МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования
"Казанский (Приволжский) федеральный университет"

Набережночелнинский институт (филиал)

Отделение информационных технологий и энергетических систем

Кафедра информационных систем

ПРОГРАММИРОВАНИЕ

Лекционный материал курса

Мингалеева Лейсэн Башировна
Тазмеев Алмаз Харисович
Галиуллин Ленар Айратович

Казань – 2018

Оглавление

Лекция 1

Тема: Введение в программирование и языки

В этом курсе в качестве языка программирования выбран язык C# и его версия 3.0, в качестве среды разработки программных проектов - Visual Studio 2008, Professional Edition и Framework .Net в версии 3.5.

Язык C# является наиболее известной новинкой в области языков программирования. По сути это язык программирования, созданный уже в 21-м веке. Явившись на свет в недрах Microsoft, он с первых своих шагов получил мощную поддержку. Язык признан международным сообществом. В июне 2006 года Европейской ассоциацией по стандартизации принята уже четвертая версия стандарта этого языка: Standard ECMA-334 C# Language Specifications, 4-th edition - http://www.ecma-international.org/publications/standards/Ecma-334.htm.

Международной ассоциацией по стандартизации эта версия языка узаконена как стандарт ISO/IEC - 23270. Заметим, что первая версия стандарта языка была принята еще в 2001 году. Компиляторы Microsoft строятся в соответствии с международными стандартами языка.

Язык C# является молодым языком и продолжает интенсивно развиваться. Каждая новая версия языка включает принципиально новые свойства. Не стала исключением и версия 3.0, рассматриваемая в данном учебном курсе.

Руководителем группы, создающей язык C#, является сотрудник Microsoft Андреас Хейлсберг. Он был известен 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 языка 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, дающее возможность повторного использования созданных компонентов. Создаваемые компилятором компоненты являются само-документируемыми, помимо программного кода содержат метаинформацию, описывающую компоненты, и поэтому могут выполняться на различных платформах.

Из других важных факторов отметим следующие:

- C# создавался и развивается параллельно с каркасом Framework .Net и в полной мере учитывает все его возможности;
- C# является полностью объектно-ориентированным языком;
- C# является мощным объектным языком с возможностями наследования и универсализации;
- C# является наследником языка C++. Общий синтаксис, общие операторы языка облегчают переход от языка C++ к C#;
- сохранив основные черты своего родителя, язык стал проще и надежнее;
- благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты C# получают преимущества работы с виртуальной машиной;
- Framework .Net поддерживает разнообразие типов приложений на C#;
- реализация, сочетающая построение надежного и эффективного кода, является немаловажным фактором, способствующим успеху C#.

Введение в язык инструмента, получившего название LINQ (Language Integrated Query). Сегодня ни один серьезный проект на C# не обходится без обмена данными с внешними источниками данных - базами данных, Интернет и прочими хранилищами. В таких ситуациях приходилось использовать специальные объекты (ADO .Net или их более ранние версии). При работе с ними нужно было применять SQL - специальный язык запросов. Благодаря LINQ язык запросов становится частью языка программирования C#. Тем самым реализована давняя мечта программистов - работать с данными, находящимися в различных внешних источниках, используя средства, принадлежащие языку программирования, не привлекая дополнительные инструментальные средства и языки.

Введение в язык инструментария, характерного для функционального стиля программирования, - лямбда-выражений, анонимных типов и функций. Андреас Хейлсберг полагает, что смесь императивного и функционального стилей программирования упрощает задачи разработчиков, поскольку функциональный стиль позволяет разработчику сказать, что нужно делать, не уточняя, как это должно делаться.

Следующая версия языка C# 4.0 должна появиться параллельно с выходом новой версии Visual Studio 2010. Продолжается работа над версией C# 5.0. Можно отметить три основные тенденции в развитии языка - декларативность, динамичность и параллельность. Разработчики пытаются придать языку C# свойства, расширяющие традиционные возможности процедурных языков. Явно заметен тренд к функциональным языкам с их декларативным стилем. Такие свойства появились уже в C# 3.0, в следующих версиях они только расширяются.

В новой версии Visual Studio 2010 должны появиться новые динамические языки программирования: "железный змей" - Iron Python и Iron Ruby. Эти языки проще устроены, во многом из-за того, что не являются строго типизированными и потому не позволяют проводить контроль типов еще на этапе компиляции. В C# 4.0 введена возможность задания динамических переменных, аналогично тому, как это делается в динамических языках.

Параллельные вычисления в ближайшие 5-10 лет станут реальностью повседневной работы программиста. В этом направлении развивается техника. Языки программирования должны поддерживать эту тенденцию.

Компилятор как сервис, программирование на лету, - такие возможности должны появиться в C# 5.0. Можно не сомневаться, что C#-программистам в ближайшие годы скучать не придется.

Visual Studio 2008

Как уже отмечалось, принципиальной новинкой этой версии является возможность построения новых типов программных проектов, что обеспечивается новой версией каркаса Framework .Net 3.5. Если не считать этой важной особенности, то идейно Visual Studio 2008 подобна предыдущим версиям Visual Studio 2005 и Visual Studio 2003.

Рассмотрим основные особенности среды разработки Visual Studio.

Открытость

Среда разработки программных проектов 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** , **Visual Basic .Net**, - в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами.

Таких расширений среды **Visual Studio** сделано уже достаточно много, практически они существуют для всех известных языков - **Fortran** и **Cobol**, **RPG** и **Component Pascal**, **Eiffel**, **Oberon** и **Smalltalk**.

Новостью является то, что Microsoft не включила в Visual Studio 2008 поддержку языка Java. Допустимые в предыдущих версиях проекты на языке J++ в Visual Studio 2008 в настоящее время создавать нельзя, ранее созданные проекты в студии не открываются.

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. Visual Studio .Net, благодаря единому каркасу, в определенной мере решает эту задачу в мире программистов.

Framework .Net - единый каркас среды разработки приложений

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, оно широко использовалось еще в четвертой версии Visual Studio. Библиотека классов MFC (Microsoft Foundation Classes) играла роль 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 традиционные для программистов классы, задающие расширенную систему типов данных, в частности, динамические типы данных - списки, деревья, коллекции, шаблоны.

За прошедшие годы 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 здесь появился ряд важных нововведений.

Единство каркаса

Каркас стал единым для всех языков среды разработки. Поэтому на каком бы языке программирования не велась разработка, она работает с классами 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 основное множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic - Integer, а в языках C++ и C# - int, проецируется на один и тот же тип каркаса - System.Int32. В 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; динамические типы данных - стеки, очереди, списки, деревья. 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.

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). 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, содержащее 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 библиотеки FCL.

Общеязыковая исполнительная среда CLR - динамический компонент каркаса

Важным шагом в развитии каркаса Framework .Net стало введение динамического компонента каркаса - **исполнительной среды CLR**. С появлением CLR процесс выполнения приложений стал принципиально другим.

Двухэтапная компиляция. Управляемый модуль и управляемый код

Компиляторы языков программирования, включенные в Visual Studio .Net, создают код на промежуточном языке **IL (Intermediate Language)** - ассемблерном языке. В результате компиляции проекта, содержащего несколько файлов, создается так называемый **управляемый модуль** - переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и метаданные - всю информацию, необходимую для CLR, чтобы под ее управлением PE-файл мог быть исполнен. Метаданные доступны и конечным пользователям. Классы, входящие в пространство имен Reflection, позволяют извлекать метаинформацию о классах, используемых в проекте. Этот процесс называется отражением. Об атрибутах классов, отображаемых в метаданные PE-файла, мы еще будем говорить неоднократно. В зависимости от выбранного типа проекта, PE-файл может иметь разные уточнения - exe, dll, mod или mdl.

Заметьте, PE-файл, имеющий уточнение exe, хотя и является exe-файлом, но это не обычный исполняемый Windows файл. При его запуске он распознается как 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 следует рассматривать как 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 и 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 , Windows Vista и семейство .Net Server. Облегченная версия Framework .Net разработана для операционных систем Windows CE и Palm.

Framework .Net развивается параллельно с развитием языков программирования, среды разработки программных проектов и операционных языков. Версия языка C# 2.0 использовала версию Framework .Net 2.0. Операционная система Windows Vista включила в качестве надстройки Framework .Net 3.0. Язык C# 3.0 и Visual Studio 2008 работают с версией Framework .Net 3.5.

Framework .Net является 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.

Компилятор JIT, входящий в состав CLR, компилирует IL код с учетом 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 особенности процессора.

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, проверяется и не допускается использование "чужой" памяти, не допускаются и другие нарушения. 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? 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 написаны 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.

В Visual Studio 2008 появились новые типы проектов, основанные на возможностях, предоставляемых технологией WPF (Windows Presentation Foundation). Эта технология позволяет строить новое поколение систем презентации - с новыми графическими возможностями, связыванием данных и прочими элементами, придающими приложению принципиально новые свойства. Предполагается, что этот тип приложений постепенно будет вытеснять традиционные Windows-приложения, основанные на понятии окна.

Windows Communication Foundation (WCF) и Windows Workflow Foundation (WF)

Технологии WCF и WF позволяют строить специализированные приложения и службы (Services), позволяющие приложениям обмениваться данными, используя асинхронный ввод-вывод.

ASP.NET

Новые возможности Framework .Net 3.5 облегчают разработку Веб- приложений, в частности, построение сайтов с AJAX (Asynchronous Javascript and XML) - свойствами. Приложения с такими свойствами становятся более быстрыми и удобными, позволяя при взаимодействии с сервером не перезагружать всю страницу полностью.

Управляемый и неуправляемый код

Как уже отмечалось, результатом проекта, написанного на C# и скомпилированного в Visual Studio 2008, является сборка (assembly), которая содержит IL-код проекта и манифест, полностью описывающий сборку. Сборка может быть создана на одном компьютере, на одной платформе, а выполняться на другом компьютере с другим типом процессора, с другой операционной системой. Для выполнения сборки необходимо и достаточно установки на целевом компьютере соответствующей версии Framework .Net, представляющего надстройку над операционной системой.

Когда мы говорим о сборках, язык программирования, на котором создавался исходный код, уже не имеет значения, его особенности никак не отражаются в сборке. Сборки, созданные на VB или C++ с управляемыми расширениями, неотличимы от сборок, которые созданы на C# или других языках, включенных в состав Visual Studio 2008 и использующих каркас Framework .Net при компиляции управляемого кода.

С другой стороны, понятно, что в состав Visual Studio 2008 могут включаться языки, не применяющие Framework .Net, не создающие сборки с управляемым кодом, а использующие собственные библиотеки и собственные каркасы приложений (Framework Applications). В частности, на языке C++ в рамках Visual Studio 2008 можно писать проекты, работающие с библиотеками MFC и ATL, ориентированные исключительно на C++ и создающие в результате компиляции проекта обычные exe-файлы.

Сегодня на всех компьютерах, работающих под управлением любой из версий Windows, установлена соответствующая версия Framework .Net, так что на таких компьютерах могут выполняться и сборки, и обычные exe-файлы. Поскольку Framework .Net, так же как и C#, стандартизован и является свободно распространяемым программным продуктом, его можно встретить и на тех компьютерах, где нет Windows.

На рис. 1.1 показана схема функционирования компьютера, позволяющего выполнять как сборки - управляемый код, так и обычные exe-файлы - неуправляемый код.

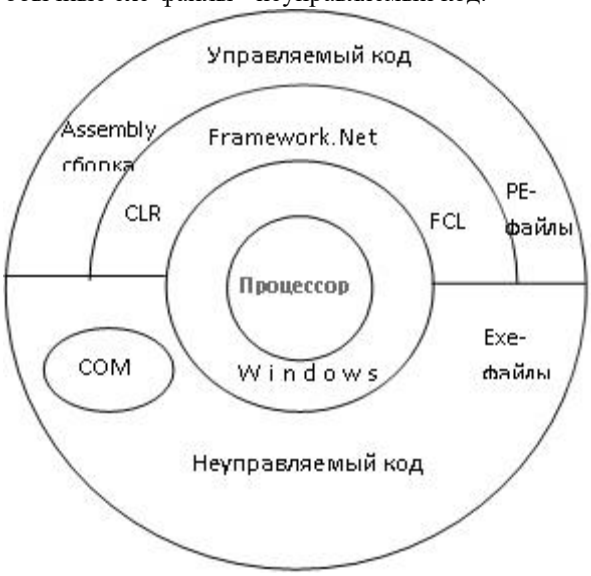


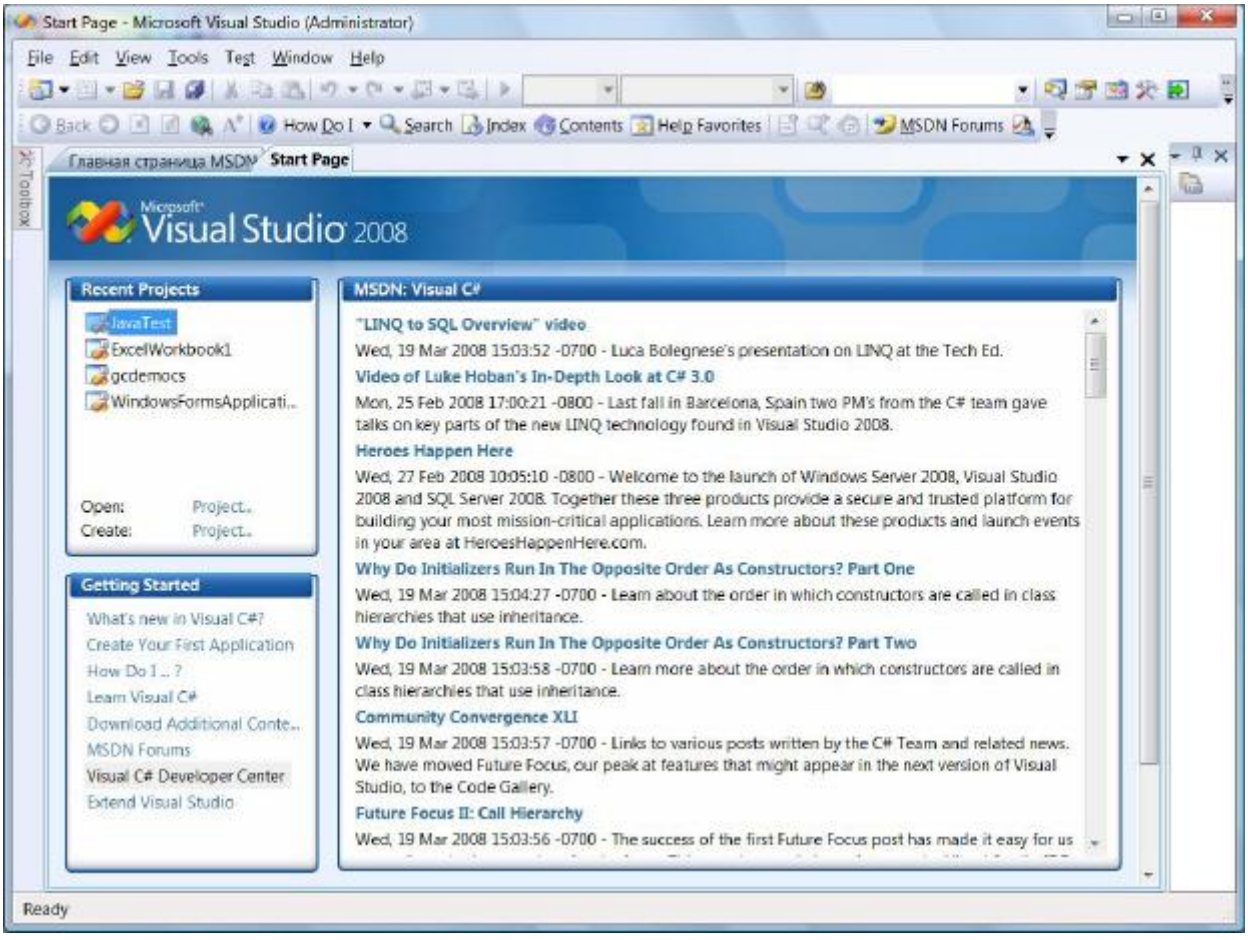
Рис. 1.1. Управляемый и неуправляемый код

Заметьте: два мира программ, выполняемые по-разному, могут взаимодействовать друг с другом - из управляемого кода возможен вызов программ с неуправляемым кодом и наоборот. В проектах, написанных на C#, можно управлять офисными приложениями - документами Word и Excel. Офисные документы - это COM-объекты, принадлежащие миру неуправляемого кода, а проекты C# - это сборки, жители страны с управляемым кодом.

Проекты C# в Visual Studio 2008

При запуске Visual Studio 2008, которая, надеюсь, уже установлена на Вашем компьютере, открывается стартовая страница. В окне "Recent Projects" стартовой страницы есть две скромные, непрезентабельного вида ссылки - "Open Project…" и "Create Project…".

Они задают две основные функции, которые может выполнять разработчик в Visual Studio 2008, - он может открыть существующий проект и работать с ним или создать и работать с новым проектом. В большинстве случаев после открытия стартовой страницы щелчком по одной из ссылок мы переходим к созданию или открытию проекта. Вид стартовой страницы показан на рис. 1.2.

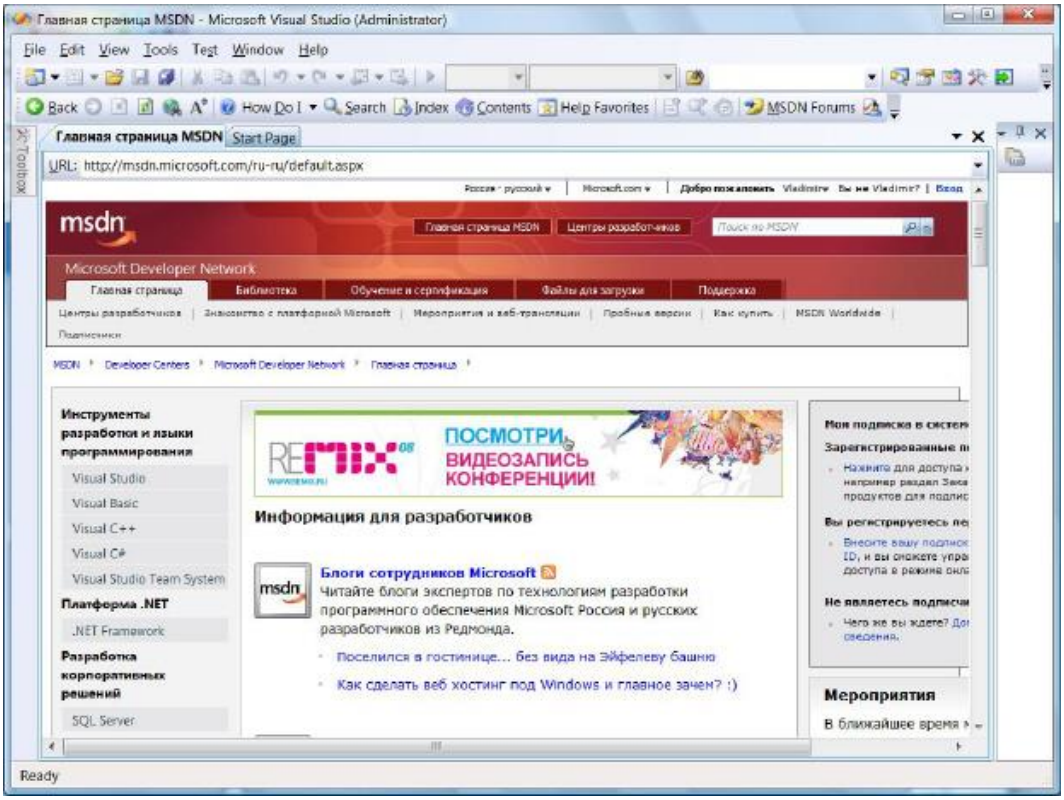


увеличить изображение

Рис. 1.2. Вид стартовой страницы

Стартовая страница, помимо перехода к выполнению основных задач разработчика, предоставляет важные дополнительные возможности. Во-первых, здесь

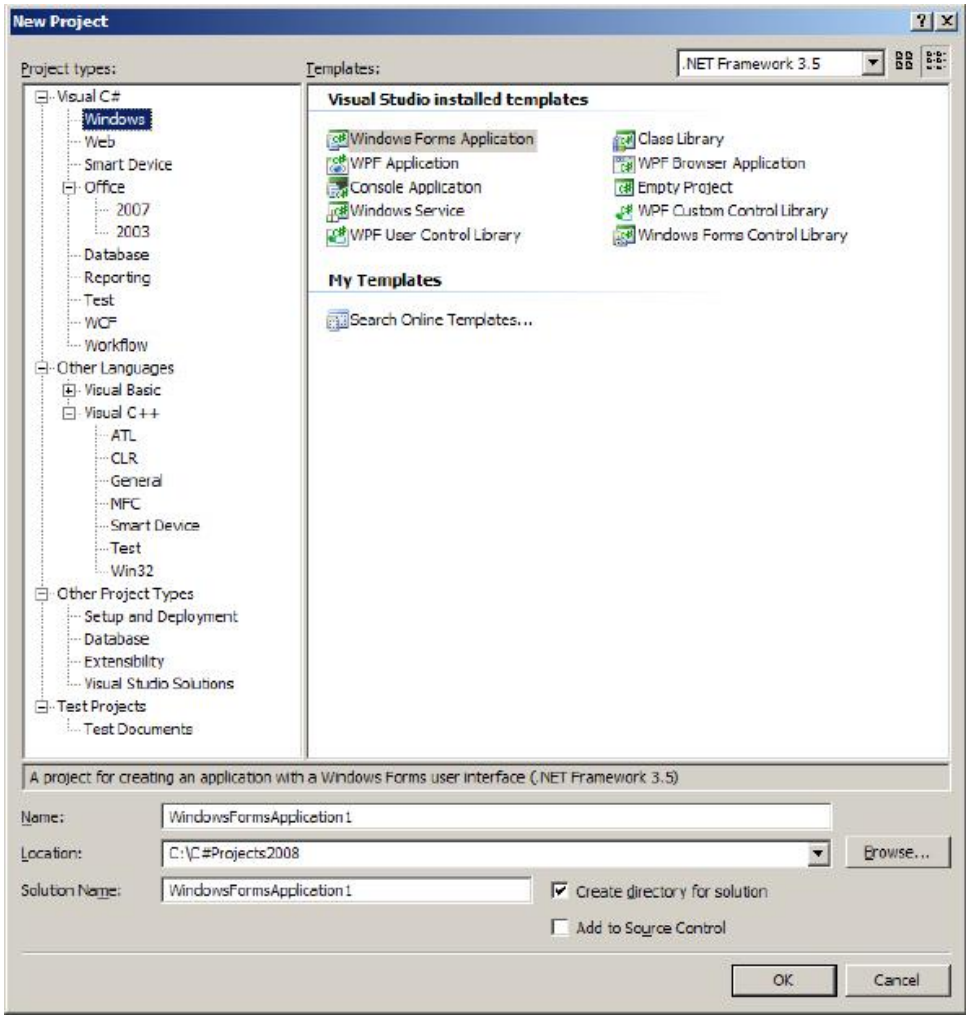
расположен список текущих проектов, позволяющий сразу же перейти к работе с нужным проектом из этого списка.
Для компьютера, подключенного к интернет, стартовая страница автоматически связывается с сайтом, содержащим текущую информацию по C# и Visual Studio 2008, - по умолчанию показываются новости с сайта msdn. Выбрав соответствующий пункт из раздела Getting Started (Давайте Начнем), можно получить информацию из центра разработчиков C#, можно подключиться к одному из трех форумов по языку C#, можно получить нужную справку в режиме "on line".
На стартовой странице, помимо вкладки "StartPage", расположена вкладка "Главная страница MSDN", позволяющая перейти к соответствующему сайту. На рис. 1.3 показана страница, открытая при выборе этой вкладки.



увеличить изображение
Рис. 1.3. Вид главной страницы MSDN

Коль скоро речь зашла о получении текущей информации и справок по языку C#, упомяну несколько полезных ресурсов:

- http://forums.msdn.microsoft.com/en-us/forums/ - англоязычный сайт предоставляет доступ к различным форумам, в том числе форумам по языку C#;
- http://msdn.microsoft.com/ru-ru/default.aspx - русскоязычный сайт msdn;
- http://msdn.microsoft.com/en-us/vcsharp/default.aspx - англоязычный сайт по языку C# на msdn;
- http://csharpfriends.com/ - англоязычный сайт, где можно найти нужную информацию, задать вопросы и получить ответы от сообщества разработчиков.



Создание проекта

Выбрав на стартовой странице ссылку "Создать проект", переходим на страницу создания нового проекта.
Категория проектов на языке C# включает в настоящее время 9 подкатегорий. Наиболее часто используемыми являются первые две - Windows и Web. Первая из них позволяет строить Windows-проекты, предназначенные для работы на локальном компьютере. Вторая подкатегория дает возможность строить Web-приложения, работающие в сети интернет или интранет. Этим типам проектов или, по крайней мере, части из них будет уделено основное внимание в нашем курсе.
Новые категории проектов на C# возникли в связи с упоминавшимися новинками Framework .Net 3.5. В частности, появились отдельные подкатегории WCF и Workflow, позволяющие строить проекты, которые используют упоминавшиеся технологии WCF и WF, включенные в новый каркас.
Рассмотрим чуть более подробно категорию Windows-проектов. Она

включает на момент написания этого текста 10 типов проектов.
Windows Forms Application - основной вид проектов, строящихся на языке C#. Большинство примеров в рамках нашего курса будут использовать этот тип проектов. Интерфейс таких приложений строится в визуальном стиле на основе популярных форм Windows. Приставка Visual в названии языка во многом определяется возможностями этого типа проектов.
Class Library - проект, позволяющий построить DLL (Dynamic Link Library) - динамически подключаемую библиотеку классов. Этот вид проектов будет столь же часто встречаться в наших примерах, как и предыдущий. Дело в том, что всегда полезно отделять содержательную часть приложения от ее интерфейса. Классы, определяющие содержание приложения, будут собираться в DLL, которая затем будет подключаться к проекту, описывающему интерфейс приложения. Такой подход в наибольшей степени соответствует повторному использованию. Один раз созданные содержательные классы, собранные в DLL, могут использоваться в разных приложениях.
Console Applications - этот тип проектов почти не используется в программных продуктах, для которых интерфейс играет крайне важную роль. Тем не менее, это весьма популярный у разработчиков тип проектов, применяемый для внутренних целей в ходе разработки. Этот тип проектов довольно часто будет появляться в наших примерах, когда для понимания тех или иных возможностей языка C# достаточно весьма простого интерфейса - ввода и вывода данных на консоль.
Windows Forms Control Library - полезный и часто используемый тип проектов. Он применяется при создании повторно используемого элемента, обладающего визуальным интерфейсом.
WPF Application, WPF Browser Application, WPF User Control Library, WPF Custom Control Library - 4 типа проектов, которые связаны с упоминавшейся новой технологией WPF, включенной в состав каркаса Framework .Net 3.5.
Windows Service - проект, который задает службы (сервисы), предоставляемые удаленным компьютером.
Empty - пустой проект. Все предыдущие типы проектов изначально предлагают разработчику проекта вполне определенную функциональность. Когда при создании проекта разработчик указывает его тип, из библиотеки классов FCL, входящей в состав каркаса Framework .Net, выбираются классы, задающие архитектуру данного типа проекта. Эти классы составляют каркас проекта, построенного по умолчанию для данного типа, они и определяют функциональность, присущую данному типу проекта. Разработчику остается дополнить каркас проекта плотью и кровью, добавив собственные классы и расширив функциональность классов, входящих в каркас проекта. Для пустого проекта начальная функциональность отсутствует - разработчик все должен делать сам - ab ovo. Мы пустыми проектами заниматься не будем.

Лекция 2

Тема: Языки программирования. Обзор современных языков программирования

Дадим определения понятий, которые будут использоваться. **Класс (Class)**

Класс - это центральное понятие объектно-ориентированного программирования и языка C#. Разработчик проектов на C# использует стандартные классы из библиотеки FCL и создает собственные классы. У класса две различные роли.

Класс - это модуль - архитектурная единица построения проекта по модульному принципу. Справиться со сложностью большого проекта можно только путем деления его на модули - сравнительно небольшие единицы, допускающие независимую разработку и последующее объединение в большую систему.

Класс - это тип данных. Тип данных - это семантическая единица, которая описывает свойства и поведение множества объектов, называемых экземплярами класса. Синтаксически класс представляет описание данных, называемых полями класса, описание методов класса и описание событий класса. Для класса, рассматриваемого как тип данных, поля определяют состояние объектов, методы - поведение объектов. События - это некоторые специальные состояния, в которых может находиться объект и которые могут обрабатываться внешними по отношению к классу обработчиками события. Так, например, объект класса Person может иметь событие "День рождения", и каждый из обработчиков этого события может принести объекту свои поздравления по этому случаю.

Как правило, класс C# играет обе роли. Но язык C# позволяет определять классы, играющие только роль модуля. Это так называемые статические классы, для которых невозможно создавать объекты. В ходе выполнения программной системы создается единственный экземпляр такого класса, обеспечивающий доступ к полям и методам этого модуля.

Хороший стиль программирования требует, чтобы каждый класс сохранялся в отдельном файле, имя которого совпадало бы с именем класса. Это требование стиля, которое на практике может и не выдерживаться. В наших примерах будем стараться выдерживать этот стиль.

Объект (Object)

Определив класс, разработчик получает возможность динамически создавать объекты класса.

Для программистов, начинающих работать в объектном стиле, типичной ошибкой является путаница понятий объекта и класса. Нужно с самого начала уяснить разницу. Класс, создаваемый разработчиком, представляет статическое описание множества объектов. Объект - это динамическое понятие, он создается в ходе выполнения программной системы, реально существует в памяти компьютера и обычно исчезает по завершении выполнения проекта. Программист может создать программную систему, включающую два-три класса, но в ходе работы такой системы могут динамически появляться сотни объектов, взаимодействующих друг с другом достаточно сложным образом.

Заметьте, путаница понятий класса и объекта характерна и для опытных разработчиков. Показателен тот факт, что центральный класс в библиотеке FCL, являющийся прародителем всех классов как библиотечных, так и создаваемых разработчиком, назван именем *Object*.

Пространство имен (Namespace)

Пространство имен - это оболочка, которая содержит множество классов, объединенных, как правило, общей тематикой или группой разработчиков. Собственные имена классов внутри пространства имен должны быть уникальны. В разных пространствах могут существовать классы с одинаковыми именами. Полное или уточненное имя класса состоит из уникального имени пространства имен и собственного имени класса. В пространстве имен могут находиться как классы, так и пространства имен.

Пространства имен позволяют задать древесную структуру на множестве классов большого проекта. Они облегчают независимую разработку проекта большим коллективом разработчиков, каждая группа которого работает в своем пространстве имен.

Пространства имен придают структуру библиотеке FCL, которая содержит большое число различных пространств имен, объединяющих классы определенной тематики. Центральным пространством имен библиотеки FCL является пространство System - оно содержит другие пространства и классы, имеющие широкое употребление в различных проектах.

Проект (Project)

Проект - это единица компиляции. Результатом компиляции проекта является сборка. Каждый проект содержит одно или несколько пространств имен. Как уже говорилось, на начальном этапе создания проекта по заданному типу проекта автоматически строится каркас проекта, состоящий из классов, которые являются наследниками классов, входящих в состав библиотеки FCL. Так, если разработчик указывает, что он хочет построить проект типа "Windows Forms Application", то в состав каркаса проекта по умолчанию войдет класс Form1 - наследник библиотечного класса Form. Разработчик проекта населит созданную форму элементами управления - объектами соответствующих классов, тем самым расширив возможности класса, построенного по умолчанию.

Каждый проект содержит всю информацию, необходимую для построения сборки. В проект входят все файлы с классами, построенные автоматически в момент создания проекта, и файлы с классами, созданные разработчиком проекта. Помимо этого, проект включает в себя ссылки на пространства имен из библиотеки FCL, которые содержат классы, используемые в ходе вычислений. Проект содержит ссылки на все подключаемые к проекту DLL, COM-объекты, другие проекты. В проект входят установки и ресурсы, требуемые для работы. Частью проекта является файл, содержащий описание сборки.

В зависимости от выбранного типа проект может быть выполняемым или невыполняемым. К выполняемым проектам относятся, например, проекты типа Console или Windows. При построении каркаса выполняемого проекта в него включается класс, содержащий статическую процедуру с именем Main. В результате компиляции такого проекта создается PE-файл

(Portable Executable file) - выполняемый переносимый файл с уточнением exe. Напомним, что PE-файл может выполняться только на компьютерах, где установлен Framework .Net, поскольку это файл с управляемым кодом.
К невыполняемым проектам относятся, например, проекты типа DLL. В результате компиляции такого проекта в сборку войдет файл с уточнением dll. Такие проекты (сборки) непосредственно не могут быть выполнены на компьютере. Они присоединяются к выполняемым сборкам, откуда и вызываются методы классов, размещенных в невыполняемом проекте (DLL).

Сборка (Assembly)

Сборка - результат компиляции проекта. Сборка представляет собой коллекцию из одного или нескольких файлов, помеченных номером версии. Каждая сборка разворачивается на компьютере как единое целое. Программист работает с проектами, CLR работает со сборками. Сборка позволяет решать вопросы безопасности, так как содержит описание требуемых ей ресурсов и права доступа к элементам сборки. Каждая сборка содержит манифест, включающий полное описание сборки, ее элементов, требуемые ресурсы, ссылки на другие сборки, исполняемые файлы. Благодаря этому описанию CLR не требуется никакой дополнительной информации для развертывания сборки, трансляции промежуточного кода и его выполнения. Манифест идентифицирует сборку, специфицирует файлы, требуемые для реализации сборки, специфицирует типы и ресурсы, составляющие сборку, задает зависимости, необходимые в период компиляции для связи с другими сборками, специфицирует множество разрешений, необходимых, чтобы сборка могла выполняться на данном компьютере.

Решение (Solution)

Каждый проект, создаваемый в Visual Studio 2008, помещается в некоторую оболочку, называемую Решением - Solution. Решение может содержать несколько проектов, как правило, связанных общей темой. Например, все проекты, рассматриваемые в одной лекции курса, можно поместить в одно Решение. В наших примерах зачастую Решение будет содержать три проекта: DLL с классами, определяющими содержательную сторону приложения, и два интерфейсных проекта - консольный и Windows.
Когда создается новый проект, он может быть помещен в уже существующее Решение или может быть создано новое Решение, содержащее проект.
Решения позволяют придать структуру множеству проектов, что особенно полезно, когда проектов много.

Пример

Для первого рассмотрения наш пример будет достаточно сложным: мы построим Решение, содержащее три проекта - проект DLL, консольный проект и Windows-проект.

Постановка задачи

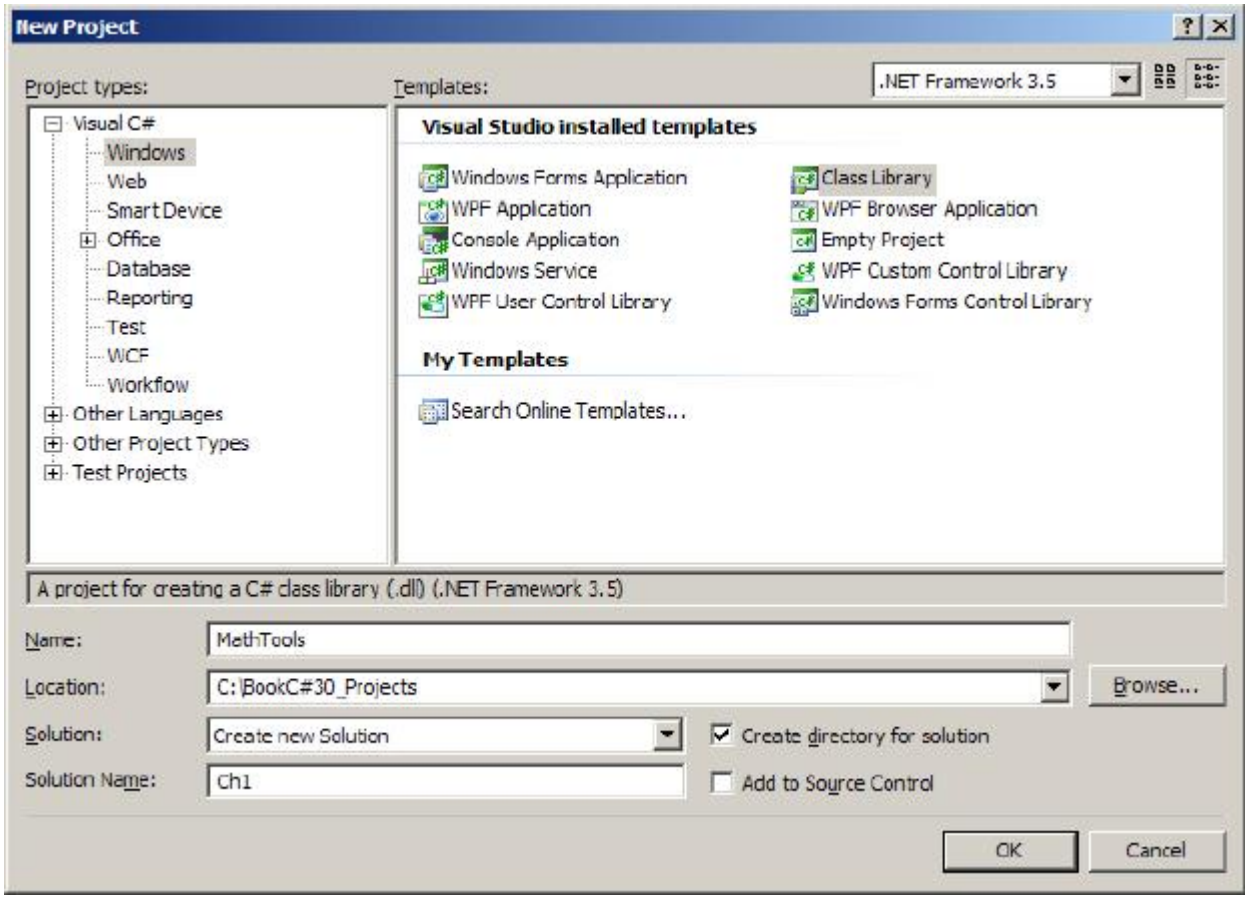
Начнем с описания содержательной постановки задачи. В системе типов языка C# есть несколько типов, задающих различные подмножества арифметического типа данных - *int*, *double* и другие. Для значения *x* любого из этих типов хорошо бы уметь вычислять математические функции - $\sin(x)$, $\ln(x)$ и другие. Встраивать вычисление этих функций в каждый из классов, задающих соответствующий арифметический подтип, кажется неразумным. Поэтому в библиотеку FCL включен класс Math, методы которого позволяют вычислять по заданному аргументу нужную математическую функцию. Класс Math является примером статического класса, играющего единственную роль - роль модуля. У этого класса нет собственных данных, если не считать двух математических констант - e и π, а его методы являются сервисами, которые он предоставляет другим классам.
Построим аналог класса Math и поместим этот класс в DLL, что позволит повторно использовать его, присоединяя при необходимости к различным проектам. В нашем примере не будем моделировать все сервисы класса Math. Ограничимся рассмотрением вычисления функции $\sin(x)$. Эту функцию, как и другие математические функции, можно вычислить, используя разложение в ряд Тэйлора:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!} \qquad (1.1)$$

Детали вычислений, использующих формулу 1.1, отложим на момент реализации. А пока продолжим уточнять цель нашего примера. Итак, мы хотим построить DLL, содержащей класс, являющийся аналогом класса Math из библиотеки FCL. Затем мы хотим построить консольный проект, позволяющий провести тестирование корректности вычислений функций построенного нами класса. Затем мы построим Windows-проект, интерфейс которого позволит провести некоторые интересные исследования. Все три проекта будут находиться в одном Решении.

Создание DLL - проекта типа "Class Library"

Запустим Visual Studio 2008, со стартовой страницы перейдем к созданию проекта и в качестве типа проекта укажем тип "Class Library". В открывшемся окне создания DLL, показанном на рис. 1.5, все поля заполнены значениями по умолчанию. Как правило, их следует переопределить, задавая собственную информацию.



увеличить **изображение**

Рис. 1.5. Создание проекта DLL

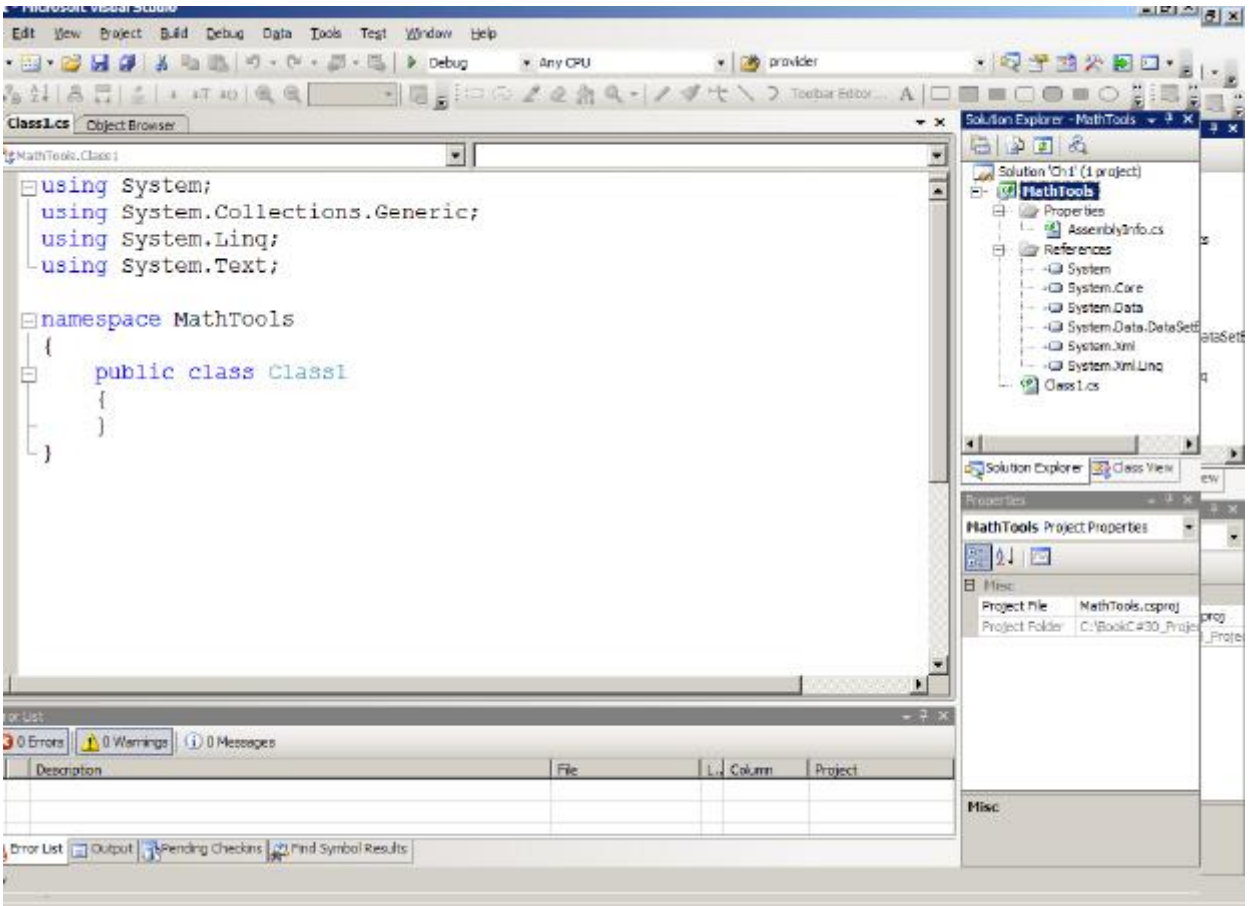
В поле Name задается имя строящейся DLL - MathTools в нашем случае.

В поле Location указывается путь к папке, где будет храниться Решение, содержащее проект. Для Решений этого курса создана специальная папка.

В поле Solution выбран элемент "Create New Solution", создающий новое Решение. Альтернативой является элемент списка, указывающий, что проект может быть добавлен к существующему Решению.

В окне Solution Name задано имя Решения. Здесь выбрано имя Ch1, указывающее на то, что все проекты первой лекции вложены в одно Решение.

Обратите внимание и на другие установки, сделанные в этом окне, - включен флажок (по умолчанию) "Create directory for solution", в верхнем окошке из списка возможных каркасов выбран каркас Framework .Net 3.5. Задав требуемые установки и щелкнув по кнопке "OK", получим автоматически построенную заготовку проекта DLL, открытую в среде разработки проектов Visual Studio 2008 . На рис. 1.6 показан внешний вид среды с построенным Решением и проектом.



увеличить изображение

Рис. 1.6. Среда Visual

Studio 2008 с начальным проектом DLL
Среду разработки можно настраивать, открывая или закрывая те или иные окна, перемещая и располагая их по своему вкусу. Это делается стандартным способом, имы не будем на этом останавливаться.
В окне проектов Solution Explorer показано Решение с именем "Ch1", содержащее проект DLL с именем "MathTools". В папке "Properties" проект содержит файл с описанием сборки - ее имя и другие характеристики. В папке "References" лежат ссылки на основные пространства имен библиотеки FCL, которые могут понадобиться в процессе работы DLL.
Поскольку всякая DLL содержит один или несколько классов, то для одного класса, которому по умолчанию дано имя "Class1", заготовка построена. Класс этот, показанный в окне кода, пока что пуст - не содержит никаких элементов.
Построенный автоматически класс вложен в пространство имен, которое по умолчанию получило имя, совпадающее с именем проекта - MathTools. Перед именем пространства заданы четыре предложения *using*, играющие роль инструкций для компилятора. В этих предложениях указываются имена пространств имен, присоединенных к проекту. Когда в коде создаваемого класса нужно сослаться на класс из пространств, указанных в предложениях *using*, можно задавать собственное имя этого класса, опуская имя пространства.
Мы рассмотрели подготовительную часть работы, которую Visual Studio 2008 выполнила для нас. Дальше предстоит потрудиться самим. С чего следует начать? С переименования! Важное правило стиля программирования говорит, что имена классов должны быть содержательными. Изменим имя "Class1" на имя "MyMath". Как следует правильно изменять имена объектов в проектах? Никак не вручную. В окне кода проекта выделите имя изменяемого объекта, затем в главном меню выберите пункт *Refactor* и подпункт *Rename*. В открывшемся окне укажите новое имя. Тогда будут показаны все места, требующие переименования объекта. В данном случае будет только одна очевидная замена, но в общем случае замен много, так что автоматическая замена всех вхождений крайне полезна.
Следующий шаг также продиктован правилом стиля: имя класса и имя файла, хранящего класс, должны совпадать. Переименование имени файла делается непосредственно в окне проектов Solution Explorer.
И следующий шаг продиктован крайне важным правилом стиля, имеющим собственное название: правило "И не вздумайте!", которое гласит - "И не вздумайте написать класс без заголовочного комментария". Для добавления документируемого комментария достаточно в строке, предшествующей заголовку класса, набрать три подряд идущих слеша (три косых черты). В результате перед заголовком класса появится заголовочный комментарий - тэг "summary", в который и следует добавить краткое, но содержательное описание сути класса. Тэги "summary", которыми следует сопровождать классы, открытые (public) методы и поля класса играют три важные роли. Они облегчают разработку и сопровождение проекта, делая его самодокументируемым. Клиенты класса при создании объектов класса получают интеллектуальную подсказку, поясняющую суть того, что можно делать с объектами. Специальный инструментарий позволяет построить документацию по проекту, включающую информацию из тегов "summary". В нашем случае комментарий к классу MyMath может быть достаточно простым - "Аналог класса Math библиотеки FCL".
Поскольку мы хотим создать аналог класса Math, в нашем классе должны быть аналогичные методы. Начнем, как уже говорилось, с метода, позволяющего вычислить функцию $\sin(x)$. Заголовок метода сделаем такой же, как и в классе аналоге. Согласно правилу стиля "И не вздумайте" зададим заголовочный комментарий к методу. В результате в тело класса добавим следующий код:

```
/// <summary>
/// Sin(x)
/// </summary>
/// <param name="x">угол в радианах - аргумент функции Sin</param>
/// <returns>Возвращает значение функции Sin для заданного угла</returns>
public static double Sin(double x)
{
}
```

Осталось написать реализацию вычисления функции, заданную формулой 1.1. Как и во всяком реальном программировании для этого требуется знание некоторых алгоритмов. Алгоритмы вычисления конечных и бесконечных сумм относятся к элементарным алгоритмам, изучаемым в самом начале программистских курсов.
Вычисление конечных и бесконечных сумм
Вычисление конечных сумм и произведений - это наиболее часто встречающийся тип элементарных задач, шаблон решения которых должен быть заучен как 2*2. Какова бы не была сложность выражений, стоящих под знаком конечной суммы с заданным числом слагаемых, задачу всегда можно записать в виде:

$$S = \sum_{k=1}^{n} a_k \qquad (1.2)$$

и применить для ее решения следующий шаблон:

```
S=0;
for(int k=1; k<=n; k++)
{
        //Вычислить текущий член суммы ak
        …
        S+=ak;
}
```

Часто приходится пользоваться слегка расширенным шаблоном:

```
Init;
for(int k=1; k<=n; k++)
{
        //Вычислить текущий член суммы ak
        …
        S+=ak;
}
```

В этом шаблоне Init представляет группу операторов, которые инициализируют используемые в цикле переменные значения, обеспечивающие корректность применения цикла. В частном случае, рассмотренном выше, инициализация сводится к заданию значения переменной S. Заметьте, если перед началом цикла не позаботиться о том, чтобы эта переменная была равна нулю, то после завершения цикла корректность результата не гарантируется.
В этой схеме основные проблемы могут быть связаны с вычислением текущего члена суммы ak. Нужно понимать, что ak - это не массив, а скаляр - простая переменная. Значения этой переменной вычисляются заново на каждом шаге цикла, задавая очередной член суммирования. Кроме того, следует заботиться об эффективности вычислений, применяя два основных правила, позволяющие уменьшить время вычислений.
Чистка цикла. Все вычисления, не зависящие от k, следует вынести из цикла (в раздел Init).
Рекуррентная формула. Часто можно уменьшить время вычислений ak, используя предыдущее значение ak, построив рекуррентную формулу $a_{k+1} = f(a_k)$. Этот прием с успехом применяется как при вычислении функции $\sin(x)$ по формуле 1.1, так и при аналогичных вычислениях большинства других математических функций.
Покажем на примере формулы 1.1, как можно построить необходимые рекуррентные соотношения. Запишем соотношения для a_0, a_k, a_{k+1}):

$$a_0 = \frac{(-1)^0 x}{1!} = x; \quad a_k = \frac{(-1)^k x^{2k+1}}{(2k+1)!}; \quad a_{k+1} = \frac{(-1)^{k+1} x^{2k+3}}{(2k+3)!} \quad (1.3)$$

Вычислив отношение a_{k+1}/a_k, получим требуемое рекуррентное соотношение:

$$a_{k+1} = a_k \frac{-x^2}{(2k+2)(2k+3)} \quad (1.4)$$

Значение a_0задает базис вычислений, позволяя инициализировать начальное значение переменной ak, а соотношение 1.4 позволяет каждый раз в теле цикла вычислять новое значение этой переменной. Заметьте: введение рекуррентного соотношения позволило избавиться от вычисления факториалов и возведения в степень на каждом шаге цикла.
Иногда следует ввести несколько дополнительных переменных, хранящих вычисленные значения предыдущих членов суммы. Рекуррентная формула выражает новое значение ak через предыдущее значение и дополнительные переменные, если они требуются. Начальные значения ak и дополнительных переменных должны быть корректно установлены перед выполнением цикла в разделе Init. Заметьте, если начальное значение ak вычисляется в разделе Init до цикла, то схема слегка модифицируется - вначале выполняется прибавление ak к S, а затем новое значение ak вычисляется по рекуррентной формуле.
А теперь поговорим о том, как справляться с бесконечными суммами, примером которых является формула 1.1. Для математики бесконечность естественна. Множество целых чисел бесконечно, множество рациональных чисел бесконечно, множество вещественных чисел бесконечно. Элементы первых двух множеств можно пронумеровать - они задаются счетными множествами, множество вещественных чисел несчетно. Сколь угодно малый промежуток вещественной оси мы бы не взяли, там находится бесконечно много вещественных чисел. Число π и другие иррациональные числа задаются бесконечным числом цифр, не имеющим периода.
Мир компьютеров - это конечный мир, хотя в нем и присутствует стремление к бесконечности. Множества, с которыми приходится оперировать в мире компьютера, всегда конечны. Тип целых чисел в языках программирования - int - всегда задает конечное множество целых из некоторого фиксированного диапазона. В библиотеке FCL это наглядно подтверждается самими именами целочисленных типов System.Int16, System.Int32, System.Int64. Типы вещественных чисел - double, float - задают конечные множества. Это достигается не только тем, что диапазон задания вещественных чисел ограничен, но и ограничением числа значащих цифр, задающих вещественное число. Поэтому для вещественных чисел компьютера всегда можно указать наборы таких двух чисел, между которыми нет никаких других чисел. Иррациональности компьютер не знает - число ? всегда задается конечным числом цифр.
Там, где в математике идет речь о пределах, бесконечных суммах, сходимости к бесконечности, в компьютерных вычислениях аналогичные задачи сводятся к вычислениям с заданной точностью - с точностью ε. Рассмотрим, например, задачу о вычислении предела числовой последовательности:

$$\lim_{n\to\infty} a_n = A$$

По определению число A является пределом числовой последовательности, если для любого сколь угодно малого числа ε существует такой номер N, зависящий от ε, что для всех n, больших N, числа a_n находятся в ε-окрестности числа A. Это определение дает основу для вычисления значения предела A. Понятно, что получить точное значение A во многих случаях принципиально невозможно, - его можно вычислить лишь с некоторой точностью и тоже не сколь угодно малой, поскольку существует понятие "машинного нуля" - минимального числа, все значения меньше которого воспринимаются как нуль. Когда два соседних члена последовательности - a_n и a_{n+1}- начинают отличаться на величину

по модулю меньшую чем δ, то можно полагать, что оба члена последовательности попали в ε-окрестность числа A и a_{n+1}можно принять за приближенное значение числа A. Это рассуждение верно только при условии, что последовательность действительно имеет предел. В противном случае этот прием может привести к ошибочным выводам. Например, рассмотрим последовательность, элементы которой равны 1, если индекс элемента делится на 3, и равны 2, если индекс не делится на 3. Очевидно, что у этой последовательности предела нет, хотя существуют полностью совпадающие соседние члены последовательности.
При вычислении на компьютере значения функции, заданной разложением в бесконечный сходящийся ряд, не ставится задача получения абсолютно точного результата. Достаточно вычислить значение функции с заданной точностью ε. На практике вычисления продолжаются до тех пор, пока текущий член суммы не станет по модулю меньше заданного ε. Чтобы этот прием корректно работал, необходима сходимость ряда.

Вернемся к задаче вычисления функции $\sin(x)$. Вот возможный шаблон решения:

```
Init;
while(Abs(ak) > EPS)
        {
                S+=ak;
                k++;
                //Вычислить новое значение ak
                …
        }
```

При применении этого шаблона предполагается, что в разделе Init объявляются и должным образом инициализируются нужные переменные - S, ak, k. По завершению цикла переменная S содержит значение функции, вычисленное с заданной точностью.
Теперь мы готовы расширить определение класса, добавив код метода.

Код

Приведем полный код проекта DLL, построенный на данный момент:

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

namespace MathTools
{
   /// <summary>
   /// Аналог класса Math библиотеки FCL
   /// </summary>
   public class MyMath
   {
      //Константы класса
      const double TWOPI = 2 * Math.PI;
      const double EPS = 1E-9;

      //Статические методы класса

      /// <summary>
      /// Sin(x)
      /// </summary>
      /// <param name="x">
      ///     угол в радианах - аргумент функции Sin
      /// </param>
      /// <returns>
      ///     Возвращает значение функции Sin для заданного угла
      /// </returns>
      public static double Sin(double x)
      {
         //Оптимизация - приведение к интервалу
         x = x % TWOPI;

         //Init
         double a = x;
         double res = 0;
         int k = 0;

         //Основные вычисления
         while (Math.Abs(a) > EPS)
```

```
        {
            res += a;
            a *= -x * x / ((2 * k + 2) * (2 * k + 3));
            k++;
        }
        return res;
    }
  }
}
```

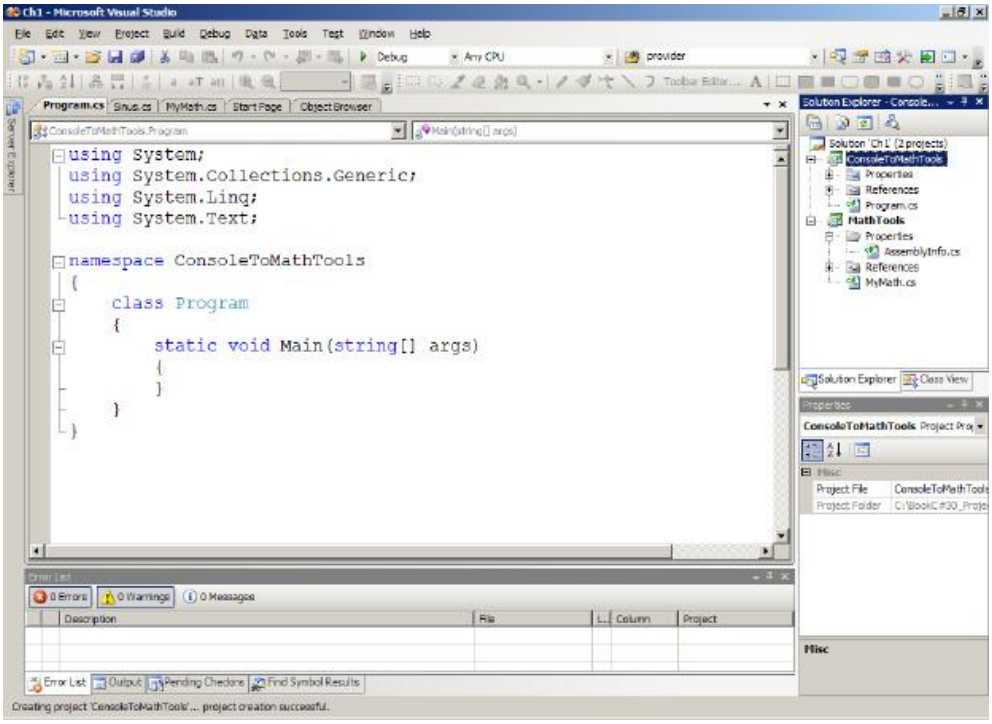
Построена DLL, содержащая класс, метод которого позволяет вычислять по заданному аргументу x функцию $\sin(x)$. Метод построен в полном соответствии с описанным алгоритмом. При его построении использованы две важные оптимизации. Во-первых, применено рекуррентное соотношение, позволяющее существенно ускорить время и точность вычисления функции (попробуйте объяснить, почему улучшаются оба эти параметра). Во-вторых, аргумент x приведен к сравнительно небольшому интервалу, что увеличивает скорость сходимости и гарантирует работоспособность метода для больших значений x.

Построим Решение, содержащее проект, для чего в Главном меню среды выберем пункт Build|Build Solution. В результате успешной компиляции будет построен файл с уточнением dll. Поскольку построенная сборка не содержит выполняемого файла, то непосредственно запустить наш проект на выполнение не удастся. Построим консольный проект, к которому присоединим нашу DLL, и протестируем, насколько корректно работают созданные нами методы. Заодно разберемся с тем, как строится консольный проект и как к нему подсоединяется сборка, содержащая DLL.

Консольный проект

Наша цель состоит в том, чтобы построить интерфейс, обеспечивающий конечному пользователю доступ к тем сервисам, которые предоставляет построенная DLL. Начнем с построения простейшего интерфейса, позволяющего пользователю с консоли вводить исходную информацию - в нашем случае аргумент x. С исходными данными пользователь может провести вычисления, вызвав сервисы, предоставляемые DLL, а затем полученные результаты вывести на консоль - экран дисплея. Для организации подобного интерфейса и служит тип проекта - Console Application.

Чтобы создать новый проект, находясь в среде разработки, вовсе не обязательно начинать со стартовой страницы. Достаточно выбрать пункт меню **File|New|Project**, приводящий на страницу создания нового проекта, показанную на рис. 1.5. В этом окне, как описано ранее, зададим тип строящегося проекта, дадим ему имя - ConsoleToMathTools, укажем, что проект добавляется к существующему Решению Ch1. В результате в уже существующее Решение добавится еще один проект, что отображено на рис. 1.7.



увеличить изображение

Рис. 1.7. Решение, включающее консольный проект

Как показано на рис. 1.7, в консольном проекте автоматически создается класс с именем Program, содержащий единственный статический метод - процедуру **Main**. Если скомпилировать этот проект и запустить его на выполнение, то начнет выполняться код этой процедуры, пока отсутствующий - его предстоит нам создать.

Начало начал - точка "большого взрыва"

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

x.F(arg1, arg2, …, argN)

В этом вызове x - это некоторый существующий объект, называемый целью вызова. Возможны три ситуации:

- x - имя класса. Объектом в этом случае является статический объект, который всегда создается в момент трансляции кода класса. Метод **F** должен быть статическим методом класса, объявленным с атрибутом static, как это имеет место для точки вызова - процедуры **Main**;
- x - имя объекта или объектное выражение. В этом случае **F** может 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. В отличие от двух первых случаев такой вызов называется неквалифицированным. Заметьте, неквалифицированный вызов вовсе не означает, что цель вызова отсутствует, - она просто задана по умолчанию. Целью является текущий объект, имеющий 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 Решение запускается на выполнение, в него должна входить сборка, отмеченная как стартовый проект, содержащая 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. Извне создается и первый объект, задающий статический модуль с методом Main. Этот объект и становится текущим.

Дальнейший сценарий зависит от содержимого точки входа. Как правило, в процессе работы метода Main создаются один или несколько объектов других классов, они и вызывают 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.

Связывание с DLL

Первым делом свяжем два построенных проекта, для чего в консольный проект добавим ссылку на проект с DLL MathTools. В окне Solution Explorer подведем указатель мыши к имени консольного проекта и из контекстного меню, появляющегося при щелчке правой кнопки, выберем пункт меню "Add Reference". В открывшемся окне добавления ссылок выберем вкладку "Projects". Поскольку проект MathTools включен в Решение, то он автоматически появится в открывшемся окне. Если ссылку нужно установить на проект, не включенный в Решение, то в окне добавления ссылок нужно задать путь к проекту. Нам проще, путь указывать не нужно, достаточно щелкнуть по появившемуся в окне имени MathTools. Ссылка на DLL появится в папке "References" консольного проекта. Теперь проекты связаны и из консольного проекта доступны сервисы, предоставляемые DLL.

Организация консольного интерфейса

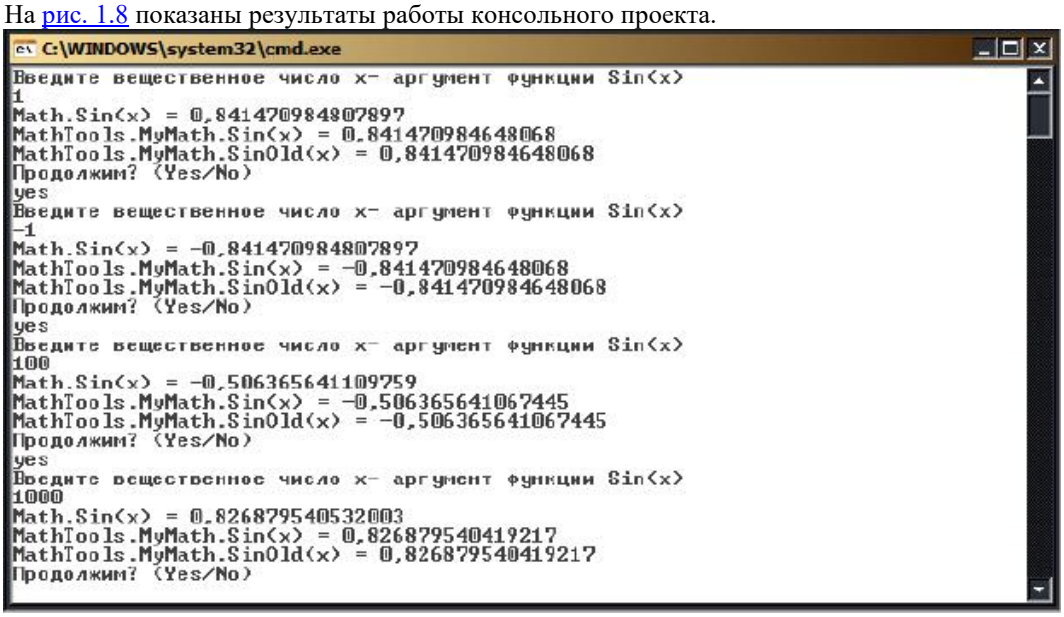
Задача кода, который мы встроим непосредственно в уже созданную процедуру Main, достаточно понятна. Необходимо объявить и создать объекты, представляющие входные данные, организовать диалог с пользователем для ввода этих данных, обратиться к сервисам DLL для получения результата и результаты вывести на консоль. Приведу вначале код консольного проекта с построенным методом Main, а затем его прокомментирую. Вот этот код:

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

namespace ConsoleToMathTools
{
   class Program
   {
      /// <summary>
      /// Точка входа в консольный проект
      /// организация интерфейса к DLL MathTools
      /// </summary>
      /// <param name="args"></param>
      static void Main(string[] args)
      {
         //Входные данные
         double x = 0;

         const string INVITE =
            "Введите вещественное число x" +
            "- аргумент функции Sin(x)";
         const string CONTINUE =
            "Продолжим? (Yes/No)";
         string answer = "yes";
         do
         {
            //Организация ввода данных
            Console.WriteLine(INVITE);
            string temp = Console.ReadLine();
            x = Convert.ToDouble(temp);

            //Вычисления и вывод результата
            double res = 0;
            res = Math.Sin(x);
            Console.WriteLine("Math.Sin(x) = " +
               res.ToString());
```

```
            res = MathTools.MyMath.Sin(x);
            Console.WriteLine("MathTools.MyMath.Sin(x) = " +
               res.ToString());

            res = MathTools.MyMath.SinOld(x);
            Console.WriteLine("MathTools.MyMath.SinOld(x) = " +
               res.ToString());

            //диалог с пользователем
            Console.WriteLine(CONTINUE);

             answer = Console.ReadLine();
        } while (answer == "yes");
    }
  }
}
```

Дадим краткие комментарии к этому коду.
Входные данные устроены просто - задается лишь одна переменная x типа double. Помните, что в языке C# все переменные являются объектами.
Вводу значения x предшествует, как и положено для хорошего стиля программирования, приглашение к вводу. Для ввода и вывода значений используются статические методы ReadLine и WriteLine класса Console, входящего в библиотеку FCL и предоставляющего свои сервисы пользователям консольных приложений. Для преобразования введенных данных, представляющих собой строки текста, к нужному типу (в нашем случае к типу double) используются статические методы класса Convert, сервисы которого позволяют проводить различные преобразования между типами данных.

Значение функции $\sin(x)$ вычисляется тремя разными методами - методом стандартного класса Math и двумя методами класса MyMath, входящего в состав библиотеки MathTools.
Следуя правилу стиля "Имена - константам", в коде метода используются именованные константы.
Применяется стандартный прием зацикливания тела метода Main, позволяющий пользователю самому решать, когда прервать выполнение метода.
На рис. 1.8 показаны результаты работы консольного проекта.



увеличить изображение
Рис. 1.8. Результаты работы консольного проекта

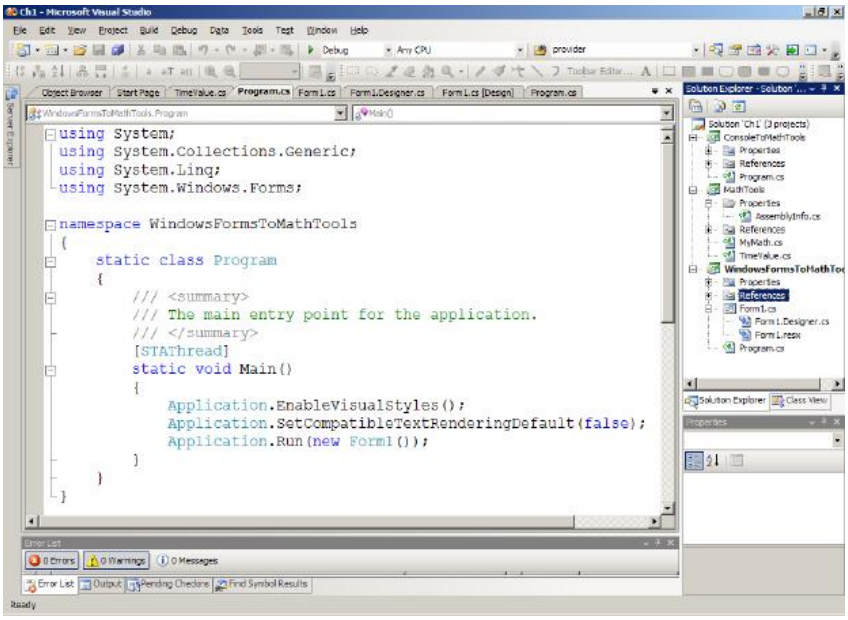
Анализируя эти результаты, можно видеть, что все три метода на всех исследуемых аргументах дают одинаковые результаты, совпадающие с точностью до 9 цифр после запятой. Точность методов в классе MyMath обеспечивается константой EPS этого класса. Достигнутая точность вполне достаточна для большинства практических задач. Остается понять, насколько написанные нами методы проигрывают методу стандартного класса по времени. Это исследование оставим для следующего проекта - Windows-проекта, обеспечивающего интерфейс, который дает пользователю больше возможностей.

Лекция 3

Тема: Основные этапы решения задач на ЭВМ. Структура программы на языке высокого уровня. Стандартные типы данных в ООЯП

Windows- проект

Добавим в Решение новый проект, аналогично тому, как был добавлен консольный проект. В качестве типа проекта выберем "Windows Forms Application", дадим проекту имя "WindowsFormsToMathTools". Результат этой работы показан на рис. 1.9.



увеличить изображение

Рис. 1.9. Решение, содержащее три проекта - Class Library, Console, Windows

При создании проекта DLL автоматически создавался в проекте один пустой класс, в консольном проекте создавался класс, содержащий метод Main с пустым кодом метода. В Windows-проекте автоматически создаются два класса - класс с именем Form1 и класс с именем Program.

Первый из этих классов является наследником класса Form из библиотеки FCL и наследует все свойства и поведение (методы и события) родительского класса. Класс Form поддерживает организацию интерфейса пользователя в визуальном стиле. Форма является контейнером для размещения визуальных элементов управления - кнопок (Button), текстовых полей (TextBox), списков (ListBox) и более экзотичных элементов - таблиц (DataGridView), деревьев (TreeView) и многих других элементов. С некоторыми элементами управления мы познакомимся уже в этом примере, другие будут встречаться в соответствующих местах нашего курса.

Классы в C# синтаксически не являются неделимыми и могут состоять из нескольких частей, каждая из которых начинается с ключевого слова "partial" (частичный). Таковым является и построенный автоматически класс Form1. Возможность разбиения описания одного класса на части появилась еще в версии языка C# 2.0, что облегчает работу над большим классом. Каждая часть класса хранится в отдельном файле со своим именем. Одна часть класса Form1 лежит в файле с именем "Form1.Designer.cs". Эта часть класса заполняется автоматически инструментарием, называемым Дизайнером формы. Когда мы занимаемся визуальным проектированием формы и размещаем на ней различные элементы управления, меняем их свойства, придаем форме нужный вид, задаем обработчиков событий для элементов управления, то Дизайнер формы транслирует наши действия в действия над объектами соответствующих классов, создает соответствующий код и вставляет его в нужное место класса Form1. Предполагается, что разработчик проекта не вмешивается в работу Дизайнера и не корректирует часть класса Form1, созданную Дизайнером. Тем не менее, понимать код, созданный Дизайнером, необходимо, а иногда полезно и корректировать его. Другая часть класса Form1, хранящаяся в файле "Form1.cs", предназначена для разработчика - именно в ней располагаются автоматически создаваемые обработчики событий, происходящих с элементами управления, код которых создается самим разработчиком. Такая технология программирования, основанная на работе с формами, называется визуальной, событийно управляемой технологией программирования.

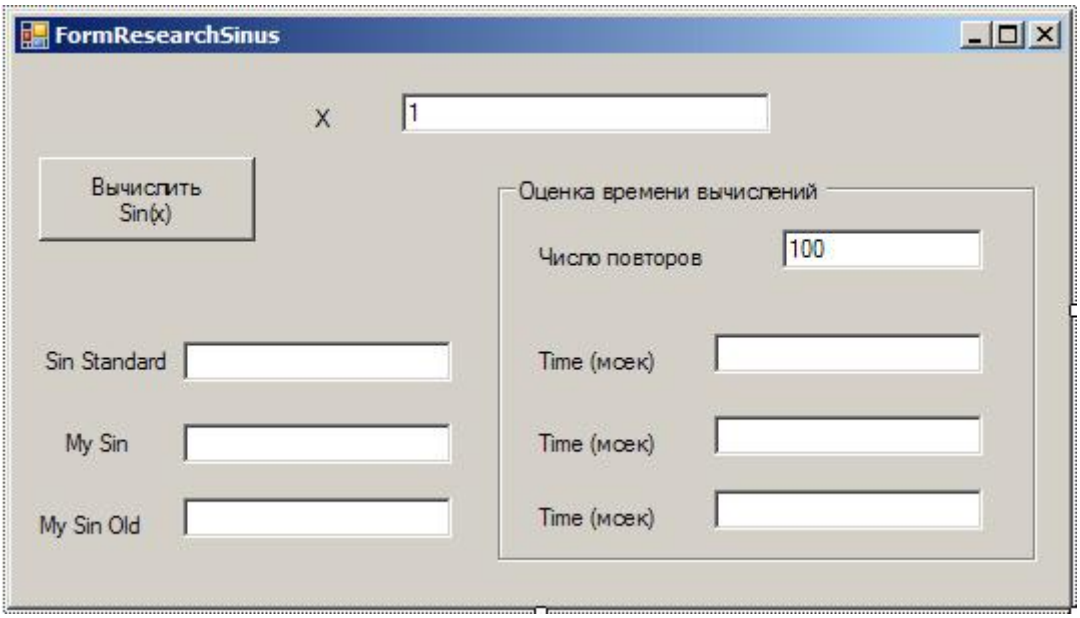
Класс Program, автоматически создаваемый в Windows-проекте, содержит точку входа - статический метод Main, о важной роли которого мы уже говорили. В отличие от консольного проекта, где тело процедуры Main изначально было пустым и должно было заполняться разработчиком проекта, в Windows-проектах процедура Main уже готова и, как правило, разработчиком не изменяется. Что же делает автоматически созданная процедура Main, текст которой можно видеть на рис. 1.9? Она работает с классом Application библиотеки FCL, вызывая поочередно три статических метода этого класса - EnableVisualStyles, SetCompatibleTextRenderingDefault, Run. О назначении первых двух методов можно судить по их содержательным именам. Основную работу выполняет метод Run - в процессе его вызова создается объект класса Form1 и открывается форма - визуальный образ объекта, с которой может работать конечный пользователь проекта. Если, как положено, форма спроектирована и заполнена элементами управления, то конечному пользователю остается вводить собственные данные в поля формы, нажимать на кнопки, вообще быть инициатором возникновения различных событий в мире объектов формы. В ответ на возникающие события начинают работать обработчики событий, что приводит к желаемым (или не желанным) изменениям мира объектов. Типичной ситуацией является проведение вычислений по данным, введенным пользователем, и отображение результатов этих вычислений в полях формы, предназначенных для этих целей.

Построение интерфейса формы

Прежде чем заняться построением интерфейса формы, переименуем класс Form1, дав ему, как положено, содержательное имя - FormResearchSinus. Заметьте, переименование объектов класса хотя и можно делать вручную, но это далеко не лучший способ, к тому же,чреватый ошибками. Для этих целей следует использовать возможности, предоставляемые меню Refactor|Rename. Параллельно с переименованием класса следует переименовать и файл (файлы) с описанием класса.

Займемся теперь построением интерфейса - размещением в форме элементов управления. Классическим примером интерфейса, поддерживающего сервисы стандартного класса Math, является инженерный калькулятор. В нашем классе реализована пока только одна функция - $\sin(x)$, так что можем построить пока калькулятор одной функции. Но и цели у нас другие - мы занимаемся исследованием того, насколько корректно и точно предложенные алгоритмы позволяют вычислить эту функцию.

Проведем еще одно важное исследование - оценим время, затрачиваемое на вычисление функции. Временные оценки работы проекта и его отдельных частей - крайне важная часть работы разработчика проекта. Во многих случаях требуется построить временной профиль работы проекта, выявить его наиболее узкие места, на выполнение которых уходит основное время работы, что позволит целенаправленно заниматься оптимизацией проекта, направленной на уменьшение времени работы. Следует помнить, что интерактивный стиль работы современных приложений требует быстрой реакции системы на действия пользователя. Пользователь имеет право задумываться при выборе своих действий, но от системы в большинстве случаев ждет немедленного ответа. Так что поставим цель - получить время, затрачиваемое компьютером на вычисление функции как стандартным методом класса Math, так и методами класса MyMath из библиотеки MathTools.



На рис. 1.10 показан интерфейс спроектированной формы.

Рис. 1.10. Интерфейс формы класса FormResearchSinus

Для наших целей достаточен скромный интерфейс. В форму включено текстовое поле для ввода значения аргумента x, три текстовых поля предназначены для отображения результата вычислений функции $\sin(x)$ тремя различными методами. В форме есть отдельный контейнер для оценки временных характеристик. В контейнер помещены три текстовых поля, в которых будет отображаться время, затрачиваемое на вычисление функции каждым из анализируемых методов. Поскольку компьютеры быстрые, замерить время, требуемое на однократное вычисление функции, просто невозможно. Замеряется время, затрачиваемое на многократное выполнение метода (отдельного участка кода). В контейнере размещено окно, позволяющее задать число повторов вычисления функции при измерении времени работы. Все текстовые поля снабжены метками, проясняющими смысл каждого поля. Для входных текстовых полей (аргумент функции и число повторов) заданы значения по умолчанию. В форме находится командная кнопка, щелчок по которой приводит к возникновению события **Click** этого объекта, а обработчик этого события запускает вычисление значений функции, получение оценок времени вычисления и вывод результатов в соответствующие текстовые поля. Каков сценарий работы пользователя? Когда при запуске проекта открывается форма, пользователь может в соответствующих полях задать значение аргумента функции и число повторов, после чего нажать кнопку с надписью "Вычислить sin(x)". В выходных текстовых полях появятся результаты вычислений. Меняя входные данные, можно наблюдать, как меняются результаты вычислений. Можно будет убедиться, что при всех задаваемых значениях аргумента функции значения функции, вычисленные тремя разными методами, совпадают с точностью до 9 знаков после запятой, а время вычислений метода, встроенного в стандартный класс Math, примерно в два раза меньше, чем время спроектированных нами методов, что ,впрочем, не удивительно и вполне ожидаемо. Реализация вычисления стандартных математических функций реализована на аппаратном уровне, поэтому практически невозможно написать собственный код, работающий эффективнее.

Что дает оптимизация метода, рассмотренная нами в классе MyMath? Оценить это не так просто, поскольку при оценке времени работы возможны погрешности, измеряемые десятками миллисекунд, что сравнимо с выигрышем, полученным в результате оптимизации. Об этом мы еще поговорим чуть позже.

Как оценить время работы метода

Давайте подумаем, как можно оценить время работы метода класса или отдельного фрагмента кода. Во-первых, можно провести теоретическую оценку. Например, для функции $\sin(x)$, как мы видели, на одном шаге цикла требуется около 10 операций (не учитывая разную сложность операций). Число итераций зависит от значения аргумента. Максимальное значение аргумента по модулю не превышает 2π, но и в этом случае 15 итераций достаточно, чтобы текущий член суммы по модулю стал меньше 10^{-9}. Современному компьютеру средней мощности с частотой 1,6 GHz потребуется менее 1 секунды для вычисления функции при числе повторов 10^6.

Чем считать операции, зачастую проще непосредственно измерить реальное время вычислений. В библиотеке CLR для этих целей создан класс DateTime, позволяющий работать с датами и временами. У этого класса есть замечательный статический метод Now, вызов которого возвращает в качестве результата объект класса DateTime, задающий текущую дату и текущее

время (по часам компьютера). Многочисленные свойства этого объекта - Year, Month, Hour, Second и многие другие позволяют получить все характеристики даты и текущего времени. Текущее время можно также измерять и в единицах, называемых "тиками", где один тик равен 100 наносекунд или, что то же, 10^{-7} секунды.
Имея в своем арсенале такой класс, не составит большого труда измерить время, требуемое на выполнение некоторого участка кода. Достаточно иметь две переменные с именами, например, start и finish класса DateTime. Переменой start присвоим значение, возвращаемое функцией Now перед началом измеряемого участка кода, а переменной finish - в конце участка кода. Разность времен даст нам требуемую оценку длительности выполнения кода.
Некоторый недостаток этого подхода состоит в том, что рабочий код нужно дополнять операторами, задействованными только для проведения исследований. Поэтому хочется иметь более удобный инструментарий. Покажу возможный образец построения подобного инструмента. Поместив его в созданную нами библиотеку классов DLL MathTools, обеспечим тем самым возможность повторного использования.
Добавим в эту библиотеку новый класс. Вот описание этого класса, пока пустого, но содержащего заголовочный комментарий:

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

namespace MathTools
{
   /// <summary>
   /// Класс спроектирован для получения оценок времени
   /// выполнения различных методов.
   /// Встроенные делегаты определяют сигнатуры методов
   /// </summary>
   public class TimeValue
   {
                        }}
```

Уточним нашу цель. Мы хотим создать удобный инструмент, позволяющий оценивать время работы исследуемых методов. Вместо того чтобы в работающей программе окружать вызовы этих методов специальными операторами, напишем специальную процедуру, производящую оценку времени работы, передавая ей в качестве параметра имя исследуемого метода. Стоит сказать несколько слов о том, как написать процедуру (метод), которой в качестве параметра можно передавать имя метода.
Предварительные сведения о делегатах - функциональном типе данных
Уже говорилось, что одна из главных ролей класса состоит в том, чтобы задать описание типа данных. Каждый тип данных характеризует некоторое множество объектов - экземпляров класса. Класс, позволяющий описать некоторое множество объектов, каждый из которых является функцией, называется функциональным типом. В языке C# для описания функциональных типов используются классы, называемые делегатами, описание которых начинается с ключевого слова - delegate. Делегаты играют важную роль в языке C#, и их описанию будет уделено достойное внимание. Пока что нам достаточно знать, как выглядит описание делегата и как оно используется во многих задачах. Описание делегата представляет описание сигнатуры функций, принадлежащих одному функциональному типу. Под сигнатурой функции понимается описание числа, порядка и типов аргументов функции и типа возвращаемого значения. В языках программирования заголовок функции определяет ее сигнатуру. Пусть задан делегат

```
public delegate double DToD(double arg1);
```

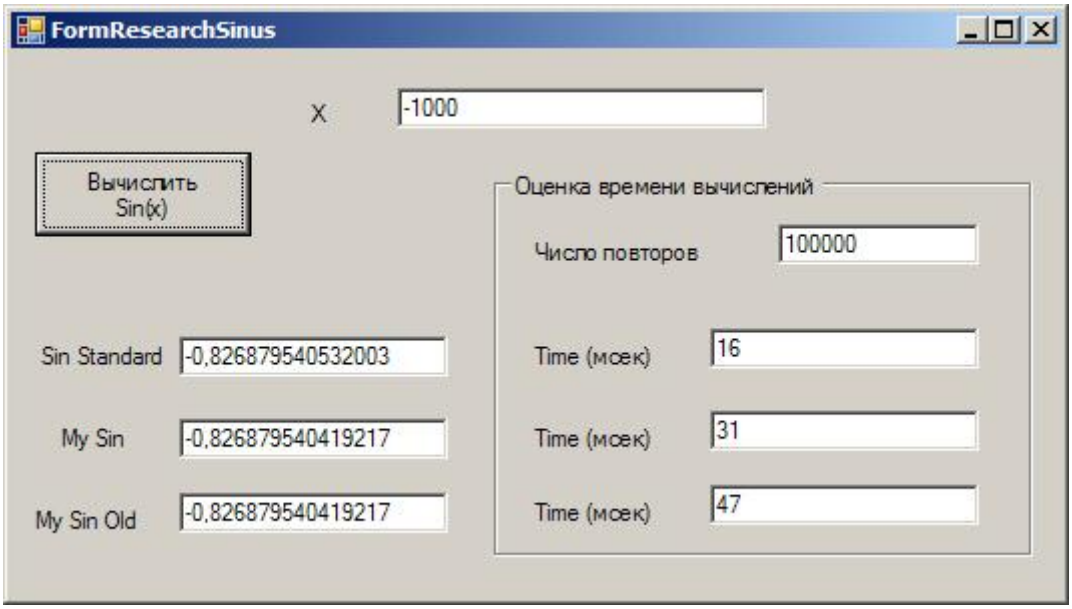
Этот делегат задает описание класса с именем DToD (Double To Double), которому принадлежат все функции с одним аргументом типа double и возвращающие результат типа double. Функция $\sin(x)$, как и многие другие математические функции, соответствует этой сигнатуре и, следовательно, является объектом этого класса.
Если задан делегат, то появляется возможность объявлять объекты этого класса, в частности, формальный аргумент метода может принадлежать такому классу, а в качестве фактического аргумента в момент вызова можно передавать имя конкретной функции, принадлежащей данному функциональному типу. Пример, иллюстрирующий эту возможность, сейчас будет продемонстрирован. Но прежде одно важное замечание о методах, процедурах и функциях.
Формально у классов языка C# есть только методы и нет ключевых слов для таких понятий, как процедуры и функции. Фактически же любой метод представляет собой либо процедуру, либо функцию. Существуют синтаксические и содержательные различия в описании методов, представляющих процедуры и функции, в способах их вызова и применения. Подробнее об этом поговорим в соответствующем разделе курса, сейчас же заметим только, что в зависимости от контекста будем использовать как термин "метод", так и термины "процедура" и "функция".
Класс TimeValue
Теперь уже можно привести код класса TimeValue со встроенным делегатом DToD, предоставляющий своим клиентам такой сервис, как оценка времени работы любого метода клиента, сигнатура которого согласована с делегатом. При необходимости этот класс всегда можно расширить, добавив соответствующие сервисы и новые делегаты. Вот этот код:

```
   public class TimeValue
   {
      public delegate double DToD(double arg1);
```

```
/// <summary>
/// Возвращает время в секундах,
/// затраченное на вычисление count раз
/// метода fun с сигнатурой, которая удовлетворяет
/// делегату DToD (double to double)
/// </summary>
/// <param name="count">число повторений</param>
/// <param name="fun">имя функции</param>
/// <param name="x">аргумент</param>
/// <returns>время в милисекундах или тиках</returns>
public static double EvalTimeDToD(int count, DToD fun, double x)
{
   DateTime start, finish;
   double res = 0;
   start = DateTime.Now;
      for (int i = 1; i < count; i++)
         fun(x);
   finish = DateTime.Now;
   //res = (finish- start).Ticks;
   res = (finish - start).Milliseconds;
   return res;    }       }
```

Время можно измерять в разных единицах, например, в тиках или миллисекундах. Статический метод EvalTimeDToD, реализующий сервис класса, устроен достаточно просто. Две переменные start и finish класса DateTime вызывают свойство Now, окаймляя цикл по числу повторов вызовов метода, функциональный тип которого задан делегатом, а имя передается в качестве фактического параметра при вызове метода EvalTimeDToD.

Еще одно важное замечание стоит сделать по поводу точности оценок, получаемых при использовании механизма объектов DateTime. Следует учитывать, что свойство Now не возвращает в точности текущее время в момент ее вызова. Это связано с механизмами операционной системы, когда в реальности на компьютере работают несколько процессов и система обработки прерываний имеет некоторый фиксированный квант времени при переключении процессов. Поэтому, запуская измерение времени вычислений на одних и тех же данных, можно получать различные данные с точностью, определяемой характеристиками системы прерываний. Это существенно не влияет в целом на получение временных характеристик, но не позволяет сравнивать методы, время выполнения которых сравнимо с погрешностью временных оценок.



В заключение приведем результаты вычислений и временных оценок, полученных для нашего примера.

Рис. 1.11. Сравнительные результаты точности и времени вычисления функции sin(x)

Лекция 4

Тема: Стандартные типы данных в ООЯП

Знакомство с новым языком программирования разумно начинать с изучения системы типов этого языка. Как в нем устроена система типов данных? Какие имеются простые типы, как создаются сложные, структурные типы, как определяются собственные типы, динамические типы? Для объектно-ориентированных языков программирования важно понимать, как связаны между собой такие близкие по духу понятия, как понятие типа и понятие класса.

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, утверждалось, что тип integer задает целые числа в некотором диапазоне. Неявно с типом всегда связывался и набор разрешенных операций. В строго типизированных языках, к которым относится большинство языков программирования, каждая переменная в момент ее объявления связывалась с некоторым типом. Связывание переменной x с типом T означало, что переменная x может принимать только значения из множества, заданного типом T, и к ней применимы операции, разрешенные этим типом. Таким образом, тип определял, говоря современным языком, свойства и поведение переменных. Значение переменной задавало ее свойства, а операции над ней - ее поведение, то есть то, что можно делать с этой переменной.

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 уже тогда содержало четкое определение операций или методов, применимых к данным. Классы стали естественным обобщением понятия типа, а объекты - экземпляры класса - стали естественным обобщением понятия переменной. Сегодня определение класса в C# и других объектных языках содержит:

- поля, синтаксически представляющие объявления переменных и задающие свойства объектов класса;
- методы, определяющие поведение объектов класса;
- события, которые могут происходить с объектами класса.

Так есть ли различие между основополагающими понятиями - типом и классом, переменной и объектом? Такое различие существует. Программистам нужны все эти понятия. Но определить это различие не так-то просто. Мне до сих пор не удается точно описать все ситуации, в которых следует использовать только понятие "тип", и ситуации, в которых приемлемо только применение понятия "класс". Во многих ситуациях эти понятия становятся синонимичными. Если, например, есть объявление <T x;>, то можно говорить, что объявлена переменная типа T, но столь же справедливо утверждение, что данное объявление задает объект x класса T. На первых порах можно считать, что класс - это хорошо определенный тип данных, объект - хорошо определенная переменная.

Есть традиционные предпочтения. Базисные встроенные типы, такие, как int или string, предпочитают называть по-прежнему типами, а их экземпляры - переменными. Когда же речь идет о создании собственных типов, моделирующих, например, такие абстракции данных, как множество автомобилей или множество служащих, то естественнее говорить о классах Car и Employee, а экземпляры этих классов называть объектами.

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 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. Принято считать, что встроенными в язык C# являются арифметические типы, булевский и строковый тип, что также в язык встроен механизм построения массивов из переменных одного типа. Эти встроенные типы будем называть базисными. Базисные встроенные типы должны быть реализованы любым компилятором, отвечающим стандарту языка C#.

Язык C#, рассматриваемый в данном курсе, изначально предполагает реализацию, согласованную с Framework .Net. Это означает, что все базисные встроенные типы проецируются на соответствующие типы библиотеки FCL. Библиотека FCL реализует базис языка C#. Но помимо этого, она предоставляет в распоряжение программиста множество других полезных типов данных. Так что для нашей реализации языка C# встроенных типов огромное число - вся библиотека FCL. Знать все типы из этой библиотеки практически невозможно, но умение ориентироваться в ней необходимо.

Типы данных разделяются также на **статические** и **динамические**. Для переменных статического типа память под данные отводится в момент объявления, требуемый размер данных известен при их объявлении. Для динамического типа размер данных в момент объявления обычно не известен, и память им выделяется динамически в процессе выполнения программы. Многие динамические типы, доступные разработчикам проектов на C#, реализованы как встроенные типы в библиотеке FCL. Например, в пространстве имен этой библиотеки System.Collections находятся классы Stack, Queue, ListArray и другие классы, описывающие широко распространенные динамические типы данных - стек, очередь, список, построенный на массиве.

Возможно, наиболее важным для C# программистов является деление типов на **значимые** и **ссылочные**. Для значимых типов, называемых также **развернутыми**, значение переменной (объекта) является неотъемлемой собственностью переменной (точнее, собственностью является память, отводимая значению, само значение может изменяться). Значимый тип принято называть развернутым, подчеркивая тем самым, что значение объекта развернуто непосредственно в памяти, отводимой объекту.

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. О ссылочных и значимых типах еще предстоит обстоятельный разговор.

Для большинства процедурных языков, реально используемых программистами, - Паскаль, C++, Java, Visual Basic, C#, - базисная 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 знание, по крайней мере, одного из процедурных языков позволяет построить общую картину базисной системы типов и для языка C#. Язык C# многое взял от языка C++, системы базисных 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#. Во многом это устройство заимствовано из 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** (class).

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 определенные пользователем.

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

- типы-значения (value), или значимые типы;
- ссылочные (reference);
- указатели (pointer);
- тип void.

Эта классификация основана на том, где и как хранятся значения типов. Переменные или, что то же в данном контексте, - объекты, хранят свои значения в памяти компьютера, которую будем называть памятью типа "Стек" (Stack). Другой вид памяти, также связанный с хранением значений переменной, будем называть памятью типа "Куча" (Heap). Об особенностях этих двух видов памяти поговорим позже. Сейчас для нас важно понимать следующее. Для значимого типа значение переменной хранится непосредственно в стеке. Поскольку значение может быть сложным и состоять, например, из множества скалярных значений, говорят, что значение разворачивается в стеке. По этой причине значимый тип называют также **развернутым** типом. Для ссылочного типа значение в стеке задает ссылку на область памяти в "куче", где хранятся собственно данные, задающие значение. Данные, хранящиеся в куче, в этом случае называют объектом, а значение, хранящееся в стеке - ссылкой на объект. Самое важное в этой модели хранения значений - это то, что разные ссылки в стеке могут указывать на один и тот же объект из кучи. И тогда у этого объекта существует много разных имен (псевдонимов), каждое из которых позволяет получить доступ к полям объекта и изменять хранящиеся там значения.

В отдельную категорию выделены указатели, что подчеркивает их особую роль в языке. Указатели и ссылки в языке C# хотя и возможны, но в большинстве проектов используются редко. Отказ от этих средств делает программы более простыми, а самое главное более надежными. В нашем курсе эти средства практически появляться не будут. Ситуация похожа на ситуацию с оператором Goto - оператор доступен в языке, но не рекомендован к использованию. Если программист C# действительно хочет применять эти средства, то ему придется предпринять для этого определенные усилия, поскольку 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. Типы - **логический, арифметический, структуры, перечисление** - относятся к значимым типам. **Массивы**, **строки** и **классы** относятся к

ссылочным типам. На первый взгляд, такая классификация может вызывать некоторое недоумение, почему это структуры 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. Поэтому естественно хранить массивы в динамической памяти - куче, а не в статической памяти, каковой является стек, где размеры хранимых данных не меняются в процессе выполнения. 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. Два объявления типа данных могут отличаться лишь одним ключевым словом, начинающим это объявление, - class или struct. В зависимости от того, какое ключевое слово использовано, данное объявление будет задавать класс (ссылочный тип) или структуру (значимый тип). Определив тип 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 не совсем полноценное средство, поскольку на структуры накладываются дополнительные и довольно жесткие ограничения по сравнению с обычными классами. В частности, для структур разрешено только наследование интерфейсов, и структура не может иметь в качестве родителя класс или структуру. Все развернутые типы языка C# - int, double и прочие - реализованы как структуры.

Все базисные 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 базисных 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, содержащую описание базисных встроенных типов языка C# и их основные характеристики.

Таблица 2.1. Базисные встроенные типы языка C#

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.Int16	[-32768, 32767]	Знаковое, 16-бит
ushort	System.UInt16	[0, 65535]	Беззнаковое, 16-бит
int	System.Int32	$[-2^{31}, 2^{31}]$	Знаковое, 32-бит
uint	System.UInt32	$[0, 2^{32}]$	Беззнаковое, 32-бит
long	System.Int64	$[-2^{63}, 2^{63}]$	Знаковое, 64-бит
ulong	System.UInt64	$\approx (0, 2^{64})$	Беззнаковое, 64-бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон (по модулю)**	**Точность**
float	System.Single	$[10^{-45}, 10^{38}]$	7 цифр
double	System.Double	$[10^{-324}, 10^{308}]$	15-16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон (по модулю)**	**Точность**
decimal	System.Decimal	$[10^{-28}, 10^{28}]$	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 базисных 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 уже перешел на язык C#.

Переменные, объекты и сущности

Уже говорилось о том, что одна из главных ролей, которую играют классы в ОО-языках, - это роль типа данных. Класс, рассматриваемый как тип данных, задает описание свойств, поведения и событий некоторого множества элементов, называемых экземплярами класса, а чаще переменными или объектами. Заметьте, класс - это описание, это текст - статическая конструкция. Чтобы программа могла выполняться, в ней должны быть определены переменные или, что то же, объекты класса, создаваемые динамически в ходе выполнения программы. Напомню: в первой лекции объяснялось, что в начальный момент работы программы - в момент "большого взрыва" - создается первый объект, который становится текущим объектом, и начинает работать метод Main - точка входа в программу. Все дальнейшее зависит от содержания метода Main: какие новые объекты создаются, какие методы и свойства вызываются этими объектами.

В строго типизированных языках всякая переменная до ее использования должна быть явно объявлена в программе. В момент объявления должно указываться имя переменной и тип, которому принадлежит эта переменная. Тип, задаваемый в момент объявления, называется базовым типом этой переменной.

В этом тексте вместо переменных можно говорить об объектах, вместо типов - о классах. Мало того, что у нас уже есть два слабо различимых понятия - объект и переменная, введем еще одно близкое понятие - понятие сущности, понимая под сущностью то имя, которое появляется непосредственно в тексте программы. Объекты и переменные - это уже динамически созданные реалии, которым отведена память. В процессе работы программы сущность связывается с физически создаваемым в стеке или в куче объектом (переменной). Объект, созданный в памяти компьютера и связанный с сущностью, может иметь тип, согласованный, но не обязательно совпадающий с базовым типом сущности.

Синтаксис объявления

Неформально уже отмечалось, что в момент объявления переменной указывается ее тип, имя и, возможно, значение. Давайте попробуем выразить это более формально. **Синтаксис объявления** сущностей в C# может быть задан следующей синтаксической формулой:

[<атрибуты>] [<модификаторы>] <тип> <список объявителей>;

В синтаксических формулах используется широко распространенный метаязык, известный еще со времен описания синтаксиса языка Алгол. В угловые скобки заключаются синтаксические понятия языка. В квадратные скобки заключаются необязательные элементы формулы.

О необязательных элементах этой формулы - атрибутах и модификаторах - будем говорить по мере необходимости, пока же будем считать, что эти элементы при объявлении сущности опущены. Список объявителей позволяет в одном объявлении задать несколько переменных одного типа. Терминальный символ "запятая" служит разделителем элементов списка. Элемент списка **объявитель** задается следующей синтаксической формулой:

<имя> | <имя> = <инициализатор>

Когда задается просто имя переменной, такое объявление называется объявлением без инициализации. Во втором случае переменная не только объявляется, но и инициализируется ее начальное значение.

О типах языка C# мы уже кое-что знаем, так что готовы уточнить, как строятся объявления сущностей в простых случаях.

Объявления простых переменных

Если тип в синтаксической формуле задавать именем типа из таблицы 2.1, это означает, что объявляются простые скалярные переменные. Объявления переменных сложных типов - массивов, перечислений, структур и других типов, определяемых программистом, - будут изучаться в последующих лекциях.

Как строится инициализатор для простых переменных? Инициализатор, как чаще всего и бывает, задается некоторым выражением - обычно константой, но может быть, достаточно сложным выражением, допускающим вычисление на этапе компиляции программы и зависящим от ранее инициализированных переменных. Инициализатор может быть задан и в объектном стиле. В этом случае он представляет вызов конструктора объектов соответствующего типа. Синтаксическая формула для инициализатора в этом случае имеет вид:

new <имя конструктора>([<список аргументов>])

Такая конструкция инициализатора, применимая и для скалярных переменных базисных типов, подчеркивает, что в C# все переменные как значимых, так и ссылочных типов, простые и сложные, являются настоящими объектами соответствующих классов.

Хороший стиль программирования требует, чтобы все простые переменные объявлялись с инициализацией. Дело в том, что при объявлении без инициализации значение переменной остается неопределенным, и всякая попытка использовать такую переменную в вычислениях, где требуется значение переменной, является ошибкой, которую компилятор обнаруживает еще на этапе компиляции. Конечно, можно объявить переменную без инициализации, а потом в процессе вычислений присвоить ей значение, тем самым инициализируя ее.

Заметьте, компилятор строго следит за тем, чтобы в вычислениях не появлялись переменные, не имеющие значения. Если присвоение значения переменной происходит внутри одной из ветвей оператора if или в теле оператора цикла, то компилятор предпочитает сигнализировать об ошибке в подобных ситуациях. Он ориентируется на худший случай, поскольку не может на этапе компиляции разобраться, будет ли выполняться ветвь с инициализацией, будет ли выполняться тело цикла или не будет.

Типы, допускающие неопределенные значения

Инициализация переменных позволяет в момент объявления каждой переменной дать значение, принадлежащее множеству возможных значений данного типа. Тем самым исключается возможность работы с неопределенными значениями переменных. Однако для ссылочных типов возможным значением объектов является значение null, задающее неопределенную ссылку. Конечно, попытка вызвать метод или свойство объекта со значением null не приведет к успеху и возникнет ошибка периода выполнения. Тем не менее, полезно для ссылочных типов иметь null в качестве возможного значения.
Для значимых типов значение null не входит в множество возможных значений. Но в ряде ситуаций полезно, чтобы переменная значимого типа имела неопределенное значение. Язык C# позволяет из любого значимого типа данных построить новый тип, отличающийся лишь тем, что множество возможных значений дополнено специальным значением null. Так построенные типы данных называются типами, допускающими неопределенное значение (Nullable Types). Если построен тип T, то тип, допускающий неопределенные значения, определяется следующим образом:
System.Nullable<T>

Чаще используется эквивалентная, но более простая форма записи -
T?

Переменные таких типов могут получать значение null либо в результате присваивания, либо в процессе вычислений. Понятно, что если при вычислении выражения один из операндов будет иметь значение null, то и результат вычисления выражения будет иметь то же значение null. Над переменными этого типа определена специальная операция склеивания:
A ?? B

Результатом вычисления этого выражения будет операнд A, если значение A не равно null, и B, если первый операнд равен null.
Рассмотрим выполнение преобразований между типами T? и T. Очевидно, что преобразование из типа T в тип T? - безопасное преобразование и потому может выполняться неявно. В другую сторону преобразование является опасным и должно выполняться явно, например, путем кастинга - приведения к типу. Рассмотрим некоторые примеры работы с переменными подобных типов.

```
static void TestNullable()
{
   int x = 3, y = 7;
   int? x1 = null, y1, z1;
   y1 = x + y;
   z1 = x1 ?? y1;
   Console.WriteLine("x1 = {0}, y1 = {1}, z1 = {2}",
      x1, y1, z1);
```

В этом фрагменте вводятся переменные типа int? и int. Демонстрируется безопасное преобразование из типа int в тип int? и выполнение операции ?? - операции склеивания. Рассмотрим следующий фрагмент тестового метода:

```
//x = (int)x1;
y = (int)y1;
Console.WriteLine("x = {0}, y = {1}",
   x, y);
z1 = x1 * y ?? y1;
y1 = z1 - y1;
Console.WriteLine("x1 = {0}, y1 = {1}, z1 = {2}",
   x1, y1, z1);
```

Первая строка фрагмента закомментирована, поскольку попытка явного приведения типа переменной со значением null приведет к ошибке периода выполнения. В следующей строчке такое приведение успешно выполняется, поскольку переменная y1 имеет значение, допустимое для типа int. Заметьте, что операция ?? имеет более низкий приоритет, чем операция умножения, поэтому первый операнд этой операции будет иметь значение null и z1 получит значение y1. В следующем фрагменте демонстрируются оба эквивалентных способа задания типа double, допускающего неопределенные значения:

```
System.Nullable<double> u = x + x1;
double? v = y + y1, w;
w = u ?? v + y1;
Console.WriteLine("u = {0}, v = {1}, w = {2}",
u, v, w);
```

В заключение взгляните на результаты работы этого тестового примера.

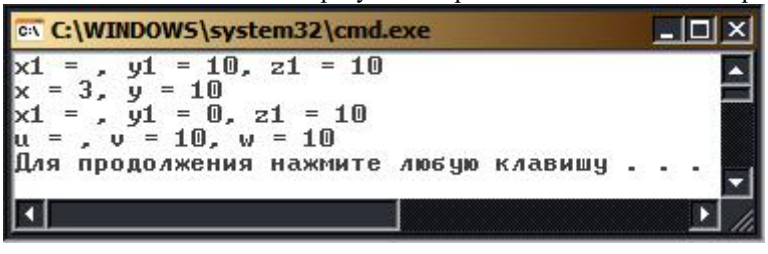


Рис. 2.1. Типы, допускающие значение null
Заметьте, при выводе на печать значение null не имеет специального обозначения и выводится как пробел.
Null, NaN и Infinity

Значение null не является единственным особым значением, входящим в множество возможных значений значимого типа. У вещественного типа данных (double и float) есть и другие особые значения, не являющиеся обыкновенными числами. Таких значений три - Infinity, NegativeInfinity и NaN. Первые два хорошо известны из математики - это бесконечность и отрицательная бесконечность. Третье значение NaN (Not a Number) появляется тогда, когда результат не является вещественным числом или значением null и Infinity. Рассмотрим правила, приводящие к появлению особых значений.
Если при выполнении операций умножения или деления результат по модулю превосходит максимально допустимое число, то значением является бесконечность или отрицательная бесконечность в зависимости от знака результата. У типов double и float есть константы, задающие эти значения. При выполнении операций сложения, вычитания и умножения бесконечности с обычными вещественными числами результат имеет значение бесконечность, возможно, с противоположным знаком. При делении вещественного числа на бесконечность результат равен нулю.
Если один из операндов вычисляемого выражения есть null, а остальные - обычные вещественные числа или бесконечность, то результат выражения будет иметь значение null - не определен.
Если бесконечность делится на бесконечность или ноль умножается на бесконечность, то результат будет NaN. Этот же результат будет появляться, когда результат выполнения некоторой операции не будет вещественным числом, например, при извлечении квадратного корня из отрицательного числа. Если NaN участвует в операциях, то результатом будет NaN. Это верно и тогда, когда другие операнды имеют значение null или бесконечность.
Рассмотрим примеры:

```
static void NullAndNan()
{
   double? u = null, v = 0, w = 1.5;
   Console.WriteLine("u = {0}, v = {1}, w = {2}",
      u, v, w);
```

Пока что введены три переменные типа double?, одна из которых получила значение null. Введем еще несколько переменных этого же типа, которые получат в результате вычислений особые значения:

```
double? x, y, z;
x = u + v; y = w / v; z = x + y;
Console.WriteLine("x = u + v = {0}, y = w / v = {1}, " +
" z = x + y = {2}", x, y, z);
```

При вычислении значения переменной x в выражении участвует null, поэтому и x получит значение null. При вычислении значения переменной y выполняется деление на ноль, поэтому y получит значение бесконечность. При вычислении значения переменной z в выражении участвует null и бесконечность, поэтому z получит значение null. Рассмотрим еще один фрагмент кода:

```
x = -y; y = v * y; z = x + y;
Console.WriteLine("x = -y = {0}, y = v * y = {1}, " +
" z = x + y = {2}", x, y, z);
```

При вычислении значения переменной x происходит смена знака и x получает значение отрицательной бесконечности. При вычислении значения переменной y бесконечность умножается на ноль, результат не определен и будет иметь значение NaN. При сложении бесконечности со значением NaN результат будет NaN. Ну и еще один заключительный фрагмент:

```
double p = -(double)w, q = double.NegativeInfinity;
Console.WriteLine("p = {0}, q = {1}, 1 / q = {2}",
   Math.Sqrt(p), q, 1 / q);
p = 1e160;
    Console.WriteLine("p = {0}, p * p = {1}", p, p * p);
float p1 = 1e20f;
Console.WriteLine("p1 = {0}, p1 * p1 = {1}", p1, p1 * p1);
```

Здесь вводятся переменные типа double и float. Показано, как при умножении появляются значения бесконечности, а также появляется одна из соответствующих констант, задающих бесконечность. В заключение взгляните на результаты работы этого тестового примера.

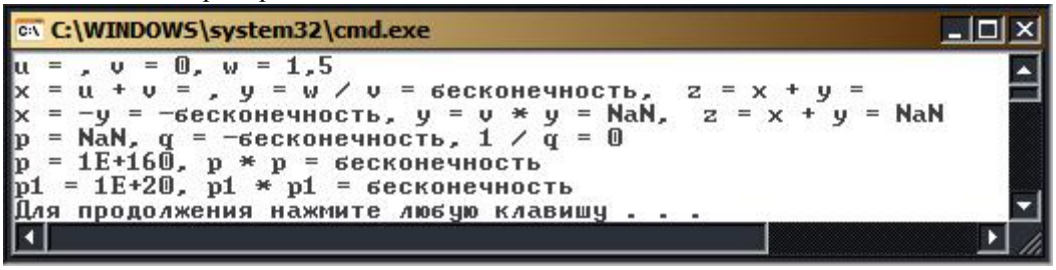


Рис. 2.2. Значения null, NaN, Infinity

Программный проект SimpleVariables

Продолжу приводить примеры и сейчас подробнее остановлюсь на том, как строилось Решение (Solution) с именем Ch2, в котором размещены проекты этой лекции. Наряду с консольным проектом ConsoleNullable, примеры из которого приводились в предыдущем разделе, создадим Windows-проект с именем SimpleVariables. В этом проекте почти не будет никаких вычислений. Его главная задача - продемонстрировать различные способы объявления простых переменных встроенных базисных типов, простейшие присваивания и вывод значений переменных.

Побочная цель состоит в том, чтобы показать работу проекта, включающего несколько форм - интерфейсных классов. Проект продемонстрирует часто встречающуюся на практике ситуацию, когда в нем есть главная кнопочная форма с множеством командных кнопок, обработчики событий которых открывают формы, решающие специальные задачи.
Поскольку в первой лекции подробно рассказывалось о том, как создаются Windows-проекты и первые шаги работы с ними, останавливаться на этом уже не буду. На рис. 2.3 показана главная форма на начальном этапе проектирования.

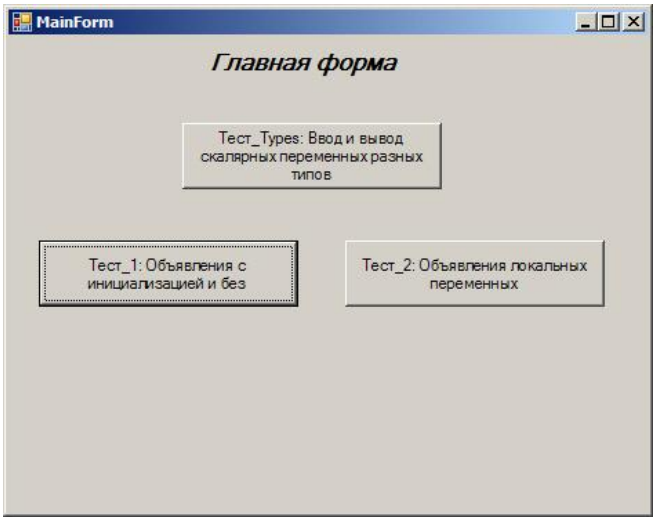


Рис. 2.3. Главная форма на начальных шагах процесса ее проектирования

Что же было сделано при проектировании главной формы? Прежде всего, она была переименована и получила содержательное имя. Переименован был и файл, содержащий описание интерфейсного класса, задающего форму. На форме размещена поясняющая надпись "Главная форма" и три командные кнопки.
Для задания надписей, поясняющих назначение формы, можно использовать элемент управления Label или текстовое окно - TextBox. В нашем примере использовалась метка.

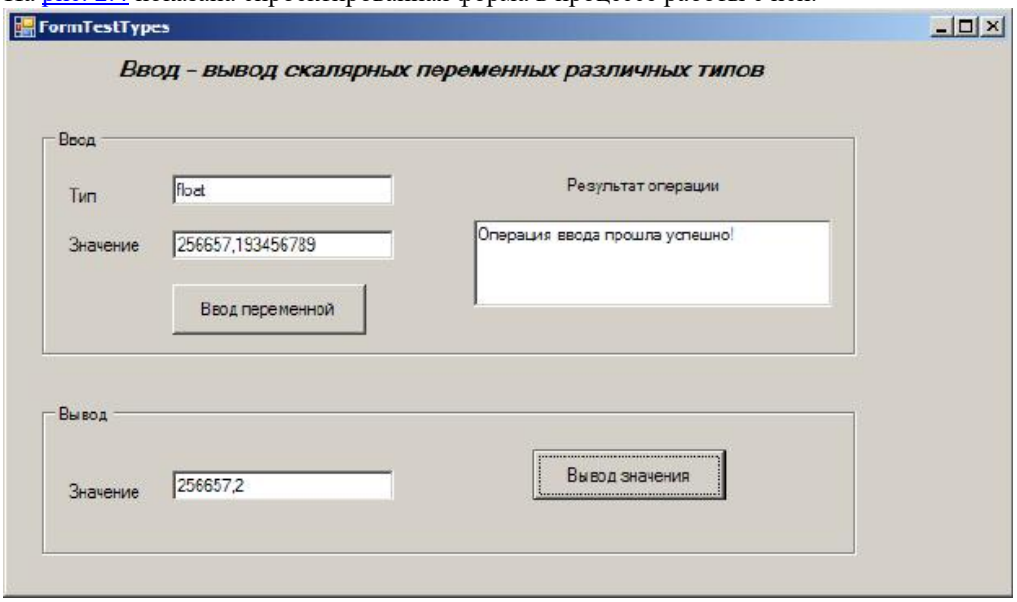
Главная кнопочная форма

Проектируемая главная форма является примером главной кнопочной формы. В каких ситуациях имеет смысл проектировать главную форму как главную кнопочную форму? Так поступают достаточно часто. Представьте себе, что создаваемый проект предоставляет конечному пользователю несколько различных сервисов, и пользователь, начиная работу с проектом, выбирает нужный ему сервис. Главная форма может иметь меню, команды которого и позволяют пользователю выбирать нужный ему сервис. Если каждый сервис достаточно сложен и требует собственного интерфейса, то в таких ситуациях вместо стандартного меню удобнее использовать главную кнопочную форму. Роль команд меню в ней играют расположенные в форме командные кнопки. Выбор командной кнопки на форме соответствует выбору команды меню.
Подведем итоги. Если в проекте предполагается n различных сервисов, каждый из которых требует собственного интерфейса, то в проект наряду с главной формой, создаваемой по умолчанию в момент создания проекта, добавляются n интерфейсных классов - наследников класса Form, каждый из которых имеет собственную форму. Каждая такая форма заселяется элементами управления, задавая интерфейс соответствующего сервиса. На главной форме располагаются n командных кнопок, а в код интерфейсного класса, задающего главную форму, добавляются n полей, каждое из которых содержит объявление объекта соответствующего интерфейсного класса. Когда пользователь выбирает в главной форме командную кнопку, то обработчик события Click этой кнопки вызывает конструктор интерфейсного класса и создает реальный объект этого интерфейсного класса и реальную форму, связанную с объектом. Затем в обработчике вызывается метод Show этого объекта, соответствующая форма открывается, показывается на экране дисплея, и пользователь начинает работать с интерфейсом формы. Такая схема работы встречается на практике достаточно часто, так что проекты с главной кнопочной формой будут появляться неоднократно.
Продемонстрирую различные аспекты применения скалярных типов и объявления переменных. В проекте роль сервисов, предоставляемых пользователю, будут играть три различных примера - три теста. Каждый тест имеет свою цель и свой интерфейс, позволяющий конечному пользователю проводить исследования возможностей и особенностей объявления скалярных переменных. Для каждого теста естественно будет создан свой интерфейсный класс и спроектирована форма, связанная с этим классом. Каждая командная кнопка главной формы будет запускать свой тест. Обработчик события Click каждой из командных кнопок будет открывать форму, спроектированную для работы с выбранным тестом нашего проекта.

Тест "Types" - ввод и вывод переменных различных типов

Какова идея этого теста? Давайте дадим конечному пользователю возможность в текстовых окнах формы задать имя скалярного типа и значение, соответствующее этому типу. После этого пользователь может нажать командную кнопку "Ввод", спроектированную в интерфейсе формы. Обработчик события Click командной кнопки "Ввод" должен построить переменную заданного типа и присвоить ей значение, заданное пользователем. Если корректно указано имя типа и значение, то операция пройдет успешно, о чем пользователю и будет выдано соответствующее сообщение. В качестве допустимых типов разрешается указывать любой допустимый в C# скалярный тип, заданный таблицей 2.1. В качестве допустимого значения разрешается указывать любое значение из диапазона, соответствующего выбранному типу, и представленному в таблице 2.1. В случае некорректной работы пользователя появляется сообщение "Неправильно указано имя типа" или, если значение не соответствует заданному типу, должно выдаваться сообщение об ошибке.
Спроектируем в интерфейсе пользователя и командную кнопку "Вывод", по нажатию которой будет выдаваться значение, хранимое в созданной переменной.
Тест должен позволить пользователю изучить все скалярные типы языка C#, понять, какие значения допустимы для каждого типа, увидеть, как можно вводить и выводить значения переменных скалярного типа.
Еще одна важная роль этого теста состоит в демонстрации правильной организации ввода данных с использованием программной конструкции try - catch блоков. Ввод данных, задаваемых пользователем, всегда должен контролироваться, поскольку человеку свойственно ошибаться. Тест показывает, как можно обнаруживать ошибки ввода.
В соответствии с задачами теста спроектируем интерфейс пользователя. Прежде всего в проект нужно добавить новый интерфейсный класс, сопровождаемый формой. Такие классы являются наследниками класса Form из библиотеки FCL. Воспользуемся пунктом меню Project|AddWindowsForm для добавления в проект интерфейсного класса и ,соответственно,

новой формы. Отслеживая наши действия, инструментальное средство Designer Form добавит в наш проект необходимые классы. Следуя правилам стиля, произведем переименование, заменив стандартные имена содержательными.
Теперь можно заняться непосредственным проектированием пользовательского интерфейса, размещая в форме нужные элементы управления. Не буду в деталях описывать весь процесс проектирования, поскольку интерфейс достаточно прост. Он отвечает заданной функциональности и включает известные элементы управления - метки, текстовые окна, командные кнопки и контейнеры GroupBox. Контейнеры в формах позволяют придать пользовательскому интерфейсу нужную структуру, объединяя в группу элементы управления, решающие общую задачу - например, ввода исходных данных или вывода результатов.
Проектирование интерфейса пользователя - важная и самостоятельная задача. Она требует определенных дизайнерских качеств, поскольку интерфейс не только должен соответствовать требуемой функциональности, но должен быть интуитивно понятен и элегантен. К сожалению, дизайнер из меня неважный, так что "элегантного" интерфейса в примерах увидеть не удастся
На рис. 2.4 показана спроектированная форма в процессе работы с ней.



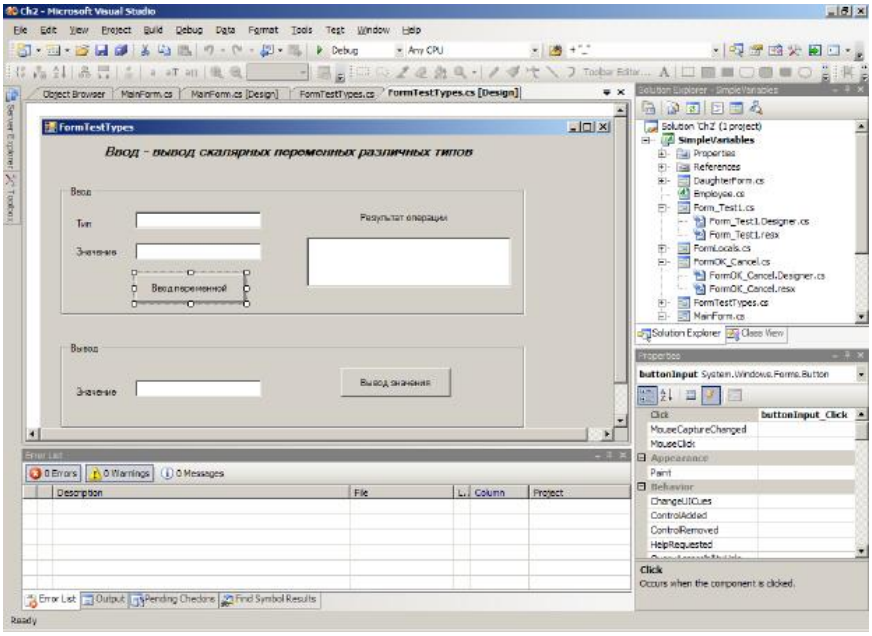
увеличить изображение
Рис. 2.4. Форма TestTypes в процессе работы

Визуальное, событийно-управляемое программирование

Прежде чем продолжить рассмотрение теста, давайте поговорим об основных принципах, лежащих в основе современного визуального стиля программирования. На начальных этапах программирования работой программы полностью управлял ее текст. Так, в программах на языке Алгол выполнение программы начиналось с оператора begin (начало) и заканчивалось оператором end (конец). Развитие программирования потребовало диалога с пользователем в ходе выполнения программы. Вначале это был простой диалог, характерный для уже рассмотренных нами консольных приложений, когда выполнение программы приостанавливается, и она ждет ответа пользователя, вводимого с консоли. Дальнейшее развитие программирования привело к визуальному стилю, когда программные объекты стали иметь визуальные образы, когда появились графические объекты. Графические образы более информативны, чем текстовые. Визуальный стиль изменил и стиль диалога с пользователем, позволив строить интерфейс пользователя, основанный на объектах, имеющих визуальный образ. Простейшим и классическим примером является объект класса TextBox, графическим образом которого является текстовое окно - оно может служить как для вывода текстов в окно, так и для ввода текста пользователем. Этот объект обладает мощной функциональностью текстового редактора, позволяет пользователю редактировать вводимый им текст, удаляя, заменяя и вставляя символы в произвольное место текста.
В библиотеке FCL имеется большое число классов, предназначенных для организации пользовательского интерфейса. Со многими из них будем знакомиться по ходу изучения нашего курса. Эти интерфейсные классы и соответствующие объекты получили название элементов управления (control). Класс Control из этой библиотеки обеспечивает базовую функциональность всех элементов управления и является родительским классом для всех интерфейсных классов. Класс Form, который неоднократно упоминался, также является непрямым наследником класса Control и прямым наследником класса ContainerControl. Последний класс определяет базовую функциональность тех элементов управления, которые, как и формы, есть контейнеры, допускающие внутри графического образа размещение образов других элементов управления. Кроме формы, уже известным нам примером контейнера служит класс GroupBox.
Выше говорилось, что проектирование пользовательского интерфейса, как правило, ведется не программным путем, хотя и такое возможно, а "руками". В Visual Studio 2008 имеется специальная инструментальная панель с элементами управления, которые можно перетаскивать на форму в процессе проектирования, располагать эти элементы в нужном месте формы, менять их размеры, устанавливать их свойства, открываемые в окне свойств. Дизайнер формы транслирует все эти действия в соответствующий программный код.
Процедура Main, задающая точку входа в Windows-проект, открывает главную форму, и пользователь попадает в спроектированный для этой формы мир графических объектов. Теперь пользователь становится у руля управления ходом выполнения проекта и является "возмутителем спокойствия" в этом мире объектов. Он начинает вводить тексты в текстовые окна, выбирать нужные ему элементы из открывающихся списков, нажимать командные кнопки и выполнять другие действия над элементами управления. Каждое такое действие пользователя приводит к возникновению соответствующего события у программного объекта. Так, когда пользователь изменяет текст в текстовом окне, у соответствующего объекта класса TextBox возникает событие TextChanged, при нажатии командной кнопки у объекта возникает событие Click, двойной

щелчок по командной кнопке приводит к событию DblClick. В ответ на возникновение события объект посылает сообщение операционной системе. Обрабатывая очередь сообщений, операционная система отыскивает обработчик события, если объект предусмотрел таковой, и передает ему управление.

Подключить к объекту обработчик события можно визуально в процессе проектирования элемента управления. В окне свойств элемента управления можно перейти к списку событий этого элемента и в этом списке выбрать (включить) нужное событие. На рис. 2.5 показан момент проектирования формы, когда для выбранного элемента управления buttonInput в окне его свойств отображается список возможных событий этого элемента.



увеличить изображение

Рис. 2.5. Список событий элемента управления buttonInput

Можно видеть, что для этого элемента включено событие Click. Заметьте, имя обработчика события строится из имен элемента управления и события, разделенных знаком подчеркивания. У каждого элемента управления есть одно основное событие, которое можно включить простым щелчком по элементу управления, например, для командных кнопок таким событием является событие Click.

При включении события интерфейсный класс автоматически дополняется специальным методом, содержащим заготовку обработчика события - заголовок метода с пустым его телом. Дополнить обработчик события содержательным кодом - задача программиста. В обработчике события программист волен предусмотреть самые разные действия: может изменять свойства других объектов, вызывать методы других объектов, создавать объекты, добавлять или удалять интерфейсные объекты, изменяя мир объектов пользовательского интерфейса. Большинство компьютерных игр - всяческие "стрелялки" - яркий пример такого стиля программирования.

С точки зрения объектного программирования обработчики событий - это специальные методы интерфейсных классов (события), особенность которых состоит в том, что они вызываются в нужный момент операционной системой в ответ на возникновение события у соответствующего объекта, инициированного работой пользователя. Механизм делегатов, на котором основана работа с событиями в языке C#, будет подробно рассматриваться в соответствующем разделе нашего курса.

Стиль программирования, основанный на проектировании визуального пользовательского интерфейса, называется визуальным программированием. Стиль программирования, основанный на событиях и системных вызовах обработчиков событий, называется событийно-управляемым программированием (event - driven programming). Современный стиль является визуальным, событийно-управляемым стилем программирования.

Построение обработчиков событий

Перейдем к решению главной программистской задачи - созданию обработчиков события Click для кнопок "Ввод" и "Вывод". Наиболее интересным является обработчик события для кнопки "Ввод". Пользователь в тестовом окне задает имя скалярного типа. В ответ на его действия обработчик события должен создать объект этого типа. Возможное имя типа - это элемент конечного множества, перечисленного в таблице 2.1. Обработчик должен уметь выполнять работу, называемую в программировании "разбором случаев". Ему нужно понять, какой конкретный тип задал пользователь в текстовом окне, и, следовательно, переменную какого типа ему нужно создать. В C# для разбора случаев есть специальный оператор switch, каждая case-ветвь которого задает один из возможных вариантов. В этой ветви создается объект типа, заданного пользователем, и этому объекту присваивается значение, введенное пользователем в текстовое окно значений.

Поскольку пользователь вводит значение как текст (тип string), возникает необходимость преобразования значения - от типа string к типу, заданному пользователем. При выполнении этого преобразования возможны ошибки по самым разным причинам. Например, пользователь мог задать значение, не принадлежащее множеству возможных значений данного типа. Другая причина - пользователь может не знать, как представляются значения данного типа, использовать точку, а не запятую для данных типа float или double. Возможны просто банальные ошибки - опечатки, неверный символ и так далее. Пользователь - человек, человеку свойственны ошибки, человек имеет право на ошибку. Задача обработчика события - выявить ошибку, если она возникла, сообщить о ней, дать возможность пользователю исправить ошибку и продолжить нормальную работу. В языке C# для этих целей предусмотрен специальный механизм охраняемых блоков, который и будет продемонстрирован в данном примере.

Давайте рассмотрим построенный код той части интерфейсного класса, в которой находятся обработчики событий, использующие упомянутые механизмы. Краткое описание этих механизмов будет дано в этой лекции. Более подробное их описание встретится позже в лекциях, специально посвященных этим механизмам. Начнем с описания полей и конструктора интерфейсного класса:

```
namespace SimpleVariables
{
   public partial class FormTestTypes : Form
```

```
{
    //fields
    string strType = "";
    string strValue = "";
    string strResult = "";

    const string OK_MESSAGE =
          "Операция ввода прошла успешно!";
    const string ERR_MESSAGE =
       "Значение, заданное при вводе, не принадлежит типу ";
    const string ERR_Type_MESSAGE =
       "Неверно задан скалярный тип!";
    public FormTestTypes()
    {
       InitializeComponent();
       textBoxType.Select();
    }
```

Как правило, текстовым полям в интерфейсе класса ставятся в соответствие поля в интерфейсном классе, что облегчает обмен информацией между интерфейсными объектами и объектом, представляющим форму. Константы, являющиеся статическими полями класса, используются при выводе информационных сообщений. В конструктор класса, построенный по умолчанию, добавлен один оператор, позволяющий установить фокус ввода на текстовом окне, в котором пользователь должен задать тип переменной.
Приведем теперь код обработчика события Click командной кнопки buttonInput:

```
private void buttonInput_Click(object sender, EventArgs e)
{
   strType = textBoxType.Text;
   strValue = textBoxValue.Text;

   //разбор вариантов
   switch (strType)
   {
      case "byte":
         {
            byte x;
            try
            {
               x = Convert.ToByte(strValue);
               textBoxResult.Text = OK_MESSAGE;
               strResult = x.ToString();
            }
            catch (Exception)
            {
               textBoxResult.Text = ERR_MESSAGE + "byte!";

            }
            break;
         }
      case "bool":
         {
            bool x;
            try
            {
               x = Convert.ToBoolean(strValue);
               textBoxResult.Text = OK_MESSAGE;
               strResult = x.ToString();
            }
            catch (Exception)
            {
               textBoxResult.Text = ERR_MESSAGE + "bool!";
            }
            break;
         }
      case "decimal":
         {
            decimal x;
```

```
            try
            {
               x = Convert.ToDecimal(strValue);
               textBoxResult.Text = OK_MESSAGE;
              strResult = x.ToString();
            }
               catch (Exception)
               {
                 textBoxResult.Text = ERR_MESSAGE + "decimal!";

               }
                break;
             }
            case "object":
             {
               object x;
               try
                {
               x = strValue;
               textBoxResult.Text = OK_MESSAGE;
               strResult = x.ToString();
            }
            catch (Exception)
            {
               textBoxResult.Text = ERR_MESSAGE + "object!";

            }
            break;
          }
        default :
          {
            textBoxResult.Text = ERR_Type_MESSAGE;
            break;
          }
    }
```

Поскольку все case ветви оператора switch устроены одинаковым образом, в данном тексте большинство ветвей опущено. Если имя типа, заданное пользователем в текстовом окне textBoxType, совпадает с именем, указанным в соответствующей case-ветви, то именно эта ветвь и начинает выполняться. По ее завершении оператором break завершает работу и оператор разбора случаев switch. Если же пользователь задал "ошибочное" имя, то ни одна из case-ветвей не сработает, и тогда управление передается последней default-ветви этого оператора. Она устроена не так, как остальные ветви, - ее задача выдать сообщение о данной ошибке в текстовое окно, представляющее результаты выполнения операции.
Давайте на примере первой case ветви рассмотрим более подробно ее устройство. Еще до выполнения оператора switch обработчик события в поле класса strType и strValue читает информацию, записанную пользователем в соответствующие текстовые поля. Первая case-ветвь сравнивает значение поля strType с возможным вариантом - byte. Если значения совпадают, то пользователь задал тип "byte", поэтому в этой ветви и объявляется переменная этого типа. Рассмотрим три оператора этой ветви:

```
x = Convert.ToByte(strValue);
textBoxResult.Text = OK_MESSAGE;
strResult = x.ToString();
```

Первый из этих операторов присваивает переменной x значение strValue, введенное пользователем в соответствующее текстовое окно. В момент присваивания значение из строкового типа преобразуется к типу byte, заданному пользователем. Это преобразование типа выполняется методом ToByte класса Convert. Следующий оператор выдает сообщение об успехе операции в текстовое окно, информирующее пользователя о результате выполнения операции ввода. Последний оператор тройки формирует значение поля strResult, преобразуя значение типа byte в значение строкового типа.
Два последних оператора этой тройки безопасны, при их выполнении никогда не может произойти ошибки, обусловленной программными причинами (конечно, всегда возможен аппаратный сбой). Но вот первый оператор нормально завершит свою работу только тогда, когда пользователь задаст значение из достаточно узкого диапазона - допустимое значение для типа byte должно быть целым числом от 0 до 255. Во всех остальных случаях преобразование строки к типу byte приведет к ошибке, и возникнет так называемая "исключительная ситуация", когда программа не может продолжать нормально выполняться. Как бы хорошо не была написана программа, избежать возникновения в ней исключительных ситуаций не удается. В данном случае причиной может быть действие пользователя, задавшего некорректное значение. Избежать ситуации нельзя, но можно ее предвидеть и корректно обработать, позволяя продолжить нормальный ход выполнения программы.

В нашем примере это удается сделать за счет того, что оператор, при выполнении которого возможно возникновение исключительной ситуации, помещен в охраняемый try блок. Следом за охраняемым блоком располагается catch блок, которому будет передано управление в случае возникновения исключительной ситуации. Если же try-блок нормально завершит свою работу, то catch-блок выполняться не будет.
Задача catch-блока проста: выдать сообщение об ошибке выполнения операции, указать ее причину и продолжить выполнение проекта, дав пользователю возможность исправить свою ошибку. В данном случае совершенно ясна причина, по которой могла возникнуть исключительная ситуация, - неверно задано значение типа byte. Аппаратные сбои, весьма редкие в наше время, можно игнорировать.
Перейдем теперь к рассмотрению обработчика события Click командной кнопки buttonOutput. Он устроен совсем просто. После того как получила значение переменная x, объявленная в case-ветви, это значение предусмотрительно преобразовалось к строковому типу и сохранялось в поле strResult. Поэтому обработчику события достаточно передать значение этой переменной в текстовое окно. Если при вводе переменной была допущена ошибка, то результатом вывода является пустая строка. Вот код этого обработчика:

```
private void buttonOutput_Click(object sender, EventArgs e)
  {
    if (textBoxResult.Text == OK_MESSAGE)
      textBoxOutputValue.Text = strResult;
   else textBoxOutputValue.Text = "";
  }
```

Исключения и охраняемые блоки.
В этом примере мы впервые встретились с исключениями и охраняемыми try-блоками. Исключениям и способам их обработки посвящена отдельная лекция, но не стоит откладывать надолго знакомство со столь важным механизмом.
Начну с определений. Любая последовательность операторов программы, заключенная в фигурные скобки, образует **блок**. Блок, которому предшествует ключевое слово try, называется **охраняемым** блоком или **try-блоком**. Блок, которому предшествует конструкция catch(<catch параметр>), называется блоком **перехватчиком исключения** или **catch-блоком**.
Блоки играют важную роль в структуре программы. Синтаксически блок воспринимается как один оператор программы - составной оператор. Там, где по синтаксису должен быть один оператор, а содержательно необходима последовательность операторов, эта последовательность заключается в скобки, образуя блок. И синтаксис удовлетворен, и содержательная сторона не страдает. Поскольку каждый оператор внутри блока может быть в свою очередь блоком, то для блоков характерна вложенность.
Еще одна важная роль, которую играют блоки, состоит в том, что они ограничивают область действия объявления локальных переменных. В блоке может быть объявлена переменная, и ее область действия распространяется от точки объявления до конца блока.
Ситуация, при которой выполнение программы прерывается из-за того, что по каким-либо причинам она не может далее нормально выполняться, называется исключительной ситуацией. В языке C# предусмотрен специальный механизм обработки исключительных ситуаций, основанный на исключениях. В момент возникновения исключительной ситуации создается специальный объект, называемый исключением, он характеризует возникшую ситуацию.
В состав библиотеки FCL входит класс **Exception**, который задает базовые свойства и методы исключений, рассматриваемых как объекты. У класса Exception большое число потомков, каждый из которых описывает определенный тип исключения. При проектировании собственного класса зачастую следует создать и собственный класс исключений, описывающий исключения, которые могут возникать при работе с объектами собственного класса. Все классы исключений, в том числе и создаваемые программистом, должны быть потомками базового класса Exception.
Как показывает практика программирования, любая программа не гарантирует, что в процессе ее работы не возникнут какие-либо неполадки, в результате которых она не сможет выполнить свою задачу. 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.
Стандартный обработчик исключений предусмотрен операционной системой. Он завершает выполнение программы, выдавая соответствующую информацию о возникновении исключения. Стандартное описание возникшего исключения может быть непонятно конечному пользователю, не говоря уже о том, что завершение программы до получения нужного результата весьма нежелательно. Хорошо построенная программа сама должна обрабатывать возникшие ошибки.
Если в некотором модуле предполагается возможность появления исключений, то разумно предусмотреть и их обработку. В этом случае в модуле создается охраняемый try-блок. Вслед за этим блоком следуют один или несколько блоков, перехватывающих 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. По сути, последовательность catch-блоков соответствует схеме разбора случаев, применяемой в операторе switch.
Конечно, плохо, когда в процессе работы программы возникает исключение. Однако его появление еще не означает, что программа 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, не приводящие к исключениям. Тогда работа продолжается с неверными данными без исправления ситуации и даже без уведомления о возникновении ошибки.

Наш пример продемонстрировал первое применение охраняемых блоков и обработку возникающих исключений. В примере для всех охраняемых блоков использовался универсальный перехватчик исключений, поскольку причина возникновения исключения в охраняемом блоке однозначно определялась, так что в разборе случаев не было необходимости.

Контролируемый ввод данных

Пользователь, работающий с программой, имеет право на ошибку. Программист, создающий программу, такого права не имеет. Если из-за ошибки пользователя программа перестает работать, то в этом вина программиста, поскольку его программа не смогла вовремя обнаружить ошибку и дать возможность пользователю исправить ее. Ошибки пользователя чаще всего возникают, когда пользователь вводит исходные данные, необходимые программе. Поэтому любой пользовательский ввод должен контролироваться. Операторы, управляющие вводом, должны, как правило, размещаться в охраняемых блоках. Обнаружение ошибки ввода должно приводить к возникновению исключения, а обработчик этого исключения должен проинформировать об ошибке и попытаться исправить ситуацию.

В рассмотренном выше примере пользовательский ввод контролируется. Поэтому стоит еще раз проанализировать этот пример с позиций того, как организован контроль ввода данных, вводимых пользователем.

Тест 1. Объявления с инициализацией и без нее

Этот тест демонстрирует объявление переменных, часть из которых инициализируется в момент объявления. Пользователю в текстовом окне показывается некоторый фрагмент программы, содержащий объявления переменных и простейшие присваивания. Задача пользователя - в текстовые окна, спроектированные для всех переменных фрагмента, ввести их значения, которые они получат в результате выполнения фрагмента. После чего он может нажать командную кнопку и сравнить введенные им результаты с теми, что дает реальное вычисление.

На рис. 2.6 показана спроектированная форма в процессе работы.

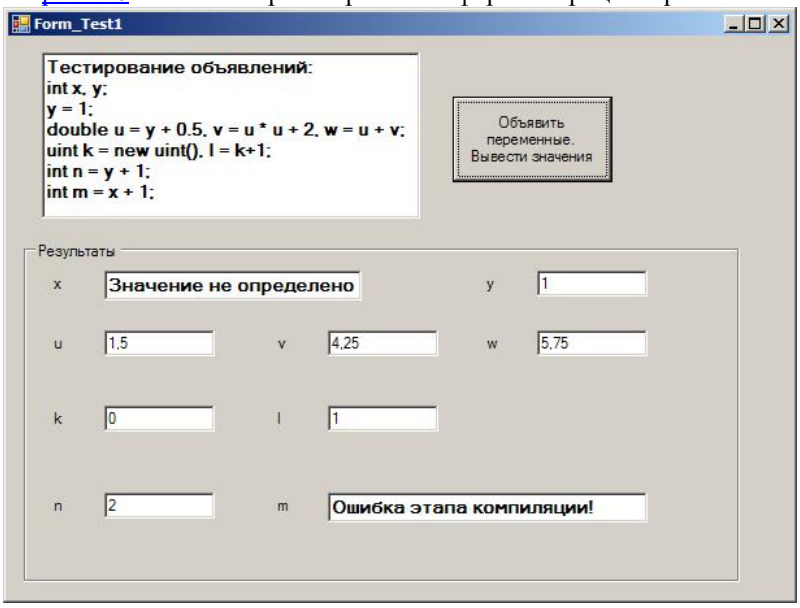


Рис. 2.6. Форма теста 1 в процессе работы

Давайте поговорим подробнее о демонстрируемом фрагменте кода:

```
 int x, y;
y = 1;
double u = y + 0.5, v = u*u +2, w = u+v;
uint k = new uint(), l = k + 1;
int n = y + 1;
//int m = x + 1;
```

В первой строке объявляются две сущности с именами x и y типа int без инициализации. В результате этого объявления в стеке будут созданы две переменные (два объекта), им будет выделена нужного размера память под значения, но сами значения останутся неопределенными. Во второй строке кода в результате присваивания переменная y получит значение. В третьей строке кода объявляются с инициализацией три сущности с именами u, v, w типа double. В стеке будут созданы три переменные, им будет отведена память под значения. Поскольку на этапе компиляции инициализирующие выражения могут быть вычислены, то значения этих переменных будут определены.

В следующей строке кода объявляются с инициализацией две сущности с именами k и l типа uint. Для сущности k используется объектный стиль инициализации с вызовом конструктора объектов класса uint. Конструктор без параметров этого класса не только создает в стеке соответствующий объект, но и инициализирует его значением по умолчанию, равным нулю для этого типа. Переменная l в результате инициализации также получит значение.

Аналогичным образом в результате объявления в стеке будет создана и переменная с именем n, которая также получит значение. А вот попытка создать и инициализировать переменную m закончится ошибкой еще на этапе компиляции, поскольку выражение инициализатора ссылается на переменную x, значение которой все еще не определено. Поэтому соответствующий оператор кода закомментирован.

Вот еще один тест на эту же тему:

```
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
 //x = s;
}// SimpleVars
```

В первой строке объявляются переменные x, s с отложенной инициализацией. Последующие объявления переменных эквивалентны по сути, но демонстрируют два стиля инициализации - обычный и объектный. Обычная форма инициализации предпочтительнее не только в силу своей естественности, она еще 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 базисных 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 присваивании x = s 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!

Переменные. Область видимости и время жизни

Давайте рассмотрим, где могут появляться объявления переменных, какую роль они играют в зависимости от уровня, на котором они объявлены. 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 класса

Первая важнейшая роль переменных - задавать свойства классов. В языке C#, как и в других ОО-языках, такие переменные называются **полями (fields)** класса. О классах и полях предстоит еще обстоятельный разговор, а сейчас сообщу лишь некоторые минимальные сведения, связанные с рассматриваемой темой.

Все переменные, объявленные на уровне класса при его описании, являются полями этого класса.

Поскольку класс, как уже многократно говорилось, задает описание типа данных, то поля класса задают представление этих данных. Необходимо крайне внимательно относиться к проектированию полей класса - всякие "лишние" объявления на этом уровне крайне нежелательны.

Когда конструктор класса создает очередной объект - экземпляр класса, то он в 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. Заметьте, для классов, представленных структурой, объект создается аналогичным способом, но разворачивается в стеке.

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 модуля, связанного с классом, и не входит в состав его экземпляров. Поэтому static-поля живут так же долго, как и сам класс. Более подробно эти вопросы будут обсуждаться при рассмотрении классов, структур, интерфейсов.

Наследование и поля

Наследование классов - это одно из важнейших отношений, существующих между классами одного проекта. О нем мы будем говорить подробно, а сейчас рассмотрим только одну сторону наследования - что происходит с полями классов в процессе наследования. Пусть класс B является наследником класса A. Тогда класс B наследует все поля класса A. Наследник не может ни удалить поле родительского класса, ни изменить его тип. Наследник может лишь добавить собственные поля к уже имеющимся полям родителя. Таким образом, объекты класса наследника обладают всеми свойствами родителя и возможно дополнительным набором свойств.

Как создаются объекты класса наследника? Конструктор этого класса первым делом вызывает конструктор родителя, и тот создает объект родителя - коробочку с набором родительских полей. Только после этого конструктор наследника добавляет, если они есть, собственные поля к уже созданному объекту.

Все сказанное относится лишь к "настоящим" классам, представляющим ссылочный тип. Для развернутых классов, заданных структурой, отношение наследования не определено. Структуры могут иметь в качестве родительских классов лишь интерфейсы.

Область видимости полей класса

Поля класса являются глобальными переменными класса. Они видимы во всех методах этого класса. Каждый метод класса может читать и изменять значение любого поля класса независимо от того, какие атрибуты доступа установлены для полей и методов класса.
Если в теле метода объявлена локальная переменная, имя которой совпадает с именем поля класса, то такая ситуация не приводит к ошибке, поскольку конфликт имен разрешим. К полю класса можно добраться, используя уточненное имя поля с префиксом this, задающим имя текущего объекта.
Поля класса видимы не только в пределах самого класса. Если в некотором классе B объявлен и создан объект класса A, то класс B является клиентом класса A. В классе клиенте у объекта видны лишь те поля класса, для которых в момент объявлении был задан атрибут доступа public, что делает эти поля общедоступными. Для классов наследников у объекта видны поля с атрибутами public или protected, но недоступны поля с атрибутом private.
Имеет место следующая иерархия доступности полей объекта в зависимости от значения атрибута доступа - public, protected, private. В самом классе доступны все поля. У наследников не доступны закрытые поля с атрибутом private, у клиентов не доступны закрытые поля и защищенные поля - поля с атрибутами private и protected.

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, хранимые в полях класса, являются тем центром, вокруг которого вращается мир класса. Методы класса в этом мире, можно сказать, играют второстепенную роль - они обрабатывают данные. Заметьте, каждый экземпляр класса - это отдельный мир. Поля экземпляра (открытые, защищенные и закрытые) - это глобальная информация, которая доступна всем методам класса.
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 на уровнях более высоких, чем класс, существенно повышает надежность создаваемых на языке C# программных продуктов, поскольку устраняется источник опасных, трудно находимых ошибок.

Локальные переменные

Перейдем теперь к рассмотрению локальных переменных. Во всех языках программирования, в том числе и в C#, основной контекст, в котором появляются переменные, - это процедуры и функции - методы класса. Тело метода, заключенное в фигурные скобки, будем называть **процедурным блоком**. Переменные, объявленные в процедурном блоке, называются **локальными** - они локализованы в методе.
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# 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, объявленные в любом внутреннем блоке, существуют от точки объявления до конца соответствующего блока.
Рассмотрим ситуацию с возможными конфликтами имен, появляющихся в различных блоках. Уже говорилось, что имя локальной переменной может совпадать с именем поля класса. Этот конфликт разрешен, поскольку для поля класса можно использовать уточненное имя. Чтобы избежать других конфликтов, не разрешается во внутреннем блоке метода объявлять локальную переменную, имя которой совпадает с именем формального параметра метода или с именем локальной переменной, объявленной в охватывающем блоке.

Класс TestingLocals

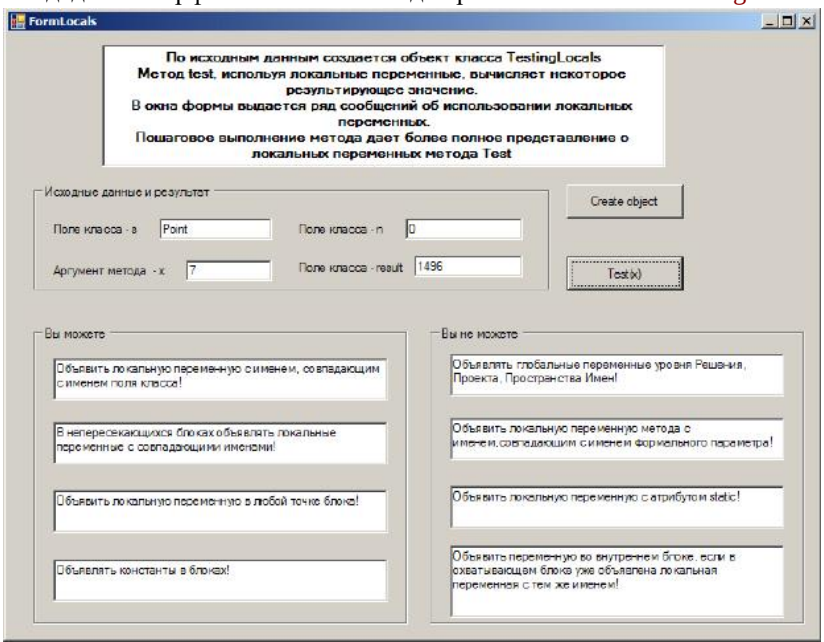
Добавим в наш проект новый класс TestingLocals. Зададим в этом классе два поля и один метод. Поля будут играть роль глобальных переменных для метода класса, а во внутренних блоках метода появятся объявления локальных переменных. Это поможет нам обсудить на примере области действия и существования объявленных переменных. Вот код этого класса:

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

```
namespace SimpleVariables
{
   class TestingLocals
   {
      //fields
      string s;
      int n;

      const string POINT = "Point_1";

      //Constructor
      public TestingLocals(string s, int n)
      {
         this.s = s;
         this.n = n;
      }

      //Method
      public int Test(int x)
      {
         int result = 0;
         int n = 9;
         if (s == POINT)
         {
            //static int sc =7;
            const int cc = 6;
            int u = 7, v = u + 2;
            x += u + v - cc;
            for (int i = 0; i < x; i++)
            {
               result += i * i;
            }
            //x += i;
         }
         else
         {
            //int n = 5;
            //int x = 10;
            int u = 7, v = u + 2;
            x += u + v - n + this.n;
            for (int i = 0; i < x; i++)
            {
               result += i * i;
            }
         }
         Return result;
    }}  }
```

Тест 2. Локальные и глобальные переменные класса

Создадим интерфейс пользователя для работы с классом TestingLocals. С этой целью добавим в проект интерфейсный класс FormLocals - наследник класса Form. На рис. 2.7 показано, как выглядит спроектированный интерфейс в процессе работы с формой.



увеличить изображение

Рис. 2.7. Форма FormLocals - интерфейс пользователя

Назначение формы поясняется в специальном текстовом окне. В разделе "исходные данные" два текстовых окна позволяют задать данные, необходимые для формирования объекта класса TestingLocals. Две командные кнопки позволяют создать объект этого класса и вызвать метод Test, тело которого представляет систему вложенных внутренних блоков, содержащих объявление

локальных переменных. Большой раздел в интерфейсе формы занимают советы, подсказывающие разработчику, что он может делать при объявлении локальных переменных и что является недопустимым.
Напомню, что код интерфейсного класса, создаваемый по умолчанию, состоит из двух частей. Одна часть предназначена для Дизайнера форм, и код в ней появляется автоматически, отражая проектирование дизайна формы, выполняемое руками. Другая часть ориентирована на разработчика интерфейса. В эту часть класса добавляются поля, необходимые для обмена информацией с элементами управления, расположенными на форме. Здесь же находится поле, в котором объявлен объект класса TestingLocals. Заметьте, если создается интерфейсный класс, обеспечивающий поддержку работы с одним или несколькими содержательными классами, то в интерфейсном классе должны быть поля с объектами этих классов. Так интерфейсный класс становится клиентом содержательного класса. В нашем случае интерфейсный класс FormLocals становится клиентом класса TestingLocals и эти два класса связываются отношением "клиент - поставщик".
Приведу код той части интерфейсного класса, которая создается разработчиком:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleVariables
{
   public partial class FormLocals : Form
   {
      //поля класса - константы и переменные
      const string YOU_CAN_1 =
         "Объявить локальную переменную с именем, " +
         "совпадающим с именем поля класса!";
      const string YOU_CAN_2 =
         "В непересекающихся блоках объявлять " +
         "локальные переменные с совпадающими именами!";
      const string YOU_CAN_3 =
         "Объявить локальную переменную в любой точке блока!";

      const string YOU_CAN_4 =
         "Объявлять константы в блоках!";
      const string YOU_CANNOT_1 =
         "Объявлять глобальные переменные " +
         "уровня Решения, Проекта, Пространства Имен!";
      const string YOU_CANNOT_2 =
         "Объявить локальную переменную метода с именем," +
         "совпадающим с именем формального параметра!";
      const string YOU_CANNOT_3 =
         "Объявить локальную переменную с атрибутом static!";
      const string YOU_CANNOT_4 =
         "Объявить переменную во внутреннем блоке, " +
         "если в охватывающем блоке уже объявлена " +
         "локальная переменная с тем же именем!";

      TestingLocals testing;
      string s;
      int n;
      public FormLocals()
      {
         InitializeComponent();
   }}  }
```

Помимо упомянутых полей класса, в нем определены строковые константы для вывода советов по использованию локальных переменных.
В этой части класса появляются и обработчики событий элементов управления формы. Вот как выглядит код обработчика события Click командной кнопки, создающей объект класса TestingLocals:

```
private void buttonCreateObject_Click(object sender, EventArgs e)
      {
         s = textBoxS.Text;
         n = Convert.ToInt32(textBoxN.Text);
         testing = new TestingLocals(s, n);
         textBoxS.Text = "Point";
```

```
        textBoxN.Text = "0";      }
```
Устроен он достаточно просто: вначале из текстовых полей формы читается информация, необходимая конструктору класса TestingLocals для создания объекта, затем этот объект создается, а текстовые окна формы получают новое значение, которое может быть изменено конечным пользователем в процессе работы с формой.
Чуть более сложно устроен обработчик события Click командной кнопки, вызывающей метод Test класса TestingLocals:
```
private void buttonTest_Click(object sender, EventArgs e)
 {
   int x = Convert.ToInt32(textBoxX.Text);
   textBoxResult.Text = testing.Test(x).ToString();

   textBoxCan1.Text = YOU_CAN_1;
   textBoxCan2.Text = YOU_CAN_2;
   textBoxCan3.Text = YOU_CAN_3;
   textBoxCan4.Text = YOU_CAN_4;

   textBoxCannot1.Text = YOU_CANNOT_1;
   textBoxCannot2.Text = YOU_CANNOT_2;
   textBoxCannot3.Text = YOU_CANNOT_3;
   textBoxCannot4.Text = YOU_CANNOT_4;
 }
```

И здесь из текстового окна формы читается значение аргумента, заданное конечным пользователем, затем созданный объект testing вызывает открытый (public) метод класса Test, и результат работы метода выводится в соответствующее текстовое окно, поддерживая необходимую связь с конечным пользователем. В качестве побочного эффекта в текстовые поля формы выводятся советы по использованию локальных переменных.
Помимо советов, анализируя текст метода Test, следует обратить внимание при использовании локальных переменных на следующие моменты.
Аргументы метода (его формальные параметры) считаются объявленными в начале блока, задающего тело метода. Таким образом, область их действия распространяется на весь этот блок и ни в одном из внутренних блоков нельзя объявлять локальную переменную с именем, совпадающим с именем аргумента.
Параметр цикла считается объявленным в блоке, задающем тело цикла. Поэтому область его действия распространяется на весь этот блок и во внутренних блоках тела цикла нельзя объявлять локальную переменную с именем, совпадающим с именем параметра цикла. Заметьте, после окончания цикла параметр цикла перестает существовать и не может быть использован. В нашем примере оператор, в котором делается попытка использовать параметр цикла после завершения цикла, закомментирован.
В параллельных блоках (в нашем примере две ветви оператора if) разрешается объявлять локальные переменные с одинаковыми именами, поскольку области существования этих переменных не пересекаются.
Поскольку объявлять локальную переменную можно в любой точке блока, хорошим стилем считается объявление локальной переменной как можно ближе к точке ее непосредственного использования. Нет смысла объявлять локальную переменную в начале блока, если она будет использована где-то в конце блока.
Глобальные переменные уровня процедуры. Существуют ли?
Поскольку процедурный блок - блок тела метода - 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. В большинстве известных языков 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 имени, совпадающего с именем, уже использованном во внешнем блоке. В нашем примере незаконная попытка объявить во внутреннем блоке уже объявленное имя 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. 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 SMALL_SIZE = 38, LARGE_SIZE =58;
    const int MIDDLE_SIZE = (SMALL_SIZE + LARGE_SIZE)/2;
    const double PI = 3.141593;
    // LARGE_SIZE = 60; //Значение константы нельзя изменить.
   }//Constants
```

Два важных правила стиля связаны с константами. Правило именования констант требует, чтобы имена констант задавались заглавными буквами и знак подчеркивания использовался бы в качестве разделителя слов для многословных имен. Правило стиля "Нет литеральным константам" требует, чтобы литеральные константы использовались только в момент объявления именованных констант, а во всех остальных местах кода применялись бы только именованные константы. Это позволяет давать константам содержательные имена, что улучшает понимание смысла программы. Кроме того, если по какой-либо причине значение константы нужно изменить, такое изменение будет сделано только в одном месте - в точке объявления константы, не затрагивая основного кода программы. Следуя этим правилам стиля, легко обеспечить многоязычный интерфейс программы.

Типы и классы

Язык C# является языком объектного программирования. Все типы - встроенные и пользовательские - определены как классы, связанные **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 классы, создаваемые пользователем, также являются потомками класса object, в них, как правило, необходимо переопределить методы родителя, поскольку реализация родителя, предоставляемая по умолчанию, не будет обеспечивать нужный эффект.

Рассмотрим вполне корректный в языке C# пример объявления переменных и присваивания им значений:

```
int x = 1;
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 поле этого int, спросите Вы? Ведь ранее говорилось, что int относится к значимым - 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. 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 ссылочные 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 принадлежащими к значимым типам.

Проекты, содержащие несколько форм

Вернемся к вопросам организации нашего проекта. Проект SimpleVariables, созданный нами, содержит четыре интерфейсных класса, наследуемых от класса Form, и ,соответственно, четыре формы. Каждая форма представляет контейнер, который в процессе проектирования заполняется элементами управления для создания необходимого пользовательского интерфейса.
Архитектура проекта является примером типовой архитектуры проекта с главной кнопочной формой. Главная форма, задающая точку входа в проект, представляет форму с множеством командных кнопок. Всякий раз, когда в проект добавляется новый интерфейсный класс и соответствующая ему форма, в класс главной формы добавляется объект нового класса, а на главную форму добавляется новая командная кнопка, обработчик события Click которой будет создавать объект этого класса и вызывать метод Show для показа формы соответствующего интерфейсного класса.
Обработчик события Click, создающий объект интерфейсного класса FormLocals и вызывающий связанную с ним форму, имеет вид:

```
private void button1_Click(object sender, EventArgs e)
  {
    testLocalsForm = new FormLocals();
    testLocalsForm.Show();
  }
```

Модальные и немодальные формы. Методы Show, ShowDialog, Hide,Close

В проекте с множеством форм одновременно на экране может быть открыто несколько форм. Может ли пользователь переключаться на работу с той или иной формой? Все зависит от того, как открыта форма. Каждую форму можно открыть как модальную или как немодальную. Хотя термин "модальная форма" широко используется, удобнее говорить, что форма может быть открыта как "диалоговое окно" (модальная форма) или как обычное окно (немодальная форма). Если форма, как в наших примерах, открывается методом Show, то она задает обычное окно; если форму открывать методом ShowDialog, то она открывается как диалоговое окно. В чем разница? Из диалогового окна нельзя выйти, не закончив диалог и не закрыв форму. Открыв диалоговое окно, нельзя переключиться на работу с другой формой, не закончив диалог. Закрыть диалоговое окно можно разными способами. Можно щелкнуть по крестику, расположенному в правом верхнем углу формы, закрывая ее "грубым" способом. В диалоговом окне часто размещают командные кнопки с предопределенной семантикой - "OK", "Cancel" и другие. Щелчки по этим кнопкам также приводят к закрытию диалогового окна. У метода ShowDialog есть еще одна особенность: в отличие от метода Show он реализован как функция, возвращающая значение типа System.DialogResult. Благодаря этому можно узнать, какая кнопка была нажата при завершении диалога (например, OK или Cancel).
Если форма открыта методом Show, как недиалоговое окно, то, не закончив работу с открытой формой, можно перейти в главную форму или другую немодальную форму, поработать там, нажав какие-нибудь командные кнопки, получив нужную информацию, а затем снова вернуться к исходной форме.
Для показа формы можно применять два метода - Show и ShowDialog, для скрытия формы можно также применять два метода - Hide и Close. Первый из этих методов скрывает форму, второй - закрывает. Для диалоговых окон можно применять как метод Hide, так и метод Close: эффект будет одинаков - диалоговое окно будет закрыто. Затем его можно открыть и показать методом Show. Метод Hide можно применять и для немодальных форм, открытых методом Show. Окно, открытое не для диалога, можно временно скрыть, вызвав метод Hide, а затем показать, вызвав метод Show. Но вот на что следует обратить особое внимание. После закрытия недиалогового окна - либо при вызове метода Close, либо "грубым" способом нажатия на крестик в окне формы - показать затем форму, вызвав метод Show, уже не удастся. Причина в том, что при закрытии формы сам объект, задающий форму, продолжает существовать, но ресурсы освобождаются и графическое окно уже не связано с программным объектом. Поскольку конечный пользователь всегда может применить грубую силу для закрытия формы, обратите внимание, что в соответствующих обработчиках события создание объекта, задающего форму, предшествует вызову метода Show. Для диалоговых окон объект можно было бы создать только один раз, например, в конструкторе главной кнопочной формы.
Я добавил в проект диалоговую форму, в которой разместил командные кнопки OK и Cancel с предустановленной семантикой, ввел еще некоторые изменения, чтобы проиграть все возможные варианты показа, скрытия и закрытия форм, открываемых как модальные (диалоговые окна), так и немодальные. Не буду приводить подробного описания этих экспериментов, но надеюсь, что при желании читатели их смогут провести самостоятельно.
Еще одно замечание, связанное с закрытием форм. Когда закрывается главная форма, открываемая в точке входа - процедуре Main, закрываются все открытые к этому времени формы и приложение заканчивает свою работу. Когда же закрывается любая другая форма, закрывается только эта форма, остальные формы остаются открытыми.

Задачи

- 1. Построить консольное приложение. Дать ему имя, разместить в выбранной директории. Проанализировать созданный программный текст. Познакомиться со средой разработки Visual Studio .Net. Выполнить приложение в пошаговом отладочном режиме.
- 2. Построить Windows-приложение. Дать ему имя, разместить в выбранной директории. Проанализировать созданный программный текст. Познакомиться со средой разработки Visual Studio .Net. Выполнить приложение в пошаговом отладочном режиме.
- 3. Построить консольное приложение "Здравствуй, Мир!", выводящее на консоль строку приветствия.

Встроенные типы данных. Ввод-вывод данных

Назначение задач этого раздела состоит в знакомстве с основными скалярными встроенными типами, их классификацией, диапазоном возможных значений. Решение задач требует умения объявлять, вводить и выводить значения переменных этих типов.

- 10. Построить циклическое консольное приложение "Целочисленные типы". Приложение поочередно вводит с консоли значения целочисленных типов: sbyte, byte, short, ushort, int, uint, long, ulong. Вводу значения предшествует приглашение к вводу. После завершения ввода приложения выводит все введенные значения с указанием их типа. Проанализировать, что происходит при вводе значений, не соответствующих требуемому типу или выходящих за пределы интервала возможных значений типа.
- 11. Построить Windows-приложение "Целочисленные типы" с 16-ю помеченными текстовыми окнами и двумя командными кнопками Пользователь вводит значения целочисленных типов: sbyte, byte, short, ushort, int, uint, long, ulong в первые 8 окон. По нажатию командной кнопки "Ввод значений" данные из текстовых окно становятся значениями переменных соответствующих типов. По нажатию командной кнопки "Вывод значений" значения переменных соответствующих типов передаются в текстовые окна, предназначенные для вывода значений. Проанализировать, что происходит при вводе значений, не соответствующих требуемому типу или выходящих за пределы интервала возможных значений типа.

Контролируемый ввод данных

В задачах этого раздела необходимо организовать контроль данных, вводимых пользователем.

- 20. Постройте консольное приложение, в котором вводится имя пользователя. Имя должно отвечать правилам, принятым в русском языке: составлено из букв кириллицы. Первый символ должен быть заглавной буквой, остальные - строчными буквами.
- 21. Постройте Windows-приложение, в котором вводится имя пользователя. Имя должно отвечать правилам, принятым в русском языке: составлено из букв кириллицы. Первый символ должен быть заглавной буквой, остальные - строчными буквами.

Выражения

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

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 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.

Таблица 3.1. Приоритеты операций языка C#

Приоритет	Категория	Операции	Порядок
0	Первичные	(expr), x.y, x->y, f(x), a[x], x++, x--, new, typeof(t), checked(expr), unchecked(expr)	Слева направо
1	Унарные	+, -, !, ~, ++x, --x, (T)x, sizeof(t)	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 И (AND)	&	Слева направо
8	Логическое исключающее ИЛИ (XOR)	^	Слева направо
9	Логическое ИЛИ (OR)	|	Слева направо

10	Условное логическое И	&&	Слева направо
11	Условное логическое ИЛИ	||	Слева направо
12	Условное выражение	? :	Справа налево
13	Присваивание Склеивание с null	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |= ??	Справа налево
14	Лямбда-оператор	=>	Справа налево

Перегрузка операций и методов

Под **перегрузкой операции** понимается существование нескольких реализаций одной и той же операции. Например, операция со знаком "+" выполняется по-разному в зависимости от того, являются ли ее операнды целыми числами, длинными целыми, целыми с фиксированной или плавающей точкой или строками текста.

Нужно понимать, что операции - это частный случай записи методов класса. Методы класса, так же как и операции, могут быть перегружены. Метод класса называется **перегруженным**, если существует несколько реализаций этого метода. Перегруженные методы имеют одно и то же имя, но должны отличаться своей **сигнатурой**. Сигнатуру метода составляет список типов формальных аргументов метода. Так что два метода класса с одним именем, но отличающиеся, например, числом параметров, имеют разную сигнатуру и удовлетворяют требованиям, предъявляемым к перегруженным методам.

Большинство операций языка C# перегружены - одна и та же операция может применяться к операндам различных типов. Поэтому прежде чем выполнять операцию, проводится 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 одного из операндов. 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. Если для данных типов операндов нет подходящей реализации операции и невозможно неявное приведение типов операндов, то, как правило, эта ошибка обнаруживается еще на этапе компиляции.

Преобразования типов

Каждый объект (переменная), каждый операнд при вычислении выражения, само выражение характеризуется парой $< v, T >$, задающей значение выражения и его тип. В процессе вычислений зачастую возникает необходимость преобразования типов - необходимость преобразовать пару $< v_s, T_s >$ к паре $< v_g, T_g >$. Исходная пара называется источником преобразования, заключительная - целью преобразования.

Необходимость в подобных преобразованиях возникает, как уже отмечалось, по ходу вычисления выражения при приведении операндов к типу, согласованному с типом операции. Преобразование типов необходимо в операторах присваивания, когда тип выражения правой части оператора приводится к типу, заданному левой частью этого оператора. Семантика присваивания имеет место и при вызове методов в процессе замены формальных аргументов метода фактическими параметрами. И здесь необходимо преобразование типов.

Преобразования типов можно разделить на безопасные и опасные. Безопасное преобразование - это преобразование, для которого гарантируется, что:

- оно выполнимо для всех возможных значений источника v_s,
- в процессе преобразования не происходит потери точности, то есть точность задания v_g соответствует точности задания v_s.

Преобразование, для которого не выполняется хотя бы одно из этих условий, называется опасным. Достаточным условием существования безопасного преобразования является, например, условие того, что тип T_s является подтипом типа T_g. Действительно, в этом случае любое значение источника является одновременно и допустимым значением цели. Так, преобразование от типа int к типу double является безопасным. Обратное преобразование, естественно, будет опасным.

Некоторые преобразования типов выполняются автоматически. Такие преобразования называются неявными, и они часто встречаются при вычислении выражений. Очевидно, что неявными могут быть только безопасные преобразования. Любое опасное преобразования должно явно задаваться самим программистом, который и берет на себя всю ответственность за выполнение опасного преобразования.

Существуют разные способы выполнения явных преобразований - операция кастинга (приведение к типу), методы специального класса Convert, специальные методы ToString, Parse. Все эти способы будут рассмотрены в данной лекции.

Поясним, как выполняются неявные преобразования при вычислении выражения. Пусть при вычислении некоторого выражения необходимо выполнить сложение $x + n$, где x имеет тип double, а n - int. Среди многочисленных реализаций сложения есть операции, выполняющие сложение операндов типа int и сложение операндов типа double, так что при выборе любой из этих реализаций сложения потребуется преобразование типа одного из операндов. Поскольку преобразование типа от int к double является безопасным, а в другую сторону это преобразование опасно, то выбирается безопасное преобразование, выполняемое автоматически, второй операнд неявно преобразуется к типу double, выполняется сложение операндов этого типа, и результат сложения будет иметь тип double.

Организация программного проекта ConsoleExpressions

Как обычно, все примеры программного кода, появляющиеся в тексте, являются частью программного проекта. Опишем структуру используемого консольного проекта, названного ConsoleExpressions. Помимо созданного по умолчанию класса

Program, в проект добавлены два класса с именами TestingExpressions и Scales. Каждый из методов класса TestingExpressions представляет тест, который позволяет анализировать особенности операций, используемых при построении выражений, так что этот класс представляет собой сборник тестов. Класс Scale носит содержательный характер, демонстрируя работу со шкалами, о которых пойдет речь в этой лекции. Чтобы иметь возможность вызывать методы этих классов, в процедуре Main класса Program объявляются и создаются объекты этих классов. Затем эти объекты используются в качестве цели вызова соответствующих методов. Общая схема процедуры Main и вызова методов класса такова:

```
static void Main(string[] args)
 {
    string answer = "Да";
    do
    {
       try
       {
                                TestingExpressions test = new TestingExpressions();
          test.Casting();
          //Вызов других методов
          …
       }
       catch (Exception e)
       {
          Console.WriteLine(
             "Невозможно нормально продолжить работу!");
          Console.WriteLine(e.Message);
       }
     Console.WriteLine("Продолжим работу? (Да/нет)");
       answer = Console.ReadLine();
    } while (answer == "Да" || answer == "да" || answer == "yes");
 }
```

Всякий раз, когда в тексте лекции нужно будет привести пример кода, будет приводиться либо полный текст вызываемого метода, например, метода Casting, либо отдельный фрагмент метода.

Операции высшего приоритета

Рассмотрим подробнее операции из таблицы 3.1, отнесенные к высшему приоритету и выполняемые в первую очередь.

Выражения в скобках

Любое выражение, взятое в скобки, получает высший приоритет и должно быть вычислено, прежде чем к нему будут применимы какие-либо операции. Скобки позволяют изменить стандартный порядок вычисления выражения и установить порядок, необходимый для вычисления в данном конкретном случае. В сложных выражениях скобки полезно расставлять даже в том случае, если стандартный порядок совпадает с требуемым, поскольку наличие "лишних" скобок зачастую увеличивает наглядность записи выражения.

Вот классический пример выражения со скобками:

```
result = (x1 + x2) * (x1 - x2);
```

Понятно, что если убрать скобки, то первой выполняемой операцией будет операция умножения и результат вычислений будет совсем другим.

Поскольку согласно стандартному порядку выполняются вначале арифметические операции, потом операции отношения, а затем логические операции, то можно было бы не ставить скобки в следующем выражении:

```
 bool temp = x1 + x2 > x1 - x2 && x1 - 2 < x2 + 1;
          result = temp? 1 : 2;
```

Однако "лишние" скобки в записи выражения явно не помешают:

```
bool temp = ((x1 + x2) > (x1 - x2)) &&
             ((x1 - 2) < (x2 + 1));
          result = temp? 1 : 2;
```

Операция вызова "точка" x.y, вызов функций F(x), вызов, инициируемый указателем x -> y

Несмотря на то, что точка - "малозаметный" символ, операция вызова x.y является одной из основных и важнейших операций в объектном программировании. Здесь x является целью вызова и представляет некоторый уже созданный объект, а y является свойством или методом этого объекта. Поскольку свойство объекта может задавать новый объект, может порождаться достаточно длинная цепочка вызовов (x.y1.y2.y3.y4), заканчивающаяся, как правило, терминальным свойством. Если объект вызывает не свойство, а метод, то вызов метода сопровождается заданием фактических аргументов:

```
x.M(a1, … ak)
```

Когда такой вызов встречается в выражениях, метод должен возвращать значение, отличное от void (быть функцией), чтобы такое выражение могло быть использовано в качестве операнда какой-либо операции. Вызов метода, возвращающего значение void, используется как отдельный оператор, что неоднократно встречалось в наших примерах.

В качестве цели вызова может применяться не только имя объекта, но и имя класса. В этом случае вызывается статическое свойство или статический метод этого класса. Для каждого класса, у которого есть статические поля и статические методы, автоматически создается специальный объект (модуль), содержащий статические поля, к которым относятся и константы класса. Имя этого объекта совпадает с именем класса. Вот несколько примеров подобных вызовов:

```
Console.WriteLine(INPUT_FLOAT);
strInput = Console.ReadLine();
x1 = Convert.ToSingle(strInput);
```

Здесь в качестве цели вызовов выступают классы Console и Convert, вызывающие статические методы этих классов.
Если цель вызова указана, то такой вызов называется квалифицированным. Когда целью вызова является текущий объект, ее (цель) можно опускать, делая вызов неквалифицированным. Такой вызов всегда можно сделать квалифицированным, указав this в качестве имени текущего объекта:

```
result += this.n * this.m;
```

В данном случае можно было бы опустить имя текущего объекта и записать выражение следующим образом:

```
result += n * m;
```

Рассмотрим выражение:

```
result += x2 * x2 + F(x1) - x1 * x1;
```

Здесь все вызовы свойств x1, x2, метода F(x) записаны без квалификации, но можно превратить их в квалифицированные, показав явным образом, что реально при вычислении выражения используется операция вызова "точка". В последних примерах предполагается, что n, m, x1, x2 являются полями класса, а F - методом класса.
В неуправляемом коде, который появляется в блоках, объявленных как небезопасные, разрешена работа с указателями. Вызов полей и методов объекта, когда целью является указатель, задается операцией "стрелка" x -> y, где x - это указатель, а y - поле объекта, на который указывает указатель. Переходя от указателей к объекту, операцию "стрелка" можно заменить операцией "точка" следующим образом: (*x).y
В нашем курсе работа с указателями рассматриваться не будет.

Операция индексации a[i, j]

О массивах подробно поговорим в одной из ближайших лекций этого курса. Сейчас скажем, что если уже объявлен массив, то в выражении можно использовать элемент этого массива, задав индексы этого элемента. Так, например, если объявлен одномерный массив w, содержащий n элементов, то выражение w[i] будет определять i-й элемент этого массива, где индекс принимает значения от 0 до n-1.

Операция new

Ключевое слово "new" в языке C# в зависимости от контекста используется по-разному. Оно может задавать модификатор метода или операцию в выражениях. Операция new предназначена для создания объектов. Поскольку каждая реальная программа немыслима без объектов, операция new встречается практически во всех программах, хотя зачастую в неявной форме. Синтаксически эта операция имеет вид:

```
new <вызов конструктора объекта>
```

Чаще всего эта операция встречается в инициализаторах объекта в момент его объявления. Но допустимы и другие способы применения этой операции, скажем, в качестве фактического аргумента при вызове метода класса. Приведу совсем экзотический пример, где new встречается в арифметическом выражении:

```
Type tip = (n + new double()).GetType();
```

Рассмотрим обычное объявление скалярной переменной значимого типа:

```
int x = 77;
```

Это объявление можно рассматривать как краткую форму записи следующих операторов:

```
int x = new int(); x = 77;
```

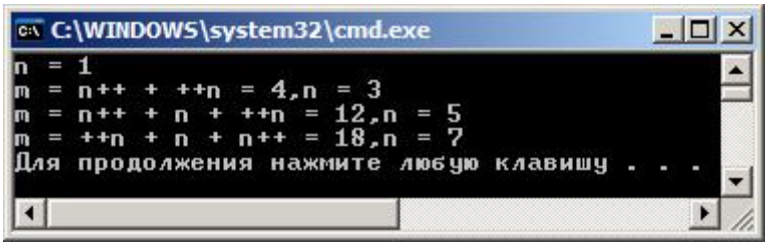
```
 Console.WriteLine("Размер типа double = " + sizeof(double));
 Console.WriteLine("Размер типа char = " + sizeof(System.Char));
  //Console.WriteLine("Размер класса TestingExpressoins = " +
  //  sizeof(TestingExpressions));

  int b1 = 1;
    Console.WriteLine("Тип переменной int b1: {0}, {1}",
         b1.GetType(), typeof(int));
    Console.WriteLine("Тип класса TestingExpressoins = {0}",
      typeof(TestingExpressions));
 }//SizeMethod
```

В этом примере операция применяется к трем встроенным типам - bool, double, char. Попытка применить эту операцию к собственному классу приводит к ошибке компиляции и потому закомментирована.

Операции "увеличить" и "уменьшить" (increment, decrement)
Операции "увеличить на единицу" и "уменьшить на единицу" могут быть префиксными и постфиксными. В справочной системе утверждается, что к высшему приоритету относятся постфиксные операции **x++** и **x--**, это нашло отражение в таблице 3.1.Префиксные операции имеют на единицу меньший приоритет.
В качестве результата обе операции возвращают значение переменной x. 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 операции значение x до изменения. Префиксные операции вначале изменяют x, а затем возвращают результат. Постфиксные операции возвращают значение, а потом изменяют саму переменную. Приведу пример применения этих операций:

```
public void IncDec()
  {
   int n = 1, m = 0;
   Console.WriteLine("n = {0}", n);
   m = n++ + ++n;
   Console.WriteLine("m = n++ + ++n = {0},n = {1}", m, n);
     m = n++ + n + ++n;
    Console.WriteLine("m = n++ + n + ++n = {0},n = {1}", m, n);
     m = ++n + n + n++;
    Console.WriteLine("m = ++n + n + n++ = {0},n = {1}", m, n);
  }
```

Обратите внимание: хотя у постфиксной операции высший приоритет, это вовсе не означает, что при вычислении выражений вначале выполняются все постфиксные операции, затем все префиксные, и только потом будет проводиться сложение. Нет, вычисления проводятся в том порядке, в котором они написаны. Префиксные и постфиксные операции выполняются тогда, когда нужно вычислить соответствующий операнд.



Консольный вывод выполнения этого метода дает результат, показанный на рис. 3.2.

Рис. 3.2. Результат выполнения метода IncDec
Следует также заметить, что рассматриваемые 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. Как только к x слева или справа приписана одна из операций, выражение перестает принадлежать к классу l-value выражений и вторую операцию приписать уже невозможно.
Подводя итоги, отмечу, что операции выполняются только тогда, когда вычисляется соответствующий операнд, а не в соответствии с приоритетом, указанным в таблице 3.1. Важнее помнить, что хороший стиль программирования рекомендует использовать эти операции только в выражениях, не содержащих других операндов. Еще лучше вообще не использовать их в выражениях, а применять их только как операторы:

```
x++;        y--;
```

В этом случае фактически исчезает побочный эффект, являющийся опасным средством, и операции используются как краткая запись операторов:

```
x = x + 1;   y = y - 1;
```

Унарные операции приоритета 1
Следующий по важности приоритет имеют унарные операции. Префиксные операции ++x и -x уже подробно рассмотрены. Арифметические унарные операции + и - не требуют особых пояснений. О логических унарных операциях отрицания, задаваемых знаками ! и ~ скажем чуть позже. А сейчас рассмотрим оставшуюся унарную операцию.
Операция кастинга - приведения к типу
Рассмотрим примеры приведения типа:

```
byte b1 = 1, b2 = 2, b3;
//b3 = b1 + b2;
b3 = (byte)(b1 + b2);
```

В этом примере необходимо сложить две переменные типа byte и, казалось бы, никакого приведении типов выполнять не нужно, результат будет также иметь тип byte, согласованный с левой частью оператора присваивания. Однако это не так по той простой причине, что отсутствует операция сложения над короткими числами. Реализация сложения начинается с типа int. Поэтому перед выполнением сложения оба операнда неявно преобразуются к типу int, результат сложения будет иметь тип int, и при попытке присвоить значение выражения переменной типа byte возникнет ошибка периода компиляции. По этой причине оператор во второй строке кода закомментирован. Программист вправе явно привести выражение к типу byte, что и демонстрирует третья строка кода, в которой использована операция приведения к типу.

В следующем фрагменте кода демонстрируется еще один пример приведения типа:

```
int tempFar, tempCels;
tempCels = -40;
tempFar = (int)(1.8 * tempCels) + 32;
```

Результат умножения имеет тип double по типу первого операнда. Перед тем как выполнять сложение, результат приводится к типу int. После приведения сложение будет выполняться над целыми числами, результат будет иметь тип int, и не потребуется никаких преобразований для присвоения полученного значения переменной tempFar. Если убрать приведение типа в этом операторе, то возникнет ошибка на этапе компиляции.
Рассмотрим еще один пример:

```
//if ((bool)1) b3 = 100;
            if (Convert.ToBoolean(1)) b3 = 100;
```

В этом примере показана попытка применить кастинг для приведения типа int к типу bool. Такое преобразование типа с помощью операции кастинга не разрешается и приводит к ошибке на этапе компиляции. Но, заметьте, это преобразование можно выполнить более мощными методами класса Convert .

Арифметические операции

В языке C# имеются обычные для всех языков **арифметические операции** - "+, -, *, /, %". Все они перегружены. Операции "+" и "-" могут быть унарными и бинарными. Унарные операции приписывания знака арифметическому выражению имеют наивысший приоритет среди арифметических операций. К следующему приоритету относятся арифметические операции типа умножения, к которому относятся три операции - умножения, деления и взятия остатка. Все эти операции перегружены и определены для разных подтипов арифметического типа. Следует, однако, помнить, что арифметические операции не определены над короткими числами (byte, short) и начинаются с типа int.
Операция деления "/" над целыми типами осуществляет деление нацело, для типов с плавающей и фиксированной точкой - обычное деление. Операция "%" возвращает остаток от деления нацело и определена не только над целыми типами, но и над типами с плавающей точкой. Тип результата зависит от типов операндов. Приведу пример вычислений с различными арифметическими типами:

```
/// <summary>
/// Арифметические операции
/// </summary>
public void Ariphmetica()
{
   byte b1 = 7, b2 = 3, b3;
   b3 = (byte)(b1 / b2);
   int n = -7, m = 3, p, q, r;
   p = n / m; q = n % m; r = p*m + q;
   Console.WriteLine("Операции над типом int");
   Console.WriteLine(
      "n = {0}, m = {1}, p = n/m = {2}, " +
      "q = n % m = {3}, r = p*m + q = {4}",
      n, m, p, q, r);

   Console.WriteLine("Операции над типом double");
   double x = 7.5, y = 3.5, u, v, w;
   u = x / y; v = u * y;
   w = x % y;
   Console.WriteLine(
      "x = {0}, y = {1}, u = x/y = {2}, " +
      "v = u*y = {3}, w = x % y = {4}",
      x, y, u, v, w);

   Console.WriteLine("Операции над типом decimal");
   decimal d1 = 7.5M, d2 = 3.5M, d3, d4, d5;
   d3 = d1 / d2; d4 = d3 * d2;
   d5 = d1 % d2;
   Console.WriteLine(
      "d1 = {0}, d2 = {1}, d3 = d1/d2 = {2}, " +
      "d4 = d3*d2 = {3}, d5 = d1 % d2 = {4}",
      d1, d2, d3, d4, d5);
}//Ariphmetica
```

Результаты вычислений при вызове этого метода показаны на рис. 3.3.

увеличить изображение
Рис. 3.3. Результаты работы метода Ariphmetica

Для целых типов можно исходить из того, что равенство $n = (n/m) * m + n$ 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 происходит при вычислениях с фиксированной точкой.

Вычисление выражений

Как уже говорилось, при записи выражения от программиста требуется знание всех операций, которые могут применяться в построении выражений, знание их точной семантики, понимание тех преобразований операндов, которые могут осуществляться при выполнении операций. Но есть и другие не менее важные цели, которые следует ставить на этом этапе.

Память и время - два основных ресурса

В распоряжении программиста при решении задач есть два основных ресурса - это память компьютера и его быстродействие. Кажется, что оба эти ресурса практически безграничны, и потому можно не задумываться о том, как они расходуются. Эти представления иллюзорны. Многие задачи, возникающие на практике, таковы, что имеющихся ресурсов не хватает и требуется жесткая их экономия. Вот два простых примера. Если в программе есть трехмерный массив A: double[,,]; A = new double[n,n,n], то уже при n =1000 оперативной памяти современных компьютеров не хватит для хранения элементов этого массива. Если приходится решать задачу, подобную задаче о "ханойской башне", где время решения задачи $T = O(2^n)$, то уже при n = 64 никакого быстродействия всех современных компьютеров не хватит для решения этой задачи в сколь-либо допустимые сроки. Программист обязан уметь оценивать объем ресурсов, требуемых программе. Говоря о ресурсах, требуемых программе P, часто используют термины "временная" и "емкостная сложность" - T(P) и V(P). Выражения представляют хорошую начальную базу для оценивания этих характеристик.

Характеристики T(P) и V(P) обычно взаимосвязаны. Увеличивая расходы памяти, можно уменьшить время решения задачи или, выбирая другое решение, сократить расходы памяти, увеличивая время работы. Одна из реальных задач, стоящих перед профессиональным программистом - это нахождение нужного компромисса между памятью и временем. Помните:

"Выбора тяжко бремя - память или время!"

Как этот компромисс достигается на уровне выражений? Если в исходном выражении можно выделить повторяющиеся подвыражения, то для них следует ввести временные переменные. Увеличивая расходы памяти на введение дополнительных переменных, уменьшаем общее время вычисления выражения, поскольку каждое из подвыражений будет вычисляться только один раз. Этот прием целесообразно применять и тогда, когда не преследуется цель экономии времени. Введение дополнительных переменных уменьшает сложность выражения, что облегчает его отладку и способствует повышению надежности программы. Вероятность допустить ошибку в записи громоздкого выражения значительно выше, чем при записи нескольких простых выражений.

Именованные константы

Еще один важный урок, который следует помнить, касается констант, участвующих в записи выражения.

"Каждой константе имя давайте, Числа без имени из программ изгоняйте!"

Исключением могут быть простые константы - 0, 1, 2, 3. Если, как это часто бывает, изменяется значение константы, то это изменение должно делаться только в одном месте - там, где эта константа определяется. Введение констант уменьшает время вычислений, поскольку константы, заданные выражениями, вычисляются еще на этапе компиляции.

Рассмотрим в качестве примера вычисление значений переменных x и y, заданных следующими выражениями:

$$x = \frac{(a + 53.5 * 33/37^2) * (a - 53.5 * 33/37^2)}{\sqrt[3]{(133 + 53.5 * 33/37^2)}}$$

$$y = \frac{(a + 53.5 * 33/37^2)}{(a - 53.5 * 33/37^2)}$$

Вычислять эти выражения, точно следуя приведенной записи, не следует. Вот как можно организовать эти вычисления:

```
public void EvalXY(double a, out double x, out double y)
  {
   const double C1 = 53.5 * 33 / (37 * 37);
   const double C2 = 133 + C1, C3 = 1.0 / 3;
   double t1 = a + C1, t2 = a - C1;
   x = t1 * t2 / Math.Pow(C2, C3);
   y = t1 / t2;
  }
```

Заметьте, константы будут вычислены еще на этапе компиляции, так что для вычисления выражений потребуется 5 арифметических операций и один вызов стандартной функции. Выигрыш кажется незначительным при тех скоростях, которыми обладают компьютеры. Но стоит учесть, что метод EvalXY может вызываться многократно. И главное - даже не выигрыш во времени вычислений. Более важно, что запись выражения становится простой и позволяет легко обнаруживать ее ошибки.
Многие из моих студентов совершают типичную ошибку, записывая, например, выражение для вычисления x следующим образом:
x = t1 * t2 / Math.Pow(133 + C1, 1 / 3)

Надеюсь, что читатель ошибку видит, но на всякий случай поясню, что она связана с вычислением второго аргумента функции возведения в степень Pow. Здесь применяется операция деления, операнды которой - целые числа, потому результат деления нацело будет равен нулю. Обнаружить ошибку студенты могут далеко не сразу. В процедуре EvalXY ошибка становится видна мгновенно, стоит только взглянуть на значения констант, вычисленных еще на этапе компиляции
Операции отношения стоит просто перечислить, в объяснениях они не нуждаются. Всего операций 6 **(==, !=, <, >, <=, >=)**, все они возвращают результат логического типа bool. Операции перегружены, так что их операнды могут быть разных типов. Понятно, что перед вычислением отношения может потребоваться преобразование типа одного из операндов. Понятно, что не всегда возможны неявные преобразования, гарантирующие возможность выполнения сравнения. Возникнет ошибка на этапе компиляции в выражении:
1 > "1"

Задав явное преобразование типа для одного из операндов, это отношение можно вычислить. Следует обратить внимание на запись отношения эквивалентности, задаваемое двумя знаками равенства. Типичной ошибкой является привычная для математики запись:
if(a = b)

Выражение в скобках синтаксически корректно и воспринимается, как запись операции присваивания, допустимой в выражениях. К счастью, в большинстве случаев возникнет ошибка на этапе компиляции при попытке преобразования значения операнда b к типу bool. Но, если a и b - переменные логического типа, то никаких сообщений об ошибке выдаваться не будет, хотя результат выполнения может быть неправильным.
Операции проверки типов
Операции проверки типов is и as будут рассмотрены в последующих лекциях.
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 1. 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
Логические операции в языке C# делятся на две категории: одни выполняются только над операндами типа bool, другие - как над булевскими, так и над целочисленными операндами.
Логические операции над булевскими операндами
Операций, которые выполняются только над операндами булевского типа, три (!, &&, ||). Высший приоритет среди этих операций имеет унарная операция отрицания ! x, которая возвращает в качестве результата значение, противоположное значению выражения x. Поскольку неявных преобразований типа к типу bool не существует, то выражение x задается либо переменной булевского типа, либо, как чаще бывает, выражением отношения. Возможна ситуация, когда некоторое выражение явным образом преобразуется к булевскому типу.

Следующая по приоритету бинарная операция (x && y) называется конъюнкцией, операцией "И" или логическим умножением. Она возвращает значение true в том и только в том случае, когда оба операнда имеют значение true. В остальных случаях возвращается значение false.
Следующая по приоритету бинарная операция (x || y) называется дизъюнкцией, операцией "ИЛИ" или логическим сложением. Она возвращает значение false в том и только в том случае, когда оба операнда имеют значение false. В остальных случаях возвращается значение true.
Когда описывается семантика операций, молчаливо предполагается, что операнды операции определены. Подразумевается, что результат операции не определен, если не определен хотя бы один из ее операндов. Это утверждение верно почти для всех операций языка C#. К исключениям относятся рассматриваемые нами логические операции && и ||. Эти операции называются условными логическими операциями. Если первый операнд операции конъюнкции && ложен, то второй операнд не вычисляется и результат операции равен false, даже если второй операнд не определен. Аналогично, если первый операнд операции дизъюнкции || истинен, то при выполнении этого условия второй операнд не вычисляется и результат операции равен true, даже если второй операнд не определен.
Ценность условных логических операций не в их эффективности по времени выполнения. Часто они позволяют вычислить имеющее смысл логическое выражение, в котором второй операнд не определен. Приведу в качестве примера классическую задачу поиска по образцу в массиве, когда в массиве разыскивается элемент с заданным значением (образец). Такой элемент может быть, а может и не быть в массиве. Вот типичное решение этой задачи:

```
        //Условное And - &&
        public int SearchPattern(int[] arr, int pattern)
{
   int result = -1, index = 0;
   int n = arr.Length;
   while (index < n && arr[index] != pattern) index++;
   if (index != n) result = index;
   return (result);
}
```

Обратите внимание на выражение, задающее условие цикла while. Здесь условная конъюнкция выполняется над двумя отношениями. В том случае, когда образца нет в массиве, наступает момент, когда первый операнд становится ложным, в этот же момент второй операнд не определен, поскольку индекс проверяемого элемента массива выходит за допустимые пределы. Классическая конъюнкция должна в этот момент приводить к ошибке, возникновению исключительной ситуации. Но условная конъюнкция прекрасно справляется, и программа корректно работает во всех случаях.
Логические операции над булевскими операндами и целыми числами. Работа со шкалами
Рассмотрим логические операции, которые могут выполняться не только над булевскими значениями, но и над целыми числами. Высший приоритет среди этих операций имеет унарная операция отрицания (~x). Заметьте: есть две операции отрицания, одна из них (!x) определена только над операндами булевского типа, другая (~x) - только над целочисленными операндами.
Говоря о логических операциях над целыми числами, следует понимать, что целые числа можно рассматривать как последовательность битов (разрядов). Каждый бит, имеющий значение 0 или 1, можно интерпретировать как логическое значение обычным образом: 0 соответствует false, 1 - true. Логическая операция, применяемая к операндам одного и того же целочисленного типа, выполняется над соответствующими парами битов, создавая результат в виде последовательности битов и интерпретируемый как целое число. По этой причине такие логические операции называются побитовыми или поразрядными операциями.
Бинарных побитовых логических операций три - & , ^ , |. В порядке следования приоритетов это конъюнкция (операция "И"), исключающее ИЛИ, дизъюнкция (операция "ИЛИ"). 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. Когда эти операции выполняются над булевскими операндами, то оба операнда вычисляются в любом случае, и если хотя бы один из операндов не определен, то и результат операции будет не определен. Когда необходима такая семантика логических операций, тогда без этих операций не обойтись.
Поразрядные логические операции определены не только над целыми числами, но и над перечислениями, которые проецируются на целочисленные типы. В реальных приложениях они чаще всего используются при работе со шкалами, часто представляемыми переменными перечислимого типа.
Шкалы
Побитовые логические операции широко применяются в реальном программировании при работе с так называемыми шкалами. Будем называть **шкалой** последовательность из n битов (n разрядов). Рассмотрим объект с n свойствами, каждым из которых объект может обладать или не обладать. Шкала позволяет однозначно задать, какими свойствами объект обладает, а какими нет. Пронумеруем свойства и будем записывать единицу в разряд с номером i, если объект обладает i-м свойством, и нуль - в противном случае.
Шкала позволяет экономно задавать информацию об объекте, а побитовые операции дают возможность весьма эффективно эту информацию обрабатывать. Поскольку эти операции определены над типами int, uint, long, ulong, C# может работать со шкалами длины 32 и 64.
Описание свойств объекта можно задать, используя перечисление - специальный тип данных, определяемый программистом. Свойства конкретного объекта, его шкалу можно задать переменной типа перечисление. При работе с такими переменными существенно используются поразрядные операции.

Рассмотрим содержательный пример. Пусть некоторая программистская фирма объявила прием на работу в фирме, предъявляя к претендентам такие требования: знание технологий и языков программирования. Возможный набор профессиональных свойств, которыми могут обладать претенденты на должность, можно задать перечислением:

```
// <summary>
/// Свойства претендентов на должность программиста,
/// описывающие знание технологий и языков программирования
/// </summary>
public enum Prog_Properties
{
   VB = 1, C_sharp = 2, C_plus_plus = 4,
   Web = 8, Prog_1C = 16
}
```

Заметьте, при определении перечисления можно указать, на какое значение целого типа проецируется значение из перечисления. Если проецировать i-е значение на i-й разряд целого числа (2^{i-1}), как это сделано в примере, то переменные перечисления будут задавать шкалу свойств.
Свойства каждого претендента на должность характеризуются своей шкалой, которую можно рассматривать как переменную типа Prog_Properties. Задать шкалу претендента можно целым числом в интервале от 0 до $2^n - 1$, приведя значение к нужному типу. Например, так:

```
Prog_Properties candidate1 = (Prog_Properties)18;
```

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;
```

Условное выражение начинается с условия, заключенного в круглые скобки, после которого следует знак вопроса и пара выражений, разделенных двоеточием " : ". Условие задается выражением 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.
Заметьте, условное выражение является примером тернарного выражения - выражения с тремя операндами. И здесь, как и в случае условных логических операций, не требуется, чтобы все операнды были определены. Если булевский операнд определен и имеет значение true, то вычисляется второй операнд, а третий операнд не вычисляется и может быть в этот момент не определен.
Операция присваивания
Нам осталось рассмотреть последнюю по приоритету, но не по важности операцию присваивания. Синтаксически присваивание в языке C# является частным случаем выражения и может встречаться всюду, где разрешено появление выражений. Выражение присваивания является выражением с побочным эффектом. Оно не только возвращает значение некоторого типа в качестве результата, но и изменяет значения одной или нескольких переменных в качестве побочного эффекта. В реальных программах присваивание чаще всего встречается не как выражение, а как оператор присваивания. Превратить выражение присваивания в оператор присваивания проще простого. Нужно выражение закончить символом точка с запятой и использовать его там, где разрешается использовать операторы.
Начнем с формального определения синтаксиса выражения присваивания:
<переменная> <знак присваивания> <выражение>

Знаков присваивания много, они перечислены в таблице 3.1. Чаще всего используется знак равенства, но иногда ему могут предшествовать и другие знаки операций. С чем связано наличие многих знаков у одной операции? Языку C# это досталось в наследство от языка C++, где авторы языка были большими любителями краткости записи в ущерб ее ясности. Поэтому в языке допустимы такие выражения, как x++, x+=y, мало понятные обычному математику. Второе из этих выражений является выражением присваивания и удовлетворяет выше приведенному синтаксису со знаком присваивания +=. Его можно рассматривать как краткую запись выражения $x = x + y$. Аналогичный смысл имеют и другие знаки присваивания - (*=, /= и другие).
В правой части синтаксической формулы, определяющей выражение присваивания, стоит выражение, которое может быть в свою очередь выражением присваивания. Отсюда следует допустимость множественного присваивания. Синтаксически вполне корректен следующий пример:

```
/// <summary>
/// Множественное присваивание
/// </summary>
static void AssignTest()
{
   double x = 1, y = 2, z = 3, w =9, u = 7, v = 5;
  if((x += y -= z += w *= (u + v) / (u - v)) < 0)
  Console.WriteLine("x = {0}, y = {1}, z = {2}," +
```

```
  "w = {3}, u = {{4}, v = {5}", x, y, z, w, u, v);
 }
```

В операторе if записано выражение, задающее множественное присваивание. Какова семантика, как вычисляются выражения присваивания?
Операция присваивания является правосторонней операцией, и особенностью вычисления выражения присваивания является то, что оно вычисляется справа налево. В нашем примере вначале будет вычислено самое правое выражение (u + v) / (u - v), значение которого будет равно 6. Двигаясь налево по ходу присваивания, значение выражения будет изменяться. Последним будет вычислено выражение, которое получит переменная x. Значение этого выражения равно -54, именно оно является окончательным значением выражения множественного присваивания и будет участвовать в сравнении с нулем. Условие в операторе if получит значение true, и метод WriteLine выведет на консоль значения переменных, полученных ими как побочный результат вычисления выражения присваивания. Эти значения соответственно равны: -54, -55, 57, 54, 7, 5. Заметьте, скобки, окружающие выражение присваивания, необходимы, иначе операция сравнения выполнялась бы раньше, чем присваивание, что приводило бы к ошибке.
Для пояснения деталей семантики выражений присваивания использована довольно экзотическая конструкция в операторе if. В реальных программах такие конструкции применять не следует. Они "от лукавого". Простота записи и понимаемость - одни из главных критериев при создании промышленного кода. При изучении возможностей языка допустимо рассмотрение экзотических случаев.

Операция ?? - новая операция C# 2.0

Эта операция уже рассматривалась в предыдущей лекции, когда речь шла о типах, допускающих значение null. Напомню, все ссылочные типы изначально допускают null в качестве возможного значения. Такое значение ссылочной переменной задает неопределенную ссылку, ссылку на несуществующий объект. Значимые типы значения null не содержат, но можно определить расширенный значимый тип, включающий значение null. Синтаксически, если T - имя значимого типа, то T? - это имя расширенного типа. Операция ?? определена над операндами, допускающими значение null. Ее главная задача - присвоить переменной значение, отличное от null, поэтому иногда ее называют операцией склеивания, поскольку она позволяет "приклеить" к null значение. Рассмотрим ее определение:

```
A ?? B
```

Если операнд A отличен от null, то он и возвращается в качества результата операции. Если же он имеет значение null, то результатом является операнд B. Эту операцию особенно удобно использовать при приведении типа T? к типу T. Рассмотрим простой пример:

```
int? x = null;
…
int y = x ?? 0;
```

Заметьте, если между двумя присваиваниями переменная x не приобрела значение, отличное от null, то переменная y в результате получит значение 0.
В отсутствии такой операции нам пришлось бы писать для вычисления y такую эквивалентную конструкцию:

```
int y = (x !=null) ? (int)x : 0
```

Лямбда-оператор - новая операция в C# 3.0

В третьей версии языка появилась новая операция, называемая лямбда-оператором, и, соответственно, новый тип выражений, называемых лямбда-выражениями. Эта операция имеет тот же приоритет, что и операция присваивания, и, так же как и последняя, является правосторонней операцией. Синтаксис лямбда-выражений следующий:

```
<(список входных аргументов)>  =>  <выражение>
```

Содержательно выражение в правой части задает описание функции, аргументы которой задаются списком левой части. Такое описание представляет собой описание анонимной функции - функции без имени - и может быть использовано, например, при задании экземпляра делегата. Зачастую функция зависит от одного аргумента, и тогда в левой части можно указывать только имя этого аргумента, опуская скобки.
Подробно этот механизм будет рассмотрен в отдельной лекции нашего курса, а пока приведем первый простой пример использования этой операции. Рассмотрим следующую задачу. Пусть дан массив чисел X и задана функция F(x). Требуется найти минимальное значение этой функции, когда аргументы задаются элементами массива X. Конечно же, можно создать метод, реализующий вычисление функции F(x), но можно воспользоваться анонимной функцией, заданной лямбда-выражением, что демонстрирует следующий пример:

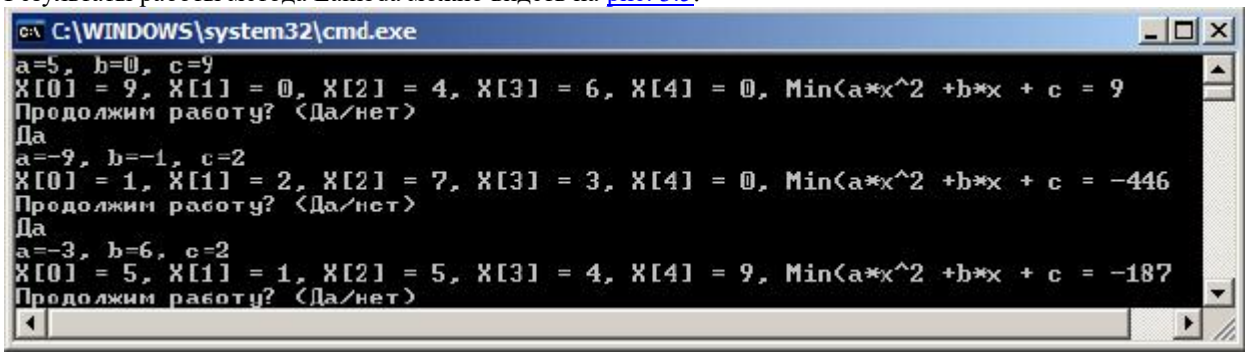
```
/// <summary>
 /// Лямбда-оператор и лямбда-выражение
 /// </summary>
 static void Lambda()
 {
    Random rnd = new Random();
    const int size = 5;
    int[] numbers = new int[size];
    int a = rnd.Next(-10, 10), b = rnd.Next(-10, 10),
      c = rnd.Next(-10, 10);
    Console.WriteLine("a={0}, b={1}, c={2}",
```

```
      a, b, c);
    for (int i = 0; i < size; i++)
    {
   numbers[i] = rnd.Next(10);
   Console.Write("X[{0}] = {1}, ", i, numbers[i]);
    }
    int minValue = numbers.Min(x => a * x * x + b * x + c);
    Console.WriteLine("Min(a*x^2 +b*x + c) = {0}", minValue);

 }
```

Большая часть в этом примере связана с моделированием массива чисел и коэффициентов функции. Нахождение минимума функции задается одной строкой:

```
int minValue = numbers.Min(x => a * x * x + b * x + c);
```

Здесь функция Min последовательно перебирает элементы массива, формируя аргумент x функции, а лямбда-выражение преобразует его в значение функции от этого аргумента. В результате возвращается минимальное значение функции. Результаты работы метода Lambda можно видеть на рис. 3.5.



увеличить **изображение**

Рис. 3.5. Результаты работы метода Lambda

На этом закончим рассмотрение операций языка C#, но продолжим разговор о некоторых вопросах, связанных с вычислением выражений.

Преобразования внутри арифметического типа

Арифметический тип распадается на 11 подтипов. На рис. 3.6 показана схема преобразований внутри арифметического типа.

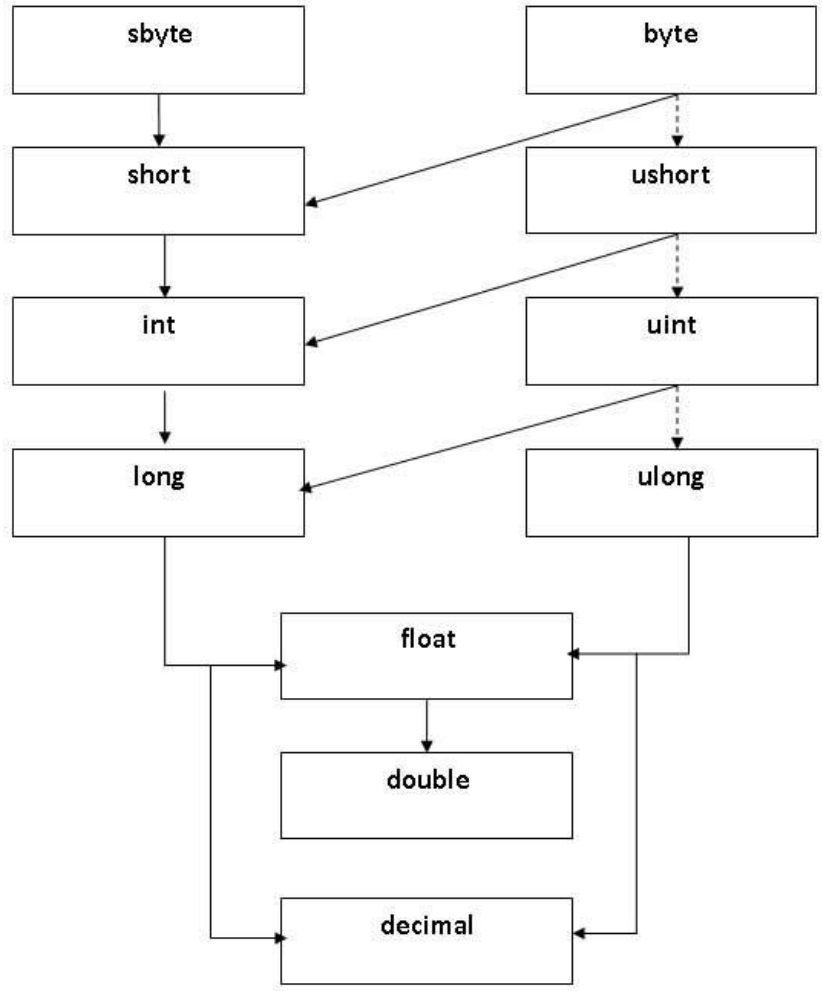


Рис. 3.6. Иерархия преобразований внутри арифметического типа

Диаграмма, приведенная на рисунке, позволяет отвечать 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 значит, что выполняется вся последовательность преобразований, следуя данному пути. Наличие пути говорит лишь о существовании неявного преобразования, само преобразование выполняется только один раз - из типа источника A в тип цели B.

Иногда встречается ситуация, при которой для одного типа источника может одновременно существовать несколько целевых типов и необходимо осуществить однозначный выбор цели. Такие проблемы выбора возникают, например, при работе с перегруженными методами в классах.

Диаграмма, приведенная на рис. 3.6, 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 TestingExpressions включена группа перегруженных методов OnLoad 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.

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 рис. 3.6.

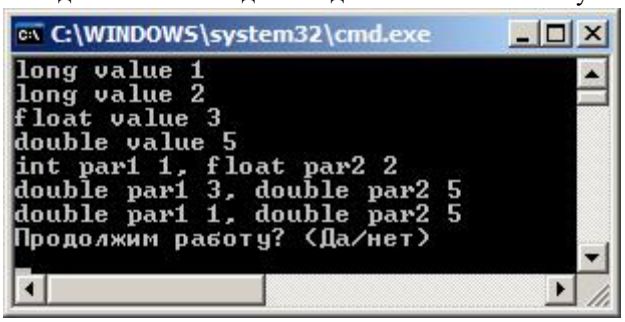


Рис. 3.7. Вывод на печать результатов теста OLoadTest

Приведу некоторые комментарии. При первом вызове метода тип источника - int, а тип аргумента у четырех возможных реализаций - 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, хотя бы так, как это сделано в примере. Обратите внимание: обе реализации допустимы, и существуй только одна из них, ошибки бы не возникало.

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 внутри арифметического типа чаще всего выполняются с использованием приведения типа - кастинга. Конечно, можно применять и более мощные методы класса Convert, но чаще используется кастинг.

Выражения над строками. Преобразования строк

Начнем с символьного типа. Давайте уточним, какие выражения можно строить над операндами этого типа. На алфавите символов определен порядок, задаваемый Unicode кодировкой символов. Знать, как кодируется тот или иной символ, не обязательно, но следует помнить, что кодировка буквенных символов таких алфавитов, как кириллица, латиница и других языковых алфавитов, являющихся частью Unicode алфавита, является плотной, так что, например, код буквы "а" на единицу меньше кода буквы "б". Исключение составляет буква "Ё", выпадающая из плотной кодировки. Большие буквы (заглавные) в кодировке предшествуют малым буквам (строчным). Для цифр также используется плотная кодировка.

Поскольку каждому символу однозначно соответствует его код, существует неявное, автоматически выполняемое преобразование в целочисленный тип (int и выше). Обратное преобразование также существует, но должно быть явным. Вот пример, показывающий, как по символу получить его код и как по коду получить символ, соответствующий коду.

```
    char sym = 'Ё';
    int code_Sym = sym;
   Console.WriteLine("sym = {0}, code = {1}",
   sym, code_Sym);
code_Sym++;
sym = (char)code_Sym;
Console.WriteLine("sym = {0}, code = {1}",
   sym, code_Sym);
```

Существование неявного преобразования между типом char и целочисленными типами позволяет рассматривать тип char как целочисленный. Как следствие, к операндам символьного типа применимы все операции, применимые к целочисленным типам - операции отношения, арифметические операции, логические операции над целыми числами. В реальных программах к таким операндам чаще всего применяются операции сравнения и операция вычитания, позволяющая определить "расстояние" между символами в кодировке.
Рассмотрим содержательный пример, в котором используются операции сравнения символов.

```
/// <summary>
 /// Соответствует ли s требованиям,
 /// предъявляемым к именам в русском языке:
 /// Первый символ - большая буква кириллицы
 /// Остальные символы - малые буквы кириллицы
 /// </summary>
 /// <param name="s">входная строка</param>
 /// <returns>
 /// true, если s соответствует правилам,
 /// false -  в противном случае
 /// </returns>
 public bool IsName(string s)
 {
    if (s == "") return false;
    char letter = s[0];
    if(!(letter >= 'А' && letter <= 'Я'))return false;
    for(int i=1; i< s.Length; i++)
    {
  letter = s[i];
  if (!(letter >= 'а' && letter <= 'я')) return false;
    }
    return true;
 }
```

Рассмотрим теперь строковый тип. В некоторых языках программирования используется тот факт, что порядок на символах алфавита порождает лексикографический порядок на словах, составленных из этих символов. Но в языке C# это не так, и здесь не существует неявного преобразования строки в целочисленный тип. Понять причину этого не трудно. Строки C# имеют переменную длину и могут быть сколь угодно длинными, так что не существует безопасного преобразования ни в один из целочисленных типов.
По этой причине над операндами строкового типа из множества операций, задаваемых знаками логических, арифметических и операций отношения, определены только три операции. Две операции позволяют сравнивать строки на эквивалентность (==, !=). Третья операция, задаваемая знаком операции "+", называется операцией конкатенации или сцепления строк и позволяет вторую строку присоединить к концу первой строки. Вот пример:

```
string s1 = "Мир";
 if (s1 == "Мир" | s1 == "мир") s1 += " Вам";
 Console.WriteLine(s1);
```

Операций над строками немного, но методов вполне достаточно. Сравнивать две строки, используя знаки операций " >, < ", нельзя, но есть методы сравнения Compare, решающие эту задачу. О работе со строками более подробно поговорим в отдельной лекции.

Преобразования строкового типа в другие типы

Хотя неявных преобразований строкового типа в другие типы нет, необходимость в преобразованиях существует. Когда пользователь вводит данные различных типов, он задает эти данные как строки текста, поскольку текстовое представление наиболее естественно для человека. Поэтому при вводе практически всегда возникает задача преобразования данных, заданных текстом, в "настоящий" тип данных.
Классы библиотеки FCL предоставляют два способа явного выполнения таких преобразований:

- метод Parse;
- методы класса Convert.

Метод Parse

Все скалярные типы (арифметический, логический, символьный) имеют статический метод Parse, аргументом которого является строка, а возвращаемым результатом - объект соответствующего типа. Метод явно выполняет преобразование

текстового представления в тот тип данных, который был целью вызова статического метода. Понятно, что строка, представляющая аргумент вызова, должна соответствовать представлению данных соответствующего типа. Если это требование не выполняется, то в ходе выполнения метода возникнет исключительная ситуация. Приведу пример преобразования данных, выполняемых с использованием метода Parse.

```
static void InputVars()
 {
    string strInput;
    Console.WriteLine(INPUT_BYTE);
    strInput = Console.ReadLine();
    byte b1;
    b1 = byte.Parse(strInput);

    Console.WriteLine(INPUT_INT);
    strInput = Console.ReadLine();
    int n;
    n = int.Parse(strInput);

    Console.WriteLine(INPUT_FLOAT);
    strInput = Console.ReadLine();
    float x;
    x = float.Parse(strInput);

    Console.WriteLine(INPUT_CHAR);
    strInput = Console.ReadLine();
    char ch;
    ch = char.Parse(strInput);
 }
```

Здесь приглашение к вводу задается соответствующей строковой константой. Поскольку вызов метода Parse способен приводить к исключительной ситуации, корректно построенный ввод пользовательских данных предполагает помещение подобного вызова в охраняемый блок и построение соответствующего обработчика исключительной ситуации. Для краткости примера эта часть работы опущена.

Преобразование в строковый тип

Преобразования в строковый тип всегда определены, поскольку все типы являются потомками базового класса **object** и наследуют метод **ToString**(). Конечно, родительская реализация этого метода чаще всего не устраивает наследников. Поэтому при определении нового класса в нем должным образом переопределяется метод ToString. Для встроенных типов определена подходящая реализация этого метода. В частности, для всех подтипов арифметического типа метод ToString() возвращает строку, задающую соответствующее значение арифметического типа. Заметьте, метод ToString следует вызывать явно. В ряде ситуаций вызов метода может быть опущен, и он будет вызываться автоматически. Его, например, можно опускать при сложении числа и строки. Если один из операндов операции "+" является строкой, то операция воспринимается как конкатенация строк и второй операнд неявно преобразуется к этому типу. Вот соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования в строку
/// данных различного типа.
/// </summary>
public void ToStringTest()
{
   string name;
   uint age;
   double salary;
   name = "Владимир Петров";
   age = 27;
   salary = 27000;
   string s = "Имя: " + name +
     ". Возраст: " + age.ToString() +
     ". Зарплата: " + salary;
   Console.WriteLine(s);
}
```

Здесь для переменной age метод был вызван явно, а для переменной salary он вызывается автоматически.

Класс Convert и его методы

Для преобразований внутри арифметического типа можно использовать кастинг - приведение типа. Для преобразований строкового типа в скалярный тип можно применять метод Parse, а в обратную сторону - метод ToString.

Во всех ситуациях, когда требуется выполнить преобразование из одного базового встроенного типа в другой базовый тип, можно использовать методы класса **Convert библиотеки FCL**, встроенного в пространство имен **System** - универсального класса, статические методы которого специально спроектированы для выполнения преобразований.

Среди других методов класса Convert отмечу общий статический метод ChangeType, позволяющий преобразование объекта к некоторому заданному типу. Отмечу также возможность преобразования к системному типу DateTime, который хотя и не является базисным типом языка C#, но допустим в программах, как и любой другой системный тип.
Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования, связанные с массивами, и другие методы. Подробности можно посмотреть в справочной системе.
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. Каждый из этих 15 методов перегружен, и его аргумент x может принадлежать к любому из упомянутых типов. С учетом перегрузки с помощью методов этого класса можно осуществить любое из возможных преобразований одного типа в другой. Все методы осуществляют проверяемые преобразования и включают исключительную ситуацию всякий раз, когда преобразование осуществить невозможно или при выполнении преобразования происходит потеря точности. Приведу пример:

```
/// <summary>
  /// Тестирование методов класса Convert
  /// </summary>
  public void ConvertTest()
  {
     string s;
     byte b;
     int n;
     double x;
     bool flag;
     char sym;
     DateTime dt;
     sym = '7';
     s = Convert.ToString(sym);
     x = Convert.ToDouble(s);
     n = Convert.ToInt32(x);
     b = Convert.ToByte(n);
     flag = Convert.ToBoolean(b);
     x = Convert.ToDouble(flag);
     s = Convert.ToString(flag);
    // sym = Convert.ToChar(flag);

     s = "300";
     n = Convert.ToInt32(s);
     //b = Convert.ToByte(s);

     s ="14.09";
     //flag = Convert.ToBoolean(s);
     //x = Convert.ToDouble(s);

     s = "14.09.2008";
     dt = Convert.ToDateTime(s);
  }
```

Этот пример демонстрирует различные преобразования между типами. Все эти преобразования опасные, выполняются явно с использованием методов класса Convert. Вначале данные символьного типа преобразуются в строку. Затем эти данные преобразуются в вещественный тип, далее проводятся преобразования внутри арифметического типа с понижением типа от double до byte. Завершающим пример преобразованием является преобразование данных строкового типа к типу DateTime.
Опасные преобразования одного типа к другому могут успешно выполняться над некоторыми данными и приводить к ошибке с другими данными. В нашем примере закомментированы операторы, приводящие к ошибкам в период выполнения. Первая ошибка возникает при попытке преобразовать данные булевского типа к символьному типу, поскольку отсутствует преобразование, которое значение true преобразовывало бы в некоторый символ (например T). Заметьте, что в предыдущих операторах эти же данные успешно были приведены к строковому и арифметическому типу. Следующая ошибка возникает при попытке преобразовать строку со значением 300 к типу byte. Соответствующий метод распознает, что значение, записанное в строке, слишком велико для типа, представляющего цель преобразования. Еще одна ошибка возникает при попытке привести к булевскому типу строку, отличную от записи булевских констант true и false. Последняя ошибка в данном примере довольно часто встречается на практике. Она связана с тем, что ошибочно использована точка вместо запятой для отделения дробной части числа.
Какие выводы следует сделать? Опасные преобразования, выполняемые методами класса Convert, действительно опасны. По этой причине их всегда следует помещать в охраняемый блок и создавать блоки, обрабатывающие возможную исключительную ситуацию. Обработчик ситуации должен решать две важные задачи. Первая из них - информационная, он должен выдать достаточно подробную информацию с описанием возникшей ошибки. Вторая задача не менее важна.

Обработчик должен попытаться исправить ситуацию, чтобы программа могла продолжить нормальное выполнение. Он должен, например, позволить пользователю повторить ввод некорректно заданных данных, вызвать для корректировки ситуации дополнительный модуль. В каждом конкретном случае меры для восстановления работы могут быть разными.

Класс Console и его методы

В заключение этой лекции рассмотрим работу с методами класса Console. Хотя этот класс не связан непосредственно с выражениями - основной темой данной лекции, но он имеет прямое отношение к преобразованиям типов данных и вводу-выводу данных. Без использования методов этого класса в консольных проектах не обойтись.

Класс Console используется в консольных проектах, позволяя вводить исходные данные с консоли и выводить результаты на консоль. По умолчанию при вводе с консоли данные вводятся с клавиатуры и отображаются на дисплее, при выводе на консоль - данные отображаются на экране дисплея. У класса Console десятки свойств и методов. Ограничимся рассмотрением основных методов, используемых при вводе и выводе.

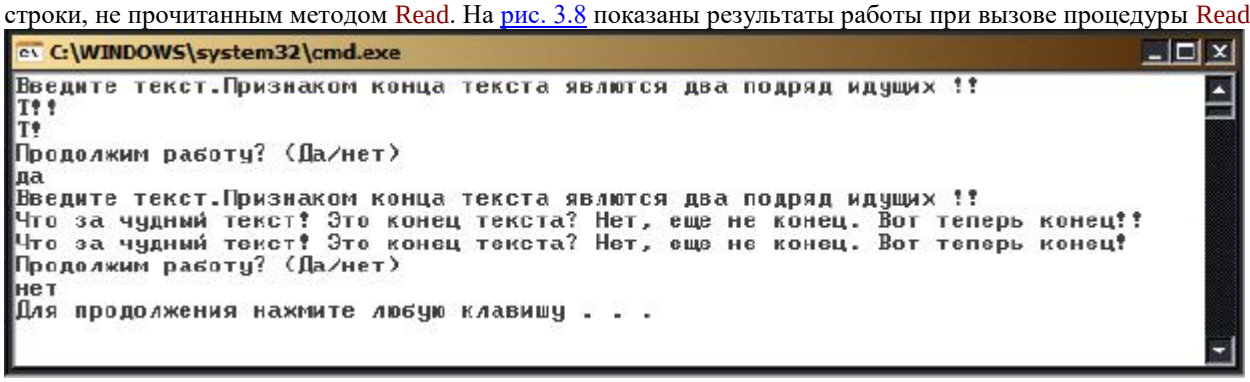
4

Методы Read и ReadLine позволяют читать с консоли текст, отображаемый на экране дисплея компьютера. Методы не имеют входных аргументов. Оператор Read читает по одному символу из входной строки и возвращает в качестве результата код прочитанного символа, имеющий тип int. Посимвольный ввод применяется довольно редко. Вот некоторый пример возможного применения чтения текста с использованием оператора Read.

```
public void ReadTest()
{
   Console.WriteLine("Введите текст." +
      "Признаком конца текста являются два подряд идущих !! ");
   char ch = Convert.ToChar(Console.Read());
   char next =' ';
   string result = "";
   bool finish = false;
   do
   {
      result += ch.ToString();
      next = Convert.ToChar(Console.Read());
      if (ch != '!')
         ch = next;
      else
      {
         if (next == '!') finish = true;
         else ch = next;
      }

   }while (!finish);
   Console.ReadLine();
   Console.WriteLine(result);
}
```

В этом примере текст, введенный пользователем, читается посимвольно до тех пор, пока не встретится специальный признак конца чтения. В данном случае таким признаком является два подряд идущих восклицательных знака.

Вызов метода ReadLine, завершающий процедуру, позволяет "дочитать" оставшиеся символы отображаемой строки текста и перевести курсор ввода на новую строку. Такие символы всегда будут, поскольку всякая строка завершается символом конца строки, не прочитанным методом Read. На рис. 3.8 показаны результаты работы при вызове процедуры ReadTest.



увеличить **изображение**

Рис. 3.8. Результаты работы метода ReadTest

Основным методом, используемым для чтения данных с консоли, является метод ReadLine. Он читает с консоли строку текста, завершаемую признаком конца строки. Эта строка и является результатом, возвращаемым методом ReadLine. Примеров применения этого метода было уже предостаточно.

Вывод данных на консоль. Методы Write и WriteLine

Методы Write и WriteLine позволяют выводить текст на консоль. Метод Write выводит текст на консоль и на этом завершает свою работу. Всякий последующий вывод на консоль продолжится с того места, на котором завершил свою работу метод

Write. В отличие от метода Write метод WriteLine выводит текст на консоль, после чего осуществляет переход на новую строку.
Выводимый текст задается аргументами методов. С аргументами методов стоит разобраться подробнее, поскольку у этих методов может быть сколь угодно много аргументов. В простейшем случае у методов один аргумент типа string, именно эта строка выводится на консоль. Но строка, задающая первый аргумент, может быть форматированной, и тогда после первого аргумента появляется дополнительный список аргументов, каждый из которых может иметь свой тип данных.
Строка называется форматированной, если она содержит форматы. Формат, включаемый в строку, задается последовательностью символов, заключенной в фигурные скобки. Каждый формат задает место подстановки. В процессе форматизации в строку вместо формата подставляется некоторая другая строка. Форматы могут быть разными, и подробнее о них поговорим при описании работы со строками. В простейшем случае задания формата в фигурных скобках стоит целое число k. Это число определяет порядковый номер аргумента из дополнительного списка, при этом нумерация аргументов списка начинается с нуля. Аргумент с номером k из дополнительного списка преобразуется в строку и подставляется вместо соответствующего формата. Преобразование аргумента в строку происходит автоматически, используя метод ToString, который имеют все типы данных.
Рассмотрим применение методов Write, WrteLine, ReadLine на примере ввода и вывода с консоли квадратной матрицы:

```
/// <summary>
/// Ввод-вывод с консоли квадратной матрицы
/// </summary>
public void InOutMatrix()
{
   int n;
   Console.WriteLine("Ввод квадратной матрицы A размерности n");
   Console.WriteLine("Введите целое - n");
   n = Convert.ToInt32(Console.ReadLine());
   double[,] A = new double[n,n];
   for(int i = 0; i<n; i++)
      for (int j = 0; j < n; j++)
      {
         Console.WriteLine("Введите A[{0}],[{1}]", i, j);
         A[i, j] = Convert.ToDouble(Console.ReadLine());
      }
   Console.WriteLine("Вывод матрицы A");
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
         Console.Write("A[{0}],[{1}] = {2}  ", i, j, A[i, j]);
      Console.WriteLine();
   }
}
```

На рис. 3.9 показаны результаты вызова этого метода.

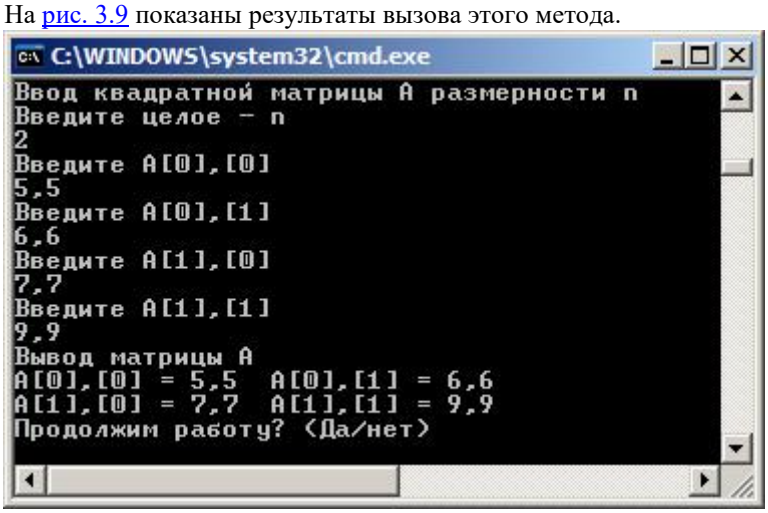


Рис. 3.9. Результаты работы метода InOutMatrix

Лекция 5

Тема: Основные управляющие структуры программирования

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

Синтаксически присваивание состоит из левой и правой частей, разделенных знаком операции присваивания. Правая часть - это выражение, в том числе выражение присваивания, как в последнем примере. Левая часть - это переменная; более точно: левая часть представляет собой lvalue - выражение левой части, которому можно присвоить значение. Переменная является наиболее распространенным частным случаем lvalue.

Выражение присваивания представляет собой пример выражения с побочным эффектом. Прямым эффектом вычисления такого выражения является вычисленное значение и тип выражения expr. Побочным эффектом является присваивание вычисленного значения переменной левой части.

Выражение с побочным эффектом в языке C# можно легко преобразовать в соответствующий оператор. Стоит такое выражение закончить символом точка с запятой, как получится оператор, который можно использовать всюду, где синтаксически допустимы операторы языка. Так что синтаксически оператор присваивания выглядит так:

X = expr;

Допустимы и многочисленные вариации:

X1 += X2 *= … = Xk = expr;

К операторам присваивания можно отнести и такие операторы, как:

X++; X--; ++X; --X;

Эти операторы получены из соответствующих выражений с побочным эффектом - приписыванием в конце символа точки с запятой. Когда выражения с побочным эффектом преобразуются в операторы, побочный эффект занимает подобающее ему место и задает семантику оператора, а вычисление значения выражения становится частью процесса выполнения оператора.

Семантика присваивания

Казалось бы, семантика присваивания проста и очевидна - вычисляем выражение правой части и его значение присваиваем соответствующей переменной левой части. Но это лишь общее описание семантики. Детали значительно сложнее. Дело в том, что левая и правая часть имеют свои типы, и эти типы могут не совпадать. В этом случае необходимо выполнить преобразование типа правой части к типу левой части. Иногда такое преобразование безопасно, и его можно выполнить автоматически. Иногда такое преобразование опасно, и тогда возникнет ошибка, которая чаще всего обнаруживается еще на этапе компиляции.

Будем называть **целью** левую часть оператора присваивания, а **источником** - правую часть оператора присваивания. Источник и цель могут быть как значимого, так и ссылочного типа. Присваивание будем называть **ссылочным**, если цель ссылочного типа. В этом случае источник должен быть ссылочного типа или быть приведенным к этому типу. Присваивание будем называть **значимым**, если цель значимого типа. В этом случае источник должен быть значимого типа или быть приведенным к этому типу.

Операции "упаковать" и "распаковать" - boxing и unboxing

Возникает естественный вопрос: можно ли ссылочным переменным, связанным с объектами, хранимыми в куче, присваивать значимые переменные, хранимые в стеке? Можно ли выполнять обратную операцию? В C# такие возможности преобразования типов предусмотрены. Операция "упаковать" (boxing) позволяет переменную значимого типа "упаковать в одежды класса", создавая объект в динамической памяти. Такое преобразование выполняется автоматически всякий раз, когда цель принадлежит классу object, а источником может быть переменная любого из значимых типов. Операция "распаковать" (unboxing) позволяет переменную типа object "распаковать и извлечь хранимое значение". Такое преобразование выполняется автоматически. Извлеченное значение не сохраняет информацию о своем типе. Поэтому, прежде чем присвоить это значение цели, его необходимо привести к нужному типу. Ответственность за это приведение лежит на программисте.

Рассмотрим подробнее, какие преобразования могут выполняться в процессе присваивания.

Цель и источник значимого типа. Здесь наличествует семантика значимого присваивания. В этом случае источник и цель имеют собственную память для хранения значений. Если типы цели и источника совпадают, то никаких проблем нет. Значения источника копируются и заменяют значения соответствующих полей цели. Источник и цель после этого продолжают жить независимо. У них своя память, хранящая после присваивания одинаковые значения. Если типы разные, то необходимо преобразование типов. Оно может быть безопасным и тогда выполняется автоматически. В противном случае оно должно явно задаваться программистом. Явные и неявные преобразования внутри арифметического типа, кастинг, метод Parse и методы класса Convert подробно рассматривались в лекции 2.

Цель и источник ссылочного типа. Здесь имеет место семантика ссылочного присваивания - присваивание ссылок. В этом случае значениями источника и цели являются ссылки на объекты, хранящиеся в динамической памяти ("куче"). Если типы источника и цели совпадают, то никаких проблем нет. 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" - бесполезным объектом, на который никто не ссылается, так что его дальнейшую судьбу определит сборщик мусора.
После присваивания 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 null - нулевой ссылки, не указывающей ни на какой объект. 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 null, а источник - нет, то в результате присваивания ранее "висячая" цель становится ссылкой на объект, связанный с источником.
Если типы источника и цели разные, то присваивание без всяких преобразований возможно лишь в том случае, если источник является потомком родительского класса, заданного целью. Цель-родитель может быть связана с объектом своего потомка, поскольку в этом случае все поля и методы родителя имеются и у потомка и будут определены.
Если же цель не принадлежит родительскому классу источника, то тогда ссылочное присваивание возможно лишь при условии явного задания приведения типов, но тогда вся ответственность за успех этого преобразования лежит на программисте, который должен быть уверен, что объект источника, связанный ссылкой, действительно принадлежит классу целевого объекта.
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 эффективно реализованной операции "упаковать" (boxing), выполняемой автоматически.
Такое присваивание возможно лишь в том случае, когда цель принадлежит классу object. Поскольку класс object является родителем для всех классов, в том числе и для значимых классов, при таком присваивании никаких ошибок возникать не будет, оно всегда возможно.
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 object 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.

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 авторы языков 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 не принадлежит оператору, а играет роль разделителя операторов. В выше приведенной 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.

Пустой оператор

Пустой оператор - это "пусто", завершаемое точкой с запятой. Иногда полезно рассматривать отсутствие операторов как существующий пустой оператор. Вот пример:

```
if (a > b) ;
        else
        {
           int temp = a; a = b; b = temp;
        }
```

Это корректно работающий пример. А вот типичная для новичков ошибка:

```
for(int i = 0; i < n; i++);
{
        …
}
```

Здесь телом цикла является пустой оператор.

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? Логические выражения if заключаются в круглые скобки и имеют значения true или false. Каждый из операторов может быть блоком, в частности, if-оператором. Поэтому возможна и такая конструкция:

```
if(выражение1) if(выражение2) if(выражение3) …
```

Ветви else 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), 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*, как следствие, его можно было бы не писать, 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.
Содержательный пример применения оператора switch подробно рассмотрен в лекции 2. Рассмотрим еще один показательный пример, в котором вычисляется арифметическое выражение с двумя аргументами.

```
/// <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, double arg1, double arg2,
   ref double 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;
         break;
   }
}//ExprResult
```

Обратите внимание: знак операции над аргументами можно задавать разными способами, что демонстрирует возможность задания списка константных выражений в ветвях оператора switch.

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 применение оператора goto. Оператор goto может использоваться в операторе switch, о чем шла речь выше.
"О вреде оператора goto" и о том, как можно обойтись без него, писал еще Эдсгар 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. Операторы break и continue специально предназначены для этих целей.

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. Классическим примером является "поиск по образцу", когда в массиве отыскивается элемент, соответствующий образцу. Понятно, что когда такой элемент найден, поиск можно прекратить. Вот пример метода, реализующего данную стратегию поиска:

```
/// <summary>
 /// Поиск образца в массиве
 /// </summary>
 /// <param name="ar">массив для поиска</param>
 /// <param name="pat">образец поиска</param>
 /// <param name="patIndex">индекс найденного элемента</param>
 /// <returns>
 /// true, если найден элемент, совпадающий с образцом
 /// false - в противном случае
 /// </returns>
 public bool SearchPattern(int[] ar, int pat, out int patIndex)
 {
    int n = ar.Length;
    patIndex = -1;
    bool found = false;
    for (int i = 0; i < n; i++)
      if (ar[i] == pat)
  {
     found = true;
     patIndex = i;
     break;
  }
    return found;
 }
```

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 еще одну 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 IsPalindrom(string str)
  {
     for (int i = 0, j = str.Length - 1; i < j; i++, j--)
        if (str[i] != str[j]) return (false);
     return (true);
  }//IsPalindrom
```

В цикле for разрешается опускать некоторые части заголовка цикла. Конструкция этого оператора, в которой все части заголовка опущены, задает бесконечный цикл:

```
for(;;) {…}
```

Эта конструкция позволяет организовать цикл с проверкой выхода в теле цикла. Ярые почитатели оператора for, к которым чаще всего относятся любители стиля языка C++, часто пользуются такой конструкцией. В таких ситуациях лучше использовать цикл типа while.

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 текст");
      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, установленном 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 = min;
            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
```

Специальные операторы
Операторы языка C#, рассмотренные выше, имеют аналоги практически во всех языках программирования. Поговорим теперь о более экзотических операторах, не столь часто появляющихся в других языках программирования.
Оператор yield
При рассмотрении оператора цикла foreach говорилось, что он применим к классам, содержащим контейнеры с элементами, и цикл foreach перебирает элементы контейнера в некотором заданном порядке. Для того чтобы класс представлял контейнер, он должен быть перечислимым и быть наследником интерфейса IEnumerable. Есть другая возможность - класс может иметь один или несколько методов, называемых итераторами, создающих контейнеры и возвращающих результат интерфейсного класса IEnumerable.
Оператор yield используется в итераторах и позволяет заполнять контейнер элементами. Его синтаксис:
yield return <выражение>;

Каждое выполнение оператора yield добавляет новый элемент в контейнер. Подробно рассмотрение этого оператора будет дано в главе, посвященной интерфейсам. Сейчас же ограничусь одним примером. В класс Testing, используемый в нашем проекте, добавим итератор, создающий коллекцию:

```
 /// <summary>
  /// Итератор, создающий коллекцию цветов
  /// </summary>
  /// <returns>коллекцию </returns>
  public System.Collections.IEnumerable Rainbow()
  {
     yield return "red";
     yield return "orange";
     yield return "yellow";
     yield return "green";
```

```
    yield return "blue";
    yield return "violet";
  }
```

Клиенты этого класса могут работать с этой коллекцией, например, так:

```
string colors = "";
        foreach(string s in tst.Rainbow())
           colors += s + "-";
```

Здесь tst - объект класса Testing, а переменная s в цикле foreach получит значения всех цветов, помещенных в контейнер оператором yield. Следует заметить, что реально никакие контейнеры не создаются, а цикл foreach на каждом шаге вызывает итератор и создает новый элемент. Именно поэтому цикл foreach работает только на чтение элементов и не работает на запись.

Операторы try, catch, finally

Об охраняемых блоках, блоках, перехватывающих исключения, задаваемых операторами try, catch, finally, мы уже говорили в лекции 2 и приводили достаточное число примеров. Тема организации обработки исключительных ситуаций и соответствующие операторы будут подробно рассматриваться в отдельной главе, а примеры их использования будут появляться повсеместно.

Операторы checked и unchecked

В лекции 3 рассматривались проверяемые и непроверяемые выражения и блоки. Блоки с предшествующими словами checked и unchecked и являются соответствующими операторами. Полагаю, что приведенных ранее сведений достаточно для понимания синтаксиса, семантики и области применения этих операторов.

Оператор fixed

Оператор fixed используется в небезопасных (unsafe) блоках, позволяя фиксировать в памяти расположение переменных, на которые ссылаются указатели. Такая фиксация не позволяет сборщику мусора перемещать зафиксированные переменные. Поскольку в данном курсе работа с указателями, прямая адресация и другие опасные средства, характерные для языка C++, не рассматриваются, то оператор fixed рассматриваться не будет и не будет встречаться в примерах этого курса.

Оператор lock

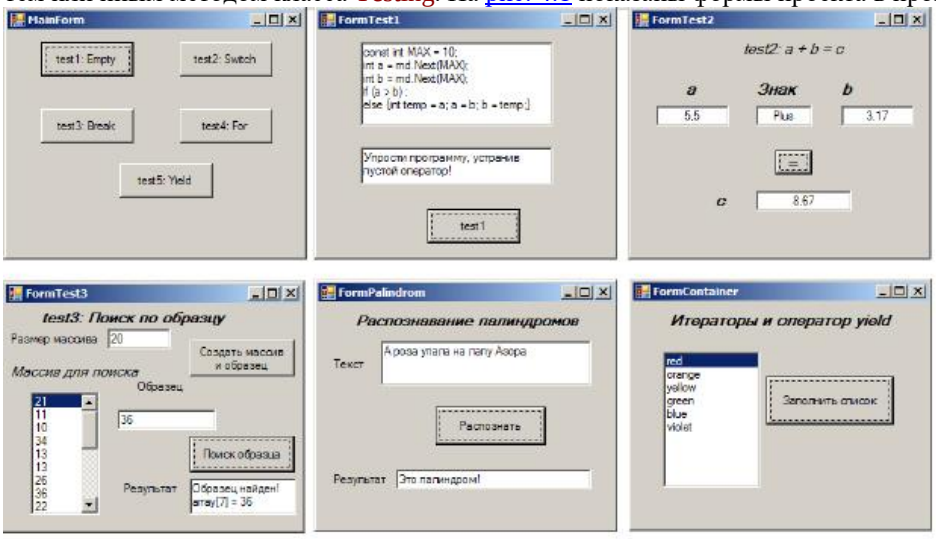
Оператором lock, блоком lock, критической секцией кода, закрытым блоком называют блок с предшествующим ключевым словом lock

```
lock {…}
```

Этот оператор используется при работе с несколькими потоками. Он позволяет закрыть блок кода для одновременной работы нескольких потоков. Ни один поток не сможет войти в закрытый блок, если другой поток уже выполняет код критической секции. Остальные потоки будут ждать, пока закрытый блок не будет освобожден. Потокам будет посвящен отдельный раздел в этом курсе, где будет подробно рассмотрен и оператор lock.

Проект Statements

Как обычно, для этой главы построено решение с именем Ch4, содержащее Windows-проект с именем Statements. В проекте создан класс Testing, методы которого позволяют тестировать работу операторов языка C#. Эти методы используются в примерах, приведенных в этой главе. Архитектурно проект представляет Windows-приложение с главной кнопочной формой. Каждый из интерфейсных классов, включенных в проект, обеспечивает пользовательский интерфейс для работы с тем или иным методом класса Testing. На рис. 4.1 показаны формы проекта в процессе работы с ними.



увеличить изображение

Рис. 4.1. Формы проекта Statements в процессе работы

Не буду приводить полного описания реализации этого проекта. Полагаю, что заинтересованный читатель сможет по рисунку и приведенным методам класса Testing самостоятельно построить аналог этого проекта.

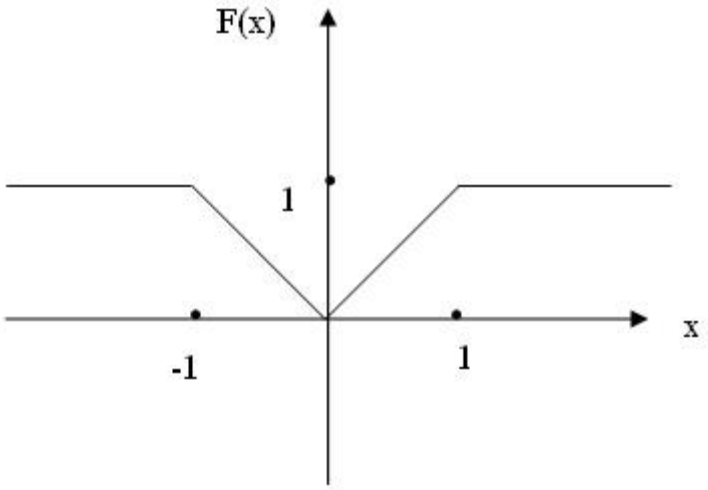
Задачи

Альтернатива и разбор случаев

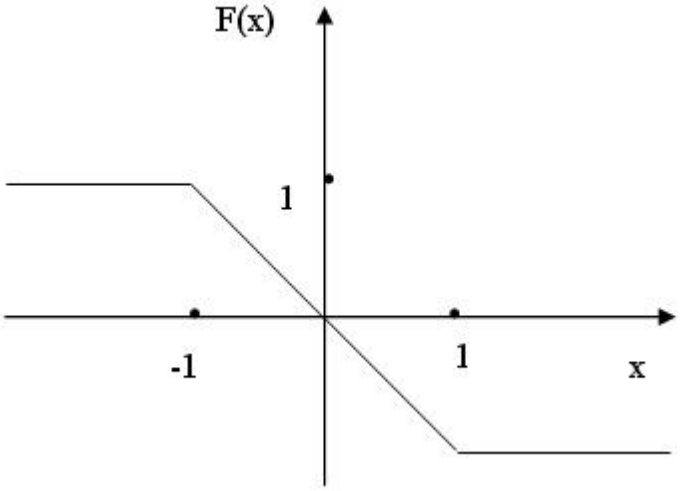
- 1. Постройте консольное и Windows-приложение, которое по заданным коэффициентам a, b, c находит корни квадратного уравнения.
- 2. Постройте консольное и Windows-приложение, которое по заданному значению аргумента x вычисляет значение функции y=F(x), где функция F(x) задана соотношением:

$$F(x) = \begin{cases} 1 & \text{если } x > 0 \\ 0 & \text{если } x = 0 \\ -1 & \text{если } x < 0 \end{cases}$$

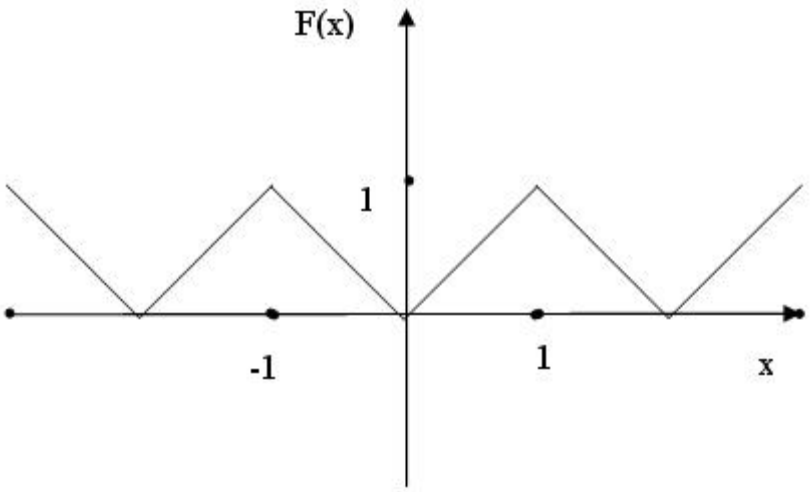
- 3. Постройте консольное и Windows-приложение, которое по заданному значению аргумента x вычисляет значение функции y=F(x), где функция F(x) задана графиком:



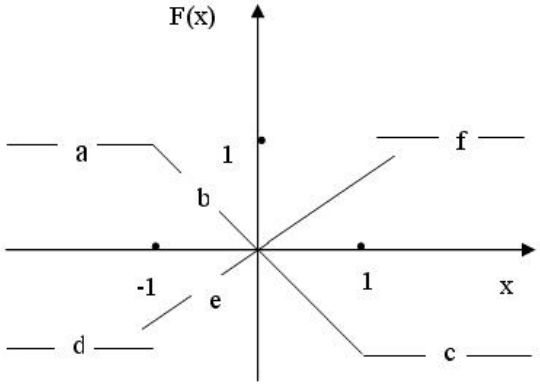
4. Постройте консольное и Windows-приложение, которое по заданному значению аргумента x вычисляет значение функции y=F(x), где функция F(x) задана графиком:



- 5. Постройте консольное и Windows-приложение, которое по заданному значению аргумента x вычисляет значение функции y=F(x), где периодическая функция F(x) задана графиком:



6. Постройте консольное и Windows-приложение, которое по заданным координатам x и y определяет, принадлежит ли точка (x, y) одной из 6 дорог (a, b, c, d, e, f), показанных на графике. Если точка принадлежит дороге, то укажите, какой именно дороге, если принадлежит двум дорогам, то и этот факт следует отразить в результирующем сообщении.



Вычисление сумм, произведений и рекуррентные соотношения

- 25. Дано натуральное число nmax и вещественное число b. Найти, если оно существует, такое наименьшее n, меньшее nmax, что:

$$\sum_{k=1}^{n} \frac{1}{k} > b$$

Если сумма nmax членов гармонического ряда меньше b, то необходимо выдать соответствующее сообщение.

- 26. Дано натуральное число n. Вычислить сумму первых n членов ряда:

$$\sum_{k=1}^{n} \frac{1}{k}$$

При суммировании исключается каждый третий член.

- 27. Дано натуральное число n. Вычислить сумму первых 2n членов ряда:

$$\sum_{k=1}^{2n} \frac{(-1)^{k+1}}{k}$$

Вычислить эту сумму четырьмя разными способами: последовательно слева направо, последовательно справа налево, слева направо, вычисляя вначале положительные члены ряда, затем отрицательные, справа налево, вычисляя вначале положительные члены ряда, затем отрицательные. Сравните результаты вычислений. Чем объясняется различие в последних цифрах при больших n? Как влияет на результат использование типов float или double для переменных, задающих суммы и текущий член при суммировании?

Лекция 6

Тема: Процедуры, функции и методы класса

Процедуры и функции - функциональные модули

Первыми формами модульности, появившимися в языках программирования, были **процедуры** и **функции**. Поскольку функции в математике использовались издавна, появление их в языках программирования было совершенно естественным. Уже с первых шагов программирования процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, - задачу повторного использования программного кода. Один раз написанную функцию можно многократно вызывать в программе с разными значениями параметров, передаваемых функции в момент вызова. Встроенные в язык функции позволяли существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из языка программирования.

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 имеющий важное значение. Вся программа рассматривалась как некоторая главная функция. В процессе проектирования программы происходила декомпозиция главной функции на подфункции, решающие частные задачи. Этот процесс декомпозиции продолжался до тех пор, пока не приходили к достаточно простым функциям, реализация которых не требовала декомпозиции и могла быть описана базовыми конструкциями языка программирования.

С появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ОО-языков, к которым относится и язык C#, роль архитектурного модуля играет класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеет содержательную начинку, задавая некоторую абстракцию данных. Процедуры и функции связываются теперь с классом, они обеспечивают требуемую функциональность класса и называются методами класса. Поскольку класс в объектно-ориентированном программировании рассматривается как некоторый тип данных, главную роль в классе начинают играть его данные - поля класса, задающие свойства объектов класса. Методы класса "служат" данным, занимаясь их обработкой. Помните: 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 - method, procedure, function, но сами понятия, конечно же, присутствуют. 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, методов этих классов совершенно необходимо для практического программирования на C#, использование всей его мощи.

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, который указывает на отсутствие результата, возвращаемого при вызове процедуры;
- 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 - 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 одного 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 может появляться в объявлении лишь 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 Cube()
  /// первый аргумент - результат
  /// представляет сумму кубов
  /// произвольного числа оставшихся аргументов
  /// Аргументы могут быть разного типа.
```

```
/// </summary>
void Cube(out long p2, int p1)
{
   p2 = (long)Math.Pow(p1, 3);
   Console.WriteLine("Метод A-1");
}
void Cube(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 Cube(out double p2, double p1)
{
   p2 = Math.Pow(p1, 3);
   Console.WriteLine("Метод A-3");
}
void Cube(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 Cube 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 Cube 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 является выходным и сопровождается ключевым словом out, в теле метода этому аргументу присваивается значение. Аргумент функции F является обновляемым, 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 методов этому 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, возвращающий значение функции в форме return <выражение>.

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.

Знание 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 тела блока. 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. Любое выражение с побочным эффектом может выступать 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 не совместимо 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, синтаксически является идентификатором - именем аргумента. Фактический аргумент представляет собой "выражение", значительно более сложную синтаксическую конструкцию. 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 T - тип формального аргумента, то выражение, задающее фактический аргумент, должно быть согласовано по типу с типом T. Это означает, что вычисленный тип выражения совпадает с типом T, или допускает неявное преобразование к типу T, или является потомком типа T (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, так что 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, фактические аргументы которой содержат вызовы функций, фактические аргументы которых содержат вызовы функций, и так далее и так далее.

Для простоты понимания семантики вызова можно полагать, что в точке вызова создается блок, соответствующий телу метода (реально все происходит значительно эффективнее). В этом блоке происходит замена имен формальных аргументов фактическими аргументами. Для выходных (ref и out) аргументов, для которых фактические аргументы также являются именами, эта замена или передача аргументов происходит по ссылке, означая замену формального аргумента ссылкой на реально существующий объект, заданный фактическим аргументом. Чуть более сложную семантику имеет вызов по значению, применяемый к формальным аргументам, объявленным 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.

Заметьте, семантика замены формальных аргументов фактическими эквивалентна семантике присваивания, подробно рассмотренной в предыдущих главах.

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, 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 свое значение, бывшее у них до вызова. Отсюда вывод: все выходные аргументы значимых типов, значения которых предполагается изменить в процессе работы, должны иметь ключевое слово ref или out.

Говоря о семантике вызова по ссылке и по значению, следует сделать 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, все поля этого объекта могут в методе меняться 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 при этом, 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 столь 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 метод.

Все дело в том, что методы класса - это не просто набор процедур и функций, это методы, обслуживающие данные класса. Все поля класса доступны любому методу по определению. Нужно четко понимать, что в момент выполнения программной системы работа идет с объектами - экземплярами класса. Из полей соответствующего объекта - цели вызова - извлекается информация, нужная методу в процессе работы, а работа метода чаще всего сводится к обновлению значений полей этого объекта. Поэтому понятно, 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 неизбежны дополнительные расходы памяти. Они могут быть значительными, когда создается большое число объектов, ведь поле должен иметь каждый объект. Если предпочесть функцию, то это потребует временных затрат на вычисление значения, они могут быть значительными, если функция вызывается многократно, а ее вычисление требует значительно больших затрат 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 он к полю или к методу. Такой синтаксис принят, например, в языке Eiffel. Преимущество такого 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().

Методы. Перегрузка

Должно ли быть уникальным имя метода в классе? Нет, это 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.

Пример этот, может быть, не совсем удачен, поскольку при перегрузке сигнатуры реализаций должны отличаться, а для вычисления площади обычно требуются три аргумента, вообще говоря, одного типа. Так что в этом случае придется использовать искусственные приемы, например, объявляя стороны треугольника типа float, а углы типа - double. Другая возможность - иметь набор методов с разными именами, но с одинаковой сигнатурой.

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.

Перегрузка требует уточнения семантики вызова метода. Когда встречается вызов не перегруженного 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 Cube, отличающиеся сигнатурой. Методы отличаются типами аргументов и ключевым словом params. Когда вызывается метод Cube с двумя аргументами, в зависимости от типа будет вызываться реализация, не содержащая аргумент с модификатором params. Когда же число аргументов больше двух, 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.

Насколько полезна перегрузка методов? Здесь нет экономии кода, поскольку каждую реализацию нужно задавать явно; нет выигрыша по времени, скорее требуются определенные затраты на поиск подходящей реализации, который может приводить к конфликтам, к счастью, обнаруживаемым на этапе компиляции. В нашем примере вполне разумно было бы отказаться от перегрузки и иметь четыре метода с разными именами, осознанно вызывая метод, применимый к конкретным данным.

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 примерно 16 реализаций в зависимости от типа источника. Согласитесь, что неразумно было бы иметь в классе Convert 256 методов вместо 16 перегруженных методов. Впрочем, так же неразумно было бы иметь один перегруженный метод, имеющий 256 реализаций. Перегрузка - это инструмент, которым следует пользоваться с осторожностью и обоснованно.

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

```
/// <summary>
  /// Тестирование перегруженных методов Cube()
  /// </summary>
  public void TestLoadMethods()
  {
     long u = 0; double v = 0;
     Cube(out u, 7); Cube(out v, 7.5);
```

```
    Console.WriteLine("u= {0}, v= {1}", u, v);
    Cube(out v, 7);
    Console.WriteLine("v = {0}", v);
    Cube(out u, 7, 11, 13);
    Cube(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. 5.3 показаны результаты этого тестирования.

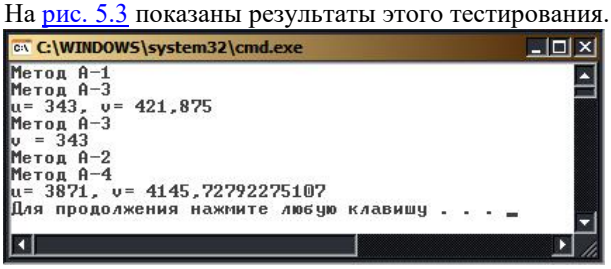


Рис. 5.3. Тестирование перегрузки методов

Лекция 7

Тема: Массивы

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**, а - **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# снято существенное ограничение языка C++ на статичность массивов. Массивы в языке C# являются динамическими. Как следствие этого, напомню, массивы относятся к ссылочным типам, память им отводится динамически в "куче". К сожалению, не снято ограничение 0-базируемости, означающее, что нижняя граница массивов C# фиксирована и равна нулю. Было бы гораздо удобнее во многих задачах иметь возможность работать с массивами, у которых нижняя граница изменения индекса 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 типа, так что запись T[] следует понимать как тип, задающий **одномерный массив с элементами типа** T.
Что же касается границ изменения индексов, то эта характеристика не является принадлежностью типа, она является характеристикой переменных данного типа - 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. 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 синтаксису, элементы константного массива необходимо 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, его элементы получают 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, ей 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 отражается на другом массиве.

- 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:

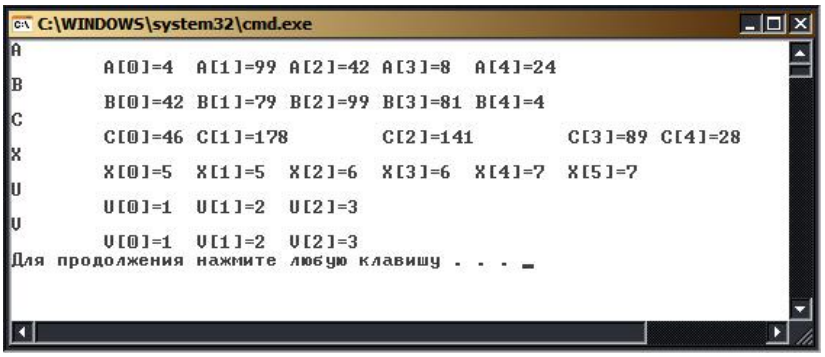
```
public static void CreateOneDimAr(int[] A)
 {
            for(int i = 0; i<A.GetLength(0);i++)
  A[i] = rnd.Next(1,100);
 }//CreateOneDimAr
```

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. 6.1 показан консольный вывод результатов работы процедуры TestDeclarations:

Рис. 6.1. 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. Действительно 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, поскольку 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.
Размерность массива это характеристика типа. Как синтаксически при объявлении типа массива указать его размерность? 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
```

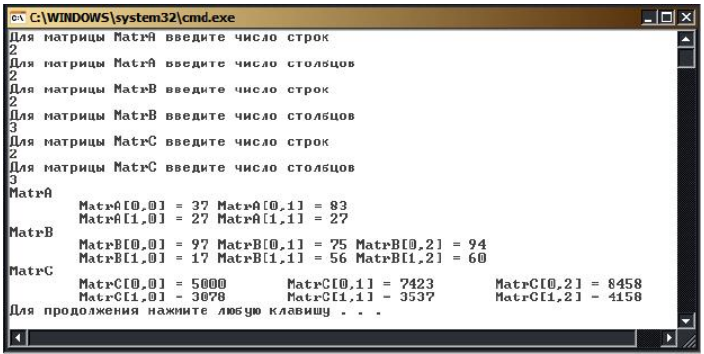
Давайте рассмотрим классическую задачу умножения прямоугольных матриц. Нам понадобится три динамических массива для представления матриц и три процедуры, одна из которых будет заполнять исходные матрицы случайными числами, другая - выполнять умножение матриц, третья - печатать сами матрицы. Вот тестовый 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. 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.
Взгляните, как выглядят результаты консольного вывода на данном этапе работы.



увеличить изображение
Рис. 6.2. 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, его элементы 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 к-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]
 };
```

Но 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":

```
/// <summary>
/// массив массивов -"Отцы и дети"
/// </summary>
public void GenTree()
 {
   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"};
   Arrs.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:

```
/// <summary>
    /// Печать дерева "Отцы и дети",
    /// заданного массивами Fathers и Children
    /// </summary>
    /// <param name="Fathers">массив отцов</param>
    /// <param name="Children"> массив массивов детей</param>
    public static 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.

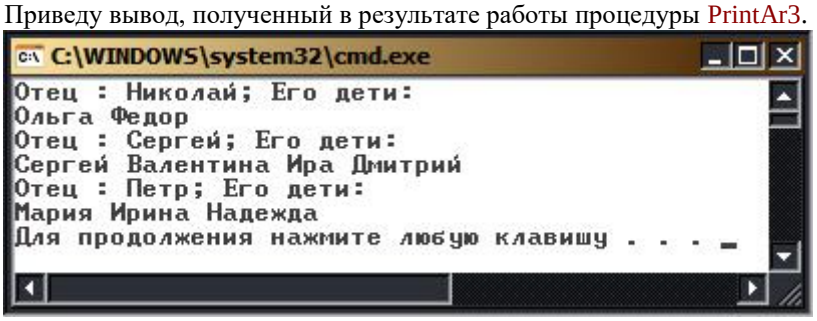


Рис. 6.3. Дерево "Отцы и дети"

Процедуры и массивы

В наших примерах массивы неоднократно передавались процедурам в качестве входных аргументов и возвращались в качестве результатов. 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.
- Функция может возвращать массив в качестве результата.

Алгоритмы и задачи

Алгоритмы и задачи, рассматриваемые в этой главе, являются частью фундамента, на котором строится образование программиста. Нет ни одной проблемной области, в задачах которой не требовались бы массивы. Поэтому задачи, требующие использования массивов, появлялись уже в предыдущих главах, появятся они и в последующих. Но здесь мы будем заниматься ими целенаправленно.
Последовательность элементов - $a_1, a_2, \ldots a_n$ - одна из любимых структур в математике. Последовательность можно рассматривать как функцию $a(i)$, которая по заданному значению индекса элемента возвращает его значение. Эта функция задает отображение $integer -> T$, где T - это тип элементов последовательности. В программировании последовательности это одномерные массивы, но от этого они не перестают быть менее любимыми.
Определение. Массив - это упорядоченная последовательность элементов одного типа. Порядок элементов задается с помощью индексов.

В отличие от математики, где последовательность может быть бесконечной, массивы всегда имеют конечное число элементов. Для программистов важно то, как массивы хранятся в памяти. Массивы занимают непрерывную область памяти, поэтому, зная адрес начального элемента массива, зная, сколько байтов памяти требуется для хранения одного элемента, и зная индекс (индексы) некоторого элемента, нетрудно вычислить его адрес, а значит, и хранимое по этому адресу значение элемента. На этом основана адресная арифметика в языках C и C++, где адрес элемента a(i) задается адресным выражением a+i, в котором имя массива a воспринимается как адрес первого элемента. При вычислении адреса i-го элемента индекс i умножается на длину слова, требуемого для хранения элементов типа T. Адресная арифметика использует 0-базируемость элементов массива, полагая индекс первого элемента равным нулю, поскольку первому элементу соответствует адресное выражение a+0.

Язык C# сохранил 0-базируемость массивов. Индексы элементов массива в языке C# изменяются в плотном интервале значений от нижней границы, всегда равной 0, до верхней границы, которая задана динамически вычисляемым выражением, возможно, зависящим от переменных. Массивы C# являются 0-базируемыми динамическими массивами. Это важно понимать с самого начала.

Не менее важно понимать и то, что массивы C# относятся к ссылочным типам.

Ввод-вывод массивов

Как у массивов появляются значения, как они изменяются? Возможны три основных способа:

- вычисление значений в программе;
- значения вводит пользователь;
- связывание с источником данных.

В задачах этого раздела ограничимся пока рассмотрением первых двух способов. Первый способ более или менее понятен. Простые примеры его применения приводились неоднократно. Стоит только отметить, что в классе, работающем с массивами, всегда полезно иметь метод FillArray, позволяющий заполнять массив случайными числами. В примерах использование возможностей класса Random для моделирования элементов массива встречалось неоднократно.

Приведу некоторые рекомендации по вводу и выводу массивов, ориентированные на работу с конечным пользователем.

Для консольных приложений ввод массива обычно проходит несколько этапов:

- ввод размеров массива;
- создание массива;
- организация цикла по числу элементов массива, в теле которого выполняется:
  - приглашение к вводу очередного элемента;
  - ввод элемента;
  - проверка корректности введенного значения.

Вначале у пользователя запрашиваются размеры массива, затем создается массив заданного размера. В цикле по числу элементов организуется ввод значений. Вводу каждого значения предшествует приглашение к вводу с указанием типа вводимого значения, а при необходимости - и диапазона, в котором должно находиться требуемое значение. Поскольку ввод значений - это ответственная операция, а на пользователя никогда нельзя положиться, после ввода часто организуется проверка корректности введенного значения. При некорректном задании значения элемента ввод повторяется, пока не будет достигнут желаемый результат.

При выводе массива на консоль обычно вначале выводится имя массива, а затем его элементы в виде пары: <имя> = <значение> (например, f[5] = 77,7). Задача осложняется для многомерных массивов, когда пользователю важно видеть не только значения, но и структуру массива, располагая строку массива в строке экрана.

Как организовать контроль ввода? Наиболее разумно использовать для этих целей конструкцию охраняемых блоков - try - catch блоков. Это общий подход, когда все опасные действия, связанные с работой пользователя, внешних устройств, внешних источников данных, размещаются в охраняемых блоках.

Как правило, для ввода-вывода массивов пишутся специальные процедуры, вызываемые в нужный момент.

Ввод-вывод массивов в Windows-приложениях

Приложения Windows позволяют построить дружелюбный интерфейс пользователя, облегчающий работу по вводу и выводу массивов. И здесь, когда данные задаются пользователем, заполнение массива проходит через те же этапы, что рассматривались для консольных приложений. Но выглядит все это более красиво, наглядно и понятно. Пример подобного интерфейса, обеспечивающего работу по вводу и выводу одномерного массива, показан на рис. 6.4.

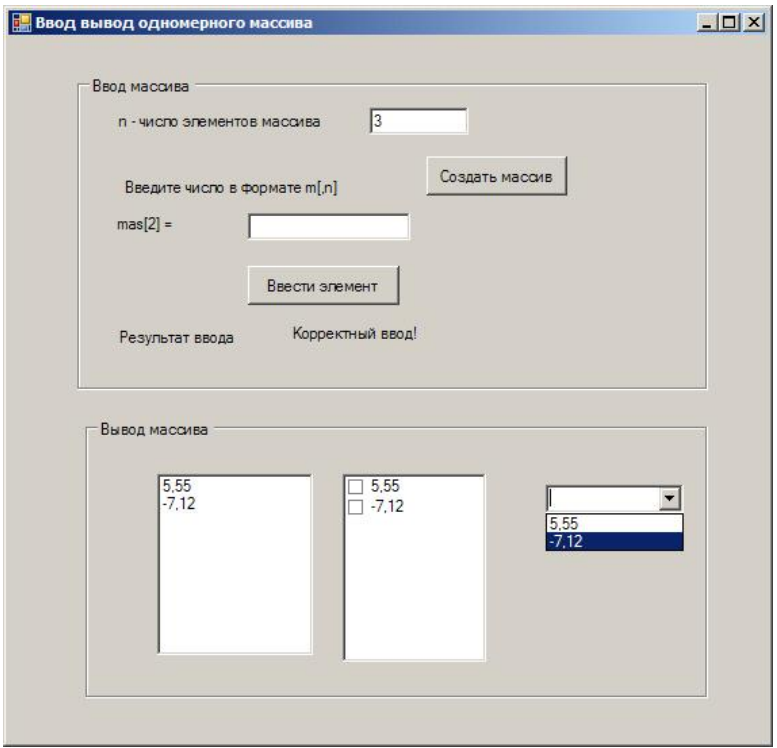


Рис. 6.4. Форма для ввода-вывода одномерного массива

Пользователь вводит в текстовое окно число элементов массива и нажимает командную кнопку "Создать массив", обработчик которой создает массив заданной размерности, если корректно задан размер массива, в противном случае выдает сообщение об ошибке и ждет корректного ввода.

В случае успешного создания массива пользователь может переходить к следующему этапу - вводу элементов массива. Очередной элемент массива

вводится в текстовое окно, а обработчик командной кнопки "Ввести элемент" обеспечивает передачу значения в массив. Корректность ввода контролируется и на этом этапе, проверяя значение введенного элемента и выводя в специальное окно сообщение в случае его некорректности, добиваясь, в конечном итоге, получения от пользователя корректного ввода.
Для облегчения работы пользователя выводится подсказка, какой именно элемент должен вводить пользователь. После того, как все элементы массива введены, окно ввода становится недоступным для ввода элементов. Интерфейс формы позволяет многократно создавать новый массив, повторяя весь процесс.
На рис. 6.4 форма разделена на две части - для ввода и вывода массива. Крайне важно уметь организовать ввод массива, принимая данные от пользователя. Не менее важно уметь отображать существующий массив в форме, удобной для восприятия пользователя. На рисунке показаны три различных элемента управления, пригодные для этих целей, - ListBox, CheckedListBox и ComboBox. Как только вводится очередной элемент, он немедленно отображается во всех трех списках. В реальности отображать массив в трех списках, конечно, не нужно, это сделано только в целях демонстрации возможностей различных элементов управления. Для целей вывода подходит любой из них, выбор зависит от контекста и предпочтений пользователя. Элемент ComboBox имеет дополнительное текстовое окно, в которое пользователь может вводить значение. Элемент CheckedListBox обладает дополнительными свойствами в сравнении с элементом ListBox, позволяя отмечать некоторые элементы списка (массива). Отмеченные пользователем элементы составляют специальную коллекцию. Эта коллекция доступна, с ней можно работать, что иногда весьма полезно. Чаще всего для вывода массива используется элемент ListBox.
Посмотрим, как это все организовано программно. Начну с полей формы OneDimArrayForm, показанной на рис. 6.4:

```
//fields
  int n = 0;
  double[] mas;
  int currentindex = 0;
  double ditem = 0;
  const string SIZE = "Корректно задайте размер массива!";
  const string INVITE = "Введите число в формате m[,n]";
  const string EMPTY =  "Массив пуст!";
  const string ITEMB =  "mas[";
  const string ITEME =  "] = ";
  const string FULL =   "Ввод недоступен!";
  const string OK =     "Корректный ввод!";
  const string ERR =    "Ошибка ввода числа! Повторите ввод!";
```

Полями этого класса является одномерный массив, его размер, текущий индекс и константы, используемые в процессе диалога с пользователем. Обработчик события Click командной кнопки, отвечающей за создание массива, имеет вид:

```
private void buttonCreateArray_Click(object sender, EventArgs e)
  {
     try
     {
    n = Convert.ToInt32(textBoxN.Text);
    mas = new double[n];
    labelInvite.Text = INVITE;
    labelItem.Text = ITEMB + "0" + ITEME;
    labelResult.Text = EMPTY;
    textBoxItem.ReadOnly = false;
    listBox1.Items.Clear();
    comboBox1.Items.Clear();
    checkedListBox1.Items.Clear();
    comboBox1.Items.Clear();
    currentindex = 0;
     }
     catch (Exception)
     {
    labelResult.Text = SIZE;
     }
  }
```

Первым делом принимается размер массива, введенный пользователем. Преобразование к типу int введенного значения помещено в охраняемый блок, поэтому ошибки некорректного ввода будут перехвачены с выдачей соответствующего сообщения. Если же массив успешно создан, то инициализируются начальными значениями все элементы интерфейса, участвующие в вводе элементов массива. Рассмотрим, как устроен ввод элементов.

```
private void buttonAddItem_Click(object sender, EventArgs e)
  {
     //Заполнение массива элементами
     if (GetItem())
     {
    mas[currentindex] = ditem;
```

```
    listBox1.Items.Add(mas[currentindex]);
    checkedListBox1.Items.Add(mas[currentindex]);
    comboBox1.Items.Add(mas[currentindex]);
    currentindex++;
    labelItem.Text = ITEMB + currentindex + ITEME;
    textBoxItem.Text = "";
    labelResult.Text = OK;
    if (currentindex == n)
    {
       labelInvite.Text = "";
       labelItem.Text = "";
       labelResult.Text = FULL;
       textBoxItem.Text = "";
       textBoxItem.ReadOnly = true;
    }
   }
}
```

Функция GetItem вводит значение очередного элемента. Если пользователь корректно задал его значение, то элемент добавляется в массив, а заодно и в списки, отображающие текущее состояние массива. Создается подсказка для ввода следующего элемента массива, а если массив полностью определен, то форма переходит в состояние окончания ввода.

```
/// <summary>
  /// Ввод с контролем текущего элемента массива
  /// </summary>
  /// <returns>true в случае корректного ввода значения</returns>
  bool GetItem()
  {
     string item = textBoxItem.Text;
     bool res = false;
     if (item == "")
    labelResult.Text = INVITE;
     else
     {
    try
     {
        ditem = Convert.ToDouble(item);
        res = true;
     }
    catch(Exception)
     {
        labelResult.Text = ERR;
     }
     }
     return res;
  }
```

Форму OneDimArrayForm можно рассматривать как некоторый шаблон, полезный при организации ввода и вывода одномерных массивов.

Организация ввода-вывода двумерных массивов

Ввод двумерного массива немногим отличается от ввода одномерного массива. Сложнее обстоит дело с выводом двумерного массива, если при выводе пытаться отобразить структуру массива. К сожалению, все три элемента управления, хорошо справляющиеся с отображением одномерного массива, плохо приспособлены для показа структуры двумерного массива. Хотя у того же элемента ListBox есть свойство MultiColumn, включение которого позволяет показывать массив в виде строк и столбцов, но это не вполне то, что нужно для наших целей - отображения структуры двумерного массива. Хотелось бы, чтобы элемент имел такие свойства, как Rows и Columns, а их у элемента ListBox нет. Нет их и у элементов ComboBox и CheckedListBox. Приходится обходиться тем, что есть. На рис. 6.5 показан пример формы, поддерживающей работу по вводу и выводу двумерного массива.

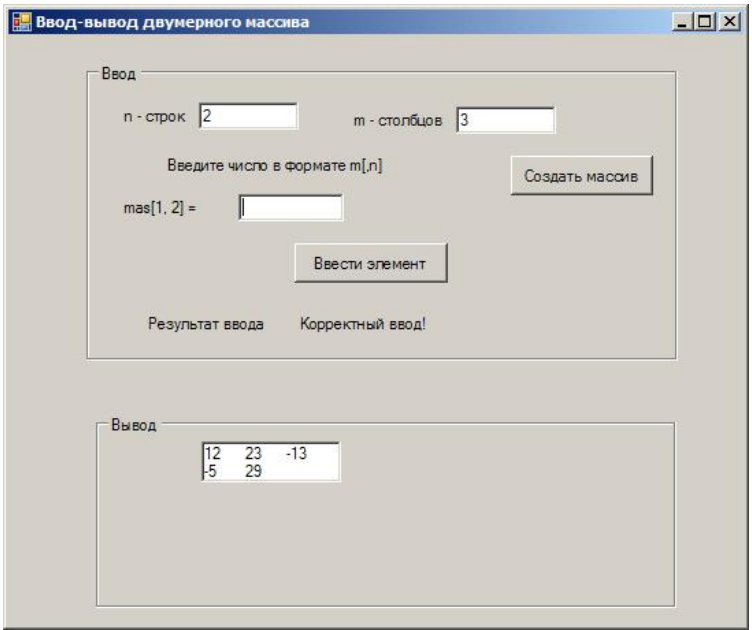


Рис. 6.5. Форма, поддерживающая ввод и вывод двумерного массива

Интерфейс формы схож с тем, что использовался для организации работы с одномерным массивом. Схожа и программная организация ввода-вывода элементов массива. Возможно отображать двумерный массив в элементе управления ListBox так, чтобы сохранялась структура строк и столбцов массива. Этого можно добиться за счет программной настройки размеров элемента управления ListBox:

```
listBox1.Height = n * HEIGHT_LINE;
        listBox1.Width = m * 2 * HEIGHT_LINE;
```

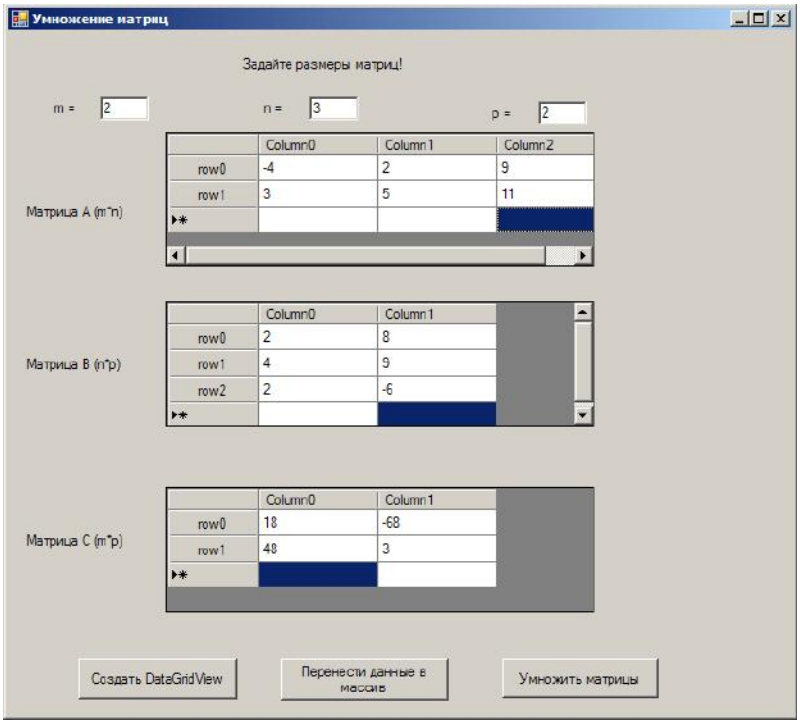
Константа HEIGHT_LINE задает высоту строки в списке. Вначале водятся элементы первого столбца; когда весь столбец введен, автоматически следующее вводимое значение будет отображаться в первой строке в следующем столбце.

В общей ситуации, когда значения, вводимые пользователем, могут колебаться в широком диапазоне, трудно гарантировать отображение структуры двумерного массива. Однако ситуация не безнадежна. Есть и другие, более мощные и более подходящие для наших целей элементы управления. Расскажем подробнее о элементе DataGridView.

Элемент управления DataGridView и отображение массивов

Элемент управления DataGridView является последней новинкой в серии табличных элементов DataGrid, позволяющих отображать таблицы. Главное назначение этих элементов - связывание с таблицами внешних источников данных, прежде всего с таблицами баз данных. Мы же сейчас рассмотрим другое его применение - в интерфейсе, позволяющем пользователю вводить и отображать матрицы - двумерные массивы.

Рассмотрим классическую задачу умножения прямоугольных матриц C=A*B. Построим интерфейс, позволяющий пользователю задавать размеры перемножаемых матриц, вводить данные для исходных матриц A и B, перемножать матрицы и видеть результаты этой операции. На рис. 6.6 показан возможный вид формы, поддерживающей работу пользователя. Форма показана в тот момент, когда пользователь уже задал размеры и значения исходных матриц, выполнил умножение матриц и получил результат.



увеличить изображение

Рис. 6.6. Форма с элементами DataGridView, поддерживающая работу с матрицами

На форме расположены три текстовых окна для задания размеров матриц, три элемента DataGridView для отображения матриц, три командные кнопки для выполнения операций, доступных пользователю. Кроме того, на форме присутствуют 9 меток (элементов управления label), семь из которых видимы на рис. 6.6. В них отображается информация, связанная с формой и отдельными элементами управления. Текст у невидимых на рисунке меток появляется тогда, когда обнаруживается, что пользователь некорректно задал значение какого-либо элемента исходных матриц.

А теперь перейдем к описанию того, как этот интерфейс реализован. В классе Form2, которому принадлежит наша форма, зададим поля, определяющие размеры матриц, и сами матрицы:

```
//поля класса Form
      int m, n, p;    //размеры матриц
      double[,] A, B, C;  //сами матрицы
```

Рассмотрим теперь, как выглядит обработчик события "Click" командной кнопки "Создать DataGridView". Предполагается, что пользователь разумен и, прежде чем нажать эту кнопку, задает размеры матриц в соответствующих текстовых окнах. Напомню, что при перемножении матриц размеры матриц должны быть согласованы - число столбцов первого сомножителя должно совпадать с числом строк второго сомножителя, а размеры результирующей матрицы определяются размерами сомножителей. Поэтому для трех матриц в данном случае достаточно задать не шесть, а три параметра, определяющие размеры.

Обработчик события выполняет три задачи - создает сами матрицы, осуществляет чистку элементов управления DataGridView, удаляя предыдущее состояние, затем добавляет столбцы и строки в эти элементы в полном соответствии с заданными размерами матриц. Вот текст обработчика:

```
private void button1_Click(object sender, EventArgs e)
  {
    //создание матриц
    m = Convert.ToInt32(textBox1.Text);
    n = Convert.ToInt32(textBox2.Text);
    p = Convert.ToInt32(textBox3.Text);
    A = new double[m, n];
    B = new double[n, p];
```

```
    C = new double[m, p];
   //Чистка DGView, если они не пусты
    int k =0;
    k = dataGridView1.ColumnCount;
    if (k != 0)
       for (int i = 0; i < k; i++)
          dataGridView1.Columns.RemoveAt(0);
    dataGridView2.Columns.Clear();
    dataGridView3.Columns.Clear();
    //Заполнение DGView столбцами
    AddColumns(n, dataGridView1);
    AddColumns(p, dataGridView2);
    AddColumns(p, dataGridView3);
    //Заполнение DGView строками
    AddRows(m, dataGridView1);
    AddRows(n, dataGridView2);
    AddRows(m, dataGridView3);
}
```

Прокомментирую этот текст.

- Прием размеров и создание матриц, надеюсь, не требует дополнительных комментариев.
- Чистка предыдущего состояния элементов DataGridView сводится к удалению столбцов. Продемонстрированы два возможных способа выполнения этой операции. Для первого элемента показано, как можно работать с коллекцией столбцов. Организуется цикл по числу столбцов коллекции, и в цикле выполняется метод RemoveAt, аргументом которого является индекс удаляемого столбца. Поскольку после удаления столбца происходит перенумерация столбцов, на каждом шаге цикла удаляется первый столбец, индекс которого всегда равен нулю. Удаление столбцов коллекции можно выполнить одним махом - вызывая метод Clear() коллекции, что и делается для остальных двух элементов DataGridView.
- После чистки предыдущего состояния, можно задать новую конфигурацию элемента, добавив в него вначале нужное количество столбцов, а затем и строк. Эти задачи выполняют специально написанные процедуры AddColumns и AddRows. Вот их текст:

```
private void AddColumns(int n, DataGridView dgw)
  {
    //добавляет n столбцов в элемент управления dgw
    //Заполнение DGView столбцами
    DataGridViewColumn column;
    for (int i = 0; i < n; i++)
    {
       column = new DataGridViewTextBoxColumn();
       column.DataPropertyName = "Column" + i.ToString();
       column.Name = "Column" + i.ToString();
       dgw.Columns.Add(column);
    }
  }
  private void AddRows(int m, DataGridView dgw)
  {
    //добавляет m строк в элемент управления dgw
    //Заполнение DGView строками
    for (int i = 0; i < m; i++)
    {
     dgw.Rows.Add();
       dgw.Rows[i].HeaderCell.Value
          = "row" + i.ToString();
    }
  }
```

Приведу краткий комментарий.

- Создаются столбцы в коллекции Columns по одному. В цикле по числу столбцов матрицы, которую должен отображать элемент управления DataGridView, вызывается метод Add этой коллекции, создающий очередной столбец. Одновременно в этом же цикле создается и имя столбца (свойство Name), отображаемое в форме. Показана возможность формирования еще одного имени (DataPropertyName), используемого при связывании со столбцом таблицы внешнего источника данных. В нашем примере это имя не используется.
- Создав столбцы, нужно создать еще и нужное количество строк у каждого из элементов DataGridView. Делается это аналогичным образом, вызывая метод Add коллекции Rows. Чуть по-другому задаются имена строк - для этого используется специальный объект HeaderCell, имеющийся у каждой строки и задающий ячейку заголовка.
- После того как сформированы строки и столбцы, элемент DataGridView готов к тому, чтобы пользователь или программа вводила значения в ячейки сформированной таблицы.

Рассмотрим теперь, как выглядит обработчик события "Click" следующей командной кнопки "Перенести данные в массив". Предполагается, что пользователь разумен и, прежде чем нажать эту кнопку, задает значения элементов перемножаемых матриц в соответствующих ячейках подготовленных таблиц первых двух элементов DataGridView. Обработчик события выполняет следующие задачи - в цикле читает элементы, записанные пользователем в таблицы DataGridView, проверяет их корректность и в случае успеха переписывает их в матрицы. Вот текст обработчика:

```
private void button2_Click(object sender, EventArgs e)
  {
     string elem = "";
     bool correct = true;
     for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
        {
          try
            {
            elem=dataGridView1.Rows[i].Cells[j].Value.ToString();
            A[i, j] = Convert.ToDouble(elem);
            label8.Text = "";
            }
          catch (Exception any)
            {
             label8.Text = "Значение элемента" +
            "A[" + i.ToString() +", " + j.ToString() + " ]"
            + " не корректно. Повторите ввод!";
  dataGridView1.Rows[i].Cells[j].Selected= true;
   return;
          }
        }
     for (int i = 0; i < n; i++)
        for (int j = 0; j < p; j++)
         {
           do
            {
             correct = true;
             try
             {
          elem =
      dataGridView2.Rows[i].Cells[j].Value.ToString();
       B[i, j] = Convert.ToDouble(elem);
       label9.Text = "";
                }
       catch (Exception any)
       {
         label9.Text = "Значение элемента" +
         "B[" + i.ToString() + ", " + j.ToString() + "]"
         + " не корректно. Повторите ввод!";
         dataGridView2.Rows[i].Cells[j].Selected=true;
     Form3 frm = new Form3();
         frm.label1.Text =
         "B[" + i.ToString() + "," + j.ToString() + "]= ";
         frm.ShowDialog();
       dataGridView2.Rows[i].Cells[j].Value =
    frm.textBox1.Text;
         correct = false;
       }
              } while (!correct);
            }
      }
```

Основная задача переноса данных из таблицы элемента DataGridView в соответствующий массив не вызывает проблем. Конструкция Rows[i].Cells[j] позволяет добраться до нужного элемента таблицы, после чего остается присвоить его значение элементу массива.

- Как всегда при вводе основной проблемой является обеспечение корректности вводимых данных. Схема, рассматриваемая нами ранее, нуждается в корректировке. Дело в том, что ранее проверка корректности осуществлялась сразу же после ввода пользователем значения элемента. Теперь проверка корректности выполняется после того, как пользователь полностью заполнил таблицы, при этом некоторые элементы он мог задать некорректно. Просматривая

таблицу, необходимо обнаружить некорректно заданные значения и предоставить возможность их исправления. В программе предлагаются два различных подхода к решению этой проблемы.

- Первый подход демонстрируется на примере ввода элементов матрицы A. Как обычно, преобразование данных, введенных пользователем, в значение, допустимое для элементов матрицы A, помещается в охраняемый блок. Если данные некорректны и возникает исключительная ситуация, то она перехватывается универсальным обработчиком catch(Exception). Заметьте, в данном варианте нет цикла, работающего до тех пор, пока не будет введено корректное значение. Обработчик исключения просто прерывает работу по переносу данных, вызывая оператор return. Но предварительно он формирует информационное сообщение об ошибке и выводит его в форму. (Помните, специально для этих целей у формы были заготовлены две метки). В сообщении пользователю предлагается исправить некорректно заданный элемент и повторить ввод - повторно нажать командную кнопку "перенести данные в массив". Этот подход понятен и легко реализуем. Недостатком является его неэффективность, поскольку повторно будут переноситься в массив все элементы, в том числе и те, что были введены вполне корректно. У программиста такая ситуация может вызывать чувство неудовлетворенности своей работой.
- На примере ввода элементов матрицы B продемонстрируем другой подход, когда исправляется только некорректно заданное значение. Прежде чем читать дальше, попробуйте найти собственное решение этой задачи. Это не так просто, как может показаться с первого взгляда. Для организации диалога с пользователем пришлось организовать специальное диалоговое окно, представляющее обычную форму с двумя элементами управления - меткой для выдачи информационного сообщения и текстовым окном для ввода пользователем корректного значения. При обнаружении ошибки ввода открывается диалоговое окно, в которое пользователь вводит корректное значение элемента и закрывает окно диалога. Введенное пользователем значение переносится в нужную ячейку таблицы DataGridView, а оттуда в матрицу.
- При проектировании диалогового окна значение свойства формы FormBorderStyle, установленное по умолчанию как "sizeable", следует заменить значением "FixedDialog", что влияет на внешний вид и поведение формы. Важно отметить, что форма, представляющая диалоговое окно, должна вызываться не методом Show, а методом ShowDialog. Иначе произойдет зацикливание, начнут порождаться десятки диалоговых окон, прежде чем вы успеете нажать спасительную в таких случаях комбинацию **Ctrl+ Alt + Del**.

Обработчик события "Click" командной кнопки "Умножить матрицы" выполняет ответственные задачи - реализует умножение матриц и отображает полученный результат в таблице соответствующего элемента DataGridView. Но оба эти действия выполняются естественным образом, не требуя, кроме циклов, никаких специальных средств и программистских ухищрений. Программный код:

```
private void button3_Click(object sender, EventArgs e)
    {
      MultMatr(A, B, C);
      FillDG();
    }
        void MultMatr(double[,] A, double[,] B, double[,] C)
    {
      int m = A.GetLength(0);
      int n = A.GetLength(1);
      int p = B.GetLength(1);
      double S =0;
      for(int i=0; i < m; i++)
        for (int j = 0; j < p; j++)
        {
          S = 0;
          for (int k = 0; k < n; k++)
            S += A[i, k] * B[k, j];
          C[i, j] = S;
        }
    }
void FillDG()
    {
      for (int i = 0; i < m; i++)
        for (int j = 0; j < p; j++)
          dataGridView3.Rows[i].Cells[j].Value
            = C[i, j].ToString();
    }
```

Задачи (ввод, вывод и другие простые задачи с массивами)

- 1. Организуйте в консольном приложении ввод и вывод одномерного массива строкового типа.
- 2. Организуйте в Windows-приложении ввод и вывод одномерного массива строкового типа.
- 3. Организуйте в консольном приложении ввод массива "Сотрудники", содержащего фамилии сотрудников. Введите массив "Заявка", элементы которого содержат фамилии сотрудников и, следовательно, должны содержаться в массиве сотрудников. Обеспечьте контроль корректности ввода данных.
- 4. Организуйте в Windows-приложении ввод массива "Сотрудники", содержащего фамилии сотрудников. Введите массив "Заявка", элементы которого содержат фамилии сотрудников и, следовательно, должны содержаться в массиве сотрудников. Обеспечьте контроль корректности ввода данных.

Лекция 8

Тема: Работа со строками

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.

Тип string в языке C# допускает двойственную интерпретацию. С одной стороны, значения переменной типа string можно рассматривать, как неделимое значение - скаляр - строку текста. С другой стороны, это значение можно интерпретировать, как массив из n элементов, где n - это длина строки. Каждый такой элемент задает отдельный символ и принадлежит символьному типу char.

```
string s1 = "рок", s2 = "око",;
char ch1, ch2, ch3;
ch1 = s1[0];          ch2 = s1[1];          ch3 = s1[2];
string s3 = s1 + s2;
```

В этом примере показано, как можно работать с отдельными символами строки и как можно работать со скалярным представлением строки.

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;
- Unicode-последовательностью, задающей Unicode код символа.

Вот несколько примеров объявления символьных переменных и работы с ними:

```
/// <summary>
/// Символы, коды, строки
/// </summary>
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. Поскольку у каждого символа есть свой код, 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 в латинском алфавите, а ее код - переменная code - 65. Хотя преобразования символа в код и обратно просты, полезно иметь процедуры, выполняющие взаимно-обратные операции, - получение по коду символа и получение символа по его коду:

```
/// <summary>
  /// Код символа
  /// </summary>
  /// <param name="sym">символ</param>
  /// <returns>его код</returns>
  public static int SayCode(char sym)
  {
     return sym;
  }//SayCode

  /// <summary>
  /// Символ
  /// </summary>
  /// <param name="code">Код символа</param>
  /// <returns>символ</returns>
  public static char SaySym(int code)
  {
     return (char)code;
  }// SaySym
```

В первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное.
Говоря о символах и их кодировке, следует помнить, что для символов алфавитов естественных языков (латиницы, кириллицы) применяется плотная кодировка. Это означает, что поскольку буква z в латинице следует за буквой y, код z на единицу больше кода y. Только буква "Ё" в кириллице не подчиняется этому правилу. Для цифр также используется плотная кодировка, и их коды предшествуют кодам букв. Заглавные буквы в кодировке предшествуют строчным. Ряд символов воспринимаются как управляющие, выполняя при их появлении определенное действие. К подобным относятся такие символы, как "перевод строки" (new line), "возврат каретки" (carriage return), "звонок". Эти символы не имеют видимого образа, а их коды задаются escape последовательностями ('\n', '\r'). Поскольку алфавит, задаваемый Unicode-кодировкой, содержит более 65000 символов, большинство кодов зарезервировано и им пока не соответствуют реальные символы.
Рассмотрим пример, демонстрирующий коды некоторых символов.

```
 // <summary>
  /// Преобразования код <-> символ
  /// </summary>
  public void SymToFromCode()
  {
     char sym1 = '0', sym2 = 'a',
    sym3 = 'A', sym4 = '\r',
    sym5 = 'a', sym6 = 'A';
     PrintCode(sym1); PrintCode(sym2);
     PrintCode(sym3); PrintCode(sym4);
     PrintCode(sym5); PrintCode(sym6);
     int code1 = 13, code2 = 122,
    code3 = 1071, code4 = 70000;
     PrintSym(code1); PrintSym(code2);
     PrintSym(code3); PrintSym(code4);
  }
```

Процедуры печати PrintCode и PrintSym достаточно просты, так что код их не приводится. Результат работы этого метода показан на рис. 7.1.

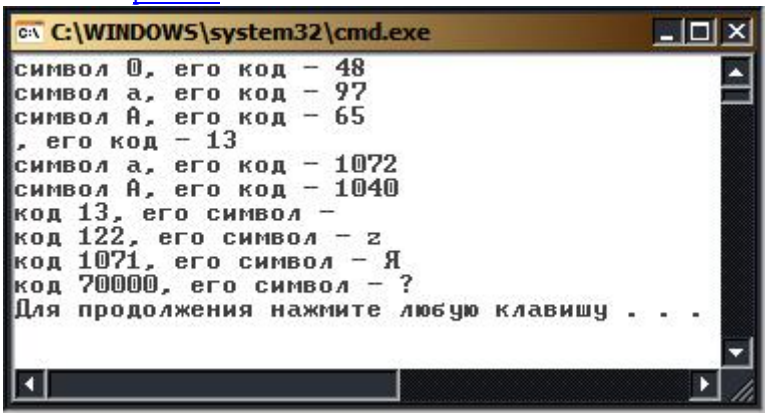


Рис. 7.1. Символы и их коды

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. Приведу сводку этих методов.

Таблица 7.1. 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 шестнадцатеричной 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.

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:

```
/// <summary>
  /// Свойства символов
  /// </summary>
  public void TestCharMethods()
  {
     Console.WriteLine("Метод GetUnicodeCategory:");
     System.Globalization.UnicodeCategory cat1, cat2;
     cat1 = char.GetUnicodeCategory('A');
     cat2 = char.GetUnicodeCategory(';');
     Console.WriteLine("'A' - category {0}", cat1);
     Console.WriteLine("';' - category {0}", cat2);
     Console.WriteLine("Метод IsLetter:");
     Console.WriteLine("'z' - IsLetter - {0}",
       char.IsLetter('z'));
     Console.WriteLine("'Я' - IsLetter - {0}",
        char.IsLetter('Я'));
     Console.WriteLine("Метод IsLetterOrDigit:");
     Console.WriteLine("'7' - IsLetterOrDigit - {0}",
        char.IsLetterOrDigit('7'));
     Console.WriteLine("'Я' - IsLetterOrDigit - {0}",
        char.IsLetterOrDigit('Я'));
     Console.WriteLine("Метод IsControl:");
     Console.WriteLine("';' - IsControl - {0}",
        char.IsControl(';'));
     Console.WriteLine(@"'\r' - IsControl - {0}",
        char.IsControl('\r'));
     Console.WriteLine("Метод IsSeparator:");
     Console.WriteLine("' ' - IsSeparator - {0}",
        char.IsSeparator(' '));
     Console.WriteLine("';' - IsSeparator - {0}",
        char.IsSeparator(';'));
     Console.WriteLine("Метод IsWhiteSpace:");
     Console.WriteLine("' ' - IsWhiteSpace - {0}",
```

```
    char.IsWhiteSpace(' '));
  Console.WriteLine(@"'\r' - IsWhiteSpace - {0}",
 char.IsWhiteSpace('\r'));
}//TestCharMethods
```

Вот как выглядят результаты консольного вывода, порожденного выполнением метода.

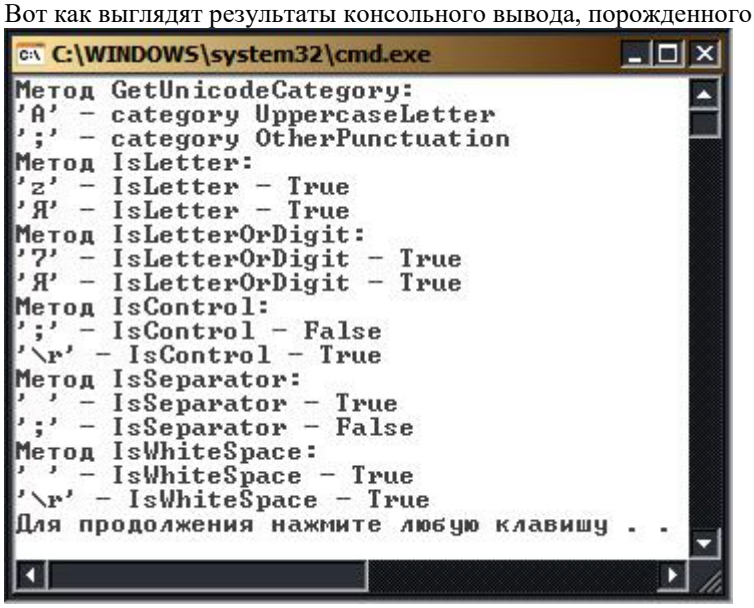


Рис. 7.2. Свойства символов

Статические свойства и методы класса string

Таблица 7.2. 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, и, как следствие, автоматически вызывается метод Format, форматирующий строку перед выдачей ее на печать. В данном примере используется простейший вид формата, - он определяет объект, который должен быть подставлен в участок строки, занятый данным форматом. Помимо неявных вызовов метода Format нередко возникает необходимость явного форматирования строки.

Давайте рассмотрим общий синтаксис метода Format 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 сигнатуры двух реализаций этого метода:

```
public static string Format(IFormatProvider, string, object);
public static string Format(string, params object[]);
```

Параметр типа string задает форматируемую строку. Заданная строка содержит один или несколько форматов, составляющих список форматов. Признаком формата в строке являются фигурные скобки, окружающие формат. Списку форматов ставится в соответствие список объектов, следующий за форматируемой строкой. Чаще всего оба списка имеют одинаковую длину, но это не обязательное требование, поскольку один и тот же объект может по-разному форматироваться. Каждый формат однозначно определяет объект из списка объектов. Этот объект преобразуется в строку текста, текст форматируется в соответствии с параметрами, задаваемыми форматом, и подставляется в то место строки, где расположен формат. Так что форматы в строке - это держатели места (placeholder), куда подставляется форматируемый текст. Метод Format в качестве результата возвращает переданную ему строку, где все форматы заменены строками, полученными в результате форматирования объектов.
Общий синтаксис, специфицирующий **формат**, таков:
{N [,M [:<коды_форматирования>]]}

Обязательный параметр N задает индекс объекта в списке объектов. 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. Параметр M может быть положительным или отрицательным, в зависимости от этого производится выравнивание подставляемой строки по левому или правому краю поля, отводимого вставляемому тексту.
Третий необязательный параметр задает коды форматирования, указывающие, как следует форматировать объект. Применяются разные коды форматирования для числовых данных, дат, перечислений. Например, для числовых данных 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. Код F для дат позволяет вывести в полном формате дату и время. Полный набор кодов форматирования можно посмотреть в справочной системе. Частично их эффект демонстрируется в данном примере:

```
enum Rainbow {красный, желтый, голубой};
/// <summary>
/// Форматирование чисел, дат, перечислений
/// </summary>
public void TestFormat()
{
   int x = 77;
   double p = 0.52;
   double d = -151.17;
   DateTime today = DateTime.Now;
   //Форматирование чисел
   string s = string.Format("Итого:{0:P}\n" +
      "Сумма_1 = {1:C}\n" +
      "x = {1:#######} рублей\n" +
      "d = {2,-10:F} рублей\n" +
      "d = {2, 10:F} рублей\n" +
      "d = {2:E}\n", p, x, d);
   Console.WriteLine(s);
   //Форматирование дат
   s = string.Format("Время: {0:t}, Дата: {0:d}\n" +
  "Дата и время - {0:F}", today);
   Console.WriteLine(s);
    //Форматирование перечислений
   s = string.Format("Цвет1: {0:G}, Цвет2: {1:F}\n",
   Rainbow.голубой, Rainbow.красный);
   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. Заметьте, консольный вывод всегда можно свести к форме Console.WriteLine(s), если строку s предварительно отформатировать, используя явный вызов метода Format. Этот метод полезно вызывать и в Windows-проектах при выводе специфических данных - денежных сумм, процентов, дат и времени. В примере показано использование различных спецификаций формата с разными кодами форматирования для таких данных. В заключительном фрагменте кода демонстрируется задание провайдером национальных особенностей. С этой целью создается объект класса CultureInfo, инициализированный так, чтобы он задавал особенности форматирования, принятые в США. 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. 7.4.

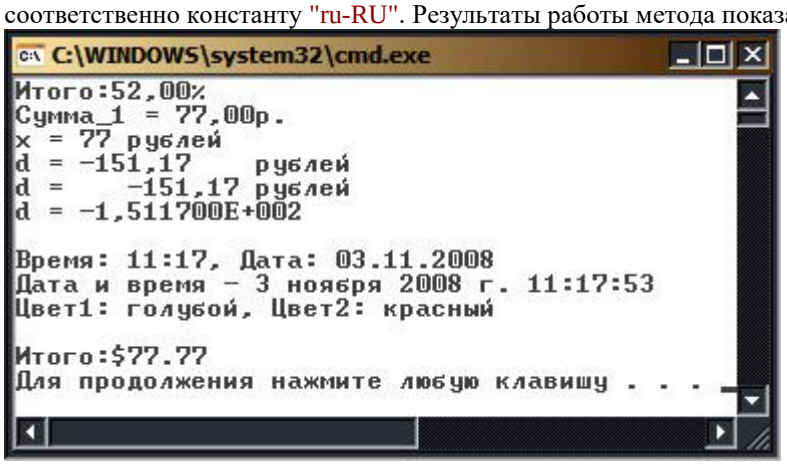


Рис. 7.4. **Результаты работы метода FormatМетоды Join и Split**

Рассмотрим наиболее характерные методы при работе со строками.

Таблица 7.3. 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.

Сводка методов, приведенная в таблице, 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, изменяющие строку, представляют собой функции, возвращающие в качестве результата новую строку.

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, вызов конструктора для создания и 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, 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. Присваивание для строк этого класса является полноценным ссылочным присваиванием, так что изменение значения строки сказывается на всех экземплярах, ссылающихся на строку в динамической памяти. Эквивалентность теперь является проверкой ссылок, а не значений. 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 пример работы со строками, используя класс StringBuilder:

```
/// <summary>
 /// Операции над строками StringBuilder
 /// </summary>
 public void TestStringBuilder()
 {
    string DEL = "->";
    StringBuilder s1 = new StringBuilder("ABC"),
       s2 = new StringBuilder("CDE");
    StringBuilder s3 = s2.Insert(0,s1.ToString());
    s3.Remove(3, 3);
    bool b1 = (s1 == s3);
    char ch1 = s1[2];
    string s = s1.ToString() + DEL + s2.ToString() +
     DEL + s3.ToString() + DEL +
     b1.ToString() + DEL + ch1.ToString();
    Console.WriteLine(s);
    s2.Replace("ABC", "Zenon");
    s1 = s2;
    s2[0] = 'L';
    s1.Append(" - это музыкант!");
    Console.WriteLine(s1.ToString() +
       " -> " + s2.ToString());  }
```

Результаты работы этого метода показаны на рис. 7.6

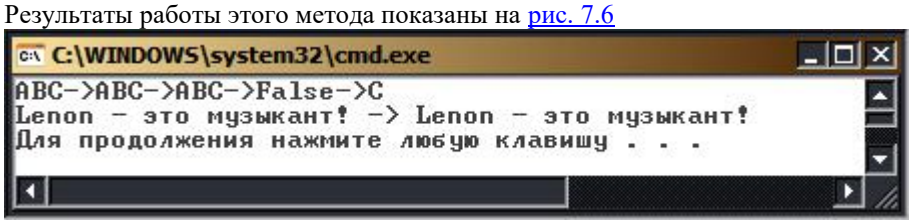


Рис. 7.6. Тип StringBuilder - это изменяемый тип

Этот пример демонстрирует возможность выполнения над строками класса StringBuilder тех же операций, что и над строками класса string. Обратите внимание, теперь методы, изменяющие строку, Replace, Insert, Remove, Append реализованы как процедуры, а не как функции. Они изменяют значение строки непосредственно в буфере, отводимом для хранения строки. Появляется новая возможность - изменять отдельные символы строки.

Основные методы

У класса StringBuilder методов 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 такова: создается обычная строка; из нее 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.

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.

Литература

1. Павловская, Т.А. C#. Программирование на языке высокого уровня: [учебник для вузов] /
2. Т. А. Павловская. - Санкт-Петербург : Питер, 2014. - 432 с. : ил. - Библиогр.: с. 425-426.
    1) Балена, Ф. Современная практика программирования на Microsoft Visual Basic и Visual C#/ пер. с англ.. - М. : Русская Редакция, 2006. - 640 с. – Режим доступа:
3. http://msdn.microsoft.com/ru-ru/library
4. Баженова, И.Ю. Языки программирования : учебник для студ. вузов/ под ред. В.А.Сухомлина. - М. : Академия, 2012. - 368 с.
5. Культин, Н. Б. Основы программирования в Microsoft Visual C# 2010. — СПб.: БХВ-
6. Петербург, 2011. — 364 с. – Режим доступа: http://znanium.com/bookread.php?book=351294
7. Златопольский, Д.М.Сборник задач по программированию. - 3-е изд.. - СПб. : БХВ Петербург, 2011. - 304 с.
8. Брайант, Р. Компьютерные системы: архитектура и программирование: взгляд программиста: для препод, студ. и программистов / пер. с англ.. - СПб. : БХВ-Петербург, 2005. - 1104 с.
9. Васильев, А.Н. Java. Объектно-ориентированное программирование: учеб. пособие для магистров и бакалавров. - СПб. : Питер, 2011. - 400 с.
10. Тяпичев, Г. А. Быстрое программирование на C++. — М.: СОЛОН-Пресс, 2008. —384 с.- – Режим доступа: http://www.bibliorossica.com/book.html?currBookId=10556

Интернет-ресурсы

1) ЭБС «БиблиоРоссика» – http://www.bibliorossica.com/
2) ЭБС «Лань» – http://e.lanbook.com/
3) ЭБС «Научная электронная библиотека» – http://eLIBRARY.RU
4) ЭБС «Знание» – http://znanium.com/
5) Computers & Applied Sciences Complete
– http://search.ebscohost.com/
6) Электронная библиотека «Academic Complete» – http://site.ebrary.com/lib/kazanst/