоив им в качестве значений функции, удовлетворяющие *контракту*. Заметьте, *контракт* является жестким: не допускается ситуация, при которой у *делегата* тип параметра - `object`, а у *экземпляра* соответствующий параметр имеет тип, согласованный с `object`, например, `int`.

Начнем примеры этой лекции с объявления трех *делегатов*. Поместив два из них в пространство имен, третий вложим непосредственно в создаваемый нами класс:

```
namespace Delegates
{
//объявление классов - делегатов
delegate void Proc(ref int x);
delegate void MesToPers(string s);
class OwnDel
{
   public delegate int Fun1(int x);
   int Plus1( int x){return(x+100);}//Plus1
   int Minus1(int x){return(x-100);}//Minus1
   void Plus(ref int x){x+= 100;}
   void Minus(ref int x){x-=100;}
   //поля класса
   public Proc p1;
   public Fun1 f1;
   char sign;
```

```
   //конструктор
   public OwnDel(char sign)
   {
      this.sign = sign;
      if (sign == '+')
      {p1 = new Proc(Plus);f1 = new Fun1(Plus1);}
      else
      {p1 = new Proc(Minus);f1 = new Fun1(Minus1);}
   }
}//class OwnDel
```

Прокомментирую этот текст.

- Первым делом объявлены три функциональных класса - три *делегата*: `Proc`, `MesToPers`, `Fun1`. Каждый из них описывает множество функций фиксированной сигнатуры.
- В классе `OwnDel` описаны четыре метода: `Plus`, `Minus`, `Plus1`, `Minus1`, сигнатуры которых соответствуют сигнатурам, задаваемых классами `Proc` и `Fun1`.
- Поля `p1` и `f1` класса `OwnDel` являются экземплярами классов `Proc` и `Fun1`.
- В конструкторе класса поля `p1` и `f1` связываются с конкретными методами `Plus` или `Minus`, `Plus1` или `Minus1`. Связывание с той или иной функцией в данном случае определяется значением поля `sign`.

Заметьте, *экземпляры делегатов* можно рассматривать как ссылки (указатели на функции), а методы тех или иных классов с соответствующей сигнатурой можно рассматривать как объекты, хранимые в динамической памяти. В определенный момент происходит связывание ссылки и объекта (в этой роли выступают не обычные объекты, имеющие поля, а методы, задающие код). Взгляд на *делегата* как на указатель функции характерен для программистов, привыкших к C++.

Приведу теперь процедуру, тестирующую работу созданного класса:

```
public void TestOwnDel()
{
   int account = 1000, account1=0;
   OwnDel oda = new OwnDel('+');
   Console.WriteLine("account = {0}, account1 = {1}",
      account, account1);
   oda.p1(ref account); account1=oda.f1(account);
   Console.WriteLine("account = {0}, account1 = {1}",
      account, account1);
}
```

Клиент класса `OwnDel` создает экземпляр класса, передавая конструктору знак той операции, которую он хотел бы выполнить над своими счетами - `account` и `account1`. Вызов `p1` и `f1`, связанных к моменту вызова с закрытыми методами класса, приводит к выполнению нужных функций.

В нашем примере объявление *экземпляров делегатов* и связывание их с внутренними методами класса происходило в самом классе. Клиенту оставалось лишь вызывать уже созданные экземпляры, но эту работу можно выполнять и на стороне клиентского класса, чем мы сейчас и займемся. Рассмотрим многократно встречавшийся класс `Person`, слегка изменив его определение:

```
class Person
{
   //конструкторы
   public Person(){name =""; id=0; salary=0.0;}
   public Person(string name){this.name = name;}
```

```
   public Person (string name, int id, double salary)
   {this.name = name; this.id=id; this.salary = salary;}
   public Person (Person pers)
   {this.name = pers.name; this.id = pers.id;
         this.salary = pers.salary;}
   //методы
   public void ToPerson(string mes)
   {
      this.message = mes;
      Console.WriteLine("{0}, {1}",name, message);
   }
   //свойства
   private string name;
   private int id;
   private double salary;
   private string message;
   //доступ к свойствам
   public string Name
   {get {return(name);}        set {name = value;}}
   public double Salary
   {get {return(salary);}      set {salary = value;}}
   public int Id
   {get {return(id);}          set {id = value;}}
}//class Person
```

Класс `Person` устроен обычным способом: у него несколько перегруженных конструкторов, закрытые поля и процедуры-свойства для доступа к ним. Особо обратить внимание прошу на метод класса `ToPerson`, сигнатура которого совпадает с сигнатурой класса, определенной введенным ранее *делегатом* `MesToPers`. Посмотрите, как клиент класса может связать этот метод с *экземпляром делегата*, определенного самим клиентом:

```
Person man1 = new Person("Владимир");
         MesToPers mestopers = new MesToPers(man1.ToPerson);
         mestopers("пора работать!");
```

Обратите внимание, что поскольку метод `ToPerson` не является статическим методом, то при связывании необходимо передать и объект, вызывающий метод. Более того, переданный объект становится доступным *экземпляру делегата*. Отсюда сразу же становится ясным, что *экземпляры делегата* - это не просто указатели на функцию, а более сложно организованные структуры. Они, по крайней мере, содержат пару указателей на метод и на объект, вызвавший метод. Вызываемый метод в своей работе использует как информацию, передаваемую ему через аргументы метода, так и информацию, хранящуюся в полях объекта. В данном примере переданное сообщение "пора работать" присоединится к имени объекта, и результирующая строка будет выдана на печать. В тех случаях, когда метод, связываемый с *экземпляром делегата*, не использует информацию объекта, этот метод может быть объявлен как статический метод класса. Таким образом, *инициализировать экземпляры делегата* можно как статическими, так и динамическими методами, связанными с конкретными объектами.

Последние три строки были добавлены в вышеприведенную тестирующую процедуру. Взгляните на результаты ее работы.

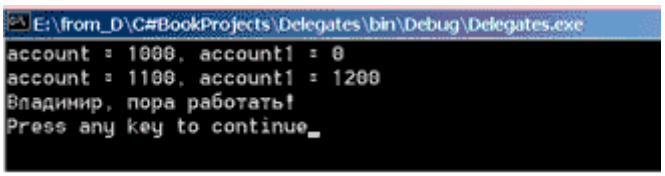

Рис. 20.1. Объявление делегатов и создание их экземпляров

Функции высших порядков

Одно из наиболее важных применений *делегатов* связано с *функциями высших порядков*. ***Функцией высшего порядка*** называется такая функция (метод) класса, у которой один или несколько аргументов принадлежат к функциональному типу. Без этих функций в программировании обойтись довольно трудно. Классическим примером является функция вычисления интеграла, у которой один из аргументов задает подынтегральную функцию. Другим примером может служить функция, сортирующая объекты. Аргументом ее является функция `Compare`, сравнивающая два объекта. В зависимости от того, какая функция сравнения будет передана на вход функции сортировки, объекты будут сортироваться по-разному, например, по имени, или по ключу, или по нескольким полям. Вариантов может быть много, и они определяются классом, описывающим сортируемые объекты.

Вычисление интеграла

Давайте более подробно рассмотрим ситуацию с *функциями высшего порядка* на примере задачи вычисления определенного интеграла с заданной точностью. С этой целью создадим класс, в котором будет описан *делегат*, определяющий *контракт*, коему должны удовлетворять подынтегральные функции. В этом же классе определим метод, вычисляющий интеграл. По сути самой задачи этот метод представляет собой функцию высшего порядка. Приведу программный код, описывающий класс:

```
public class HighOrderIntegral
{
   //delegate
   public delegate double SubIntegralFun(double x);
   public double EvalIntegral(double a, double b,
      double eps,SubIntegralFun sif)
   {
      int n=4;
      double I0=0, I1 = I( a, b, n,sif);
      for( n=8; n < Math.Pow(2.0,15.0); n*=2)
      {
         I0 =I1; I1=I(a,b,n,sif);
         if(Math.Abs(I1-I0)<eps) break;
      }
      if(Math.Abs(I1-I0)< eps)
         Console.WriteLine("Требуемая точность достигнута! "+
            " eps = {0}, достигнутая точность ={1}, n= {2}",
            eps,Math.Abs(I1-I0),n);
      else
         Console.WriteLine("Требуемая точность не достигнута! "+
            " eps = {0}, достигнутая точность ={1}, n= {2}",
            eps,Math.Abs(I1-I0),n);
      return(I1);
   }
   private double I(double a, double b, int n,
      SubIntegralFun sif)
   {
      //Вычисляет частную сумму по методу трапеций
      double x = a, sum = sif(x)/2, dx = (b-a)/n;
      for (int i= 2; i <= n; i++)
      {
         x += dx;       sum += sif(x);
      }
      x = b; sum += sif(x)/2;
      return(sum*dx);
   }
}//class HighOrderIntegral
```

Прокомментирую этот текст:

- Класс `HighOrderIntegral` предназначен для работы с функциями. В него вложено описание функционального класса - *делегата* `SubIntegralFun`, задающего класс функций с одним аргументом типа `double` и возвращающих значение этого же типа.
- Метод `EvalIntegral` - основной метод класса позволяет вычислять определенный интеграл. Этот метод есть *функция высшего порядка*, поскольку одним из его аргументов является подынтегральная функция, принадлежащая классу `SubIntegralFun`.
- Для вычисления интеграла применяется классическая схема. Интервал интегрирования разбивается на n частей, и вычисляется частичная сумма по методу трапеций, представляющая приближенное значение интеграла. Затем n удваивается, и вычисляется новая сумма. Если разность двух приближений по модулю меньше заданной точности `eps`, то вычисление интеграла заканчивается, иначе процесс повторяется в цикле. Цикл завершается либо по достижении заданной точности, либо когда `n` достигнет некоторого предельного значения (в нашем случае - 215).
- Вычисление частичной суммы интеграла по методу трапеций реализовано закрытой процедурой `I`.
- Впоследствии класс может быть расширен, и помимо вычисления интеграла он может вычислять и другие характеристики функций.

Чтобы продемонстрировать работу с классом `HighOrderIntegral`, приведу еще класс `Functions`, где описано несколько функций, удовлетворяющих *контракту*, который задан классом `SubIntegralFun`:

```
class functions
{
   //подынтегральные функции
   static public double sif1(double x)
   {
      int k = 1; int b = 2;
      return (double)(k*x +b);
   }
   static public double sif2(double x)
   {
      double a = 1.0; double b = 2.0; double c= 3.0;
      return (double)(a*x*x +b*x +c);
   }
}//class functions
```

А теперь рассмотрим метод класса клиента, выполняющий создание нужных объектов и тестирующий их работу:

```
public  void TestEvalIntegrals()
{
   double myint1=0.0;
   HighOrderIntegral.SubIntegralFun hoisif1 =
      new HighOrderIntegral.SubIntegralFun(functions.sif1);
   HighOrderIntegral hoi = new HighOrderIntegral();
   myint1 = hoi.EvalIntegral(2,3,0.1e-5,hoisif1);
   Console.WriteLine("myintegral1 = {0}",myint1);
   HighOrderIntegral.SubIntegralFun hoisif2 =
      new HighOrderIntegral.SubIntegralFun(functions.sif2);
   myint1= hoi.EvalIntegral(2,3,0.1e-5,hoisif2);
   Console.WriteLine("myintegral2 = {0}",myint1);
}//EvalIntegrals
```

Здесь создаются два *экземпляра делегата* и объект класса `HighOrderIntegral`, вызывающий метод вычисления интеграла. Результаты работы показаны на 20.2.

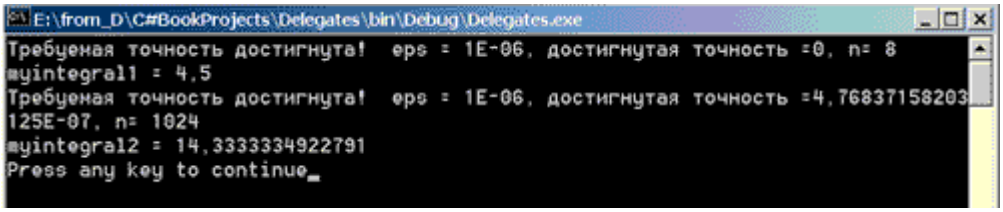

Рис. 20.2. Вычисление интеграла с использованием функций высших порядков

Построение программных систем методом "раскрутки". Функции обратного вызова

Метод "раскрутки" является одним из основных методов функционально-ориентированного построения сложных программных систем. Суть его состоит в том, что программная система создается *слоями*. Вначале пишется ядро системы - нулевой *слой*, реализующий базовый набор функций. Затем пишется первый *слой* с новыми функциями, которые интенсивно вызывают в процессе своей работы функции ядра. Теперь система обладает большим набором функций. Каждый новый *слой* расширяет функциональность системы. Процесс продолжается, пока не будет достигнута заданная функциональность. На рис.20.3, изображающем схему построения системы *методом раскрутки*, стрелками показано, как функции внешних *слоев* вызывают функции внутренних *слоев*.

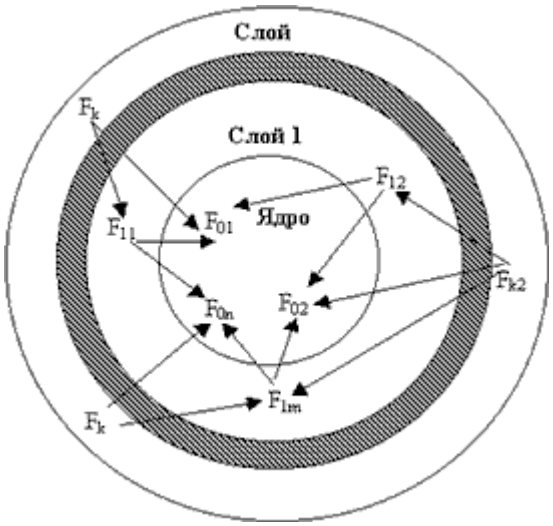

Рис. 20.3. Построение системы методом "раскрутки"

Успех языка С и операционной системы Unix во многом объясняется тем, что в свое время они были созданы *методом раскрутки*. Это позволило написать на 95% на языке С транслятор с языка С и операционную систему. Благодаря этому, обеспечивался легкий перенос транслятора и операционной системы на компьютеры с разной системой команд. Замечу, что в те времена мир компьютеров отличался куда большим разнообразием, чем в нынешнее время. Для переноса системы на новый тип компьютера достаточно было написать ядро системы в соответствии с машинным кодом данного компьютера, далее работала раскрутка.

При построении систем *методом раскрутки* возникает одна проблема. Понятно, что функциям внешнего *слоя* известно все о внутренних *слоях* и они без труда могут вызывать функции внутренних *слоев*. Но как быть, если функциям внутреннего *слоя* необходимо вызывать функции внешних, еще не написанных и, возможно, еще не спроектированных *слоев*? Возможна ли симметрия вызовов? На первый взгляд, это кажется невозможным. Но программисты придумали, по крайней мере, два способа

этой проблемы. Оба они используют *контракты*. Один основан на *функциях обратного вызова*, другой - на *наследовании и полиморфизме*. Мы разберем оба способа, но начнем с *функций обратного вызова*.

Пусть `F` - *функция высшего порядка* с параметром `G` функционального типа. Тогда функцию `G`, задающую параметр (а иногда и саму функцию `F`), называют ***функцией обратного вызова (callback функцией)***. Термин вполне понятен. Если в некотором внешнем *слое* функция `Q` вызывает функцию внутреннего *слоя* `F`, то предварительно во внешнем *слое* следует позаботиться о создании функции `G`, которая и будет передана `F`. Таким образом, функция `Q` внешнего *слоя* вызывает функцию `F` внутреннего *слоя*, которая, в свою очередь (обратный вызов) вызывает функцию `G` внешнего *слоя*. Чтобы эта техника работала, должен быть задан *контракт*. *Функция высших порядков*, написанная во внутреннем *слое*, задает следующий *контракт*: "всякая функция, которая собирается меня вызвать, должна передать мне *функцию обратного вызова*, принадлежащую определенному мной функциональному классу, следовательно, иметь известную мне сигнатуру".

Наш пример с вычислением интеграла хорошо демонстрирует *функции обратного вызова* и технику "раскрутки". Можно считать, что класс `HighOrderIntegral` - это внутренний *слой* нашей системы. В нем задан *делегат*, определяющий *контракт*, и функция `EvalIntegral`, требующая задания *функции обратного вызова* в качестве ее параметра. Функция `EvalIntegral` вызывается из внешнего *слоя*, где и определяются `callback` функции из класса `Functions`.

Многие из функций операционной системы Windows, входящие в состав `Win API 32`, требуют при своем вызове задания `callback`-**функций**. Примером может служить работа с объектом операционной системы `Timer`. Конструктор этого объекта является *функцией высшего порядка*, и ей в момент создания объекта необходимо в качестве параметра передать `callback`-**функцию**, вызываемую для обработки событий, которые поступают от таймера.

Пример работы с таймером приводить сейчас не буду, ограничусь лишь сообщением синтаксиса объявления конструктора объекта `Timer`:

```
public Timer(TimerCallback callback,object state, int dueTime, int period);
```

Первым параметром конструктора является *функция обратного вызова* `callback`, которая принадлежит функциональному классу `TimerCallback`, заданному *делегатом*:

```
public delegate void TimerCallback(object state);
```

Наследование и полиморфизм - альтернатива обратному вызову

Сегодня многие программные системы проектируются и разрабатываются не в функциональном, а в объектно-ориентированном стиле. Такая система представляет собой одно или несколько семейств интерфейсов и классов, связанных отношением наследования. Классы-потомки наследуют методы своих родителей, могут их переопределять и добавлять новые методы. Переопределив метод родителя, потомки без труда могут вызывать как собственный метод, так и метод родителя; все незакрытые методы родителя им известны и доступны. Но может ли родитель вызывать методы, определенные потомком, учитывая, что в момент создания родительского метода потомок не только не создан, но еще, скоре всего, и не спроектирован? Тем не менее, ответ на этот вопрос положителен. Достигается такая возможность опять-таки благодаря *контрактам*, заключаемым при реализации полиморфизма.

О полиморфизме говорилось достаточно много в предыдущих лекциях. Тем не менее, позволю напомнить суть дела. Родитель может объявить свой метод виртуальным, в

этом случае в *контракте* на метод потомку разрешается переопределить реализацию, но он не имеет права изменять сигнатуру виртуального метода. Когда некоторый метод родителя `Q` вызывает виртуальный метод `F`, то, благодаря позднему связыванию, реализуется полиморфизм и реально будет вызван не метод родителя `F`, а метод `F`, который реализован потомком, вызвавшим родительский метод `Q`. Ситуация в точности напоминает раскрутку и вызов обратных функций. Родительский метод `Q` находится во внутреннем *слое*, а потомок с его методом F определен во внешнем *слое*. Когда потомок вызывает метод Q из внутреннего *слоя*, тот, в свою очередь, вызывает метод `F` из внешнего *слоя*. Сигнатура вызываемого метода `F` в данном случае задается не *делегатом*, а сигнатурой виртуального метода, которую, согласно *контракту*, потомок не может изменить. Давайте вернемся к задаче вычисления интеграла и создадим реализацию, основанную на *наследовании и полиморфизме*.

Идея примера такова. Вначале построим родительский класс, метод которого будет вычислять интеграл от некоторой подынтегральной функции, заданной виртуальным методом класса. Далее построим класс-потомок, наследующий родительский метод вычисления интеграла и переопределяющий виртуальный метод, в котором потомок задаст собственную подынтегральную функцию. При такой технологии, всякий раз, когда нужно вычислить интеграл, нужно создать класс-потомок, в котором переопределяется виртуальный метод. Приведу пример кода, следующего этой схеме:

```
class FIntegral
{
   //базовый класс, в котором определен метод вычисления
   //интеграла и виртуальный метод, задющий базовую
   //подынтегральную функцию
   public double EvaluateIntegral(double a, double b,
      double eps)
   {
      int n=4;
      double I0=0, I1 = I( a, b, n);
      for( n=8; n < Math.Pow(2.0,15.0); n*=2)
      {
         I0 =I1; I1=I(a,b,n);
         if(Math.Abs(I1-I0)<eps)break;
      }
      if(Math.Abs(I1-I0)< eps)
         Console.WriteLine("Требуемая точность достигнута! "+
            " eps = {0}, достигнутая точность ={1}, n= {2}",
            eps,Math.Abs(I1-I0),n);
      else
         Console.WriteLine("Требуемая точность не достигнута! "+
            " eps = {0}, достигнутая точность ={1}, n= {2}",
            eps,Math.Abs(I1-I0),n);
      return(I1);
   }
   private double I(double a, double b, int n)
   {
      //Вычисляет частную сумму по методу трапеций
      double x = a, sum = sif(x)/2, dx = (b-a)/n;
      for (int i= 2; i <= n; i++)
      {
         x += dx; sum += sif(x);
      }
      x = b; sum += sif(x)/2;
      return(sum*dx);
   }
   protected virtual double sif(double x)
   {return(1.0);}

}//FIntegral
```

Этот код большей частью знаком. В отличие от класса `HighOrderIntegral`, здесь нет *делегата*, у функции `EvaluateIntegral` нет параметра функционального типа. Вместо этого тут же в классе определен защищенный виртуальный метод, задающий конкретную подынтегральную функцию. В качестве таковой выбрана самая простая функция, тождественно равная единице.

Для вычисления интеграла от реальной функции единственное, что теперь нужно сделать - это задать класс-потомок, переопределяющий виртуальный метод. Вот пример такого класса:

```
class FIntegralSon:FIntegral
{
   protected override double sif(double x)
   {
      double a = 1.0; double b = 2.0; double c= 3.0;
      return (double)(a*x*x +b*x +c);
   }
}//FIntegralSon
```

Принципиально задача решена. Осталось только написать фрагмент кода, запускающий вычисления. Он оформлен в виде следующей процедуры:

```
public void TestPolymorphIntegral()
{
   FIntegral integral1 = new FIntegral();
   FIntegralSon integral2 = new FIntegralSon();
   double res1 = integral1.EvaluateIntegral(2.0,3.0,0.1e-5);
   double res2 = integral2.EvaluateIntegral(2.0,3.0,0.1e-5);
   Console.WriteLine("Father = {0}, Son = {1}", res1,res2);
}//PolymorphIntegral
```

Взгляните на результаты вычислений.

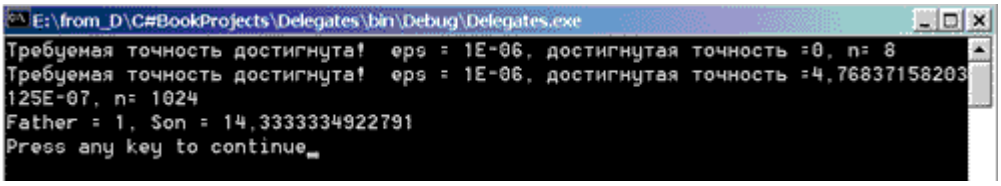

Рис. 20.4. Вычисление интеграла, использующее полиморфизм

Делегаты как свойства

В наших примерах рассматривалась ситуация, при которой в некотором классе объявлялись функции, удовлетворяющие *контракту* с *делегатом*, но создание *экземпляров делегата* и их инициирование функциями класса выполнялось в другом месте, там, где предполагалось вызывать соответствующие функции. Чаще всего, создание экземпляров удобнее возложить на класс, создающий требуемые функции. Более того, в этом классе *делегат* можно объявить *как свойство* класса, что позволяет "убить двух зайцев". Во-первых, с пользователей класса снимается забота создания *делегатов*, что требует некоторой квалификации, которой у пользователя может и не быть. Во-вторых, *делегаты* создаются динамически, в тот момент, когда они требуются. Это важно как при работе с *функциями высших порядков*, когда реализаций, например, подынтегральных функций, достаточно много, так и при работе с событиями класса, в основе которых лежат *делегаты*.

Рассмотрим пример, демонстрирующий и поясняющий эту возможность при работе с *функциями высших порядков*. Идея примера такова. Спроектируем два класса:

- класс объектов `Person` с полями: имя, идентификационный номер, зарплата. В этом классе определим различные реализации функции `Compare`, позволяющие сравнивать два объекта по имени, по номеру, по зарплате, по нескольким полям. Самое интересное, ради чего и строится данный пример: для каждой реализации `Compare` будет построена процедура-свойство, которая задает реализацию *делегата*, определенного в классе `Persons`;
- класс `Persons` будет играть роль *контейнера объектов* `Person`.

В этом классе будут определены операции над объектами. Среди операций нас, прежде всего, будет интересовать сортировка объектов, реализованная в виде *функции высших порядков*. Функциональный параметр будет задавать класс функций сравнения объектов, реализации которых находятся в классе `Person`. *Делегат*, определяющий класс функций сравнения, будет задан в классе `Persons`.

Теперь, когда задача ясна, приступим к ее реализации. Класс `Person` уже появлялся в наших примерах, поэтому он просто дополнен до нужной функциональности. Добавим методы сравнения двух объектов `Person`:

```
//методы сравнения
private static int CompareName(Person obj1, Person obj2)
{
   return(string.Compare(obj1.name,obj2.name));
}
private static int CompareId(Person obj1, Person obj2)
{
   if( obj1.id > obj2.id) return(1);
   else return(-1);
}
private static int CompareSalary(Person obj1, Person obj2)
{
   if( obj1.salary > obj2.salary) return(1);
      else if(obj1.salary < obj2.salary)return(-1);
         else return(0);
}
private static int CompareSalaryName(Person obj1, Person obj2)
{
   if( obj1.salary > obj2.salary) return(1);
   else if(obj1.salary < obj2.salary)return(-1);
   else return(string.Compare(obj1.name,obj2.name));
}
```

Заметьте, методы закрыты и, следовательно, недоступны извне. Их четыре, но могло бы быть и больше, при возрастании сложности объекта растет число таких методов. Все методы имеют одну и ту же сигнатуру и удовлетворяют *контракту*, заданному *делегатом*, который будет описан чуть позже. Для каждого метода необходимо построить *экземпляр делегата*, который будет задавать ссылку на метод. Поскольку не все экземпляры нужны одновременно, то хотелось бы строить их динамически, в тот момент, когда они понадобятся. Это можно сделать, причем непосредственно в классе `Person`. Закрытые методы будем рассматривать как закрытые свойства и для каждого из них введем статическую процедуру-свойство, возвращающую в качестве результата экземпляр *делегата* со ссылкой на метод. Проще написать, чем объяснить на словах:

```
//делегаты как свойства
public static Persons.CompareItems SortByName
{
   get  {return(new Persons.CompareItems(CompareName));}
}
public static Persons.CompareItems SortById
{
   get  {return(new Persons.CompareItems(CompareId));}
```

```
}
public static Persons.CompareItems SortBySalary
{
   get  {return(new Persons.CompareItems(CompareSalary));}
}
public static Persons.CompareItems SortBySalaryName
{
   get  {return(new Persons.CompareItems(CompareSalaryName));}
}
```

Всякий раз, когда будет запрошено, например, свойство `SortByName` класса `Person`, будет возвращен объект функционального класса `Persons.CompareItems`, задающий ссылку на метод `CompareName` класса `Person`. Объект будет создаваться динамически в момент запроса.

Класс `Person` полностью определен, и теперь давайте перейдем к определению *контейнера*, содержащего объекты `Person`. Начну с определения свойств класса `Persons`:

```
class Persons
{       //контейнер объектов Person
   //делегат
   public delegate int CompareItems(Person obj1, Person obj2);
   private int freeItem = 0;
   const int  n = 100;
   private Person[]persons = new Person[n];
}
```

В классе определен функциональный класс - *делегат* `CompareItems`, задающий *контракт*, которому должны удовлетворять функции сравнения элементов.

Контейнер объектов реализован простейшим образом в виде массива объектов. Переменная `freeItem` - указатель на первый свободный элемент массива. Сам массив является закрытым свойством, и доступ к нему осуществляется благодаря индексатору:

```
//индексатор
public Person this[int num]
{
   get { return(persons[num-1]); }
   set { persons[num-1] = value; }
}
```

Добавим классический для контейнеров набор методов - добавление нового элемента, загрузка элементов из базы данных и печать элементов:

```
public void AddPerson(Person pers)
{
   if(freeItem < n)
   {
      Person p = new Person(pers);
      persons[freeItem++]= p;
   }
   else Console.WriteLine("Не могу добавить Person");
}
public void LoadPersons()
{
   //реально загрузка должна идти из базы данных
   AddPerson(new Person("Соколов",123, 750.0));
   AddPerson(new Person("Синицын",128, 850.0));
   AddPerson(new Person("Воробьев",223, 750.0));
```

```
   AddPerson(new Person("Орлов",129, 800.0));
   AddPerson(new Person("Соколов",133, 1750.0));
   AddPerson(new Person("Орлов",119, 750.0));
}//LoadPersons
public void PrintPersons()
{
   for(int i =0; i<freeItem; i++)
   {
      Console.WriteLine("{0,10} {1,5} {2,5}",
      persons[i].Name, persons[i].Id, persons[i].Salary);
   }
}//PrintPersons
```

Конечно, метод `LoadPerson` в реальной жизни устроен по-другому, но в нашем примере он свою задачу выполняет. А теперь определим метод сортировки записей с функциональным параметром, задающим тот или иной способ сравнения элементов:

```
//сортировка
public void SimpleSortPerson(CompareItems compare)
{
   Person temp = new Person();
   for(int i = 1; i<freeItem;i++)
      for(int j = freeItem -1; j>=i; j--)
         if (compare(persons[j],persons[j-1])==-1)
         {
            temp = persons[j-1];
            persons[j-1]=persons[j];
            persons[j] = temp;
         }
}//SimpleSortObject
}//Persons
```

Единственный аргумент метода `SimpleSortPerson` принадлежит классу `CompareItems`, заданному *делегатом*. Что касается метода сортировки, то реализован простейший алгоритм пузырьковой сортировки, со своей задачей он справляется. На этом проектирование классов закончено, нужная цель достигнута, показано, как можно в классе экземпляры *делегатов* задавать *как свойства*. Для завершения обсуждения следует продемонстрировать, как этим нужно пользоваться. Зададим, как обычно, тестирующую процедуру, в которой будут использоваться различные критерии сортировки:

```
public void TestSortPersons()
{
   Persons persons = new Persons();
   persons.LoadPersons();
   Console.WriteLine ("       Сортировка по имени: ");
   persons.SimpleSortPerson(Person.SortByName);
   persons.PrintPersons();
   Console.WriteLine ("       Сортировка по идентификатору: ");
   persons.SimpleSortPerson(Person.SortById);
   persons.PrintPersons();
   Console.WriteLine ("       Сортировка по зарплате: ");
   persons.SimpleSortPerson(Person.SortBySalary);
   persons.PrintPersons();
   Console.WriteLine ("       Сортировка по зарплате и имени: ");
   persons.SimpleSortPerson(Person.SortBySalaryName);
   persons.PrintPersons();
}//SortPersons
```

Результаты работы сортировки данных изображены на рис. 20.5.

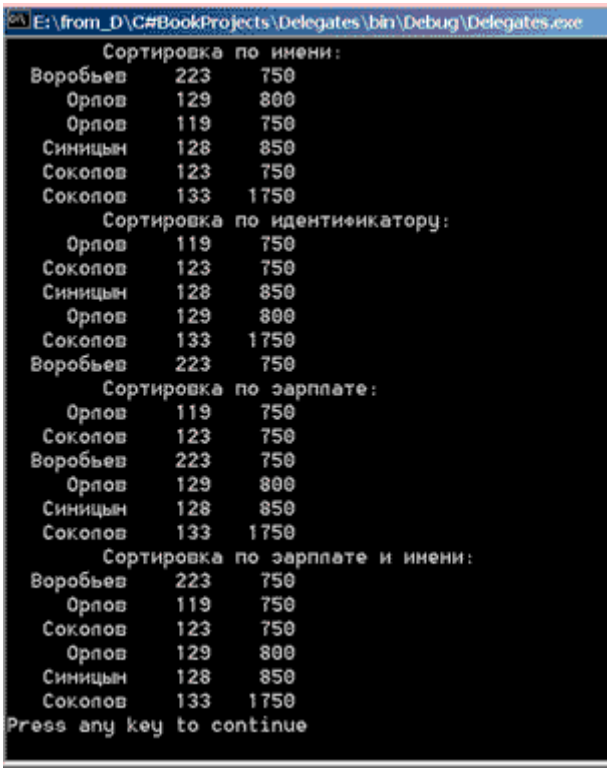

Рис. 20.5. Сортировка данных

Операции над делегатами. Класс Delegate

Давайте просуммируем то, что уже известно о функциональном типе данных. Ключевое слово `delegate` позволяет задать определение функционального типа (класса), фиксирующее *контракт*, которому должны удовлетворять все функции, принадлежащие классу. Функциональный класс можно рассматривать как ссылочный тип, экземпляры которого являются ссылками на функции. Заметьте, ссылки на функции - это безопасные по типу указатели, которые ссылаются на функции с жестко фиксированной сигнатурой, заданной *делегатом*. Следует также понимать, что это не простая ссылка на функцию. В том случае, когда *экземпляр делегата* инициирован динамическим методом, то экземпляр хранит ссылку на метод и на объект X, вызвавший этот метод.

Вместе с тем, объявление функционального типа не укладывается в синтаксис, привычный для C#. Хотелось бы писать, как принято:

```
Delegate FType = new Delegate(<определение типа>)
```

Но так объявлять переменные этого класса нельзя, и стоит понять, почему. Есть ли вообще *класс Delegate*? Ответ положителен - есть такой класс. При определении функционального типа, например:

```
public delegate int FType(int X);
```

переменная `FType` принадлежит *классу Delegate*. Почему же ее нельзя объявить привычным образом? Дело не только в синтаксических особенностях этого класса. Дело в том, что *класс Delegate* является абстрактным классом. Вот его объявление:

```
public abstract class Delegate: ICloneable, ISerializable
```

Для абстрактных классов реализация не определена, и это означает, что нельзя создавать экземпляры класса. *Класс Delegate* служит базовым классом для классов -

наследников. Но создавать наследников могут только компиляторы и системные программы - этого нельзя сделать в программе на C#. Именно поэтому введено ключевое слово `delegate`, которое косвенно позволяет работать с *классом Delegate*, создавая уже не абстрактный, а реальный класс. Заметьте, при этом все динамические и статические методы *класса Delegate* становятся доступными программисту.

Трудно, кажется, придумать, что можно делать с *делегатами*. Однако, у них есть одно замечательное свойство - их можно комбинировать. Представьте себе, что существует список работ, которые нужно выполнять, в зависимости от обстоятельств, в разных комбинациях. Если функции, выполняющие отдельные работы, принадлежат одному классу, то для решения задачи можно использовать *делегатов* и использовать *технику их комбинирования*. Замечу, что возможность *комбинирования делегатов* появилась, в первую очередь, для поддержания работы с событиями. Когда возникает некоторое событие, то сообщение о нем посылается разным объектам, каждый из которых по-своему обрабатывает событие. Реализуется эта возможность на основе *комбинирования делегатов*.

В чем суть *комбинирования делегатов*? Она прозрачна. К *экземпляру делегату* разрешается поочередно присоединять другие *экземпляры делегата* того же типа. Поскольку каждый экземпляр хранит ссылку на функцию, то в результате создается список ссылок. Этот список называется ***списком вызовов (invocation list)***. Когда вызывается экземпляр, имеющий *список вызова*, то поочередно, в порядке присоединения, начинают вызываться и выполняться функции, заданные ссылками. Так один вызов порождает выполнение списка работ.

Понятно, что, если есть операция присоединения *делегатов*, то должна быть и обратная операция, позволяющая удалять *делегатов* из списка.

Рассмотрим основные *методы и свойства класса Delegate*. Начнем с двух статических методов - `Combine` и `Remove`. Первый из них присоединяет *экземпляры делегата* к списку, второй - удаляет из списка. Оба метода имеют похожий синтаксис:

```
Combine(del1, del2)
Remove(del1, del2)
```

Аргументы `del1` и `del2` должны быть одного функционального класса. При добавлении `del2` в список, в котором `del2` уже присутствует, будет добавлен второй экземпляр. При попытке удаления `del2` из списка, в котором `del2` нет, Remove благополучно завершит работу, не выдавая сообщения об ошибке.

Класс Delegate относится к неизменяемым классам, поэтому оба метода возвращают ссылку на нового *делегата*. Возвращаемая ссылка принадлежит родительскому *классу Delegate*, поэтому ее необходимо явно преобразовать к нужному типу, которому принадлежат `del1` и `del2`. Обычное использование этих методов имеет вид:

```
del1 = (<type>) Combine(del1, del2);
del1 = (<type>) Remove(del1, del2);
```

Метод `GetInvocationList` является динамическим методом класса - он возвращает *список вызовов* экземпляра, вызвавшего метод. Затем можно устроить цикл `foreach`, поочередно получая элементы списка. Чуть позже появится пример, поясняющий необходимость подобной работы со списком.

Два динамических свойства `Method` и `Target` полезны для получения подробных сведений о *делегате*. Чаще всего они используются в процессе отражения, когда *делегат* поступает извне и необходима метаинформация, поставляемая с *делегатом*. Свойство `Method` возвращает объект класса `MethodInfo` из пространства имен

Reflection. Свойство Target возвращает информацию об объекте, вызвавшем *делегата*, в тех случаях, когда *делегат* инициируется не статическим методом класса, а динамическим, связанным с вызвавшим его объектом.

У *класса Delegate*, помимо методов, наследуемых от класса Object, есть еще несколько методов, но мы на них останавливаться не будем, они используются не столь часто.

Операции "+" и "-"

Наряду с методами, над *делегатами* определены и две **операции**: *"+" и "-"*, которые являются более простой формой записи добавления *делегатов* в *список вызовов* и удаления из списка. Операции заменяют собой методы Combine и Remove. Выше написанные присваивания объекту del1 с помощью этих операций могут быть переписаны в виде:

```
del1 +=del2;
del1 -=del2;
```

Как видите, запись становится проще, исчезает необходимость в задании явного приведения к типу. Ограничения на del1 и del2, естественно, остаются те же, что и для методов Combine и Remove.

Пример "Комбинирование делегатов"

Рассмотрим следующую ситуацию. Пусть есть городские службы: милиция, скорая помощь, пожарные. Каждая из служб по-своему реагируют на события, происходящие в городе. Построим примитивную модель жизни города, в которой случаются события и сообщения о них посылаются службам. В последующей лекции эта модель будет развита. Сейчас она носит формальный характер, демонстрируя, главным образом, работу с *делегатами*, заодно поясняя ситуации, в которых разумно *комбинирование делегатов*.

Начнем с построения класса с именем Combination, где, следуя уже описанной технологии, введем *делегатов* как закрытые свойства, доступ к которым идет через процедуру-свойство get. Три *делегата* одного класса будут описывать действия трех городских служб. Класс будет описываться ранее введенным *делегатом* MesToPers, размещенным в пространстве имен проекта. Вот программный код, в котором описаны функции, задающие действия служб:

```
class Combination
{
   private static void policeman(string mes)
   {
      //анализ сообщения
      if(mes =="Пожар!")
         Console.WriteLine(mes + " Милиция ищет виновных!");
      else
         Console.WriteLine(mes +" Милиция здесь!");
   }
   private static void ambulanceman(string mes)
   {
      if(mes =="Пожар!")
         Console.WriteLine(mes + " Скорая спасает пострадавших!");
      else
         Console.WriteLine(mes + " Скорая помощь здесь!");
   }
   private static void fireman(string mes)
   {
```

```
      if(mes =="Пожар!")
         Console.WriteLine(mes + " Пожарные тушат пожар!");
      else
         Console.WriteLine( mes + " Пожарные здесь!");
   }
}
```

Как видите, все три функции имеют не только одинаковую сигнатуру, но и устроены одинаково. Они анализируют приходящее к ним сообщение, переданное через параметр `mes`, а затем, в зависимости от результата, выполняют ту или иную работу, которая в данном случае сводится к выдаче соответствующего сообщения. Сами функции закрыты, и мы сейчас организуем к ним доступ:

```
public static MesToPers Policeman
{
   get {return (new MesToPers(policeman));}
}
public static MesToPers Fireman
{
   get {return (new MesToPers(fireman));}
}
public static MesToPers Ambulanceman
{
   get {return (new MesToPers(ambulanceman));}
}
```

Три статических открытых свойства - `Policeman`, `Fireman`, `Ambulanceman` - динамически создают экземпляры класса `MesToPers`, связанные с соответствующими закрытыми функциями класса.

Службы у нас есть, покажем, как с ними можно работать. С этой целью добавим в класс `Testing`, где проводятся различные эксперименты, следующую процедуру:

```
public void TestSomeServices()
{
   MesToPers Comb;
   Comb = (MesToPers)Delegate.Combine(Combination.Ambulanceman,
         Combination.Policeman);
   Comb = (MesToPers)Delegate.Combine(Comb,Combination.Fireman);
         Comb("Пожар!");
```

Вначале объявляется без инициализации функциональная переменная `Comb`, которой в следующем операторе присваивается ссылка на *экземпляр делегата*, созданного методом `Combine`, чей *список вызова* содержит ссылки на *экземпляры делегатов* `Ambulanceman` и `Policeman`. Затем к *списку вызовов* экземпляра `Comb` присоединяется новый кандидат `Fireman`. При вызове *делегата* `Comb` ему передается сообщение "Пожар!". В результате вызова `Comb` поочередно запускаются все три экземпляра входящие в список, каждому из которых передается сообщение.

Давайте теперь начнем поочередно отключать *делегатов*, вызывая затем `Comb` с новыми сообщениями:

```
Comb = (MesToPers)Delegate.Remove(Comb,Combination.Policeman);
//Такое возможно: попытка отключить не существующий элемент
Comb = (MesToPers)Delegate.Remove(Comb,Combination.Policeman);
Comb("Через 30 минут!");
Comb = (MesToPers)Delegate.Remove(Comb,Combination.Ambulanceman);
Comb("Через час!");
Comb = (MesToPers)Delegate.Remove(Comb,Combination.Fireman);
//Comb("Через два часа!"); // Comb не определен
```

В этом фрагменте поочередно отключаются разные службы - милиция, скорая помощь, пожарные, и каждый раз вызывается `Comb`. После последнего отключения, когда *список вызовов* становится пустым, вызов `Comb` приводит к ошибке, потому оператор вызова закомментирован.

Покажем теперь, что ту же работу можно выполнить, используя не методы, а операции:

```
//операции + и -
         Comb = Combination.Ambulanceman;
         Console.WriteLine( Comb.Method.Name);
         Comb+= Combination.Fireman;
         Comb+= Combination.Policeman;
         Comb("День города!");
         Comb -= Combination.Ambulanceman;
         Comb -= Combination.Fireman;
         Comb("На следующий день!");
      }//TestSomeServices
```

Обратите внимание, здесь демонстрируется вызов свойства `Method`, возвращающее объект, свойство `Name` которого выводится на печать. Результаты, порожденные работой этой процедуры, изображены на рис. 20.6.

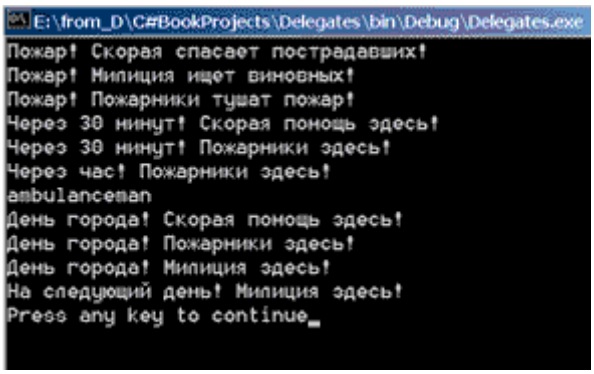

Рис. 20.6. Службы города

Пример "Плохая служба"

Как быть, если в *списке вызовов* есть "плохой" экземпляр, при вызове которого возникает ошибка, приводящая к выбрасыванию исключительной ситуации? Тогда стоящие за ним в очереди экземпляры не будут вызваны, хотя они вполне могли бы выполнить свою часть работы. В этом случае полезно использовать метод `GetInvocationList` и в цикле поочередно вызывать *делегатов*. Вызов *делегата* следует поместить в охраняемый блок, тогда при возникновении исключительной ситуации в обработчике ситуации можно получить и выдать пользователю всю информацию о нарушителе, а цикл продолжит выполнение очередных *делегатов* из *списка вызова*.

Добавим в класс `Combination` "плохого" кандидата, который пытается делить на ноль:

```
//метод, вызывающий исключительную ситуацию
public static void BadService(string mes)
{
   int i =7, j=5, k=0;
   Console.WriteLine("Bad Service: Zero Divide");
   j=i/k;
}
```

Создадим процедуру, в которой в *списке вызовов* есть хорошие и плохие кандидаты. Эта процедура использует управление исключительными ситуациями, о которых подробнее будет рассказано в последующих лекциях.

```
public void TestBadJob()
{
   MesToPers Comb;
   Comb = (MesToPers)Delegate.Combine(Combination.Ambulanceman,
      Combination.Policeman);
   Comb = (MesToPers)Delegate.Combine(Comb,
      new MesToPers(Combination.BadService));
   Comb = (MesToPers)Delegate.Combine(Comb,Combination.Fireman);
   foreach(MesToPers currentJob in Comb.GetInvocationList())
   {
      try
      {
         currentJob("Пожар!");
      }
      catch(Exception e)
      {
         Console.WriteLine(e.Message);
         Console.WriteLine(currentJob.Method.Name);
      }
   }
}//BadJob
```

Поясню, как будет работать эта процедура при ее вызове. Вначале две службы нормально отработают, но при вызове третьей службы возникнет исключительная ситуация "деление на ноль". Универсальный обработчик `Exception` перехватит эту ситуацию и напечатает как свойство `Message` объекта e, так и имя метода, вызвавшего исключительную ситуацию, используя свойство `Method` объекта, вызвавшего ситуацию. После завершения работы блока обработчика ситуации выполнение программы продолжится, выполнится следующий шаг цикла, и служба пожарных благополучно выполнит свою работу. Вот результаты вывода:

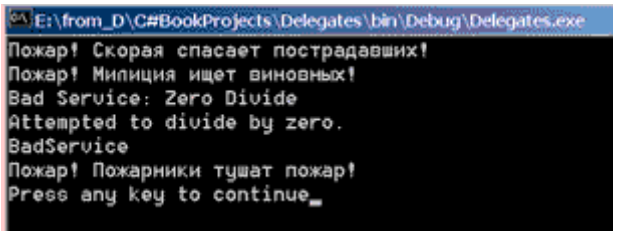

Рис. 20.7. "Плохая служба"

Разговор о *делегатах* еще не закончен. Он будет продолжен в следующей лекции, в которой рассмотрим классы с событиями. События основываются на *делегатах*.

21. Лекция: События

Классы с событиями

Каждый объект является экземпляром некоторого класса. Класс задает свойства и поведение своих экземпляров. Методы класса определяют поведение объектов, свойства - их состояние. Все объекты обладают одними и теми же методами и, следовательно, ведут себя одинаково. Можно полагать, что методы задают врожденное поведение объектов. Этого нельзя сказать о свойствах - значения свойств объектов различны, так что экземпляры одного класса находятся в разных состояниях. Объекты класса "человек" могут иметь разные свойства: один - высокий, другой - низкий, один - сильный, другой - умный. Но методы у них одни: есть и спать, ходить и бегать. Как сделать поведение объектов специфическим? Как добавить им поведение, характерное для данного объекта? Один из наиболее известных путей - это наследование. Можно создать класс-наследник, у которого, наряду с унаследованным родительским поведением, будут и собственные методы. Например, наследником класса "человек" может быть класс "человек_образованный", обладающий методами: читать и писать, считать и программировать.

Есть еще один механизм, позволяющий объектам вести себя по-разному в одних и тех же обстоятельствах. Это механизм событий, рассмотрением которого мы сейчас и займемся. Класс, помимо свойств и методов, может иметь события. Содержательно, событием является некоторое специальное состояние, в котором может оказаться объект класса. Так, для объектов класса "человек" событием может быть рождение или смерть, свадьба или развод. О событиях в мире программных объектов чаще всего говорят в связи с интерфейсными объектами, у которых события возникают по причине действий пользователя. Так, командная кнопка может быть нажата - событие `Click`, документ может быть закрыт - событие `Close`, в список может быть добавлен новый элемент - событие `Changed`.

Интерфейсные и многие другие программные объекты обладают стандартным набором предопределенных событий. В конце этой лекции мы поговорим немного об особенностях работы с событиями таких объектов. Сейчас же наше внимание будет сосредоточено на классах, создаваемых программистом. Давайте разберемся, как для таких классов создаются и обрабатываются события. Класс, решивший иметь события, должен уметь, по крайней мере, три вещи:

- объявить событие в классе;
- зажечь в нужный момент событие, передав обработчику необходимые для обработки аргументы. (Под зажиганием или включением события понимается некоторый механизм, позволяющий объекту уведомить клиентов класса, что у него произошло событие.);
- проанализировать, при необходимости, результаты события, используя значения выходных аргументов события, возвращенные обработчиком.

Заметьте, что, зажигая событие, класс посылает сообщение *получателям события* - объектам некоторых других классов. Будем называть класс, зажигающий событие, классом - ***отправителем сообщения (sender)***. Класс, чьи объекты получают сообщения, будем называть классом - ***получателем сообщения (receiver)***. Класс-отправитель сообщения, в принципе, не знает своих получателей. Он отправляет сообщение в межмодульное пространство. Одно и то же сообщение может быть получено и по-разному обработано произвольным числом объектов разных классов. Взгляните на схему, демонстрирующую взаимодействие объектов при посылке и получении сообщения.

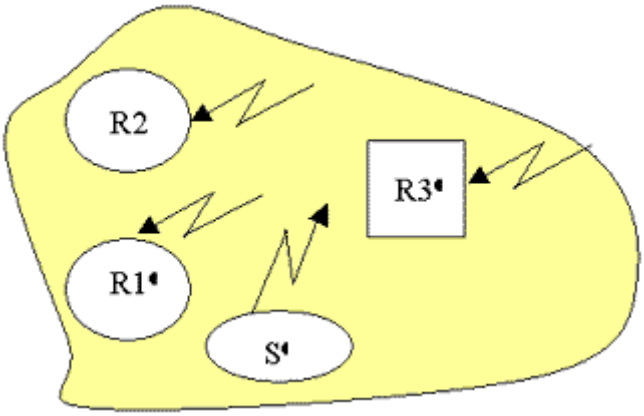

Рис. 21.1. Взаимодействие объектов. Посылка и получение сообщения о событии

Класс sender. Как объявляются события?

При проектировании *класса с событиями*, возможно, самое трудное - содержательная сторона дела. Какими событиями должен обладать класс, в каких методах и в какой момент зажигать то или иное событие?

Содержательную сторону будем пояснять на содержательных примерах. А сейчас рассмотрим технический вопрос: как объявляются события средствами языка C#? Прежде всего, уточним, что такое событие с программистской точки зрения. Начнем не с самого события, а с его обработчика. Обработчик события - это обычная процедура с аргументами. Понятно, что сообщение, посылаемое при зажигании события, является аналогом вызова процедуры. Поскольку сигнатура посылаемого сообщения должна соответствовать сигнатуре принимаемого сообщения, то объявление события синтаксически должно задавать сигнатуру процедуры.

Делегаты и события

Наверное, вы уже заметили, что схема работы с событиями вполне укладывается в механизм, определяемый делегатами. В C# каждое событие определяется делегатом, описывающим сигнатуру сообщения. Объявление события - это двухэтапный процесс:

- Вначале объявляется делегат - функциональный класс, задающий сигнатуру. Как отмечалось при рассмотрении делегатов, объявление делегата может быть помещено в некоторый класс, например, *класс Sender*. Но, чаще всего, это объявление находится вне класса в пространстве имен. Поскольку одна и та же сигнатура может быть у разных событий, то для них достаточно иметь одного делегата. Для некоторых событий можно использовать стандартных делегатов, встроенных в каркас. Тогда достаточно знать только их имена.
- Если делегат определен, то в *классе Sender*, создающем события, достаточно объявить событие как экземпляр соответствующего делегата. Это делается точно так же, как и при объявлении функциональных экземпляров делегата. Исключением является добавление служебного слова `event`. Формальный синтаксис объявления таков:

```
[атрибуты] [модификаторы]event [тип, заданный делегатом] [имя события]
```

Есть еще одна форма объявления, но о ней чуть позже. Чаще всего, атрибуты не задаются, а модификатором является модификатор доступа - `public`. Приведу пример объявления *делегата и события*, представляющего экземпляр этого делегата:

```
namespace Events
{
public delegate void FireEventHandler(object Sender,
```

```
   int time, int build);
public class TownWithEvents
{
   public event FireEventHandler FireEvent;
   ...
}//TownWithEvents
   ...
}//namespace Events
```

Здесь делегат `FireEventHandler` описывает класс событий, сигнатура которых содержит три аргумента. Событие `FireEvent` в классе `TownWithEvents` является экземпляром класса, заданного делегатом.

Как зажигаются события

Причины возникновения события могут быть разными. Поэтому вполне вероятно, что одно и то же событие будет зажигаться в разных методах класса в тот момент, когда возникнет одна из причин появления события. Поскольку действия по включению могут повторяться, полезно в состав методов класса добавить защищенную процедуру, включающую событие. Даже если событие зажигается только в одной точке, написание такой процедуры считается признаком хорошего стиля. Этой процедуре обычно дается имя, начинающееся со слова `On`, после которого следует имя события. Будем называть такую процедуру ***On-процедурой***. Она проста и состоит из вызова объявленного события, включенного в тест, который проверяет перед вызовом, а есть ли хоть один обработчик события, способный принять соответствующее сообщение. Если таковых нет, то нечего включать событие. Приведу пример:

```
protected virtual void OnFire(int time, int build)
{
   if (FireEvent!=null)
      FireEvent(this,time, build);
}
```

Хочу обратить внимание: те, кто принимает сообщение о событии, должны заранее присоединить обработчики событий к объекту `FireEvent`, задающему событие. Присоединение обработчиков должно предшествовать зажиганию события. При таком нормальном ходе вещей, найдется хотя бы один слушатель сообщения о событии - следовательно, `FireEvent` не будет равно `null`.

Заметьте также, что процедура `On` объявляется, как правило, с модификаторами `protected virtual`. Это позволяет потомкам класса переопределить ее, когда, например, изменяется набор аргументов события.

Последний шаг, который необходимо выполнить в *классе Sender* - это в нужных методах класса вызвать процедуру `On`. Естественно, что перед вызовом нужно определить значения входных аргументов события. После вызова может быть выполнен анализ выходных аргументов, определенных обработчиками события. Чуть позже рассмотрим более полные примеры, где появится вызов процедуры `On`.

Классы receiver. Как обрабатываются события

Объекты *класса Sender* создают события и уведомляют о них объекты, возможно, разных классов, названных нами *классами Receiver*, или клиентами. Давайте разберемся, как должны быть устроены *классы Receiver*, чтобы вся эта схема заработала.

Понятно, что *класс receiver* должен:

- иметь обработчик события - процедуру, согласованную по сигнатуре с функциональным типом делегата, который задает событие;
- иметь ссылку на объект, создающий событие, чтобы получить доступ к этому событию - event-объекту;
- уметь присоединить обработчик события к event-объекту. Это можно реализовать по-разному, но технологично это делать непосредственно в конструкторе класса, так что когда создается объект, получающий сообщение, он изначально готов принимать и обрабатывать сообщения о событиях. Вот пример, демонстрирующий возможное решение проблем:

```
public class FireMen
{
   private TownWithEvents MyNativeTown;
   public FireMen(TownWithEvents TWE)
   {
      this.MyNativeTown=TWE;
      MyNativeTown.FireEvent += new
         FireEventHandler(FireHandler);
   }
   private void FireHandler(object Sender, int time, int build)
   {
      Console.WriteLine("Fire at day {0}, in build {1}!",
         time, build);
   }
   public void GoOut()
   {
      MyNativeTown.FireEvent -= new FireEventHandler(FireHandler);
   }
}//FireMan
```

В классе `Fireman` есть ссылка на объект класса `TownWithEvents`, создающий события. Сам объект передается в конструкторе класса. Здесь же происходит присоединение обработчика события к event-объекту. Обработчик события `FireHandler` выводит сообщение на консоль.

Классы с событиями, допустимые в каркасе .Net Framework

Если создавать повторно используемые компоненты с событиями, работающие не только в проекте C#, то необходимо удовлетворять некоторым ограничениям. Эти требования предъявляются к делегату; они носят, скорее, синтаксический характер, не ограничивая существа дела.

Перечислю эти ограничения:

- делегат, задающий тип события, должен иметь фиксированную сигнатуру из двух аргументов: `delegate <Имя_делегата> (object sender, <Тип_аргументов> args);`
- первый аргумент задает объект `sender`, создающий сообщение. Второй аргумент `args` задает остальные аргументы - входные и выходные, - передаваемые обработчику. Тип этого аргумента должен задаваться классом, производным от встроенного в .Net Framework класса `EventArgs`. Если обработчику никаких дополнительных аргументов не передается, то следует просто указать класс `EventArgs`, передавая `null` в качестве фактического аргумента при включении события;
- рекомендуемое имя делегата - составное, начинающееся именем события, после которого следует слово `EventHandler`, например, `FireEventHandler`. Если никаких дополнительных аргументов обработчику не передается, то тогда можно

вообще делегата не объявлять, а пользоваться стандартным делегатом с именем `EventHandler`.

Пример "Списки с событиями"

В этом примере строится класс `ListWithChangedEvent`, являющийся потомком встроенного класса `ArrayList`, который позволяет работать со списками. В класс добавляется событие `Changed`, сигнализирующее обо всех изменениях элементов списка. Строятся два класса - `Receiver1` и `Receiver2`, получающие сообщения. В примере рассматривается взаимодействие нескольких объектов: два объекта посылают сообщения, три - принимают.

Начнем с объявления делегата:

```
// Объявление делегата
public delegate void ChangedEventHandler(object sender,
                        ChangedEventArgs args);
```

Здесь объявлен делегат `ChangedEventHandler`, по всем правилам хорошего стиля - его имя и его форма соответствует всем требованиям. Второй аргумент, задающий аргументы события, принадлежит классу `ChangedEventArgs`, производному от встроенного класса `EventArgs`. Рассмотрим, как устроен этот производный класс:

```
public class ChangedEventArgs:EventArgs
{
   private object item;
   private bool permit;
   public object Item
   {
      get {return(item);}
      set { item = value;}
   }
   public bool Permit
   {
      get {return(permit);}
      set { permit = value;}
   }
}//class ChangedEventArgs
```

У класса два закрытых свойства, доступ к которым осуществляется через процедуры-свойства `get` и `set`. Конечно, можно было бы в данной ситуации сделать их просто `public` - общедоступными. Свойство `Item` задает входной аргумент события, передаваемый обработчику события. Булево свойство `Permit` задает выходной аргумент события, получающий в обработчике значение `True`, если обработчик события дает добро на изменение элемента.

В модели, которую мы рассматриваем, предполагается, что обработчик события, получив уведомление об изменении элемента, анализирует ситуацию и может разрешить или не разрешить изменение, например, если значение элемента больше некоторого предельного значения.

Правильно ли, что обработчик события, а не сам класс, создающий событие, принимает решение о допуске изменения элемента списка? Все зависит от контекста. В прошлые времена молодые могли объявить о своей помолвке, но требовалось разрешение родителей на брак. Времена изменились - теперь на брак родительского благословения не требуется. Но в программистском мире ситуации, требующие внешнего разрешения, встречаются довольно часто.

Класс sender

Рассмотрим теперь, как устроен в нашем примере класс, создающий события. Начнем со свойств класса:

```
// Класс, создающий событие. Потомок класса ArrayList.
public class ListWithChangedEvent: ArrayList
{
   //Свойства класса: событие и его аргументы
   //Событие Changed, зажигаемое при всех изменениях
   //элементов списка.
   public event ChangedEventHandler Changed;
   //Аргументы события
   private ChangedEventArgs evargs = new ChangedEventArgs();
```

Первое свойство описывает событие `Changed`. Оно открыто, что позволяет присоединять к нему обработчиков событий. Второе закрытое свойство определяет аргументы события, передаваемые обработчикам.

Хороший стиль требует задания в классе процедуры `On`, включающей событие. Так и поступим:

```
//Методы класса: процедура On и переопределяемые методы.
//Процедура On, включающая событие
protected virtual void OnChanged(ChangedEventArgs args)
{
   if (Changed != null)
      Changed(this, args);
}
```

Процедура `OnChanged` полностью соответствует ранее описанному образцу, поэтому не требует дополнительных комментариев.

Наш класс, являясь наследником класса `ArrayList`, наследует все его методы. Переопределим методы, изменяющие элементы:

- метод `Add`, добавляющий новый элемент в конец списка;
- индексатор `this`, дающий доступ к элементу списка по индексу;
- метод `Clear`, производящий чистку списка.

```
//Переопределяемые методы, вызывающие событие Changed
//Добавление нового элемента
//при получении разрешения у обработчиков события
public override int Add(object value)
{
   int i=0;
   evargs.Item = value;
   OnChanged(evargs);
   if (evargs.Permit)
      i = base.Add(value);
   else
      Console.WriteLine("Добавление элемента запрещено." +
         "Значение =  {0}", value);
   return i;
}
public override void Clear()
{
   evargs.Item=0;
   OnChanged(evargs);
   base.Clear();
```

```
}
public override object this[int index]
{
   set
   {
      evargs.Item = value;
      OnChanged(evargs);
      if (evargs.Permit)
         base[index] = value;
      else
         Console.WriteLine("Замена элемента запрещена." +
            " Значение =  {0}", value);
   }
   get{return(base[index]);}
}
```

Обратите внимание на схему включения события, например, в процедуре `Add`. Вначале задаются входные аргументы, в данном случае `Item`. Затем вызывается процедура включения `OnChanged`. При зажигании выполнение процедуры `Add` прерывается. Запускаются обработчики, присоединенные к событию. Процедура `Add` продолжит работу только после окончания их работы. Анализ выходной переменной `Permit` позволяет установить, получено ли разрешение на изменение значения; при истинности значения этой переменной вызывается родительский метод `Add`, осуществляющий изменение значения. Это достаточно типичная схема работы с событиями.

Классы receiver

Мы построим два класса, объекты которых способны получать и обрабатывать событие `Changed`. Получать они будут одно и то же сообщение, а обрабатывать его будут по-разному. В нашей модельной задаче различие обработчиков сведется к выдаче разных сообщений. Поэтому достаточно разобраться с устройством одного класса, названного `EventReceiver1`. Вот его код:

```
class EventReceiver1
{
   private ListWithChangedEvent List;
   public EventReceiver1(ListWithChangedEvent list)
   {
      List = list;
      // Присоединяет обработчик к событию.
      OnConnect();
   }
   //Обработчик события - выдает сообщение.
   //Разрешает добавление элементов, меньших 10.
   private void ListChanged(object sender,
      ChangedEventArgs args)
   {
      Console.WriteLine("EventReceiver1: Сообщаю об
         изменениях:" + "Item ={0}", args.Item);
            args.Permit = ((int)args.Item < 10);
   }
   public void OnConnect()
   {
      //Присоединяет обработчик к событию
      List.Changed += new ChangedEventHandler(ListChanged);
   }
   public void OffConnect()
   {
      //Отсоединяет обработчик от события и удаляет список
      List.Changed -= new ChangedEventHandler(ListChanged);
```

```
      List = null;
   }
}//class EventReceiver1
```

Дам краткие комментарии.

- Среди закрытых свойств класса есть ссылка `List` на объект, создающий события.
- Конструктору класса передается фактический объект, который и будет присоединен к `List`. В конструкторе же происходит присоединение обработчика события к событию. Для этого, как положено, используется созданный в классе метод `OnConnect`.
- Класс содержит метод `OffConnect`, позволяющий при необходимости отключить обработчик от события.
- Обработчик события, анализируя переданный ему входной аргумент события `Item`, разрешает или не разрешает изменение элемента, формируя значение выходного аргумента `Permit`. Параллельно обработчик выводит на консоль сообщение о своей работе.

Класс `Receiver2` устроен аналогично. Приведу его текст уже без всяких комментариев:

```
class Receiver2
{
   private ListWithChangedEvent List;
   public Receiver2(ListWithChangedEvent list)
   {
      List = list;
      // Присоединяет обработчик к событию.
      OnConnect();
   }
   // Обработчик события - выдает сообщение.
   //Разрешает добавление элементов, меньших 20.
   private void ListChanged(object sender,
      ChangedEventArgs args)
   {
      Console.WriteLine("Receiver2: Сообщаю об изменениях:"
         + " Объект класса {0} : " + "Item ={1}",
            sender.GetType(), args.Item);
      args.Permit = ((int)args.Item < 20);
   }
   public void OnConnect()
   {
      //Присоединяет обработчик к событию
      List.Changed += new ChangedEventHandler(ListChanged);
      //Заметьте, допустимо только присоединение (+=),
      //но не замена (=)
      //List.Changed = new ChangedEventHandler(ListChanged);
   }
   public void OffConnect()
   {
      //Отсоединяет обработчик от события и удаляет список
      List.Changed -= new ChangedEventHandler(ListChanged);
      List = null;
   }
}//class Receiver2
```

Классы созданы, теперь осталось создать объекты и заставить их взаимодействовать, чтобы одни создавали события, а другие их обрабатывали. Эту часть работы будет выполнять тестирующая процедура класса `Testing`:

```
public void TestChangeList()
{
```

```
   //Создаются два объекта, вырабатывающие события
   ListWithChangedEvent list = new ListWithChangedEvent();
   ListWithChangedEvent list1 = new ListWithChangedEvent();
   //Создаются три объекта двух классов EventReceiver1 и
   //Receiver2, способные обрабатывать события класса
   //ListWithChangedEvent
   EventReceiver1 Receiver1 = new EventReceiver1(list);
   Receiver2 Receiver21 = new Receiver2 (list);
   Receiver2 Receiver22 = new Receiver2(list1);
   Random rnd = new Random();
   //Работа с объектами, приводящая к появлению событий
   list.Add(rnd.Next(20)); list.Add(rnd.Next(20));
      list[1] =17;
   int val = (int)list[0] + (int)list[1];list.Add(val);
   list.Clear();
   list1.Add(10);      list1[0] = 25; list1.Clear();
   //Отсоединение обработчика событий
   Receiver1.OffConnect();
   list.Add(21); list.Clear();
}
```

В заключение взгляните на результаты работы этой процедуры.

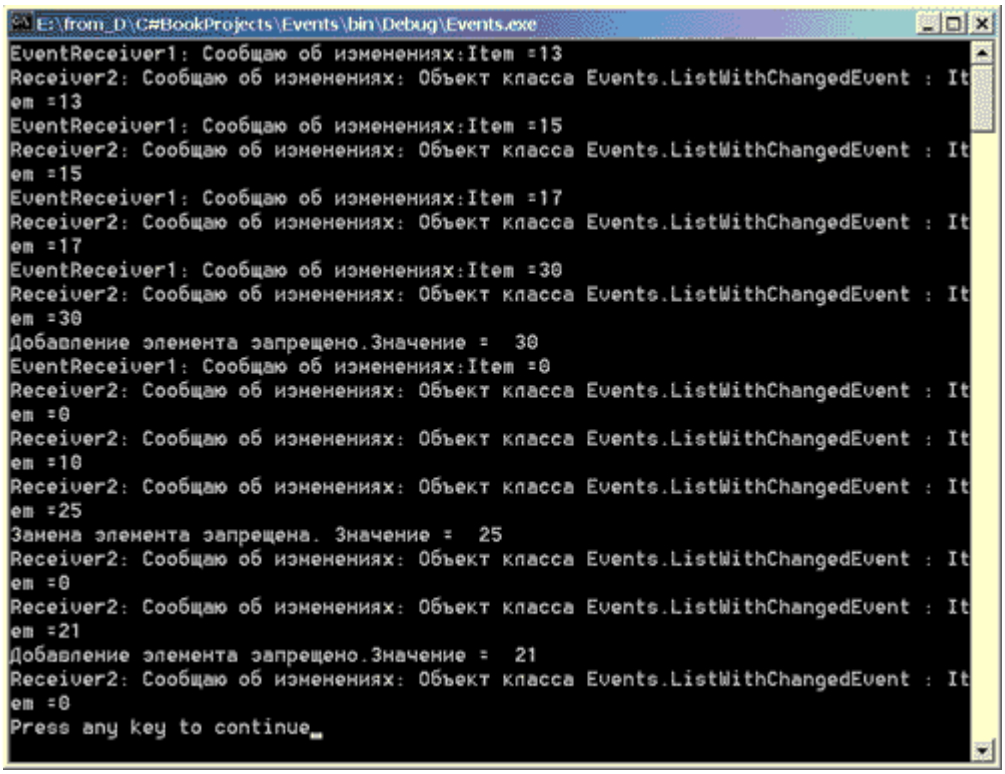

Рис. 21.2. События в мире объектов

Две проблемы с обработчиками событий

Объекты, создающие события, ничего не знают об объектах, обрабатывающих эти события. Объекты, обрабатывающие события, ничего не знают друг о друге, независимо выполняя свою работу. В такой модели могут возникать определенные проблемы. Рассмотрим некоторые из них.

Игнорирование коллег

Задумывались ли вы, какую роль играет ключевое слово `event`, появляющееся при объявлении события? Событие, объявленное в классе, представляет экземпляр

делегата. В предыдущей лекции, когда речь шла о делегатах, их экземпляры объявлялись без всяких дополнительных ключевых слов.

Слово "event" играет важную роль, позволяя решить проблему, названную нами "***игнорированием коллег***". В чем ее суть? В том, что некоторые из *классов Receiver* могут вести себя некорректно по отношению к своим коллегам, занимающимся обработкой того же события. При присоединении обработчика события в *классе Receiver* можно попытаться вместо присоединения обработчика выполнить операцию присваивания, игнорируя, тем самым, уже присоединенный список обработчиков. Взгляните еще раз на процедуру `OnConnect` класса `Receiver2`; там демонстрируется такая попытка в закомментированном операторе. Аналогично, в процедуре `OffConnect` вместо отсоединения (операции `-`) можно попытаться присвоить событию значение `null`, отсоединяя тем самым всех других обработчиков.

С этим как-то нужно бороться. Ключевое слово "event" дает указание компилятору создать для события закрытое поле, доступ к которому можно получить только через два автоматически создаваемых для события метода: `Add`, выполняющий операцию присоединения "`+=`", и `Remove`, выполняющий обратную операцию отсоединения "`-=`". Никаких других операций над событиями выполнять нельзя. Тем самым, к счастью, решается проблема *игнорирования коллег*. Ошибки некорректного поведения *класса Receiver* ловятся еще на этапе трансляции.

Переопределение значений аргументов события

Обработчику события, как правило, передаются входные и выходные аргументы, характеризующие событие. Они необходимы, чтобы обработчик мог нужным образом обработать событие. Но работа с аргументами требует аккуратного с ними обращения. Могут возникать проблемы, связанные с тем, что обработчик может *переопределить значения аргументов* в процессе своей работы.

Приведенный выше пример "Работа со списками" демонстрирует не самый лучший способ определения аргументов, провоцирующий *классы Receiver* на некорректное обращение с аргументами. Напомню, в классе `ChangedEventArgs`, определяющем аргументы события, оба свойства `item` и `permit` являются закрытыми. Но определены процедуры - свойства `Item` и `Permit`, реализующие полный доступ к свойствам, поскольку определены обе процедуры `get` и `set`. Это несколько облегчило задачу, поскольку позволило изменять значение входного аргумента `item` перед зажиганием события для передачи его обработчику. Но входной аргумент оказался не защищенным, и обработчик события может не только использовать это значение для анализа, но и изменить его в качестве побочного эффекта своей работы. В этом случае другой обработчик будет работать уже с некорректным значением. Что еще хуже - это измененное значение может использовать и класс, в процессе своей дальнейшей работы. Поэтому входные аргументы события должны быть закрытыми для обработчиков событий. Это нетрудно сделать, и я приведу необходимые уточнения.

- В классе `ChangedEventArgs` следует изменить процедуру-свойство `Item`, удалив процедуру `Set`, разрешающую изменение свойства. В качестве компенсации в класс следует добавить конструктор с аргументом, что позволит в классе, создающем событие, создавать объект класса `ChangedEventArgs` с нужным значением свойства `item`. Приведу соответствующий код:

```
public object Item
{
   get {return(item);}
   //set { item = value;}
}
public ChangedEventArgs(object item)
{
```

```
        this.item = item;
    }
```

- В методы класса `ListWithChangedEvent`, зажигающие события, нужно ввести изменения. Теперь перед каждым вызовом нужно создавать новый объект, задающий аргументы. Вот измененный код:

```
  public override int Add(object value)
  {
     int i=0;
     ChangedEventArgs evargs = new ChangedEventArgs(value);
     //evargs.Item = value;
     OnChanged(evargs);
     if (evargs.Permit) i = base.Add(value);
     else
        Console.WriteLine("Добавление элемента запрещено." +
           "Значение =  {0}", value);
     return i;
  }
  public override void Clear()
  {
     ChangedEventArgs evargs = new ChangedEventArgs(0);
     //evargs.Item=0;
     OnChanged(evargs);
     base.Clear();
  }
  public override object this[int index]
  {
     set
     {
        ChangedEventArgs evargs = new ChangedEventArgs(value);
        //evargs.Item = value;
        OnChanged(evargs);
        if (evargs.Permit)
           base[index] = value;
        else
           Console.WriteLine("Замена элемента запрещена." +
              " Значение =  {0}", value);
     }
     get  {return(base[index]);}
  }
```

Таким образом, обработчикам можно запретить изменение входных аргументов события. Но есть еще выходные аргументы события, значения которых определяются в обработчике; в нашем примере это аргумент `Permit`.

И здесь возникает коллизия интересов - каждый обработчик по своему может формировать значения выходных аргументов, не обращая внимания на результаты работы предыдущих обработчиков. Преимуществом в таких ситуациях обладает последний работающий обработчик события.

Эта проблема остается открытой, в языке C# здесь "дыра" - нет специальных средств, позволяющих избежать или, по крайней мере, предупредить о возникновении подобной ситуации. Вся ответственность лежит на программисте, который может выбрать некоторую стратегию решения проблемы, отдавая, например, предпочтение решению одного из обработчиков или вырабатывая итоговое решение, учитывающее все частные решения.

Итак, если событие имеет аргументы, то все входные аргументы должны быть закрыты для обработчиков события. Если обработчиков несколько, то лучше или не использовать выходных аргументов, или аккуратно запрограммировать логику обработчиков, которая учитывает решения, полученные коллегами - ранее отработавшими обработчиками события.

Классы с большим числом событий

Как было сказано, каждое событие класса представляется полем этого класса. Если у класса много объявленных событий, а реально возникает лишь малая часть из них, то предпочтительнее *динамический подход*, когда память отводится только фактически возникшим событиям. Это несколько замедляет время выполнения, но экономит память. Решение зависит от того, что в данном контексте важнее - память или время. Для реализации *динамического подхода* в языке предусмотрена возможность задания пользовательских методов `Add` и `Remove` в момент объявления события. Это и есть другая форма объявления события, упоминавшаяся ранее. Вот ее примерный синтаксис:

```
public event <Имя Делегата> <Имя события>
{
   add {...}
   remove {...}
}
```

Оба метода должны быть реализованы, при этом для хранения делегатов используется некоторое хранилище. Именно так реализованы классы для большинства интерфейсных объектов, использующие хэш-таблицы для хранения делегатов.

Давайте построим небольшой пример, демонстрирующий такой способ объявления и работы с событиями. Вначале построим класс с несколькими событиями:

```
class ManyEvents
{
   //хэш таблица для хранения делегатов
   Hashtable DStore = new Hashtable();
   public event EventHandler Ev1
   {
      add
      {
      DStore["Ev1"]= (EventHandler)DStore["Ev1"]+ value;
      }
      remove
      {
         DStore["Ev1"]= (EventHandler)DStore["Ev1"]- value;
      }
   }
   public event EventHandler Ev2
   {
      add
      {
         DStore["Ev2"]= (EventHandler)DStore["Ev2"]+ value;
      }
      remove
      {
         DStore["Ev2"]= (EventHandler)DStore["Ev2"]- value;
      }
   }
   public event EventHandler Ev3
   {
      add
```

```
        {
            DStore["Ev3"]= (EventHandler)DStore["Ev3"]+ value;
        }
        remove
        {
            DStore["Ev3"]= (EventHandler)DStore["Ev3"]- value;
        }
    }
    public event EventHandler Ev4
    {
        add
        {
            DStore["Ev4"]= (EventHandler)DStore["Ev4"]+ value;
        }
        remove
        {
            DStore["Ev4"]= (EventHandler)DStore["Ev4"]- value;
        }
    }
    public void SimulateEvs()
    {
        EventHandler ev = (EventHandler) DStore["Ev1"];
        if(ev != null) ev(this, null);
        ev = (EventHandler) DStore["Ev3"];
        if(ev != null) ev(this, null);
    }
}//class ManyEvents
```

В нашем классе созданы четыре события и хэш-таблица `DStore` для их хранения. Все события принадлежат встроенному классу `EventHandler`. Когда к событию будет присоединяться обработчик, автоматически будет вызван метод `add`, который динамически создаст элемент хэш-таблиц. Ключом элемента является, в данном случае, строка с именем события. При отсоединении обработчика будет исполняться метод `remove`, выполняющий аналогичную операцию над соответствующим элементом хэш-таблицы. В классе определен также метод `SimulateEvs`, при вызове которого зажигаются два из четырех событий - `Ev1` и `Ev3`.

Рассмотрим теперь класс `ReceiverEvs`, слушающий события. Этот класс построен по описанным ранее правилам. В нем есть ссылка на класс, создающий события; конструктор с параметром, которому передается реальный объект такого класса; четыре обработчика события - по одному на каждое, и метод `OnConnect`, связывающий обработчиков с событиями. Вот код класса:

```
class ReceiverEvs
{
    private ManyEvents manyEvs;
    public ReceiverEvs( ManyEvents manyEvs)
    {
        this.manyEvs = manyEvs;
        OnConnect();
    }
    public void OnConnect()
    {
        manyEvs.Ev1 += new EventHandler(H1);
        manyEvs.Ev2 += new EventHandler(H2);
        manyEvs.Ev3 += new EventHandler(H3);
        manyEvs.Ev4 += new EventHandler(H4);
    }
    public void H1(object s, EventArgs e)
    {
        Console.WriteLine("Событие Ev1");
    }
```

```
   public void H2(object s, EventArgs e)
   {
      Console.WriteLine("Событие Ev2");
   }
   public void H3(object s, EventArgs e)
   {
      Console.WriteLine("Событие Ev3");
   }
   public void H4(object s, EventArgs e)
   {
      Console.WriteLine("Событие Ev4");
   }
}//class ReceiverEvs
```

Тестирующая процедура состоит из нескольких строчек, в которых создаются нужные объекты и запускается метод `Simulate`, зажигающий события:

```
public void TestManyEvents()
{
   ManyEvents me = new ManyEvents();
   ReceiverEvs revs = new ReceiverEvs(me);
   me.SimulateEvs();
}
```

Все работает предусмотренным образом.

Проект "Город и его службы"

Завершить лекцию о событиях хочется содержательным учебным проектом, в котором моделируется жизнь города, происходящие в нем события и реакция на них городских служб. Наша главная цель в данном проекте - еще раз показать, как возникающее событие, в данном случае - пожар в одном из домов города, обрабатывается по-разному городскими службами - пожарными, милицией, скорой помощью. Конечно, все упрощено, в реальном городе событиями являются не только пожары и преступления, но и более приятные ситуации: день города, открытие фестивалей и выставок, строительство новых театров и институтов.

Начнем с описания класса, задающего наш город. Этот класс уже появлялся и в этой, и в предыдущей лекции, здесь его описание будет расширено. Начнем со свойств класса:

```
public class NewTown
{
   //свойства
   private int build, BuildingNumber; //дом и число домов в городе
   private int day, days;             //Текущий день года
   //городские службы
   private Police policeman ;
   private Ambulance ambulanceman ;
   private FireDetect fireman ;
   //события в городе
   public event FireEventHandler Fire;
   //моделирование случайных событий
   private Random rnd = new Random();
   //вероятность пожара в доме в текущий день: p= m/n
   private int m = 3, n= 10000;
```

В нашем городе есть дома; есть время, текущее день за днем; городские службы; событие "пожар", которое, к сожалению, может случайно с заданной вероятностью возникать каждый день в каждом доме. Рассмотрим конструктор объектов нашего класса:

```
//конструктор класса
public NewTown(int TownSize, int Days)
{
   BuildingNumber = rnd.Next(TownSize);
   days = Days;
   policeman = new Police(this);
   ambulanceman= new Ambulance(this);
   fireman= new FireDetect(this);
   policeman.On();
   ambulanceman.On();
   fireman.On();
}
```

При создании объектов этого класса задается размер города - число его домов и период времени, в течение которого будет моделироваться жизнь города. При создании объекта создаются его службы - объекты соответствующих классов `Police`, `Ambulance`, `FireDetect`, которым предается ссылка на сам объект "город". При создании служб вызываются методы `On`, подключающие обработчики события `Fire` каждой из этих служб к событию.

В соответствии с ранее описанной технологией определим метод `OnFire`, включающий событие:

```
protected virtual void OnFire(FireEventArgs e)
{
   if(Fire != null)
      Fire(this, e);
}
```

Где и когда будет включаться событие `Fire`? Напишем метод, моделирующий жизнь города, где для каждого дома каждый день будет проверяться, а не возник ли пожар, и, если это случится, то будет включено событие `Fire`:

```
public void LifeOurTown()
{
   for(day = 1; day<=days; day++)
      for(build =1; build <= BuildingNumber; build++)
      {
         if( rnd.Next(n) <=m) //загорелся дом
         {
            //аргументы события
         FireEventArgs e = new FireEventArgs(build, day, true);
            OnFire(e);
            if(e.Permit)
               Console.WriteLine("Пожар потушен!" +
                  " Ситуация нормализована.");
            else Console.WriteLine("Пожар продолжается." +
               " Требуются дополнительные средства!");
         }
      }
}
```

Рассмотрим теперь *классы Receiver*, обрабатывающие событие `Fire`. Их у нас три, по одному на каждую городскую службу. Все три класса устроены по одному образцу. Напомню, каждый такой разумно устроенный класс, кроме обработчика события, имеет конструктор, инициализирующий ссылку на объект, создающий события, методы подключения и отсоединения обработчика от события. В такой ситуации целесообразно построить вначале абстрактный *класс Receiver*, в котором будет предусмотрен обработчик события, но не задана его реализация, а затем для каждой службы построить класс-потомок. Начнем с описания родительского класса:

```
public abstract class Receiver
   {
      private NewTown town;
      public Receiver(NewTown town)
         {this.town = town;}
      public void On()
      {
         town.Fire += new FireEventHandler(It_is_Fire);
      }
      public void Off()
      {
         town.Fire -= new FireEventHandler(It_is_Fire);
         town = null;
      }
      public abstract void It_is_Fire(object sender,
         FireEventArgs e);
   }//class Receiver
```

Для классов потомков абстрактный метод `It_is_Fire` будет определен. Вот их описания:

```
public class Police : Receiver
   {
      public Police (NewTown town): base(town){}
      public override void It_is_Fire(object sender,
         FireEventArgs e)
      {
         Console.WriteLine("Пожар в доме {0}. День {1}-й."
            + " Милиция ищет виновных!", e.Build,e.Day);
         e.Permit &= true;
      }
   }// class Police
   public class FireDetect : Receiver
   {
      public FireDetect (NewTown town): base(town){}
      public override void It_is_Fire(object sender, FireEventArgs e)
      {
         Console.WriteLine("Пожар в доме {0}. День {1}-й."+
            " Пожарные тушат пожар!", e.Build,e.Day);
         Random rnd = new Random(e.Build);
         if(rnd.Next(10) >5)
            e.Permit &= false;
         else e.Permit &=true;
      }
   }// class FireDetect
   public class Ambulance : Receiver
   {
      public Ambulance(NewTown town): base(town){}
      public override void It_is_Fire(object sender,
         FireEventArgs e)
      {
         Console.WriteLine("Пожар в доме {0}. День {1}-й."+
            " Скорая спасает пострадавших!", e.Build,e.Day);
         e.Permit &= true;
      }
   }// class Ambulance
```

Для каждого потомка задан конструктор, вызывающий базовый метод родителя. Каждый потомок по-своему определяет обработчика события `Fire`. Обратите внимание на то, как в данном проекте решается проблема с выходным параметром события - `Permit`. Принята следующая стратегия: возвращаемое значение `Permit` будет истинно, если все обработчики согласны с этим. Поэтому каждый обработчик использует

конъюнкцию выработанного им значения со значением, пришедшим от предыдущего обработчика. В данном примере все зависит от пожарных, которые могут вырабатывать разные решения.

Для полноты картины необходимо показать, как выглядит класс, задающий аргументы события, который, как и положено, является потомком класса `EventArgs`:

```
public class FireEventArgs : EventArgs
{
   private int build;
   private int day;
   private bool permit;
   public int Build
   {
      get{ return(build);} ///set{ build = value;}
   }
   public int Day
   {
      get{ return(day);} ///set{ day = value;}
   }
   public bool Permit
   {
      get{ return(permit);}
      set{ permit = value;}
   }
   public FireEventArgs(int build, int day, bool permit)
   {
      this.build = build; this.day = day; this.permit = permit;
   }
}//class FireEventArgs
```

Входные параметры события - `build` и `day` защищены от обработчиков события, а корректность выходного параметра гарантируется тщательным программированием самих обработчиков.

Для завершения проекта нам осталось определить тестирующую процедуру в классе `Testing`, создающую объекты и запускающую моделирование жизни города:

```
public void TestLifeTown()
{
   NewTown sometown = new NewTown(100,100);
   sometown.LifeOurTown();
}
```

Результаты ее работы зависят от случайностей. Вот как выглядит один из экспериментов:

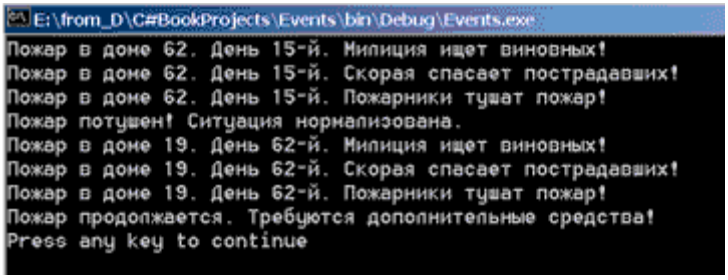

Рис. 21.3. События в жизни города

22. Лекция: Универсальность. Классы с родовыми параметрами

Наследование и универсальность

Необходимость в универсализации возникает с первых шагов программирования. Одна из первых процедур, появляющихся при обучении программированию - это процедура ***свопинга***:обмен значениями двух переменных одного типа. Выглядит она примерно так:

```
public void Swap(ref T x1, ref T x2)
{
   T temp;
   temp = x1; x1 = x2; x2 = temp;
}
```

Если тип `T` - это вполне определенный тип, например `int`, `string` или `Person`, то никаких проблем не существует, все совершенно прозрачно. Но как быть, если возникает необходимость обмена данными разного типа? Неужели нужно писать копии этой процедуры для каждого типа? Проблема легко решается в языках, где нет контроля типов - там достаточно иметь единственный экземпляр такой процедуры, прекрасно работающий, но лишь до тех пор, пока передаются аргументы одного типа. Когда же процедуре будут переданы фактические аргументы разного типа, то немедленно возникнет ошибка периода выполнения, и это слишком дорогая плата за *универсальность*.

В типизированных языках, не обладающих механизмом универсализации, выхода практически нет - приходится писать многочисленные копии Swap.

До недавнего времени Framework .Net и соответственно язык C# не поддерживали *универсальность*. Так что те, кто работает с языком C#, входящим в состав Visual Studio 2003 и ранних версий, должны смириться с отсутствием *универсальных классов*. Но в новой версии Visual Studio 2005, носящей кодовое имя Whidbey, проблема решена, и программисты получили наконец долгожданный механизм *универсальности*. Я использую в примерах этой лекции бета-версию Whidbey.

Замечу, что хотя меня прежде всего интересовала реализация *универсальности*, но и общее впечатление от Whidbey самое благоприятное.

Для достижения *универсальности* процедуры Swap следует рассматривать тип `T` как ее параметр, такой же, как и сами аргументы `x1` и `x2`. Суть *универсальности* в том, чтобы в момент вызова процедуры передавать ей не только фактические аргументы, но и их *фактический тип*.

Под ***универсальностью (genericity)*** понимается способность класса объявлять используемые им типы как параметры. Класс с параметрами, задающими типы, называется ***универсальным классом (generic class)***. Терминология не устоялась и синонимами термина "*универсальный класс*" являются термины: родовой класс, параметризованный класс, класс с родовыми параметрами. В языке C++ *универсальные классы* называются ***шаблонами (template)***.

Синтаксис универсального класса

Объявить класс C# универсальным просто: для этого достаточно указать в объявлении класса, какие из используемых им типов являются параметрами. Список *типовых параметров* класса, заключенный в угловые скобки, добавляется к имени класса:

```
class MyClass<T1, ... Tn> {...}
```

Как и всякие формальные параметры, `Ti` являются именами (идентификаторами). В теле класса эти имена могут задавать типы некоторых *полей* класса, типы аргументов и возвращаемых значений методов класса. В некоторый момент (об этом скажем чуть позже) формальные имена типов будут заменены фактическими параметрами, представляющими уже конкретные типы - имена встроенных классов, классов библиотеки FCL, классов, определенных пользователем.

В C# универсальными могут быть как классы, так и все их частные случаи - интерфейсы, структуры, делегаты, события.

Класс с универсальными методами

Специальным частным случаем *универсального класса* является класс, не объявляющий сам параметров, но разрешающий делать это своим методам. Давайте начнем рассмотрение *универсальности* с этого частного случая. Вот как выглядит класс, содержащий универсальный метод `swap`:

```
class Change
{
   static public void Swap<T>(ref T x1, ref T x2)
   {
      T temp;
      temp = x1; x1 = x2; x2 = temp;
   }
}
```

Как видите, сам класс в данном случае не имеет родовых параметров, но зато универсальным является статический метод класса `swap`, имеющий родовой параметр типа `T`. Этому типу принадлежат аргументы метода и локальная переменная `temp`. Всякий раз при вызове метода ему, наряду с фактическими аргументами, будет передаваться и *фактический тип*, заменяющий тип `T` в описании метода. О некоторых деталях технологии подстановки и выполнения метода поговорим в конце лекции, сейчас же лишь отмечу, что реализация вызова универсального метода в C# не приводит к существенным накладным расходам.

Рассмотрим тестирующую процедуру из традиционного для наших примеров класса `Testing`, в которой интенсивно используется вызов метода swap для различных типов переменных:

```
public void TestSwap()
{
   int x1 = 5, x2 = 7;
   Console.WriteLine("до обмена: x1={0}, x2={1}",x1, x2);
   Change.Swap<int>(ref x1, ref x2);
   Console.WriteLine("после обмена: x1={0}, x2={1}", x1, x2);
   string s1 = "Савл", s2 = "Павел";
   Console.WriteLine("до обмена: s1={0}, s2={1}", s1, s2);
   Change.Swap<string>(ref s1, ref s2);
   Console.WriteLine("после обмена: s1={0}, s2={1}", s1, s2);
   Person pers1 = new Person("Савлов", 25, 1500);
   Person pers2 = new Person("Павлов", 35, 2100);
   Console.WriteLine("до обмена: ");
   pers1.PrintPerson(); pers2.PrintPerson();
   Change.Swap<Person>(ref pers1, ref pers2);
   Console.WriteLine("после обмена:");
   pers1.PrintPerson(); pers2.PrintPerson();
}
```

Обратите внимание на строки, осуществляющие вызов метода:

```
Change.Swap<int>(ref x1, ref x2);
Change.Swap<string>(ref s1, ref s2);
Change.Swap<Person>(ref pers1, ref pers2);
```

В момент вызова метода передаются фактические аргументы и *фактические типы*. В данном примере в качестве *фактических типов* использовались встроенные типы `int` и `string` и тип `Person`, определенный пользователем. Общая ситуация такова: если в классе объявлен универсальный метод со списком параметров `M<T1, ...Tn> (...)`, то метод вызывается следующим образом: `M<TYPE1, ... TYPEn>(...)`, где `TYPEi` - это конкретные типы.

Еще раз напомню, что все эти примеры построены в Whidbey, и вот как выглядят внешний вид среды разработки и окно с результаты работы этой процедуры.

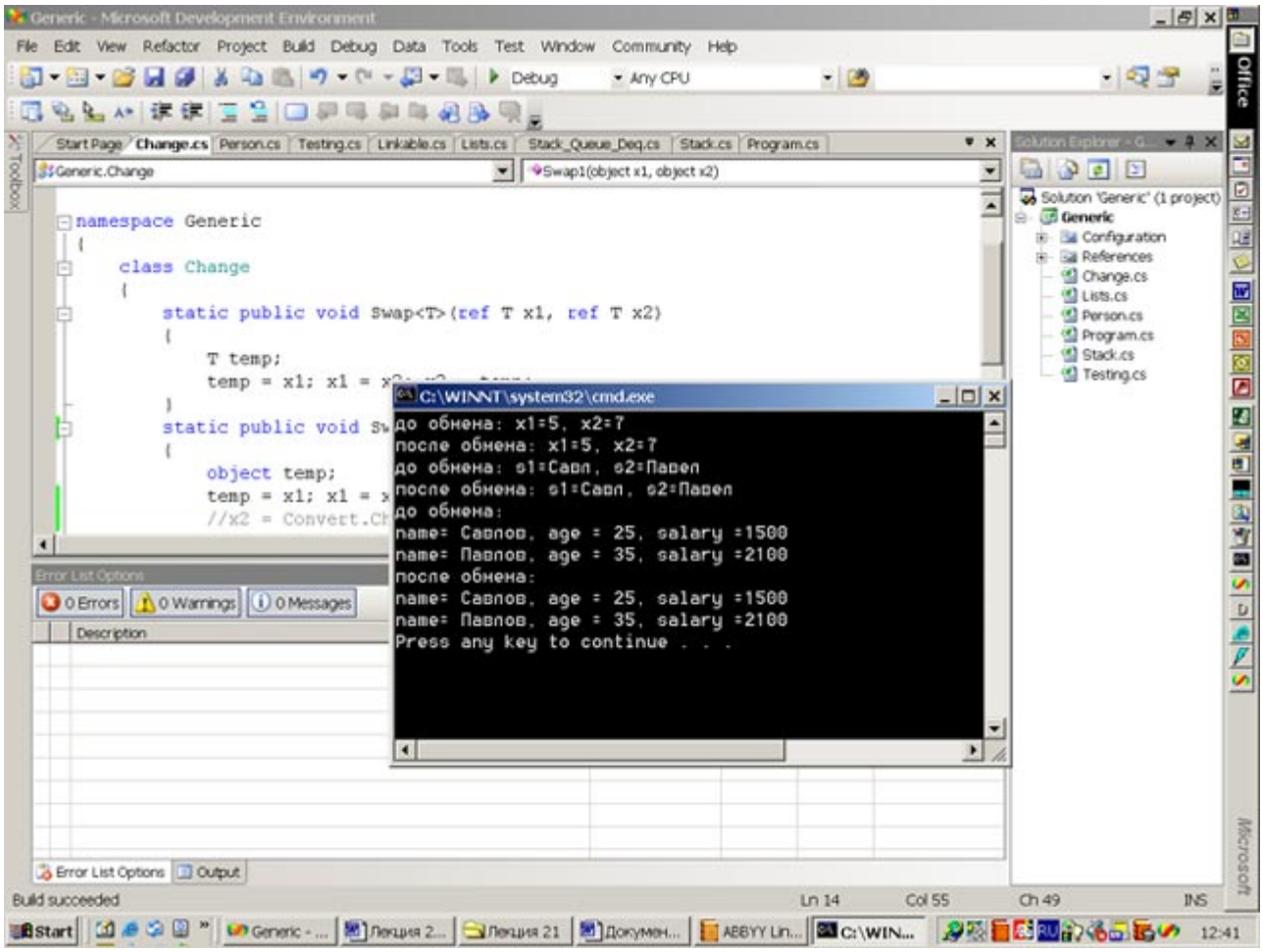

Рис. 22.1. Результаты работы универсальной процедуры swap

В этом примере использовался класс `Person`, и поскольку он появится и в следующих примерах, то приведу его текст:

```
class Person
{
   public Person(string name, int age, double salary)
   {
      this.name = name; this.age = age; this.salary = salary;
   }
   public string name;
   public int age;
   public double salary;
   public void PrintPerson()
```

```
    {
        Console.WriteLine("name= {0}, age = {1}, salary ={2}",
            name, age, salary);
    }
}
```

Два основных механизма объектной технологии

Наследование и *универсальность* являются двумя основными механизмами, обеспечивающими мощность объектной технологии разработки. *Наследование* позволяет специализировать операции класса, уточнить, как должны выполняться операции. Универсализация позволяет специализировать данные, уточнить, над какими данными выполняются операции.

Эти механизмы взаимно дополняют друг друга. *Универсальность* можно ограничить (об этом подробнее будет сказано ниже), указав, что тип, задаваемый родовым параметром, обязан быть наследником некоторого класса и/или ряда интерфейсов. С другой стороны, когда формальный тип `T` заменяется *фактическим типом* `TFact`, то там, где разрешено появляться объектам типа `TFact`, разрешены и объекты, принадлежащие классам-потомкам `TFact`.

Эти механизмы в совокупности обеспечивают бесшовный процесс разработки программных систем, начиная с этапов спецификации и проектирования системы и заканчивая этапами реализации и сопровождения. На этапе задания спецификаций появляются абстрактные, *универсальные классы*, которые в ходе разработки становятся вполне конкретными классами с конкретными типами данных. Механизмы *наследования* и универсализации позволяют существенно сократить объем кода, описывающего программную систему, поскольку потомки не повторяют наследуемый код своих родителей, а единый код *универсального класса* используется при каждой конкретизации типов данных. На рис. 22.2 показан схематически процесс разработки программной системы.

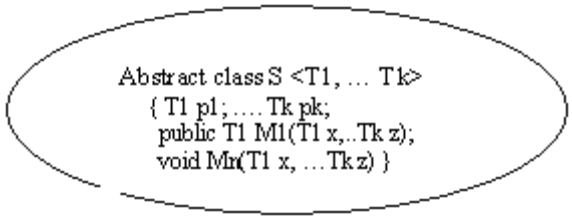

Рис. 22.2.1. 1: Этап проектирования: абстрактный класс с абстрактными типами

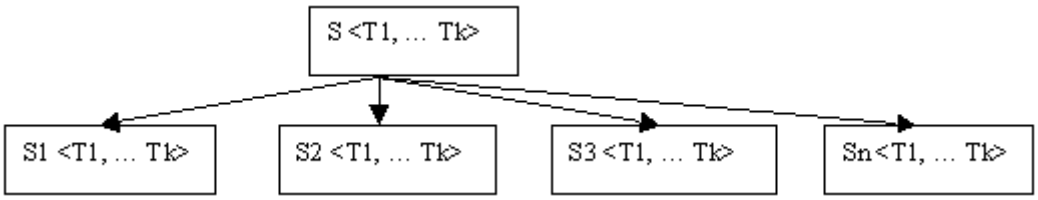

Рис. 22.2.2. 2: Наследование: уточняется представление данных; задается или уточняется реализация методов родителя

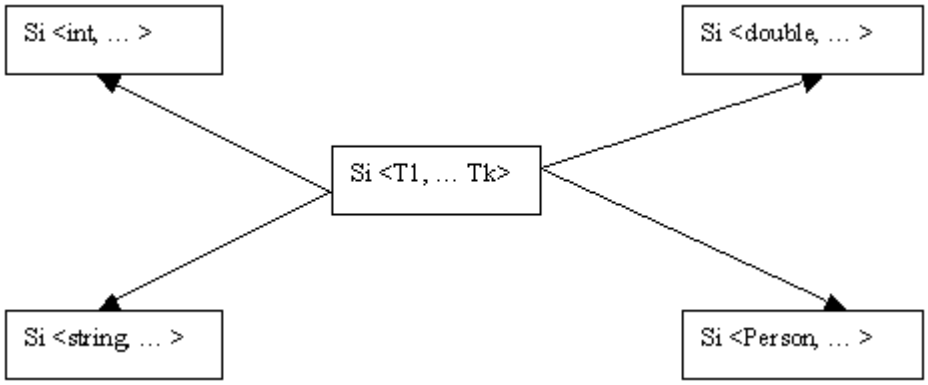

Рис. 22.2.3. 3: Родовое порождение: уточняются типы данных; порождается класс путем подстановки конкретных типов

На этапе спецификации, как правило, создается абстрактный, *универсальный класс*, где задана только сигнатура методов, но не их реализация; где определены имена типов, но не их конкретизация. Здесь же, используя возможности тегов класса, формально или неформально задаются спецификации, описывающие семантику методов класса. Далее в ходе разработки, благодаря механизму *наследования*, появляются потомки абстрактного класса, каждый из которых задает реализацию методов. На следующем этапе, благодаря механизму универсализации, появляются экземпляры *универсального класса*, каждый из которых выполняет операции класса над данными соответствующих типов.

Для наполнения этой схемы реальным содержанием давайте рассмотрим некоторый пример с прохождением всех трех этапов.

Стек. От абстрактного, универсального класса к конкретным версиям

Возьмем классическую задачу определения *стека*. Следуя схеме, определим абстрактный *универсальный класс*, описывающий всевозможные представления *стеков*:

```
/// <summary>
/// Абстрактный класс GenStack<T> задает контейнер с
/// доступом LIFO:
/// Функции:
/// конструктор new: -> GenStack<T>
/// запросы:
/// item: GenStack -> T
/// empty: GenStack -> Boolean
/// процедуры:
/// put: GenStack*T -> GenStack
/// remove: GenStack -> GenStack
/// Аксиомы:
/// remove(put(s,x)) = s
/// item(put(s,x)) = x
/// empty(new)= true
/// empty(put(s,x)) = false
/// </summary>
abstract public class GenStack<T>
{
   /// <summary>
   /// require: not empty();
   /// </summary>
   /// <returns>элемент вершины(последний пришедший)</returns>
   abstract public T item();
   /// <summary>
   /// require: not empty();
```

```
   /// ensure: удален элемент вершины(последний пришедший)
   /// </summary>
   abstract public void remove();
   /// <summary>
   /// require: true; ensure: elem находится в вершине стека
   /// </summary>
   /// <param name="elem"></param>
   abstract public void put(T t);
   /// <summary>
   /// require: true;
   /// </summary>
   /// <returns>true если стек пуст, иначе false </returns>
   abstract public bool empty();
}// class GenStack
```

В приведенном примере программного текста чуть-чуть. Это объявление абстрактного *универсального класса*:

```
abstract public class GenStack<T>
```

и четыре строки с объявлением сигнатуры его методов. Основной текст задает описание спецификации класса и его методов. Заметьте, здесь спецификации заданы достаточно формально с использованием аксиом, характеризующих смысл операций, которые выполняются над *стеком*.

Не хочется вдаваться в математические подробности, отмечу лишь, что, если задать последовательность операций над *стеком*, то аксиомы позволяют точно определить состояние *стека* в результате выполнения этих операций. Как неоднократно отмечалось с первых лекций курса, XML-отчет, построенный по этому проекту, будет содержать в читаемой форме все спецификации нашего класса. Отмечу еще, что все потомки класса должны удовлетворять этим спецификациям, хотя могут добавлять и собственные ограничения.

Наш класс является универсальным - *стек* может хранить элементы любого типа, и конкретизация типа будет производиться в момент создания экземпляра *стека*.

Наш класс является абстрактным - не задана ни реализация методов, ни то, как *стек* будет представлен. Эти вопросы будут решать потомки класса.

Перейдем теперь ко второму этапу и построим потомков класса, каждый из которых задает некоторое представление *стека* и соответствующую этому представлению реализацию методов. Из всех возможных представлений ограничимся двумя. В первом из них *стек* будет представлен *линейной односвязной списковой структурой*. Во втором - он строится на *массиве фиксированного размера*, задавая *стек* ограниченной емкости. Вот как выглядит первый потомок абстрактного класса:

```
/// <summary>
/// Стек, построенный на односвязных элементах списка GenLinkable<T>
/// </summary>
public class OneLinkStack<T> : GenStack<T>
{
   public OneLinkStack()
   {
      last = null;
   }
   GenLinkable<T> last; //ссылка на стек (вершину стека)
   public override T item()
   {
      return (last.Item);
   }//item
```

```
   public override bool empty()
   {
      return (last == null);
   }//empty
   public override void put(T elem)
   {
      GenLinkable<T> newitem = new GenLinkable<T>();
      newitem.Item = elem; newitem.Next = last;
      last = newitem;
   }//put
   public override void remove()
   {
      last = last.Next;
   }//remove
}//class OneLinkStack
```

Посмотрите, что происходит при *наследовании* от *универсального класса*. Во-первых, сам потомок также является *универсальным классом*:

```
public class OneLinkStack<T> : GenStack<T>
```

Во-вторых, если потомок является клиентом некоторого класса, то и этот класс, возможно, также должен быть универсальным, как в нашем случае происходит с классом `GenLinkable<T>`:

```
GenLinkable<T> last; //ссылка на стек (элемент стека)
```

В-третьих, тип `T` встречается в тексте потомка всюду, где речь идет о типе элементов, добавляемых в *стек*, как, например:

```
public override void put(T elem)
```

По ходу дела нам понадобился класс, задающий представление элементов *стека* в списковом представлении. Объявим его:

```
public class GenLinkable<T>
{
   public T Item;
   public GenLinkable<T> Next;
   public GenLinkable()
   { Item = default(T); Next = null; }
}
```

Класс устроен достаточно просто, у него два *поля*: одно для хранения элементов, помещаемых в *стек* и имеющее тип `T`, другое - указатель на следующий элемент. Обратите внимание на *конструктор класса*, в котором для инициализации элемента используется новая конструкция `default(T)`, которая возвращает значение, устанавливаемое по умолчанию для типа `T`.

Второй потомок абстрактного класса реализует *стек* по-другому, используя представление в виде массива. Потомок задает *стек* ограниченной емкости. Емкостью *стека* можно управлять в момент его создания. В ряде ситуаций использование такого *стека* предпочтительнее по соображениям эффективности, поскольку не требует динамического создания элементов. Приведу текст этого класса уже без дополнительных комментариев:

```
public class ArrayUpStack<T> : GenStack<T>
{
   int SizeOfStack;
```

```
   T[] stack;
   int top;
   /// <summary>
   /// конструктор
   /// </summary>
   /// <param name="size">размер стека</param>
   public ArrayUpStack(int size)
   { SizeOfStack = size; stack = new T[SizeOfStack]; top = 0; }
   /// <summary>
   /// require: (top < SizeOfStack)
   /// </summary>
   /// <param name="x"> элемент, помещаемый в стек</param>
   public override void put(T x)
   { stack[top] = x; top++; }

   public override void remove()
   { top--; }
   public override T item()
   { return (stack[top-1]); }
   public override bool empty()
   { return (top == 0); }
}//class ArrayUpStack
```

Созданные в результате *наследования* классы-потомки перестали быть абстрактными, но все еще остаются универсальными. На третьем этапе порождаются конкретные экземпляры потомков - *универсальных классов*, в этот момент и происходит конкретизация типов, и два экземпляра одного *универсального класса* могут работать с данными различных типов. Этот процесс создания экземпляров с подстановкой конкретных типов называют ***родовым порождением экземпляров***. Вот как в тестирующей процедуре создаются экземпляры созданных нами классов:

```
public void TestStacks()
{
   OneLinkStack<int> stack1 = new OneLinkStack<int>();
   OneLinkStack<string> stack2 = new OneLinkStack<string>();
   ArrayUpStack<double> stack3 = new ArrayUpStack
      <double>(10);
   stack1.put(11); stack1.put(22);
   int x1 = stack1.item(), x2 = stack1.item();
   if ((x1 == x2) && (x1 == 22)) Console.WriteLine("OK!");
   stack1.remove(); x2 = stack1.item();
   if ((x1 != x2) && (x2 == 11)) Console.WriteLine("OK!");
   stack1.remove(); x2 = (stack1.empty())? 77 : stack1.item();
   if ((x1 != x2) && (x2 == 77)) Console.WriteLine("OK!");
   stack2.put("first"); stack2.put("second");
   stack2.remove(); string s = stack2.item();
   if (!stack2.empty()) Console.WriteLine(s);
   stack3.put(3.33); stack3.put(Math.Sqrt(Math.PI));
   double res = stack3.item();
   stack3.remove(); res += stack3.item();
   Console.WriteLine("res= {0}", res);
}
```

В трех первых строках этой процедуры порождаются три экземпляра *стеков*. Все они имеют общего родителя - абстрактный *универсальный класс* `GenStack`, но каждый из них работает с данными своего типа и по-разному реализует методы родителя. На рис. 22.3 показаны результаты работы этой процедуры.

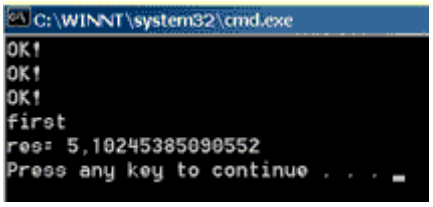

Рис. 22.3. Три разных стека, порожденных абстрактным универсальным классом

Дополним наше рассмотрение еще одним примером работы с вариацией *стеков*, в том числе хранящим объекты класса `Person`:

```
public void TestPerson()
{
   OneLinkStack<int> stack1 = new OneLinkStack<int>();
   OneLinkStack<string> stack2 = new OneLinkStack<string>();
   ArrayUpStack<double> stack3 = new ArrayUpStack
      <double>(10);
   ArrayUpStack<Person> stack4 = new ArrayUpStack<Person>(7);
   stack2.put("Петров"); stack2.put("Васильев");
      stack2.put("Шустов");
   stack1.put(27); stack1.put(45); stack1.put(53);
   stack3.put(21550.5); stack3.put(12345.7);
      stack3.put(32458.8);
   stack4.put(new Person(stack2.item(), stack1.item(),
      stack3.item()));
   stack1.remove(); stack2.remove(); stack3.remove();
   stack4.put(new Person(stack2.item(), stack1.item(),
      stack3.item()));
   stack1.remove(); stack2.remove(); stack3.remove();
   stack4.put(new Person(stack2.item(), stack1.item(),
      stack3.item()));
   Person pers = stack4.item(); pers.PrintPerson();
   stack4.remove(); pers = stack4.item(); pers.PrintPerson();
   stack4.remove(); pers = stack4.item(); pers.PrintPerson();
   stack4.remove(); if (stack4.empty()) Console.WriteLine("OK!");
}
```

Результаты работы этой процедуры приведены на рис. 22.4.

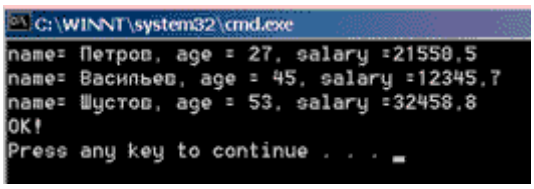

Рис. 22.4. Работа со стеками

Ограниченная универсальность

Хорошо, когда есть свобода. Еще лучше, когда свобода ограничена. Аналогичная ситуация имеет место и с *универсальностью*. *Универсальность* следует ограничивать. На типы *универсального класса*, являющиеся его параметрами, следует накладывать **ограничения**. Звучит парадоксально, но, наложив ограничения на типы, программист получает гораздо большую свободу в работе с объектами этих типов.

Если немного подумать, то это совершенно естественная ситуация. Когда имеет место неограниченная *универсальность*, над объектами типов можно выполнять только те операции, которые допускают все типы, - в C# это эквивалентно операциям, разрешенным над объектами типа object, прародителя всех типов. В нашем предыдущем примере, где речь шла о *свопинге*, над объектами выполнялась

единственная операция присваивания. Поскольку присваивание внутри одного типа разрешено для всех типов, то неограниченная *универсальность* приемлема в такой ситуации. Но что произойдет, если попытаться выполнить сложение элементов, сравнение их или даже простую проверку элементов на равенство? Немедленно возникнет ошибка еще на этапе компиляции. Эти операции не разрешены для всех типов, поэтому в случае компиляции такого проекта ошибка могла бы возникнуть на этапе выполнения, когда вместо формального типа появился бы тип конкретный, не допускающий подобную операцию. Нельзя ради *универсальности* пожертвовать одним из важнейших механизмов C# и Framework .Net - **безопасностью** типов, поддерживаемой статическим контролем типов. Именно поэтому неограниченная *универсальность* существенно ограничена. Ее ограничивает статический контроль типов. Бывают, разумеется, ситуации, когда необходимо на типы наложить ограничения, позволяющие ослабить границы статического контроля. На практике *универсальность* почти всегда ограничивается, что, повторяю, дает большую свободу программисту.

В языке C# допускаются три вида ограничений, накладываемых на родовые параметры.

- ***Ограничение наследования***. Это основной вид ограничений, указывающий, что тип `T` является наследником некоторого класса и ряда интерфейсов. Следовательно, над объектами типа `T` можно выполнять все операции, заданные базовым классом и интерфейсами. Эти операции статический контроль типов будет разрешать и обеспечивать для них интеллектуальную поддержку, показывая список разрешенных операций. *Ограничение наследования* позволяет выполнять над объектами больше операций, чем в случае неограниченной *универсальности*. Синтаксически ограничение выглядит так: `where T: BaseClass, I1, ...Ik`.
- ***Ограничение конструктора***. Это ограничение указывает, что тип `T` имеет *конструктор* без аргументов и, следовательно, позволяет создавать объекты типа `T`. Синтаксически ограничение выглядит так: `where T: new()`.
- ***Ограничение value/reference***. Это ограничение указывает, к значимым или к ссылочным типам относится тип `T`. Для указания значимого типа задается слово `struct`, для ссылочных - `class`. Так что синтаксически этот тип ограничений выглядит так: `where T: struct`.

Возникает законный вопрос: насколько полна предлагаемая система ограничений? Конечно, речь идет о практической полноте, а не о математически строгих определениях. С позиций практики систему хотелось бы дополнить, в первую очередь, введением ограничений операций, указывающим допустимые знаки операций в выражениях над объектами соответствующего типа. Хотелось бы, например, указать, что к объектам типа `T` применима операция сложения `+` или операция сравнения `<`. Позже я покажу, как можно справиться с этой проблемой, но предлагаемое решение довольно сложно. Наличие ограничения операций намного элегантнее решало бы эту проблему.

Синтаксис ограничений

Уточним некоторые синтаксические правила записи ограничений. Если задан *универсальный класс* с *типовыми параметрами* `T1, ... Tn`, то на каждый параметр могут быть наложены ограничения всех типов. Ограничения задаются предложением `where`, начинающимся соответствующим ключевым словом, после которого следует имя параметра, а затем через двоеточие - ограничения первого, второго или третьего типа, разделенных запятыми. Порядок их важен: если присутствует ограничение третьего типа, то оно записывается первым. Заметьте, предложения `where` для разных параметров отделяются лишь пробелами; как правило, они записываются на отдельных строчках. Предложения `where` записываются в конце заголовка класса после имени и

списка его *типовых параметров*, после родительских классов и интерфейсов, если они заданы для *универсального класса*. Вот синтаксически корректные объявления классов с ограничением *универсальности*:

```
public class Father<T1, T2>
{ }
public class Base
{
   public void M1() { }
   public void M2() { }
}
public class Child<T1,T2> :Father<T1,T2>
   where T1:Base,IEnumerable<T1>, new()
   where T2:struct,IComparable<T2>
{       }
```

Класс `Child` с ограниченной *универсальностью* к данным типа `T1` имеет право применять методы `M1` и `M2` базового класса `Base`; так же, как и методы интерфейса `IEnumerable<T1>`, он может создавать объекты типа `T1`, используя *конструктор* по умолчанию. *Фактический тип*, подставляемый вместо формального типа `T2`, должен быть значимым, и объекты этого типа разрешается сравнивать между собой.

Список с возможностью поиска элементов по ключу

Ключевые идеи ограниченной *универсальности*, надеюсь, понятны. Давайте теперь рассмотрим пример построения подобного класса, где можно будет увидеть все детали. Возьмем классическую и саму по себе интересную задачу построения списка с курсором. Как и всякий контейнер данных, список следует сделать универсальным, допускающим хранение данных разного типа. С другой стороны, мы не хотим, чтобы в одном списке происходило смешение типов, - уж если там хранятся персоны, то чисел `int` в нем не должно быть. По этим причинам класс должен быть универсальным, имея в качестве параметра тип `T`, задающий тип хранимых данных. Мы потребуем также, чтобы данные хранились с их ключами. И поскольку не хочется заранее накладывать ограничения на тип ключей - они могут быть строковыми или числовыми, - то тип хранимых ключей будет еще одним параметром нашего класса. Поскольку мы хотим определить над списком операцию поиска по ключу, то нам придется выполнять проверку ключей на равенство, поэтому универсальность типа ключей должна быть ограниченной. Проще всего сделать этот тип наследником стандартного интерфейса `IComparable`.

Чтобы не затемнять ситуацию сложностью списка, рассмотрим достаточно простой односвязный список с курсором. Элементы этого списка будут принадлежать классу `Node`, два *поля* которого будут хранить ключ и сам элемент, а третье *поле* будет задавать указатель на следующий элемент списка. Очевидно, что этот класс должен быть *универсальным классом*. Вот как выглядит текст этого класса:

```
class Node<K, T> where K:IComparable<K>
{
   public Node()
   {
      next = null; key = default(K); item = default( T);
   }
   public K key;
   public T item;
   public Node<K, T> next;
}
```

Класс `Node` имеет два родовых параметра, задающих тип ключей и тип элементов. Ограничение на тип ключей позволяет выполнять их сравнение. В *конструкторе класса* поля инициализируются значениями по умолчанию соответствующего типа.

Рассмотрим теперь организацию односвязного списка. Начнем с того, как устроены его данные:

```
public class OneLinkList<K, T> where K : IComparable<K>
{
   Node<K, T> first, cursor;
}
```

Являясь клиентом *универсального класса* `Node`, наш класс сохраняет родовые параметры клиента и ограничения, накладываемые на них. Два *поля* класса - `first` и `cursor` - задают указатели на первый и текущий элементы списка. Операции над списком связываются с курсором, позволяя перемещать курсор по списку. Рассмотрим вначале набор операций, перемещающих курсор:

```
public void start()
{ cursor = first; }
public void finish()
{
   while (cursor.next != null)
      cursor = cursor.next;
}
public void forth()
{ if (cursor.next != null) cursor = cursor.next; }
```

Операция `start` передвигает курсор к началу списка, `finish` - к концу, а `forth` - к следующему элементу справа от курсора. Операции `finish` и `forth` определены только для непустых списков. Конец списка является барьером, и курсор не переходит через барьер. Нарушая принципы ради краткости текста, я не привожу формальных спецификаций методов, записанных в тегах `<summary>`.

Основной операцией является операция добавления элемента с ключом в список. Возможны различные ее вариации, из которых рассмотрим только одну - новый элемент добавляется за текущим, отмеченным курсором. Вот текст этого метода:

```
public void add(K key, T item)
{
   Node<K, T> newnode = new Node<K, T>();
   if (first == null)
   {
      first = newnode; cursor = newnode;
      newnode.key = key; newnode.item = item;
   }
   else
   {
      newnode.next = cursor.next; cursor.next = newnode;
      newnode.key = key; newnode.item = item;
   }
}
```

Заметьте, аргументы метода имеют соответствующие родовые параметры, чем и обеспечивается универсальный характер списка. При добавлении элемента в список различаются два случая - добавление первого элемента и всех остальных.

Рассмотрим теперь операцию поиска элемента по ключу, реализация которой потребовала ограничения *универсальности* типа ключа `K`:

```
public bool findstart(K key)
{
   Node<K, T> temp = first;
   while (temp != null)
   {
      if (temp.key.CompareTo(key) == 0) {cursor=temp;
         return(true);}
      temp= temp.next;
   }
   return (false);
}
```

Искомые элементы разыскиваются во всем списке. Если элемент найден, то курсор устанавливается на найденном элементе и метод возвращает значение `true`. Если элемента с заданным ключом нет в списке, то позиция курсора не меняется, а метод возвращает значение `false`. В процессе поиска для каждого очередного элемента списка вызывается допускаемый ограничением метод `CompareTo` интерфейса `IComparable`. При отсутствии ограничений *универсальности* вызов этого метода или операции эквивалентности приводил бы к ошибке, обнаруживаемой на этапе компиляции.

Два метода класса являются запросами, позволяющими извлечь ключ и элемент списка, который отмечен курсором:

```
public K Key()
{
   return (cursor.key);
}
public T Item()
{
   return(cursor.item);
}
```

Давайте рассмотрим теперь тестирующую процедуру - клиента нашего списка, демонстрирующую работу со списками, в которых элементы и ключи имеют разные типы:

```
public void TestConstraint()
{
   OneLinkList<int, string> list1 = new OneLinkList
      <int, string>();
   list1.add(33, "thirty three"); list1.add(22, "twenty two");
   if(list1.findstart(33)) Console.WriteLine
      ("33 - найдено!");
   else Console.WriteLine("33 - не найдено!");
   if (list1.findstart(22)) Console.WriteLine ("22 - найдено!");
   else Console.WriteLine("22 - не найдено!");
   if (list1.findstart(44)) Console.WriteLine ("44 - найдено!");
   else Console.WriteLine("44 - не найдено!");
   Person pers1 = new Person("Савлов", 25, 1500);
   Person pers2 = new Person("Павлов", 35, 2100);
   OneLinkList<string, Person> list2 = new OneLinkList
      < string, Person>();
   list2.add("Савл", pers1); list2.add( "Павел", pers2);
   if (list2.findstart("Павел")) Console.WriteLine
      ("Павел - найдено!");
   else Console.WriteLine("Павел - не найдено!");
   if (list2.findstart("Савл")) Console.WriteLine
      ("Савл - найдено!");
   else Console.WriteLine("Савл - не найдено!");
   if (list2.findstart("Иоанн")) Console.WriteLine
```

```
      ("Иоанн - найдено!");
   else Console.WriteLine("Иоанн - не найдено!");
   Person pers3 = new Person("Иванов", 33, 3000);
   list2.add("Иоанн", pers3); list2.start();
   Person pers = list2.Item(); pers.PrintPerson();
   list2.findstart("Иоанн"); pers = list2.Item();
      pers.PrintPerson();
}
```

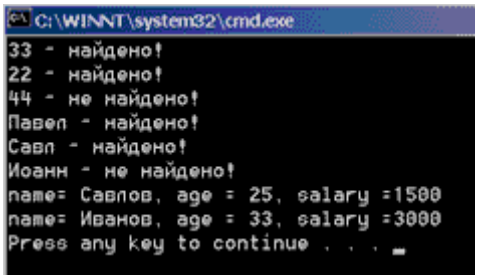

Рис. 22.5. Поиск в списке с ограниченной универсальностью

Обратите внимание на строки, где создаются два списка:

```
OneLinkList<int, string> list1 = new OneLinkList<int, string>();
OneLinkList<string, Person> list2 = new OneLinkList< string, Person>();
```

У списка `list1` ключи имеют тип `int`, у списка `list2` - `string`. Заметьте, оба *фактических типа*, согласно обязательствам, реализуют интерфейс `IComparable`. У первого списка тип элементов - `string`, у второго - `Person`. Все работает прекрасно. Вот результаты вычислений по этой процедуре:

Как справиться с арифметикой

Представьте себе, что мы хотим иметь специализированный вариант нашего списка, элементы которого допускали бы операцию сложения и одно из *полей* которого сохраняло бы сумму всех элементов, добавленных в список. Как задать соответствующее ограничение на класс?

Как уже говорилось, наличие ограничения операции, где можно было бы указать, что над элементами определена операция `+`, решало бы проблему. Но такого типа ограничений нет. Хуже того, нет и интерфейса `INumeric`, аналогичного `IComparable`, определяющего метод сложения `Add`. Так что нам не может помочь и *ограничение наследования*.

Вот один из возможных выходов, предлагаемых в такой ситуации. Стратегия следующая: определим абстрактный *универсальный класс* `Calc` с методами, выполняющими вычисления. Затем создадим конкретизированных потомков этого класса. В классе, задающем список с суммированием, введем *поле* класса `Calc`. При создании экземпляров класса будем передавать *фактические типы* ключа и элементов, а также соответствующий калькулятор, но уже не как тип, а как аргумент *конструктора класса*. Этот калькулятор, согласованный с типом элементов, и будет выполнять нужные вычисления. Давайте приступим к реализации этой стратегии. Начнем с определения класса `Calc`:

```
public abstract class Calc<T>
{
   public abstract T Add(T a, T b);
   public abstract T Sub(T a, T b);
   public abstract T Mult(T a, T b);
   public abstract T Div(T a, T b);
```

```
}
```

Наш абстрактный *универсальный класс* определяет четыре арифметические операции. Давайте построим трех его конкретизированных потомков:

```
public class IntCalc : Calc<int>
{
   public override int Add(int a, int b) { return (a + b);}
   public override int Sub(int a, int b) { return (a - b);}
   public override int Mult(int a, int b) { return (a * b);}
   public override int Div(int a, int b) { return (a / b); }
}
public class DoubleCalc : Calc<double>
{
   public override double Add(double a, double b)
      {return (a + b);}
   public override double Sub(double a, double b)
      {return (a - b);}
   public override double Mult(double a, double b)
      {return (a * b);}
   public override double Div(double a, double b)
      {return (a / b);}
}
public class StringCalc : Calc<string>
{
   public override string Add(string a, string b)
      {return (a + b);}
   public override string Sub(string a, string b)
      {return (a );}
   public override string Mult(string a, string b)
      {return (a );}
   public override string Div(string a, string b)
      {return (a);}
}
```

Здесь определяются три разных калькулятора: один - над целочисленными данными, другой - над данными с плавающей точкой, третий - над строковыми данными. В последнем случае определена, по сути, только операция сложения строк (конкатенации).

Теперь нам нужно ввести изменения в ранее созданный класс `OneLinkList`. Обратите внимание на важный технологический принцип работы с объектными системами. Пусть уже есть нормально работающий класс с нормально работающими клиентами класса. Не следует изменять этот класс. Класс закрыт для изменений. Используйте *наследование* и открывайте класс-потомок, в который и вносите изменения, учитывающие добавляемую специфику класса. Принцип "Закрыт - Открыт" является одним из важнейших принципов построения программных систем в объектном стиле.

В полном соответствии с этим принципом построим класс `SumList` - потомок класса `OneLinkList`. То, что родительский класс является универсальным, ничуть не мешает строить потомка класса, сохраняющего универсальный характер родителя.

```
public class SumList<K, T> : OneLinkList<K, T> where K :
   IComparable<K>

{
   Calc<T> calc;
   T sum;
   public SumList(Calc<T> calc)
   { this.calc = calc; sum = default(T); }
   public new void add(K key, T item)
```

```
    {
       Node<K, T> newnode = new Node<K, T>();
       if (first == null)
       {
          first = newnode; cursor = newnode;
          newnode.key = key; newnode.item = item;
          sum = calc.Add(sum, item);
       }
       else
       {
          newnode.next = cursor.next; cursor.next = newnode;
          newnode.key = key; newnode.item = item;
          sum = calc.Add(sum, item);
       }
    }
    public T Sum()
    {return (sum); }
}//SumList
```

У класса добавилось *поле* `sum`, задающее сумму хранимых элементов, и *поле* `calc` - калькулятор, выполняющий вычисления. Метод `add`, объявленный в классе с модификатором `new`, скрывает родительский метод `add`, задавая собственную реализацию этого метода. Родительский метод можно было бы определить как виртуальный, переопределив его у потомка, но я не стал трогать код родительского класса. К классу добавился еще один запрос, возвращающий значение *поля* `sum`.

Некоторые изменения в уже существующем проекте пришлось-таки сделать, изменив статус доступа у *полей*. А все потому, что в целях экономии текста кода я не стал закрывать *поля* и вводить, как положено, открытые процедуры-свойства для закрытых *полей*.

Проведем теперь эксперименты с новыми вариантами списков, допускающих суммирование элементов:

```
public void TestSum()
{
   SumList<string, int> list1 =
      new SumList<string, int>(new IntCalc());
   list1.add("Петр", 33); list1.add("Павел", 44);
   Console.WriteLine("sum= {0}", list1.Sum());
   SumList<string, double> list2 =
      new SumList<string, double> (new DoubleCalc());
   list2.add("Петр", 33.33); list2.add("Павел", 44.44);
   Console.WriteLine("sum= {0}", list2.Sum());
   SumList<string, string> list3 =
      new SumList<string, string> (new StringCalc());
   list3.add("Мама", " Мама мыла "); list3.add("Маша",
      "Машу мылом!");
   Console.WriteLine("sum= {0}", list3.Sum());
}
```

Обратите внимание на создание списков:

```
   SumList<string, int> list1 =
      new SumList<string, int>(new IntCalc());
   SumList<string, double> list2 =
      new SumList<string, double>(new DoubleCalc());
   SumList<string, string> list3 =
      new SumList<string, string>(new StringCalc());
```

Как видите, конструктору объекта передается калькулятор, согласованный с типами данных, которые хранятся в списке. Результаты вычислений, полученных при работе с этими списками, приведены на рис. 22.6.

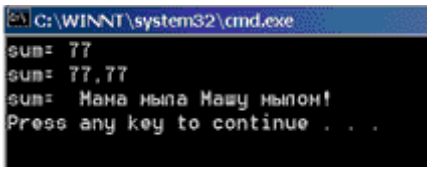

Рис. 22.6. Списки с суммированием

Родовое порождение класса. Предложение using

До сих пор рассматривалась ситуация *родового порождения экземпляров универсального класса*. *Фактические типы* задавались в момент создания экземпляра. Это наглядно показывает преимущества применяемой технологии, поскольку очевидно, что не создается дублирующий код для каждого класса, порожденного *универсальным классом*. И все-таки остается естественный вопрос: можно ли породить класс из *универсального класса* путем подстановки фактических параметров, а потом спокойно использовать этот класс обычным образом? Такая вещь возможна. Это можно сделать не совсем обычным путем - не в программном коде, а в предложении `using`, назначение которого и состоит в выполнении подобных подстановок.

Давайте вернемся к *универсальному классу* `OneLinkStack<T>`, введенному в начале этой лекции, и породим на его основе вполне конкретный класс `IntStack`, заменив формальный параметр `T` фактическим - `int`. Для этого достаточно задать следующее предложение `using`:

```
using IntStack = Generic.OneLinkStack<int>;
```

Вот тест, в котором создаются несколько объектов этого класса:

```
public void TestIntStack()
{
   IntStack stack1 = new IntStack();
   IntStack stack2 = new IntStack();
   IntStack stack3 = new IntStack();
   stack1.put(11); stack1.put(22);
   int x1 = stack1.item(), x2 = stack1.item();
   if ((x1 == x2) && (x1 == 22)) Console.WriteLine("OK!");
   stack1.remove(); x2 = stack1.item();
   if ((x1 != x2) && (x2 == 11)) Console.WriteLine("OK!");
   stack1.remove(); x2 = (stack1.empty()) ? 77 :
      stack1.item();
   if ((x1 != x2) && (x2 == 77)) Console.WriteLine("OK!");
   stack2.put(55); stack2.put(66);
   stack2.remove(); int s = stack2.item();
   if (!stack2.empty()) Console.WriteLine(s);
   stack3.put(333); stack3.put((int)Math.Sqrt(Math.PI));
   int res = stack3.item();
   stack3.remove(); res += stack3.item();
   Console.WriteLine("res= {0}", res);
}
```

Все работает заданным образом, можете поверить.

Универсальность и специальные случаи классов

Универсальность - это механизм, воздействующий на все элементы языка. Поэтому он применим ко всем частным случаям классов C# .

Универсальные структуры

Так же, как и обычный класс, структура может иметь родовые параметры. Синтаксис объявления, ограниченная *универсальность*, другие детали *универсальности* естественным образом распространяются на структуры. Вот типичный пример:

```
public struct Point<T>
{
   T x, y;//координаты точки, тип которых задан параметром
   // другие свойства и методы структуры
}
```

Универсальные интерфейсы

Интерфейсы чаще всего следует делать *универсальными*, предоставляя большую гибкость для позднейших этапов создания системы. Возможно, вы заметили применение в наших примерах *универсальных интерфейсов* библиотеки FCL - `IComparable<T>` и других. Введение *универсальности*, в первую очередь, сказалось на библиотеке FCL - внутренних классов, определяющих поведение системы. В частности, для большинства интерфейсов появились универсальные двойники с параметрами. Если бы в наших примерах мы использовали не *универсальный интерфейс*, а обычный, то потеряли бы в эффективности, поскольку сравнение объектов потребовало бы создание временных объектов типа `object`, выполнения операций `boxing` и `unboxing`.

Универсальные делегаты

Делегаты также могут иметь родовые параметры. Чаще встречается ситуация, когда делегат объявляется в *универсальном классе* и использует в своем объявлении параметры *универсального класса*. Давайте рассмотрим ситуацию с делегатами более подробно. Вот объявление *универсального класса*, не очень удачно названного `Delegate`, в котором объявляется функциональный тип - `delegate`:

```
class Delegate<T>
{
   public delegate T Del(T a, T b);
}
```

Как видите, тип аргументов и возвращаемого значения в сигнатуре функционального типа определяется классом `Delegate`.

Добавим в класс функцию высшего порядка `FunAr`, одним из аргументов которой будет функция типа `Del`, заданного делегатом. Эта функция будет применяться к элементам массива, передаваемого также функции `FunAr`. Приведу описание:

```
public T FunAr(T[] arr, T a0, Del f)
{
   T temp = a0;
   for(int i =0; i<arr.Length; i++)
   {
      temp = f(temp, arr[i]);
   }
   return (temp);
}
```

Эта универсальная функция с успехом может применяться для вычисления сумм, произведения, минимума и других подобных характеристик массива.

Рассмотрим теперь клиентский класс `Testing`, в котором определен набор функций:

```
public int max2(int a, int b)
      { return (a > b) ? a : b; }
public double min2(double a, double b)
      { return (a < b) ? a : b; }
public string sum2(string a, string b)
      { return a + b; }
public float prod2(float a, float b)
      { return a * b; }
```

Хотя все функции имеют разные типы, все они соответствуют определению класса `Del` - имеют два аргумента одного типа и возвращают результат того же типа. Посмотрим, как они применяются в тестирующем методе класса `Testing`:

```
public void TestFun()
{
   int[] ar1 = { 3, 5, 7, 9 };
   double[] ar2 = { 3.5, 5.7, 7.9 };
   string[] ar3 = { "Мама ", "мыла ", "Машу ", "мылом." };
   float[] ar4 = { 5f, 7f, 9f, 11f };
   Delegate<int> d1 = new Delegate<int>();
   Delegate<int>.Del del1;
   del1= this.max2;
   int max = d1.FunAr(ar1, ar1[0], del1);
   Console.WriteLine("max= {0}", max);
   Delegate<double> d2 = new Delegate<double>();
   Delegate<double>.Del del2;
   del2 = this.min2;
   double min = d2.FunAr(ar2, ar2[0], del2);
   Console.WriteLine("min= {0}", min);
   Delegate<string> d3 = new Delegate<string>();
   Delegate<string>.Del del3;
   del3 = this.sum2;
   string sum = d3.FunAr(ar3, "", del3);
   Console.WriteLine("concat= {0}", sum);
   Delegate<float> d4 = new Delegate<float>();
   Delegate<float>.Del del4;
   del4 = this.prod2;
   float prod = d4.FunAr(ar4, 1f, del4);
   Console.WriteLine("prod= {0}", prod);
}
```

Обратите внимание на объявление экземпляра делегата:

```
Delegate<int>.Del del1;
```

В момент объявления задается *фактический тип*, и сигнатура экземпляра становится конкретизированной. Теперь экземпляр можно создать и связать с конкретной функцией. В C# 2.0 это делается проще и естественнее, чем ранее, - непосредственным присваиванием:

```
del1= this.max2;
```

При выполнении этого присваивания производятся довольно сложные действия - проверяется соответствие сигнатуры функции в правой части и экземпляра делегата, в

случае успеха создается новый экземпляр делегата, который и связывается с функцией.

Покажем, что и сам функциональный тип-делегат можно объявлять с родовыми параметрами. Вот пример такого объявления:

```
public delegate T FunTwoArg<T>(T a, T b);
```

Добавим в наш тестовый пример код, демонстрирующий работу с этим делегатом:

```
FunTwoArg<int> mydel;
         mydel = max2;
         max = mydel(17, 21);
         Console.WriteLine("max= {0}", max);
```

Вот как выглядят результаты работы тестового примера:

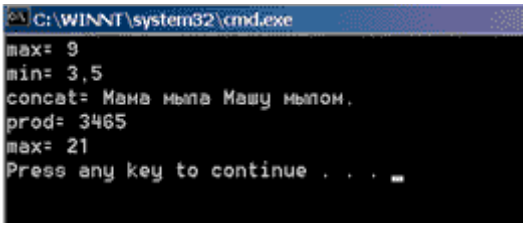

Рис. 22.7. Результаты работы с универсальными делегатами

Универсальные делегаты с успехом используются при определении событий. В частности, класс `EventHandler`, применяемый для всех событий, не имеющих собственных аргументов, теперь дополнен универсальным аналогом, определенным следующим образом:

```
public void delegate EventHandler<T> (object sender, T args)
               where T:EventArgs
```

Этот делегат может применяться и для событий с собственными аргументами, поскольку вместо параметра `T` может быть подставлен конкретный тип - потомок класса `EventArgs`, дополненный нужными аргументами.

Framework .Net и универсальность

Универсальность принадлежит к основным механизмам языка. Ее введение в язык C# не могло не сказаться на всех его основных свойствах. Как уже говорилось, классы и все частные случаи стали обладать этим свойством. Введение *универсальности* не должно было ухудшить уже достигнутые свойства языка - статический контроль типов, динамическое связывание и полиморфизм. Не должна была пострадать и эффективность выполнения программ, использующих *универсальные классы*.

Решение этих задач потребовало введения *универсальности* не только в язык C#, но и поддержки на уровне каркаса Framework .Net и языка IL, включающем теперь параметризованные типы. *Универсальный класс* C# не является *шаблоном*, на основе которого строится конкретизированный класс, компилируемый далее в класс (тип) IL. Компилятору языка C# нет необходимости создавать классы для каждой конкретизации типов *универсального класса*. Вместо этого происходит компиляция *универсального класса* C# в параметризованный тип IL. Когда же CLR занимается исполнением управляемого кода, то вся необходимая информация о конкретных типах извлекается из метаданных, сопровождающих объекты.

При этом дублирования кода не происходит и на уровне JIT-компиляторов, которые, однажды сгенерировав код для конкретного типа, сохраняют ссылку на этот участок кода и передают ее, когда такой код понадобится вторично. Это справедливо как для ссылочных, так и значимых типов.

Естественно, что *универсальность* потребовала введения в библиотеку FCL соответствующих классов, интерфейсов, делегатов и методов классов, обладающих этим свойством.

Так, например, в класс System.Array добавлен ряд универсальных статических методов. Вот один из них:

```
public static int BinarySearch<T>(T[] array, T value);
```

В таблице 22.1 показаны некоторые *универсальные классы* и интерфейсы библиотеки FCL 2.0 из пространства имен `System.Collections.Generic` и их аналоги из пространства `System.Collections`.

Таблица 22.1. Соответствие между универсальными классами и их обычными двойниками

Универсальный класс	**Обычный класс**	***Универсальный интерфейс***	**Обычный интерфейс**
Comparer<T>	Comparer	ICollection<T>	ICollection
Dictionary<K,T>	HashTable	IComparable<T>	IComparable
LinkedList<T>	----	IDictionary<K,T>	IDictionary
List<T>	ArrayList	IEnumerable<T>	IEnumerable
Queue<T>	Queue	IEnumerator<T>	IEnumerator
SortedDictionary<K,T>	SortedList	IList<T>	IList
Stack<T>	Stack		

Сериализация и универсализация также согласуются друг с другом, так что можно иметь *универсальный класс*, для которого задан атрибут сериализации.

23. Лекция: Отладка и обработка исключительных ситуаций

Корректность и устойчивость программных систем

Корректность и *устойчивость* - два основных качества программной системы, без которых все остальные ее достоинства не имеют особого смысла. Понятие *корректности* программной системы имеет смысл только тогда, когда задана ее *спецификация*. В зависимости от того, как формализуется *спецификация*, уточняется понятие *корректности*.

В лекции 9 введено строгое понятие *корректности* метода по отношению к его *спецификациям*, заданным в виде предусловия и постусловия метода.

Корректность - это способность программной системы работать в строгом соответствии со своей *спецификацией*. ***Отладка*** - процесс, направленный на достижение *корректности*.

Во время работы системы могут возникать ситуации, выходящие за пределы, предусмотренные *спецификацией*. Такие **ситуации** называются *исключительными*. ***Устойчивость*** - это способность программной системы должным образом реагировать на *исключительные ситуации*. *Обработка* ***исключительных ситуаций*** - процесс, направленный на достижение *устойчивости*.

Почему так трудно создавать корректные и *устойчивые* программные системы? Все дело в сложности разрабатываемых систем. Когда в 60-х годах прошлого века фирмой IBM создавалась операционная система OS-360, то на ее создание потребовалось 5000 человеко-лет, и проект по сложности сравнивался с проектом высадки первого человека на Луну. Сложность нынешних сетевых операционных систем, систем управления хранилищами данных, прикладных систем программирования на порядки превосходит сложность OS-360, так что, несмотря на прогресс, достигнутый в области технологии программирования, проблемы, стоящие перед разработчиками, не стали проще.

Жизненный цикл программной системы

Под "*жизненным циклом*" понимается период от замысла программного продукта до его "кончины". Обычно рассматриваются следующие фазы этого процесса:

Проектирование <-> Разработка <-> Развертывание и Сопровождение

Все это называется циклом, поскольку после каждой фазы возможен возврат к предыдущим этапам. В объектной технологии этот процесс является бесшовным, все этапы которого тесно переплетены. Не следует рассматривать его как однонаправленный - от проектирования к сопровождению. Чаще всего, ситуация обратная: уже существующая реализация системы, прошедшая сопровождение, и существующие библиотеки компонентов оказывают решающее влияние на то, какой будет новая система, каковы будут ее *спецификации*.

Вот некоторые типовые правила, характерные для процесса разработки ПО:

- Уделяйте этапу проектирования самое пристальное внимание. Успех дела во многом определяется первым этапом. Нет смысла торопиться с переходом на последующие этапы, пока не составлены ясные и четкие *спецификации*. Ошибки этого этапа - самые дорогие и трудно исправляемые.
- Помните о тех, для кого разрабатывается программный продукт. Идите "в люди", чтобы понять, что нужно делать. Вместе с тем, не следует полностью полагаться

на пользователей - их опыт консервативен, новые идеи могут часто приходить от разработчиков, а не от пользователей.

- Разработка не начинается "с нуля". Только используя уже готовые компоненты, можно своевременно создать новую систему. Работая над проектом, думайте о будущем, Не создавайте компоненты, допускающие их *повторное использование* в других проектах.
- Создавайте как можно раньше прототип свой системы и передавайте его пользователям в опытную эксплуатацию. Это поможет устранить множество недостатков и ошибок в заключительной версии программного продукта.
- Какие бы хорошие *спецификации* не были написаны, какими бы хорошими технологиями и инструментами не пользовались разработчики, какими бы профессионалами они ни были - этого еще не достаточно для успеха дела. Необходимым условием является управление проектом, наличие специальных средств управления. Но и этого не достаточно. Третьим важным фактором является существование команды. Коллектив разработчиков должен представлять собой единый коллектив. Умение работать в команде так же важно, как и профессиональные навыки разработчика.

Три закона программотехники

Первый закон (закон для разработчика)

Корректность **системы - недостижима. Каждая последняя найденная ошибка является предпоследней.**

Этот *закон* отражает сложность нетривиальных систем. Разработчик всегда должен быть готов к тому, что в работающей системе имеются ситуации, в которых система работает не в точном соответствии со своей *спецификацией*, так что от него может требоваться очередное изменение либо системы, либо ее *спецификации*.

Второй закон (закон для пользователя)

Не бывает некорректных систем. Каждая появляющаяся ошибка при эксплуатации системы - это следствие незнания *спецификации* **системы.**

Есть два объяснения справедливости второго *закона*. Несерьезное объяснение состоит в том, что любая система, что бы она ни делала, при любом постусловии *корректна* по отношению к предусловию `False`, поскольку невозможно подобрать ни один набор входных данных, удовлетворяющих этому предусловию. Так что все системы *корректны*, если задать `False` в качестве их предусловия. Если вам пришлось столкнуться с системой, предусловие которой близко к `False`, то лучшее, что можно сделать, это отложить ее в сторону и найти другую.

Более поучительна реальная ситуация, подтверждающая второй *закон* и рассказанная мне в былые годы Виталием Кауфманом - специалистом по тестированию трансляторов. В одной серьезной организации была разработана серьезная прикладная система, имеющая для них большое значение. К сожалению, при ее эксплуатации сплошь и рядом возникали ошибки, из-за которых организация вынуждена была отказаться от использования системы. Разработчики обратились к нему за помощью. Он, исследуя систему, не внес в нее ни строчки кода. Единственное, что он сделал, это описал точную *спецификацию* системы, благодаря чему стала возможной нормальная эксплуатация.

Обратите внимание на философию, характерную для этих законов: при возникновении ошибки разработчик и пользователь должны винить себя, а не кивать друг на друга. Так что часто встречающиеся фразы "Ох уж эта фирма Чейтософт - вечно у них ошибки!" характеризует, мягко говоря, непрофессионализм говорящего.

Третий закон (закон чечако)

Если *спецификацию* **можно нарушить, - она будет нарушена. Новичок (чечако) способен "подвесить" любую систему.**

Неквалифицированный пользователь в любом контексте всегда способен выбрать наименее подходящее действие, явно не удовлетворяющее *спецификации*, которая ориентирована на "разумное" поведение пользователей. Полезным практическим следствием этого *закона* является привлечение к этапу тестирования системы неквалифицированного пользователя - "человека с улицы".

Отладка

Что должно делать для создания *корректного* и *устойчивого* программного продукта? Как минимум, необходимо:

- создать *надежный код*, *корректность* которого предусматривается с самого начала;
- отладить этот код;
- предусмотреть в нем *обработку исключительных ситуаций*.

Создание надежного кода

Большинство вопросов, затрагиваемых в этой лекции, в том числе и проблемы создания *надежного кода*, заслуживают отдельного и глубокого рассмотрения. К сожалению, придется ограничиться лишь высказыванием ряда тезисов.

Для повышения надежности нужно уменьшить сложность системы, и главное в этом процессе - это *повторное использование*. В идеале большая часть системы должна быть собрана из уже готовых компонентов. Объектная технология проектирования вносит свой вклад в повышение надежности кода. **Наследование и универсализация** позволяют, не изменяя уже существующие классы, создать новые классы, новые типы данных, придающие проектируемой системе новые свойства при минимальных добавлениях нового кода. *Статический контроль типов* позволяет выявить многие ошибки еще на этапе компиляции. *Динамическое связывание* и *полиморфизм* позволяют автоматически включать объекты классов-потомков в уже существующие схемы работы - методы родителя могут вызывать методы потомков, ничего не зная о появлении этих новых потомков. *Автоматическая сборка мусора* позволяет снять с разработчика обязанности управления освобождением памяти и предотвратить появление крайне неприятных и опасных ошибок, связанных с некорректным удалением объектов.

Крайне важную роль в создании *надежного кода* играют *спецификации* **методов** класса, класса в целом, системы классов. *Спецификации* являются частью документации, встроенной в проект, и вообще важной его частью. Их существование облегчает не только создание *корректного* кода, соответствующего *спецификации*, но и создание системы тестов, проверяющих *корректность* кода. Нужно сказать, что существуют специальные инструментальные средства, поддерживающие автоматическое создание тестов на основе *спецификаций*. Незаменима роль *спецификаций* на этапе сопровождения и *повторного использования* **компонентов**. Невозможно повторно использовать компонент, если у него нет ясной и полной *спецификации*.

Искусство отладки

Нужно стараться создавать *надежный код*. Но без *отладки* пока обойтись невозможно. Роль *тестеров* в современном процессе разработки ПО велика.

Отладка - это некоторый детективный процесс. Программа, в которую внесены изменения, подозревается в том, что она работает некорректно. Презумпция невиновности здесь не применима. Если удается предъявить тест, на котором программа дает неверный результат, то доказано, что подозрения верны. Втайне мы всегда надеемся, что программа работает правильно. Но цель тестирования другая - попытаться опровергнуть это предположение. *Отладка* может доказать некорректность программы, но она не может доказать ее правильность. *Отладка* не гарантирует *корректности программы*, даже если все тесты пройдены успешно. Искусное тестирование создает высокую степень уверенности в *корректности программы*.

Часть ошибок программы ловится автоматически еще на этапе компиляции. Сюда относятся все синтаксические ошибки, ошибки несоответствия типов и некоторые другие. Это простые ошибки и их исправление, как правило, не вызывает трудностей. В *отладке* нуждается синтаксически *корректная программа*, результаты вычислений которой получены, но не соответствуют требуемым *спецификациям*. Чаще всего еще не отлаженная программа на одних исходных данных работает правильно, на других - дает ошибочный результат. Искусство *отладки* состоит в том, чтобы обнаружить все ситуации, в которых работа программы приводит к ошибочным вычислениям.

Как и во всякой детективной деятельности, в ходе *отладки* необходим сбор улик, для чего применяется *две группы средств*. Первая позволяет контролировать ход вычислительного процесса: порядок следования операторов в методах, порядок вызова самих методов, условия окончания циклов, правильность переходов. Вторая отслеживает изменение состояния вычислительного процесса (значения свойств объектов) в процессе выполнения.

Есть и другая классификация. Средства, используемые при *отладке*, можно разделить на **инструментарий**, предоставляемый средой разработки Visual Studio .Net, и *программные средства*, предоставляемые языком и специальными классами библиотеки FCL. Начнем рассмотрение с программных средств.

Отладочная печать и условная компиляция

Одним из основных средств *отладки* является *отладочная печать*, позволяющая получить данные о ходе и состоянии процесса вычислений. Обычно разрабатываются специальные отладочные методы, вызываемые в критических точках программы - на входе и выходе программных модулей, на входе и выходе циклов и так далее. Искусство *отладки* в том и состоит, чтобы получить нужную информацию о прячущихся ошибках, проявляющихся, возможно, только в редких ситуациях.

Хотелось бы иметь легкий механизм управления отладочными методами, позволяющий включать при необходимости те или иные инструменты. Для этого можно воспользоваться *механизмом условной компиляции*, встроенным в язык C#. Этот механизм состоит из двух частей. К проекту, точнее, к конфигурации проекта можно добавить специальные *константы условной компиляции*. Вызов отладочного метода может быть сделан условным. Если соответствующая *константа компиляции* определена, то происходит компиляция вызова метода и он будет вызываться при выполнении проекта. Если же константа не определена (выключена), то вызов метода даже не будет компилироваться и никаких динамических проверок - вызывать метод или нет - делаться не будет.

Как задавать константы компиляции? Напомню, что проекты в Visual Studio существуют в нескольких конфигурациях. В ходе работы с проектом можно легко переключаться с одной конфигурации на другую, после чего она становится активной, можно изменять настройки конфигурации, можно создать собственные конфигурации проекта. По умолчанию проект создается в двух конфигурациях - `Debug` и `Release`, первая из которых предназначена для *отладки*, вторая - для окончательных вычислений. Первая не предполагает оптимизации и в ней определены две *константы условной компиляции*

- DEBUG и TRACE, во второй - определена только константа TRACE. Отладочная версия может содержать вызовы, зависящие от константы DEBUG, которые будут отсутствовать в финальной версии. Используя страницу свойств, к конфигурации проекта можно добавлять новые константы компиляции.

В лекции 2 рассказывалось, как добраться до страницы свойств проекта. Взгляните еще раз на рис. 2.3 этой лекции, где показана страница свойств, и обратите внимание на первую строчку, содержащую список *констант условной компиляции* активной конфигурации (в данном случае - Debug). К этому списку можно добавлять собственные константы.

Можно также задавать *константы условной компиляции* в начале модуля проекта вперемешку с предложениями using. Предложение define позволяет определить новую константу:

```
#define COMPLEX
```

Как используются *константы условной компиляции*? В языке C++, где имеется подобный механизм, определен специальный препроцессорный IF-оператор, анализирующий, задана константа или нет. В языке C# используется вместо этого гораздо более мощный механизм. Как известно, методы C# обладают набором атрибутов, придающих методу разные свойства. Среди встроенных атрибутов языка есть атрибут Conditional, аргументом которого является строка, задающая имя константы:

```
[Conditional ("COMPLEX")] public void ComplexMethod () {...}
```

Если *константа условной компиляции* COMPLEX определена для активной конфигурации проекта, то произойдет компиляция вызова метода ComplexMethod, когда он встретится в тексте программы. Если же такая константа отсутствует в конфигурации, то вызов метода игнорируется.

На методы, для которых возможно задание атрибута Conditional, накладывается ряд ограничений. Метод не должен быть:

- функцией, возвращающей значение;
- методом интерфейса;
- методом с модификатором override. Возможно его задание для virtual-метода. В этом случае атрибут наследуется методами потомков.

Атрибут Conditional, обычно с аргументом DEBUG, сопровождает модули, написанные для целей *отладки*. Но использование этого атрибута не ограничивается интересами *отладки*. Зачастую проект может использоваться в нескольких вариантах, например, в облегченном и более сложном. Методы, вызываемые в сложных ситуациях, например, ComplexMethod, имеющий атрибут условной компиляции, будут вызываться только в той конфигурации, где определена константа COMPLEX.

Приведу пример работы с отладочными методами. Рассмотрим класс, в котором определены три метода, используемые при *отладке*:

```
public class DebugPrint
{
   [Conditional("DEBUG")] static public void
      PrintEntry(string name)
   {
      Console.WriteLine("Начал работать метод " + name);
   }
```

```
   [Conditional("DEBUG")] static public void
      PrintExit(string name)
   {
      Console.WriteLine("Закончил работать метод " + name);
   }
   [Conditional("DEBUG")]
      static public void PrintObject(object obj, string name)
   {
      Console.WriteLine("Объект {0}: {1}", name,
         obj.ToString());
   }
}
```

В классе `Testing` определено поле класса:

```
int state = 1;
```

и группа методов:

```
public void TestDebugPrint()
{
   DebugPrint.PrintEntry("Testing.TestDebugPrint");
   PubMethod();
   DebugPrint.PrintObject(state, "Testing.state");
   DebugPrint.PrintExit("Testing.TestDebugPrint");
}
void InMethod1()
{
   DebugPrint.PrintEntry("InMethod1");
   // body
   DebugPrint.PrintExit("InMethod1");
}
void InMethod2()
{
   DebugPrint.PrintEntry("InMethod2");
   // body
   DebugPrint.PrintExit("InMethod2");
}
public void PubMethod()
{
   DebugPrint.PrintEntry("PubMethod");
   InMethod1();
   state++;
   InMethod2();
   DebugPrint.PrintExit("PubMethod");
}
```

Этот пример демонстрирует трассировку хода вычислений, для чего в начало и конец каждого метода вставлены вызовы отладочных методов, снабжающие нас информацией о ходе вычислений. Такая трассировка иногда бывает крайне полезной на этапе *отладки*, но, естественно, она не должна присутствовать в финальной версии проекта. Взгляните на результаты, полученные при вызове метода `TestDebugPrint` в конфигурации `Debug`.

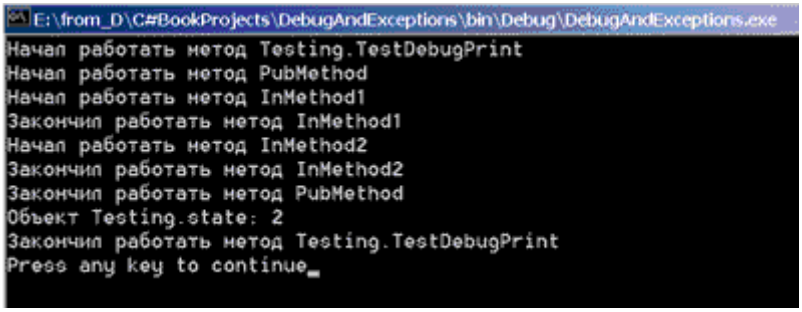

Рис. 23.1. Трассировка вычислений в процессе отладки

При переходе к конфигурации Release отладочная информация появляться не будет.

Классы Debug и Trace

Атрибут условной компиляции `Conditional` характеризует метод, но не отдельный оператор. Иногда хотелось бы иметь условный оператор печати, не создавая специального метода, как это было сделано в предыдущем примере. Такую возможность и многие другие полезные свойства предоставляют классы `Debug` и `Trace`.

Классы `Debug` и `Trace` - это классы-двойники. Оба они находятся в пространстве имен Diagnostics, имеют идентичный набор статических свойств и методов с идентичной семантикой. В чем же разница? Методы класса `Debug` имеют атрибут условной компиляции с константой `DEBUG`, действуют только в Debug-конфигурации проекта и игнорируются в Release-конфигурации. Методы класса `Trace` включают два атрибута `Conditional` с константами `DEBUG` и `TRACE` и действуют в обеих конфигурациях.

Одна из основных групп методов этих классов - *методы печати данных*: `Write`, `WriteIf`, `WriteLine`, `WriteLineIf`. Методы перегружены, в простейшем случае позволяют выводить некоторое сообщение. Методы со словом `If` могут сделать печать условной, задавая условие печати в качестве первого аргумента метода, что иногда крайне полезно. Методы со словом `Line` дают возможность дополнять сообщение символом перехода на новую строку.

По умолчанию методы обоих классов направляют вывод в окно `Output`. Однако это не всегда целесообразно, особенно для Release-конфигурации. Замечательным свойством методов классов `Debug` и `Trace` является то, что они могут иметь много "слушателей", направляя вывод каждому из них. Свойство `Listeners` этих классов возвращает разделяемую обоими классами коллекцию слушателей - `TraceListenerCollection`. Как и всякая коллекция, она имеет ряд методов для добавления новых слушателей: `Add`, `AddRange`, `Insert` - и возможность удаления слушателей: `Clear`, `Remove`, `RemoveAt` и другие методы. Объекты этой коллекции в качестве предка имеют абстрактный класс `TraceListener`. Библиотека FCL включает три неабстрактных потомка этого класса:

- `DefaultTraceListener` - слушатель этого класса, добавляется в коллекцию по умолчанию, направляет вывод, поступающий при вызове методов классов `Debug` и `Trace`, в окно Output;
- `EventLogTraceListener` - посылает сообщения в журнал событий Windows;
- `TextWriterTraceListener` - направляет сообщения объектам класса `TextWriter` или `Stream`; обычно один из объектов этого класса направляет вывод на консоль, другой - в файл.

Можно и самому создать потомка абстрактного класса, предложив, например, XML-слушателя, направляющего вывод в соответствующий XML-документ. Как видите, система управления выводом очень гибкая, позволяющая получать и сохранять информацию о ходе вычислений в самых разных местах.

Помимо свойства `Listeners` и *методов печати*, классы `Debug` и `Trace` имеют и другие важные методы и свойства:

- `Assert` и `Fail`, проверяющие *корректность* хода вычислений - о них мы поговорим особо;
- `Flush` - метод, отправляющий содержание буфера слушателю (в файл, на консоль и так далее). Следует помнить, что данные буферизуются, поэтому применение метода `Flush` зачастую необходимо, иначе метод может завершиться, а данные останутся в буфере;
- `AutoFlush` - булево свойство, указывающее, следует ли после каждой операции записи данные из буфера направлять в соответствующий канал. По умолчанию свойство выключено, и происходит только буферизация данных;
- `Close` - метод, опустошающий буфера и закрывающий всех слушателей, после чего им нельзя направлять сообщения.

У классов есть и другие свойства и методы, позволяющие, например, заниматься структурированием текста сообщений.

Рассмотрим пример работы, в котором отладочная информация направляется в разные каналы - окно вывода, консоль, файл:

```
public void Optima()
{
   double x, y=1;
   x= y - 2*Math.Sin(y);
   FileStream f = new FileStream("Debuginfo.txt",
      FileMode.Create, FileAccess.Write);
   TextWriterTraceListener writer1 =
      new TextWriterTraceListener(f);
   TextWriterTraceListener writer2 =
      new TextWriterTraceListener(System.Console.Out);
   Trace.Listeners.Add( writer1);
   Debug.Listeners.Add( writer2);
   Debug.WriteLine("Число слушателей:" +
      Debug.Listeners.Count);
   Debug.WriteLine("автоматический вывод из буфера:"+
      Trace.AutoFlush);
   Trace.WriteLineIf(x<0, "Trace: " + "x= " + x.ToString()
       + " y = " + y);
   Debug.WriteLine("Debug: " + "x= " + x.ToString() +
      " y = " + y);
   Trace.Flush();
   f.Close();
}
```

В коллекцию слушателей вывода к слушателю по умолчанию добавляются еще два слушателя класса `TextWriterTraceListener`. Заметьте, что хотя они добавляются методами разных классов `Debug` и `Trace`, попадают они в одну коллекцию. Как и обещано, один из этих слушателей направляет вывод в файл, другой на консоль. На рис. 23.2 на фоне окна кода показаны три канала вывода - окно Output, консоль, файл - содержащие одну и ту же информацию.

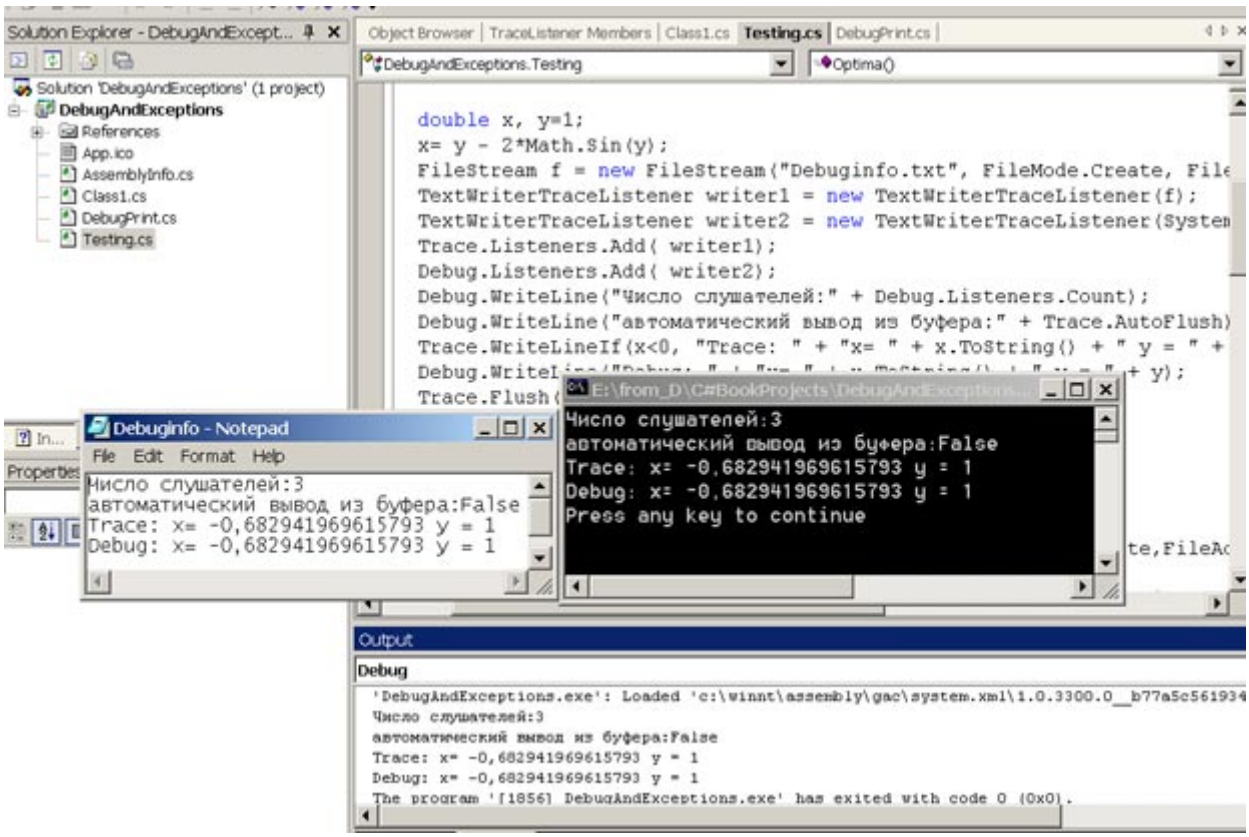

Рис. 23.2. Три канала вывода

Метод Флойда и утверждения Assert

Лет двадцать назад большие надежды возлагались на формальные *методы доказательства правильности программ*, позволяющие доказывать *корректность программ* аналогично доказательству теорем. Реальные успехи формальных доказательств невелики. Построение такого доказательства не проще написания *корректной программы*, а ошибки столь же возможны и часты, как и ошибки программирования. Тем не менее, эти методы оказали серьезное влияние на культуру проектирования *корректных программ*, появление в практике программирования понятий предусловия и постусловия, инвариантов и других важных понятий.

Одним из *методов доказательства правильности программ* был *метод Флойда*, при котором программа разбивалась на участки, окаймленные утверждениями - булевскими выражениями (предикатами). Истинность начального предиката должна была следовать из входных данных программы. Затем для каждого участка доказывалось, что из истинности предиката, стоящего в начале участка, после завершения выполнения соответствующего участка программы гарантируется истинность следующего утверждения - предиката в конце участка. Конечный предикат описывал постусловие программы.

Схема Флойда используется на практике, по крайней мере, программистами, имеющими вкус к строгим методам доказательства. Утверждения становятся частью программного текста. Само доказательство может и не проводиться: чаще всего у программиста есть уверенность в справедливости расставленных утверждений и убежденность, что при желании он мог бы провести и строгое доказательство. В C# эта схема поддерживается тем, что классы `Debug` и `Trace` имеют метод `Assert`, аргументом которого является утверждение. Что происходит, когда вычисление достигает соответствующей точки и вызывается метод `Assert`? Если истинно булево выражение в `Assert`, то вычисления продолжаются, не оказывая никакого влияния на нормальный ход вычислений. Если оно ложно, то *корректность* вычислений под сомнением, их выполнение

приостанавливается и появляется окно с уведомлением о произошедшем событии, что показано на рис. 23.3.

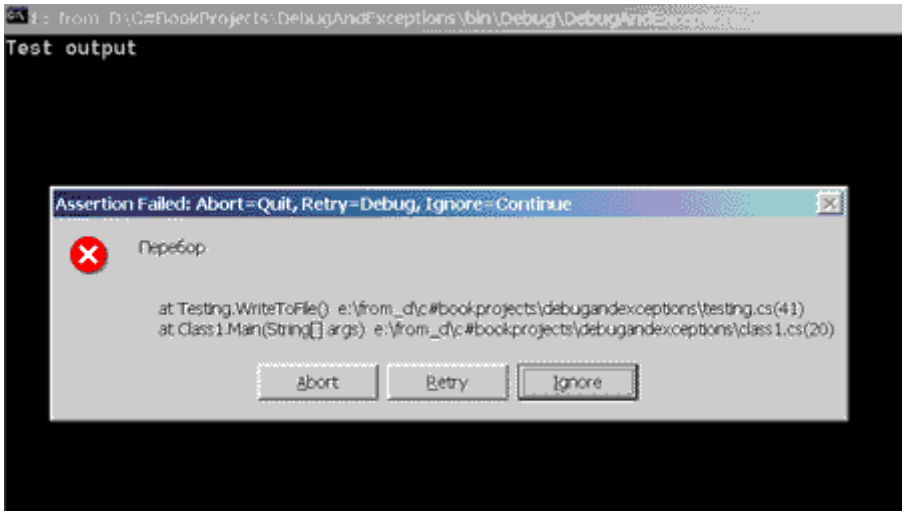

Рис. 23.3. Нарушение утверждения Assert

В этой ситуации у программиста есть несколько возможностей:

- прервать выполнение, нажав кнопку Abort;
- перейти в режим отладки (Retry);
- продолжить вычисления, проигнорировав уведомление.

В последнем случае сообщение о возникшей ошибке будет послано всем слушателям коллекции `TraceListenerCollection`.

Рассмотрим простой пример, демонстрирующий нарушение утверждения:

```
public void WriteToFile()
{
   Stream myFile = new
    FileStream("TestFile.txt",FileMode.Create,FileAccess.Write);
   TextWriterTraceListener myTextListener =
      new TextWriterTraceListener(myFile);
   int y = Debug.Listeners.Add(myTextListener);
   TextWriterTraceListener myWriter =
      new TextWriterTraceListener(System.Console.Out);
   Trace.Listeners.Add(myWriter);
   Trace.AutoFlush = true;
   Trace.WriteLine("автоматический вывод из буфера:"
      + Trace.AutoFlush);
   int x = 22;
   Trace.Assert(x<=21, "Перебор");
   myWriter.WriteLine("Вывод только на консоль");
   //Trace.Flush();
   //Вывод только в файл
   byte[] buf = {(byte)'B',(byte)'y'};
   myFile.Write(buf,0, 2);
   myFile.Close();
}
```

Как и в предыдущем примере, здесь создаются два слушателя, направляющие вывод отладочных сообщений на консоль и в файл. Когда произошло нарушение утверждения `Assert`, оно было проигнорировано, но сообщение о нем автоматически было направлено всем слушателям. Метод также демонстрирует возможность параллельной работы с консолью и файлом. На рис. 23.4 показаны результаты записи в файл:

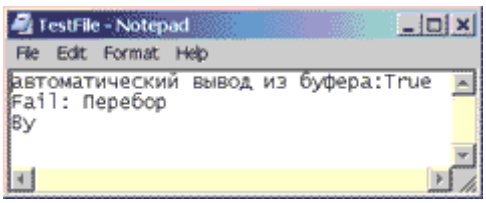

Рис. 23.4. Файл с записью сообщения о нарушении утверждения Assert

Вариацией метода `Assert` является метод `Fail`, всегда приводящий к появлению окна с сообщением о нарушении утверждения, проверка которого осуществляется обычным программным путем.

Классы StackTrace и BooleanSwitch

В библиотеке FCL имеются и другие классы, полезные при *отладке*. Класс `StackTrace` позволяет получить программный доступ к стеку вызовов. Класс `BooleanSwitch` предоставляет механизм, аналогичный *константам условной компиляции*. Он разрешает определять константы, используемые позже в методе условной печати `WriteIf` классов `Debug` и `Trace`. Мощь этого механизма в том, что константы можно менять в файле конфигурации проекта, не изменяя код проекта и не требуя его перекомпиляции.

Отладка и инструментальная среда Visual Studio .Net

Инструментальная среда студии предоставляет программисту самый широкий спектр возможностей слежения за ходом вычислений и отслеживания состояний, в котором находится процесс вычислений. Поскольку все современные инструментальные среды организованы сходным образом и хорошо известны работающим программистам, я позволю себе не останавливаться на описании возможностей среды.

Обработка исключительных ситуаций

Какой бы *надежный код* ни был написан, сколь бы тщательной ни была *отладка*, в версии, переданной в эксплуатацию и на сопровождение, при запусках будут встречаться нарушения *спецификаций*. Причиной этого являются выше упомянутые законы *программотехники*. В системе остается последняя ошибка, находятся пользователи, не знающие *спецификаций*, и если *спецификацию* можно нарушить, то это событие когда-нибудь да произойдет. В таких *исключительных ситуациях* продолжение выполнения программы либо становится невозможным (попытка выполнить неразрешенную операцию деления на ноль, попытки записи в защищенную область памяти, попытка открытия несуществующего файла, попытка получить несуществующую запись базы данных), либо в возникшей ситуации применение алгоритма приведет к ошибочным результатам.

Что делать при возникновении *исключительной ситуации*? Конечно, всегда есть стандартный способ - сообщить о возникшей ошибке и прервать выполнение программы. Понятно, что это приемлемо лишь для безобидных приложений; даже для компьютерных игр этот способ не годится, что уж говорить о критически важных приложениях!

В языках программирования для *обработки исключительных ситуаций* предлагались самые разные подходы.

Обработка исключений в языках C/C++

Для стиля программирования на языке C характерно описание методов класса как булевых функций, возвращающих `true` в случае нормального завершения метода и

`false` - при возникновении *исключительной ситуации*. Вызов метода встраивался в `If`-оператор, обрабатывающий ошибку в случае неуспеха завершения метода:

```
bool MyMethod(...){...}
if !MyMethod(){// обработка ошибки}
{//нормальное выполнение}
```

Недостатки этой схемы понятны. Во-первых, мало информации о причине возникновения ошибки, поэтому либо через поля класса, либо через аргументы метода нужно передавать дополнительную информацию. Во-вторых, блок обработки встраивается в каждый вызов, что приводит к раздуванию кода.

Поэтому в C/C++ применяется схема try/catch блоков, суть которой в следующем. Участок программы, в котором может возникнуть *исключительная ситуация*, оформляется в виде охраняемого try-блока. Если при его выполнении возникает *исключительная ситуация*, то происходит прерывание выполнения try-блока с классификацией исключения. Это исключение начинает обрабатывать один из catch-блоков, соответствующий *типу исключения*. В C/C++ применяются две такие схемы. Одна из них - схема с возобновлением - соответствует так называемым структурным, или C-исключениям. Вторая схема - *без возобновления* - соответствует C++ исключениям. В первой схеме обработчик исключения - catch-блок - возвращает управление в некоторую точку try-блока. Во второй схеме управление не возвращается в try-блок.

С некоторыми синтаксическими отличиями схема с возобновлением применяется в языках VB/VBA.

Многообразие подходов к *обработке исключений* говорит о том, что не найден единый, удовлетворяющий всех подход. Чуть позже я расскажу о наиболее разумной, с моей точки зрения, схеме. К сожалению, в C# применяется не она.

Схема обработки исключений в C#

Язык C# наследовал схему исключений языка C++, внеся в нее свои коррективы. Рассмотрим схему подробнее и начнем с синтаксиса конструкции **try-catch-finally**:

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

Всюду в тексте модуля, где синтаксически допускается использование блока, этот блок можно сделать охраняемым, добавив ключевое слово `try`. Вслед за try-блоком могут следовать catch-блоки, называемые ***блоками-обработчиками исключительных ситуаций***, их может быть несколько, они могут и отсутствовать. Завершает эту последовательность **finally-блок** - блок финализации, который также может отсутствовать. Вся эта конструкция может быть вложенной - в состав try-блока может входить конструкция **try-catch-finally**.

Выбрасывание исключений. Создание объектов Exception

В теле try-блока может возникнуть *исключительная ситуация*, приводящая к *выбрасыванию исключений*. Формально *выбрасывание исключения* происходит при выполнении оператора `throw`. Этот оператор, чаще всего, выполняется в недрах операционной системы, когда система команд или функция API не может сделать свою работу. Но этот оператор может быть частью программного текста try-блока и

выполняться, когда в результате проведенного анализа становится понятным, что дальнейшая нормальная работа невозможна.

Синтаксически оператор `throw` имеет вид:

```
throw[выражение]
```

Выражение `throw` задает объект класса, являющегося наследником *класса Exception*. Обычно это выражение `new`, создающее новый объект. Если оно отсутствует, то повторно выбрасывается текущее исключение. Если исключение выбрасывается операционной системой, то она сама классифицирует исключение, создает объект соответствующего класса и автоматически заполняет его поля.

В рассматриваемой нами модели исключения являются объектами, класс которых представляет собой наследника *класса Exception*. Этот класс и многочисленные его наследники является частью библиотеки FCL, хотя и разбросаны по разным пространствам имен. Каждый класс задает определенный *тип исключения* в соответствии с классификацией, принятой в Framework .Net. Вот лишь некоторые *классы исключений* из пространства имен `System`: `Argument Exception`, `ArgumentOutOfRangeException`, `ArithmeticException`, `BadImageFormatException`, `DivideByZeroException`, `OverflowException`. В пространстве имен `System.IO` собраны *классы исключений*, связанных с проблемами ввода-вывода: `DirectoryNotFoundException`, `FileNotFoundException` и многие другие. Имена всех *классов исключений* заканчиваются словом `Exception`. Разрешается создавать собственные *классы исключений*, наследуя их от *класса Exception*.

При выполнении оператора `throw` создается объект `te`, класс `TE` которого характеризует текущее исключение, а поля содержат информацию о возникшей *исключительной ситуации*. Выполнение оператора `throw` приводит к тому, что нормальный процесс вычислений на этом прекращается. Если это происходит в охраняемом try-блоке, то начинается этап *"захвата" исключения* одним из обработчиков исключений.

Захват исключения

Блок `catch` - обработчик исключения имеет следующий синтаксис:

```
catch (T e) {...}
```

Класс `T`, указанный в заголовке `catch`-блока, должен принадлежать *классам исключений*. Блок `catch` с формальным аргументом e класса `T` потенциально способен захватить текущее исключение `te` класса `TE`, если и только если объект `te` совместим по присваиванию с объектом e. Другими словами, потенциальная способность захвата означает допустимость присваивания `e = te`, что возможно, когда класс `TE` является потомком класса `T`. Обработчик, класс `T` которого является *классом Exception*, является *универсальным обработчиком*, потенциально он способен захватить любое исключение, поскольку все они являются его потомками.

Потенциальных захватчиков может быть много, исключение захватывает лишь один - тот из них, кто стоит первым в списке проверки. Каков порядок проверки? Он довольно естественный. Вначале проверяются обработчики в порядке следования их за `try`-блоком, и первый потенциальный захватчик становится активным, захватывая исключение и выполняя его обработку. Отсюда становится ясно, что порядок следования в списке `catch`-блоков крайне важен. Первыми идут наиболее *специализированные обработчики*, далее по мере возрастания универсальности. Так, вначале должен идти обработчик исключения `DivideByZeroException`, а уже за ним -

ArithmeticException. *Универсальный обработчик*, если он есть, должен стоять последним. За этим наблюдает *статический контроль типов*. Если потенциальных захватчиков в списке catch-блоков нет (сам список может отсутствовать), то происходит переход к списку обработчиков охватывающего блока. Напомню, что try-блок может быть вложен в другой try-блок. Когда же будет исчерпаны списки вложенных блоков, а потенциальный захватчик не будет найден, то произойдет подъем по стеку вызовов. На рис. 23.5 показана цепочка вызовов, начинающаяся с точки "большого взрыва" - процедуры Main.

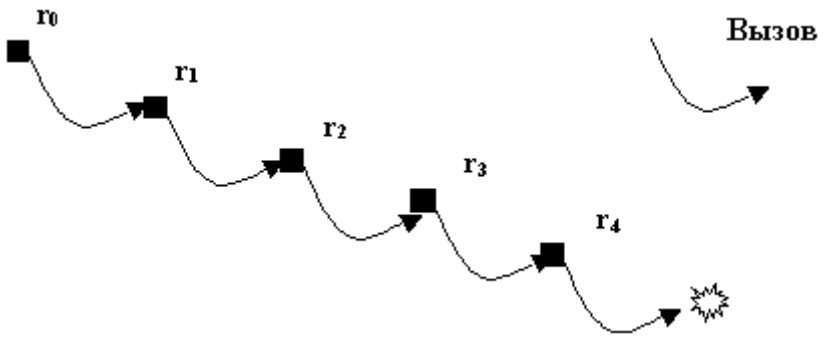

Рис. 23.5. Цепочка вызовов, хранящаяся в стеке вызовов

О точке большого взрыва и цепочке вызовов мы говорили еще в лекции 2.

Исключение возникло в последнем вызванном методе цепочки - на рисунке метод r5. Если у этого метода не нашлось обработчиков события, способных обработать исключение, то это пытается сделать метод r4, вызвавший r5. Если вызов r5 находится в охраняемом блоке метода r4, то начнет проверяться список обработчиков в охраняемом блоке метода r4. Этот процесс подъема по списку вызовов будет продолжаться, пока не будет найден обработчик, способный захватить исключение, или не будет достигнута начальная точка - процедура Main. Если и в ней нет потенциального захватчика исключения, то сработает стандартный обработчик, прерывающий выполнение программы с выдачей соответствующего сообщения.

Параллельная работа обработчиков исключений

Обработчику исключения - catch-блоку, захватившему исключение, - передается текущее исключение. Анализируя свойства этого объекта, обработчик может понять причину, приведшую к возникновению *исключительной ситуации*, попытаться ее исправить и в случае успеха продолжить вычисления. Заметьте, в принятой C# *схеме без возобновления* обработчик исключения не возвращает управление try-блоку, а сам пытается решить проблемы. После завершения catch-блока выполняются операторы текущего метода, следующие за конструкцией try-catch-finally.

Зачастую обработчик исключения не может исправить ситуацию или может выполнить это лишь частично, предоставив решение оставшейся части проблем вызвавшему методу - предшественнику в цепочке вызовов. Механизм, реализующий такую возможность - это тот же механизм исключений. Как правило, в конце своей работы обработчик исключения выбрасывает исключение, выполняя оператор throw. При этом у него есть две возможности: повторно выбросить текущее исключение или выбросить новое исключение, содержащее дополнительную информацию.

Некоторые детали будут пояснены позже при рассмотрении примеров.

Таким образом, обработку возникшей *исключительной ситуации* могут выполнять несколько обработчиков, принадлежащие разным уровням цепочки вызовов.

Блок finally

До сих пор ничего не было сказано о важном участнике схемы *обработки исключений* - блоке `finally`. Напомню, рассматриваемая схема является *схемой без возобновления*. Это означает, что управление вычислением неожиданно покидает try-блок. Просто так этого делать нельзя - нужно выполнить определенную чистку. Прежде всего удаляются все локальные объекты, созданные в процессе работы блока. В языке C++ эта работа требовала вызова деструкторов объектов. В C#, благодаря *автоматической сборке мусора*, освобождением памяти можно не заниматься, достаточно освободить стек. Но в блоке try могли быть заняты другие ресурсы - открыты файлы, захвачены некоторые устройства. Освобождение ресурсов, занятых try-блоком, выполняет finally-блок. Если он присутствует, он выполняется всегда, сразу же после завершения работы try-блока, как бы последний ни завершился. Блок try может завершиться вполне нормально без всяких происшествий и управление достигнет конца блока, выполнение может прервано оператором throw, управление может быть передано другому блоку из-за выполнения таких операторов как goto, return - во всех этих случаях, прежде чем управление будет передано по предписанному назначению (в том числе, прежде чем произойдет *захват исключения*), предварительно будет выполнен finally-блок, который освобождает ресурсы, занятые try-блоком, а параллельно будет происходить освобождение стека от локальных переменных.

Схема Бертрана обработки исключительных ситуаций

Схема обработки *исключительных ситуаций*, предложенная в языке C#, обладает одним существенным изъяном - ее можно применить некорректно. Она позволяет, в случае возникновения *исключительной ситуации*, уведомить о ее возникновении и спокойно продолжить работу, что в конечном счете приведет к неверным результатам. Из двух зол - прервать вычисление с уведомлением о невозможности продолжения работы или закончить вычисления с ошибочным результатом вычисления - следует выбирать первое. Некорректно примененная схема C# приведет к ошибочным результатам. Приведу несколько примеров. Представьте, оформляется заказ на отдых где-нибудь на Канарах. В ходе оформления возникает *исключительная ситуация* - нет свободных мест в гостинице - обработчик исключения посылает уведомление с принесением извинений, но оформление заказа продолжается. Вероятно, предпочтительнее отказаться от отдыха на Канарах и выбрать другое место, чем оказаться без крыши над головой, ночуя на берегу океана. Эта ситуация не является критически важной. А что, если в процессе подготовки операции выясняется, что проведение ее в данном случае опасно? Никакие извинения не могут избавить от вреда, нанесенного операцией. Операция должна быть отменена.

Бертран Мейер в книге [1], в которой все механизмы, используемые в объектной технологии, тщательно обосновываются, предложил следующую схему обработки *исключительных ситуаций*. В основе ее лежит подход к проектированию программной системы на принципах *Проектирования по Контракту*. Модули программной системы, вызывающие друг друга, заключают между собой контракты. Вызывающий модуль обязан обеспечить истинность предусловия, необходимого для *корректной* работы вызванного модуля. Вызванный модуль обязан гарантировать истинность постусловия по завершении своей работы. Если в вызванном модуле возникает *исключительная ситуация*, то это означает, что он не может выполнить свою часть контракта. Что должен делать обработчик *исключительной ситуации*? У него только две возможности - Retry и Rescue. Первая (Retry) - попытаться внести некоторые коррективы и вернуть управление охраняемому модулю, который может предпринять очередную попытку выполнить свой контракт. Модуль может, например в следующей попытке запустить другой алгоритм, использовать другой файл, другие данные. Если все закончится успешно и работа модуля будет соответствовать его постусловию, то появление *исключительной ситуации* можно рассматривать как временные трудности, успешно преодоленные. Если же ситуация возникает вновь и вновь, тогда обработчик события применяет вторую стратегию (Rescue), выбрасывая исключение и передавая

управление вызывающему модулю, который и должен теперь попытаться исправить ситуацию. Важная тонкость в *схеме*, предложенной Бертраном, состоит в том, что исключение, выбрасываемое обработчиком, следует рассматривать не как панику, не как бегство, а как отход на заранее подготовленные позиции. Обработчик исключения должен позаботиться о восстановлении состояния, предшествующего вызову модуля, который привел к *исключительной ситуации*, и это гарантирует нахождение всей системы в *корректном* состоянии.

Схема Бертрана является схемой с возобновлением, и она наиболее точно описывает разумную стратегию обработки *исключительных ситуаций*. Не следует думать, что эта схема не может быть реализована на C#, просто она требует понимания сути и определенной структурной организации модуля. Приведу возможную реализацию такой схемы на C#:

```
public void Pattern()
{
   do
   {
      try
      {
         bool Danger = false;
         Success = true;
         MakeJob();
         Danger = CheckDanger();
         if (Danger)
            throw (new MyException());
         MakeLastJob();
      }
      catch (MyException me)
      {
         if(count > maxcount)
            throw(new MyException("Три попытки были
               безуспешны"));
               Success = false; count++;
         //корректировка ситуации
         Console.WriteLine("Попытка исправить ситуацию!");
         level +=1;
      }
   }while (!Success);
}
```

Приведу краткие комментарии к этой процедуре, которую можно рассматривать как некоторый образец организации обработки *исключительной ситуации*:

- Конструкция `try-catch` блоков помещается в цикл `do-while(!Success)`, завершаемый в случае успешной работы охраняемого блока, за чем следит булева переменная Success.
- В данном образце предполагается, что в теле охраняемого блока анализируется возможность возникновения *исключительной ситуации* и, в случае обнаружения опасности, выбрасывается собственное исключение, класс которого задан программно. В соответствии с этим тело `try`-блока содержит вызов метода `MakeJob`, выполняющего некоторую часть работы, после чего вызывается метод `CheckDanger`, выясняющий, не возникла ли опасность нарушения *спецификации* и может ли работа быть продолжена. Если все нормально, то выполняется метод `MakeLastJob`, выполняющий заключительную часть работы. Управление вычислением достигает конца `try`-блока, он успешно завершается и, поскольку остается истинной переменная `Success`, значение `true` которой установлено в начале `try`-блока, то цикл `while`, окаймляющий охраняемый блок и его обработчиков исключений, также успешно завершается.

- Если в методе `CheckDanger` выясняется, что нормальное продолжение вычислений невозможно, то выбрасывается исключение класса `MyException`. Оно перехватывается обработчиком исключения, стоящим за `try`-блоком, поскольку класс `MyException` указан как класс формального аргумента.
- Для простоты приведен только один `catch`-блок. В общем случае их может быть несколько, но все они строятся по единому образцу. Предполагается, что обработчик исключения может сделать несколько попыток исправить ситуацию, после чего повторно выполняется охраняемый блок. Если же число попыток, за которым следит переменная count, превосходит максимально допустимое, то обработчик выбрасывает новое исключение, задавая дополнительную информацию и передавая тем самым обработку ошибки на следующий уровень - вызываемой программе.
- Когда число попыток еще не исчерпано, обработчик исключения переменной `Success` дает значение `false`, гарантирующее повтор выполнения `try`-блока, увеличивает счетчик числа попыток и пытается исправить ситуацию.
- Как видите, эта схема реализует два *корректных* исхода обработки *исключительной ситуации* - `Retry` и `Rescue` - повтору с надеждой выполнить обязательства и передачи управления вызывающей программе, чтобы она предприняла попытки исправления ситуации, когда вызванная программа не смогла с этим справиться.

Доведем этот образец до реально работающего кода, где угроза исключения зависит от значения генерируемого случайного числа, а обработчик исключения может изменять границы интервала, повышая вероятность успеха.

Определим первым делом собственный *класс исключений*:

```
public class MyException :Exception
{
   public MyException()
   {}
   public MyException (string message) : base(message)
   {}
   public MyException (string message, Exception e) :
      base(message, e)
   {}
}
```

Минимум того, что нужно сделать, определяя свои исключения, - это задать три конструктора класса, вызывающие соответствующие конструкторы базового *класса Exception*.

В классе `Excepts`, методом которого является наш образец `Pattern`, определим следующие поля класса:

```
Random rnd = new Random();
      int level = -10;
      bool Success; //true - нормальное завершение
      int count =1; // число попыток выполнения
      const int maxcount =3;
```

Определим теперь методы, вызываемые в теле охраняемого блока:

```
void MakeJob()
{
   Console.WriteLine("Подготовительные работы завершены");
}
bool CheckDanger()
{
```

```
   //проверка качества и возможности продолжения работ
   int low = rnd.Next(level,10);
   if ( low > 6) return(false);
   return(true);
}
void MakeLastJob()
{
   Console.WriteLine("Все работы завершены успешно");
}
```

В классе `Testing` зададим метод, вызывающий метод `Pattern`:

```
public void TestPattern()
{
   Excepts ex1 = new Excepts();
   try
   {
      ex1.Pattern();
   }
   catch (Exception e)
   {
      Console.WriteLine("исключительная ситуация при
         вызове Pattern");
      Console.WriteLine(e.ToString());
   }
}
```

Обратите внимание, что вызов метода `Pattern` находится внутри охраняемого блока. Поэтому, когда `Pattern` не справится с обработкой *исключительной ситуации*, ее обработку возьмет на себя *универсальный обработчик*, стоящий за `try`-блоком.

На рис. 23.6 показаны три варианта запуска метода `TestPattern`. В одном из них *исключительной ситуации* при вызове метода `Pattern` вообще не возникало, в другом - ситуация возникала, но коррекция обработчика исключения помогла и при повторе выполнения охраняемого блока в `Pattern` все прошло нормально. В третьем варианте метод `Pattern` не смог справиться с *исключительной ситуацией*, и она обрабатывалась в `catch`-блоке метода `TestPattern`.

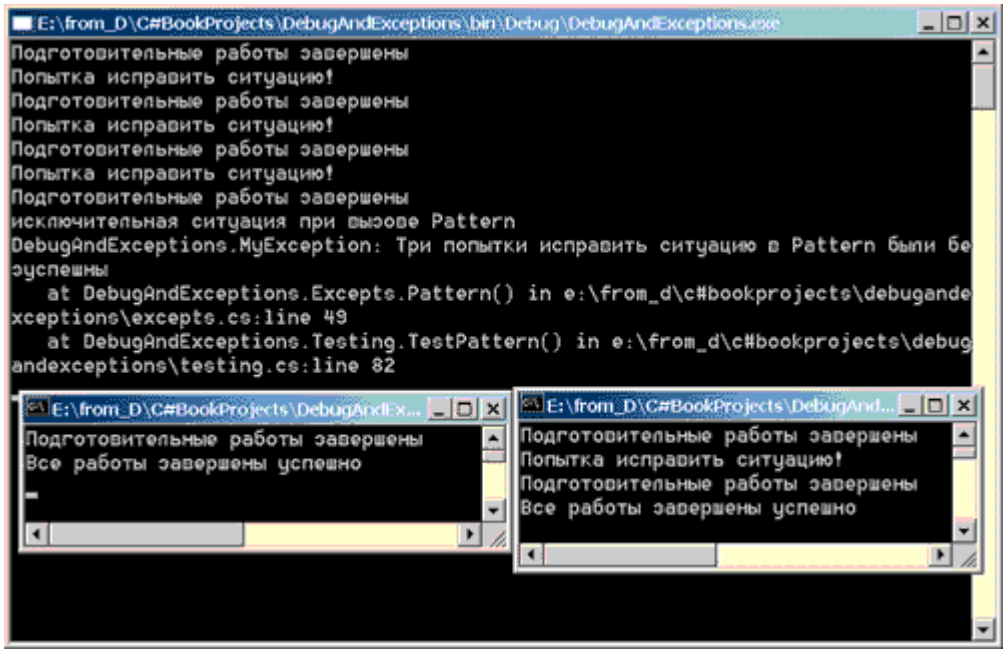

Рис. 23.6. Обработка исключительных ситуаций. Три случая

Класс Exception

Рассмотрим устройство базового *класса Exception*, чтобы понять, какую информацию может получить обработчик исключения, когда ему передается объект, задающий текущее исключение.

Основными свойствами класса являются:

- `Message` - строка, задающая причину возникновения исключения. Значение этого свойства устанавливается при вызове конструктора класса, когда создается объект, задающий исключение;
- `HelpLink` - ссылка (URL) на файл, содержащий подробную справку о возможной причине возникновения *исключительной ситуации* и способах ее устранения;
- `InnerException` - ссылка на внутреннее исключение. Когда обработчик выбрасывает новое исключение для передачи обработки на следующий уровень, то текущее исключение становится внутренним для вновь создаваемого исключения;
- `Source` - имя приложения, ставшего причиной исключения;
- `StackTrace` - цепочка вызовов - методы, хранящиеся в стеке вызовов в момент возникновения исключения;
- `TargetSite` - метод, выбросивший исключение.

Из методов класса отметим метод `GetBaseException`. При подъеме по цепочке вызовов он позволяет получить исходное исключение -- первопричину возникновения последовательности выбрасываемых исключений.

Класс имеет четыре конструктора, из которых три уже упоминались. Один из них - конструктор без аргументов, второй - принимает строку, становящуюся свойством `Message`, третий - имеет еще один аргумент: исключение, передаваемое свойству `InnerException`.

В предыдущий пример я внес некоторые изменения. В частности, добавил еще один аргумент при вызове конструктора исключения в `catch`-блоке метода `Pattern`:

```
throw(new MyException("Все попытки Pattern безуспешны", me));
```

В этом случае у создаваемого исключения заполняется свойство `InnerExceptions`. Для слежения за свойствами исключений я добавил метод печати всех свойств, вызываемый во всех обработчиках исключений:

```
static public void PrintProperties(Exception e)
{
   Console.WriteLine("Свойства исключения:");
   Console.WriteLine("TargetSite = {0}", e.TargetSite);
   Console.WriteLine("Source = {0}", e.Source);
   Console.WriteLine("Message = {0}",e.Message);
   if (e.InnerException == null)
      Console.WriteLine("InnerException = null");
   else Console.WriteLine("InnerException = {0}",
      e.InnerException.Message);
   Console.WriteLine("StackTrace = {0}", e.StackTrace);
   Console.WriteLine("GetBaseException = {0}",
      e.GetBaseException());
}
```

Из-за громоздкости не привожу результаты, но отмечу, что они соответствуют описанию, приведенному в тексте лекции.

В заключение темы исключений хочу еще раз подчеркнуть, что *корректное* применение механизма исключений должно поддерживаться целенаправленными усилиями программиста. Следует помнить о двух важных правилах:

- *обработка исключений* должна быть направлена не столько на уведомление о возникновении ошибки, сколько на корректировку возникшей ситуации;
- если исправить ситуацию не удается, то программа должна быть прервана так, чтобы не были получены некорректные результаты, не удовлетворяющие *спецификациям* программы.

Организация интерфейса

Практически все проекты, построенные в наших лекциях, были *консольными приложениями*. В реальной жизни консольные проекты - это большая редкость. Причина, по которой из 12 возможных типов проектов мы выбирали наименее используемый, понятна. Нашей целью являлось изучение свойств языка, *классов* библиотеки FCL, для этих целей консольный проект вполне подходит, позволяя избегать введения не относящихся к сути дела деталей. Теперь цель достигнута - основные средства языка C# рассмотрены, учебный курс завершается. Остались важные темы, требующие более подробного рассмотрения, такие, как, например, работа с атрибутами, создание собственных атрибутов, *класс* `Reflection`, работа с файлами и базами данных; но все это предмет будущего курса. Тем не менее, нельзя окончить этот курс, не посвятив две последние лекции *Windows-приложениям*. Мне бы хотелось, чтобы активные слушатели (читатели) все консольные проекты переделали в Windows-проекты, построив подходящий для них интерфейс.

Первое знакомство с Windows-проектами состоялось в лекции 2, я настоятельно рекомендую перечитать ее, прежде чем продолжить чтение данной лекции. Вкратце напомню, как создается и выполняется Windows-проект. По умолчанию он содержит *класс* `Form1` - наследника *класса* `Form`. Этот *класс* содержит точку входа в проект - процедуру `Main`, вызывающую статический метод `Run` *класса* `Application`, который создает объект *класса* `Form1` и открывает *форму* - видимый образ объекта - для интерактивной работы пользователя. Открываемая *форма* содержит пользовательский интерфейс - окошки, *кнопки*, списки, другие *элементы управления*, *меню* . Все эти *элементы* способны реагировать на события, возникающие при выполнении пользователем каких-либо действий - нажатии *кнопок*, ввода текста, выбора *пунктов меню*.

Форма и элементы управления

Как населить *форму элементами управления*? Чаще всего, это делается вручную в режиме проектирования. Доступные *элементы управления*, отображаемые на специальной панели (Toolbox), перетаскиваются на *форму*. Этот процесс поддерживается особым инструментарием - дизайнером *форм* (Designer Form). Как только на этапе проектирования вы сажаете на *форму элемент управления*, немедленно в тексте *класса* появляются соответствующие строки кода (в лекции 2 об этом подробно рассказано). Конечно, все можно делать и программно - появление соответствующих строк кода приводит к появлению *элементов управления* на *форме*. Нужно понимать, что *форма* - это видимый образ *класса* `Form`, а *элементы управления*, размещенные на *форме* - это видимые *образы клиентских объектов* соответствующих *классов*, наследников *класса* `Control`. Так что *форма* с ее *элементами управления* есть прямое отражение программного кода.

Каждый вид *элементов управления* описывается собственным *классом*. Библиотека FCL содержит большое число *классов*, задающих различные *элементы управления*. Одним из типов проектов, доступных на C#, является проект, создающий *элемент управления*, так что ничто не мешает создавать собственные *элементы управления* и размещать их

на *формах* наряду со встроенными элементами. Многие фирмы специализируются на создании *элементов управления* - это один из видов повторно используемых компонентов.

В каких отношениях находятся *класс* `Form`, *класс* `Control`, *классы элементов управления*? На рис. 24.1 показана иерархия отношений, связывающих эти *классы*.

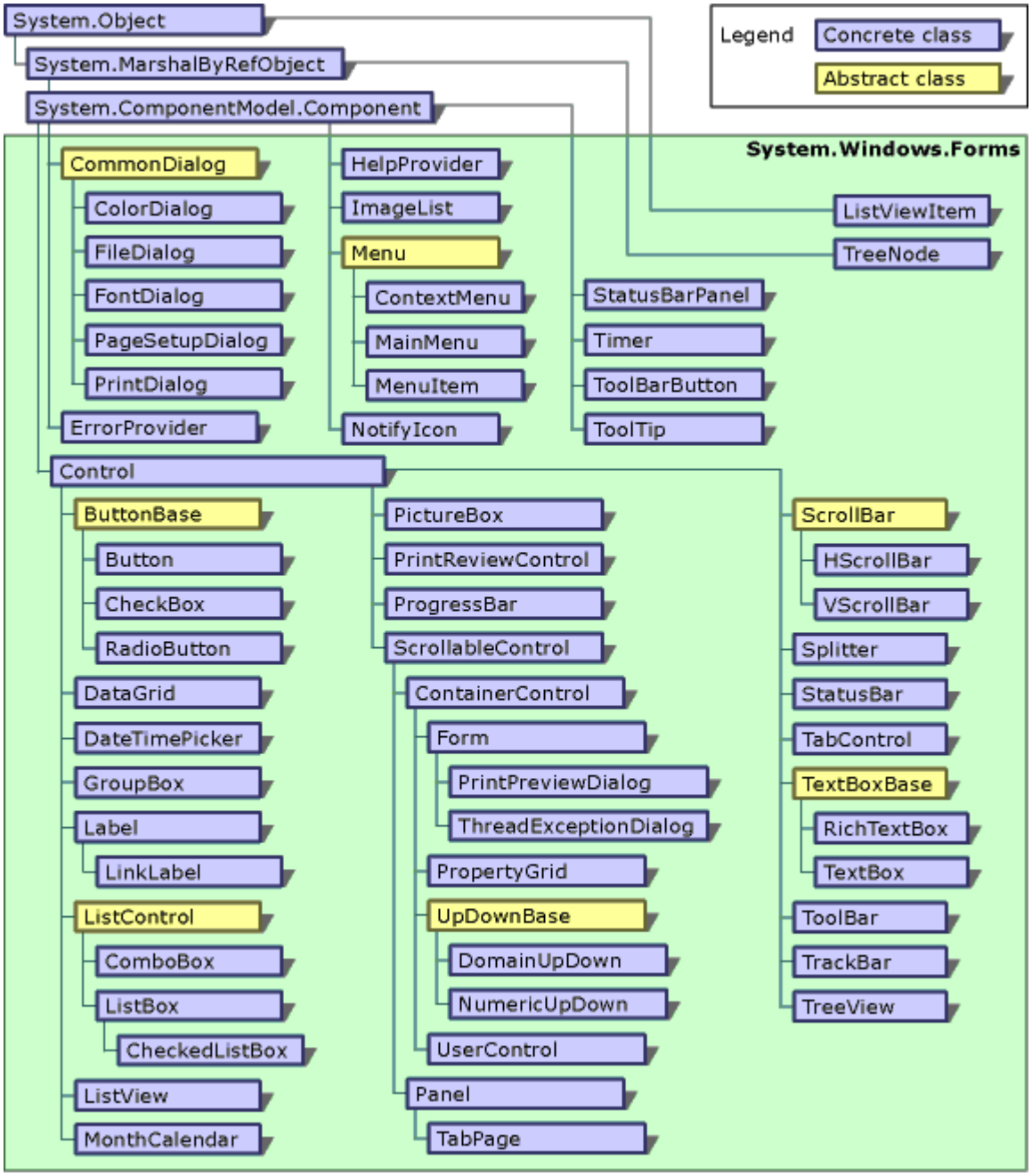

Рис. 24.1. Иерархия классов элементов управления

Естественно, все эти *классы* являются потомками прародителя - *класса* `Object`. Заметьте, *класс* `Control` в иерархии *классов* занимает довольно высокое положение, хотя и у него есть два важных родительских *класса* - *класс* `Component`, определяющий возможность *элементам управления* быть компонентами, и *класс* `MarshalByRefObject`, задающий возможность передачи *элементов управления* по сети. *Класс* `Control` задает важные свойства, методы и события, наследуемые всеми его потомками. Все *классы элементов управления* являются наследниками *класса* `Control`. Чаще всего, это прямые наследники, но иногда они имеют и непосредственного родителя, которым может быть абстрактный *класс* - это верно для *кнопок*, списков, текстовых *элементов управления*.

Может показаться удивительным, но *класс* `Form` является одним из потомков *класса* `Control`, так что *форма* - это *элемент управления* со специальными свойствами. Будучи наследником *классов* `ScrollableControl` и `ContainerControl`, *форма* допускает прокрутку и размещение *элементов управления*.

Взаимодействие форм

Обычное *Windows-приложение* всегда содержит несколько *форм*. Одни открываются в процессе работы, другие закрываются. В каждый текущий момент на экране может быть открыта одна или несколько *форм*, пользователь может работать с одной *формой* или переключаться по ходу работы с одной на другую.

Следует четко различать процесс создания *формы* - соответствующего объекта, принадлежащего *классу* `Form` или наследнику этого *класса*, - и процесс *показа формы* на экране. Для *показа формы* служит метод `Show` этого *класса*, вызываемый соответствующим объектом; для **скрытия** *формы* используется метод `Hide`. Реально методы `Show` и `Hide` изменяют свойство `Visible` объекта, так что вместо вызова этих методов можно менять значение этого свойства, устанавливая его либо в `true`, либо в `false`.

Заметьте разницу между *сокрытием и закрытием формы* - между методами `Hide` и `Close`. Первый из них делает *форму* невидимой, но сам объект остается живым и невредимым. Метод `Close` отбирает у *формы* ее ресурсы, делая объект отныне недоступным; вызвать метод `Show` после вызова метода `Close` невозможно, если только не создать объект заново. Открытие и *показ формы* всегда означает одно и то же - вызов метода `Show`. У *формы* есть метод `Close`, но нет метода `Open`. *Формы*, как и все объекты, создаются при вызове *конструктора формы* при выполнении операции `new`.

Форма, открываемая в процедуре `Main` при вызове метода `Run`, называется *главной формой проекта*. Ее закрытие приводит к закрытию всех остальных *форм* и завершению *Windows-приложения*. Завершить приложение можно и программно, вызвав в нужный момент статический метод `Exit` *класса* `Application`. Закрытие других *форм* не приводит к завершению проекта. Зачастую *главная форма проекта* всегда открыта, в то время как остальные *формы* открываются и закрываются (скрываются). Если мы хотим, чтобы в каждый текущий момент была открыта только одна *форма*, то нужно принять определенные меры, чтобы при закрытии (скрытии) *формы* открывалась другая. Иначе возможна клинчевая ситуация - все *формы* закрыты, предпринять ничего нельзя, а приложение не завершено. Конечно, выход всегда есть - всегда можно нажать магическую тройку клавиш **CTRL+ALT+DEL** и завершить любое приложение.

Можно создавать *формы* как объекты *класса* `Form`. Однако такие объекты довольно редки. Чаще всего создается специальный *класс* `FormX` - наследник *класса* `Form`. Так, в частности, происходит в *Windows-приложении*, создаваемом по умолчанию, когда создается *класс* `Form1` - наследник *класса* `Form`. Так происходит в режиме проектирования, когда в проект добавляется новая *форма* с использованием *пункта меню* `Add Windows Form`. Как правило, каждая *форма* в проекте - это объект собственного *класса*. Возможна ситуация, когда вновь создаваемая *форма* во многом должна быть похожей на уже существующую, и тогда *класс* новой *формы* может быть сделан наследником *класса формы* существующей. Наследование *форм* мы рассмотрим подробнее чуть позже.

Модальные и немодальные формы

Первичным является понятие модального и немодального окна. Окно называется модальным, если нельзя закончить работу в открытом окне до тех пор, пока оно не будет закрыто. Модальное окно не позволяет, если оно открыто, временно

переключиться на работу с другим окном. Выйти из модального окна можно, только закрыв его. Немодальные окна допускают параллельную работу в окнах. *Форма* называется модальной или немодальной в зависимости от того, каково ее окно. Метод `Show` открывает *форму* как немодальную, а метод `ShowDialog` - как модальную. Название метода отражает основное назначение модальных *форм* - они предназначены для организации диалога с пользователем, и пока диалог не завершится, покидать *форму* не разрешается.

Передача информации между формами

Часто многие *формы* должны работать с одним и тем же объектом, производя над ним различные операции. Как это реализуется? Обычная схема такова: объект создается в одной из *форм*, чаще всего, в главной. При создании следующей *формы* глобальный объект передается *конструктору* новой *формы* в качестве аргумента. Естественно, одно из полей новой *формы* должно представлять ссылку на объект соответствующего *класса*, так что *конструктору* останется только связать ссылку с переданным ему объектом. Заметьте, все это эффективно реализуется, поскольку объект создается лишь один раз, а разные *формы* содержат ссылки на этот единственный объект.

Если такой глобальный объект создается в *главной форме*, то можно передавать не объект, требуемый другим *формам*, а содержащий его контейнер - *главную форму*. Это удобнее, поскольку при этом можно передать несколько объектов, можно не задумываться над тем, какой объект передавать той или иной *форме*. Иметь ссылку на *главную форму* часто необходимо, хотя бы для того, чтобы при закрытии любой *формы* можно было бы открывать главную, если она была предварительно скрыта.

Представим себе, что несколько *форм* должны работать с объектом *класса* `Books`. Пусть в *главной форме* такой объект объявлен:

```
public Books myBooks;
```

В *конструкторе главной формы* такой объект создается:

```
myBooks = new Books(max_books);
```

где `max_books` - заданная константа. Пусть еще в *главной форме* объявлена *форма* - объект *класса* `NewBook`:

```
public NewBook form2;
```

При создании объекта `form2` его *конструктору* передается ссылка на *главную форму*:

```
form2 = new NewBook(this);
```

Класс `newBook` содержит *поля*:

```
private Form1 mainform;
private Books books;
```

а его *конструктор* следующий код:

```
mainform = form;
books = mainform.myBooks;
```

Теперь объекту `form2` доступны ранее созданные объекты, задающие книги и *главную форму*, так что в *обработчике события* `Closed`, возникающего при закрытии *формы*, можно задать код:

```
private void NewBook_Closed(object sender, System.EventArgs e)
{
   mainform.Show();
}
```

открывающий *главную форму*.

Образцы форм

Создание элегантного, интуитивно ясного интерфейса пользователя - это своего рода искусство, требующее определенного художественного вкуса. Здесь все играет важную роль: размеры и расположение *элементов управления*, шрифты, важную роль играет цвет. Но тема красивого интерфейса лежит вне нашего рассмотрения. Нас сейчас волнует содержание. Полезно знать некоторые образцы организации интерфейса.

Главная кнопочная форма

Одним из образцов, применимых к *главной форме*, является *главная кнопочная форма*. Такая *форма* состоит из текстового окна, в котором описывается приложение и его возможности, и ряда командных *кнопок*; обработчик каждой *кнопки* открывает *форму*, позволяющую решать одну из задач, которые поддерживаются данным приложением. В качестве примера рассмотрим *Windows-приложение*, позволяющее работать с различными динамическими структурами данных. *Главная кнопочная форма* такого приложения показана на рис. 24.2.

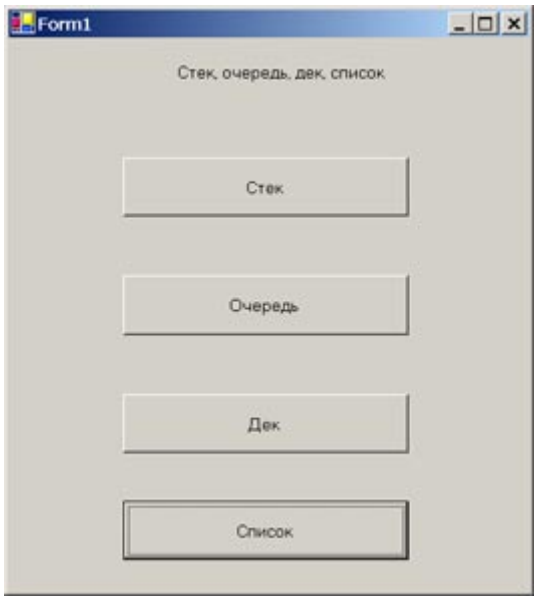

Рис. 24.2. Главная кнопочная форма

Обработчик события `Click` для каждой командной *кнопки* открывает *форму* для работы с соответствующей динамической структурой данных. Вот как выглядит *обработчик события кнопки* Список:

```
private void button4_Click(object sender, System.EventArgs e)
{
   //Переход к показу формы для работы со списком
   FormList fl= new FormList();
   fl.Show();
}
```

Как видите, открывается новая *форма* для работы со списком, но *главная форма* не закрывается и остается открытой.

Шаблон формы для работы с классом

Можно предложить следующий образец *формы*, предназначенной для поддержки работы с объектами некоторого *класса*. Напомню, каждый *класс* представляет тип данных. Операции над типом данных можно разделить на три категории: *конструкторы*, *команды* и *запросы*. *Конструкторы класса* позволяют создать соответствующий объект; *команды*, реализуемые процедурами, изменяют состояние объекта; *запросы*, реализуемые функциями без побочных эффектов, возвращают информацию о состоянии объекта, не изменяя самого состояния. Исходя из этого, можно сконструировать интерфейс *формы*, выделив в нем три секции. В первой секции, разделенной на три раздела, будут представлены *команды*, *запросы* и *конструкторы*. Следующая секция выделяется для окон, в которые можно вводить аргументы исполняемых *команд*. Последняя секция предназначается для окон, в которых будут отображаться результаты *запросов*.

На рис. 24.3 показана *форма* для списка с курсором, построенная в соответствии с описанным шаблоном.

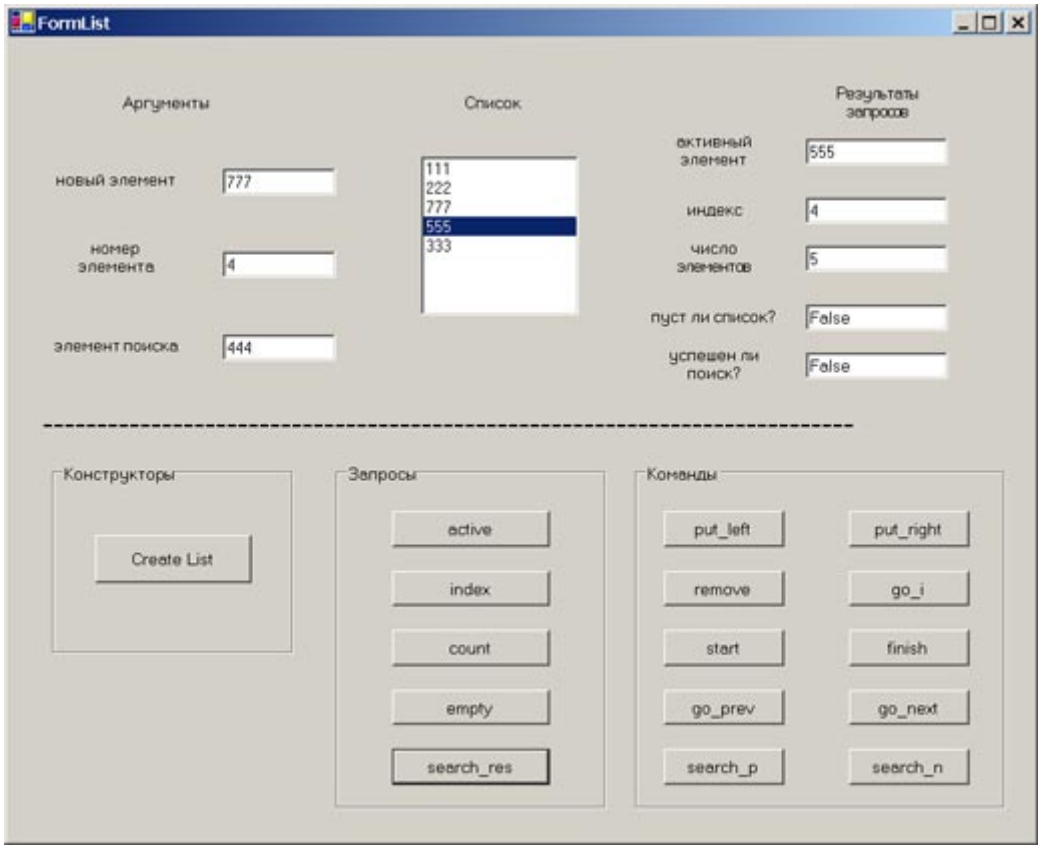

Рис. 24.3. Форма для списка с курсором, построенная по образцу

Список с курсором имеет группу *команд*, позволяющих перемещать курсор влево, вправо, к началу и концу списка, к элементу с заданным номером. Другая группа *команд* позволяет производить операции по вставке элементов слева или справа от курсора, удалять элемент, отмеченный курсором. Еще одна группа позволяет производить поиск элементов в списке. *Запросы* позволяют получить данные об активном элементе, отмеченном курсором, определить число элементов в списке и получить другую полезную информацию.

Работа со списками (еще один шаблон)

Для организации интерфейса разработано большое число *элементов управления*, часть из них показана на рис. 24.1. Все они обладают обширным набором свойств, методов и

событий, их описание может занять отдельную книгу. Такие элементы, как, например, ListView, TreeView, DataGrid, несомненно, заслуживают отдельного рассмотрения, но не здесь и не сейчас. Я ограничусь более подробным разбором лишь одного *элемента управления* - *ListBox*, - позволяющего отображать данные в виде некоторого списка.

Элемент управления класса ListBox

Во многих задачах пользователю предлагается некоторый список товаров, гостиниц, услуг и прочих прелестей, и он должен выбрать некоторое подмножество элементов из этого списка. *Элемент управления ListBox* позволяет собрать в виде списка некоторое множество объектов и отобразить для каждого объекта связанную с ним строку. Он дает возможность пользователю выбрать из списка один или несколько элементов.

В списке могут храниться строки, тогда объект совпадает с его отображением. Если же хранятся объекты, то в *классе* объекта следует переопределить метод ToString, возвращаемый результат которого и будет строкой, отображаемой в списке.

Давайте рассмотрим главный вопрос: как список заполняется элементами? Есть несколько разных способов. Новой и интересной технологией, применимой к самым разным *элементам управления*, является связывание *элемента управления* с данными, хранящимися в различных хранилищах, прежде всего, в базах данных. Для этого у списка есть ряд свойств - DataBinding и другие. Эта технология заслуживает отдельного рассмотрения, я о ней только упоминаю, но рассматривать ее не буду. Рассмотрим три других способа.

Заполнить список элементами можно еще на этапе проектирования. Для этого достаточно выбрать свойство Items: появится специальное окно для заполнения списка строками - элементами списка. Добавлять объекты других *классов* таким способом невозможно.

Но это можно делать при программной работе со свойством Items, возвращающим специальную коллекцию объектов, которая задана *классом* ObjectCollection. Эта коллекция представляет объекты, хранимые в списке, и является основой для работы со списком. *Класс* ObjectCollection предоставляет стандартный набор методов для работы с коллекцией - вставки, удаления и поиска элементов. Метод Add позволяет добавить новый объект в конец коллекции, метод Insert позволяет добавить элемент в заданную позицию, указанную индексом. Метод AddRange позволяет добавить сразу множество элементов, заданное обычным массивом, массивом *класса* ListArray или коллекцией, возвращаемой свойством Items другого списка. Для удаления элементов используются методы Remove, RemoveAt, Clear. Метод Contains позволяет определить, содержится ли заданный объект в коллекции, а метод IndexOf позволяет определить индекс такого элемента. Коллекция может автоматически сортироваться, для этого достаточно задать значение true свойства Sorted, которым обладает список ListBox.

Еще один способ задания элементов списка поддерживается свойством DataSource, значение которого позволяет указать источник данных, ассоциируемый со списком. Понятно, что этот способ является альтернативой коллекции, задаваемой свойством Items. Так что, если источник данных определен свойством DataSource, то нельзя использовать методы *класса* ObjectCollection - Add и другие для добавления или удаления элементов списка, - необходимо изменять сам источник данных.

Главное назначение *элемента ListBox* - предоставить пользователю возможность осуществлять выбор из отображаемых списком элементов. Свойство SelectionMode позволяет указать, сколько элементов разрешается выбирать пользователю - один или несколько. Для работы с отобранными элементами имеется ряд свойств. SelectedItem и SelectedIndex возвращают первый отобранный элемент и его индекс. Свойства SelectedItems и SelectedIndices возвращают коллекции, заданные *классами*

SelectedObjectCollection и SelectedIndexCollection, которые дают возможность анализировать все отобранные пользователем объекты. Методы Contains и IndexOf позволяют определить, выбрал ли пользователь некоторый элемент. Добавлять или удалять элементы из этих коллекций нельзя.

Среди других методов и свойств ListBox - упомяну свойство MultiColumn, с помощью которого можно организовать показ элементов списка в нескольких столбцах; свойство HorizonalScrollBar, задающее горизонтальный скроллинг; методы BeginUpdate и EndUpdate, позволяющие повысить эффективность работы со списком. Все методы по добавлению и удалению элементов, стоящие после BeginUpdate, не будут приводить к перерисовке списка, пока не встретится метод EndUpdate.

У *элемента управления ListBox* большое число событий, с некоторыми из которых мы встретимся при рассмотрении примеров. Перейдем теперь к рассмотрению примеров работы с этим *элементом управления* и, как обещано, построим некоторый шаблон, демонстрирующий работу с двумя списками, когда пользователь может переносить данные из одного списка в другой и обратно. На рис. 24.4 показано, как выглядит *форма*, реализующая данный шаблон.

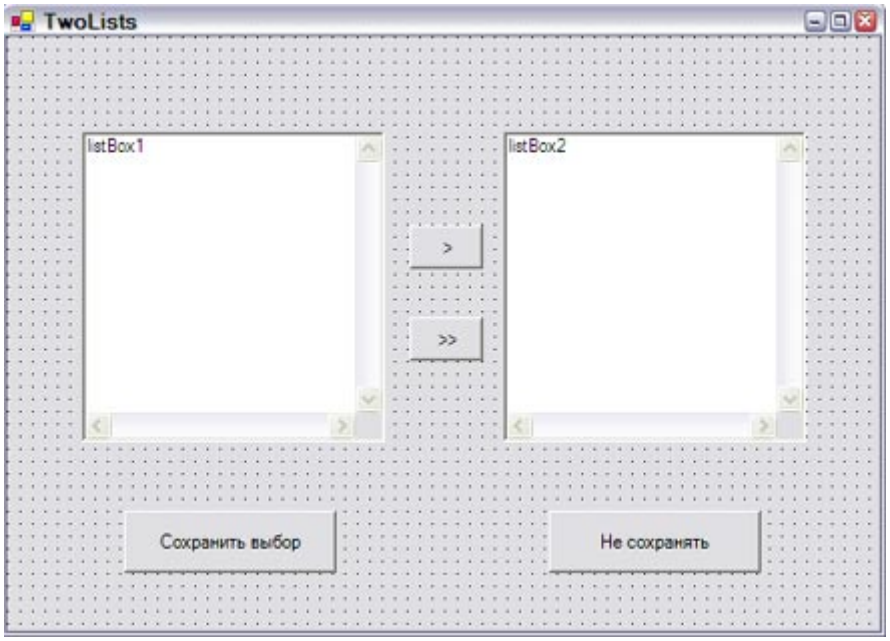

Рис. 24.4. Шаблон формы для обмена данными двух списков

На *форме* показаны два списка - listBox1 и listBox2, между которыми расположены две командные *кнопки*. *Обработчик события* Click первой *кнопки* переносит выбранную группу элементов одного списка в конец другого списка, (если включено свойство Sorted, то автоматически поддерживается сортировка списка). Переносимые элементы удаляются из первого списка. Вторая *кнопка* реализует операцию переноса всех элементов списка. Направление переноса - из левого списка в правый и обратно - задается заголовками (">", ">>") или ("<", "<<"), изображенными на *кнопках*. Заголовки меняются автоматически в *обработчиках события* Enter, возникающих при входе в левый или правый списки - listBox1 или listBox2. Еще две командные *кнопки*, как следует из их заголовков, предназначены для закрытия *формы* с сохранением или без сохранения результатов работы пользователя. Таково общее описание шаблона. А теперь рассмотрим реализацию. Начнем с *обработчиков события* Enter наших списков:

```
private void listBox1_Enter(object sender, System.EventArgs e)
{
   /*** Событие Enter у списка возникает при входе в список ***/
   button1.Text = ">"; button2.Text = ">>";
```

```
}
private void listBox2_Enter(object sender,
   System.EventArgs e)
{
   /*** Событие Enter у списка возникает при входе в список ***/
   button1.Text = "<"; button2.Text = "<<";
}
```

Посмотрим, как устроены *обработчики события* `Click` для командных *кнопок*, осуществляющих перенос данных между списками:

```
private void button1_Click(object sender, System.EventArgs e)
{
   /* Обработчик события Click кнопки "> <"
       * Выборочный обмен данными между списками
       * ListBox1 <-> ListBox2*****************/
   if(button1.Text == ">")
      MoveSelectedItems(listBox1, listBox2);
   else
      MoveSelectedItems(listBox2, listBox1);
}
private void button2_Click(object sender, System.EventArgs e)
{
   /* Обработчик события Click кнопки ">> <<"
       * Перенос всех данных одного списка в конец другого списка
       * ListBox1 <-> ListBox2*****************/
   if(button2.Text == ">>")
      MoveAllItems(listBox1, listBox2);
   else
      MoveAllItems(listBox2, listBox1);
}
```

Обработчики событий устроены достаточно просто - они вызывают соответствующий метод, передавая ему нужные аргументы в нужном порядке. Рассмотрим метод, переносящий множество отобранных пользователем элементов из одного списка в другой:

```
private void MoveSelectedItems(ListBox list1, ListBox list2)
{
   /*** Выделенные элементы списка list1 ****
    *** помещаются в конец списка List2 *****
    *** и удаляются из списка list1 ********/
   list2.BeginUpdate();
   foreach (object item in list1.SelectedItems)
   {
      list2.Items.Add(item);
   }
   list2.EndUpdate();
   ListBox.SelectedIndexCollection indeces = list1.SelectedIndices;
   list1.BeginUpdate();
   for (int i = indeces.Count -1; i>=0 ; i--)
   {
      list1.Items.RemoveAt(indeces[i]);
   }
   list1.EndUpdate();
}
```

Некоторые комментарии к этому тексту. Заметьте, для добавления выделенных пользователем элементов к другому списку используется коллекция `SelectedItems` и метод `Add`, поочередно добавляющий элементы коллекции. Метод AddRange для добавления всей коллекции здесь не проходит:

```
list2.Items.AddRange(list1.SelectedItems);
```

поскольку нет автоматического преобразования между коллекциями `ObjectCollection` и `SelectedObjectCollection`.

Для удаления выделенных элементов из списка `list1` используется коллекция индексов. Обратите внимание, при удалении элемента с заданным индексом из любой коллекции индексы оставшихся элементов автоматически пересчитываются. Поэтому удаление элементов происходит в обратном порядке, начиная с последнего, что гарантирует корректность оставшихся индексов.

Намного проще устроен метод, переносящий все элементы списка:

```
private void MoveAllItems(ListBox list1, ListBox list2)
{
   /*** Все элементы списка list1 ****
    **** переносятся в конец списка list2 ****
    **** список list1 очищается ************/
   list2.Items.AddRange(list1.Items);
   list1.Items.Clear();
}
```

Добавим еще одну функциональную возможность - разрешим переносить элементы из одного списка в другой двойным щелчком *кнопки* мыши. Для этого зададим *обработчики события* `DoubleClick` наших списков:

```
private void listBox1_DoubleClick(object sender,
   System.EventArgs e)
{
   /* Обработчик события DoubleClick левого списка
       * Выбранный элемент переносится в правый список
       * ListBox1 <-> ListBox2*****************/
   MoveSelectedItems(listBox1, listBox2);
}
private void listBox2_DoubleClick(object sender,
   System.EventArgs e)
{
   /* Обработчик события DoubleClick правого списка
          * Выбранный элемент переносится в левый список
          * ListBox1 <-> ListBox2*****************/
   MoveSelectedItems(listBox2, listBox1);
   }
```

Обработчики вызывают уже рассмотренные нами методы.

На этом закончим рассмотрение функциональности проектируемого образца *формы*. Но, скажете вы, остался не заданным целый ряд вопросов: непонятно, как происходит заполнение списков, как сохраняются элементы после завершения переноса, *обработчики события* `Click` для двух оставшихся *кнопок* не определены. Ничего страшного. Сделаем нашу *форму* родительской, возложив решение оставшихся вопросов на потомков, пусть каждый из них решает эти вопросы по-своему.

Наследование форм

Для объектного программиста *форма* - это обычный *класс*, а населяющие ее *элементы управления* - это *поля класса*. Так что создать новую *форму* - новый *класс*, наследующий все *поля*, методы и события уже существующей *формы* - не представляет никаких проблем. Достаточно написать, как обычно, одну строку:

```
public class NewForm : InterfacesAndDrawing.TwoLists
```

Нужно учитывать, что имени класса родителя должно предшествовать имя пространства имен.

Чаще всего, наследуемые *формы* создаются в режиме проектирования при выборе *пункта меню* Add Inherited Form. (Добраться до этого пункта можно двояко. Можно выбрать пункт Project/ AddInheritedForm из *главного меню* либо выбрать имя проекта в окне проекта и выбрать пункт Add/Add Inherited Form из *контекстного меню*, открывающегося при щелчке правой *кнопкой*.)

В результате открывается окно Inheritance Picker, в котором можно выбрать родительскую *форму*. Заметьте, родительская *форма* может принадлежать как текущему, так и любому другому проекту. Единственное ограничение - проект, содержащий родительскую *форму*, должен быть скомпилирован как exe или dll. Вот как выглядит окно для задания родительской *формы*.

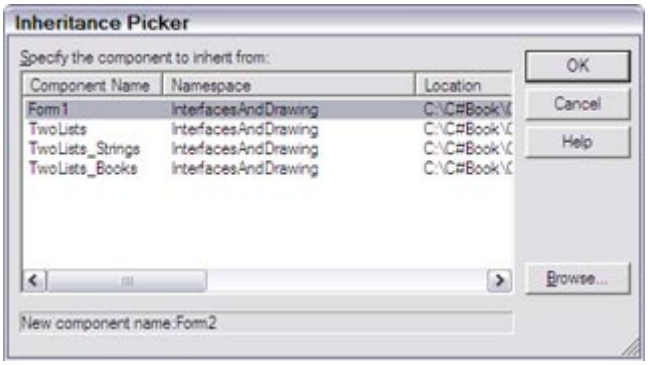

Рис. 24.5. Окно Inheritance Picker наследования форм

При наследовании *форм* следует обратить внимание на *модификаторы доступа элементов управления* родительской *формы*. По умолчанию они имеют статус `private`, означающий запрет на изменение свойств и *обработчиков событий* этих элементов. Чаще всего, такая концепция не верна. Мы не можем знать причин, по которым наследникам захочется изменить созданные родителем свойства элементов. Правильным решением является изменение значения модификатора для всех *элементов управления* родительской *формы* на `protected`. У всех элементов родительской *формы* есть свойство `modifiers`, в котором можно указать статус *элемента управления*, что и было сделано для всех элементов нашего шаблона - *формы* `TwoLists`.

Наследованную *форму* можно затем открыть в дизайнере *форм*, добавить в нее новые элементы и новые *обработчики событий* или изменить установки наследуемых элементов, если родительская *форма* предоставила такую возможность. (Хочу предупредить об одном возможном "жучке", связанном с наследованием *форм*. На одном из моих компьютеров установлена ОС Windows 2000, на другом - Windows XP. Так вот, в Windows 2000 дизайнер отказывается открывать наследуемую *форму*, хотя она создается и нормально работает. Это происходит как для Visual Studio 2003, так и для beta2 Visual Studio 2005. В Office XP все работает нормально. Не могу утверждать совершенно определенно, что это "жучок", поскольку не проводил тщательного исследования. Но полагаю, что предупредить о такой ситуации полезно.)

Два наследника формы TwoLists

Построим по указанной технологии двух наследников *формы* `TwoLists`. Дадим им имена: `TwoLists_Strings` и `TwoLists_Books`. Они будут отличаться тем, что первый из них будет заполнять левый список строками, а второй - "настоящими объектами" *класса* Book. Второй список при открытии *форм* будет оставаться пустым и служить для хранения выбора, сделанного пользователем. Оба наследника будут также задавать

обработчики события `Click` для командных *кнопок*, завершающих работу с этими *формами*. На рис. 24.6 показана наследуемая *форма*, открытая в дизайнере *форм*.

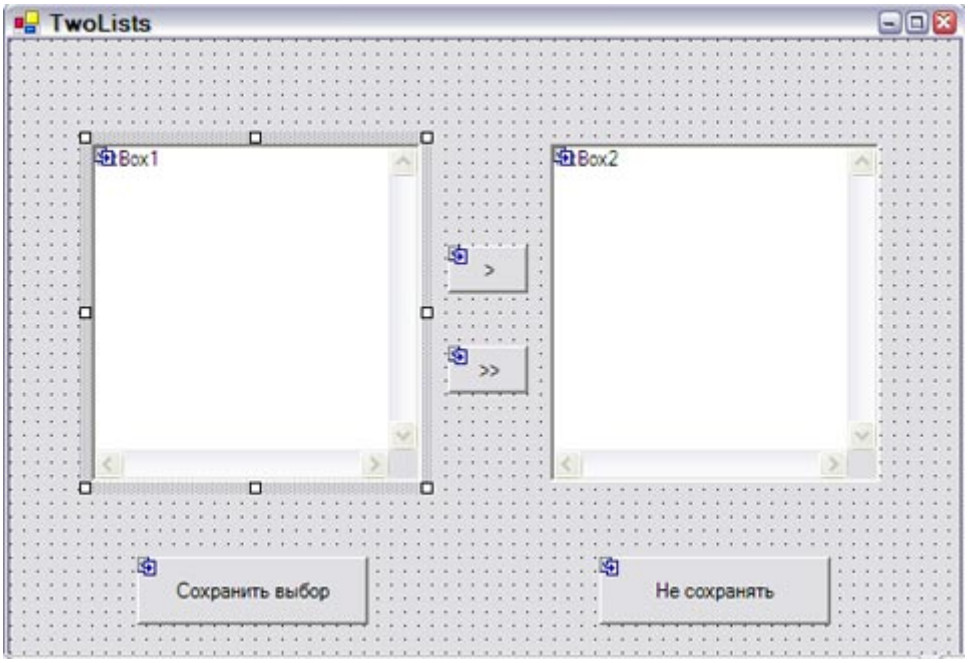

Рис. 24.6. Наследуемая форма, открытая в дизайнере

Обратите внимание на значки, сопровождающие все наследуемые *элементы управления*. В *классе* `TwoLists_Strings` добавлены *поля*:

```
string[] source_items;
string[] selected_items;
const int max_items = 20;
```

В конструктор *класса* добавлен код, инициализирующий массивы:

```
source_items = new string[max_items];
selected_items = new string[max_items];
InitList1();
```

Вызываемый в *конструкторе* закрытый метод *класса* `InitList` заполняет массив `source_items` - источник данных - строками, а затем передает эти данные в левый список *формы*. По-хорошему, следовало бы организовать заполнение списка *формы* из базы данных, но я здесь выбрал самый примитивный способ:

```
void InitList1()
{
   //задание элементов источника и инициализация списка формы
   source_items[0] ="Бертран Мейер:   Методы программирования";
   //аналогично заполняются другие элементы массива
   //перенос массива в список ListBox1
   int i = 0;
   while (source_items[i] != null)
   {
      this.listBox1.Items.Add(source_items[i]);       i++;
   }
   //this.listBox1.DataSource = source_items;
}
```

Закомментирована альтернативная возможность заполнения списка *формы*, использующая свойство `DataSource`. Когда *форма* откроется, ее левый список будет

заполнен, пользователь сможет выбрать из списка понравившиеся ему книги и перенести их в правый список. Зададим теперь *обработчики события* `Click` для командных *кнопок* ("Сохранить выбор" и "Не сохранять"):

```
private void button3_Click(object sender, System.EventArgs e)
{
   int i =0;
   foreach(string item in listBox2.Items)
   {
      selected_items[i] = item;
      Debug.WriteLine(selected_items[i]);
      i++;
   }
   this.Hide();
}
private void button4_Click(object sender, System.EventArgs e)
{
   foreach(string item in listBox2.Items)
   {
      Debug.WriteLine(item);
   }
   this.Hide();
}
```

Оба они в Debug-версии проекта выводят данные о книгах, выбранных пользователем, и скрывают затем *форму*. Но первый из них сохраняет результаты выбора в поле `selected_items`.

Второй наследник `TwoLists_Books` устроен аналогично, но хранит в списке не строки, а объекты *класса* `Book`. Приведу уже без комментариев соответствующие фрагменты кода:

```
Book[] source_items;
Book[] selected_items;
const int max_items = 20;
```

Код, добавляемый в *конструктор*:

```
source_items = new Book[max_items];
   selected_items = new Book[max_items];
   InitList1();
```

Метод `InitList1` скорректирован для работы с книгами:

```
void InitList1()
{
   //задание элементов источника и инициализация списка формы
   Book newbook;
   newbook = new Book("Бертран Мейер",
         "Методы программирования",3,1980);
   source_items[0] =newbook;
   //остальные элементы массива заполняются аналогичным
   //образом
   //перенос массива в список ListBox1
   int i = 0;
   while (source_items[i] != null)
   {
      this.listBox1.Items.Add(source_items[i]);
      i++;
   }
}
```

Обработчики событий `Click` командных *кнопок*, завершающих работу с *формой*, имеют вид:

```
private void button3_Click(object sender, System.EventArgs e)
{
   int i =0;
   foreach(object item in listBox2.Items)
   {
      selected_items[i] = (Book)item;
      selected_items[i].PrintBook();
      i++;
   }
   this.Hide();
}
private void button4_Click(object sender, System.EventArgs e)
{
   Book book;
   foreach(object item in listBox2.Items)
   {
      book = (Book)item;
      book.PrintBook();
   }
   this.Hide();
}
```

Класс `Book` определен следующим образом:

```
public class Book
{
   //поля
   string author, title;
   int price, year;
   public Book(string a, string t, int p, int y)
   {
      author = a; title = t; price = p; year = y;
   }
   public override string ToString()
   {
      return( title + " : " + author);
   }
   public void PrintBook()
   {
      Debug.WriteLine("автор:" + author + " название: " +
         title + " цена: " + price.ToString() +"
            год издания: " + year.ToString());
   }
}
```

Обратите внимание, что в *классе*, как и положено, переопределен метод `ToString`, который задает строку, отображаемую в списке.

В завершение проекта нам осталось спроектировать *главную форму*. Сделаем ее в соответствии с описанным ранее шаблоном кнопочной *формой* (рис. 24.7).

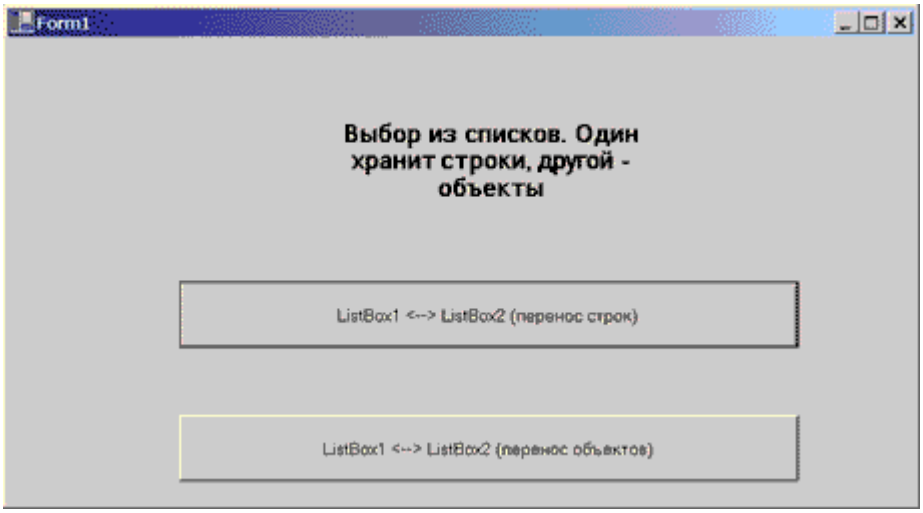

Рис. 24.7. Главная кнопочная форма проекта

Обработчики событий `Click` вызывают соответствующую *форму* для работы либо со списком, хранящим строки, либо со списком, хранящим объекты. На рис. 24.8 показана *форма*, хранящая строки, в процессе работы с ней.

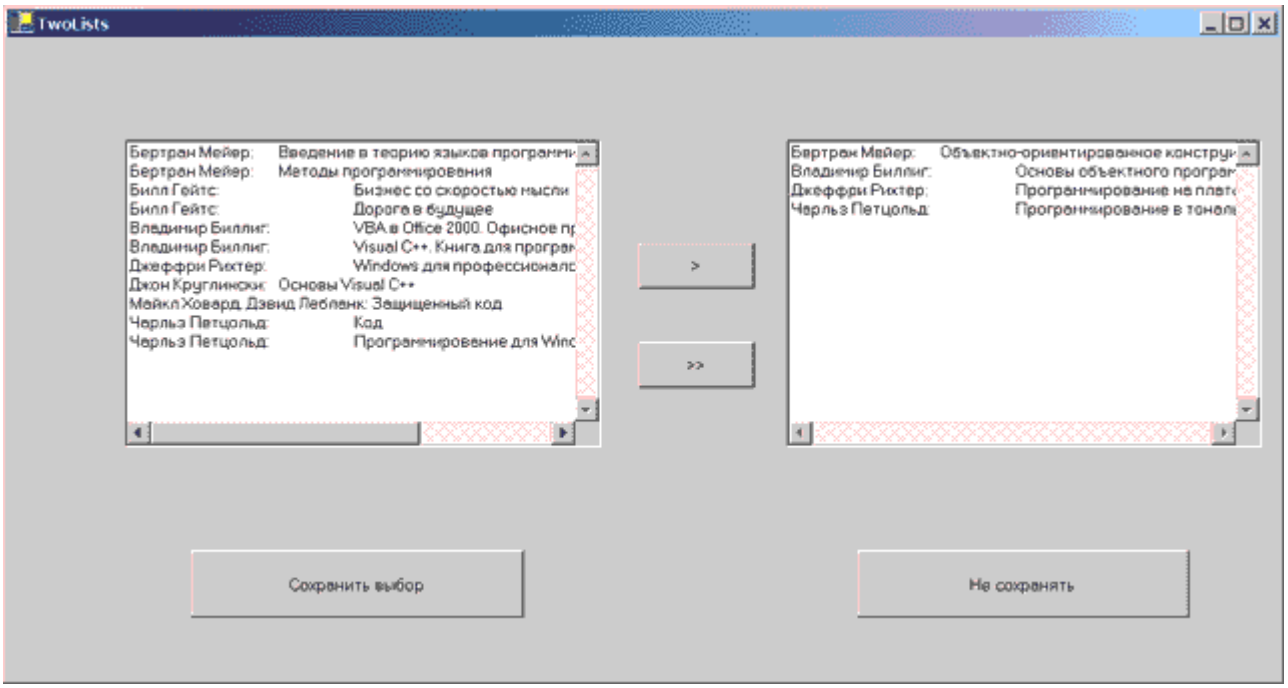

Рис. 24.8. Форма TwoLists_Strings в процессе работы

Ограничение меню в формах

Важными атрибутами интерфейса являются *меню* и *инструментальные панели* **с** *кнопками*. Рассмотрим, как организуются эти элементы интерфейса в *формах*. *Меню* и панели с *кнопками* можно создавать как вручную в режиме проектирования, так и программно.

Несколько слов о терминологии. Когда мы говорим о *меню*, то имеем в виду некоторую структуру, организованную в виде дерева. *Меню* состоит из *элементов меню*, часто называемых ***пунктами меню***. Каждый пункт - *элемент меню* - может быть либо *меню* (*подменю*), состоящим из пунктов, либо быть конечным *элементом меню* - *командой*, при выборе которой выполняются определенные действия. ***Главным меню*** называется строка, содержащая *элементы меню* верхнего уровня и обычно появляющаяся в

вершине окна приложения - в нашем случае, в вершине *формы*. Как правило, *главное меню* всегда видимо, и только оно видимо всегда. Можно из *главного меню* выбрать некоторый элемент, и, если он не задает *команду*, под ним появятся *пункты меню*, заданные этим элементом - говорят, что появляется выпадающее *меню*. Поскольку каждый из пунктов выпадающего *меню* может быть тоже *меню*, то при выборе этого пункта соответствующее выпадающее *меню* появляется слева или справа от него.

Кроме структуры, заданной *главным меню*, в *форме* и в *элементах управления* разрешается организовывать *контекстные меню*, появляющиеся (всплывающие) при нажатии правой *кнопки* мыши.

Создание меню в режиме проектирования

Для построения в режиме проектирования *главного меню* и связанной с ним структуры достаточно перетащить на *форму элемент управления*, называемый `MainMenu`. (В Visual Studio 2005 *элемент управления* для создания *меню* называется `MenuStrip`, а для создания *инструментальных панелей* - ToolStrip.)

После перетаскивания метка с изображением этого *элемента управления* появляется ниже *формы*, а на *форме* появляется *элемент меню* с информационным полем, в котором можно задать название *пункта меню*, и двумя указателями на правого брата и старшего сына, позволяющими перейти к следующему *пункту меню* того же уровня или опуститься на нижний уровень. Технология создания *меню* вручную интуитивно ясна и не вызывает обычно никаких проблем. На рис. 24.9 показан процесс создания *меню*.

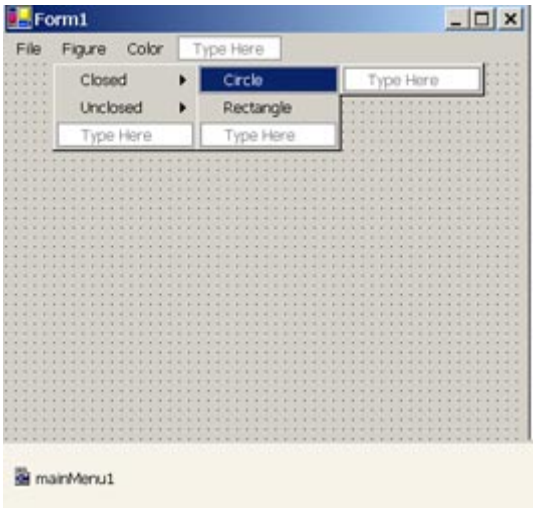

Рис. 24.9. Создание меню в режиме проектирования

Рассмотрим пример, в котором *главное меню* содержит 3 пункта - `File`, `Figure`, `Color`. *Меню* `File` содержит две *команды* - `Open` и `Save`. *Меню* `Figure` состоит из двух пунктов - `Closed` и `Unclosed`, первый из которых содержит две *команды* - `Circle` и `Rectangle`, второй содержит одну *команду* - `Line`. Пункт `Color` *главного меню* в данном случае является *командой* и не содержит выпадающего *меню*. Полагаю, что для демонстрации возможностей этой структуры вполне достаточно. Создать ее вручную - минутное дело. Содержательный пример появится в следующей заключительной главе, а в этой ограничимся демонстрационной версией.

Посадим на *форму* еще один *элемент управления* - текстовое окно - и свяжем с *командами меню обработчики события* `Click`. Для *команд* `Open`, `Save` и `Color`, имеющих общепринятый смысл, обработчики будут открывать соответствующие этим *командам* диалоговые окна, позволяющие в диалоге с пользователем открыть файл, сохранить файл и выбрать подходящий цвет. Диалоговые окна - это важный элемент организации интерфейса, который, пользуясь случаем, хочется продемонстрировать.

Связывание *команды меню* с *обработчиком события* в режиме проектирования выполняется стандартным образом - выделяется соответствующая *команда меню*, затем в окне `Properties` щелкается значок молнии и из списка событий выбирается событие `Click`, после чего открывается заготовка *обработчика события*, заполняемая нужным кодом.

Вот как выглядят *обработчики события* `Click` *команд* `Open`, `Save` и `Color`:

```
private void menuItem4_Click(object sender, System.EventArgs e)
{
   OpenFileDialog openFileDialog1 = new OpenFileDialog();
   openFileDialog1.ShowDialog();
   //код, показывающий, что делать с открытым файлом
   textBox1.Text = "Открытие Файла!";
}
private void menuItem10_Click(object sender, System.EventArgs e)
{
   SaveFileDialog saveFileDialog1 = new SaveFileDialog();
   saveFileDialog1.ShowDialog();
   //код, анализирующий результат операции сохранения файла
   textBox1.Text = "Сохранение Файла!";
}
private void menuItem3_Click(object sender, System.EventArgs e)
{
   ColorDialog colorDialog1 = new ColorDialog();
   if (colorDialog1.ShowDialog()== DialogResult.OK)
      this.textBox1.BackColor =colorDialog1.Color;
}
```

На рис. 24.10 показано диалоговое окно для выбора цвета, открытое при выборе *команды* `Color`.

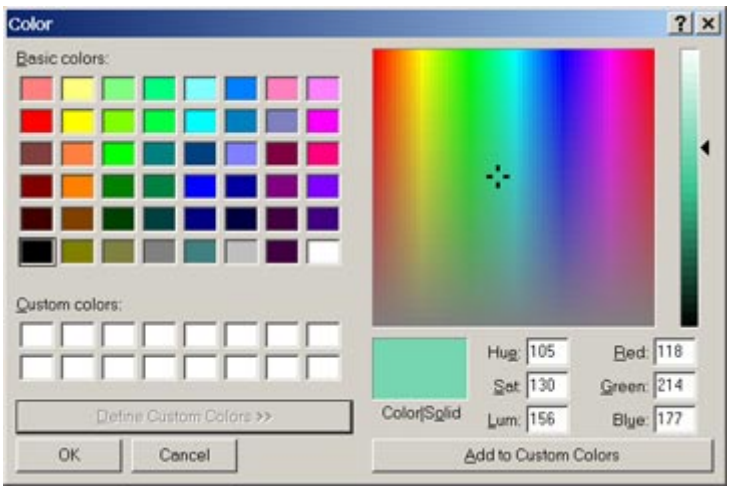

Рис. 24.10. Диалоговое окно ColorDialog, позволяющее выбрать цвет

Для полноты картины зададим *обработчики событий* для *команд меню* `Circle`, `Rectangle`, `Line`, не выполняющие пока содержательной работы, а лишь информирующие о намерениях:

```
private void menuItem7_Click(object sender, System.EventArgs e)
{
   textBox1.Text = "Рисование круга!";
}
private void menuItem8_Click(object sender, System.EventArgs e)
{
   textBox1.Text = "Рисование прямоугольника!";
```

```
}
private void menuItem9_Click(object sender, System.EventArgs e)
{
   textBox1.Text = "Рисование прямой!";
}
```

Закончу на этом рассмотрение процесса создания *меню* в режиме проектирования, опуская ряд деталей, например, возможность задания горячих клавишей для *элементов меню*.

Классы меню

Все, что можно делать руками, можно делать программно. Рассмотрим *классы*, используемые при работе с *меню*. Основным родительским *классом* является *абстрактный класс Menu*, задающий базовую функциональность трех своих потомков - *классов* `MainMenu`, `ContextMenu` и `MenuItem`. *Класс* `MenuItem` задает *элемент меню*, который, напомню, сам может являться *меню* (*подменю*). Свойство `MenuItems`, которым обладают все *классы меню*, возвращает коллекцию `MenuItems` из *элементов меню класса* `MenuItem`. С коллекцией можно работать обычным образом. Создание *меню* означает создание объектов контейнерных *классов* `MainMenu` и `ContextMenu` и множества объектов *класса* `MenuItem`. Последние добавляются в коллекцию либо контейнерных *классов*, либо в коллекцию соответствующих элементов `MenuItem`. Созданные объекты *классов* `MainMenu` и `ContextMenu` связываются со свойствами *формы* - `Menu` и `ConextMenu`. Проанализируем код, созданный в процессе проектирования Дизайнером Меню и Дизайнером Формы для нашего примера.

Вот какие поля *формы*, задающие объекты *меню*, были сформированы:

```
private System.Windows.Forms.MainMenu mainMenu1;
private System.Windows.Forms.MenuItem menuItem1;
//другие элементы меню
private System.Windows.Forms.MenuItem menuItem10;
```

Основной код, создаваемый дизайнерами, помещается в метод `InitializeComponent`. Приведу лишь фрагменты этого кода:

```
this.mainMenu1 = new System.Windows.Forms.MainMenu();
this.menuItem1 = new System.Windows.Forms.MenuItem();
...
   // mainMenu1
this.mainMenu1.MenuItems.AddRange(new
   System.Windows.Forms.MenuItem[]
{this.menuItem1,this.menuItem2,this.menuItem3});
   // menuItem1
   this.menuItem1.Index = 0;
   this.menuItem1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[]
                     {this.menuItem4,this.menuItem10});
   this.menuItem1.Text = "File";
...
// menuItem4
this.menuItem4.Index = 0;
this.menuItem4.Text = "Open";
this.menuItem4.Click += new System.EventHandler(this.menuItem4_Click);
...
// Form1
...
this.Controls.AddRange(new System.Windows.Forms.Control[] {
   this.textBox1});
this.Menu = this.mainMenu1;
this.Name = "Form1";   this.Text = "Form1";
```

Надеюсь, что данный программный код прозрачен и не требует дополнительных комментариев.

Создание инструментальной панели с командными кнопками

Панель с командными *кнопками* дополняет *меню*. Панель устроена проще, поскольку здесь нет иерархии. На панели располагаются *кнопки*, щелчок по каждой из которых запускает на выполнение соответствующую *команду*, заданную *обработчиком события* `Click`. Как правило, *команды*, задаваемые *кнопками* панелей, соответствуют наиболее часто используемым *командам меню* и являются альтернативным способом их запуска. Но это не обязательно, и *команды*, задаваемые *кнопками* панели, могут не пересекаться с *командами меню*.

Роль контейнерного *класса* для командных *кнопок* играет *класс*, определяющий панель - ToolBar. Командные *кнопки* - элементы, располагаемые на панели, - задаются *классом* `ToolBarButton`.

Давайте спроектируем панель с тремя *кнопками*, задающими *команды* `Open`, `Save` и `Color`, повторяющие *команды меню*. Принято *кнопки* делать красивыми, вынося на них рисунки, ассоциированные с *командами*. Поэтому посадим на *форму* два *элемента управления* - `ImageList`, хранящий рисунки, связываемые с *кнопками*, и `ToolBar` - панель, на которой будут располагаться *кнопки*. В коллекцию объекта `imageList1` добавим три подходящие картинки, свяжем этот объект со свойством `ImageList` объекта `toolBar1`. Затем добавим три *кнопки* в коллекцию объекта `toolBar1` и зададим для них нужные свойства - текст, появляющийся на *кнопке*, подсказку к *кнопке* и индекс элемента из списка `ImageList`. На рис. 24.11 показан процесс задания *кнопки* и установки ее свойств.

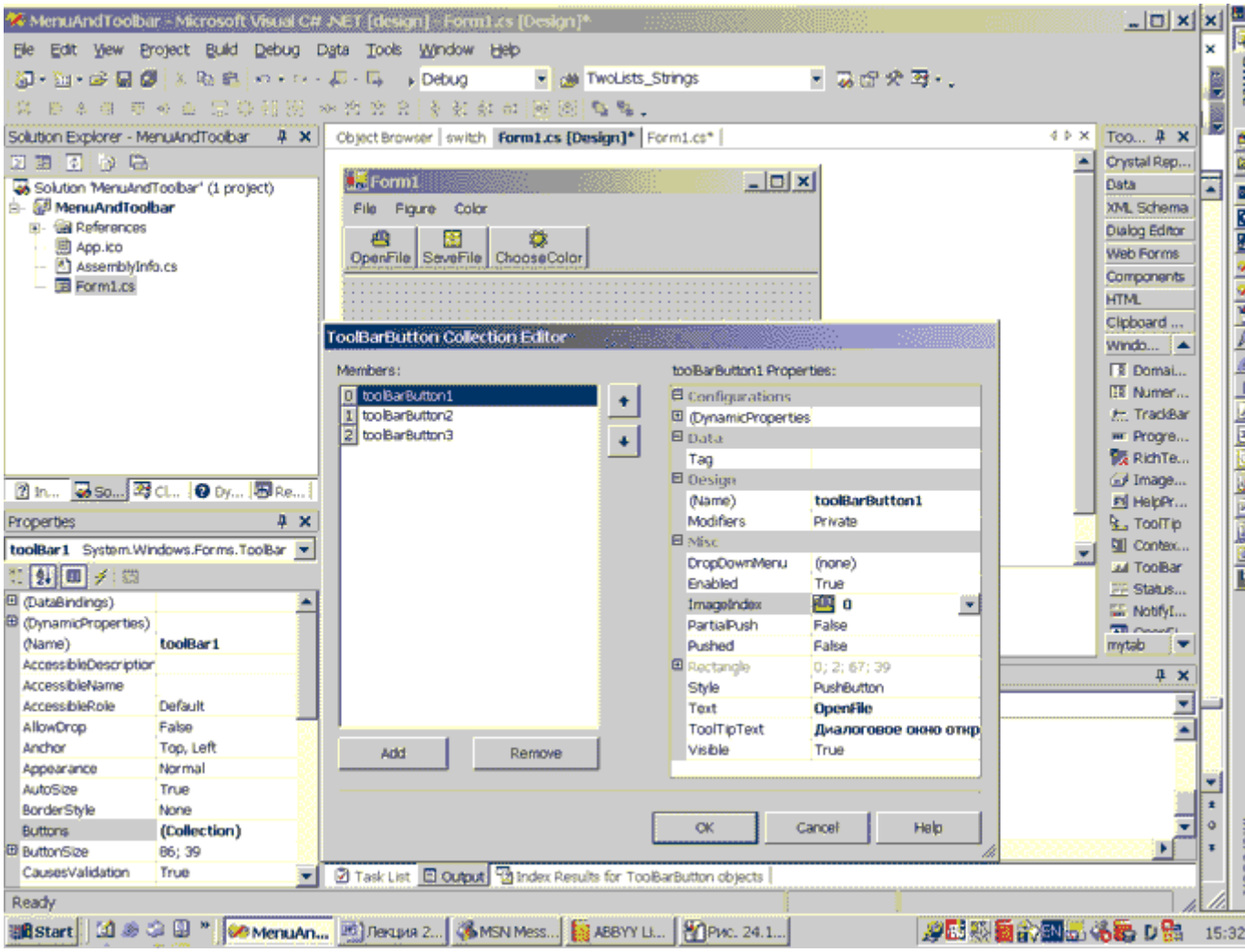

Рис. 24.11. Проектирование панели с командными кнопками

Проанализируем теперь созданный дизайнером программный код. Как всегда начнем с *полей класса*, хранящих созданные в процессе проектирования элементы:

```
private System.Windows.Forms.ToolBar toolBar1;
private System.Windows.Forms.ImageList imageList1;
private System.Windows.Forms.ToolBarButton toolBarButton1;
private System.Windows.Forms.ToolBarButton toolBarButton2;
private System.Windows.Forms.ToolBarButton toolBarButton3;
```

В методе `InitializeComponent` эти объекты создаются и инициализируются:

```
this.toolBar1 = new System.Windows.Forms.ToolBar();
this.imageList1 = new System.Windows.Forms.ImageList(this.components);
this.toolBarButton1 = new System.Windows.Forms.ToolBarButton();
this.toolBarButton2 = new System.Windows.Forms.ToolBarButton();
this.toolBarButton3 = new System.Windows.Forms.ToolBarButton();
...
// toolBar1
this.toolBar1.Buttons.AddRange(new System.Windows.Forms.ToolBarButton[]
   {this.toolBarButton1, this.toolBarButton2,this.toolBarButton3});
   this.toolBar1.DropDownArrows = true;
   this.toolBar1.ImageList = this.imageList1;
   this.toolBar1.Name = "toolBar1";
   this.toolBar1.ShowToolTips = true;
   this.toolBar1.Size = new System.Drawing.Size(432, 42);
```

```
   this.toolBar1.TabIndex = 1;
   this.toolBar1.ButtonClick +=
      new System.Windows.Forms.ToolBarButtonClickEventHandler(
         this.toolBar1_ButtonClick);
   // toolBarButton1
   this.toolBarButton1.ImageIndex = 0;
   this.toolBarButton1.Text = "OpenFile";
   this.toolBarButton1.ToolTipText =
      "Диалоговое окно открытия файла";
...
```

Этот текст должен быть понятен без комментариев, а вот об *обработчике события* `Click` стоит сказать несколько слов. Во-первых, событие `Click` не связывается с каждой командной *кнопкой*, расположенной на панели, - оно связано с самой панелью. Так что в обработчике происходит разбор случаев с анализом того, какая *кнопка* была нажата. Вот как это делается:

```
private void toolBar1_ButtonClick(object sender,
System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
   int buttonNumber = toolBar1.Buttons.IndexOf(e.Button);
   switch (buttonNumber)
   {
      case 0:
         OpenFileDialog openFileDialog1 = new OpenFileDialog();
         openFileDialog1.ShowDialog();
         //код, показывающий, что делать с открытым файлом
         textBox1.Text = "Открытие Файла!";
         break;
      case 1:
         SaveFileDialog saveFileDialog1 = new SaveFileDialog();
         saveFileDialog1.ShowDialog();
         //код, анализирующий результат операции сохранения файла
         textBox1.Text = "Сохранение Файла!";
         break;
      default:
         ColorDialog colorDialog1 = new ColorDialog();
         if (colorDialog1.ShowDialog()== DialogResult.OK)
            this.textBox1.BackColor =colorDialog1.Color;
         break;
   }
}
```

В заключение взгляните на спроектированную *форму* с *меню* и панелью с командными *кнопками*.

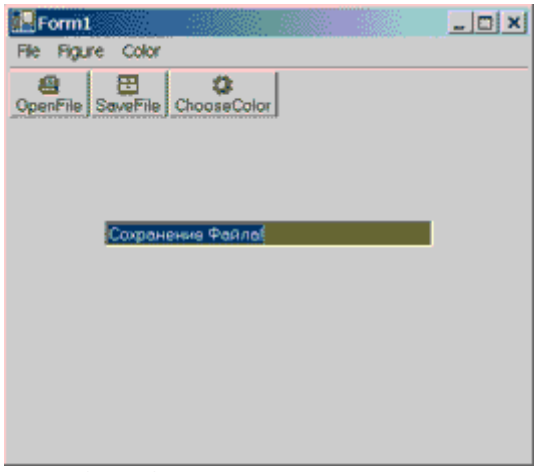

Рис. 24.12. Форма с меню и инструментальной панелью

Рисование в форме

Графика необходима при организации пользовательского интерфейса. Образы информативнее текста. Framework .Net реализует расширенный графический интерфейс GDI+, обладающий широким набором возможностей. Но для *рисования в формах* достаточно иметь три объекта - перо, кисть и, хочется сказать, бумагу, но третий нужный объект - это объект *класса Graphics*, методы которого позволяют в *формах* заниматься *графикой* - рисовать и раскрашивать.

Класс Graphics

Класс Graphics - это основной *класс*, необходимый для *рисования*. *Класс Graphics*, так же, как и другие рассматриваемые здесь *классы* для перьев и кистей, находятся в пространстве имен `Drawing`, хотя *классы* некоторых кистей вложены в подпространство `Drawing2D`.

Объекты этого *класса* зависят от контекста устройства, (*графика* не обязательно отображается на дисплее компьютера, она может выводиться на принтер, графопостроитель или другие устройства), поэтому создание объектов *класса Graphics* выполняется не традиционным способом - без вызова *конструктора класса*. Создаются объекты специальными методами разных *классов*. Например, метод `CreateGraphics` *класса* `Control` - наследника *класса* `Form` - возвращает объект, ассоциированный с выводом *графики* на *форму*.

При *рисовании в формах* можно объявить в *форме* поле, описывающее объект *класса Graphics*:

```
Graphics graph;
```

а в *конструкторе формы* произвести связывание с реальным объектом:

```
graph = CreateGraphics();
```

Затем всюду в программе, где нужно работать с *графикой*, используется глобальный для *формы* объект `graph` и его методы. Есть другой способ получения этого объекта - обработчики некоторых событий получают объект *класса Graphics* среди передаваемых им аргументов. Например, в *обработчике события Paint*, занимающегося перерисовкой, этот объект можно получить так:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   Graphics gr = e.Graphics;
   //перерисовка, использующая методы объекта gr
}
```

Для получения этого объекта можно использовать и статические методы самого *класса Graphics*.

Методы класса Graphics

У *класса Graphics* большое число методов и свойств. Упомяну лишь о некоторых из них. Группа статических методов *класса* позволяет создать объект этого *класса*, задавая например описатель (handle) контекста устройства.

Для *рисования* наиболее важны три группы методов. К первой относится перегруженный метод `DrawString`, позволяющий выводить *тексты в графическом режиме*. Вторую группу составляют методы `Draw` - `DrawEllipse`, `DrawLine`, `DrawArc` и

другие, позволяющие цветным пером (объектом *класса Pen*) рисовать *геометрические фигуры*: линии, различные кривые, прямоугольники, многоугольники, эллипсы и прочее. К третьей группе относятся методы `Fill` - `FillEllipse`, `FillPie`, `FillRectangle` и другие, позволяющие *нарисовать и закрасить фигуру* кистью. Кисти (объекты *классов*, производных от *Brush*), могут быть разные - сплошные, узорные, градиентные.

Класс Pen

Методам группы `Draw` *класса Graphics*, рисующим контур фигуры, нужно передать перо - объект *класса Pen*. В *конструкторе* этого *класса* можно задать цвет пера и его толщину (чаще говорят "ширину пера"). Цвет задается объектом *класса* (структурой) `Color`. Для выбора подходящего цвета можно использовать упоминавшееся выше диалоговое окно `Color` либо одно из многочисленных статических свойств *класса* `Color`, возвращающее требуемый цвет. Возможно и непосредственное задание элементов структуры в виде комбинации RGB - трех цветов - красного, зеленого и голубого. Вместо создания нового пера с помощью *конструктора* можно использовать специальный *класс* предопределенных системных перьев.

Класс Brush

Класс Brush, задающий кисти, устроен более сложно. Начну с того, что *класс Brush* является абстрактным *классом*, так что создавать кисти этого *класса* нельзя, но можно создавать кисти классов-потомков *Brush*. Таких *классов* пять - они задают кисть:

- `SolidBrush` - для сплошной закраски области заданным цветом;
- `TextureBrush` - для закраски области заданной картинкой (image);
- `HatchBrush` - для закраски области предопределенным узором;
- `LinearGradientBrush` - для сплошной закраски с переходом от одного цвета к другому, где изменение оттенков задается линейным градиентом;
- `PathGradientBrush` - для сплошной закраски с переходом от одного цвета к другому, где изменение оттенков задается более сложным путем.

Первые два *класса* кистей находятся в пространстве имен `System.Drawing`, остальные - в `System.Drawing.Drawing2D`.

У каждого из этих *классов* свои *конструкторы*. В примере, обсуждаемом далее, рассмотрим создание кистей трех разных *классов*, там и поговорим о *конструкторах классов*.

Проект "Паутина Безье, кисти и краски"

Построим проект для *рисования в формах*. В одной из *форм* будем рисовать пером, в другом - кистями различного типа. *Главную форму* сделаем простой кнопочной *формой*. Вот как она выглядит.

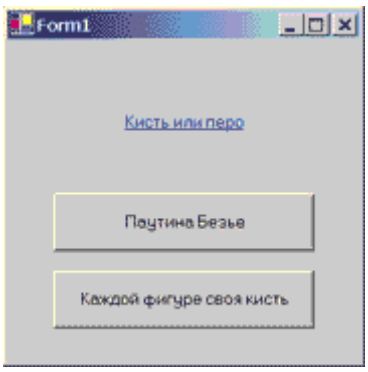

Рис. 24.13. Кнопочная форма "кисть или перо"

Выбор соответствующей командной *кнопки* открывает *форму* для *рисования* пером или кистью.

Паутина Безье

В *форме* BezierWeb будем рисовать несколько *кривых Безье*, исходящих из одной точки - центра. Положение центра определяется курсором. Перемещая мышь, меняем положение курсора, а, следовательно, и центра, так что рисунок в *форме* будет все время перерисовываться, следуя за мышью. (***кривые Безье*** - это широко используемый в *графике* и технических приложениях вид гладких кривых. *Кривая Безье* задается четырьмя точками, первая и последняя из которых являются начальной и конечной точками кривой. Две оставшиеся точки являются точками притяжения. Прямую, заданную началом и концом, они притягивают к себе, превращая ее в гладкую кривую. Строгое математическое определение несложно, но мы приводить его не будем.)

Прежде чем рассмотреть программный код, давайте посмотрим, как выглядят нарисованные программой *кривые Безье*, исходящие из одной точки.

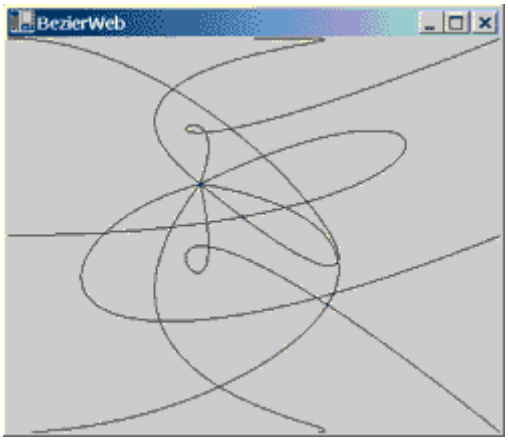

Рис. 24.14. Паутина Безье

Перейдем к рассмотрению кода. Первым делом добавим в поля *формы* нужные нам объекты:

```
//fields
      Point center;
      Point[] points = new Point[10];
      Pen pen;
      Graphics graph;
      int count;
```

Точка center будет задавать общую начальную точку для всех рисуемых *кривых Безье*, массив `points` будет задавать остальные точки, используемые при построении *кривых Безье*. О роли объектов `pen` и `graph`, необходимых при *рисовании*, уже говорилось. Объект `count` играет техническую роль, о которой скажу чуть позже, прямого отношения к *рисованию* он не имеет.

В *конструкторе формы* вызывается метод `MyInit`, инициализирующий введенные объекты:

```
void MyInit()
{
   int cx = ClientSize.Width;
   int cy = ClientSize.Height;
   points[0] = new Point(0,0);
   points[1] = new Point(cx/2,0);
   points[2] = new Point(cx,0);
   points[3] = new Point(0,cy/2);
   points[4] = new Point(cx,cy/2);
   points[5] = new Point(0,cy);
   points[6] = new Point(cx/2,cy);
   points[7] = new Point(cx,cy);
   points[8] = new Point(0,0);
   points[9] = new Point(cx/2,0);
   graph = this.CreateGraphics();
   center = new Point(cx/2,cy/2);
   count =1;
}
```

Рисование кривых Безье выполняется в методе `DrawWeb`, устроенном очень просто. В цикле рисуется 8 кривых, используя точку `center` и массив `points`:

```
void DrawWeb()
{
   for (int i = 0; i<8; i++)
      graph.DrawBezier(pen,center,points[i],points[i+2],
         points[i+1]);
}
```

Метод `DrawBezier`, вызываемый объектом `graph` *класса Graphics*, принадлежит группе рассмотренных нами методов `Draw`. Первым аргументом у всех этих методов является объект *класса Pen*, а остальные зависят от типа рисуемой фигуры. Для *кривой Безье*, как уже говорилось, необходимо задать четыре точки.

Главный вопрос, требующий решения: где же вызывать сам метод `DrawWeb`, где инициализировать *рисование в форме*? Будем вызывать этот метод в двух местах - в двух *обработчиках событий*. Поскольку нам хочется реализовать стратегию, по которой точка `center` будет следовать за курсором мыши, то естественно, чтобы *рисование* инициировалось *обработчиком события* `MouseMove` нашей *формы* `BezierWeb`. (Напомню, для подключения события *формы* или *элемента управления* достаточно в режиме проектирования выбрать нужный элемент, в окне свойств этого элемента щелкнуть по значку с изображением молнии и из списка возможных событий данного элемента выбрать нужное, что приведет к созданию заготовки *обработчика событий*.)

Вот текст обработчика этого события:

```
private void BezierWeb_MouseMove(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   pen = SystemPens.Control;
```

```
    DrawWeb();
    center.X = e.X; center.Y = e.Y;
    //pen = new Pen(Color.Aquamarine);
    pen = SystemPens.ControlText;
    DrawWeb();
}
```

Метод `DrawWeb` вызывается дважды - первый раз с пером цвета фона, другой - с цветом, принятым системой для отображения текста. Обратите внимание, для создания нужного пера в данном случае не вызывается *конструктор класса*, а используется *класс* предопределенных системных перьев. Оператор, создающий объект `pen` с помощью *конструктора*, закомментирован. Он может использоваться, если нужно рисовать кривые определенным цветом.

Перед *рисованием* кривых цветом переднего плана общая для всех кривых точка `center` получает координаты курсора мыши, передаваемые аргументом *обработчика события*.

Событие Paint

Вызов метода `DrawWeb` добавлен еще и в *обработчик события Paint*:

```
protected override void OnPaint(System.Windows.Forms.PaintEventArgs e)
{
    pen = SystemPens.ControlText;
    DrawWeb();
    Debug.WriteLine(count++);
}
```

Говоря о *рисовании*, нельзя не упомянуть о *событии Paint*. Оно возникает всякий раз, когда область, в которой происходило *рисование*, повреждена. Причины этого могут быть разные - пользователь свернул *форму*, изменил ее размеры, произошло перекрытие другой *формой*, был вызван метод `Invalidate` - во всех этих случаях требуется перерисовать область. Тогда-то и возникает *событие Paint*, в задачу его обработчика входит перерисовка поврежденной области. Первый раз *событие Paint* возникает при открытии *формы*. Переменная `count`, введенная нами, позволяет в режиме отладки подсчитывать число вызовов *события Paint*.

Событие Paint подключают обычно не так, как это делалось, например, для события `MouseMove`. Вместо этого переопределяют родительский метод `OnPaint`. (Как переопределяются родительские методы группы `On`, занимающиеся обработкой событий, другие методы *классов* родителей и базовых интерфейсов? В режиме проектирования в окне *классов*, отражающем структуру *класса*, нужно выбрать соответствующий *класс* (в нашем случае *класс формы* BezierWeb), раскрыть узел BasesAndInterfaces этого *класса* и из появившегося списка всех наследованных свойств и методов выбрать нужный (в нашем случае метод `OnPaint`). В результате появится заготовка для переопределения метода.)

В данном контексте перерисовка сводится, как это обычно делается, к вызову метода, выполняющего *рисование*. Для повышения эффективности можно анализировать поврежденную область и выполнять *рисование* только в ее пределах.

Закончим на этом с *рисованием* пером и перейдем к рассмотрению *рисования* кистью.

Кисти и краски

Создадим в нашем проекте новую *форму* `RandomShapes`, в которой будем рисовать и закрашивать *геометрические фигуры* трех разных типов - эллипсы, сектора, прямоугольники. Для каждого типа фигуры будем использовать свой тип кисти: эллипсы будем закрашивать градиентной кистью, сектора - сплошной, а прямоугольники - узорной. Цвет фигуры, ее размеры и положение будем выбирать случайным образом. *Рисование* фигур будет инициироваться в *обработчике события* `Click`. При каждом щелчке *кнопкой* мыши на *форме* будут рисоваться три новых экземпляра фигур каждого типа. В отличие от *кривых Безье*, старые фигуры стираться не будут.

На рис. 24.15 показана *форма* после нескольких щелчков *кнопки* мыши. Конечно, черно-белый рисунок в книге не может передать цвета, особенно смену оттенков для градиентной кисти. На экране дисплея или цветном рисунке все выглядит красивее.

А теперь приведем программный код, реализующий *рисование*. Начнем, как обычно, с *полей класса*:

```
//fields
   int cx,cy;
   Graphics graph;
   Brush brush;
   Color color;
   Random rnd;
```

Инициализация полей производится в методе `MyInit`, вызываемом *конструктором класса*:

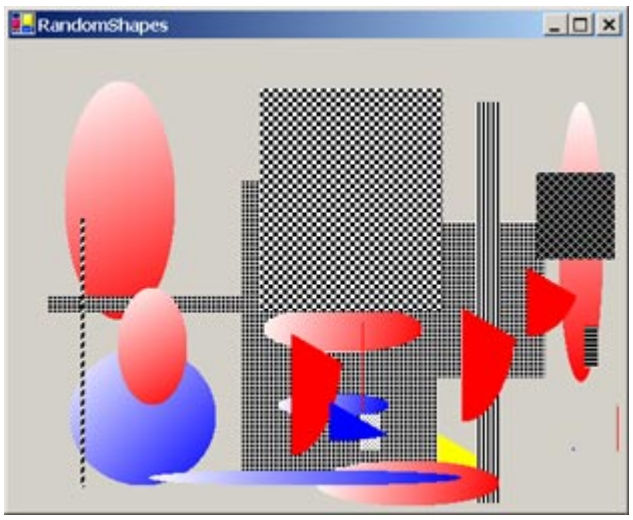

Рис. 24.15. Рисование кистями разного типа

```
void MyInit()
   {
      cx = ClientSize.Width;
      cy = ClientSize.Height;
      graph = CreateGraphics();
      rnd = new Random();
   }
```

Рассмотрим теперь основной метод, реализующий *рисование* фигур различными кистями:

```
void DrawShapes()
   {
      for(int i=0; i<3; i++)
```

```
        {
            //выбирается цвет - красный, желтый, голубой
            int numcolor = rnd.Next(3);
            switch (numcolor)
            {
                case 0:
                    color = Color.Blue; break;
                case 1:
                    color = Color.Yellow; break;
                case 2:
                    color = Color.Red; break;
            }
            //градиентной кистью рисуется эллипс,
            //местоположение случайно
            Point top = new Point(rnd.Next(cx), rnd.Next(cy));
            Size sz = new Size(rnd.Next(cx-top.X), rnd.Next(cy-top.Y));
            Rectangle rct = new Rectangle(top, sz);
            Point bottom = top + sz;
            brush = new LinearGradientBrush(top, bottom,
                Color.White,color);
            graph.FillEllipse(brush,rct);
            //сплошной кистью рисуется сектор,
            //местоположение случайно
            top = new Point(rnd.Next(cx), rnd.Next(cy));
            sz = new Size(rnd.Next(cx-top.X), rnd.Next(cy-top.Y));
            rct = new Rectangle(top, sz);
            brush = new SolidBrush(color);
            graph.FillPie(brush,rct,30f,60f);
            //узорной кистью рисуется прямоугольник,
            //местоположение случайно
            top = new Point(rnd.Next(cx), rnd.Next(cy));
            sz = new Size(rnd.Next(cx-top.X), rnd.Next(cy-top.Y));
            rct = new Rectangle(top, sz);
            HatchStyle hs = (HatchStyle)rnd.Next(52);
            brush = new HatchBrush(hs,Color.White, Color.Black);
            graph.FillRectangle(brush,rct);
        }
    }
```

Приведу некоторые комментарии в дополнение к тем, что встроены в текст метода. Здесь многое построено на работе со случайными числами. Случайным образом выбирается один из возможных цветов для *рисования* фигуры, ее размеры и положение. Наиболее интересно рассмотреть создание кистей разного типа. Когда создается градиентная кисть.

```
brush = new LinearGradientBrush(top, bottom, Color.White,color);
```

то нужно в *конструкторе* кисти задать две точки и два цвета. Точки определяют интервал изменения оттенков цвета от первого до второго. В начальной точке имеет место первый цвет, в конечной - второй, в остальных точках - их комбинация. Разумно, как это сделано у нас, в качестве точек выбирать противоположные углы прямоугольника, ограничивающего рисуемую фигуру.

Наиболее просто задается сплошная кисть:

```
brush = new SolidBrush(color);
```

Для нее достаточно указать только цвет. Для узорной кисти нужно задать предопределенный тип узора, всего их возможно 52. В нашем примере тип узора выбирается случайным образом:

```
HatchStyle hs = (HatchStyle)rnd.Next(52);
brush = new HatchBrush(hs,Color.White, Color.Black);
```

Помимо первого аргумента, задающего тип узора, указываются еще два цвета - первый определяет цвет повторяющегося элемента, второй - цвет границы между элементами узора.

Непосредственное *рисование* кистью осуществляют методы группы `Fill`:

```
graph.FillEllipse(brush,rct);
graph.FillPie(brush,rct,30f,60f);
graph.FillRectangle(brush,rct);
```

Первый аргумент всегда задает кисть, а остальные зависят от типа рисуемой фигуры. Как правило, всегда задается прямоугольник, ограничивающий данную фигуру.

Вызов метода `DrawShapes`, как уже говорилось, встроен в *обработчик события* `Click` *формы* `RandomShapes`:

```
private void RandomShapes_Click(object sender, System.EventArgs e)
{
   DrawShapes();
}
```

На этом поставим точку в рассмотрении данной темы. По сути, этим завершается и наш учебный курс. В последней лекции будет рассмотрен некоторый заключительный проект.

25. Лекция: Финальный проект

В этой заключительной лекции новый материал появляться не будет, не будет и традиционных вопросов в конце лекции. Лекция особенная - она посвящена описанию финального проекта, в котором объединены многие, надеюсь, уже хорошо знакомые элементы.

В финальном проекте создается семейство классов, описывающих геометрические фигуры. Проектирование начинается с абстрактного класса поведения, который описывает общие свойства и методы, характерные для всех фигур семейства. Затем, используя наследование, создаются классы конкретных геометрических фигур, начиная с простейших, таких, как круги и прямоугольники, до составных, таких, как класс Person. Мы добавим в наш проект динамические структуры данных, такие, как список с курсором, для хранения в нем фигур семейства. Наконец, мы создадим интерфейс, включающий меню с десятками команд и панель с инструментальными кнопками. Интерфейс позволяет конечному пользователю выполнять различные действия над геометрическими фигурами - создавать, рисовать их на форме, перемещать их с помощью команд и перетаскивать их мышью, менять их размеры и цвет, сохранять в списке и удалять из списка, отображать все фигуры списка или очистить его полностью.

Проект может служить образцом полноценного Windows-приложения, примером проектирования в классах с демонстрацией преимуществ, предоставляемых наследованием. Закончим на этом рекламную часть и приступим к делу. Хочу предупредить, вас ждут программные тексты, почти без всяких комментариев. Все нужные комментарии были даны в предыдущих лекциях. С моей точки зрения, наиболее интересная часть программистских книжек - это та, в которой приводится программный код. И значит, эта глава самая интересная.

Абстрактный класс Figure

Приведем код класса:

```
using System;
using System.Drawing;
namespace Shapes
{
/// <summary>
/// Figure - это абстрактный класс; прародитель семейства
/// классов геометрических фигур. Все фигуры имеют:
/// центр - center, масштаб - scale. статус
/// перетаскивания - dragged center - объект встроенного
/// класса (структуры) Point. Этот объект задает характерную
/// точку фигуры - чаще всего ее центр (тяжести)
/// scale задает масштаб фигуры, первоначально единичный.
/// drugged = true, когда фигура следует за курсором мыши.
/// над фигурами определены операции: параллельный
/// перенос - Move(a,b) масштабирование - Scale(s)
/// Показ фигуры - Show. Область захвата - Region_Capture
/// возвращает прямоугольник, характерный для фигуры,
/// перетаскивание фигуры возможно при установке курсора
/// мыши в области захвата.
/// </summary>
abstract public class Figure
{
   /// <summary>
   /// закрытые для клиентов атрибуты класса - center, scale
   /// </summary>
   protected Point center;
   protected double scale ;
```

```
protected bool dragged;
protected Color color;
//Доступ к свойствам
public Point center_figure
{
   get{return(center);}
   get {center = value;}
}
public double scale_figure
{
   get{return(scale);}
   get {scale = value;}
}
public bool dragged_figure
{
   get{return(dragged);}
   get {dragged = value;}
}
public Color color_figure
{
   get{return(color);}
   set {color = value;}
}
/// <summary>
/// базовый конструктор фигур
/// </summary>
/// <param name="x">координата X характерной точки
///фигуры</param>
/// <param name="y">Координата Y характерной точки
///фигуры</param>
public Figure(int x, int y)
{
   center = new Point(x,y);
   scale = 1;
   dragged = false;
   color = Color.ForestGreen;
}
/// <summary>
/// отложенный метод
/// Параллельный перенос фигуры на (a,b)
/// require : true;
/// ensure : для любой точки фигуры p(x,y):
/// x = old(x) +a; y = old(y) + b;
/// </summary>
/// <param name="a">a - перемещение по горизонтали
///вправо </param>
/// <param name="b">b - перемещение по вертикали
///вниз</param>
/// Замечание: Для того, чтобы фигура при рисовании была
/// полностью видимой, координаты всех ее точек должны
/// быть в пределах области рисования.
public void Move (int a,int b)
{
   center.X +=a; center.Y += b;
}
/// <summary>
/// изменяет масштаб фигуры
/// </summary>
/// <param name="s">масштаб изменяется в s раз</param>
public void Scale(double s)
{
   scale*=s;
}
/// <summary>
```

```
   /// рисование фигуры в окне, передающем объекты g и pen
   /// </summary>
   /// <param name="g"> графический объект, методы которого
   /// рисуют фигуру</param>
   /// <param name="pen">перо рисования</param>
   public abstract void Show(Graphics g, Pen pen,
      Brush brush);
   public abstract System.Drawing.Rectangle Region_Capture();
}
```

Абстрактный класс, относящийся к этапу проектирования системы, вместе с тем является важнейшим элементом заключительного семейства классов. В этом проявляется мощь объектно-ориентированного подхода к разработке программных систем. Заметьте, на данном уровне большая часть текста представляет документацию, являющуюся неотъемлемой частью программного проекта. Документация записана в тегах `<summary>`, что позволяет автоматически ее извлечь и сохранить в виде XML-отчета.

Классы семейства геометрических фигур

Приведем теперь программные коды классов, являющихся потомками класса `Figure`.

Класс Ellipse

Вот программный код этого класса:

```
using System;
using System.Drawing;
namespace Shapes
{
   /// <summary>
   /// Класс Ellipse - потомок класса Figure.
   /// </summary>
   public class Ellipse: Figure
   {
      int axisA,axisB;
      Rectangle rect;
      public Ellipse(int A, int B, int x, int y ): base(x,y)
      {
         axisA = A; axisB = B;
         rect =Init();
      }
      public override void Show(Graphics g, Pen pen, Brush brush)
      {
         rect = Init();
         g.DrawEllipse(pen,rect);
         g.FillEllipse(brush, rect);
      }
      public override Rectangle Region_Capture()
      {
         rect = Init();
         return rect;
      }
      Rectangle Init()
      {
         int a  =Convert.ToInt32(axisA*scale);
         int b =Convert.ToInt32(axisB*scale);
         int leftupX = center.X - a;
         int leftupY = center.Y - b;
         return( new Rectangle(leftupX,leftupY,2*a,2*b));
      }
```

```
    }
}
```

Класс Circle

Этот класс является потомком класса `Ellipse`:

```
using System;
using System.Drawing;
namespace Shapes
{
    /// <summary>
    /// Класс Circle - потомок класса Ellipse.
    /// </summary>
    public class Circle: Ellipse
    {
        public Circle( int radius,int x,
            int y):base(radius,radius,x,y)
        {
        //Круг - это эллипс с равными полуосями (радиусом круга)
        }
    }
}
```

Здесь опять-таки проявляется мощь наследования. Потомок наследует все свойства и методы родителя. Ему остается только указать собственный конструктор объектов класса, да и в нем главное состоит в вызове конструктора родительского класса с передачей ему нужных аргументов.

Класс LittleCircle

Этот класс, задающие маленькие кружочки фиксированного радиуса, в свою очередь, является наследником класса `Circle`. Устроен он также чрезвычайно просто:

```
using System;
namespace Shapes
{
    /// <summary>
    /// Класс LittleCircle - потомок класса Circle.
    /// </summary>
    public class LittleCircle:Circle
    {
        public LittleCircle(int x,int y): base(4,x,y)
        {
            // маленький круг радиуса 4
        }
    }
}
```

Класс Rect

Этот класс является еще одним прямым потомком класса `Figure`:

```
using System;
using System.Drawing;
namespace Shapes
{
    /// <summary>
    /// Класс Rect - потомок класса Figure.
    /// </summary>
```

```
   public class Rect:Figure
   {
      int sideA, sideB;
      Rectangle rect;
      public Rect(int sideA,int sideB, int x, int y): base(x,y)
      {
         this.sideA = sideA; this.sideB = sideB;
         rect =Init();
      }
      public override void Show(Graphics g, Pen pen, Brush brush)
      {
         rect = Init();
         g.DrawRectangle(pen,rect);
         g.FillRectangle(brush,rect);
      }
      public override Rectangle Region_Capture()
      {
         rect = Init();
         return rect;
      }
      Rectangle Init()
      {
         int a  =Convert.ToInt32(sideA*scale);
         int b =Convert.ToInt32(sideB*scale);
         int leftupX = center.X - a/2;
         int leftupY = center.Y - b/2;
         return( new Rectangle(leftupX,leftupY,a,b));
      }
   }
}
```

Класс Square

Квадрат - это частный случай прямоугольника. Соответствующий класс является потомком класса `Rect`:

```
using System;
namespace Shapes
{
   /// <summary>
   /// Класс Square - потомок класса Rect.
   /// </summary>
   public class Square:Rect
   {
      public Square(int side, int x, int y): base(side,side,x,y)
      {
         //квадрат - это прямоугольник с равными сторонами
      }
   }
}
```

Класс Person

Этот класс является прямым потомком класса `Figure`. Вместе с тем, класс является клиентом трех других классов семейства - `Circle`, `Rect` и `LittleCircle`, поскольку элементы фигуры, составляющие человечка, являются объектами этих классов%

```
namespace Shapes
{
   /// <summary>
   /// Класс Person - потомок класса Figure,
```

```
   /// клиент классов Circle, Rect, LittleCircle.
   /// </summary>
   public class Person:Figure
   {
      int head_h;
      Circle head;
      Rect body;
      LittleCircle nose;
      public Person(int head_h, int x, int y): base(x,y)
      {
         //head_h - радиус головы, x,y - ее центр.
         //остальные размеры исчисляются относительно
         //размера головы.
         this.head_h = head_h;
         head = new Circle(head_h,x,y);
         int body_x = x;
         int body_y = y + 3*head_h;
         int body_w =2*head_h;
         int body_h = 4*head_h;
         body = new Rect(body_w, body_h, body_x,body_y);
         nose = new LittleCircle(x+head_h +2, y);
      }
      public override void Show(System.Drawing.Graphics g,
         System.Drawing.Pen pen, System.Drawing.Brush brush)
      {
         int h = Convert.ToInt32(head_h*scale);
         //head
         int top_x = center.X - h;
         int top_y = center.Y - h;
         g.DrawEllipse(pen, top_x,top_y, 2*h,2*h);
         g.FillEllipse(brush, top_x,top_y, 2*h,2*h);
         //body
         top_y += 2*h;
         g.DrawRectangle(pen, top_x,top_y, 2*h,4*h);
         g.FillRectangle(brush, top_x,top_y, 2*h,4*h);
         //nose
         top_y -=h;
         top_x += 2*h;
         g.DrawEllipse(pen, top_x,top_y, 8,8);
         g.FillEllipse(brush, top_x,top_y, 8,8);
      }
      public override System.Drawing.Rectangle
         Region_Capture()
      {
         int h = Convert.ToInt32(head_h*scale);
         int top_x = center.X - h;
         int top_y = center.Y - h;
         return new
            System.Drawing.Rectangle(top_x,top_y,2*h,2*h);
      }
   }
}
```

Список с курсором. Динамические структуры данных

Добавим в проект классы, задающие динамические структуры данных. Конечно, можно было бы воспользоваться стандартными... Но для обучения крайне полезно уметь создавать собственные классы, задающие такие структуры данных. Список с курсором - один из важнейших образцов подобных классов%:

```
using System;
namespace Shapes
```

```
{
   /// <summary>
   /// Класс TwoWayList(G)  описывает двусвязный список с
   /// курсором. Элементами списка являются объекты
   /// TwoLinkable, хранящие, помимо указателей на двух
   /// преемников, объекты типа G.Курсор будет определять    ///       текущий
(активный) элемент списка. Класс будет
   /// определять симметричные операции по отношению к
   /// курсору.
   /// Конструкторы:
   /// Конструктор без параметров будет создавать пустой
   /// список
   /// Запросы:
   /// empty: require: true; возвращает true для
   /// непустого списка item: require: not empty();
   /// возвращает активный элемент типа G; count:
   /// require: true; возвращает число элементов списка;
   /// count in[0,n]
   /// (count == 0) eqviv empty();
   /// index: require: not empty(); возвращает индекс
   /// активного элемента.
   /// search_res: require: true; возвращает true,
   /// если последний поиск был успешным.
   /// Команды:
   /// put_left(elem): require: true;
   /// ensure: добавить новый элемент (elem) слева от курсора;
   /// put_right(elem): require: true;
   /// ensure: добавить новый элемент (elem) справа от
   /// курсора;
   /// remove: require: not empty();
   /// ensure: удалить активный элемент;
   /// особо обрабатывается удаление последнего и
   /// единственного элементов
   /// операции с курсором:
   /// start: require: true;
   /// ensure: сделать активным первый элемент;
   /// finish: require: true;
   /// ensure: сделать активным последний элемент;
   /// go_prev: require: not (index = 1);
   /// ensure: сделать активным предыдущий элемент;
   /// go_next: require: not (index = count);
   /// ensure: сделать активным последующий элемент;
   /// go_i(i): require:  (i in [1, count]);
   /// ensure: сделать активным элемент с индексом i;
   /// операции поиска:
   /// search_prev(elem): require: not (index = 1);
   /// ensure: сделать активным первый элемент elem
   /// слева от курсора;
   /// Успех или неуспех поиска сохранять в булевской
   /// переменной search_res
   /// search_next: require: not (index = count);
   ///  ensure: сделать активным первый элемент elem
   /// справа от курсора;
   /// успех или неуспех поиска сохранять в булевской
   /// переменной search_res
   /// </summary>
   public class TwoWayList
   {
      public TwoWayList()
      {
         first = cursor = last = null;
         count = index = 0;
         search_res = false;
      }//конструктор
```

```
/// <summary>
/// first, cursor, last - ссылки на первый,
/// активный и последний элементы списка
/// Запросы count, index search_res также
/// реализуются атрибутами.
/// Запросы empty, item реализуются функциями
/// </summary>
protected TwoLinkable first, cursor, last;
protected int count, index;
protected bool search_res;
//доступ на чтение к закрытым свойствам;
public int Count
{
   get
   {
      return(count);
   }
}
public int Index
{
   get
   {
      return(index);
   }
}
public bool Search_res
{
   get
   {
      return(search_res);
   }
}
/// <summary>
/// require: true; возвращает true для непустого списка
/// </summary>
/// <returns></returns>
public bool empty()
{
   return(first == null);
}//empty
/// <summary>
/// require: not empty(); возвращает активный
/// элемент типа G;
/// </summary>
/// <returns></returns>
public Figure item()
{
   return(cursor.Item);
}//item
/// <summary>
/// require: true;
/// ensure: добавить новый элемент (elem) слева
/// от курсора;
/// </summary>
/// <param name="elem">Тип Figure играет роль
/// родового типа G
/// хранимого элемента elem</param>
public void put_left(Figure elem)
{
   TwoLinkable newitem = new TwoLinkable();
   newitem.Item = elem;
   newitem.Next = cursor;
   if (empty())  //список пуст
   {
```

```
      first = cursor = last = newitem;
      index =1; count = 1;
   }
   else
   {
      if (index == 1)
         first =newitem;
      else
         cursor.Prev.Next = newitem;
      newitem.Prev = cursor.Prev; cursor.Prev = newitem;
      count++; index++;
   }
}//put_right
/// <summary>
/// require: true;
/// ensure: добавить новый элемент (elem) справа
/// от курсора;
/// </summary>
/// <param name="elem">Тип Figure играет роль
/// родового типа G
/// хранимого элемента elem</param>
public void put_right(Figure elem)
{
   TwoLinkable newitem = new TwoLinkable();
   newitem.Item = elem;
   newitem.Prev = cursor;
   if (empty())  //список пуст
   {
      first = cursor = last = newitem;
      index =1; count = 1;
   }
   else
   {
      if (index == count)
         last =newitem;
      else
         cursor.Next.Prev = newitem;

      newitem.Next = cursor.Next; cursor.Next = newitem;
      count++;
   }
}//put_right
public void remove()
{
   if(count == 1)
   {
      first = last = cursor = null;
      index=0;
   }
   else if(index==1)
   {
      first = cursor.Next;
      cursor.Prev = null;
      cursor = cursor.Next;
   }
   else if(index == count)
   {
      last = cursor.Prev;
      cursor.Next = null;
      cursor = cursor.Prev;
      index--;
   }
   else
   {
```

```
      cursor.Prev.Next = cursor.Next;
      cursor.Next.Prev = cursor.Prev;
      cursor = cursor.Next;
   }
   count--;
}//remove
/// операции с курсором:
/// <summary>
/// start: require: true;
/// ensure: сделать активным первый элемент;
/// </summary>
public void start()
{
   cursor = first; index = 1;
}//start
/// <summary>
/// finish: require: true;
/// ensure: сделать активным последний элемент;
/// </summary>
public void finish()
{
   cursor = last; index = count;
}//finish
/// <summary>
/// go_prev: require: not (index = 1);
/// ensure: сделать активным предыдущий элемент;
/// </summary>
public void go_prev()
{
   cursor = cursor.Prev; index--;
}// go_prev
/// <summary>
/// go_next: require: not (index = count);
/// ensure: сделать активным последующий элемент;
/// </summary>
public void go_next()
{
   cursor = cursor.Next; index++;
}// go_next
/// <summary>
/// go_i(i): require:  (i in [1, count]);
/// ensure: сделать активным элемент с индексом i;
/// </summary>
/// <param name="i"></param>
public void go_i(int i)
{
   if(i >index)
      while (i>index)
      {
         cursor = cursor.Next; index++;
      }
   else if(i<index)
      while (i<index)
      {
         cursor = cursor.Prev; index--;
      }
}// go_i
/// операции поиска:
/// <summary>
/// search_prev(elem): require: not (index = 1);
/// ensure: сделать активным первый элемент elem
/// слева от курсора;
/// </summary>
/// <param name="elem">искомый элемент</param>
```

```
      public virtual void search_prev(Figure elem)
      {
         bool found = false;
         while (!found && (index !=1))
         {
            cursor = cursor.Prev; index--;
            found = (elem == item());
         }
         search_res = found;
      }// search_prev
   /// <summary>
   /// успех или неуспех поиска сохранять в булевской
   /// переменной search_res
   /// search_next: require: not (index = count);
   /// ensure: сделать активным первый элемент elem
   /// справа от курсора;
   /// успех или неуспех поиска сохранять в булевской
   /// переменной search_res
   /// </summary>
   /// <param name="elem"></param>
      public virtual void search_next(Figure elem)
      {
         bool found = false;
         while (!found && (index !=count))
         {
            cursor = cursor.Next; index++;
            found = (elem == item());
         }
         search_res = found;
      }//search_next
   }
}
```

Заметьте, класс подробно документирован. Для методов класса указываются предусловия и постусловия. Обратите внимание, в соответствии с принципами контрактного программирования клиент класса, прежде чем вызвать метод, должен проверить выполнимость предусловия, что повышает корректность работы системы в целом. Именно так и будет реализован вызов этих методов в классе формы, где осуществляется работа со списком.

Классы элементов списка

Рассмотрим классы, описывающие элементы списков - элементы с одним и с двумя указателями:

```
using System;
namespace Shapes
{
   /// <summary>
   /// Класс Linkable(T)задает элементы списка,включающие:
   /// информационное поле типа T - item
   /// ссылку на элемент типа Linkable - next
   /// Функции:
   /// конструктор new: -> Linkable
   /// запросы:
   /// Get_Item: Linkable -> T
   /// Get_Next: Linkable -> Linkable
   /// процедуры:
   /// Set_Item: Linkable*T -> Linkable
   /// Set_Next: Linkable*Linkable -> Linkable
   /// Роль типа T играет Figure
   /// </summary>
```

```
public class Linkable
{
   public Linkable()
   {
      item =null; next = null;
   }
   /// <summary>
   /// закрытые атрибуты класса
   /// </summary>
   Figure item;
   Linkable next;
   /// <summary>
   /// процедуры свойства для доступа к полям класса
   /// </summary>
   public Figure Item{
   get{
      return(item);
      }
   set{
      item = value;
      }
   }
   public Linkable Next{
   get{
      return(next);
      }
   set{
      next = value;
      }
   }
}//class Linkable
/// <summary>
/// Класс TwoLinkable задает элементы с двумя ссылками
/// </summary>
public class TwoLinkable
{

   public TwoLinkable()
   {
      prev = next = null;
   }
   /// <summary>
   /// закрытые атрибуты класса
   /// </summary>
   TwoLinkable prev, next;
   Figure item;
   /// <summary>
   /// процедуры свойства для доступа к полям класса
   /// </summary>
   public Figure Item
   {
      get
      {
         return(item);
      }
      set
      {
         item = value;
      }
   }
   public TwoLinkable Next
   {
      get
      {
```

```
            return(next);
         }
         set
         {
            next = value;
         }
      }
      public TwoLinkable Prev
      {
         get
         {
            return(prev);
         }
         set
         {
            prev = value;
         }
      }
   }//class TwoLinkable
}
```

Организация интерфейса

Создадим теперь интерфейс, позволяющий конечному пользователю работать с объектами наших классов. Как всегда, интерфейс создавался вручную в режиме проектирования. На форме я создал меню с большим числом команд и инструментальную панель с 18 кнопками, команды которых повторяли основную команду меню. Описывать процесс создания интерфейса не буду - он подробно рассмотрен в предыдущей главе. Поскольку вся работа по созданию интерфейса транслируется в программный код формы, то просто приведу этот достаточно длинный текст почти без всяких купюр:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using Shapes;

namespace Final
{
   /// <summary>
   /// Эта форма обеспечивает интерфейс для создания,
   /// рисования, показа, перемещения, сохранения в списке
   /// и выполнения других операций над объектами семейства
   /// геометрических фигур. Форма имеет меню и
   /// инструментальные панели.
   /// </summary>
   public class Form1 : System.Windows.Forms.Form
   {
      //fields
      Graphics graphic;
      Brush brush, clearBrush;
      Pen pen, clearPen;
      Color color;
      Figure current;
      TwoWayList listFigure;
      private System.Windows.Forms.MainMenu mainMenu1;
      private System.Windows.Forms.ImageList imageList1;
      private System.Windows.Forms.ToolBar toolBar1;
      private System.Windows.Forms.MenuItem menuItem1;
```

```
// аналогичные определения для других элементов меню
      private System.Windows.Forms.MenuItem menuItem35;
      private System.Windows.Forms.ToolBarButton
         toolBarButton1;
// аналогичные определения для других командных кнопок
      private System.Windows.Forms.ToolBarButton
         toolBarButton18;
      private System.ComponentModel.IContainer components;
      public Form1()
      {
         InitializeComponent();
         InitFields();
      }
      void InitFields()
      {
         graphic = CreateGraphics();
         color = SystemColors.ControlText;
         brush = new SolidBrush(color);
         clearBrush = new SolidBrush(SystemColors.Control);
         pen = new Pen(color);
         clearPen = new Pen(SystemColors.Control);
         listFigure = new Figure_List();
         current = new Person(20, 50, 50);
      }
      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose( bool disposing )
      {
         if( disposing )
         {
            if (components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose( disposing );
      }
      #region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         // Код, инициализирующий компоненты и построенный
         // дизайнером, опущен
      }
      #endregion
      /// <summary>
      /// Точка входа в приложение - процедура Main,
      /// запускающая форму
      /// </summary>
      [STAThread]
      static void Main()
      {
         Application.Run(new Form1());
      }
      private void menuItem7_Click(object sender,
         System.EventArgs e)
      {
         createEllipse();
      }
      void createEllipse()
```

```
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create ellipse
   current = new Ellipse(50, 30, 180,180);
}
private void menuItem8_Click(object sender,
   System.EventArgs e)
{
   createCircle();
}
void createCircle()
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create circle
   current = new Circle(30, 180,180);
}
private void menuItem9_Click(object sender,
   System.EventArgs e)
{
   createLittleCircle();
}
void createLittleCircle()
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create littlecircle
   current = new LittleCircle(180,180);
}
private void menuItem10_Click(object sender,
   System.EventArgs e)
{
   createRectangle();
}
void createRectangle()
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create rectangle
   current = new Rect(50, 30, 180,180);
}
private void menuItem11_Click(object sender,
   System.EventArgs e)
{
   createSquare();
}
void createSquare()
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create square
   current = new Square(30, 180,180);
}
private void menuItem12_Click(object sender,
   System.EventArgs e)
{
   createPerson();
}
```

```
void createPerson()
{
   //clear old figure
   if (current != null) current.Show(graphic,
      clearPen, clearBrush);
   //create person
   current = new Person(20, 180,180);
}
private void menuItem13_Click(object sender,
   System.EventArgs e)
{
   showCurrent();
}
void showCurrent()
{
   //Show current
   current.Show(graphic, pen, brush);
}
private void menuItem14_Click(object sender,
   System.EventArgs e)
{
   clearCurrent();
}
void clearCurrent()
{
   //Clear current
   current.Show(graphic, clearPen, clearBrush);
}
private void menuItem17_Click(object sender,
   System.EventArgs e)
{
   incScale();
}
void incScale()
{
   //Increase scale
   current.Show(graphic, clearPen, clearBrush);
   current.Scale(1.5);
   current.Show(graphic, pen, brush);
}
private void menuItem18_Click(object sender,
   System.EventArgs e)
{
   decScale();
}
void decScale()
{
   //Decrease scale
   current.Show(graphic, clearPen, clearBrush);
   current.Scale(2.0/3);
   current.Show(graphic, pen, brush);
}
private void menuItem19_Click(object sender,
   System.EventArgs e)
{
   moveLeft();
}
void moveLeft()
{
   //Move left
   current.Show(graphic, clearPen, clearBrush);
   current.Move(-20,0);
   current.Show(graphic, pen, brush);
}
```

```
private void menuItem20_Click(object sender,
   System.EventArgs e)
{
   moveRight();
}
void moveRight()
{
   //Move right
   current.Show(graphic, clearPen, clearBrush);
   current.Move(20,0);
   current.Show(graphic, pen, brush);
}
private void menuItem21_Click(object sender,
   System.EventArgs e)
{
   moveTop();
}
void moveTop()
{
   //Move top
   current.Show(graphic, clearPen, clearBrush);
   current.Move(0,-20);
   current.Show(graphic, pen, brush);
}
private void menuItem22_Click(object sender,
   System.EventArgs e)
{
   moveDown();
}
void moveDown()
{
   //Move down
   current.Show(graphic, clearPen, clearBrush);
   current.Move(0, 20);
   current.Show(graphic, pen, brush);
}
private void menuItem23_Click(object sender,
   System.EventArgs e)
{
   //choose color
   ColorDialog dialog = new ColorDialog();
   if (dialog.ShowDialog() ==DialogResult.OK)
      color =dialog.Color;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem24_Click(object sender,
   System.EventArgs e)
{
   //Red color
   color =Color.Red;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem25_Click(object sender,
   System.EventArgs e)
{
   //Green color
   color =Color.Green;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem26_Click(object sender,
   System.EventArgs e)
```

```
{
   //Blue color
   color =Color.Blue;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem27_Click(object sender,
   System.EventArgs e)
{
   //Black color
   color =Color.Black;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem28_Click(object sender,
   System.EventArgs e)
{
   //Gold color
   color =Color.Gold;
   pen = new Pen(color); brush =
      new SolidBrush(color);
}
private void menuItem29_Click(object sender,
   System.EventArgs e)
{
   //put_left: добавление фигуры в список
   listFigure.put_left(current);
}
private void menuItem30_Click(object sender,
   System.EventArgs e)
{
   //put_right: добавление фигуры в список
   listFigure.put_right(current);
}
private void menuItem31_Click(object sender,
   System.EventArgs e)
{
   //remove: удаление фигуры из списка
   if(!listFigure.empty())
      listFigure.remove();
}
private void menuItem32_Click(object sender,
   System.EventArgs e)
{
   goPrev();
}
void goPrev()
{
   //go_prev: передвинуть курсор влево
   if(!(listFigure.Index == 1))
   {
      listFigure.go_prev();
      current = listFigure.item();
   }
}
private void menuItem33_Click(object sender,
   System.EventArgs e)
{
   goNext();
}
void goNext()
{
   //go_next: передвинуть курсор вправо
   if( !(listFigure.Index == listFigure.Count))
```

```
   {
      listFigure.go_next();
      current = listFigure.item();
   }
}
private void menuItem34_Click(object sender,
   System.EventArgs e)
{
   //go_first
   listFigure.start();
   if(!listFigure.empty())
      current = listFigure.item();
}
private void menuItem35_Click(object sender,
   System.EventArgs e)
{
   //go_last
   listFigure.finish();
   if(!listFigure.empty())
      current = listFigure.item();
}
private void menuItem15_Click(object sender,
   System.EventArgs e)
{
   showList();
}
void showList()
{
   //Show List
   listFigure.start();
   while(listFigure.Index <= listFigure.Count)
   {
      current = listFigure.item();
      current.Show(graphic,pen,brush);
      listFigure.go_next();
   }
   listFigure.finish();
}
private void menuItem16_Click(object sender,
   System.EventArgs e)
{
   clearList();
}
void clearList()
{
   //Clear List
   listFigure.start();
   while(!listFigure.empty())
   {
      current = listFigure.item();
      current.Show(graphic,clearPen,clearBrush);
      listFigure.remove();
   }
}
private void Form1_MouseMove(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   if((current != null) && current.dragged_figure)
   {
      current.Show(graphic,clearPen,clearBrush);
      Point pt = new Point(e.X, e.Y);
      current.center_figure = pt;
      current.Show(graphic,pen,brush);
   }
```

```
}
private void Form1_MouseUp(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   current.dragged_figure = false;
}
private void Form1_MouseDown(object sender,
   System.Windows.Forms.MouseEventArgs e)
{
   Point mousePoint = new Point (e.X, e.Y);
   Rectangle figureRect = current.Region_Capture();
   if ((current != null) &&
      (figureRect.Contains(mousePoint)))
      current.dragged_figure = true;
}
protected override void
   OnPaint(System.Windows.Forms.PaintEventArgs e)
{
   //show current figure
   current.Show(graphic, pen, brush);
}
private void toolBar1_ButtonClick(object sender,
   System.Windows.Forms.ToolBarButtonClickEventArgs e)
{
   int buttonNumber =
      toolBar1.Buttons.IndexOf(e.Button);
   switch (buttonNumber)
   {
      case 0:
         createEllipse(); break;
      case 1:
         createCircle(); break;
      case 2:
         createLittleCircle(); break;
      case 3:
         createRectangle(); break;
      case 4:
         createSquare(); break;
      case 5:
         createPerson();  break;
      case 6:
         showCurrent(); break;
      case 7:
         clearCurrent(); break;
      case 8:
         showList();  break;
      case 9:
         clearList();  break;
      case 10:
         incScale();  break;
      case 11:
         decScale();  break;
      case 12:
         moveLeft();  break;
      case 13:
         moveRight();  break;
      case 14:
         moveTop();  break;
      case 15:
         moveDown();  break;
      case 16:
         goNext();  break;
      case 17:
         goPrev();  break;
```

```
            }
        }
    }
}
```

Команд меню и кнопок в нашем интерфейсе много, поэтому много и обработчиков событий, что приводит к разбуханию кода. Но каждый из обработчиков событий довольно прост. Ограничусь кратким описанием главного меню:

- команды пункта главного меню `Create` позволяют создавать геометрические фигуры разных классов;
- команды пункта главного меню `Show` позволяют показать или стереть текущую фигуру или все фигуры, сохраняемые в списке;
- две команды пункта `Scale` позволяют изменить масштаб фигуры (увеличить ее или уменьшить);
- команды пункта `Move` позволяют перемещать текущую фигуру в четырех направлениях;
- команды пункта `Color` позволяют либо задать цвет фигур в диалоговом окне, либо выбрать один из предопределенных цветов;
- группа команд пункта `List` позволяет помещать текущую фигуру в список, перемещаться по списку и удалять из списка ту или иную фигуру;
- командные кнопки инструментальной панели соответствуют наиболее важным командам меню;
- реализована возможность перетаскивания фигур по экрану мышью.

В заключение взгляните, как выглядит форма в процессе работы с объектами:

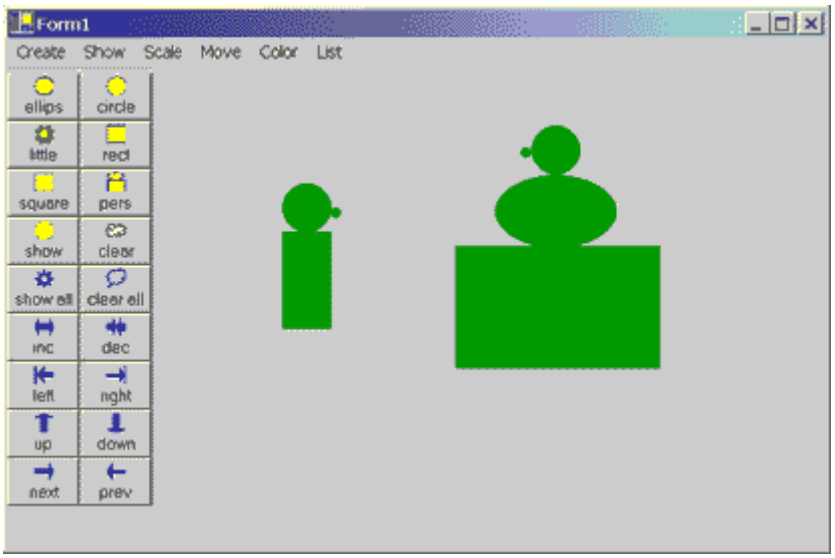

Рис. 25.1. Финальный проект. Форма в процессе работы