Марченко А.Л.

C#. Введение в программирование

Учебное пособое

Издательство Московского университета
2005

Московский Государственный Университет
им. М.В.Ломоносова

Геологический факультет
кафедра сейсмометрии и геоакустики

Марченко А.Л.

C#. Введение в программирование

Учебное пособое

Издательство Московского университета
2005

Рецензенты:

В пособии описывается синтаксис языка программирования Microsoft C# .NET, множества классов, применяемых для разработки приложений на платформе .NET, излагаются основы объектно-ориентированного программирования, методы создания элементов пользовательского интерфейса и работы с базами данных.

Для студентов младших курсов.

Предисловие

Настоящее учебное пособие представляет собой краткое изложение лекций, читаемых в рамках двухсеместрового учебного курса по информатике.

Основные задачи пособия заключаются:

- в ознакомлении слушателей с синтаксисом и семантикой языка программирования C#,
- в описании особенностей архитектуры .NET,
- в формировании навыков разработки приложений в рамках парадигмы объектно-ориентированного программирования.

При описании синтаксиса некоторых языковых конструкций C# в пособии использовалась нотация Бэкуса-Наура. Формы Бэкуса-Наура (БНФ) традиционно используются при описании грамматики формальных языков (в том числе и языков программирования). Несмотря на непривычный внешний вид, эти формы достаточно просто интерпретируются, отличаются лаконичностью и точностью. Формы состоят из доступных пониманию буквосочетаний, называемых нетерминальными и терминальными символами. Особого внимания в БНФ заслуживает символ ‘::=’, который в буквальном смысле переводится как “СОСТОИТ ИЗ”. Определение языковой конструкции в БНФ предполагает размещение какого-либо нетерминального символа слева от символа ‘::=’. В правой части формы размещается последовательность отображающих структуру определяемого понятия нетерминальных и терминальных символов. Терминальные символы не требуют расшифровки (дополнительных БНФ), поскольку являются конструкциями описываемого языка программирования. Некоторые элементы в рамках БНФ заключаются в прямые скобки. Так в данной нотации обозначаются те элементы описываемой синтаксической конструкции, количество вхождений которых с точки зрения систаксиса не ограничено. В данном контексте их может быть один, два, три, … , много, а может и не быть вовсе.

Пособие разрабатывалось на основе находящейся в открытом доступе литературы и не исключает необходимости ознакомления по крайней мере с несколькими из приводимых в списке литературы и ставшими на сегодняшний день классическими книг по языку программирования C#.

В приложении содержится большое количество примеров. В ряде случаев это всего лишь документированные фрагменты программного кода, однако значительная часть приводимых в пособии примеров является законченными работающими приложениями. И хотя к пособию НЕ прилагается никаких контрольных вопросов, упражнений или задач, необходимым условием успешного усвоения материала (и обязательным заданием!) является воспроизведение, анализ и модификация приводимого в пособии кода.

В примерах часто используются операторы консольного вывода вида:

```
System.Console.WriteLine(…);
```

До момента подробного обсуждения темы ввода-вывода в соответствующем разделе пособия, предложения этого вида можно рассматривать как некоторые “волшебные” заклинания, в результате которых в окошке консольного приложения появляются последовательности выводимых символов.

Введение

Обзор .NET. Основные понятия

ПЛАТФОРМА - это как минимум среда выполнения программ и… ещё что-либо, что определяет особенности разработки и выполнения программного кода – парадигмы программирования, языки программирования, множества базовых классов.

Microsoft.NET (.NET Framework) – программная платформа. Содержит следующие основные компоненты: the common language runtime (CLR) and the .NET Framework class library (.NET FCL).

CLS (Common Language Specification) – общая спецификация языков программирования. Это набор конструкций и ограничений, которые являются руководством для создателей библиотек и компиляторов в среде .NET Framework. Библиотеки, построенные в соответствии с CLS, могут быть использованы из любого языка программирования, поддерживающего CLS. Языки, соответствующие CLS (к их числу относятся языки Visual C#, Visual Basic, Visual C++), могут интегрироваться друг с другом. CLS – это основа межъязыкового взаимодействия в рамках платформы Microsoft.NET.

CLR (Common Language Runtime) – Среда Времени Выполнения или Виртуальная Машина. Обеспечивает выполнение сборки. Основной компонент .NET Framework. Под Виртуальной Машиной понимают абстракцию инкапсулированной (обособленной) управляемой операционной системы высокого уровня, которая обеспечивает выполнение программного кода и предполагает решение следующих задач:

- управление кодом (загрузку и выполнение),
- управление памятью при размещении объектов,
- изоляцию памяти приложений,
- проверку безопасности кода,
- преобразование промежуточного языка в машинный код,
- доступ к метаданным (расширенная информация о типах),
- обработка исключений, включая межъязыковые исключения,
- взаимодействие между управляемым и неуправляемым кодом (в том числе и COM-объектами),
- поддержка сервисов для разработки (профилирование, отладка и т.д.).

Короче, CLR – это набор служб, необходимых для выполнения сборки. При этом программный код сборки может быть как управляемым (код, при выполнении которого CLR, в частности, активизирует систему управления памяти), так и неуправляемым (“старый” программный код).

Сама CLR состоит из двух главных компонентов: ядра (mscoree.dll) и библиотеки базовых классов (mscorlib.dll). Наличие этих файлов на диске – верный признак того, что на компьютере, по крайней мере, была предпринята попытка установки платформы .NET.

Ядро среды выполнения реализовано в виде библиотеки mscoree.dll. При компоновке сборки в неё встраивается специальная информация, которая при запуске приложения (EXE) или при загрузке библиотеки (обращение к DLL из неуправляемого модуля – вызов функции LoadLibrary для загрузки управляемой сборки) приводит к загрузке и инициализации CLR. После загрузки CLR в адресное пространство процесса, ядро среды выполнения выполняет следующие действия:

- находит местонахождение сборки,
- загружает сборку в память,
- производит анализ содержимого сборки (выявляет классы, структуры, интерфейсы),
- производит анализ метаданных,
- обеспечивает компиляцию кода на промежуточном языке (IL) в платформозависимые инструкции (ассемблерный код),
- выполняет проверки, связанные с обеспечением безопасности,
- используя основной поток приложения, передаёт управление преобразованному в команды процессора фрагменту кода сборки.

FCL (.NET Framework Class Library) – соответствующая CLS спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы Microsoft .NET.

Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена в качестве основы при разработке .NET приложений, компонент, элементов управления.

.NET библиотека классов является вторым компонентом CLR.

.NET FCL могут использовать ВСЕ .NET-приложения, независимо от назначения, архитектуры, используемого при разработке языка программирования. В частности, содержит:

- встроенные (элементарные) типы, представленные в виде классов (на платформе .NET всё построено на структурах или классах),
- классы для разработки графического пользовательского интерфейса (Windows Form),
- классы для разработки Web-приложений и Web-служб на основе технологии ASP.NET (Web Forms),
- классы для разработки XML и Internet-протоколами (FTP, HTTP, SMTP, SOAP),
- классы для разработки приложений, работающих с базами данных (ADO.NET),
- и многое другое.

.NET-приложение – приложение, разработанное для выполнения на платформе Microsoft.NET. Реализуется на языках программирования, соответствующих CLS.

MSIL (Microsoft Intermediate Language, он же IL – Intermedia Language) – промежуточный язык платформы Microsoft.NET. Исходные тексты программ для .NET приложений пишутся на языках программирования, соответствующих спецификации CLS. Для языков программирования, соответствующих спецификации CLS может быть построен преобразователь в MSIL. Таким образом, программы на этих языках могут транслироваться в промежуточный код на MSIL. Благодаря соответствию CLS, в результате трансляции программного кода, написанного на разных языках, получается совместимый IL код.

Фактически MSIL является ассемблером виртуального процессора.

МЕТАДАННЫЕ - при преобразовании программного кода в MSIL также формируется блок МЕТАДАННЫХ, содержащий информацию о данных, используемых в программе. Фактически это наборы таблиц, содержащих информацию о типах данных, определяемых в модуле, о типах данных, на которые ссылается данный модуль. Ранее такая информация сохранялась отдельно. Например, приложение могло включать информацию об интерфейсах, которая описывалась на Interface Definition Language (IDL). Теперь метаданные являются частью управляемого модуля.

В частности, метаданные используются для:

- сохранения информации о типах. При компиляции теперь не требуются заголовочные и библиотечные файлы. Всю необходимую информацию компилятор читает непосредственно из управляемых модулей,
- верификации кода в процессе выполнения модуля,
- управления динамической памятью (освобождение памяти) в процессе выполнения модуля,
- при разработке программы стандартными инструментальными средствами (Microsoft Visual Studio.NET) на основе метаданных обеспечивается динамическая подсказка (IntelliSense).

Языки, для которых реализован перевод на MSIL:
Visual Basic,
Visual C++,
Visual C#,
и ещё много других языков.

Исполняемый модуль - независимо от компилятора (и входного языка) результатом трансляции .NET приложения является управляемый исполняемый модуль (управляемый модуль). Это стандартный переносимый исполняемый (PE – Portable Executable) файл Windows.

Элементы управляемого модуля представлены в таблице.

Заголовок PE	Показывает тип файла (например, DLL), содержит временную метку (время сборки файла), содержит сведения о процессорном коде.
Заголовок CLR	Содержит информацию для среды выполнения модуля (версию требуемой среды исполнения, характеристики метаданных, ресурсов и т.д.). Собственно эта информация делает модуль управляемым.
Метаданные	Таблицы метаданных: 1. типы, определённые в исходном коде, 2. типы, на которые имеются в коде ссылки.
IL	Собственно код, который создаётся компилятором при компиляции исходного кода. На основе IL в среде выполнения впоследствии формируется множество команд процессора.

Управляемый модуль содержит управляемый код.

Управляемый код - это код, который выполняется в среде CLR. Код строится на основе объявляемых в исходном модуле структур и классов, содержащих объявления методов. Управляемому коду должен соответствовать определенный уровень информации (метаданных) для среды выполнения. Код C#, Visual Basic, и JScript является управляемым по умолчанию. Код Visual C++ не является управляемым по умолчанию, но компилятор может создавать управляемый код, для этого нужно указать аргумент в командной строке(/CLR). Одной из особенностей управляемого кода является наличие механизмов, которые позволяют работать с УПРАВЛЯЕМЫМИ ДАННЫМИ.

Управляемые данные - объекты, которые в ходе выполнения кода модуля размещаются в управляемой памяти (в управляемой куче) и уничтожаются сборщиком мусора CLR. Данные C#, Visual Basic и JScript .NET являются управляемыми по умолчанию. Данные C# также могут быть помечены как неуправляемые.

Сборка (Assembly) - базовый строительный блок приложения в .NET Framework. Управляемые модули объединяются в сборки. Сборка является логической группировкой одного или нескольких управляемых модулей или файлов ресурсов. Управляемые модули в составе сборок исполняются в Среде Времени Выполнения (CLR). Сборка может быть либо исполняемым приложением (при этом она размещается в файле с расширением .EXE), либо библиотечным модулем (в файле с расширением .DLL). При этом ничего общего с обычными (старого образца!) исполняемыми приложениями и библиотечными модулями сборка не имеет.

Декларация сборки (Manifest) - составная часть сборки. Ещё один набор таблиц метаданных, который:

- идентифицирует сборку в виде текстового имени, её версию, культуру и цифровую сигнатуру (если сборка разделяется среди приложений),
- определяет входящие в состав файлы (по имени и хэшу),
- указывает типы и ресурсы, существующие в сборке, включая описание тех, которые экспортируются из сборки,
- перечисляет зависимости от других сборок,
- указывает набор прав, необходимых сборке для корректной работы.

Эта информация используется в период выполнения для поддержки корректной работы приложения.

Процессор НЕ МОЖЕТ выполнять IL код. И перевод IL кода осуществляется JIT-компилятором (just in time – в нужный момент), который активизируется CLR по мере необходимости и выполняется процессором. При этом результаты деятельности JIT-компилятора сохраняются в оперативной памяти. Между фрагментом оттранслированного IL кода и соответствующим блоком памяти устанавливается соответствие, которое в дальнейшем позволяет CLR передавать управление командам процессора, записанным в этом блоке памяти, минуя повторное обращение к JIT-компилятору.

В среде CLR допускается совместная работа и взаимодействие компонентов программного обеспечения, реализованных на различных языках программирования.

На основе ранее сформированного блока метаданных CLR обеспечивает ЭФФЕКТИВНОЕ взаимодействие выполняемых .NET приложений.

Для CLR все сборки одинаковы, независимо от того на каких языках программирования они были написаны. Главное – это чтобы они соответствовали CLS. Фактически CLR разрушает границы языков программирования (cross-language interoperability). Таким образом, благодаря CLS и CTS .NET-приложения фактически оказываются приложениями на MSIL (IL).

CLR берёт на себя решение многих проблем, которые традиционно находились в зоне особого внимания разработчиков приложений. К числу функций, выполняемых CLR, относятся:

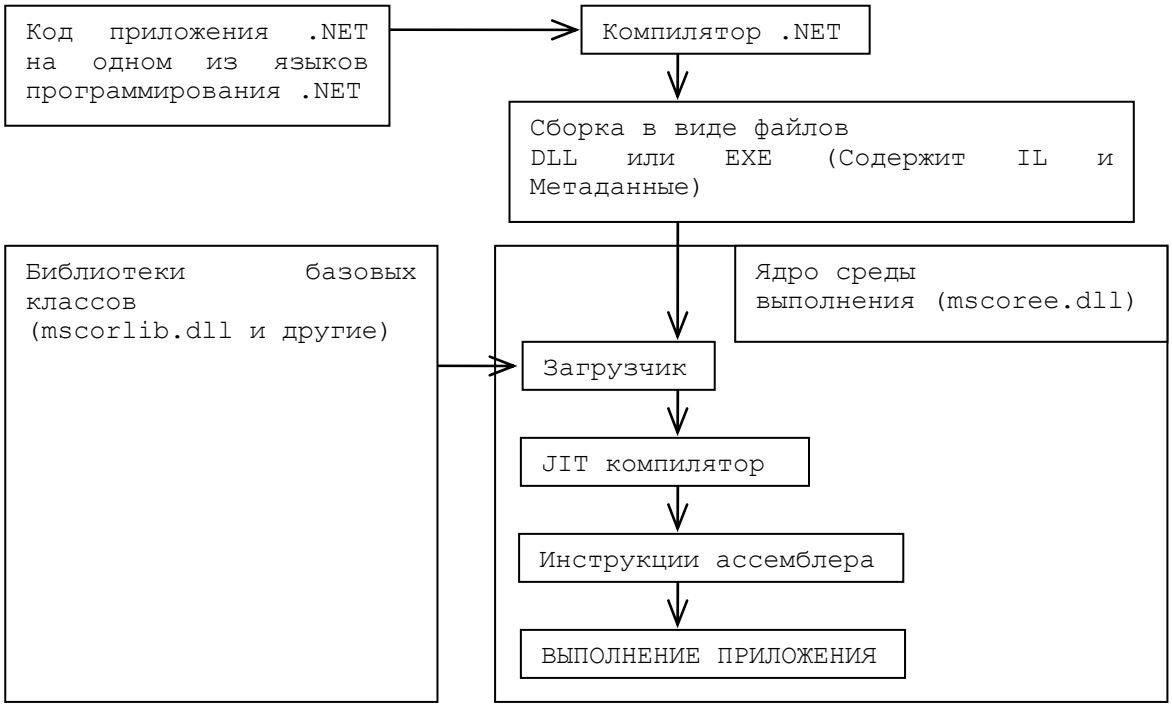
- проверка и динамическая (JIT) компиляция MSIL кода в команды процессора,
- управление памятью, процессами и потоками,
- организация взаимодействия процессов,
- решение проблем безопасности (в рамках существующей в системе политики безопасности).

AppDomain (домен приложения) - это логический контейнер сборок, который используется для изоляции приложения в рамках адресного пространства процесса. Все объекты, создаваемые приложением, создаются в рамках определенного домена приложения. Несколько доменов приложений могут существовать в одном процессе операционной системы. CLR изолирует приложения, управляя памятью в рамках домена приложения.

Код, выполняемый в CLR (CLR процесс) отделён от других процессов, выполняемых на компьютере в это же самое время.

Обычный процесс запускается системой в рамках специально выделяемого процессу адресного пространства. CLR предоставляет возможность выполнения множества управляемых приложений в ОДНОМ ПРОЦЕССЕ. Каждое управляемое приложение связывается с собственным доменом приложения (сокращенно AppDomain). В приложении помимо основного домена может быть создано несколько дополнительных доменов.

Структура среды выполнения CLR представлена на картинке.



Свойства доменов:

- домены изолированы друг от друга. Объекты, созданные в рамках одного домена недоступны из другого домена,
- CLR способна выгружать домены вместе со всеми сборками, связанными с этими доменами,
- возможна дополнительная конфигурация и защита доменов,
- для обмена данными между доменами реализован специальный механизм безопасного доступа (маршалинг).
- В .NET Framework разработана собственная компонентная модель, элементами которой являются .NET-сборки (.NET-assembly), а для прямой и обратной совместимости с моделью COM/COM+ в CLR встроены механизмы (COM Interop), обеспечивающие доступ к COM-объектам по правилам .NET и к .NET-сборкам по

правилам COM. При этом для .NET-приложений не требуется регистрации компонентов в системном реестре Windows.

Для выполнения .NET-приложения достаточно разместить относящиеся к данному приложению сборки в одном каталоге. Если при этом сборка может быть использована в нескольких приложениях, то она размещается и регистрируется с помощью специальной утилиты в GAC (Global Assembly Cache Общем КЭШе сборок).

CTS - Common Type System Стандартная Система Типов. Поддерживается всеми языками платформы. В силу того, что .NET дитя ООП – то речь здесь идёт об элементарных типах, классах, структурах, интерфейсах, делегатах и перечислениях.

Common Type System является важной частью среды выполнения, определяет структуру синтаксических конструкций, способы объявления, использования, и применения ОБЩИХ типов среды выполнения. В CTS сосредоточена основная информация о системе ОБЩИХ ПРЕДОПРЕДЕЛЁННЫХ типов, об их использовании и управлении (правилах преобразования значений). CTS играет важную роль в деле интеграции разноязыких управляемых приложений.

Пространство имён – это способ организации системы типов в единую группу. Существует общая общеязыковая библиотека базовых классов. И концепция пространства имён обеспечивает эффективную организацию и навигацию в этой библиотеке. Вне зависимости от языка программирования доступ к определённым классам обеспечивается за счёт их группировки в рамках общих пространств имён.

Пространство имён	Назначение
System	
System.Data	Для обращения к базам данных
System.Data.Common	
System.Data.OleDb	
System.Data.SqlClient	
System.Collections	Классы для работы с контейнерными объектами
System.Diagnostics	Классы для трассировки и отладки кода
System.Drawing	Классы графической поддержки
System.Drawing.Drawing2D	
System.Drawing.Printing	
System.IO	Поддержка ввода-вывода
System.Net	Поддержка передачи данных по сетям
System.Reflection	Работа с пользовательскими типами во время выполнения приложения
System.Reflection.Emit	
System.Runtime.InteropServices	Поддержка взаимодействия с “обычным кодом” – DLL, COM-серверы, удалённый доступ
System.Runtime.Remoting	
System.Security	Криптография, разрешения
System.Threading	Работа с потоками
System.WEB	Работа с web-приложениями
System.Windows.Form	Работа с элементами интерфейса Windows
System.XML	Поддержка данных в формате XML

Выполнение неуправляемых исполняемых модулей (обычные Windows приложения), обеспечивается непосредственно системой Windows. Неуправляемые модули выполняются в среде Windows как “простые” процессы. Единственное требование, которому должны отвечать подобные модули – корректная работа в среде Windows. Они должны “правильно” работать (не вешать систему, не допускать утечек памяти, не блокировать другие процессы и корректно использовать средства самой ОС для работы от имени процессов). То есть, соответствовать наиболее общим правилам работы под Windows.

При этом большинство проблем корректного выполнения неуправляемого модуля (проблемы взаимодействия, выделения и освобождения памяти) являются проблемами разработчиков приложений. Например, известная технология COM является способом организации взаимодействия разнородных компонентов в рамках приложения.

Объект – в широком смысле это область памяти (стеке или куче), выделяемая в процессе выполнения программы для записи каких-либо значений. Характеризуется типом (фиксированным набором свойств, определяющих размер занимаемой области, способ интерпретации значения, диапазон значений, множество действий, допустимых при манипуляциях с объектом) местом расположения в памяти (адресом).

Сборка мусора - механизм, позволяющий CLR определить, когда объект становится недоступен в управляемой памяти программы. При сборке мусора

управляемая память освобождается. Для разработчика приложения наличие механизма сборки мусора означает, что он больше не должен заботиться об освобождении памяти. Однако это может потребовать изменения в стиле программирования, например, особое внимание следует уделять процедуре освобождения системных ресурсов. Необходимо реализовать методы, освобождающие системные ресурсы, находящиеся под управлением приложения.

Стек - специальным образом организованная область памяти, предназначенный для временного хранения значений объектов (переменных и констант), для передачи параметров при вызове методов, для сохранения адреса возврата. Управление стеком по сравнению с кучей достаточно просто. Оно основано на изменении значения соответствующего регистра вершины стека. При сокращении размера стека объекты просто теряются.

Программа на C#

Программа – правильно построенная (не вызывающая возражений со стороны C# компилятора) последовательность предложений, на основе которой формируется сборка.

В общем случае, программист создаёт файл, содержащий объявления класов, который подаётся на вход компилятору. Результат компиляции представляется транслятором в виде сборки и определяется предпочтениями программиста. В принципе сборка может быть двух видов:

- Portable Executable File (PE-файл с расширением .exe), пригоден к непосредственному исполнению CLR,
- Dynamic Link Library File (DLL-файл с расширением .dll), предназначен для повторного использования как компонент в составе какого-либо приложения.

В любом случае на основе входного кода транслятор строит модуль на IL, манифест, и формирует сборку. В дальнейшем, сборка либо может быть выполнена после JIT компиляции, либо может быть использована в составе других программ.

Основы языка

Пространство имён

.NET Framework располагает большим набором полезных функций. Каждая из них является членом какого-либо класса. Классы группируются по пространствам имён, которые имеют (как правило) вложенную структуру.

Средством навигации по множествам классов в пространствах имён является оператор

```
usung <ИмяПространстваИмён>;
```

В приложении объявляется собственное пространство имён и используются ранее объявленные пространства.

В процессе построения сборки Visual Studio.NET должен знать расположение сборок с заявленными для использования пространствами имён. Расположение части сборок системе известно изначально. Расположение прочих требуемых приложению сборок указывается явно (окно Solution Explorer проекта, пункт References, Add Reference…). Там надо указать соответствующий .dll или .exe файл.

В частности, сборка, содержащая классы, сгруппированные в пространстве имён System, располагается в файле mscorlib.dll.

Наиболее часто используемое пространство имён – System. Расположение соответствующей сборки известно. Если не использовать оператора

```
usung System;
```

корректное обращение к функции WriteLine(…) члену класса Console выглядело бы следующим образом:

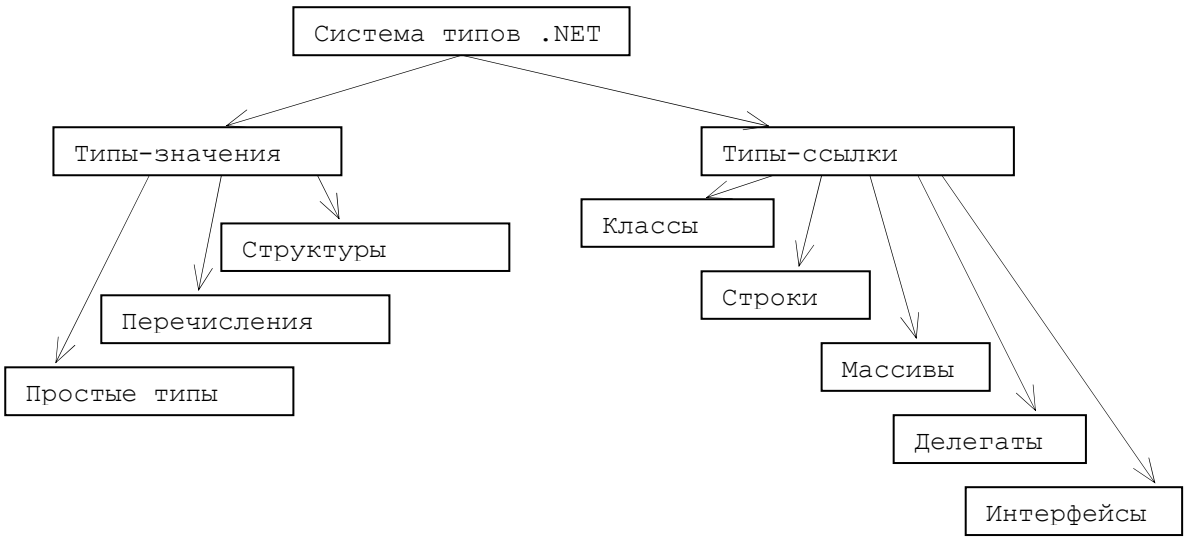
```
System.Console.Writeline("На-На-На!"); // Полное квалифицированное
//имя функции-члена класса Console, отвечающей за вывод строки в окно приложения.
```

При компиляции модуля, транслятор по полному имени функции (если используется оператор using – то по восстановленному) находит её код, который и используется при выполнении сборки.

Система типов

Система типов поддерживает две категории типов, каждая из которых разделена на подкатегории: типы значений (типы-значения) и ссылочные типы (типы-ссылки).

Схема типов представлена ниже.



Все типы, за исключением простых типов, могут определяться программистом. Все остальные типы (производные типы) требуют предварительного объявления.

Простые (элементарные) типы – это типы, имя и основные свойства которых известны компилятору. Относительно встроенных типов компилятору не требуется никакой дополнительной информации. Он поддерживает эти типы самостоятельно.

Среди простых типов различаются:

- ЦЕЛОЧИСЛЕННЫЕ,
- С ПЛАВАЮЩЕЙ ТОЧКОЙ,
- DECIMAL,
- БУЛЕВСКИЙ.

Для обозначения простых (элементарных) типов в C# используется следующая система обозначений. Некоторые характеристики типов отражены в следующей таблице. Смысл точечной нотации в графе "Соответствует FCL-типу" будет разъяснён позже. Пока эту нотацию достаточно воспринимать как составное имя:

Имя типа в C#	Соответствует FCL-типу	Описание
sbyte	System.SByte	Целый. 8-разрядное со знаком. Диапазон значений: 128 … 127
byte	System.Byte	Целый. 8-разрядное без знака. Диапазон значений: 0 … 255
short	System.Int16	Целый. 16-разрядное со знаком. Диапазон значений: -32768 … 32767
ushort	System.UInt16	Целый. 16-разрядное без знака. Диапазон значений: 0 … 65535
int	System.Int32	Целый. 32-разрядное со знаком. Диапазон значений: -2147483648 … 2147483647
uint	System.UInt32	Целый. 32-разрядное без знака. Диапазон значений: -0 … 4294967295
long	System.Int64	Целый. 64-разрядное со знаком. Диапазон значений: -9223372036854775808 … 9223372036854775807
ulong	System.UInt64	Целый. 64-разрядное без знака. Диапазон значений: 0 … 18446744073709551615
char	System.Char	16 (!) разрядный символ UNICODE.
float	System.Single	Плавающий. 32 разряда. Стандарт IEEE.
double	System.Double	Плавающий. 64 разряда. Стандарт IEEE.
decimal	System.Decimal	128-разрядное значение повышенной точности с плавающей точкой.
bool	System.Boolean	Значение true или false.

При создании объекта элементарного типа производится его начальная инициализация предопределённым значением. И за это отвечают предопределённые недоступные для модификации конструкторы.

В C# различаются ВСТРОЕННЫЕ (элементарные - primitive, предопределённые – predefined) и ПРОИЗВОДНЫЕ типы.

Встроенные типы – это типы, имя и основные свойства которых известны компилятору. Производные типы требуют предварительного объявления. Относительно встроенных типов компилятору не требуется никакой дополнительной информации. Он поддерживает эти типы самостоятельно.

В разных CLS-языках типам, удовлетворяющим CLS спецификации, будут соответствовать одни и те же элементарные типы. Согласованная поддержка типов, НЕ соответствующих CLS спецификации, в разных языках не гарантируется.

Система встроенных типов C# основывается на системе типов .NET Framework Class Library. При создании IL кода компилятор осуществляет их отображение в типы из .NET FCL.

Ниже представлены основные отличия ссылочных типов и типов-значений.

	Типы-значения	Типы-ссылки
Объект содержит	значение	ссылку
Располагается	в стеке	в динамической памяти
Значение по умолчанию	0, false, '\0'	null
При присваивании копируется	значение	ссылка

В C# объявление любой структуры и класса основывается на объявлении предопределённого класса Object (наследует класс Object). Следствием этого является возможность вызова от имени объектов-представителей любой структуры

или класса унаследованных от класса Object методов. В частности, метода ToString. Этот метод возвращает строковое (значение типа string) представление объекта.

В C# также существует ещё одно важное ограничение на использование объектов размерных типов. Необходимым условием применения этих объектов является их ЯВНАЯ инициализация.

Класс и Структура. Первое приближение

Эти категории типов-ссылок и типов-значений заслуживают первоочередного внимания. Классы и структуры являются программно-определяемыми типами, которые позволяют определять (создавать) новые типы, специально приспособленные для решения конкретных задач. В рамках объявления класса и структуры описывается множество переменных различных типов (набор данных-членов класса), правила порождения объектов-представителей структур и классов, их основные свойства и методы, применение которых обеспечивает решение задачи.

В программе класс объявляется с помощью специальной синтаксической конструкции, которая называется объявлением класса. Фактически, объявление структур и классов является основным элементом любой C# программы. В программе нет ничего, кроме их объявлений и конструкций, облегчающих процедуру этого объявления.

С точки зрения синтаксиса, между объявлениями классов и структур существует незначительные различия (ключевое слово struct и class, в структуре не допускается объявлений членов класса со спецификаторами доступа protected, protected internal, особенности объявления конструкторов – не допускается объявления конструктора без параметров), часть из которых будет обсуждаться далее.

В этом разделе обсуждаются основные правила объявления классов.

Объявление класса состоит из нескольких элементов:

- объявление атрибутов - необязательный элемент объявления,
- модификаторы прав доступа - необязательный элемент объявления,
- class (struct для структуры),
- имя класса,
- имена предков (класса и интерфейсов) - необязательный элемент объявления,
- тело класса (структуры).

Атрибуты – являются средство добавления ДЕКЛАРАТИВНОЙ (вспомогательной) информации к элементам программного кода. Назначение атрибутов: организация взаимодействия между программными модулями, дополнительная информация об условиях выполнения кода, управление сериализацией (правила сохранения информации), отладка, многое другое.

Модификаторы прав доступа – средство реализации принципа инкапсуляции, используются при объявлении классов, структур и их составляющих компонентов. Представлены следующими значениями:

public	обозначение для общедоступных членов класса. К ним можно обратиться из любого метода любого класса программы.
protected	обозначение для членов класса, доступных в рамках объявляемого класса и из методов производных классов.
internal	обозначение для членов класса, доступных из методов классов, объявляемых в рамках сборки, содержащей объявление данного класса.
protected internal	обозначение для членов класса, доступных в рамках объявляемого класса и из методов производных классов, а также доступных из методов классов, объявляемых в рамках сборки, содержащей объявление данного класса.
private	обозначение для членов класса, доступных в рамках объявляемого класса.

Сочетание ключевого слова class и имя объявляемого класса задаёт имя объявляемого типа как класса (struct ИМЯ задаёт имя структуры).

конструкции
:имя класса
:список имён интерфейсов
:имя класса, список имён интерфейсов

с обязательным разделителем ‘:’ обеспечивают реализацию принципа наследования и будут обсуждаться позже.

Тело класса в объявлении ограничивается парой разделителей ‘{‘, ‘}’, между которыми располагаются объявления данных-членов и методов класса.

Следующий пример является демонстрирует использование основных элементов объявления структуры. При объявлении класса допускается лишь один спецификатор – public (здесь он опущен). Отсутствие спецификаторов доступа в объявлениях членов структуры (класса) эквивалентно явному указанию спецификаторов private.

```
// Указание на используемые пространства имён.
using System;
using System.Drawing;

namespace qwe // Объявление собственного пространства имён. Начало.
{
// Начало объявления структуры
struct S1
{// Тело структуры - НАЧАЛО
// Объявление данных-членов.
private Point p;
// protected int qwe; // Спецификатор protected в объявлении членов
// структуры недопустим.

// Структура не может иметь конструктора без параметров.
public S1(int x, int y)
{
p = new Point(10,10);
}

// Объявление методов.
// Статический метод. Точка входа.
static void Main(string[] args)
{
    // Тело метода. Здесь обычно располагается программный код,
    // определяющий функциональность класса.
}
}// Тело структуры - КОНЕЦ
} // Объявление собственного пространства имён. Конец.
```

Литералы. Представление значений

В программах на языках высокого уровня (C# в том числе) литералами называют последовательности входящих в алфавит языка программирования символов, обеспечивающих явное представление значений, которые используются для обозначения начальных значений в объявлении членов классов, переменных и констант в методах класса.

Различаются литералы арифметические (разных типов), логические, символьные (включая Escape-последовательности), строковые.

Арифметические литералы

Арифметические литералы кодируют значения различных (арифметических) типов. Тип арифметического литерала определяется следующими интуитивно понятными внешними признаками:

- стандартным внешним видом. Значение целочисленного типа обычно кодируется интуитивно понятной последовательностью символов ‘1’,…, ‘9’, ‘0’. Значение плавающего типа также предполагает стандартный вид (точка-разделитель между целой и дробной частью, либо научная или экспоненциальная нотация – 1.2500E+052). Шестнадцатеричное представление целочисленного значения кодируется шестнадцатеричным литералом, состоящим из символов ‘0’,…,‘9’, а также ‘a’,…, ‘f’, либо ‘A’,…,‘F’ с префиксом ‘0x’,
- собственно значением. 32768 никак не может быть типа short,
- дополнительным префиксом. Префиксы l,L соответствуют типу long, ul,UL – unsigned long, f,F – float, d,D – decimal. Значения типа double кодируются без префикса.

Логические литералы

К логическим литералам относятся следующие последовательности символов: true и false. Больше логических литералов в C# нет.

Символьные литералы

заключённые в одинарные кавычки вводимые с клавиатуры одиночные символы: ‘X’, ‘p’, ‘Q’, ‘7’,

целочисленные значения в диапазоне от 0 до 65535, перед которыми располагается конструкция вида (char) – операция явного приведения к типу char: (char)34 – ‘”’, (char)44 – ‘,’, (char)7541 – что здесь будет – не ясно.

Символьные escape-последовательности

Следующие заключённые в одинарные кавычки последовательности символов являются Escape-последовательностями. Эта категория литералов используется для создания дополнительных эффектов (звонок), простого форматирования выводимой информации и кодирования символов при выводе и сравнении (в выражениях сравнения).

Escape-последовательность	Описание
\a	Предупреждение (звонок)
\b	Возврат на одну позицию
\f	Переход на новую страницу
\n	Переход на новую строку
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\0	Ноль
\’	Одинарная кавычка
\”	Двойная кавычка
\\	Обратная косая черта

Строковые литералы

Это последовательность символов и символьных escape-последовательностей, заключённых в двойные кавычки.

A verbatim string – строковый литерал, интерпретируемый компилятором так, как он записан. Escape-последовательности воспринимаются строго как последовательности символов.

Verbatim string представляется при помощи символа @, располагаемого непосредственно перед строковым литералом, заключённым в парные двойные кавычки. Представление двойных кавычек в Verbatim string обеспечивается их дублированием. Пара литералов (второй - Verbatim string):

```
…“c:\\My Documents\\sample.txt”…
…@“c:\My Documents\sample.txt”…
```

имеют одно и то же значение:
c:\My Documents\sample.txt

Представление двойных кавычек внутри Verbatim string достигается за счёт их дублирования:

```
…@“““Focus”””
```

имеет значение

```
“Focus”
```

Строковые литералы являются литералами типа string.

Операции и выражения

Для каждого определённого в C# типа существует собственный набор операций, определённых на множестве значений этого типа.

Эти операции определяют диапазон возможных преобразований, которые могут быть осуществлены над элементами множеств значений типа. Несмотря на специфику разных типов, в C# существует общее основание для классификации соответствующих множеств операций. Каждая операция является членом определённого подмножества операций и имеет собственное графическое представление.

Общие характеристики используемых в C# операций представлены ниже.

Категории операций	Операции
Arithmetic	+ - * / %
Логические (boolean и побитовые)	& | ^ ! ~ && ||
Строковые (конкатенаторы)	+
Increment, decrement	++ --
Сдвига	>> <<
Сравнения	== != < > <= >=
Присвоения	= += -= *= /= %= &= |= ^= <<= >>=
Member access	.
Индексации	[]
Cast (приведение типа)	()
Conditional (трёхоперандная)	?:
Delegate concatenation and removal	+ -
Создания объекта	new()
Type information	is sizeof typeof
Overflow exception control (управление исключениями)	checked unchecked
Indirection and Address (неуправляемый код)	* -> [] &

Так вот на основе элементарных (первичных) выражений с использованием этих самых операций и дополнительных разделителей в виде открывающихся и закрывающихся скобочек формируются выражения всё более сложной структуры. Кроме того, при создании, трансляции, а, главное, на стадии выполнения (определения значения выражения) учитываются следующие обстоятельства:

- приоритет операций,
- типы операндов и приведение типов.

Приоритет операций

Приоритет	Операции
1	() [] . (постфикс)++ (постфикс)-- new sizeof typeof unchecked
2	! ~ (имя типа) +(унарный) -(унарный) ++(префикс) --(префикс)
3	* / %
4	+ -
5	<< >>
6	< > <= => is
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= += -= *= /= %= &= |= ^= <<= >>=

Приведение типов

В одном выражении могут быть сгруппированы операнды различных типов. Однако возможность смешения в одном выражении операндов различных типов связана с определёнными проблемами на этапе выполнения программы и в ряде случаев требует дополнительного внимания со стороны программиста.

Прежде всего, КАК производится это преобразование. Иногда оно производится абсолютно незаметно и не требует дополнительных усилий со стороны программиста. Все необходимые преобразования осуществляется автоматически в ходе выполнения программы. Такие преобразования называются неявными. Однако чаще программист вынужден явным образом указывать необходимость преобразования, используя выражения приведения типа или обращаясь к специальным методам преобразования, определённым в классе System.Convert.

Приведение типов – один из аспектов безопасности языка. Всё должно быть устроено таким образом, чтобы логика преобразования значений одного типа к другому типу была бы понятной, а результаты этих преобразований предсказуемы.

Используемые в программе типы характеризуются собственными диапазонами значений, которые определяются свойствами типов. В том числе и размером области памяти, предназначенной для кодирования значений соответствующего типа. При этом области значений различных типов пересекаются. Многие значения можно выразить более чем одним типом. Например, значение 4 можно представить как целое число или как число с плавающей точкой.

Преобразование типа создает значение нового типа, эквивалентное значению старого типа, однако при этом не обязательно сохраняется идентичность (или точные значения) двух объектов.

Различаются:

Расширяющее преобразование – значение одного типа преобразуется к значению другого типа, которое имеет такой же или больший размер. Например, значение, представленное в виде 32-разрядного целого числа со знаком, может быть преобразовано в 64-разрядное целое число со знаком. Расширяющее преобразование считается безопасным, поскольку исходная информация при таком преобразовании не искажается.

Тип	Возможно безопасное преобразование к типу…
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64, Double, Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

Некоторые расширяющие преобразования типа могут привести к потере точности. Следующая таблица описывает варианты преобразований, которые иногда приводят к потере информации.

Тип	Возможна потеря точности при преобразовании к типу…
Int32	Single
UInt32	Single
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

Сужающее преобразование - значение одного типа преобразуется к значению другого типа, которое имеет меньший размер (из 64-разрядного в 32-разрядное). Такое преобразование потенциально опасно потерей значения.

Сужающие преобразования могут приводить к потере информации. Если тип, к которому осуществляется преобразование, не может правильно передать значение источника, то результат преобразования оказывается равен константе PositiveInfinity или NegativeInfinity.

При этом значение PositiveInfinity интерпретируется как результат деления положительного числа на ноль, а значение NegativeInfinity интерпретируется как результат деления отрицательного числа на ноль.

Если сужающее преобразование обеспечивается методами класса System.Convert, то потеря информации сопровождается генерацией исключения (об этом позже).

Тип	Возможна потеря значения и генерация исключения при преобразовании в:
Byte	Sbyte
SByte	Byte, UInt16, UInt32, UInt64
Int16	Byte, SByte, UInt16
UInt16	Byte, SByte, Int16
Int32	Byte, SByte, Int16, UInt16, UInt32
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32, UInt32, UInt64
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

Особенности выполнения арифметических операций

Особенности выполнения операций над целочисленными операндами и операндами с плавающей точкой связаны с особенностями выполнения арифметических операций и с ограниченной точностью переменных типа float и double.

Представление величин:

float – 7 значащих цифр.

double – 16 значащих цифр.

1000000*100000==1000000000000, но максимально допустимое положительное значение для типа System.Int32 составляет 2147483647. В результате переполнения получается неверный результат -727379968.

Ограниченная точность значений типа System.Single проявляется при присвоении значений переменной типа System.Double. Приводимый ниже простой программный код иллюстрирует некоторые особенности арифметики .NET.

```
using System;

class Class1
{
const double epsilon = 0.00001D;
static void Main(string[] args)
{
int valI = 1000000, resI;
resI = (valI*valI)/valI;

// -727379968/1000000 == -727
Console.WriteLine("The result of action (1000000*1000000/1000000) is {0}", resI);

float valF00 = 0.2F, resF;
double valD00 = 0.2D, resD;

// Тест на количество значащих цифр для значений типа double и float.
resD = 12345678901234567890; Console.WriteLine(">>>>> {0:F10}",resD);
resF = (float)resD; Console.WriteLine(">>>>> {0:F10}",resF);
resD = (double)(valF00 + valF00); // 0.400000005960464
if (resD == 0.4D) Console.WriteLine("Yes! {0}",resD);
else Console.WriteLine("No! {0}",resD);

resF = valF00*5.0F;
resD = valD00*5.0D;

resF = (float)valD00*5.0F;
resD = valF00*5.0D; //1.0000000149011612

if (resD == 1.0D) Console.WriteLine("Yes! {0}",resD);
else              Console.WriteLine("No! {0}",resD);

resF = valF00*5.0F;
resD = valF00*5.0F; //1.0000000149011612

if (resD.Equals(1.0D))  Console.WriteLine("Yes! {0}",resD);
else                    Console.WriteLine("No! {0}",resD);

if (Math.Abs(resD - 1.0D) < epsilon)
      Console.WriteLine("Yes! {0:F7}, {1:F7}",resD - 1.0D, epsilon);
else
      Console.WriteLine("No! {0:F7}, {1:F7}",resD - 1.0D, epsilon);
}
}
```

Особенности арифметики с плавающей точкой

- Если переменной типа float присвоить величину x из интервала
  -1.5E-45 < x < 1.5E-45 (x != 0),

результатом операции окажется положительный (x > 0) или отрицательный (x < 0) нуль (+0, -0),

- Если переменной типа double присвоить величину x из интервала -5E-324 < x < 5E-324 (x != 0), результатом операции окажется положительный (x > 0) или отрицательный (x < 0) нуль (+0, -0),
- Если переменной типа float присвоить величину x, которая -3.4E+38 > x или x < 3.4E+38, результатом операции окажется положительная (x > 0) или отрицательная (x < 0) бесконечность (+Infinity, - Infinity),
- Если переменной типа double присвоить величину x, для которой -1.7E+308 > x или x < 1.7E+308, результатом операции окажется положительная (x > 0) или отрицательная (x < 0) бесконечность (+Infinity, - Infinity),
- Выполнение операции деления над значениями типами с плавающей точкой (0.0/0.0) даёт NaN (Not a Number).

Константное выражение

Константное выражение – это либо элементарное константное выражение, к которым относятся:

- символьный литерал,
- целочисленный литерал,
- символьная константа,
- целочисленная константа,

либо выражение, построенное на основе элементарных константных выражений с использованием скобок и символов операций, определённых на множестве значений данного типа.

Отличительные черты константного выражения:

- значение константного выражения не меняется при выполнении программы,
- значение константного выражения становится известно на этапе компиляции модуля, до начала выполнения модуля.

Переменные элементарных типов. Объявление и инициализация

Объявление – это предложение языка C#, которое используется непосредственно в теле класса для объявления членов класса (в этом случае объявлению может предшествовать спецификатор доступа) или для объявления переменных в конструкторах и методах класса.

Выполнение оператора объявления переменной элементарного типа в методе класса приводит к созданию в памяти (в стеке) объекта соответствующего типа, возможно проинициализированного определённым значением. Это значение может быть задано в виде литерала соответствующего типа, либо в виде выражения (синтаксис выражений рассматривается далее).

Предложение объявления предполагает (возможное) наличие различных спецификаторов, указание имени типа, имени объекта и (возможно) выражения инициализации. При этом имя типа может быть задано как Действительное Имя Типа (Имя FCL типа), либо как псевдоним типа (имя типа, как оно объявляется в C#). Соответствующее выражение инициализации может быть представлено литералом, либо выражением более сложной структуры.

Эквивалентные формы записи операторов определения переменных элементарных размерных типов:

```
int a;
System.Int32 a;
```

Эквивалентные формы записи операторов определения и инициализации переменных элементарных размерных типов:

```
int a = 0;
int a = new int();
System.Int32 a = 0;
System.Int32 a = new System.Int32();
```

Здесь следует учитывать важное обстоятельство! CLR не допускает использования в выражениях неинициализированных локальных переменных. В C# к

таковым относятся переменные, объявленные в теле метода. Так что при разработке алгоритма следует обращать на это особое внимание.

```
int a; // Объявление a.
int b; // Объявление b.
b = 10; // Инициализация a.
a=b+b; // Инициализация b.
```

Константы

Объявляются с дополнительным спецификатором const. Требуют непосредственной инициализации. В данном примере инициализируется литералом 3.14.

```
const float Pi = 3.14;
```

Перечисления

Перечисление объявляется с помощью ключевого слова enum, идентифицируется по имени и представляет собой непустой список неизменяемых именованных значений интегрального типа. Первое значение в перечислении по умолчанию инициализируется нулём. Каждое последующее значение отличается от предыдущего по крайней мере на единицу, если объявление значения не содержит явного дополнительного присвоения нового значения. Пример объявления перечисления приводится ниже.

```
enum Colors { Red = 1, Green = 2, Blue = 4, Yellow = 8 };
```

Обращение к элементу перечисления осуществляется посредством сложного выражения, состоящего из имени класса перечисления, операции доступа к элементу перечисления ‘.’, имени элемента перечисления:

```
int xVal = Colors.Red; //Переменная xVal инициализируется значением перечисления.
```

Перечисление является классом, а это означает, что в распоряжении программиста оказываются методы сравнения значений перечисления, методы преобразования значений перечисления в строковое представление, методы перевода строкового представления значения в перечисление, а также (судя по документации) средства для создания объектов-представителей класса перечисления.

Далее приводится список членов класса перечисления.

Открытые методы

CompareTo	Сравнивает этот экземпляр с заданным объектом и возвращает сведения об их относительных значениях.
Equals	Переопределен. Возвращает значение, показывающее, равен ли данный экземпляр заданному объекту.
Format	Статический. Преобразует указанное значение заданного перечисляемого типа в эквивалентное строчное представление в соответствии с заданным форматом.
GetHashCode	Переопределен. Возвращает хеш-код для этого экземпляра.
GetName	Статический. Выводит имя константы в указанном перечислении, имеющем заданное значение.
GetNames	Статический. Выводит массив имен констант в указанном перечислении.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
GetTypeCode	Возвращает базовый тип TypeCode для этого экземпляра.
GetUnderlyingType	Статический. Возвращает базовый тип указанного перечисления.
GetValues	Статический. Выводит массив значений констант в указанном перечислении.
IsDefined	Статический. Возвращает признак наличия константы с указанным значением в заданном

	перечислении.
Parse	Статический. Перегружен. Преобразует строковое представление имени или числового значения одной или нескольких перечисляемых констант в эквивалентный перечисляемый объект.
ToObject	Статический. Перегружен. Возвращает экземпляр указанного типа перечисления, равный заданному значению.
ToString	Перегружен. Переопределен. Преобразует значение этого экземпляра в эквивалентное ему строковое представление.

Защищенные конструкторы

Enum - конструктор	[Поставка ожидается.] Во как!

Защищенные методы

Finalize (унаследовано от Object)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

Объявление переменных. Область видимости и время жизни

Любая используемая в программе сущность вводится объявлением.

Объявлению подлежат:

- классы и структуры. Класс (структура) может содержать вложенные объявления других классов и структур,
- перечисления,
- объекты - переменные и константы, представляющие классы и структуры. Корректное объявление объекта предполагает, что информация, содержащая характеристики объявляемого объекта доступна транслятору.
- элементы перечислений – объекты, представляющие перечисления,
- конструкторы классов и методы (функции) – члены классов (в том числе и специальные).

Объявляемой сущности присваивается имя. Обращение к ранее объявленным сущностям в программе обеспечивается различными вариантами имён. Имена в программе повсюду и главная проблема заключается в том, чтобы не допустить неоднозначности при обращении из разных мест программы к классам, членам классов, перечислениям, переменным и константам.

Конфликта имён (проблемы с обращением к одноименным объектам) позволяет избежать принятая в языке дисциплина именования. В каждом языке программирования она своя.

Избежать конфликта имён в C# позволяет такая синтаксическая конструкция как блок операторов. Блок – это (возможно пустое) множество предложений, заключённое в фигурные скобки.

Различается несколько категорий блоков:

- тело класса (структуры). Место объявления членов класса,
- тело метода. Место расположения операторов метода,
- блок в теле метода.

Переменные можно объявлять в любом месте блока. Точка объявления переменной в буквальном смысле соответствует месту её создания. Обращение к объявляемой сущности (переменной или константе) “выше” точки её объявления лишено смысла.

Новый блок – новая область видимости. Объекты, объявляемые во внутренних блоках не видны во внешних блоках. Объекты, объявленные в методе и во внешних блоках видны и во внутренних блоках. Одноимённые объекты во вложенных областях конфликтуют.

Объекты, объявляемые в блоках одного уровня вложенности в методе не видны друг для друга. Конфликта имён не происходит.

В теле класса не допускается объявления одноимённых данных-членов. Нарушение этого правила приводит к неоднозначности.

В теле класса не допускается объявления одноимённых функций-членов (методов класса) с пустыми списками параметров и с совпадающим порядком расположения типа параметра в списках параметров (с сигнатурой метода). Здесь не важны имена параметров. Важно, что выражение вызова метода в этом случае будет неоднозначным.

Имена данных-членов класса не конфликтуют с одноименными переменными, объявляемыми в теле методов, поскольку в теле метода обращение к членам класса обеспечивается выражениями с операцией доступа и никакого конфликта в принципе быть не может.

Ниже приводится простой программный код, отражающий особенности принятой в C# концепции областей видимости.

```
using System;
class Class1
{
static int s;

static void Main(string[] args)
 {
{int s;}
{int s;}

 {
Class1.s = 100; // Классифицированное имя!
int s;
 {
//int s;
 }
 }

System.Int32 z = new int();
char a = (char)41;
char b = 'X';
Console.WriteLine("{0} {1}",a,b);
 }

//void QQQ(int q0)
//{
//Class1.s = 10;
//int s;
//}

//void QQQ(int q1)
//{
//Class1.s = 10;
//int s;
//}

int QQQ(int q2)
{
Class1.s = 10;
int s;
return 0;
}

int QQQ(int q2, int q3)
{
//s = 100;
Class1.s = 10;
int s;
return 0;
}
}
```

Управляющие операторы

Назначение управляющих операторов. Используются в рамках методов, задают логику выполнения программы, являются основным средством реализации алгоритма.

Различаются следующие категории управляющих операторов:

- операторы выбора. Вводятся ключевыми словами if, if … else …, switch,
- итеративные операторы. Вводятся ключевыми словами while, do … while, for, foreach,
- операторы перехода (в рамках методов). Вводятся ключевыми словами goto, break, continue.

if, if … else …

После ключевого слова if располагается взятое в круглые скобки условное выражение (булево выражение), следом за которым располагается оператор (блок операторов) произвольной сложности.

Далее в операторе if … else … после ключевого слова else размещается ещё один оператор.

В силу того, что в C# отсутствуют предопределённые алгоритмы преобразования значений к булевскому типу, в условное выражение должно быть выражением типа bool – переменной, константой, либо выражением на основе операций сравнения и логических операций.

В соответствии с синтаксисом условного оператора в части else (если таковая имеется) располагается блок операторов. Частный случай оператора – оператор if.

Какой бы сложности ни были составляющие части if…else… – это всего лишь ДВЕ равноправных части одного оператора. “Каскад” в if…else… – это всего лишь оператор (блок), содержащий вхождение if или if…else… операторов.

```
if (…)
 {

 }
else
 {
if (…)
 {

 }
else
 {

 }
 }
```

переписывается

```
if (…)
{

}
else
if (…)
{

}
else
{

}
```

Вложенный (Nesting) if. Оператор if часто сам в свою очередь является условным оператором произвольной сложности.

И в этом случае if…else… оператор включает ДВЕ части.

```
if (…)
 {
if (…)
 {

 }
```

```
else
 {

 }

 }
else
 {

 }
```

переписывается

```
if (…) if (…)
 {

 }
else
 {

 }
else
 {

 }
```

Главное - не перепутать соответствующие части if…else… оператора.

Ещё одно важное замечание связано с использованием “простых” (не блоков) операторов. Невозможно построить оператор if…else… на основе одиночного оператора объявления.

```
if (true) int XXX = 125;
if (true) int XXX = 125; else int ZZZ = 10;
```

Такие конструкции воспринимаются как ошибочные. Всё-таки в C# одиночный оператор это не блок, и ставить в зависимость от условия (пусть даже всегда истинного) такое ответственное дело как создание объекта здесь не принято.

Совсем другое дело при работе с блоками операторов!

```
if (true) {int XXX = 125;}
if (true) {int XXX = 125;} else {int ZZZ = 10;}
```

Даже в таких примитивных блоках своя область видимости и создаваемые в них объекты никому не мешая существуют по своим собственным правилам.

switch

При описании синтаксиса оператора switch использована нотация Бэкуса-Наура (БНФ).

```
ОператорSWITCH ::= switch (switchВыражение){switchБлок}
switchВыражение ::= Выражение
```

switchВыражение может быть либо целочисленным, либо символьным. switchБлок не может быть пустым.

```
switchБлок ::= СписокCaseЭлементов
СписокCaseЭлементов ::= [СписокCaseЭлементов] CaseЭлемент
CaseЭлемент ::=
               СписокРазделённыхМеток [СписокОператоров] ОператорТерминатор
СписокРазделённыхМеток ::= [СписокРазделённыхМеток] Метка РазделительМетки
РазделительМетки ::= :
Метка ::= case КонстантноеВыражение | default
ОператорТерминатор ::= breakОператор | gotoОператор | return [Выражение];
breakОператор ::= break;
gotoОператор ::= goto Метка;
```

Таким образом, минимальный switch оператор имеет следующий вид:

```
int val;
::::::::::
switch (val)
{
default: break; // Список операторов пуст.
}
```

Более сложные образования:

```
int val;
::::::::::
switch (val)
{
case 0: break;
}
::::::::::
switch (val)
{
case 0: break;
default: break; // Список операторов пуст.
}
::::::::::
switch (val)
{
default: … break; // defaul БЛОКИРУЕТ выполнение операторов
// всех ниже лежащих CaseЭлементов.
case 100: … break;
}
::::::::::
switch (val)
{
default: // default БЛОКИРУЕТ выполнение операторов
// всех ниже лежащих CaseЭлементов, кроме операторов,
// входящих в CaseЭлемент под меткой 100.
case 100: … break;
case 1: … break;
case 10: … break;
}
```

Поскольку при выполнении модуля выбор списков операторов для выполнения в рамках CaseЭлемента определяется значением константных выражений в case метках, константные выражения ВСЕХ меток данного switchБлока должны различаться ПО СВОИМ ЗНАЧЕНИЯМ.

Следующий пример некорректен. Константные выражения в списках разделённых меток CaseЭлемента

```
case 1+1: case 2: … break;
```

различаются ПО ФОРМЕ (1+1 и 2), а НЕ ПО ЗНАЧЕНИЮ!

По тем же причинам в рамках switch оператора может быть не более одного вхождения метки default.

Списки операторов в CaseЭлементах НЕ могут включать операторы объявления. switchОператор строится на основе ОДНОГО блока, разделяемого метками на фрагменты, выполнение которых производится в зависимости от значений константных выражений в case метках. Разрешение объявления констант и переменных в CaseЭлементе означает риск обращения к ранее необъявленной переменной.

```
switch (val)
{
case 0:
int XXX = 100; // Нельзя!
break;
case 1:
XXX += 125;
break;
```

```
}
```

Каждый CaseЭлемент в обязательном порядке ЗАВЕРШАЕТСЯ оператором-терминатором, который является ОБЩИМ для ВСЕХ операторов данного CaseЭлемента:

```
int val, XXX;
:::::
switch (val)
{
case 0:
if (XXX == 25) {XXX *= -1; break;}
else XXX = 17;
goto default; // Общий терминатор.
case 1:
XXX += 125;
break; // Общий терминатор.
default:
return XXX; // Общий терминатор.
}
```

while

```
ОператорWHILE ::= while (УсловиеПродолжения) Оператор
УсловиеПродолжения ::= БулевоВыражение
Здесь опять же:
Оператор ::= Оператор
::= БлокОператоров
```

Правило выполнения этого итеративного опера состоит в следующем: сначала проверяется условие продолжения оператора и в случае, если значение условного выражения равно true, соответствующий оператор (блок операторов) выполняется.

Невозможно построить операторWHILE на основе одиночного оператора объявления. Оператор

```
while (true) int XXX = 0;
```

С самого первого момента своего существования (ещё до начала трансляции!) сопровождается предупреждением:

Embedded statement cannot be a declaration or labeled statement.

do … while

```
ОператорDOWHILE ::= do Оператор while (УсловиеПродолжения)
УсловиеПродолжения ::= БулевоВыражение
Оператор ::= Оператор
::= БлокОператоров
```

Разница с ранее рассмотренным оператором цикла состоит в том, что здесь сначала выполняется оператор (блок операторов), а затем проверяется условие продолжения оператора.

for

```
ОператорFOR ::=
for
([ВыраженияИнициализации];[УсловиеПродолжения];[ВыраженияШага])
                                                                Оператор
ВыраженияИнициализации ::= СписокВыражений
СписокВыражений ::= [СписокВыражений ,] Выражение
УсловиеПродолжения ::= БулевоВыражение
ВыраженияШага ::= [СписокВыражений ,] Выражение
```

Здесь опять же:

```
Оператор ::= Оператор
::= БлокОператоров
```

ВыраженияИнициализации, УсловиеПродолжения, ВыраженияШага в заголовке оператора цикла for могут быть пустыми. Однако наличие пары символов ‘;’ в заголовке цикла for обязательно.

Список выражений представляет собой разделённую запятыми последовательность выражений.

Следует иметь в виду, что оператор объявления также строится на основе списка выражений (выражений объявления), состоящих из спецификаторов типа, имён и, возможно, инициализаторов. Этот список завершается точкой с запятой, что позволяет рассматривать список выражений инициализации как самостоятельный оператор в составе оператора цикла for. При этом область видимости имён переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла. Это значит, что переменные, объявленные в операторе инициализации данного оператора цикла НЕ МОГУТ быть использованы непосредственно после оператора до конца блока, содержащего этот оператор. А следующие друг за другом в рамках общего блока операторы МОГУТ содержать в заголовках одни и те же выражения инициализации.

операторFOR также невозможно построить на основе одиночного оператора объявления.

foreach

```
ОператорFOREACH ::=
foreach (ОбъявлениеИтератора in ВыражениеIN) Оператор
ОбъявлениеИтератора ::= ИмяТипа Идентификатор
ВыражениеIN ::= Выражение
Оператор ::= Оператор
::= БлокОператоров
```

ИмяТипа – Обозначает тип итератора.

identifier – обозначает переменную которая представляет элемент коллекции.

ВыражениеIN – объект, представляющий массив или коллекцию.

Этим оператором обеспечивается повторение множества операторов, составляющих тело цикла, для каждого элемента массива или коллекции. После перебора ВСЕХ элементов массива или коллекции и применения множества операторов для каждого элемента массива или коллекции, управление передаётся следующему за ОператорFOREACH оператору (разумеется, если таковые имеются).

Область видимости имён переменных, определяемых этим оператором, распространяется только на операторы, относящиеся к данному оператору цикла.

```
int[] array = new int[10]; // Объявили и определили массив
foreach (int i in array) {/*:::::*/}; // Для каждого элемента массива надо сделать…
```

Специализированный оператор, приспособленный для работы с массивами и коллекциями. Обеспечивает повторение множества (единичного оператора или блока операторов) операторов для КАЖДОГО элемента массива или коллекции. Конструкция экзотическая и негибкая. Предполагает выполнение примитивных последовательностей действий над массивами и коллекциями (начальная инициализация или просмотр ФИКСИРОВАННОГО количества элементов). Действия, связанные с изменениями размеров и содержимого коллекций в рамках этого оператора могут привести к непредсказуемым результатам.

goto, break, continue

goto в операторе switch уже обсуждалось.

Второй вариант использования этого оператора непосредственно тело метода.

Объявляется метка (правильнее, оператор с меткой). Оператор может быть пустым. Метка – идентификатор отделяется от оператора двоеточием. В качестве дополнительного разделителя могут быть использованы пробелы, символы табуляции и перехода на новую строку. Метка, как и любое другое имя, подчиняется правилам областей видимости. Она видна в теле метода только в

одном направлении: из внутренних (вложенных) блоков. Поэтому оператор перехода

```
goto ИмяПомеченногоОператора;
```

позволяет ВЫХОДИТЬ из блоков, но не входить в них.

Вообще, об этом операторе всегда говорится очень много нехороших слов и его описание сопровождается рекомендациями к его НЕИСПОЛЬЗОВАНИЮ.

Операторы

```
break;
```

и

```
continue;
```

используются как вспомогательные средства управления в операторах цикла.

Методы

В C# методы определяются в рамках объявления класса. Методы (функции) являются членами класса и определяют функциональность объектов-членов класса (нестатические методы – методы объектов), и непосредственно функциональность самого класса (статические методы - методы класса) класса.

Метод может быть объявлен и метод может быть вызван. Поэтому различают объявление метода (метод объявляется в классе), вызов метода (выражение вызова метода располагается в теле метода).

Различают статические (со спецификатором static) и нестатические методы (объявляются без спецификатора).

Статические методы вызываются от имени класса, в котором они были объявлены. Считается, что статические методы определяют функциональность класса, а нестатические методы определяют поведение конкретных объектов-представителей класса и потому вызываются от имени (ссылки) объекта-представителя класса, содержащего объявление вызываемого метода.

Синтаксис объявления метода

```
ОбъявлениеМетода ::= ЗаголовокМетода ТелоМетода
ЗаголовокМетода ::= [СпецификаторМетода]
ТипВозвращаемогоЗначения
Имя
([СписокПараметров])

СпецификаторМетода ::= СпецификаторДоступности
::= new
::= static
::= virtual
::= sealed
::= override
::= abstract
::= extern

СпецификаторДоступности ::= public
::= private
::= protected
::= internal
::= protected internal

ТипВозвращаемогоЗначения ::= void | ИмяТипа
ТелоМетода ::= БлокОператоров
 ::= ;

Имя ::= Идентификатор
СписокПараметров ::= [СписокПараметров ,] ОбъявлениеПараметра
ОбъявлениеПараметра ::= [СпецификаторПередачи] ИмяТипа ИмяПараметра
::= [СпецификаторСписка] ИмяТипаСписка ИмяПараметра
```

```
СпецификаторПараметра ::= СпецификаторПередачи | СпецификаторСписка
СпецификаторПередачи ::= ref | out
СпецификаторСписка ::= params
ИмяТипаСписка ::= ИмяТипа[]
```

Тело метода может быть пустым! В этом случае за заголовком метода располагается точка с запятой.

Вызов метода

```
ВыражениеВызоваМетода ::= ИмяМетода([СписокПараметровВызова])
 ::= *****
СписокПараметровВызова ::= [СписокПараметровВызова ,] ПараметрВызова
ПараметрВызова ::= [СпецификаторПередачи] Выражение
```

Перегрузка методов

Имя и список типов параметров являются важной характеристикой метода и называются СИГНАТУРОЙ метода. В C# методы, объявляемые в классе, идентифицируются по сигнатуре метода.

Эта особенность языка позволяет объявлять в классе множество одноименных методов. Такие методы называются перегруженными, а деятельность по объявлению таких методов – перегрузкой.

При написании программного кода, содержащего ВЫРАЖЕНИЯ ВЫЗОВА переопределённых методов корректное соотнесение выражения вызова метода определяет метод, которому будет передано управление.

```
// Класс содержит объявление четырёх одноименных методов
// с различными списками параметров.
class C1
{
void Fx(float key1)
{
return;
}

int Fx(int key1)
{
return key1;
}

int Fx(int key1, int key2)
{
return key1;
}

int Fx(byte key1, int key2)
{
return (int)key1;
}

static void Main(string[] args)
{
C1 c = new C1();
// Нестатические методы вызываются от имени объекта c.
// Передача управления соответствующему методу
// обеспечивается явными преобразованиями к типу.
c.Fx(Convert.ToSingle(1));
c.Fx(3.14F);
c.Fx(1);
c.Fx(1,2);
c.Fx((byte)10, 125);
}
}
```

Информация о типе возвращаемого значения при этом не учитывается, поскольку в выражении вызова возвращаемое значение метода может не использоваться вовсе.

Способы передачи параметров при вызове метода

Известно два способа передачи параметров при вызове метода:

- по значению (в силу специфики механизма передачи параметров только входные),
- по ссылке (входные и/или выходные).

По значению – БЕЗ спецификаторов (для размерных типов этот способ предполагается по умолчанию). Локальная копия значения в методе. Параметр представляет собой означенную переменную. Его можно использовать в методе наряду с переменными, объявленными в теле метода. Изменение значения параметра НЕ влияет на значение параметра в выражении вызова.

Для организации передачи по ссылке параметра РАЗМЕРНОГО типа требуется явная спецификация ref или out. По ссылке – спецификатор ref (для ссылочных типов предполагается по умолчанию (другого способа передачи параметра для ссылочных типов просто нет), для размерных типов спецификатор необходимо явно указывать).

Параметр по ссылке и параметр по значению – большая разница! Это основание для перегрузки метода!

```
using System;

class XXX {
public int mmm;
 }

class Class1
 {//=============================================
static int i;
static void f1 (ref int x)
 {
x = 125;
 }

static void f1 (int x)
 {
x = 0;
 }

static void f1 (XXX par)
 {
par.mmm = 125;
 }

static void f1 (out XXX par)
 {
par = new XXX(); // Ссылка на out XXX par ДОЛЖНА быть
// проинициализирована в теле
// метода НЕПОСРЕДСТВЕННО перед обращением к ней!
// Способ инициализации - любой! В том числе и созданием объекта!
// А можно и присвоением в качестве значения какой-либо другой ссылки.
par.mmm = 125;
 }

// Для размерного типа параметра метода со списками параметров, которые
// различаются только спецификаторами out - ref
// несовместимы. out – частный случай ссылки.
//static void f1 (out int x)
 //{
// x = 125;
 //}

static void Main(string[] args)
 {//====================================
int a = 0;
f1(ref a);
//f1(out a);
f1(a);
```

```
XXX xxx = new XXX();
xxx.mmm = 0;
f1(xxx);
f1(ref xxx);
// По возвращении из метода это уже другая ссылка!
// Под именем xxx – другой объект.

 }//=====================================
}//=============================================
```

Передача параметров. Ссылка и ссылка на ссылку как параметры

Параметры ВСЕГДА передаются по значению. Это означает, что в области активации создаётся копия ЗНАЧЕНИЯ параметра. При этом важно, ЧТО копируется в эту область. Для ссылочных типов возможны два варианта:

- можно скопировать адрес объекта (ссылка как параметр),
- можно скопировать адрес переменной, которая указывает на объект (ссылка на ссылку как параметр).

В первом случае параметр обеспечивает изменение полей объекта. Непосредственное изменение значения этой ссылки (возможно в результате создания новых объектов и присвоения значения новой ссылки параметру) означает лишь изменение значения параметра, который в данном случае играет роль обычной переменной, сохраняющей значение какого-то адреса (адреса ранее объявленного объекта).

Во втором случае параметр сохраняет адрес переменной, объявленной в теле вызывающего метода. При этом в вызываемом методе появляется возможность как изменения значений полей объекта, так и непосредственного изменения значения переменной.

Следующий программный код демонстрирует специфику передачи параметров.

```
using System;
namespace Ref_RefRef
{
/// <summary>
/// Ссылка и ссылка на ссылку.
/// </summary>

class WorkClass
{
public int x;
public WorkClass()
 {
x = 0;
 }

public WorkClass(int key)
 {
x = key;
 }

public WorkClass(WorkClass wKey)
 {
x = wKey.x;
 }
}

class ClassRef
{
static void Main(string[] args)
 {
WorkClass w0 = new WorkClass();
Console.WriteLine("on start: {0}", w0.x); //_: 0
f0(w0);
Console.WriteLine("after f0: {0}", w0.x); //0: 1
f1(w0);
Console.WriteLine("after f1: {0}", w0.x); //1: 1
f2(ref w0);
Console.WriteLine("after f2: {0}", w0.x); //2: 10
f3(ref w0);
```

```
Console.WriteLine("after f3: {0}", w0.x); //3: 3

// Ещё один объект...
WorkClass w1 = new WorkClass(w0);
ff(w0, ref w1);

if (w0 == null) Console.WriteLine("w0 == null");
else Console.WriteLine("w0 != null"); // !!!

if (w1 == null) Console.WriteLine("w1 == null"); // !!!
else Console.WriteLine("w1 != null");

}

static void f0(WorkClass wKey)
 {
wKey.x = 1;
Console.WriteLine(" in f0: {0}", wKey.x);
 }

static void f1(WorkClass wKey)
 {
wKey = new WorkClass(2);
Console.WriteLine(" in f1: {0}", wKey.x);
 }

static void f2(ref WorkClass wKey)
 {
wKey.x = 10;
Console.WriteLine(" in f2: {0}", wKey.x);
 }

static void f3(ref WorkClass wKey)
 {
wKey = new WorkClass(3);
Console.WriteLine(" in f3: {0}", wKey.x);
 }

static void ff(WorkClass wKey, ref WorkClass refKey)
 {
wKey = null;
refKey = null;
 }

 }
}
```

Сравнение значений ссылок

Имеют место быть следующие варианты:

- Ссылка может быть пустой (ref0 == null || ref1 != null),
- Разные ссылки могут быть настроены на разные объекты (ref0 != ref1),
- Разные ссылки могут быть настроены на один объект (ref0 == ref1).
- Четвёртого не дано (больше-меньше в условиях управляемой памяти) не имеет никакого смысла.

this в нестатическом методе

Первичное выражение this в теле нестатического метода ссылается на объект, "от имени" которого был произведён вызов данного метода. Если метод возвращает ссылку, this можно использовать в качестве возвращаемого значения оператора return.

Свойства

Объявляемые в классе данные-члены обычно используются как переменные. Статические члены сохраняют значения, актуальные для всех объектов-представителей класса. Нестатические данные-члены сохраняют в переменных объекта информацию, актуальную для данного объекта.

Обращение к этим переменным производится с использованием точечной нотации, с явным указанием данного-члена, предназначенного для сохранения (или получения) значения.

В отличие от переменных, свойства не указывают конкретные места хранения. Вместо этого в свойствах используются блоки операторов, обеспечивающие доступ к данным-членам для чтения и записи значений.

Для задания свойств в языке C# используется специальный синтаксис и предполагает описание способов получения и установки значения — они называются get accessor и set accessor.

Наличие accessor'ов определяет доступность свойства для чтения и записи. При обращении к значению свойства вызывается механизм чтения (get accessor), при изменении значения вызывается механизм записи (set accessor).

```
class TestClass
{
int xVal; // Переменная объекта.

// Свойство, обслуживающее переменную объекта.
// Предоставляет возможность чтения и записи значений
// поля xVal.
public int Xval
{
 // Эти значения реализованы в виде блоков программного кода,
 // обеспечивающих получение и изменение значения поля.
 // get accessor.
 get
 {
  // Здесь можно расположить любой код.
  // Он будет выполняться после обращения к свойству для
  // прочтения значения.
  // Например, так. Здесь получается,
  // что возвращаемое значение зависит от количества
       // обращений к свойству.
       xVal++;
       return xVal;
      }
      set
      {
       // set accessor.
       // Здесь можно расположить любой код.
       // Он будет выполняться после обращения к свойству для
       // записи значения.
       xVal = value;
      }
}
}

class Class1
{
static void Main(string[] args)
{
    // Создали объект X.
    TestClass X = new TestClass();
    // Обратились к свойству для записи в поле Xval
    // значения. Обращение к свойству располагается
    // СЛЕВА от операции присвоения. В свойстве Xval
    // будет активизирован блок кода set.
    X.Xval = 100;
    // Обратились к свойству для чтения из поля Xval
    // значения. Обращение к свойству располагается
    // СПРАВА от операции присвоения. В свойстве Xval
    // будет активизирован блок кода get.
    int q = X.Xval;
}
}
```

Поскольку объект для доступа к данным Get концептуально равнозначен чтению значения переменной, хорошим стилем программирования считается отсутствие заметных побочных эффектов при использовании объектов для доступа к данным Get.

Значение свойства не должно зависеть от каких-либо обстоятельств, в частности, от количества обращений к объекту. Если доступ к свойству порождает (как в нашем случае) заметный побочный эффект, свойство, согласно рекомендациям по соблюдению "хорошего стиля", желательно реализовывать как метод.

Обработка исключений

Пусть в классе объявляются методы A и B.

При этом из метода A вызывается метод B, который выполняет свою работу, возможно, возвращает результаты. В теле метода A есть точка вызова метода B, и точка возврата, в которой оказывается управление после успешного возвращения из метода B.

Если всё хорошо, метод A, возможно, анализирует полученные результаты и продолжает свою работу непосредственно из точки возврата.

Если при выполнении метода B возникла исключительная ситуация (например, целочисленное деление на 0), возможно, что метод A узнает об этом, анализируя возвращаемое из B значение. Таким может быть один из сценариев "обратной связи", при котором вызывающий метод узнаёт о результатах деятельности вызывающего метода.

Недостатки этого сценария заключаются в том, что:

- метод B может в принципе не возвращать никаких значений,
- среди множества возвращаемых методом B значений невозможно выделить подмножество значений, которые можно было бы воспринимать как уведомление об ошибке,
- работа по подготовке уведомления об ошибке требует неоправданно больших усилий.

Решение проблемы состоит в том, что в среде выполнения поддерживается модель обработки исключений, основанная на понятиях объектов исключения и защищенных блоков кода. Следует отметить, что схеме обработки исключений не нова и успешно реализована во многих языках и системах программирования.

Некорректная ситуация в ходе выполнения программы (деление на нуль, выход за пределы массива) рассматривается как исключительная ситуация, на которую метод, в котором она произошла, реагирует ГЕНЕРАЦИЕЙ ИСКЛЮЧЕНИЯ, а не обычным возвращением значения, пусть даже изначально ассоциированного с ошибкой.

Среда выполнения создает объект для представления исключения при его возникновении. Одновременно с этим прерывается обычный ход выполнения программы. Происходит так называемое разматывание стека, при котором управление НЕ оказывается в точке возврата и если ни в одном из методов, предшествующих вызову, не было предпринято предварительных усилий по ПЕРЕХВАТУ ИСКЛЮЧЕНИЯ, приложение аварийно завершается.

Можно писать код, обеспечивающий корректный перехват исключений, можно создать собственные классы исключений, получив производные классы из соответствующего базового исключения.

Все языки программирования, использующие среду выполнения, обрабатывают исключения одинаково. В каждом языке используется форма try/catch/finally для структурированной обработки исключений.

Следующий пример демонстрирует основные принципы организации генераторов и перехватчиков исключений.

```
using System;

// Объявление собственного исключения.
// Наследуется базовый класс Exception.
public class xException:Exception
{
// Собственное исключение имеет специальную строчку
// и в этом её отличие.
public string xMessage;
```

```
// Кроме того, в поле её базового элемента
// также фиксируется особое сообщение.
public xException(string str):base("xException is here...")
{
       xMessage = str;
}

}

// Объявление собственного исключения.
// Наследуется базовый класс Exception.
public class MyException:Exception
{
// Собственное исключение имеет специальную строчку
// и в этом её отличие.
public string MyMessage;

// Кроме того, в поле её базового элемента
// также фиксируется особое сообщение.
public MyException(string str):base("MyException is here...")
{
    MyMessage = str;
}
}

public class StupidCalcule
{
public int Div(int x1, int x2)
{
// Вот здесь метод проверяет корректность операндов и с помощью оператора
// throw возбуждает исключение.
 if (x1 != 0 && x2 == 0)
    throw new Exception("message from Exception: Incorrect x2!");
 else if (x1 == 0 && x2 == 0)
    throw new MyException("message from MyException: Incorrect x1 && x2!");
 else if (x1 == -1 && x2 == -1)
    throw new xException("message from xException: @#$%*&^???");

 // Если же ВСЁ ХОРОШО, счастливый заказчик получит ожидаемый результат.
 return (int)(x1/x2);
}
}


public class XXX
{

public static void Main()
{
int ret;
StupidCalcule sc = new StupidCalcule();

// Наше приложение специально подготовлено к обработке исключений!
// Потенциально опасное место (КРИТИЧЕСКАЯ СЕКЦИЯ) ограждено
//     (заключено в блок try)
try
{
 // Вызов...
 ret = sc.Div(-1,-1);
 // Если ВСЁ ХОРОШО - будет выполнен оператор Console.WriteLine.
 // Потом операторы блока finally, затем операторы за пределами
 // пределами блоков try, catch, finally.
 Console.WriteLine("OK, ret == {0}.", ret.ToString());
}
catch (MyException e)
{
 // Здесь перехватывается MyException.
 Console.WriteLine((Exception)e);
 Console.WriteLine(e.MyMessage);
}
// Если этот блок будет закомментирован -
```

```
// возможные исключения типа Exception и xException
// окажутся неперехваченными. И после блока
// finally приложение аварийно завершится.
catch (Exception e)
{
 // А перехватчика xException у нас нет!
 // Поэтому в этом блоке будут перехватываться
 // все ранее неперехваченные потомки исключения Exception.
 // Это последний рубеж.
 // Ещё один вариант построения последнего рубежа
 // выглядит так:
 // catch
 //{
 //  Операторы обработки - сюда.
 //}
     Console.WriteLine(e);
}
finally
{
  // Операторы в блоке finally выполняются ВСЕГДА, тогда как
  // операторы, расположенные за пределами блоков try, catch, finally,
  // могут и не выполниться вовсе.
  Console.WriteLine("finally block is here!");
}
// Вот если блоки перехвата исключения окажутся не
// соответствующими возникшему исключению - нормальное
// выполнение приложения будет прервано - и мы никогда не увидим
// этой надписи.
Console.WriteLine("Good Bye!");
}
}
```

Массив. Объявление

Массив – множество однотипных элементов. Это тоже ТИП. Любой массив наследует классу (является производным от класса – о принципе наследования позже) System.Array.

Существует несколько способов сознания группировок однотипных объектов:

- объявление множества однотипных элементов в рамках перечисления (класса, структуры),
- определение собственно массива.

Принципиальная разница состоит в следующем:

- доступ к данным-членам перечисления, класса, массива производится ПО имени данного-члена (элементы перечисления, класса или структуры ИНОГДА могут быть одного типа, но каждый член всегда имеет собственное имя),
- доступ к элементу массива осуществляется по индексу (элементы массива ВСЕГДА однотипны, располагаются в contiguous memory) при этом допускается случайный доступ.
-

Многомерные массивы. Массив размерности (или ранга) N (N определяет число измерений массива) – это Массив массивов (или составляющих массива) ранга N-1. Составляющие массива – это массивы меньшей размерности, являющиеся элементами данного массива. Составляющая массива – это либо массив, либо элемент массива.

```
ОбъявлениеМассива ::=
ИмяТипа СписокСпецификаторовРазмерности ИмяМассива [ИнициализацияМассива];
ИмяТипа ::= Идентификатор
ИмяМассива ::= Идентификатор

СписокСпецификаторовРазмерности
::= [СписокСпецификаторовРазмерности] СпецификаторРазмерности
::= [СписокНеявныхСпецификаторов]
СпецификаторРазмерности ::= []
СписокНеявныхСпецификаторов
СписокНеявныхСпецификаторов ::= [СписокНеявныхСпецификаторов ,] НеявныйСпецификатор
НеявныйСпецификатор ::= ПУСТО | РАЗДЕЛИТЕЛЬ
```

ПУСТО – оно и есть пусто.

РАЗДЕЛИТЕЛЬ – пробел, несколько пробелов, символ табуляции, символ перехода на новую строку, комбинация символов “новая строка/возврат каретки” и прочая икебана…

При объявлении массива действуют следующие правила:

- Спецификатор размерности, состоящий из одного неявного спецификатора [] специфицирует составляющую массива размерности 1.
- Спецификатор размерности, состоящий из N неявных спецификаторов [,,, … ,] специфицирует составляющую массива размерности N.
- Длина списка спецификаторов размерности массива не огранена.

При этом информация о типе составляющих массива в объявлении массива определяется на основе типа массива и списка его спецификаторов размерности.

Синтаксис объявления массива (ссылки на массив) позволяет специфицировать массивы произвольной конфигурации без какого-либо намёка на количественные характеристики составляющих массивы элементов.

Ниже представлены способы ОБЪЯВЛЕНИЯ ссылок на массивы РАЗЛИЧНОЙ размерности и конфигурации.

```
// Объявлены ссылки на массивы размерности 3 элементов типа int.
// Это массивы составляющих, представляющих собой массивы элементов
// размерности 2 одномерных массивов элементов типа int.
// Размеры всех составляющих массивов одного уровня равны
// между собой (так называемые “прямоугольные” массивы).
int[,,] arr0;
int[   ,      ,      ] arr1;
int[
,
,
] arr2;
// Объявлена ссылка на
// ОДНОМЕРНЫЙ(!) массив
// ОДНОМЕРНЫХ(!) элементов массива, каждый из которых является
// ОДНОМЕРНЫМ(!) массивом элементов типа int.
int[][][] arr3;
// Объявлена ссылка на
// ОДНОМЕРНЫЙ(!) массив составляющих, каждая из которых является
// ДВУМЕРНЫМ(!) массивом массивов элементов типа int.
// При этом никаких ограничений на размеры “прямоугольных” составляющих
// данное объявление не содержит. У всех составляющих могут быть разные
// размеры.
int[][,] arr4;
// Объявлена ссылка на
// ДВУМЕРНЫЙ(!) массив составляющих, каждая из которых является
// ОДНОМЕРНЫМ(!) массивом элементов типа int.
// При этом никаких ограничений на размеры одномерных составляющих
// данное объявление не содержит. У всех составляющих могут быть разные
// размеры.
int[,][] arr5;
```

Рассмотренный синтаксис объявления и инициализации массива позволяет определять ДВЕ различных категории массивов:

- простые (прямоугольные) массивы,
- jagged (зубчатый, зазубренный; неровно оторванный; пьяный; находящийся под влиянием наркотиков) массивы.

Особенности инициализации и использования массивов разных категорий рассматриваются дальше.

Инициализация массивов

Массив мало объявить, его нужно ещё проинициализировать. Только тогда ранее специфицированное множество однотипных объектов будет размещено в памяти.

Определение предполагает спецификацию КОЛИЧЕСТВА объектов по каждому измерению.

```
ИнициализацияМассива ::= = newИнициализация
::= = ИнициализацияСписком
```

```
::= = ЗаполняющаяИнициализация
::= = ИнициализацияКопированием

newИнициализация ::= new ИмяТипа СписокОписателейРазмерности
ЗаполняющаяИнициализация ::= newИнициализация ИнициализацияСписком

СписокОписателейРазмерности ::= [СписокОписателейРазмерности] ОписательРазмерности
ОписательРазмерности ::= [СписокОписателей] | ПустойОписатель
СписокОписателей ::= [СписокОписателей ,] Описатель
Описатель ::= Выражение
ПустойОписатель ::= [ПУСТО]
```

Выражение-описатель должно быть:

- целочисленным,
- с определённым значением.

Такой фокус при определении и инициализации массивов не проходит:

```
int x; // К моменту определения массива значение x должно быть определено!
int[] arr0 = new int[x];
```

Так нормально! В момент определения массива CLR знает ВСЕ НЕОБХОДИМЫЕ характеристики определяемого массива.

```
int x = 10;
int[][] arr1 = new int[125][x];
```

Естественно, общее количество описателей размерности должно соответствовать количеству неявных спецификаторов.

ВСЕ НЕОБХОДИМЫЕ характеристики – это не всегда ВСЕ!

При определении массивов произвольной размерности инициализирующая запись должна содержать ПОЛНУЮ информацию о характеристиках первой составляющей массива. Все остальные описатели могут оставаться пустыми.

```
int val1 = 100;
// На этом этапе определения массива массивов
// элементов типа int принципиально знание характеристик первой составляющей!
// Все остальные могут быть заданы после!
int [][] x0 = new int[15][];
int [][][] x1 = new int[val1][][];
int [,][] x2 = new int[val1,7][];
int [,,][,][] x3 = new int[2,val1,7][,][];

// Следующие способы объявления корректными не являются.
// int [][][] y0 = new int[val1][2][];
// int [][][] y1 = new int[][2][7];
// int [,][] y2 = new int[ ,2][7];
// int [,,] y3 = new int[val1,2, ];
```

При такой форме определения массива предполагается многоступенчатая инициализация, при которой производится последовательная инициализация составляющих массива.

```
ИнициализацияСписком ::= {СписокВыражений}
::= {СписокИнициализаторовСоставляющихМассива}
СписокИнициализаторовСоставляющихМассива ::=
[СписокИнициализаторовСоставляющихМассива , ] ИнициализаторМассива
ИнициализаторМассива ::= newИнициализация {СписокВыражений} | {СписокВыражений}
СписокВыражений ::= [СписокВыражений ,] Выражение
```

При инициализации списком каждый элемент массива (или составляющая массива) получает собственное значение из списка.

```
int[] r = {val1, 1, 4, 1*val1};
```

Избыточная инициализация с дополнительной возможностью уточнения характеристик массива.

```
int[] r = new int[] {val1, 1, 4, 1*val1};
int[] t = new int[4] {val1, 1, 4, 1*val1};
```

В первом случае избыточная инициализация – рецидив многословия. Тип значений элементов массива определяется спецификатором массива, количество элементов массива определяется количеством элементов списка выражений.

Во втором случае избыточная инициализация – способ дополнительного контроля размеров массива. Транслятор сосчитает количество элементов списка и сопоставит его со значением описателя.

Инициализация многомерных массивов – дело ответственное, которое требует особого внимания и не всегда может быть компактно наглядно представлено одним оператором.

При этом само объявление оператора может содержать лишь частичную инициализацию, а может и не содержать её вовсе. Делу инициализации одного массива может быть посвящено множество операторов. В этом случае инициализирующие операторы строятся на основе операций присвоения. Главное при этом – не выходить за пределы характеристик массива, заданных при объявлении массива.

```
ИнициализацияКопированием ::= СпецификацияСоставляющей
СпецификацияСоставляющей ::= ИмяМассива СписокИндексныхВыражений
СписокИндексныхВыражений ::= [СписокИндексныхВыражений] [ИндексноеВыражение]
ИндексноеВыражение ::= Выражение
```

В силу того, что массив является объектом ссылочного типа, составляющие одного массива могут быть использованы для инициализации другого массива. Ответственность за возможные переплетения ссылок возлагается на программиста.

```
int [] q = {0,1,2,3,4,5,6,7,8,9};
int [][] d1 = {
new int[3],
new int[5]
       };
int [][][] d2 = new int[2][][];
d2[0] = new int[50][]; d2[0][0] = d1[0];
// d2[1][0] ссылается на составляющую массива d1.
d2[0][1] = new int[125];
d2[1] = new int[50][];
d2[1][1] = new int[10]{1,2,3,4,5,6,7,8,9,10};
d2[1][0] = q;
// d2[1][0] ссылается на ранее объявленный и определённый массив q.
```

Примеры инициализации массивов

Одномерный массив из int.

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

Этот массив содержит элементы от myArray[0] до myArray[4].

Операция new используется для создания массива и инициализации его элементов предопределёнными значениями. В результате выполнения этого оператора все элементы массива будут установлены в ноль.

Простой строковый массив можно объявить и проинициализировать аналогичным образом:

```
string[] myStringArray = new string[6];
```

Пример совмещения явной инициализации элементов массива с его объявлением. При этом спецификатора размерности не требуется, поскольку соответствующая информация может быть получена непосредственно из инициализирующего списка. Например:

```
int[] myArray = new int[] {1, 3, 5, 7, 9};
```

Строковый массив может быть проинициализирован аналогичным образом:

```
string[] weekDays = new string[] {”Sun”,”Sat”,”Mon”,”Tue”,”Wed”,”Thu”,”Fri”};
```

И это не единственный способ объявления и инициализации.

Если объявление совмещается с инициализацией, операция new может быть опущена. Предполагается, что транслятор знает, что при этом нужно сделать:

```
string[] weekDays = {”Sun”,”Sat”,”Mon”,”Tue”,”Wed”,”Thu”,”Fri”};
```

Объявление и инициализация вообще могут быть размещены в разных операторах. Но в этом случае без операции new ничего не получится:

```
int[] myArray;
myArray = new int[] {1, 3, 5, 7, 9}; // Так можно.
myArray = {1, 3, 5, 7, 9}; // Так нельзя.
```

Два типа массивов: Value Type and Reference Type

Рассмотрим следующее объявление:

```
MyType[] myArray = new MyType[10];
```

Результат выполнения этого оператора зависит от того, что собой представляет тип MyType.

Возможны всего два варианта: MyType может быть размерным или ссылочным типом.

Если это тип размерный, результатом выполнения оператора будет массив, содержащий 10 объектов MyType.

Если MyType является ссылочным типом, то в результате выполнения данного оператора будет создан массив из 10 элементов типа ссылка, каждый из которых будет проинициализирован пустой ссылкой – значением null.

Доступ к элементам массива реализуется в соответствии с правилом индексации – по каждому измерению индексация осуществляется с НУЛЯ до n-1, где n – количество элементов размерности.

Ещё один пример совмещения инициализации массива с его передачей как параметра методу. Перебор элементов массива реализуется в соответствии с правилом индексации:

```
using System;
public class ArrayClass
{
static void PrintArray(int[,] w)
 {
// Display the array elements:
for (int i=0; i < 4; i++)
for (int j=0; j < 2; j++)
Console.WriteLine(“Element({0},{1})={2}”, i, j, w[i,j]);
 }

public static void Main()
 {
// Pass the array as a parameter:
PrintArray(new int[,] {{1,2}, {3,4}, {5,6}, {7,8}});
 }
}
```

Встроенный сервис по обслуживанию простых массивов

При работе с массивами следует иметь в виду одно важное обстоятельство.

В .NET ВСЕ массивы происходят от ОДНОГО общего (базового) класса Array. Это означает, что ВСЕ созданные в программе массивы обеспечиваются специальным

набором методов для создания, управления, поиска, и сортировки, элементов массива. К числу таких методов и свойств, в частности, относятся свойства:

public int Length {get;}
Возвращает целое число представляющее общее количество элементов во всех измерениях массива.
public int Rank {get;}
Возвращает целое число представляющее количество измерений массива.

И методы:
public static Array CreateInstance(Type, int, int);
Статический метод (один из вариантов), создаёт массив элементов заданного типа и определённой размерности.
public void SetValue(object, int, int);
Присваивает элементу массива значение, представленное первым параметром (один из вариантов).
public object GetValue(int, int);
Извлекает значение из двумерного массива по индексам (один из вариантов).

```
using System;
public class DemoArray
{

public static void Main()
{
// Создали и проинициализировали двумерный массив строк.
Array myArray=Array.CreateInstance( typeof(String), 2, 4 );
myArray.SetValue( "The quarter moon ", 0, 0 );
myArray.SetValue( "comes out, ", 0, 1 );
myArray.SetValue( "leaves", 0, 2 );
myArray.SetValue( "a fog,", 0, 3 );
myArray.SetValue( "takes out", 1, 0 );
myArray.SetValue( "a knife", 1, 1 );
myArray.SetValue( "from", 1, 2 );
myArray.SetValue( "a pocket.", 1, 3 );

// Показали содержимое массива.
Console.WriteLine( "The Array contains the following values:" );
for ( int i = myArray.GetLowerBound(0); i <= myArray.GetUpperBound(0); i++ )
      for ( int j = myArray.GetLowerBound(1); j <= myArray.GetUpperBound(1); j++ )
            Console.WriteLine( "\t[{0},{1}]:\t{2}", i, j, myArray.GetValue( i, j ) );
}
}
```

Реализация сортировки в массиве стандартными методами

Массив элементов – это всегда особый класс, независимо от типа образующих его элементов, производный от базового класса Array.

Класс Array является основой для любого массива и предоставляет методы для создания, манипулирования (преобразования), поиска и сортировки на множестве элементов массива.

В частности, варианты метода Sort() обеспечивают реализацию механизмов сортировки элементов ОДНОМЕРНОГО массива объектов (в смысле представителей класса Object).

Сортировка элементов предполагает:

ПЕРЕБОР всего множества (или его части) элементов массива,

СРАВНЕНИЕ (значений) элементов массива в соответствии с определённым критерием сравнения. Критерии и алгоритмы сравнения могут задаваться либо с помощью массивов ключей, либо реализацией интерфейса IComparable.

Следующий вариант метода Sort сортирует полное множество элементов одноразмерного массива с использованием стандартного интерфейса сравнения.

public static void Sort(Array);

Стандартный интерфейс сравнения способен распознавать значения элементов массива и сравнивать эти элементы между собой.

Таким образом, никаких проблем не существует, если надо сравнить и переупорядочить массивы элементов базовых типов. Относительно …, System.Int16,

System.Int32, System.Int64, …,System.Double,… всегда можно сказать, какой из сравниваемых элементов больше, а какой меньше.

Сортировка элементов массива (массива целевых элементов) также может быть произведена с помощью вспомогательного массива ключей. Суть подобной сортировки заключается в том, что между элементами массива ключей и элементами сортируемого массива устанавливается соответствие. В частном случае, если количество элементов массива ключей оказывается равным количеству элементов сортируемого массива, это соответствие оказывается взаимнооднозначным. При этом сортируются элементы массива ключей. Предполагается, что используемый при этом интерфейс сравнения способен обеспечить необходимые алгоритмы.

При сортировке элементов массива ключей каждая из возможных перестановок элементов массива сопровождается перестановкой соответствующих item'ов массива. Таким образом, упорядочение массива целевых элементов осуществляется вне зависимости от реальных значений этих элементов данного массива. Читаются и сравниваются значения массива ключей, сами же элементы целевого массива перемещаются в соответствии с перемещениями элементов массива.

```
public static void Sort(Array, Array);
```

```
using System;
namespace SortArrays
{
class Class1
{
static void Main(string[] args)
 {
int[] iArr = {0,1,2,3,4,5,6,7,8,9}; // Целевой массив.
int[] keys = {10,1,2,3,4,5,6,7,8,9}; // Массив ключей.
for (i = 0; i < 10; i++) Console.WriteLine("{0}: {1}", i, iArr[i]);
// Сортировка массива с использованием ключей.
System.Array.Sort(keys,iArr);
Console.WriteLine("================================================");
for (i = 0; i < 10; i++) Console.WriteLine("{0}: {1}", i, iArr[i]);
 }
}
}
```

В процессе сортировки массива ключей элемент со значением ключа 10 перемещается в конец массива. При этом нулевой элемент целевого массива со значением 0 перемещается на последнюю позицию.

Целевой массив и массив ключей вида при сортировке

```
int[] iArr = {0,1,2,3,4,5,6,7,8,9}; // Целевой массив.
int[] keys = {9,8,7,6,5,4,3,2,1,0}; // Массив ключей.
```

обеспечивают размещение элементов целевого массива в обратном порядке.

Следующее сочетание значений целевого массива и массива ключей обеспечивает изменение расположения первых четырёх элементов целевого массива:

```
int[] iArr = {0,1,2,3,4,5,6,7,8,9}; // Целевой массив.
int[] keys = {3,2,1,0}; // Массив ключей.
```

А такие наборы значений целевого и ключевого элементов массива при сортировке с использованием массива ключей обеспечивают изменение порядка целевых элементов массива с индексами 4, 5, 6, 7.

```
int[] iArr = {0,1,2,3,4,5,6,7,8,9};
int[] keys = {0,1,2,9,8,7,6,10,11,12};
```

Но самое главное и самое интересное в методе сортировки с использованием ключевого массива – это то, что в этом никаким образом не участвуют значения элементов целевого массива.

Надо отсортировать (изменить порядок расположения) составляющих массив массивов компонентов – надо всего лишь установить соответствие между элементами целевого массива и элементами массива ключей – и вызвать

соответствующий вариант метода сортировки. Составляющие массива массивов будут восприняты как объекты-представители класса object и, не вдаваясь в подробности (а по какому принципу упорядочивать массивы?) будут переупорядочены в соответствии с новым порядком расположения элементов ключевого массива.

```
int[][] iArr = new int[3][]{
new int[]{0},
new int[]{0,1},
new int[]{0,1,2,3,4,5,6,7,8,9}
 };
int[] keys = {3,2,1};
```

Вариант метода, позволяющего организовать сортировку подмножества элементов целевого массива, начиная с элемента, заданного вторым параметром метода и включающим количество элементов, заданным вторым параметром.

```
public static void Sort(Array,int,int);
```

Вариант метода сортировки основанный на сортировке элементов ключевого массива. Обеспечивает сортировку подмножества массива. Принципы выделения подмножества аналогичны рассмотренному выше варианту метода сортировки.

```
public static void Sort(Array,Array,int,int);
```

Сортировка на основе сравнения пар объектов-членов одномерного массива с использованием стандартного интерфейса сравнения

```
public static void Sort(Array,IComparer);

// Сортировка элементов с использованием стандартного Компарера.
// iArr – одномерный массив целочисленных значений.
// System.Collections.Comparer.DefaultInvariant стандартный Компарер.
try
{
System.Array.Sort(iArr,System.Collections.Comparer.DefaultInvariant);
}
catch (Exception ex)
{
Console.WriteLine(ex.ToString());
}
```

System.Collections.Comparer.DefaultInvariant - интерфейс определяет обобщённый метод сравнения значений (класса или типа), переопределение которого позволяет задать собственный специфический для данного класса метод сравнения элементов массива (interface implemented by each element of the Array).

```
using System;
using System.Collections;

namespace SortArrays
{
/// <summary>
/// Данные для массива элементов.
/// Подлежат сортировке в составе массива методом Sort.
/// Главная проблема заключается в том, что никто из тех, кто обеспечивает
/// встроенную сортировку, ни в классе Array, ни в классе Items не знает,
/// как воспринимать и каким образом сравнивать между собой
/// объекты-представители класса Items. Объекты itObj0 b itObj1
/// со значениями полей (7,5) и (5,7) - кто из них «больше»?
/// </summary>
class Items
{
/// <summary>
/// Конечно, можно об определении аксиом для установления отношения
/// порядка между элементами множества объектов-представителей класса
/// Items и о реализации соответствующих методов сравнения
/// можно было позаботиться при объявлении класса и просто не допускать
/// к стандартной сортировке объекты классов, в которых не определено
```

```
/// отношение порядка, однако в .NET используется другой подход.
/// <summary>
public int val1;
public int val2;

public Items (int key1, int key2)
{
val1 = key1;
val2 = key2;
}
}


/// <summary>
/// Предполагается, что отношение порядка над элементами
/// множества объектов-представителей соответствующего
/// класса может быть введено в любой момент. Для этого
/// достаточно иметь доступ к значениям соответствующих
/// полей объектов.
/// Для этого достаточно воспользоваться стандартными
/// библиотечными средствами .NET и реализовать (доопределить)
/// соответствующие интерфейсы (заготовки методов) сравнения.
/// Ниже объявляются два варианта процедур сравнения,
/// использование которых позволяет реализовать стандартные
/// алгоритмы сортировки, применяемые в классе Array.
/// ...КОМПАРАТОРЫ...
/// </summary>

class myComparer0: IComparer
{

int IComparer.Compare(object obj1, object obj2)
{
if (((Items)obj1).val1 == ((Items)obj2).val1) return 0;
if (((Items)obj1).val1 > ((Items)obj2).val1) return 1;
if (((Items)obj1).val1 < ((Items)obj2).val1) return -1;
else return 0;
}
}


class myComparer1: IComparer
{
int IComparer.Compare(object obj1, object obj2)
{
if (((Items)obj1).val2 == ((Items)obj2).val1) return 0;
if (((Items)obj1).val2 > ((Items)obj2).val2) return 1;
if (((Items)obj1).val2 < ((Items)obj2).val2) return -1;
else return 0;
}
}

/// <summary>
/// После реализации соответствующего интерфейса объекты-компараторы,
/// «профессиональные сравниватели», обеспечивают реализацию
/// стандартных алгоритмов сортировки.
/// </summary>

class Class1
{
[STAThread]
static void Main(string[] args)
{
// Объект-генератор «случайных» чисел.
//     Стартует с использованием a time-dependent default seed value.
Random rnd = new Random();
int i;
// Массив Items.
Items[] itArr = new Items[10];

// Создали две версии Компареров, способных сравнивать пары
```

```
// объектов-представителей класса Items.
myComparer0 c0 = new myComparer0();
myComparer1 c1 = new myComparer1();

Console.WriteLine("================================================");
// Проинициализировали массив объектов-представителей класса Items.
for (i = 0; i < 10; i++)
      {
itArr[i] = new Items(rnd.Next(0,10),rnd.Next(0,10));
      }

for (i = 0; i < 10; i++)
      {
Console.WriteLine("{0}: {1},{2}", i, itArr[i].val1, itArr[i].val2);
      }

// Сортируются элементы массива типа Items.
// Условием успешной сортировки элементов массива является реализация
// интерфейса IComparer. Если Компарер не сумеет справиться с
// поставленной задачей - будет возбуждено исключение.
try
      {
System.Array.Sort(itArr,c0);
      }
catch (Exception ex)
      {
Console.WriteLine(ex);
      }

Console.WriteLine("================================================");
for (i = 0; i < 10; i++)
      {
Console.WriteLine("{0}: {1},{2}", i, itArr[i].val1, itArr[i].val2);
      }

Console.WriteLine("================================================");

// Сортируются элементы массива типа Items.
// Условием успешной сортировки элементов массива является реализация
// интерфейса IComparer.
try
      {
System.Array.Sort(itArr,c1);
      }
catch (Exception ex)
      {
Console.WriteLine(ex);
      }

for (i = 0; i < 10; i++)
      {
Console.WriteLine("{0}: {1},{2}", i, itArr[i].val1, itArr[i].val2);
      }

}
}
}
```

Подробнее о массивах массивов (jagged array)

A jagged array также называют "массивом массивов".

Прежде всего, элементы зубчатого массива на одном и том же уровне (элементы одной размерности) могут иметь разные размеры. На этих массивах функция GetLength() и свойство Rank показывают результаты, аналогичные простым одномерным массивам.

Рассмотрим примеры объявления, инициализации и доступа к элементам jagged arrays.

Вот объявление a single-dimensional (ОДНОМЕРНОГО!) массива, состоящего из трёх элементов, каждый из которых является одномерным массивом целых:

```
int[][] myJaggedArray = new int[3][];
```

Дальнейшее использование этого массива требует инициализации его элементов. Например, так:

```
myJaggedArray[0] = new int[5];
myJaggedArray[1] = new int[4];
myJaggedArray[2] = new int[2];
```

Каждый из элементов является single-dimensional массивом целых. Количество элементов каждого массива очевидно из соответствующих операторов определения.

Ниже показан пример использования заполняющей инициализации, при которой одновременно с определением (созданием) массивов производится присвоение элементам новорожденных массивов конкретных значений:

```
myJaggedArray[0] = new int[] {1,3,5,7,9};
myJaggedArray[1] = new int[] {0,2,4,6};
myJaggedArray[2] = new int[] {11,22};
```

Вышеупомянутый массив может быть объявлен и проинициализирован и таким образом:

```
int[][] myJaggedArray = new int [][]
 {
new int[] {1,3,5,7,9},
new int[] {0,2,4,6},
new int[] {11,22}
 };
```

И ещё один эквивалентный способ инициализации массива. В этом случае используется неявная инициализация на верхнем уровне, как при инициализации обычного одномерного массива. Важно (!) что при определении составляющих этого массива операция new опущена быть не может. Каждая компонента требует явного применения операции new или присвоения ссылки на ранее созданный одномерный массив:

```
int[][] myJaggedArray = {
new int[] {1,3,5,7,9},
new int[] {0,2,4,6},
new int[] {11,22}
 };
```

Доступ к элементам ступенчатого массива обеспечивается посредством выражений индексации:

```
// Assign 33 to the second element of the first array:
myJaggedArray[0][1] = 33;
// Assign 44 to the second element of the third array:
myJaggedArray[2][1] = 44;
```

C# позволяет собирать разнообразные конструкции на основе jagged многомерных массивов. Ниже приводится пример объявления и инициализации одномерного jagged array, содержащего в качестве элементов двумерные массивы различных размеров:

```
int[][,] myJaggedArray = new int [3][,]
 {
new int[,] { {1,3}, {5,7} },
new int[,] { {0,2}, {4,6}, {8,10} },
new int[,] { {11,22}, {99,88}, {0,9} }
 };
```

Доступ к отдельным элементам jagged массива обеспечивается различными комбинациями выражений индексации. В приводимом ниже примере выводится значение элемента массива [1,0], расположенного по нулевому индексу myJaggedArray (это 5):

```
Console.Write("{0}", myJaggedArray[0][1,0]);
```

И ещё один пример, в котором строится массив myArray, элементами которого являются массивы. Каждый из составляющих имеет собственные размеры.

```
// cs_array_of_arrays.cs
using System;
public class ArrayTest
{
public static void Main()
 {
// Declare the array of two elements:
int[][] myArray = new int[2][];
// Initialize the elements:
myArray[0] = new int[5] {1,3,5,7,9};
myArray[1] = new int[4] {2,4,6,8};

// Display the array elements:
for (int i=0; i < myArray.Length; i++)
 {
Console.Write("Element({0}):", i);
for (int j = 0 ; j < myArray[i].Length ; j++)
Console.Write("{0}{1}", myArray[i][j], j == (myArray[i].Length-1) ? "" : " ");
Console.WriteLine();
 }
 }
}
```

Результат:

```
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
```

Массивы как параметры

В качестве параметра методу всегда можно передать ссылку на ОДНОМЕРНЫЙ массив. Массив, первая спецификация размерности которого в объявлении имеет вид …[] …

Тип и количество составляющих данный массив компонентов для механизма передачи параметров значения не имеют. Важно, что в стеке будет выделено определённое (соответствующее значению первой размерности) количество проинициализированных ссылок на составляющие данный одномерный массив элементов.

Этот принцип действует во всех случаях. В том числе и при вызове приложения из командной строки.

Вызов приложения из командной строки с передачей ему в качестве параметров массива ОДНОТИПНЫХ (строковых) значений.

```
using System;
class C1
{
static void Main(string[] args)
 {
Console.WriteLine("Values of parameters:");
for (int i = 0; I < args.Length; i++)
 {
Console.WriteLine("{0}: {1}", i, args[i]);
 }
 }
}
```

Сначала создаётся консольное приложение (пусть под именем ComLine, в файле ComLine.cs).

Транслятор можно запустить из командной строки, выглядит это примерно так:

```
C:> csc /t:exe ConLine.cs
```

Результат деятельности транслятора размещается в .exe файле под именем ConLine.exe и также может быть запущен из командной строки с передачей строк символов в качестве входных параметров.

```
C:>ConLine qwerty asdfgh zxcvbn ok ok ok-k
```

В окне консольного приложения отобразится следующее множество строк:

```
Values of parameters:
0: qwerty
1: asdfgh
2: zxcvbn
3: ok
4: ok
5: ok-k
```

А теперь с использованием оператора foreach:

```
using System;
class C1
{
static void Main(string[] args)
 {
Console.WriteLine("Values of parameters, using foreach:");
foreach (string arg in args)
 {
Console.WriteLine("{0}", arg);
 }
 }
}
```

Запускаем:

```
C:>ConLine qwerty asdfgh zxcvbn ok ok ok-k
```

Получаем:

```
Values of parameters, using foreach:
qwerty
asdfgh
zxcvbn
ok
ok
ok-k
```

Полностью построенный массив можно передать в качестве входного параметра методу.

Пустая ссылка на массив может быть передана методу в качестве выходного параметра.

Например:

```
int[] myArray; // myArray == null
PrintArray(myArray);
```

Можно также совместить передачу массива как параметра методу с предварительной инициализацией этого массива. Например:

```
PrintArray(new int[] {1, 3, 5, 7, 9});
In the following example, a string array is initialized and passed as a parameter to
the PrintArray method, where its elements are displayed:
// cs_sd_arrays.cs
using System;
public class ArrayClass
{
static void PrintArray(string[] w)
 {
for (int i = 0 ; i < w.Length ; i++)
 {
```

```
Console.Write(w[i] + "{0}", i < w.Length - 1 ? " " : "");
 }

Console.WriteLine();
 }

public static void Main()
 {
// Declare and initialize an array:
string[] WeekDays = new string []
{"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};
// Pass the array as a parameter:
PrintArray(WeekDays);
 }
}
```

Вывод:

Sun Sat Mon Tue Wed Thu Fri

Спецификатор params

Неопределённое (переменное) количество (ОДНОТИПНЫХ!) параметров или список параметров переменной длины.

Это всегда последний параметр в списке параметров со спецификатором params.

В выражении вызова метода должен располагаться либо список однотипных параметров (этот список преобразуется в массив значений), либо ссылка на ОДНОМЕРНЫЙ массив значений определённого типа.

```
using System;

class Class1
{
static void Main(string[] args)
{
f1(0,1,2,3,4,5);
f1(0,1,2,3,4,5,6,7,8,9,10);

f2(new int[,]{{0,1},{2,3}}, new int[,]{{0,1,2,3,4,5},{6,7,8,9,10,11}});
f2(new int [,]{{0,1},{2,3}});
}

static void f1(params int[] m)
{
Console.WriteLine(m.Length.ToString());
}

static void f2(params int[][,] p)
{
Console.WriteLine(p.Length.ToString());
}
}
```

Main в классе. Точка входа

Без статической функции (метода) Main невозможно построить выполняемую программу. Без явно обозначенной точки входа сборка не можен выполняться.

В сборке можно располагать несколько классов. Каждый класс располагает собственным набором методов. В каждом классе могут располагаться одноименные методы. В следующем примере объявляются три класса в одном пространстве имён. В каждом классе объявляется независимая точка входа. Три (!) СТАТИЧЕСКИЕ функции Main. Возможно и такое. При этом главная проблема – при компиляции надо явным образом указать точку входа.

- Это можно сделать из командной строки при вызове компилятора. Например, так:

c:\ csc /main:Class1.Class3 Example1.cs

- Можно через диалог The Startup Object property среды разработки приложений (меню Проект-Свойства проекта, далее - General, Common Properties, <Projectname> Property Pages Dialog Box (Visual C#)), который обеспечивает спецификацию значений, которые явным образом НЕ ПРОПИСАНЫ в проекте. В разделе Startup object, надо раскрыть список классов и указать соответствующий класс.

Транслятор соберёт сборку, В которой будет обеспечена передача управления соответствующей функции Main (одной из трёх!).

```
using System;
namespace Example1
{
//================================================
public class Class1
{
// Спецификатор public нужен здесь. Третий класс.
public class Class3
 {
public static void Main()
 {
string[] sss = new string[]{Class1.s.ToString(),”12345”};
Class1.Main(sss);
 }
}

int d = 0;
public static int s;
static void Main(string[] args)
{
Class1 c1 = new Class1();
f1(c1);
c1.f2();
Class2 c2 = new Class2();
//c2.f2();
c2.f3();
string[] sss = new string[] {”qwerty”, c1.ToString()};
Class2.Main(sss);
 }

static void f1(Class1 x)
 {
//x.s = 100;
s = 0;
Class1.s = 125;
x.d = 1;
//d = 100;
 }

void f2()
{
s = 0;
Class1.s = 100;
//this.s = 5;
//Class1.d = 125;
this.d = 100;
d = 100;
}

}
//================================================
class Class2
{
int d;
static int s;

public static void Main(string[] args)
 {
Class1.Class3.Main();
Class2 c2 = new Class2();
```

```
f1(c2);
c2.f2();
//Class1.Main();
 }

static void f1(Class2 x)
 {
//x.s = 100;
s = 0;
Class2.s = 125;
x.d = 1;
//d = 100;
}

void f2()
{
s = 0;
Class1.s = 100;
//this.s = 5;
//Class1.d = 125;
this.d = 100;
d = 100;
}

public void f3()
{
s = 0;
Class1.s = 100;
//this.s = 5;
//Class1.d = 125;
this.d = 100;
d = 100;
}

}
//=============================================
}
```

Структура также может иметь свою точку входа!

```
using System;
namespace defConstructors
{
struct MyStruct
 {
static void Main(string[] args)
 {
Console.WriteLine("Ha-ha-ha");
 }
 }
}
```

Создание объекта. Конструктор

Конструктором называется группировка кода, которой передаётся управление при создании объекта. Синтаксис объявления конструктора аналогичен обявлению метода. Те же спецификаторы доступа, имя, список параметров. Особенности конструктора заключаются в том, что:

- конструктор НЕ ИМЕЕТ НИКАКОГО спецификатора возвращаемого даже void,
- имя конструктора полностью совпадает с именем класса или структуры,
- в классе и в структуре можно объявлять множество вариантов конструкторов. Они должны отличаться списками параметров. В структуре невозможно объявить конструктор с пустым списком параметров,
- не существует выражения вызова для конструктора, управление в конструктор передаётся посредством выполнения специальной операции new.

В C# ключевое слово new может быть использовано как ОПЕРАЦИЯ или как МОДИФИКАТОР.

new как операция используется:

- для ссылочных типов (типов-ссылок) при создании объектов в уравляемой памяти и передачи управления конструкторам,
- для размерных типов (типов-значений) при создании объектов в СТЕКЕ. Возможно, что при этом может потребоваться обращение к конструктору.

new как модификатор используется для сокрытия наследуемых из базового класса данных и методов. Об этом позже. После того, как рассмотрим подробнее принцип наследования.

Операция new

new операция используется для создания объектов и передачи управления конструкторам, например:

```
Class1 myVal = new Class1(); // Объект ссылочного типа. Создаётся в куче.
```

new также используется для обращения к конструкторам типа-значений (размерного типа), например:

```
int myInt = new int(); // Объект типа int размещается в стеке!
```

При определении объекта myInt ему было присвоено начальное значение 0, которое является значением по умолчанию для типа int. Следующий оператор имеет тот же самый эффект:

```
int myInt = 0; // Для размерного типа аналогично.
```

Конструктор БЕЗ ПАРАМЕТРОВ (конструктор умолчания) обеспечивает инициализацию переменной предопределённым значением. Со списком предопределённых значений, которыми инициализируются объекты предопределённых типов, можно ознакомиться в Default Values Table.

У структуры конструктор умолчания (конструктор без параметров) НЕ ПЕРЕОПРЕДЕЛЯЕТСЯ! Для них объявляются только параметризованные конструкторы.

А вот для предопределённых типов конструкторов с параметрами в принципе нет!

```
int q = new int();
//int q = new int(125); // Такого нет.
```

Сколько может иметь конструкторов структура. При использовании правил перегрузки – неограниченное количество.

Сколько может иметь конструкторов класс. Всегда на один больше, чем структура.

В следующем примере, создаются с использованием операции new и конструктора объекты-представители класса и структуры. Для инициализации полей-членов класса используются параметры конструкторов. Присвоение значений осуществляется в теле конструктора с использованием операций присвоения.

```
// cs_operator_new.cs
// The new operator
using System;
class NewTest
{
struct MyStruct
 {
public int x;
public int y;

public MyStruct (int x, int y)
 {
this.x = x;
this.y = y;
 }
 }

class MyClass
 {
```

```
public string name;
public int id;

public MyClass ()
 {
 }

public MyClass (int id, string name)
 {
this.id = id;
this.name = name;
 }
 }

public static void Main()
 {
// Create objects using default constructors:
MyStruct Location1 = new MyStruct();
MyClass Employee1 = new MyClass();

// Display values:
Console.WriteLine("Default values:");
Console.WriteLine(" Struct members: {0}, {1}",
Location1.x, Location1.y);
Console.WriteLine(" Class members: {0}, {1}",
Employee1.name, Employee1.id);
// Create objects using parameterized constructors::
MyStruct Location2 = new MyStruct(10, 20);
MyClass Employee2 = new MyClass(1234, "John Martin Smith");
// Display values:
Console.WriteLine("Assigned values:");
Console.WriteLine(" Struct members: {0}, {1}",
Location2.x, Location2.y);
Console.WriteLine(" Class members: {0}, {1}",
Employee2.name, Employee2.id);
 }
}
```

В управляемой памяти нет ничего, что бы создавалось без конструктора

Даже если его присутствие не обозначено явным образом.

```
string s = "qwerty";
int[] intArray = {1,2,3,4,5};
```

Эти операторы являются всего лишь сокращённой формой следующих операторов определения:

```
string s = new string(new char[]{'q','w','e','r','t','y'});
int[] intArray = new int[5]{1,2,3,4,5};
```

Кто строит конструктор умолчания

Конструктор умолчания (конструктор с пустым списком параметров) строится транслятором по умолчанию. Если класс не содержит явных объявлений конструкторов, именно этот конструктор берёт на себя работу по превращению области памяти в объект.

```
Class X
{

}
::::::::::
X x = new X(); // Работает конструктор умолчания.
```

Однако попытка объявления ЛЮБОГО варианта конструктора (с параметрами или без) приводит к тому, что транслятор перестаёт заниматься построением собственных версий конструкторов. Отныне в классе нет больше конструктора умолчания. Теперь всё зависит от соответствия оператора определения объекта

построенному нами конструктору. Объявим в производном классе оба варианта конструкторов.

```
Class X
{
public X(int key){}
}
::::::::::
X x0 = new X(125); // Работает конструктор с параметрами.
X x1 = new X(); // Не дело! Конструктора умолчания уже нет!
```

Однажды взявшись за дело объявления конструкторов, разработчик класса должен брать на себя ответственность за создание ВСЕХ без исключения версий конструкторов. Таковы правила.

```
Class X
{
public X(int key){}
public X(){}
}
::::::::::
X x0 = new X(125); // Работает конструктор с параметрами.
X x1 = new X(); // Работает новый конструктор без параметров.
```

this в контексте конструктора

Конструктор не вызывается. Передача управления конструктору осуществляется при выполнении операции new.

Ситуация: сложная структура класс. Различные варианты конструкторов, специализирующиеся на создании и инициализации различных вариантов объектов вынуждены выполнять общий список регламентных работ по инициализации объектов.

Оставить ОДИН конструктор, выполняющий ВСЮ “черновую” работу по созданию объектов. “Тонкую” настройку объектов производить после выполнения кода конструктора, непосредственно вызывая соответствующие методы-члены. При этом вызываются методы именно ПОСЛЕ того, как отработает конструктор.

Определить множество специализированных конструкторов, после выполнения которых вызывать дополнительный метод инициализации для общей “доводки” объекта.

Некрасиво. Код получается сложный.

В C# реализован другой подход, который состоит в следующем:

Определяется множество разнообразных специализированных конструкторов.

Выделяется наименее специализированный конструктор, которому поручается выполнение обязательной работы.

Обеспечивается передача управления из конструктора конструктору.

Так вот this в контексте конструктора обеспечивает в C# передачу управления от конструктора к конструктору. Таким образом, программист освобождается от необходимости повторного кодирования алгоритмов инициализации для каждого из вариантов конструктора.

Следующий фрагмент программного кода демонстрирует объявление нескольких конструкторов с передачей управления “золушке”.

```
class Point2D
{
private float x, y;
public Point2D(float xKey, float yKey)
 {
Console.WriteLine(“Point2D({0}, {1}) is here!”, xKey, yKey);
// Какой - нибудь сложный обязательный
// код инициализации данных-членов класса.

int i = 0;
while (i < 100)
 {
x = xKey;
y = yKey;
i++;
```

```
 }
 }

// А все другие конструкторы в обязательном порядке предполагают
// регламентные работы по инициализации значений объекта - и делают при этом
// ещё много чего...
public Point2D():this(0,0)
 {
int i;
for (i = 0; i < 100; i++)
 {
// Хорошо, что значения уже проинициализированы!
// Здесь своих проблем хватает.
 }

Console.WriteLine("Point2D() is here!");
 }

public Point2D(Point2D pKey):this(pKey.x, pKey.y)
 {
int i;
for (i = 0; i < 100; i++)
 {
// Хорошо, что значения уже проинициализированы!
// Здесь своих проблем хватает.
 }
Console.WriteLine("Point2D({0}) is here!", pKey.ToString());
 }
}
```

Перегрузка операций

Основная конструкция C# - объявление класса или структуры.

Классы и структуры есть типы. Тип характеризуется неизменяемым набором свойств и методов. Для встроенных типов определены множества преобразований (операций), которые кодируются с использованием предопределённого множества операций. Язык позволяет строить сложные выражения с использованием этих операций, причём результат выполнения (определения результирующего значения) зависит от типа составляющих сложное выражение элементарных выражений. Например, сложение целочисленных значений определяется и выполняется иначе, нежели сложение чисел с плавающей точкой.

Программист может строить сложные выражения с использованием символов арифметических, логических, операций сравнения и прочих операций на основе элементарных выражений встроенных типов.

Вновь объявляемые классы служат основой для создания объектов. Эти объекты в принципе ничем не отличаются от других объектов, в том числе от объектов-представителей элементарных аривметических типов. В частности, ссылки на такие объекты могут использоваться как элементарные выражения в выражениях более сложной структуры.

Для построения сложных выражений на основе элементарных выражений производных (объявляемых программиистом) типов C# предоставляет те же возможности, что и для выражений всех прочих типов. При этом главная проблема заключается в том, что алгоритм вычисления значения представленного операндами вновь объявляемого типа, в сочетании с символом операции '+' (например) для операндов этого типа неизвестен. Семантика операции должна быть специальным образом определена программистом при определении класса.

Перегрузка операций в C# как раз и является способом объявления семантики операций, обозначаемых привычным набором символов операций. Перегрузка операций строится на основе общедоступных (public) статических (вызываемых от имени класса) функций-членов с использованием ключевого слова operator.

Не все операции множества могут быть переопределены подобным образом. Не все операции могутбыть перегружены. Некоторые операции могут перегружаться с ограничениями.

В таблице приводится соответствующая информация различных категорий символов операций:

Операция	Перегружаемость

+, -, !, ~, ++, --, true, false	Унарные символы операций, допускающие перегрузку.
+, -, *, /, %, &, |, ^, <<, >>	Бинарные символы операций, допускающие перегрузку.
==, !=, <, >, <=, >=	Операции сравнения перегружаются.
&&, ||	Условные логические операции моделируются с использованием ранее переопределённых операций & и |.
[]	Операции доступа к элементам массивов моделируются за счёт индексаторов.
()	Операции преобразования реализуются с ипользованием ключевых слов implicit и explicit.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Операции не перегружаются, по причине невозможности перегрузки операции прсвоения.
=, ., ?:, ->, new, is, sizeof, typeof	Операции, не подлежащие перегрузке.

Префиксные операции ++ и -- перегружаются парами.

Операции сравнения перегружаются парами. Если перегружается операция ==, также должна перегружаться опеация !=. То же самое относится к парам < и >, <= и >=.

Синтаксис объявления операторной функции

Перегрузка операций основывается на следующем принципе.

С использованием специфического синтаксиса определяется статическая операторная функция, в заголовке которой указывается символ переопределяемой операции. В теле функции реализуется соответствующий алгоритм, определяющий семантику операторной функции (семантику новой операции). Выражение вызова операторной функции напоминает синтаксис выражения, построенного на основе соответствующего символа операции. При трансляции сложного выражения, включающего символы операций, транслятор проверяет находит соответствующую операторную функцию и обеспецивает передачу управления этой функции.

Синтаксис объявления операторных функций представлен в виде множества БНФ.

```
operator-declaration ::=
attributes opt operator-modifiers operator-declarator operator-body

operator-modifiers ::= operator-modifier
                   ::= operator-modifiers operator-modifier

operator-modifier ::= public | static | extern
operator-declarator ::= unary-operator-declarator
                    ::= binary-operator-declarator
                    ::= conversion-operator-declarator

unary-operator-declarator ::=
                  type operator overloadable-unary-operator (type identifier)
overloadable-unary-operator ::= +
                            ::= -
                            ::= !
                            ::= ~
                            ::= ++
                            ::= --
                            ::= true
                            ::= false

binary-operator-declarator ::=
          type operator overloadable-binary-operator (type identifier ,type identifier)
overloadable-binary-operator ::= +
                             ::= -
                             ::= *
                             ::= /
                             ::= %
                             ::= &
                             ::= |
                             ::= ^
                             ::= <<
```

```
                                ::= >>
                                ::= ==
                                ::= !=
                                ::= >
                                ::= <
                                ::= >=
                                ::= <=

conversion-operator-declarator ::= implicit operator type (type identifier)
                                 ::= explicit operator type (type identifier)

operator-body ::= block
              ::= ;
```

Унарные операции. Пример объявления и вызова

```
using System;

class Point2D
{
int x,y;

Point2D()
{
x=0;
y=0;
}

Point2D(Point2D key)
{
x=key.x;
y=key.y;
}


// Перегруженные операции обязаны возвращать значения!
// Операторные функции!
// Они должны объявляться как public и static.
// Префиксная и постфиксная формы операций ++ и –
// НЕ различаются по результату выполнения (что есть криво)!
// Тем не менее, они могут быть объявлены:
// Одна как префиксная!
public static Point2D operator ++ (Point2D par)
{
par.x++;
par.y++;
return par;
}

// Другая как постфиксная!
public static Point2D operator -- (Point2D par)
{
Point2D tmp = new Point2D(par);
// Скопировали старое значение.
par.x--;
par.y--;
// Модифицировали исходное значение. Но возвращаем старое!
return tmp;
}


static void Main(string[] args)
{
Point2D p = new Point2D();

// В соответствии с объявлением, плюсы ВСЕГДА ПРЕФИКСНЫ, а минусы ПОСТФИКСНЫ
p++; // Префиксная.
++p; // Префиксная.
p--; // Постфиксная.
```

```
--p; // Постфиксная.
}
}
```

Бинарные операции

Семантика перегружаемой операторной функции определяется решаемыми задачами и фантазией разработчика.

```
// Бинарные операции также обязаны возвращать значения!
public static Point2D operator + (Point2D par1, Point2D par2)
{
return new Point2D(par1.x+par2.x,par1.y+par2.y);
}

// Реализуется алгоритм "сложения" значения типа Point2D со значением типа float.
// От перемены мест слагаемых сумма НЕ ИЗМЕНЯЕТСЯ. Однако эта особенность нашей
// операторной функции "сложения" (операции "сложения") должна быть прописана
// программистом. В результате получаем ПАРУ операторных функций, которые отличаются
// списками параметров.
// Point2D + float
public static Point2D operator + (Point2D par1, float val)
{
return new Point2D(par1.x+val,par1.y+val);
}

// float + Point2D
public static Point2D operator + (float val, Point2D par1)
{
return new Point2D(val+par1.x,val+par1.y);
}
```

А вот применение этих функций. Внешнее сходство выражений вызова операторных функций с обычными выражениями очевидно. И при этом иного способа вызова операторных функций нет!

```
…p1 + p2…
…3.14 + p2…
… p2 + 3.14…
```

Операции сравнения реализуются аналогично. Хотя не существует никаких ограничений на тип возвращаемого значения, в силу специфики применения (обычно в условных выражениях операторов управления) операций сравнения всё же имеет смысл определять их как операторные функции, возвращающие значения true и false.

```
public static bool operator == (myPoint2D par1, myPoint2D par2)
{
    if (par1.x==par2.x && par1.y==par2.y)
       return true;
    else
       return false;
}

public static bool operator != (myPoint2D par1, myPoint2D par2)
{
if (par1.x!=par2.x || par1.y!=par2.y)
       return true;
else
       return false;
}
```

true и false Operator

В известном мультфильме о Винни Пухе и Пятачке Винни делает заключение относительно НЕПРАВИЛЬНОСТИ пчёл. Очевидно, что по его представлению объекты-представители ЭТОГО класса пчёл НЕ удовлетворяют некоторому критерию.

В программе можно непосредственно поинтересоваться по поводу значения некоторого поля объекта:

```
Point2D p1 = new Point2D(GetVal(), GetVal());
 ::::::::::
// Это всё равно логическое выражение!
if (p1.x == 125) {/*…*/}
```

Так почему же не спросить об этом у объекта напрямую?

Хотя бы так:

```
// Критерий истинности объекта зависит от разработчика.
// Если значение выражения в скобках примет значение true,
// то пчела окажется правильной!
if (p1) {/*…*/}
```

В классе может быть объявлена операция (операторная функция) true, которая возвращает значение bool типа true для указания факта true и возвращает false в противном случае. Подобная операция полезна для типов, представляющих true, false, и null (ни истина, ни ложь), как это бывает при работе с базами данных.

Подобные типы (классы) могут быть использованы для управляющих выражений в if, do, while, for предложениях, а также в условных выражениях.

Если в классе была определена операция true, необходимо там же определить операцию false.

```
// Перегрузка булевских операторов. Это ПАРНЫЕ операторы.
// Объекты типа Point2D приобретают способность судить о правде и лжи!
// А что есть истина? Критерии ИСТИННОСТИ (не путать с истиной)
// могут быть самые разные.
public static bool operator true (Point2D par)
{
if (par.x == 1.0F && par.y == 1.0F) return true;
else return false;
}

public static bool operator false (Point2D par)
{
if (par.x == 0.0F && par.y == 0.0F) return false;
else return true;
}
```

Определение операций конъюнкция и дизъюнкции

Операторные функции конъюнкция и дизъюнкция объявляются только полсе того, как объявлены операторы true и false.

```
public static Point2D operator | (Point2D par1, Point2D par2)
{
if (par1) return par1;
if (par2) return par2;
else return new Point2D(-1.0F, -1.0F);
}

public static Point2D operator & (Point2D par1, Point2D par2)
{
if (par1 && par2) return par1;
else return new Point2D(-1.0F, -1.0F);
}
```

Выражение вызова операторной функции “дизъюнкция” имеет вид:

```
if (p0 | p1) Console.WriteLine(“true!”);
```

А как же || и &&

Эти операции сводятся к ранее объявленным операторным функциям.

Обозначим символом T тип, в котором была объявлена данная операторная функция.

Если при этом операнды операций && или || являются операндами типа T, и для них были объявлены соответствующие операторные функции operator &() и/или operator |(), то для успешной эмуляции операций && или || должны выполняться следующие ууслоия:

- Тип возвращаемого значения и типы каждого из параметров данной операторной функции должны быть типа T. Операторные функции operator & и operator |, определённые на множестве операндов типа T, должны возвращать результирующее значение типа T.
- К результирующему значению применяется объявленная в классе T операторная функция operator true (operator false).

При этом приобретают смысл операции && или ||. Их значение вычисляется в результате комбинации операторных функций operator true() или operator false() со следующими операторными функциями:

x && y представляется в виде выражения, построенного на основе трёхместной операции

T.false(x)? x: T.&(x, y),

где T.false(x) является выражением вызова объявленной в классе операторной функции false, а T.&(x, y) – выражением вызова объявленной в классе T операторной функции &.

Таким образом, сначала определяется "истинность" операнда x, и если значением соответствующей операторной функции является ложь, результатом операции оказывается значение, вычисленное для x. В противном случае определяется "истинность" операнда y, и результирующее значение определяется как КОНЪЮНКЦИЯ истинностных значений операндов x и y.

x || y представляется в виде выражения, построенного на основе трёхместной операции

T.true(x)? x: T.|(x, y),

где T.true(x) является выражением вызова объявленной в классе операторной функции, а T.|(x, y) – выражением вызова объявленной в классе T операторной функции |.

Таким образом, сначала определяется "истинность" операнда x, и если значением соответствующей операторной функции является истина, результатом операции оказывается значение, вычисленное для x. В противном случае определяется "истинность" операнда y, и результирующее значение определяется как ДИЗЪЮНКЦИЯ истинностных значений операндов x и y.

При этом в обоих случаях "истинностное" значение x вычисляется один раз, значение выражения, представленного операндом y не вычисляется вообще, либо определяется один раз.

И вот результат…

```
using System;

namespace ThisInConstruct
{
class Point2D
{
private float x, y;
public float X
{
get
{
return x;
}
}

public float PropertyY
{
get
```

```
{
return y;
}
set
{
string ans = null;
Console.Write("Are You sure to change the y value of object of Point2D? (y/n) >> ");
ans = Console.ReadLine();
if (ans.Equals("Y") || ans.Equals("y"))
{
y = value;
Console.WriteLine("The value y of object of Point2D changed...");
}
}
}

public Point2D(float xKey, float yKey)
{
Console.WriteLine("Point2D({0}, {1}) is here!", xKey, yKey);
// Какой - нибудь сложный обязательный
// код инициализации данных-членов класса.
int i = 0;
while (i < 100)
{
x = xKey;
y = yKey;

i++;
}
}

// А все другие конструкторы в обязательном порядке предполагают
// регламентные работы по инициализации значений объекта - и делают при этом
// ещё много чего...
public Point2D():this(0,0)
{
int i;
for (i = 0; i < 100; i++)
{
// Хорошо, что значения уже проинициализированы!
// Здесь своих проблем хватает.
}

Console.WriteLine("Point2D() is here!");
}

public Point2D(Point2D pKey):this(pKey.x, pKey.y)
{
int i;
for (i = 0; i < 100; i++)
{
// Хорошо, что значения уже проинициализированы!
// Здесь своих проблем хватает.
}

Console.WriteLine("Point2D({0}) is here!", pKey.ToString());
}
// Перегруженные операции обязаны возвращать значения!
// Операторные функции! Must be declared static and public.
// Префиксная и постфиксная формы ++ и -— не различаются по результату выполнения
// (что есть криво)! Тем не менее, они здесь реализуются:
// одна как префиксная…
public static Point2D operator ++ (Point2D par)
{
par.x++;
par.y++;
return par;
}

// другая как постфиксная…
public static Point2D operator -- (Point2D par)
```

```
{
Point2D tmp = new Point2D(par);
par.x--;
par.y--;
return tmp;
}

// Бинарные операции также обязаны возвращать значения!
public static Point2D operator + (Point2D par1, Point2D par2)
{
return new Point2D(par1.x+par2.x,par1.y+par2.y);
}

// От перемены мест слагаемых сумма ... :
// Point2D + float
public static Point2D operator + (Point2D par1, float val)
{
return new Point2D(par1.x+val,par1.y+val);
}

// float + Point2D
public static Point2D operator + (float val, Point2D par1)
{
return new Point2D(val+par1.x,val+par1.y);
}

// Перегрузка булевских операторов. Это ПАРНЫЕ операторы.
// Объекты типа Point2D приобретают способность судить об истине и лжи!
// А что есть истина? Критерии ИСТИННОСТИ (не путать с истиной)
// могут быть самые разные.
public static bool operator true (Point2D par)
{
if (par.x == 1.0F && par.y == 1.0F) return true;
else                                return false;
}

public static bool operator false (Point2D par)
{
if (par.x == 0.0F && par.y == 0.0F) return false;
else                                return true;
}
//=========================================================================

public static Point2D operator | (Point2D par1, Point2D par2)
{
if (par1) return par1;
if (par2) return par2;
else return new Point2D(-1.0F, -1.0F);
}

public static Point2D operator & (Point2D par1, Point2D par2)
{
if (par1 && par2) return par1;
else return new Point2D(-1.0F, -1.0F);
}
}

/// <summary>
/// Стартовый класс.
/// </summary>

class C1
{

static void Main(string[] args)
{
Console.WriteLine("__________");
Point2D p0 = new Point2D();
Console.WriteLine("__________");
Point2D p1 = new Point2D(GetVal(), GetVal());
Console.WriteLine("__________");
```

```
Point2D p2 = new Point2D(p1);
Console.WriteLine("__________");

// Меняем значение y объекта p1...
Console.WriteLine("**********");
p1.PropertyY = GetVal();
Console.WriteLine("**********");

Console.WriteLine("p0.x == {0}, p0.y == {1}", p0.X,p0.PropertyY);
Console.WriteLine("p1.x == {0}, p1.y == {1}", p1.X,p1.PropertyY);
Console.WriteLine("p2.x == {0}, p2.y == {1}", p2.X,p2.PropertyY);
Console.WriteLine("##########");
p0++;
++p1;
// После объявления операторных функций true и false объекты класса Point2D
// могут могут включаться в контекст условного оператора и оператора цикла.
if (p1) Console.WriteLine("true!");
else Console.WriteLine("false!");

// Конъюнкции и дизъюнкции. Выбирается объект p0 или p1,
// для которого вычисляется истинностное значение.
if (p0 | p1) Console.WriteLine("true!");
if (p0 & p1) Console.WriteLine("true!");
if (p0 || p1) Console.WriteLine("true!");
if (p0 && p1) Console.WriteLine("true!");

for ( ; p2; p2++)
{
Console.WriteLine("p2.x == {0}, p2.y == {1}", p2.X,p2.PropertyY);
}


Console.WriteLine("p0.x == {0}, p0.y == {1}", p0.X,p0.PropertyY);
Console.WriteLine("p1.x == {0}, p1.y == {1}", p1.X,p1.PropertyY);
Console.WriteLine("p2.x == {0}, p2.y == {1}", p2.X,p2.PropertyY);
}

public static float GetVal()
{
float fVal;
string str = null;

while (true)
{
try
{
Console.Write("float value, please >> ");
str = Console.ReadLine();
fVal = float.Parse(str);
return fVal;
}
catch
{
Console.WriteLine("This is not a float value...");
}
}
}
}
}
```

Пример. Свойства и индексаторы

```
using System;

namespace PointModel
{
class SimplePoint
{//============================================================================
float x, y;
public SimplePoint[] arraySimplePoint = null;
```

```
public SimplePoint()
{
x = 0.0F;
y = 0.0F;
}

public SimplePoint(float xKey, float yKey): this()
{
x = xKey;
y = yKey;
}

public SimplePoint(SimplePoint PointKey):this(PointKey.x, PointKey.y)
{
}

public SimplePoint(int N)
{
int i;
if (N > 0 && arraySimplePoint == null)
{
arraySimplePoint = new SimplePoint[N];
for (i = 0; i < N; i++)
{
 arraySimplePoint[i] = new SimplePoint((float)i, (float)i);
}
}

}

}//============================================================================

class MyPoint
{//============================================================================
float x, y;
MyPoint[] arrayMyPoint = null;

int arrLength = 0;
int ArrLength
{
get
{
return arrLength;
}
set
{
arrLength = value;
}
}

bool isArray = false;
bool IsArray // Это свойство только для чтения!
{
get
{
return isArray;
}
}

MyPoint()
{
x = 0.0F;
y = 0.0F;
}


public MyPoint(float xKey, float yKey): this()
{
x = xKey;
y = yKey;
```

```
}

public MyPoint(MyPoint PointKey): this(PointKey.x, PointKey.y)
{
}

public MyPoint(int N)
{
int i;
if (N > 0 && IsArray == false)
{
this.isArray = true;
this.arrLength = N;
arrayMyPoint = new MyPoint[N];
for (i = 0; i < N; i++)
{
       arrayMyPoint[i] = new MyPoint((float)i, (float)i);
}
}
}

bool InArray(int index)
{
if (!IsArray) return false;
if (index >= 0 && index < this.ArrLength) return true;
else return false;
}

// Объявление операторной функции индексации
public MyPoint this[int index]
{
get
{
if (IsArray == false) return null;
if (InArray(index)) return arrayMyPoint[index];
else return null;
}
set
{
if (IsArray == false) return;
if (InArray(index))
{
       arrayMyPoint[index].x = value.x;
       arrayMyPoint[index].y = value.y;
}
else return;
}
}

// Объявление ещё одной (перегруженной) операторной функции индексации.
// В качестве значения для индексации используется символьная строка!
public MyPoint this[string strIndex]
{
get
{
int index = int.Parse(strIndex);
if (IsArray == false) return null;
if (InArray(index)) return arrayMyPoint[index];
else return null;
}
set
{
int index = int.Parse(strIndex);
if (IsArray == false) return;
if (InArray(index))
{
       arrayMyPoint[index].x = value.x;
       arrayMyPoint[index].y = value.y;
}
else return;
}
```

```
}

}//========================================================================


class Class1
{

static void Main(string[] args)
{
SimplePoint spArr = new SimplePoint(8);
SimplePoint wsp = new SimplePoint(3.14F, 3.14F);
SimplePoint wsp0;

spArr.arraySimplePoint[3] = wsp;
try
{
spArr.arraySimplePoint[125] = wsp;
}
catch (IndexOutOfRangeException exc)
{
Console.WriteLine(exc);
}

try
{
wsp.arraySimplePoint[7] = wsp;
}
catch (NullReferenceException exc)
{
Console.WriteLine(exc);
}

try
{
wsp0 = spArr.arraySimplePoint[125];
}
catch (IndexOutOfRangeException exc)
{
Console.WriteLine(exc);
}

wsp0 = spArr.arraySimplePoint[5];
//========================================================================

MyPoint mpArr = new MyPoint(10);
MyPoint wmp = new MyPoint(3.14F, 3.14F);
MyPoint wmp0;

mpArr[3] = wmp;
mpArr[125] = wmp;
wmp[7] = wmp;
wmp0 = mpArr[125];
wmp0 = mpArr[5];
// В качестве индексатора используются строковые выражения.
wmp0 = mpArr["5"];
// В том числе, использующее конкатенацию строк.
wmp0 = mpArr["1"+"2"+"5"];
}
}
}
```

explicit и implicit. Преобразования явные и неявные

Возвращаемся к проблеме преобразования значений одного типа к другому. Что, в частности, актуально при выполнении операций присваивания. И если для элементарных типов проблема преобразования решается (с известными ограничениями, связанными с “опасными” и “безопасными” преобразованиями), то

для вновь объявляемых типов алгоритмы преобразования должны реализовываться разработчиками этих классов.

Логика построения явных и неявных преобразователей достаточно проста. Программист самостоятельно принимает решение относительно того:

- каковым должен быть алгоритм преобразования,
- будет ли этот алгоритм выполняться неявно или необходимо будет явным образом указывать необходимость соответствующего преобразования.

Ниже рассматривается пример, содержащий объявления классов Point2D и Point3D. В классах предусмотрены алгоритмы преобразования значений от одного типа к другому, которые активизируются при выполнении операций присвоения.

```
using System;

// Объявления классов.
class Point3D
{
public int x,y,z;

public Point3D()
{
      x = 0;
      y = 0;
      z = 0;
}

public Point3D(int xKey, int yKey, int zKey)
{
      x = xKey;
      y = yKey;
      z = zKey;
}

// Операторная функция, в которой реализуется алгоритм преобразования
// значения типа Point2D в значение типа Point3D. Это преобразование
// осуществляется НЕЯВНО.
public static implicit operator Point3D(Point2D p2d)
{
      Point3D p3d = new Point3D();
    p3d.x = p2d.x;
      p3d.y = p2d.y;
    p3d.z = 0;
    return p3d;
}
}

class Point2D
{
public int x,y;

public Point2D()
{
      x = 0;
      y = 0;
}

public Point2D(int xKey, int yKey)
{
      x = xKey;
      y = yKey;
}

// Операторная функция, в которой реализуется алгоритм преобразования
// значения типа Point3D в значение типа Point2D. Это преобразование
// осуществляется с ЯВНЫМ указанием необходимости преобразования.
// Принятие решения относительно присутствия в объявлении ключевого
// слова explicit вместо implicit оправдывается тем, что это
// преобразование сопровождается потерей информации. По мнению
// разработчика классов об этом обстоятельстве следует напоминать
// каждый раз, когда данное преобразование случается в программе.
```

```
public static explicit operator Point2D(Point3D p3d)
{
       Point2D p2d = new Point2D();
       p2d.x = p3d.x;
       p2d.y = p3d.y;
       return p2d;
}

}

// Тестовый класс. Здесь всё происходит.
class TestClass
{
 static void Main(string[] args)
 {
  Point2D p2d = new Point2D(125,125);
  Point3D p3d; // Сейчас это только ссылка!
  // Этой ссылке присваивается значение в результате
  // НЕЯВНОГО преобразования значения типа Point2D к типу Point3D
  p3d = p2d;

  // Изменили значения полей объекта.
  p3d.x = p3d.x*2;
  p3d.y = p3d.y*2;
  p3d.z = 125; // Главное - появилась новая информация,
  // которая неизбежно будет потеряна в случае присвоения значения типа Point3D
  // значению типа Point2D. Ключевое слово explicit в объявлении соответствующего
  // метода преобразования приводит к тому, что программист всякий раз вынужден
  // повторять, что он в курсе возможных последствий.
  p2d = (Point2D)p3d;
}
}
```

Наследование

Наследование является одним из принципов ООП.

Основы классификации и реализация механизмов повторного использования и модификации кода. Базовый класс задаёт общие признаки и общее поведение для классов-наследников.

Общие (наиболее общие) свойства и методы наследуются от базового класса, в дополнение к которым добавляются и определяются НОВЫЕ свойства и методы.

Таким образом, прежде всего, наследование реализует механизмы расширения базового класса.

Реализация принципов наследования на примере.

```
using System;
namespace Inheritance_1
{
public class A
{
public int val1_A;
public void fun1_A (String str)
 {
Console.WriteLine("A's fun1_A:" + str);
 }
}

public class B:A
{
public int val1_B;
public void fun1_B (String str)
 {
Console.WriteLine("B's fun1_B:" + str);
 }
}

class Class1
{
static void Main(string[] args)
```

```
 {
B b0 = new B();
// От имени объекта b0 вызвана собственная функция fun1_B.
b0.fun1_B("from B");
// От имени объекта b0 вызвана унаследованная от класса A функция fun1_A.
b0.fun1_A("from B");
 }
}
}
```

Наследование и проблемы доступа

Производный класс наследует от базового класса ВСЁ, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем наследством.

Добавляем в базовый класс частные члены:

```
public class A
{
public int val1_A = 0;
public void fun1_A (String str)
 {
Console.WriteLine("A's fun1_A:" + str);
this.fun2_A("private function from A:");
 }

// При определении переменных
// в C# ничего не происходит без конструктора и оператора new.
// Даже если они и не присутствуют явным образом в коде.
private int val2_A = 0;
private void fun2_A (String str)
      {
Console.WriteLine(str + "A's fun2_A:" + val2_A.ToString());
      }
}
```

И объект-представитель класса B в принципе НЕ может получить доступ к частным данным-членам и функциям-членам класса A. Косвенное влияние на такие данные-члены и функции-члены лишь через публичные функции класса A.

Следует иметь в виду ещё одно важное обстоятельство.

Если упорядочить все (известные) спецификаторы доступа C# по степени их открытости:

Максимальная степень открытости		Минимальная степень открытости
public	…	private

то наследуемый класс не может иметь более открытый спецификатор доступа, чем его предок.

```
protected class A
{
:::::::::
}

public class B:A // Неправильное наследование.
{
:::::::::
}
```

Используем ещё один спецификатор доступа – protected. Этот спецификатор обеспечивает открытый доступ к членам базового класса, но только для производного класса!

```
public class A
{
:::::::::
protected int val3_A = 0;
}

public class B:A
```

```
{
:::::::::
public void fun1_B (String str)
 {
:::::::::
this.val3_A = 125;
 }

}

static void Main(string[] args)
 {
:::::::::
//b0.val3_A = 125; // Это член класса закрыт для внешнего использования!
 }
```

Защищённые члены базового класса доступны для ВСЕХ прямых и косвенных наследников данного класса.

И ещё несколько важных замечаний относительно использования спецификаторов доступа:

в C# структуры НЕ поддерживают наследования. Поэтому спецификатор доступа protected в объявлении данных-членов и функций-членов структур НЕ ПРИМЕНЯЕТСЯ,

спецификаторы доступа действуют и в рамках пространства имён (поэтому и классы в нашем пространстве имён были объявлены со спецификаторами доступа public). Но в пространстве имён явным образом можно использовать лишь один спецификатор – спецификатор public, либо не использовать никаких спецификаторов.

Явное обращение к конструктору базового класса

Продолжаем совершенствовать наши классы A и B. Очередная задача – выяснить способы передачи управления конструктору базового класса при создании объекта-представителя производного класса.

```
using System;

/*private*/ class X
 {
 }

/*public*/ class A
{
public A(){val2_A = 0; val3_A = 0;}
// К этому конструктору также можно обратиться из производного класса.
protected A(int key):this() {val1_A = key;}
// А вот этот конструктор предназначен исключительно
// для внутреннего использования.
private A(int key1,int key2,int key3){val1_A = key1; val2_A = key2; val3_A = key3;}

public int val1_A = 0;
public void fun1_A (String str)
 {
Console.WriteLine("A's fun1_A:" + str);
this.fun2_A("private function from A:");
fun3_A();
 }

private void fun2_A (String str)
{
Console.WriteLine(str + "A's fun2_A:" + val2_A.ToString());
}

protected int val3_A;
private void fun3_A ()
{
A a = new A(1,2,3);
a.fun2_A("Это наше внутреннее дело!");
}
}
```

```
/*public*/ class B:A
{
public B():base(){val1_B = 0;}
public B(int key):base(key){val1_B = key;}
}

class Class1
{
static void Main(string[] args)
{
B b0 = new B(125);
}
}
```

Таким образом, в программе для создания объектов можно применять конструкторы трёх степеней защиты:

public – при создании объектов в рамках данного пространства имён, в методах любого класса-члена данного пространства имён,

protected – при создании объектов в рамках производного класса, в том числе при построении объектов производного класса, а также для внутреннего использования классом-владельцем данного конструктора,

private – применяется исключительно для внутреннего использования классом-владельцем данного конструктора.

Кто строит БАЗОВЫЙ ЭЛЕМЕНТ

Теперь (в рамках того же приложения) построим два новых класса.

```
class X
{
}
class Y:X
{
}

// Это уже в Main()
Y y = new Y();
```

Вся работа по созданию объекта-представителя класса Y при явном отсутствии конструкторов по умолчанию возлагается на КОНСТРУКТОРЫ УМОЛЧАНИЯ – те самые, которые самостоятельно строит транслятор. Особых проблем не будет, если в производном классе явным образом начать объявлять конструкторы.

```
class X
{
}
class Y:X
public Y(int key){}
public Y(){}
}

// Это уже в Main()
Y y0 = new Y();
Y y1 = new Y(125);
```

Отныне в производном классе нет больше конструктора умолчания. Теперь всё зависит от соответствия оператора определения объекта построенному нами конструктору. Объявим в производном классе оба варианта конструкторов. И опять всё хорошо. Конструктор умолчания базового класса (тот, который строится транслятором) продолжает исправно выполнять свою работу.

Проблемы возникнут, если в базовом классе попытаться объявить вариант конструктора с параметрами:

```
class X
{
public X(int key){}
}
```

```
class Y:X
public Y(int key){} // Нет конструктора умолчания базового класса!
public Y(){} // Нет конструктора умолчания базового класса!
}

// Это уже в Main()
Y y0 = new Y();
Y y1 = new Y(125);
```

И здесь транслятор начнёт обижаться на конструкторы производного класса, требуя ЯВНОГО объявления конструктора базового класса БЕЗ параметров. Если вспомнить, что при ЛЮБОМ вмешательстве в дело построения конструкторов транслятор снимает с себя всю ответственность, причина негодования транслятора становится очевидной. Возможны два варианта решения проблемы:

- явным образом заставить работать новый конструктор базового класса,
- самостоятельно объявить новый вариант конструктора без параметров.

```
class X
{
public X(){}
public X(int key){}
}
class Y:X
public Y(int key){}
public Y():base(125){}
}

// Это уже в Main()
Y y0 = new Y();
Y y1 = new Y(125);
```

Переопределение членов базового класса

При объявлении членов производного класса в C# разрешено использовать те же самые имена, которые использовались при объявлении членов базового класса. Это касается как методов, так и данных-членов объявляемых классов.

В этом случае соответствующие члены базового класса считаются переопределёнными.

Следующий простой фрагмент кода содержит объявления базового и производного классов. Особое внимание в этом примере следует уделить ключевому слову new в объявлении одноименных членов в производном классе. В данном контексте это слово выступает в качестве модификатора, который используется для явного обозначения переопределяемых членов базового класса.

При переопределении в производном классе наследуемого члена его объявление в производном классе должно предваряться модификатором new. Если этот модификатор в объявлении производного класса опустить - ничего страшного не произойдёт. Транслятор всего лишь выдаст предупреждение об ожидаемом спецификаторе new в точке объявления переопределяемого члена класса.

Дело в том, что переопределённые члены базового класса при “взгляде” на производный класс “извне” не видны. В производном классе они замещаются переопределёнными членами и непосредственный доступ к этим членам возможен только из функций-членов базового и производного классов. При этом для обращения к переопределённому члену из метода производного класса используется ключевое слово base.

По мнению создателей языка C# тот факт, что ранее (до момента его переопределения) видимый член (базового) класса стал недоступен и невидим извне (в результате его переопределения в производном классе), требует явного дополнительного подтверждения со стороны программиста.

```
using System;
namespace Inheritance_2
{
class X
 {
public X(){}
public X(int key){}
```

```
public int q0;
public int q;
public void fun0()
 {
Console.WriteLine("class X, function fun0()");
q = 125;
 }

public void fun1()
 {
Console.WriteLine("class X, function fun1()");
 }
 }

class Y:X
{
public Y(int key){}
public Y():base(125){}
new public int q; // Если опустить модификатор new -
new public void fun1() // появится предупреждение об ожидаемом new...
 {
base.fun1(); // Обращение к переопределённым членам базового класса.
base.q = 125;
Console.WriteLine("class Y, function fun1()");
 }

static void Main(string[] args)
 {
Y y0 = new Y();
Y y1 = new Y(125);
// А извне переопределённые члены базового класса не видны.
y0.fun1();
y0.q = 100;
y0.fun0();
y0.q0 = 125;
 }
}
}
```

Объявление класса может содержать вложенное объявление класса. C# допускает и такие объявления:

```
class X
{
public class XX
 {
 }
}
```

Так вот одновременное включение явного объявления класса или структуры с одним и тем же именем в базовый и производный классы также требует явной спецификации с помощью модификатора new:

```
class X
 {
public class XX
 {
 }
}

class Y:X
{
new public class XX
 {
 }
}
```

Модификатор new используется при переопределении общедоступных объявлений и защищённых объявлений в производном классе для явного указания факта переопределения.

Наследование и new модификатор

Этот модификатор используется при объявлении класса-наследника в том случае, если надо явным образом скрыть факт наследования объявляемого члена класса.

```
public class MyBaseC
{
public int x;
public void Invoke();
}
```

Объявление члена класса Invoke в наследуемом классе скрывает метод Invoke в базовом классе:

```
public class MyDerivedC : MyBaseC
{
new public void Invoke();
}
```

А вот данное-член x не было скрыто в производном классе соответствующим членом, следовательно, остаётся доступным в производном классе.

Сокрытие имени члена базового класса в производном классе возможно в одной из следующих форм:

Константа, поле, свойство, или внедрённый в класс тип или структура скрывают ВСЕ одноименные члены базового класса.

Внедрённый в класс или структуру метод скрывает в базовом классе свойства, поля, типы, обозначенные тем же самым идентификатором. Также в базовом классе скрываются одноимённые методы с той же самой сигнатурой.

Объявляемые в производном классе индексаторы, скрывают одноименные индексаторы в базовом классе с аналогичной сигнатурой.

В производном классе одновременное использование члена базового класса и его переопределённого потомка является ошибкой.

Полное квалифицированное имя. Примеры использования

В этом примере базовый и BC и производный DC классы используют одно и то же имя для обозначения объявляемого в обоих классах члена типа int. new модификатор подчеркивает факт недоступности члена x базового класса в производном классе. Однако из производного класса всё-таки можно обратиться к переопределённому полю базового класса с использованием полного квалифицированного имени.

```
using System;
public class BC
{
public static int x = 55;
public static int y = 22;
}

public class DC : BC
{
new public static int x = 100; // Переопределили члена базового класса
public static void Main()
 {
// Доступ к переопределённому члену x:
Console.WriteLine(x);
// Доступ к члену базового класса x:
Console.WriteLine(BC.x);
// Доступ к члену y:
Console.WriteLine(y);
 }
}
```

В производном классе DC переопределяется вложенный класс C. В рамках производного класса легко можно создать объект-представитель вложенного

переопределённого класса C. Для создания аналогичного объекта-представителя базового класса, необходимо использовать полное квалифицированное имя.

```
using System;
public class BC
{
public class C
 {
public int x = 200;
public int y;
 }
}

public class DC:BC
{
new public class C // Вложенный класс базового класса скрывается
 {
public int x = 100;
public int y;
public int z;
 }

public static void Main()
 {
// Из производного класса виден переопределённый вложенный класс:
C s1 = new C();
// Полное имя используется для доступа к вложенному в базовый класс:
BC.C s2 = new BC.C();
Console.WriteLine(s1.x);
Console.WriteLine(s2.x);
 }
}
```

Прекращение наследования. sealed спецификатор

Принцип наследования допускает неограниченную глубину иерархии наследования. Производный класс, являющийся наследником базового класса, может в свою очередь сам оказаться в роли базового класса. Однако не всегда продолжение цепочки предок-потомок может оказаться целесообразным.

Если при разработке класса возникла ситуация, при котором дальнейшее совершенствование и переопределение возможностей класса в деле решения специфических задач окажется нецелесообразным (сколько можно заниматься переопределением функций форматирования), класс может быть закрыт для дальнейшего наследования. Закрытие класса обеспечивается спецификатором sealed. При этом закрываться для наследования может как класс целиком, так и отдельные его члены.

```
sealed class X
{

}

class Y:X // Наследование от класса X невозможно
{

}
```

А так закрывается для переопределения функция-член класса:

```
class X
{
sealed public void f0()
{

}
}

class Y:X
{
public void f0(){} // Наследование (переопределение) f0, объявленной в X запрещено!
```

```
}
```

Абстрактные функции и абстрактные классы

При реализации принципа наследования базовый класс воплощает НАИБОЛЕЕ ОБЩИЕ черты разрабатываемого семейства классов. Поэтому на этапе разработки базового класса часто бывает достаточно лишь обозначить множество функций, которые будут определять основные черты поведения объектов-представителей производных классов.

Если базовый класс объявляется исходя из следующих предпосылок:

- базовый класс используется исключительно как основа для объявления классов-наследников,
- базовый класс никогда не будет использован для определения объектов,
- часть функций-членов (возможно, все) базового класса в обязательном порядке переопределяется в производных классах,

то определение таких функций даже в самых общих чертах (заголовок функции и тело, содержащее вариант оператора return) в базовом классе лишено всякого смысла.

```
class X
{
public int f0()
{
// Если в соответствии с замыслом разработчика, этот код ВСЕГДА будет недоступен,
// то зачем он в принципе нужен?
return 0;
 }
}

class Y:X
{
new public void f0()
{
::::::::::
// Здесь размещается код переопределённой функции.
::::::::::
return 0;
}
}
```

Такой код никому не нужен и C# позволяет избегать таких странных конструкций. Вместо переопределяемой “заглушки” можно использовать объявление абстрактной функции.

Синтаксис объявления абстрактной функции предполагает использование ключевого слова abstract и полное отсутствие тела. Объявление абстрактной функции завершается точкой с запятой.

Класс, содержащий вхождения абстрактных (хотя бы одной!) функций также должен содержать в заголовке объявления спецификатор abstract.

В производном классе соответствующая переопределяемая абстрактная функция обязательно должна включать в заголовок функции спецификатор override. Его назначение – явное указание факта переопределения абстрактной функции.

Абстрактный класс фактически содержит объявления нереализованных функций базового класса. На основе абстрактного класса невозможно определить объекты. Попытка создания соответствующего объекта-представителя абстрактного класса приводит к ошибке, поскольку в классе не определены алгоритмы, определяющие поведение объект.

```
abstract class X // Абстрактный класс с одной абстрактной функцией.
{
public abstract int f0();
}

class Y:X
{
// Переопределение абстрактной функции должно
// содержать спецификатор override.
```

```
public override void f0()
{
 ::::::::::
return 0;
}
}
::::::::::
static void Main(string[] args)
{
X x = new X(); // NO!
Y y0 = new Y(125);
// Работает переопределённая абстрактная функция!
y0.f0();
}
```

Ещё пример

```
using System;
namespace Interface01
{
// Абстрактный класс.
abstract class aX1
{
public int xVal;
// Его конструкторы могут использоваться при построении
// объектов класов - наследников.
public aX1(int key)
{
Console.WriteLine("aX1({0})...", key);
xVal = key;
}

public aX1()
{
Console.WriteLine("aX1()...");
xVal = 0;
}
public void aX1F0(int xKey)
{
xVal = xKey;
}

public abstract void aX2F0();
}

class bX1:aX1
{
new public int xVal;

public bX1():base(10)
{
xVal = 125;
Console.WriteLine("bX1():base(10)... xVal=={0},base.xVal=={1}...", xVal, base.xVal);
}

public bX1(int key):base(key*10)
{
xVal = key;
Console.WriteLine("bX1({0}):base({1})...", xVal, base.xVal);
}

public override void aX2F0()
{
xVal = xVal*5;
base.xVal = base.xVal*100;
}

}

class Class1
{
```

```
static void Main(string[] args)
{
// Ни при каких обстоятельствах не может служить основой для
// построения объектов. Даже если не содержит ни одного объявления
// абстрактной функции.
//aX1 x = new aX1();

bX1 x0 = new bX1();
x0.aX1F0(10); // Вызвали неабстрактную функцию базового абстрактного класса.
bX1 x1 = new bX1(5);
x1.aX2F0();//Вызвали переопределённую функцию. В базовом классе это
// абстрактная функция.
}
}
}
```

Ссылка на объект базового класса

Это универсальная ссылка для любого объекта производного типа, наследующего данный базовый класс.

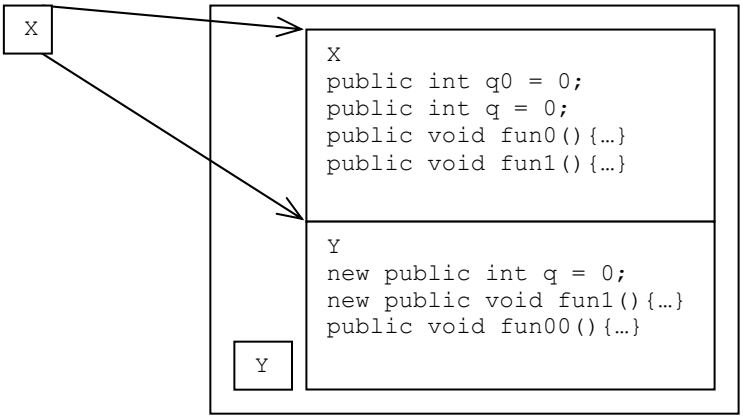
```
using System;
namespace Inheritance_3
{
class X
{//=====================================================
public int q0 = 0;
public int q = 0;

public void fun0()
{
Console.WriteLine("class X, function fun0()");
}

public void fun1()
 {
Console.WriteLine("class X, function fun1()");
 }
}//=====================================================

class Y:X
{//=====================================================
new public int q = 0;
new public void fun1()
 {
Console.WriteLine("class Y, function fun1()");
 }

public void fun00()
 {
Console.WriteLine("class Y, fun00()");
 }
}//=====================================================

class Z:X
{//=====================================================
new public int q = 0;
new public void fun1()
 {
Console.WriteLine("class Z, function fun1()");
 }

public void fun00()
 {
Console.WriteLine("class Z, fun00()");
 }
}//=====================================================

class StartClass
{//=====================================================
static void Main(string[] args)
```

```
{
X x = null; // Просто ссылка!
// Объекты-представители производных классов-наследников.
Y y = new Y(); y.fun0(); y.fun00(); y.fun1();
Z z = new Z(); z.fun0(); z.fun00(); z.fun1();
// Настройка базовой ссылки.
x = y; x.fun0(); x.fun1(); x.q = 100; x.q0 = 125;
x = z; x.fun0(); x.fun1(); x.q = 100; x.q0 = 125;
}
}//================================================

}
```

Результат:

```
class X, fun0()
class Y, fun00()
class Y, fun1()
class X, fun0()
class Z, fun00()
class Z, fun1()
class X, fun0()
class X, fun1()
class X, fun0()
class X, fun1()
```

Вопросов не будет, если рассмотреть схему объекта-представителя класса Y.



Вот что видно от ссылки на объект класса X, настроенного на объект-представитель производного класса. Схема объекта-представителя класса Z и соответствующий "вид" от ссылки x выглядят аналогичным образом.

Операции is и as

Вот код, иллюстрирующий особенности применения операций is и as.

```
using System;

namespace derivation01
{
// Базовый класс...
class X
{//_________________________________.
public int f1(int key)
{
 Console.WriteLine("X.f1");
 return key;
}
}//_________________________________.

// Производный...
class Y:X
```

```
{//_____________________________.
new public int f1(int key)
{
 Console.WriteLine("Y.f1");
 base.f1(key);
 return key;
}

public int yf1(int key)
{
 Console.WriteLine("Y.yf1");
 return key;
}

}//_____________________________.

// Производный...
class Z:X
{//_____________________________.
public int zf1(int key)
{
 Console.WriteLine("Z.zf1");
 return key;
}
}//_____________________________.


class Class1
{
static void Main(string[] args)
{
  int i;
  // Всего лишь ссылки на объекты...
  X x;
  Y y;
  Z z;

  Random rnd = new Random();

// И вот такой тестовый пример позволяет выявить особенности применения
// операций is и as.
for (i = 0; i < 10; i++)
{//=============================================================================.
 // Ссылка на объект базового класса случайным образом
 // настраивается на объекты производного класса.
 if (rnd.Next(0,2) == 1)
   x = new Y();
 else
    x = new Z();

// Вызов метода f1 не вызывает проблем.
// Метод базового класса имеется у каждого объекта.
x.f1(0);

// А вот вызвать метод, объявленный в производном классе
// (с использованием операции явного приведения типа)
// удаётся не всегда. Метод yf1 был объявлен лишь в классе Y.
// Ниже операция is принимает значение true лишь в том случае,
// если ссылка на объект базового класса была настроена на объект класса Y.
if (x is Y)
{
 ((Y)x).yf1 (0); // И только в этом случае можно вызвать метод,
                 // объявленный в Y.
}
else
{
 ((Z)x).zf1 (1); // А в противном случае попытка вызова yf1
                 // привела бы к катастрофе.
  try
 {
   ((Y)x).yf1 (0);
```

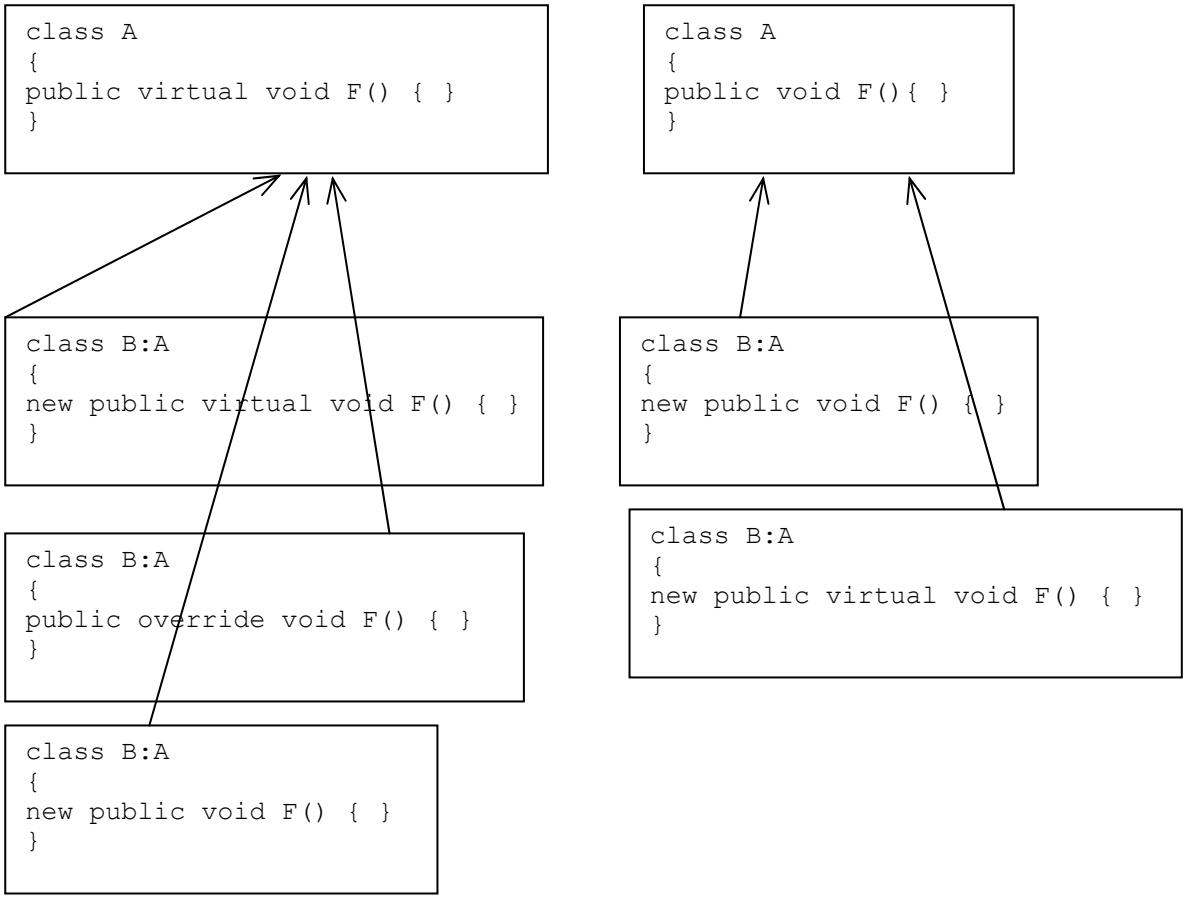
```
  }
  catch (Exception ex)
  {
   Console.WriteLine("-1-" + ex.ToString());
  }
}

// А теперь объект, адресуемый ссылкой на базовый класс надо попытаться
// ПРАВИЛЬНО переадресовать на ссылку соответствующего типа. И это тоже
// удаётся сделать не всегда. Явное приведение может вызвать исключение.

try
{
 z = (Z)x;
}
catch (Exception ex)
{
 Console.WriteLine("-2-" + ex.ToString());
}

try
{
 y = (Y)x;
}
catch (Exception ex)
{
 Console.WriteLine("-3-" + ex.ToString());
}

// Здесь помогает операция as.
// В случае невозможности переадресации соответствующая ссылка оказывается
// установленной в null. А эту проверку выполнить проще...

if (rnd.Next(0,2) == 1)
{
 z = x as Z;
 if (z != null) z.zf1(2);
 else           Console.WriteLine("?????");
}
else
{
 y = x as Y;
 if (y != null) y.yf1(3);
 else           Console.WriteLine("!!!!!");
}
}//=============================================================================.

}
}
}
```

Виртуальные функции. Принцип полиморфизма

Основа реализации принципа полиморфизма – наследование. Ссылка на объект базового класса, настроенная на объект производного может обеспечить выполнение методов ПРОИЗВОДНОГО класса, которые НЕ БЫЛИ ОБЪЯВЛЕНЫ В БАЗОВОМ КЛАССЕ. При реализации принципа полиморфизма происходят вещи, которые не укладываются в ранее описанную схему.

Одноимённые функции с одинаковой сигнатурой в базовом и производном классах: между ними может быть установлено отношение замещения. Замещаемая функция базового класса должна при этом дополнительно специфицироваться спецификатором virtual.

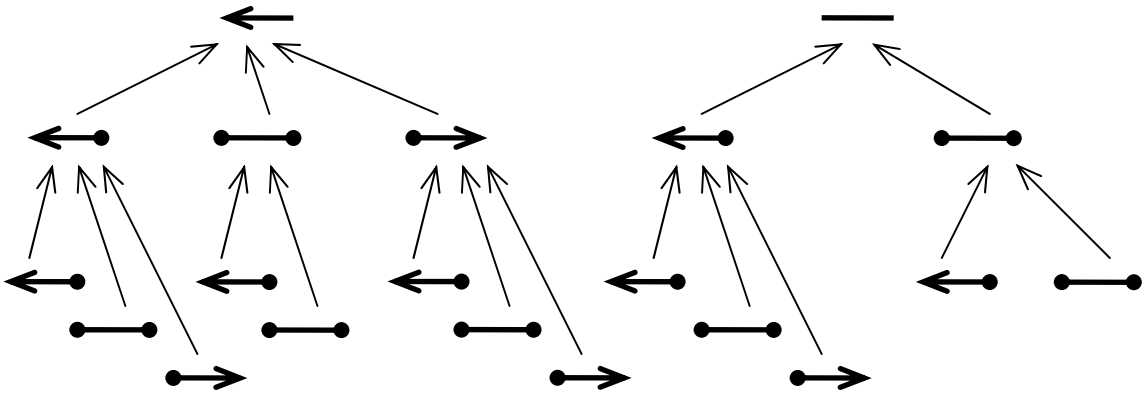
Отношение замещения между функциями базового и производного класса устанавливается, если соответствующая (одноименная функция с соответствующей сигнатурой) функция производного класса специфицируется дополнительным спецификатором override.



Введём следующие обозначения и построим схему наследования.

Спецификация и заголовок функции- члена	Уровень наследования	Обозначение
public void F(){ }	0	
public virtual void F() { }	0	
new public virtual void F() { }	1..N	
new public void F() { }	1..N	
public override void F() { }	1..N	

Схема возможных вариантов объявления методов в иерархии наследования трёх уровней.



Наконец, замечательный пример,

```
using System;
namespace Implementing
{
class A
{
public virtual void F() { Console.WriteLine("A");}
}

class B:A
{
public override void F() { Console.WriteLine("B"); }
}

class C:B
{
new public virtual void F() { Console.WriteLine("C"); }
}

class D:C
{
public override void F() { Console.WriteLine("D"); }
}

class Starter
{
static void Main(string[] args)
{
D d = new D();
C c = d;
B b = c;
A a = b;
d.F(); /* D */
c.F(); /* D */
b.F(); /* B */
a.F(); /* B */
}
}
}
```

в котором становится ВСЁ понятно, если ПОДРОБНО нарисовать схемы классов, структуру объекта-представителя класса D и посмотреть, КАКАЯ ссылка ЗА КАКОЕ место этот самый объект удерживает.

Интерфейсы

Фактически это те же самые абстрактные классы, НЕ СОДЕРЖАЩИЕ объявлений данных – членов и объявлений ОБЫЧНЫХ функций.

Все без исключения функции-члены интерфейса – абстрактные. Поэтому интерфейс объявляется с особым ключевым словом interface, а функции интерфейса, несмотря на свою "абстрактность" объявляются без ключевого слова abstract.

Основное отличие интерфейса от абстрактного класса заключается в том, что производный класс может наследовать одновременно несколько интерфейсов.

```
using System;

namespace Interface01
{

// Интерфейсы.
// Этот интерфейс характеризуется уникальными
// именами объявленных в нём методов.
interface Ix
{
void IxF0(int xKey);
void IxF1();
}
```

```
// Пара интерфейсов, содержащих объявления одноимённых методов
// с одной и той же сигнатурой.
interface Iy
{
void F0(int xKey);
void F1();
}

interface Iz
{
void F0(int xKey);
void F1();
}

// А этому интерфейсу уделим особое внимание.
// Он содержит тот же набор методов, но в производном классе этот интерфейс
// будет реализован явным образом.
interface Iw
{
void F0(int xKey);
void F1();
}

// В классе TestClass наследуются интерфейсы...
class TestClass:Ix
               ,Iy
               ,Iz
               ,Iw
{
public int xVal;

// Конструкторы.
public TestClass()
{
      xVal = 125;
}

public TestClass(int key)
{
      xVal = key;
}


// Реализация функций интерфейса Ix.
// Этот интерфейс имеет специфические названия функций.
// В данном пространстве имён его реализация неявная и однозначная.
public void IxF0(int key)
{
      xVal = key*5;
      Console.WriteLine("IxF0({0})...", xVal);
}

public void IxF1()
{
      xVal = xVal*5;
      Console.WriteLine("IxF1({0})...", xVal);
}

// Реализация интерфейсов Iy и Iz в классе TestClass неразличима.
// Это неявная неоднозначная реализация интерфейсов.
// Однако, неважно, чью конкретно функцию реализуем. Оба интерфейса довольны...
public void F0(int xKey)
{
      xVal = (int)xKey/5;
      Console.WriteLine("(Iy/Iz)F0({0})...", xVal);
}

public void F1()
{
      xVal = xVal/5;
      Console.WriteLine("(Iy/Iz)F1({0})...", xVal);
```

```
}

// А это явная непосредственная реализация интерфейса Iw.
// Таким образом, класс TestClass содержит ТРИ варианта реализации функций интерфейсов
// с одной и той же сигнатутой. Два варианта реализации неразличимы. Третий
// (фактически второй) вариант реализации отличается квалифицированными именами.
       void Iw.F0(int xKey)
{
       xVal = xKey+5;
       Console.WriteLine("Iw.F0({0})...", xVal);
}

       void Iw.F1()
{
       xVal = xVal-5;
       Console.WriteLine("Iw.F1({0})...", xVal);
}

public void bF0()
{
       Console.WriteLine("bF0()...");
}

}

class Class1
{
static void Main(string[] args)
{

TestClass x0 = new TestClass();
TestClass x1 = new TestClass(5);

x0.bF0();

// Эти методы представляют собой неявную ОДНОЗНАЧНУЮ реализацию
// интерфейса Ix.
x0.IxF0(10);
x1.IxF1();

// Эти методы представляют собой неявную НЕОДНОЗНАЧНУЮ реализацию
// интерфейсов Iy и Iz.
x0.F0(5);
x1.F1();

// А вот вызов функций с явным приведением к типу интерфейса.
// Собственный метод класса bF0() при подобных преобразованиях
// не виден.
(x0 as Iy).F0(7);
(x1 as Iz).F1();

// А теперь настраиваем ссылки различных типов интерфейсов
// на ОДИН И ТОТ ЖЕ объект-представитель класса TestClass.
// И через "призму" интерфейса всякий раз объект будет
// выглядеть по-разному.
Console.WriteLine("==========Prism test==========");

Console.WriteLine("==========Ix==========");
Ix ix = x1;
ix.IxF0(5);
ix.IxF1();

Console.WriteLine("==========Iy==========");
Iy iy = x1;
iy.F0(5);
iy.F1();

Console.WriteLine("==========Iz==========");
Iz iz = x1;
iz.F0(5);
iz.F1();
```

```
Console.WriteLine("==========Iw==========");
Iw iw = x1;
iw.F0(10);
iw.F1();
}
}
}
```

Преимущества программирования с использованием интерфейсов проявляются в том случае, когда ОДНИ И ТЕ ЖЕ ИНТЕРФЕЙСЫ наследуются РАЗНЫМИ классами. И здесь всё определяется спецификой данной конкретной реализации.

```
using System;

namespace Interface02
{
// Объявляются два интерфейса, каждый из которых содержит объявление
// одноименного метода с единственным параметром соответствующего типа.

interface ICompare0
{
bool Eq(ICompare0 obj);
}

interface ICompare1
{
bool Eq(ICompare1 obj);
}

// Объявляются классы, наследующие оба интерфейса.
// В каждом из классов реализуются функции интерфейсов.
// В силу того, что объявленные в интерфейсах методы одноименные,
// в классах применяется явная реализация методов интерфейсов.
//_____________________________________________________________________________.
class C0:ICompare0,ICompare1
{

public int commonVal;
int valC0;

public C0(int commonKey, int key)
{
commonVal = commonKey;
valC0 = key;
}

// Метод реализуется для обеспечения сравнения объектов СТРОГО одного типа - C0.
bool ICompare0.Eq(ICompare0 obj)
{
C0 test = obj as C0;
if (test == null) return false;
if (this.valC0 == test.valC0)
       return true;
else
       return false;
}


// Метод реализуется для обеспечения сравнения объектов разного типа.
bool ICompare1.Eq(ICompare1 obj)
{
C1 test = obj as C1;
if (test == null) return false;
if (this.commonVal == test.commonVal)
       return true;
else
       return false;
}

}
```

```
//_____________________________________________________________________________.

class C1:ICompare0,ICompare1
{
public int commonVal;
string valC1;

public C1(int commonKey, string key)
{
commonVal = commonKey;
valC1 = string.Copy(key);
}

// В классе C1 при реализации функции интерфейса ICompare0 реализован
// метод сравнения, который обеспечивает сравнение как объектов типа C1,
// так и объектов типа C0.
bool ICompare0.Eq(ICompare0 obj)
{
C1 test;

// Попытка приведения аргумента к типу C1.
// В случае успеха - сравнение объектов по специфическому признаку,
// который для данного класса представлен строковой переменной valC1.
// В случае неуспеха приведения (очевидно, что сравниваются объекты разного типа)
// предпринимается попытка по второму сценарию (сравнение объектов разных типов).
// Таким образом, в рамках метода, реализующего один интерфейс,
// используется метод второго интерфейса. Разумеется, при явном приведении
// аргумента к типу второго интерфейса.

 test = obj as C1;
 if (test == null)
       return ((ICompare1)this).Eq((ICompare1)obj);

if (this.valC1.Equals(test.valC1))
 return true;
else
 return false;
}


// Метод реализуется для обеспечения сравнения объектов разного типа.
bool ICompare1.Eq(ICompare1 obj)
{
 C0 test = obj as C0;
 if (test == null) return false;
 if (this.commonVal == test.commonVal)
  return true;
 else
 return false;
}
}

//=========================================================================
// Место, где порождаются и сравниваются объекты.
class Class1
{
static void Main(string[] args)
{
  C0 x1 = new C0(0,1);
  C0 x2 = new C0(1,1);

  // В выражениях вызова функций-членов интерфейсов НЕ ТРЕБУЕТСЯ явного
  //   приведения значения параметра к типу интерфейса.

   // Сравнение объектов-представителей одного класса (C0).
   if ((x1 as ICompare0).Eq(x2))
        Console.WriteLine("Yes!");
   else
        Console.WriteLine("No!");

   C1 y1 = new C1(0,"1");
```

```
   C1 y2 = new C1(1,"1");

   // Сравнение объектов-представителей одного класса (C1).
   if ((y1 as ICompare0).Eq(y2))
      Console.WriteLine("Yes!");
   else
      Console.WriteLine("No!");

    // Попытка сравнения объектов-представителей разных классов.
    if (((ICompare0)x1).Eq(y2))
      Console.WriteLine("Yes!");
    else
      Console.WriteLine("No!");


    if ((x1 as ICompare1).Eq(y2))
      Console.WriteLine("Yes!");
    else
      Console.WriteLine("No!");


    if (((ICompare1)y2).Eq(x2))
      Console.WriteLine("Yes!");
    else
      Console.WriteLine("No!");

   // Здесь будут задействован метод сравнения, реализованный
   // в классе C0 по интерфейсу ICompare0, который не является универсальным.
   // Отрицательный результат может быть получен не только по причине неравенства
   // значений сравниваемых величин, но и по причине несоответствия типов  операндов.
   Console.WriteLine("__________x2==y2__________");
   if ((x2 as ICompare0).Eq(y2))
      Console.WriteLine("Yes!");
   else
      Console.WriteLine("No!");

   // А здесь вероятность положительного результата выше, поскольку в классе
   // C1 метод интерфейса ICompare0 реализован как УНИВЕРСАЛЬНЫЙ. И это значит,
   // что данный метод никогда не вернёт отрицательного значения по причине
   // несоответствия типов операндов.
   Console.WriteLine("__________y2==x2__________");
   if ((y2 as ICompare0).Eq(x2))
      Console.WriteLine("Yes!");
   else
      Console.WriteLine("No!");
}
}
}
```

Делегаты

Класс, структура, интерфейс, перечисление, делегат – это всё разнообразные категории классов. Каждая категория имеет свои особенности объявления, своё назначение и строго определённую область применения.

Делегат – это тоже класс.

Объявление класса делегата начинается ключевым словом delegate и выглядит следующим образом:

```
ОбъявлениеКлассаДелегата ::=
                              [СпецификаторДоступа]
                                delegate
                                 СпецификаторВозвращаемогоЗначения
                                  ИмяКлассаДелегата
                                    (СписокПараметров);
```

При этом

```
СпецификаторВозвращаемогоЗначения ::= ИмяТипа
ИмяКлассаДелегата ::= Идентификатор
```

а синтаксис элемента СписокПараметров аналогичен списку параметров функции. Но сначала примеры объявления классов делегатов.

```
delegate int ClassDelegate(int key);
delegate void XXX(int intKey, float fKey);
```

Подобие объявления класса делегата и заголовка функции не случайно.

Класс делегат способен порождать объекты. При этом назначение объекта-представителя класса-делегата заключается в представлении методов (функций-членов) РАЗЛИЧНЫХ классов.

Тех классов, которые оказываются видимыми из пространства имён, содержащих объявление данного класса-делегата. Объект этого класса способен представлять ЛЮБЫЕ (статические и нестатические) функции-члены классов, лишь бы спецификация их возвращаемого значения и список параметров соответствовали бы характеристикам данного класса-делегата.

В основе любой класс-делегат является многоадресным (multicast delegate). Это означает, что в ходе выполнения приложения делегат способен запоминать ссылки на произвольное количество функций-членов. Эта многоадресность обеспечивается внутренним списком, в который и заносятся ссылки на разнообразные функции-члены, соответствующие заданной сигнатуре и спецификации возвращаемого значения.

Класс-делегат является производным от System.MulticastDelegate, что и позволяет объектам-представителям этого класса реализовывать достаточно сложные алгоритмы настройки на функции-члены различных классов.

Унаследованные свойства и методы классов-делегатов.

Свойства и методы	Назначение
Method	Свойство. Возвращает имя метода, на который указывает делегат.
Target	Свойство. Возвращает имя класса, если делегат указывает на нестатический метод класса. Возвращает значение типа null, если делегат указывает на статический метод.
Combine(), operator+(), operator+=(), operator-(), operator-=()	Функция и операторные функции. Обеспечивают реализацию многоадресного делегата. Работа с операторными функциями смотрится как прибавление и вычитание ДЕЛЕГАТОВ. Арифметика делегатов.
GetInvocationList()	Основываясь на внутреннем списке ссылок на функции, строится соответствующий массив описателей типов функций. Попытка применения метода к пустому делегату приводит к возникновению исключения.
object DynamicInvoke (object[] args)	В соответствии со списком ссылок обеспечивается выполнение функций, на которые был настроен делегат.
static Remove()	Статический метод, обеспечивающий удаление элементов внутреннего списка ссылок на функции.

Пример использования возможностей делегата представлен ниже.

```
using System;
namespace Delegates_1
{
// Класс делегат. Его объявление соответствует типу функции,
// возвращающей значение int с одним параметром типа int.
delegate int xDelegate(int key);

//=========================================
class ASD
{
// Делегат как член класса.
public xDelegate d;

// А вот свойство, которое возвращает ссылку на делегат.
public xDelegate D
{
get
{
```

```
return d;
}
}
}

//=========================================================
// Класс, содержащий функцию-член, на которую может
// быть настроен делегат-представитель класса xDelegate.
//=========================================================
class Q
{

public int QF(int key)
{
Console.WriteLine("int QF({0})", key);
return key;
}

// А вот функция, в которой делегат используется как параметр!
public void QQQ(int key, xDelegate par)
{
Console.WriteLine("Делегат как параметр!");

// И этот делегат используется по назначению - обеспечивает вызов
// НЕКОТОРОЙ функции. Какой функции? А неизвестно какой! Той,
// на которую был настроен делегат перед тем, как его ссылка была передана
// в функцию QQQ. Здесь не интересуются тем, что за функция пришла.
// Здесь запускается ЛЮБАЯ функция, на которую настраивался делегат.
par(key);
}
}

//=========================================
// Стартовый класс. Также содержит пару пригодных для настройки
// делегата функций.
class StartClass
{

// Одна статическая...
public static int StartClassF0(int key)
{
Console.WriteLine("int StartClassF0({0})", key);
return key;
}

// Вторая нестатическая...
public int StartClassF1(int key)
{
Console.WriteLine("int StartClassF1({0})", key);
return key;
}

// Процесс пошёл!
static void Main(string[] args)
{//=================================================================

// Ссылка на делегат.
xDelegate localDelegate;
int i, n;

// Объект - представитель класса StartClass.
StartClass sc = new StartClass();

// Объект - представитель класса Q.
Q q = new Q();

// Объект - представитель класса ASD.
ASD asd = new ASD();

// Статическая функция-член класса - в списке делегата.
// Поскольку делегат настраивается непосредственно в
```

```
// класса, которому принадлежит данная статическая функция,
// здесь обходимся без дополнительной спецификации имени
// функции.

asd.d = new xDelegate(StartClassF0);
// Вот показали имя метода, на который настроен делегат.

// Попытались показать имя класса – хозяина метода, на который
// настроили делегат. Ничего не получится, поскольку метод
// статический.
if (asd.d.Target != null) Console.WriteLine(asd.d.Target);

// неСтатическая функция-член класса - в списке делегата.
// добавляется к списку функций делегата посредством
// операторной функции +=.

asd.d += new xDelegate(sc.StartClassF1);
Console.WriteLine(asd.d.Method.ToString());
if (asd.d.Target != null) Console.WriteLine(asd.d.Target);

// Делегат также включил в список функцию-член класса Q.

asd.d += new xDelegate(q.QF);
Console.WriteLine(asd.d.Method.ToString());
if (asd.d.Target != null) Console.WriteLine(asd.d.Target);


// Делегат разряжается последовательностью
// вызовов разнообразных функций.

// Либо так.
asd.d(125);

// Либо так. Параметр
// при этом пришлось упаковать в одномерный массив типа object
// длиной в 1.
asd.d.DynamicInvoke(new object[]{0});

// Также есть возможность удаления функции из списка делегата.
// Для этого воспользуемся локальным делегатом.
localDelegate = new xDelegate(q.QF);
asd.d -= localDelegate;

asd.d(725);
// А теперь опять добавим функцию в список делегата.
asd.d += localDelegate;

asd.d(325);
Console.WriteLine(asd.d.Method.ToString());

// А теперь - деятельность по построению массива описателей типа ссылок.
// Преобразование ОДНОГО объекта, содержащего массив ссылок на функции
// в массив объектов, содержащих по одной ссылке на функцию.
// Таким образом, для каждой ссылки на функцию делегата в рамках массива
// делегатов строится свой собственный делегат.
// Естественно что элемент массива, уже не является Multicast'овым делегатом.
// Поэтому с его помощью можно выполнить ОДНУ функцию Multicast’ового делегата.

Console.WriteLine(“—In array!------------------------- ”);
// Вот ссылка на массив делегетов. Сюда будем сгружать содержимое Multicast’а.
Delegate[] dlgArray;
try
{
dlgArray = asd.d.GetInvocationList();
}
catch (System.NullReferenceException e)
{
Console.WriteLine(e.Message + “:Delegate is empty”);
return;
}
```

```
// Прочитали содержимое массива описателей типа ссылок.
for (i = 0, n = dlgArray.Length; i < n; i++)
{
Console.WriteLine(dlgArray[i].Method.ToString());
// А можно и так, без помощи метода ToString()… С тем же результатом.
//Console.WriteLine(dlgArray[i].Method);
}

// Вот как запускается ОДНА функция из списка ссылок на функции!
// Используются разные варианты запуска.
Console.WriteLine("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
dlgArray[1].DynamicInvoke(new object[]{75});
((xDelegate)(dlgArray[2]))(123);
Console.WriteLine("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");


// Используем этот массив для преобразования массива ссылок делегата.
// Выкидываем из массива один элемент.
// Возвращаемое значение (НОВЫЙ делегат с изменённым списком)
// может быть "поймано" и другим делегатом!
asd.d = (xDelegate)xDelegate.Remove(asd.d,dlgArray[1]);
// Таблица ссылок модифицированного делегата сократилась!
Console.WriteLine("Таблица ссылок сократилась! ");
asd.d(125);

// Через свойство получили ссылку на делегат.
xDelegate dFROMasd = asd.D;

// Таблица ссылок опустошается!
// Ещё бы! Здесь из списка методов одного делегата
// удаляются ВСЕ делегаты, которые встречаются в списке
// методов второго делегата. Буквально: удали из этого списка
// ВСЕ методы, которые в этом списке встретятся!
// Что-то из жизни того парня, который вытягивал сам себя
// за волосы из болота! И если бы НОВЫЙ делегат не был бы
// перехвачен на другую ссылку – весь исходный список
// методов Multicast'ового делегата asd.d был бы потерян.
dFROMasd = (xDelegate)xDelegate.RemoveAll(asd.d,asd.d);

// В чём и можно убедиться вот таким способом!
try
{
dlgArray = dFROMasd.GetInvocationList();
}
catch (System.NullReferenceException e)
{
Console.WriteLine(e.Message + ":Delegate is empty");
}

// Но только не исходный делегат!
Console.WriteLine("Но только не исходный делегат!");
// В чём и можно убедиться вот таким способом!
try
{
dlgArray = asd.d.GetInvocationList();
}
catch (System.NullReferenceException e)
{
Console.WriteLine(e.Message + ":Delegate is empty");
}

// Вот! исходный объект класса-делегата не пустой!
asd.d(125);
q.QQQ(75,asd.d);

}//================================================================
}
}
```

События

Если в классе объявить член-событие, то объект-представитель этого класса сможет уведомлять объекты других классов о данном событии.

Класс, содержащий объявление события поддерживает:

- регистрацию объектов-представителей других классов, "заинтересованных" в получении уведомления о событии,
- отмену регистрации объектов, получающих уведомление о событии,
- управление списком зарегистрированных объектов и процедурой уведомления о событии.

Реализация механизма управления по событиям предполагает два этапа:

- объявление делегата,
- объявление события.

Пара примеров иллюстрирует технику применения событий и функциональные фозможности объекта события.

Первый пример…

```
using System;

namespace Events01
{
// Делегат.
delegate int ClassDLG (int key);

// Классы, содержащие методы - обработчики событий.
class C1
{//================================================================
public int C1f (int key)
{//________________________________________________________________
  Console.WriteLine("C1.C0f");
  return key*2;
}//________________________________________________________________
}//================================================================

class C2
{//================================================================
public int C2f (int key)
{//________________________________________________________________
 Console.WriteLine("C2.C0f");
 return key*2;
}//________________________________________________________________
}//================================================================

// Стартовый класс.
class C0
{//================================================================
static event ClassDLG ev0;
static event ClassDLG ev1;

public static int C0F (int key)
{//________________________________________________________________
 Console.WriteLine("C0.C0F");
 return key*2;
}//________________________________________________________________

public int C0f (int key)
{//________________________________________________________________
 Console.WriteLine("C0.C0f");
 return key*2;
}//________________________________________________________________


static void Main(string[] args)
{//________________________________________________________________
int i;
```

```
Random rnd = new Random();

 // Создаётся объект-представитель класса C1.
 C1 obj1 = new C1();

 // Подписная компания. Подписываются ВСЕ!

 // В событии ev0 заинтересован:
 //   класс C0 статическим методом C0F.
       ev0 += new ClassDLG(C0.C0F);

// В событии ev1 заинтересован:
//   класс C0 методом C0f, вызываемым от имени безымянного объекта,
//   класс C1 методом C1f, вызываемым от имени объекта obj1,
//   класс C2 методом C2f, вызываемым от имени безымянного объекта.

ev1 += new ClassDLG((new C0()).C0f);
ev1 += new ClassDLG(obj1.C1f);
ev1 += new ClassDLG((new C2()).C2f);

// И вот какие события...
// Орёл или Решка!
for (i = 0; i < 100; i++)
{
Console.WriteLine("==={0}===",i);
switch (rnd.Next(0,2))
{
       case 0: // Уведомляются ВСЕ заинтересованные в событии "выпадение 0".
              ev0(i);
              break;
       case 1: // Уведомляются ВСЕ заинтересованные в событии "выпадение 0".
              ev1(i);
              break;
}
}

}//_____________________________________________________________________
}//=====================================================================
}
```

Второй пример…

```
using System;
namespace EVENTS
{
// Объявленние классов-делегатов.
// Класс-делегат со стандартными сигнатурой и спецификатором возвращаемого значения.
delegate int ClassDlg_1(int key, string str);
// Сигнатура делегата и возвращаемое значение могут быть любыми!
delegate int ClassDlg_2(int key);
// Тип второго параметра делегата для этого класса-делегата объявлен ниже.
delegate void ClassDlg_TimeInfo(object sourseKey, TimeInfoEventArgs eventKey);
//======================================================================

class TimeInfoEventArgs: EventArgs
{//====================================================================
public TimeInfoEventArgs(int hh, int mm, int ss)
{
hour = hh;
minute = mm;
second = ss;
}
public readonly int hour;
public readonly int minute;
public readonly int second;
}//====================================================================

//======================================================================
// Объявляются два класса
//======================================================================
class Doll_1
```

```
{
public ClassDlg_1 dlg1Doll_1; // Объявлен делегат “стандартного” вида.
public ClassDlg_2 dlg2Doll_1; // Объявлен делегат.
public ClassDlg_TimeInfo dlgTimeInfo; // Нужна ссылка на объект - параметр делегата.

// Объявлено соответствующее событие.
// Событие объявляется на основе класса делегата.
// Нет класса делегата – нет и события!
public event ClassDlg_2 eventDoll_1;

// Конструктор. В конструкторе производится настройка делегатов.
public Doll_1()
{
dlg1Doll_1 = new ClassDlg_1(F2Doll_1);
dlgTimeInfo = new ClassDlg_TimeInfo(TimeInfoDoll_1);
}

// Функции – члены класса, которые должны вызываться
// в ответ на уведомление о событии, на которое предположительно
// должен подписаться объект – представитель данного класса.
//=========================================================================
public int F0Doll_1(int key)
{
// Эта функция только сообщает...
Console.WriteLine(“this is F0Doll_1({0})”, key);
return 0;
}

public void F1Doll_1(int key)
{
// Эта функция ещё и УВЕДОМЛЯЕТ подписавшийся у неё объект...
// Что это за объект функция не знает!
Console.WriteLine(“this is F1Doll_1({0}), fire eventDoll_1!”, key);
if (eventDoll_1 != null) eventDoll_1(key);
}

public int F2Doll_1(int key, string str)
{
// Эта функция сообщает и возвращает значение...
Console.WriteLine(“this is F2Doll_1({0},{1})”, key, str);
return key;
}

void TimeInfoDoll_1(object sourseKey,TimeInfoEventArgs eventKey)
{
// А эта функция вникает в самую суть события, которое несёт
// Информацию и времени возбуждения события.
Console.Write(“event from {0}:”, ((SC)sourseKey).ToString());
Console.WriteLine(“the time is {0}:{1}:{2}”,eventKey.hour.ToString(),
eventKey.minute.ToString(),
eventKey.second.ToString());
}

}

//=========================================================================
//=========================================================================
// Устройство второго класса проще. Нет обработчика события “времени”.
class Doll_2
{

public ClassDlg_1 dlg1Doll_2;
public ClassDlg_2 dlg2Doll_2;
public event ClassDlg_2 eventDoll_2;

// В конструкторе производится настройка делегатов.
public Doll_2()
{
dlg1Doll_2 = new ClassDlg_1(F1Doll_2);
dlg2Doll_2 = new ClassDlg_2(F0Doll_2);
}
```

```
public int F0Doll_2(int key)
{
// Эта функция только сообщает...
Console.WriteLine("this is F0Doll_2({0})", key);
return 0;
}

public int F1Doll_2(int key, string str)
{
// Эта функция сообщает и возвращает значение...
Console.WriteLine("this is F1Doll_2({0},{1})", key, str);
return key;
}

public void F2Doll_2(int key)
{
// Эта функция ещё и УВЕДОМЛЯЕТ подписавшийся у неё объект...
Console.WriteLine("this is F2Doll_2({0}), fire eventDoll_2!", key);
if (eventDoll_2 != null) eventDoll_2(key);
}
}

//========================================================================
//========================================================================
class SC // Start Class
{
// Объявлен класс-делегат…
public static ClassDlg_2 dlgSC;

// В стартовом класса объявлены два события.
public static event ClassDlg_1 eventSC;
public static event ClassDlg_TimeInfo timeInfoEvent;

static public int FunSC(int key)
{
Console.WriteLine("this is FunSC({0}) from SC",key);
// Первое уведомление. В теле этой функции осуществляется передача
// управления методу ОБЪЕКТА, который заинтересован в данном событии.
// В данном случае событием в буквальном смысле является факт передачи
// управления функции FunSC. Какие одбъекты заинтересованы в этом
// событии – сейчас сказать невозможно. Здесь только уведомляют о
// событии. Подписывают заинтересованных – в другом месте!
eventSC(2*key, "Hello from SC.FunSC !!!");

DateTime dt = System.DateTime.Now;
// Второе уведомление.
timeInfoEvent(new SC(), new TimeInfoEventArgs(dt.Hour,dt.Minute,dt.Second));
return key;
}

static void Main(string[] args)
{
// Объявили и определили два объекта.
// Так вот кто интересуется событиями!
Doll_1 objDoll_1 = new Doll_1();
Doll_2 objDoll_2 = new Doll_2();

// Подписали их на события.
// В событии eventSC заинтересованы объекты
// objDoll_1 и objDoll_2, которые здесь подписываются на
// события при помощи делегатов, которые были настроены
// на соответствующие функции в момент создания объекта.
// Конструкторы обоих классов только и сделали, что настроили
// собственные делегаты.
eventSC += objDoll_1.dlg1Doll_1;
eventSC += objDoll_2.dlg1Doll_2;

// А в событии timeInfoEvent заинтересован лишь
// объект-представитель класса Doll_1.
timeInfoEvent += objDoll_1.dlgTimeInfo;
```

```
// Определили делегата для функции SC.FunSC.
// Это собственная статическая функция класса SC.
dlgSC = new ClassDlg_2(SC.FunSC);

// А теперь достраиваем делегаты, которые
// не были созданы и настроены в конструкторах
// соответствующих классов.
objDoll_1.dlg2Doll_1 = new ClassDlg_2(objDoll_2.F0Doll_2);
objDoll_2.dlg1Doll_2 = new ClassDlg_1(objDoll_1.F2Doll_1);

// Подписали объекты на события.
objDoll_1.eventDoll_1 +=new ClassDlg_2(objDoll_2.F0Doll_2);
objDoll_2.eventDoll_2 +=new ClassDlg_2(objDoll_1.F0Doll_1);

// SC будет сообщать о времени!
// И не важно, откуда последует вызов функции.
// Событие в классе objDoll_1 используется для вызова
// статического метода класса SC.
objDoll_1.eventDoll_1 +=new ClassDlg_2(SC.dlgSC);
// Событие в классе objDoll_2 используется для вызова
// статического метода класса SC.
objDoll_2.eventDoll_2 +=new ClassDlg_2(SC.dlgSC);

// Работа с делегатами.
objDoll_1.dlg2Doll_1(125);
objDoll_2.dlg1Doll_2(521, "Start from objDoll_2.dlg1Doll_2");

// Работа с событиями. Уведомляем заинтересованные классы и объекты!
// Что-то не видно особой разницы с вызовами через делегатов.
objDoll_1.F1Doll_1(125);
objDoll_2.F2Doll_2(521);

// Отключили часть приемников.
// То есть отказались от получения некоторых событий.
// И действительно. Нечего заботиться о чужом классе.
objDoll_1.eventDoll_1 -= new ClassDlg_2(SC.dlgSC);
objDoll_2.eventDoll_2 -= new ClassDlg_2(SC.dlgSC);

// Опять работа с событиями.
objDoll_1.F1Doll_1(15);
objDoll_2.F2Doll_2(51);
}

}
}
```

События и делегаты. Различия

Так в чём же разница между событиями и делегатами в .NET?

В последнем примере предыдущего раздела при объявлении события очевидно его строгое соответствие определённому делегату.

```
public static event System.EventHandler xEvent;
```

System.EventHandler – это ТИП ДЕЛЕГАТА! Оператор, который обеспечивает процедуру "подписания на уведомление", полностью соответствует оператору модификации многоадресного делегата. Аналогичным образом дело обстоит и с процедурой "отказа от уведомления".

```
BaseClass.xEvent += new System.EventHandler(this.MyFun);
BaseClass.xEvent -= new System.EventHandler(xxx.MyFun);
```

И это действительно так. За операторными функциями += и -= скрываются методы классов-делегатов (в том числе и класса-делегата System.EventHandler).

Более того. Если в последнем примере в объявлении события ВЫКИНУТЬ ключевое слово event –

```
public static event System.EventHandler xEvent;
```

и заменить его на:

```
public static System.EventHandler xEvent;
```

System.EventHandler – это класс-делегат! То ничего не произойдёт. Вернее, ВСЁ будет происходить, как и раньше! Вместо пары СОБЫТИЕ-ДЕЛЕГАТ будет работать пара ДЕЛЕГАТ-ДЕЛЕГАТ. Таким образом, функционально событие является всего лишь разновидностью класса-делегата, главной задачей которого является обеспечение строгой привязки делегата к соответствующему событию.

Модификатор event вносит лишь незначительные синтаксические нюансы в использование этого МОДИФИЦИРОВАННОГО делегата. Чтобы хоть как-нибудь различать возбудителя и получателя события.

Атрибуты, сборки, рефлексия

Рефлексия (отражение) типов

Процесс анализа типов (структуры типов) в ходе выполнения приложения (сборки). В .NET реализуется при помощи класса System.Type и пространства имён System.Reflection.

reflection - система, предоставляющая выполняемому коду информацию о нем самом.

Пространство имён System.Reflection содержит классы и интерфейсы, которые позволяют организовать просмотр загруженных в сборку типов, методов, полей (данных-членов), и обеспечивают динамические механизмы реализации типов и вызова методов. Включает множество взаимосвязанных классов, интерфейсов, структур и делегатов, предназначенных для реализации процесса отражения. Неполный перечень классов представлен ниже:

Тип	Назначение
Assembly	Методы для загрузки, описания и выполнения разнообразных операций над сборкой.
AssemblyName	Информация о сборке (идентификатор, версия, язык реализации).
EventInfo	Информация о событиях.
FieldInfo	Информация о полях.
MemberInfo	Абстрактный базовый класс, определяющий общие члены для EventInfo, FieldInfo, MethodInfo, PropertyInfo.
MethodInfo	Информация о методе.
Module	Позволяет обратиться к модулю в многофайловой сборке.
ParameterInfo	Информация о параметре.
PropertyInfo	Информация о свойстве.

Класс System.Type содержит методы, позволяющие получать информацию о типах приложения, пространства имён System.Reflection определяет типы для организации позднего связывания и динамической загрузки типов.

Класс Type является основой для реализации функциональности System.Reflection и средством для получения доступа к метаданным.

Использование членов класса Тип, позволяет получить информацию о:

- типе (GetType(string)),
- конструкторах (GetConstructors()),
- методах (GetMethods()),
- данных-членах (GetFields()),
- свойствах (GetProperties()),
- событиях, объявленных в классе (GetEvents()),
- модуле,
- сборке, в которой реализуется данный класс.

Объект-представитель класса Type уникален. Две ссылки на объекты-представители класса Type оказываются эквивалентными, если только они были построены в результате обращения к одному и тому же типу.

Объект-представитель класса Type может представить любой из следующих типов:

- Классы
- Размерные типы
- Массивы
- Интерфейсы
- Указатели
- Перечисления

Ссылка на объект-представитель класса Type, ассоциированная с некоторым типом, может быть получена одним из следующих способов:

1. В результате вызова метода

```
Type Object.GetType()
```

(метода-члена класса Object), который возвращает ссылку на объект-представитель типа Type, представляющий информацию о типе. Вызов производится

от имени объекта-представителя данного типа. Вызов становится возможен в связи с тем, что любой класс наследует тип Object.

2. В результате вызова статического метода-члена класса Type:

```
public static Type Type.GetType(string)
```

Параметром является строка со значением имени типа. Возвращает объект-представитель класса Type, с информацией о типе, специфицированном параметром метода.

3. От имени объекта-представителя класса Assembly - от имени объекта-сборки (самоописываемого, многократно используемого, версифицируемого БЛОКА (фрагмента) CLR-приложения) вызываются методы

```
Type[] Assembly.GetTypes()
Type Assembly.GetType(string)

// Получаем ссылку на сборку, содержащую объявление типа MyType,
// затем – массив объектов-представителей класса Type.
Type[] tt = (Assembly.GetAssembly(typeof(MyType))).GetTypes();
// Без комментариев.
Type[] tt = (Assembly.GetAssembly(typeof(MyType))).GetType("MyType");
От имени объекта-представителя класса Module (Модуль - portable executable файл с расширением .dll или .exe, состоящий из одного и более классов и интерфейсов)
Type[] Module.GetTypes()
Type Module.GetType(string)
Type[] Module.FindTypes(TypeFilter filter, object filterCriteria)

где TypeFilter – класс-делегат
// The TypeFilter делегаты используются to filter списка классов.
// А как этим хозяйством пользоваться - не знаю. В хелпах не нашёл.
public delegate bool TypeFilter(Type m, object filterCriteria);
```

4. В результате выполнения операции typeof(), которая применяется для построения объекта-представителя класса System.Type. Выражение, построенное на основе операции typeof, имеет следующий вид:

```
typeof(type)
```

Операнд выражения – тип, для которого может быть построен объект-представитель класса System.Type.

Пример применения операции:

```
using System;
using System.Reflection;

public class MyClass
{
public int intI;
public void MyMeth()
 {
 }

public static void Main()
 {
Type t = typeof(MyClass);
// Альтернативная эквивалентная конструкция
// MyClass t1 = new MyClass();
// Type t = t1.GetType();

MethodInfo[] x = t.GetMethods();
foreach (MethodInfo m in x)
 {
Console.WriteLine(m.ToString());
 }

Console.WriteLine();
MemberInfo[] x2 = t.GetMembers();
foreach (MemberInfo m in x2)
 {
```

```
Console.WriteLine(m.ToString());
 }
 }
}
```

Реализация отражения. Type, InvokeMember, BindingFlags

Раннее связывание – деятельность, выполняемая на стадии компиляции, позволяющая:

- обнаружить и идентифицировать объявленные в приложении типы,
- выявить и идентифицировать члены класса,
- подготовить при выполнении приложения вызов методов и свойств, доступ к значениям полей-членов класса.

Позднее, динамическое связывание – деятельность, выполняемая непосредственно при выполнении приложения, позволяющая:

- обнаружить и идентифицировать объявленные в приложении типы,
- выявить и идентифицировать члены класса,
- обеспечить в ходе выполнения приложения вызов методов и свойств, доступ к значениям полей-членов класса.

При этом вызов методов и свойств при выполнении приложения обеспечивается методом InvokeMember. Этот метод выполняет достаточно сложную работу и поэтому нуждается в изощрённой системе управления, для реализации которой применяется перечисление BindingFlags. Перечисление также применяется для управления методом GetMethod.

В рамках этого перечисления определяются значения флажков, которые управляют процессом динамического связывания в ходе реализации отражения.

Список элементов перечисления прилагается.

Имя элемента	Описание
CreateInstance	Определяет, что отражение должно создавать экземпляр заданного типа. Вызывает конструктор, соответствующий указанным аргументам. Предоставленное имя пользователя не обрабатывается. Если тип поиска не указан, будут использованы флаги (Instance | Public). Инициализатор типа вызвать нельзя.
DeclaredOnly	Определяет, что должны рассматриваться только члены, объявленные на уровне переданной иерархии типов. Наследуемые члены не учитываются.
Default	Определяет отсутствие флагов связывания.
ExactBinding	Определяет, что типы представленных аргументов должно точно соответствовать типам соответствующих формальных параметров. Если вызывающий оператор передает непустой объект Binder, отражение создает исключение, так как при этом вызывающий оператор предоставляет реализации BindToXXX, которые выберут соответствующий метод. Отражение моделирует правила доступа для системы общих типов. Например, если вызывающий оператор находится в той же сборке, ему не нужны специальные разрешения относительно внутренних членов. В противном случае вызывающему оператору потребуется ReflectionPermission. Этот метод применяется при поиске защищенных, закрытых и т. п. членов. Главный принцип заключается в том, что ChangeType должен выполнять только расширяющее преобразование, которое никогда не теряет данных. Примером расширяющего преобразования является преобразование 32-разрядного целого числа со знаком в 64-разрядное целое число со знаком. Этим оно отличается от сужающего преобразования, при котором возможна потеря данных. Примером сужающего преобразования является преобразование 64-разрядного целого числа со знаком в 32-разрядное целое число со знаком. Связыватель по умолчанию не обрабатывает этот флаг, но пользовательские связыватели используют семантику этого флага.
FlattenHierarchy	Определяет, что должны быть возвращены статические члены вверх по иерархии. Статические члены — это поля, методы, события и свойства. Вложенные типы не возвращаются.

GetField	Определяет, что должно возвращаться значение указанного поля.
GetProperty	Определяет, что должно возвращаться значение указанного свойства.
IgnoreCase	Определяет, что при связывании не должен учитываться регистр имени члена.
IgnoreReturn	Используется при COM-взаимодействии для определения того, что возвращаемое значение члена может быть проигнорировано.
Instance	Определяет, что в поиск должны быть включены члены экземпляра.
InvokeMethod	Определяет, что метод должен быть вызван. Метод не может быть ни конструктором, ни инициализатором типа.
NonPublic	Определяет, что в поиск должны быть включены члены экземпляра, не являющиеся открытыми (public).
OptionalParamBinding	Возвращает набор членов, у которых количество параметров соответствует количеству переданных аргументов. Флаг связывания используется для методов с параметрами, у которых есть значения методов, и для функций с переменным количеством аргументов (varargs). Этот флаг должен использоваться только с Type.InvokeMember. Параметры со значениями по умолчанию используются только в тех вызовах, где опущены конечные аргументы. Они должны быть последними аргументами.
Public	Определяет, что открытые (public) члены должны быть включены в поиск.
PutDispProperty	Определяет, что для COM-объекта должен быть вызван член PROPPUT. PROPPUT задает устанавливающую свойство функцию, использующую значение. Следует использовать PutDispProperty, если для свойства заданы и PROPPUT, и PROPPUTREF и нужно различать вызываемые методы.
PutRefDispProperty	Определяет, что для COM-объекта должен быть вызван член PROPPUTREF. PROPPUTREF использует устанавливающую свойство функцию, использующую ссылку, вместо значения. Следует использовать PutRefDispProperty, если для свойства заданы и PROPPUT, и PROPPUTREF и нужно различать вызываемые методы.
SetField	Определяет, что должно устанавливаться значение указанного поля.
SetProperty	Определяет, что должно устанавливаться значение указанного свойства. Для COM-свойств задание этого флага связывания эквивалентно заданию PutDispProperty и PutRefDispProperty.
Static	Определяет, что в поиск должны быть включены статические члены.
SuppressChangeType	Не реализован.

Далее демонстрируется применение класса Type, в частности, варианты использования метода-члена класса Type InvokeMember, который обеспечивает выполнения методов и свойств класса.

```
using System;
using System.Reflection;

// В классе объявлены поле myField, конструктор, метод String ToString(), свойство.
class MyType
{
int myField;
public MyType(ref int x)
{
x *= 5;
}

public override String ToString()
{
Console.WriteLine("This is: public override String ToString() method!");
return myField.ToString();
}

// Свойство MyProp нашего класса обладает одной замечательной особенностью:
// значение поля myField объекта-представителя класса MyType не может быть
// меньше нуля. Если это ограничение нарушается - возбуждается исключение.
```

```
public int MyProp
{
get
{
return myField;
}
set
{
if (value < 1)
throw new ArgumentOutOfRangeException("value", value, "value must be > 0");
myField = value;
}
}
}

class MyApp
{
static void Main()
{

// Создали объект-представитель класса MyType
// на основе объявления класса MyType.
Type t = typeof(MyType);

// А это одномерный массив объектов, содержащий ОДИН элемент.
// В этом массиве будут передаваться параметры конструктору.
Object[] args = new Object[] {8};
Console.WriteLine("The value of x before the constructor is called is {0}.", args[0]);

// Вот таким образом в рамках технологии отражения производится
// обращение к конструктору. Наш объект адресуется по ссылке obj.
Object obj = t.InvokeMember(
null,
       //_______________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.CreateInstance, // Вот распоряжение о создании объекта…
       //_______________________________
null,
null,
args // А так организуется передача параметров в конструктор.
);

Console.WriteLine("Type: " + obj.GetType().ToString());
Console.WriteLine("The value of x after the constructor returns is {0}.", args[0]);
// Изменение (запись и чтение) значения поля myField. Только что созданного
// объекта-представителя класса MyType. Как известно, этот объект адресуется по
// ссылке obj. Мы сами его по этой ссылке расположили!
t.InvokeMember(
"myField", // Будем менять значение поля myField…
//_____________________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.SetField, // Вот инструкция по изменению значения поля.
//_____________________________________
null,
obj, // Вот указание на то, ГДЕ располагается объект…
new Object[] {5} // А вот и само значение. Оно упаковывается в массив объектов.
);

int v = (Int32) t.InvokeMember(
"myField",
        //_____________________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
```

```
BindingFlags.GetField, // А сейчас мы извлекаем значение поля myField.
        //_____________________________________
null,
obj, // “Работаем” всё с тем же объектом. Значение поля myField
     // присваивается переменной v.
null
        );

// Вот распечатали это значение.
Console.WriteLine(“myField: ” + v);
// “От имени” объекта будем вызывать нестатический метод.
String s = (String) t.InvokeMember(
“ToString”, // Имя переопределённого виртуального метода.
        //_____________________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.InvokeMethod, // Сомнений нет! Вызываем метод!
        //_____________________________________
null,
obj,   // От имени нашего объекта вызываем метод без параметров.
null
        );

// Теперь обращаемся к свойству.
Console.WriteLine(“ToString: “ + s);
// Изменение значения свойства. Пытаемся присвоить недозволенное значение.
// И посмотрим, что будет… В конце-концов, мы предусмотрели перехватчик исключения.
try
{
t.InvokeMember(
“MyProp”, // Работаем со свойством.
       //_____________________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.SetProperty, // Установить значение свойства.
       //_____________________________________
null,
obj,
new Object[] {0} // Вот пробуем через обращение к свойству
// установить недозволенное значение.
       );
}
catch (TargetInvocationException e)
{
// Фильтруем исключения... Реагируем только на исключения типа
// ArgumentOutOfRangeException. Все остальные «проваливаем дальше».
if (e.InnerException.GetType() != typeof(ArgumentOutOfRangeException)) throw;
// А вот как реагируем на ArgumentOutOfRangeException.
// Вот так скромненько уведомляем о попытке присвоения запрещённого значения.
Console.WriteLine(“Exception! Catch the property set.”);
}

t.InvokeMember(
“MyProp”,
//_____________________________________
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.SetProperty, // Установить значение свойства.
//_____________________________________
null,
obj,
new Object[] {2} // Вновь присваиваемое значение. Теперь ПРАВИЛЬНОЕ.
);

v = (int) t.InvokeMember(
```

```
"MyProp",
BindingFlags.DeclaredOnly |
BindingFlags.Public |
BindingFlags.NonPublic |
BindingFlags.Instance |
BindingFlags.GetProperty, // Прочитать значение свойства.
null,
obj,
null
      );

Console.WriteLine("MyProp: " + v);
}
}
```

Ну вот. Создавали объект, изменяли значение его поля (данного-члена), вызывали его (нестатический) метод, обращались к свойству (подсовывали ему некорректные значения). И при этом НИ РАЗУ НЕ НАЗЫВАЛИ ВЕЩИ СВОИМИ ИМЕНАМИ! В сущности, ЭТО И ЕСТЬ ОТРАЖЕНИЕ.

Атрибуты

Атрибут – средство добавления ДЕКЛАРАТИВНОЙ информации к элементам программного кода. Назначение атрибутов – внесение всевозможных не предусмотренных обычным ходом выполнения приложения изменений:

- описание взаимодействия между модулями,
- дополнительная информация, используемая при работе с данными (управление сериализацией),
- отладка,
- много чего другого.

Эта декларативная информация составляет часть метаданных кода. Она может быть использована при помощи механизмов отражения.

Структура атрибута регламентирована. Атрибут – это класс. Общий предок всех атрибутов – класс System.Attribute.

Информация, закодированная с использованием атрибутов, становится доступной в процессе ОТРАЖЕНИЯ (рефлексии типов).

Атрибуты типизированы.

.NET способна прочитать информацию в атрибутах и использовать её в соответствии с предопределёнными правилами или замыслами разработчика. Различаются

- Предопределённые атрибуты. В .NET реализовано множество атрибутов с предопределёнными значениями:

  DllImport – для загрузки .dll файлов.

  Serializable – означает возможность сериализации свойств объекта-представителя класса.

  NonSerialized – обозначает данные-члены класса как несериализуемые. Карандаши не сереализуются.
- Производные (пользовательские) атрибуты могут определяться и использоваться в соответствии с замыслами разработчика. Возможно создание собственных (пользовательских) атрибутов. Главные условия:

  соблюдение синтаксиса,

  соблюдение принципа наследования.

В основе пользовательских атрибутов – всё та же система типов с наследованием от базового класса System.Attribute.

И не спроста! В конце-концов, информация, содержащаяся в атрибутах, предназначается для Framework.NET и она должна суметь в ней разбираться. Пользователи или другие инструментальные средства должны уметь кодировать и декодировать эту информацию.

Добавлять атрибуты можно к:

- сборкам,
- классам,
- элементам класса,
- структурам,
- элементам структур,

- параметрам,
- возвращаемым значениям.

Следующий пример является демонстрацией объявление и применения производных атрибутов.

```
using System;
using System.Reflection;

namespace CustomAttrCS
{
// Перечисление of animals.
// Start at 1 (0 = uninitialized).
public enum Animal
{
// Pets.
Dog = 1,
Cat,
Bool,
}

// Перечисление of animals.
// Start at 1 (0 = uninitialized).
public enum Color
{
// Colors.
Red = 1,
Brown,
White,
}


// Класс пользовательских атрибутов.
public class AnimalTypeAttribute : Attribute
{//=====================================================================
// Данное-член типа перечисление.
protected Animal thePet;

protected string WhoIs(Animal keyPet)
{
string retStr = "";
switch (keyPet)
{
case Animal.Dog: retStr = "This is the Dog!"; break;
case Animal.Cat: retStr = "This is the Cat!"; break;
case Animal.Bool: retStr = "This is the Bool!"; break;
default: retStr = "Unknown animal!"; break;
}

return retStr;
}

// Конструктор вызывается при установке атрибута.
public AnimalTypeAttribute(Animal pet)
{
thePet = pet;
Console.WriteLine(«{0}», WhoIs(pet));
}

// Свойство, демонстрирующее значение атрибута.
public Animal Pet
{
get
{
return thePet;
}
set
{
thePet = Pet;
}
```

```
}
}//=========================================================================

// Ещё один класс пользовательских атрибутов.
public class ColorTypeAttribute : Attribute
{//=========================================================================
// Данное-член типа перечисление.
protected Color theColor;

// Конструктор вызывается при установке атрибута.
public ColorTypeAttribute(Color keyColor)
{
theColor = keyColor;
}

// Свойство, демонстрирующее значение атрибута.
public Color ColorIs
{
get
{
return theColor;
}
set
{
theColor = ColorIs;
}
}
}//=========================================================================


// A test class where each method has its own pet.
class AnimalTypeTestClass
{//=========================================================================
// Содержит объявления трёх методов, каждый из которых
// предваряется соответствующим ПОЛЬЗОВАТЕЛЬСКИМ атрибутом.
// У метода может быть не более одного атрибута данного типа.
[AnimalType(Animal.Dog)]
[ColorType(Color.Brown)]
public void DogMethod()
{
Console.WriteLine("This is DogMethod()...");
}

[AnimalType(Animal.Cat)]
public void CatMethod()
{
Console.WriteLine("This is CatMethod()...");
}


[AnimalType(Animal.Bool)]
[ColorType(Color.Red)]
public void BoolMethod(int n, string voice)
{
int i;
Console.WriteLine("This is BoolMethod!");

if (n > 0) for (i = 0; i < n; i++)
 {
Console.WriteLine(voice);
 }
}
}//=========================================================================

class DemoClass
{//=========================================================================
static void Main(string[] args)
{
int invokeFlag;
int i;
```

```
// И вот ради чего вся эта накрутка производилась...
// Объект класса AnimalTypeTestClass под именем testClass
// представляет собой КОЛЛЕКЦИЮ методов, каждый из которых
// снабжён соответствующим ранее определённым пользовательским
// СТАНДАРТНЫМ атрибутом. У класса атрибута AnimalTypeAttribute есть всё,
// что положено иметь классу, включая конструктор.
AnimalTypeTestClass testClass = new AnimalTypeTestClass();
// Так вот создали соответствующий объект-представитель класса...

// Объект-представитель класса сам может служить источником информации
// о собственном классе. Информация о классе представляется методом GetType()
// в виде объекта - представителя класса Type. Информационная капсула!
Type type = testClass.GetType();
// Из этой капсулы можно извлечь множество всякой «полезной» информации...
// Например, можно получить коллекцию (массив) элементов типа MethodInfo
// (описателей методов), содержащую список описателей методов, объявленных
// в данном классе. В список будет включена информация о ВСЕХ методах класса.
// О тех, которые были определены явным образом и те, которые были унаследованы
// от базовых классов. И по этому списку описателей методов мы пройдём
// победным маршем («На-На-На») оператором foreach.
i = 0;

foreach(MethodInfo mInfo
in
type.GetMethods())
{

invokeFlag = 0;
Console.WriteLine("#####{0}#####{1}#####", i, mInfo.Name);
// И у каждого из методов мы спросим относительно множества атрибутов,
// которыми метод был снабжён при объявлении класса.
foreach (Attribute attr
in
Attribute.GetCustomAttributes(mInfo))
{
Console.WriteLine("~~~~~~~~~~");
// Check for the AnimalType attribute.
if (attr.GetType() == typeof(AnimalTypeAttribute))
{
Console.WriteLine("Method {0} has a pet {1} attribute.",
mInfo.Name,
((AnimalTypeAttribute)attr).Pet);
// Посмотрели значение атрибута - и если это Animal.Bool - подняли флажок.
if (((AnimalTypeAttribute)attr).Pet.CompareTo(Animal.Bool) == 0) invokeFlag++;
}

if (attr.GetType() == typeof(ColorTypeAttribute))
{
Console.WriteLine("Method {0} has a color {1} attribute.",
mInfo.Name,
((ColorTypeAttribute)attr).ColorIs);

// Посмотрели значение атрибута - и если это Animal.Bool - подняли флажок.
if (((ColorTypeAttribute)attr).ColorIs.CompareTo(Color.Red) == 0) invokeFlag++;
}

// И если случилось счастливое совпадение значений атрибутов метода,
// метод выполняется.
// Метод Invoke в варианте с двумя параметрами:
// объект-представитель исследуемого класса
// (в данном случае AnimalTypeTestClass), и массив объектов - параметров.
if (invokeFlag == 2)
{
object[] param = {5,"Mmmuuu-uu-uu!!! Mmm..."};
mInfo.Invoke(new AnimalTypeTestClass(),param);
}

Console.WriteLine("~~~~~~~~~~");
}

Console.WriteLine("#####{0}#####", i);
```

```
i++;
}
}
}//========================================================================
}
```

Сборка. Класс Assembly

Класс Assembly определяет Сборку – основной строительный блок common language runtime приложения. Как строительный блок clr, сборка обладает следующими основными свойствами:

- возможностью многократного применения,
- versionable (версифицированностью),
- самоописываемостью.

Эти понятия являются ключевыми для решения проблемы отслеживания версии и для упрощения развертывания приложений во время выполнения.

Сборки обеспечивают инфраструктуру, которая позволяет во время выполнения полностью "понимать" структуру и содержимое приложения, и контролировать версии и зависимости элеметнов выполняемого приложения.

Сборки бывают:

- частными (private). Представляют наборы типов, которые могут быть использованы только теми приложениями, в состав которых они входят. Располагаются в файлах с расширениями .dll (.exe) и .pdb (program debug Database). Для того чтобы использовать в приложении частную сборку, её надо ВКЛЮЧИТЬ в приложение, то есть, разместить в каталоге приложения (application directory) или в одном из его подкаталогов.
- общего доступа (shared). Также набор типов и ресурсов внутри модулей (модуль – двоичный файл сборки). Предназначены для использования НЕОГРАНИЧЕННЫМ количеством приложений на клиентском компе. Эти сборки устанавливаются не в каталог приложения, а в специальный каталог, называемый Глобальным Кэшем Сборок (Global Assembly Cache - GAC). Этот каталог на платформе Windows XP имеет путь C:\WINDOWS\assembly. Таким образом, в .NET ВСЕ совместно используемые сборки собираются в одном месте. Имя ("общее имя" или "строгое имя") сборки общего доступа строится с использованием информации о версии сборки.

Загружаемая сборка строится как БИБЛИТОТЕКА КЛАССОВ (файл с расширением .dll), либо как выполняемый модуль (файл с расширением .exe).

Если это файл с расширением .dll, то в среде Visual Studio её использование поддерживается специальными средствами среды. Это "полуавтоматическая" загрузка частной сборки в Reference приложения (Add Reference…). Сборки, располагаемые в .exe файлах, особой поддержкой для включения сборки не состав приложения не пользуется.

Для анализа сборки применяется утилита IlDasm.exe, которую можно подключить к непосредственно вызываемому из среды разработки VisualStudio списку утилит.

Ниже представлены члены класса Сборки.

Открытые свойства

CodeBase	Возвращает местонахождение сборки, указанное первоначально, например, в объекте AssemblyName.
EntryPoint	Возвращает точку входа для этой сборки.
EscapedCodeBase	Возвращает URI, предоставляющий базовый код, включая escape-знаки.
Evidence	Возвращает свидетельство для этой сборки.
FullName	Возвращает отображаемое имя сборки.
GlobalAssemblyCache	Возвращает значение, показывающее, была ли сборка загружена из глобального кэша сборок.
ImageRuntimeVersion	Возвращает версию общеязыковой среды выполнения (CLR), сохраненной в файле, содержащем манифест.
Location	Возвращает местонахождение в формате базового кода загруженного файла, содержащего манифест, если для него не было теневого копирования.

Открытые методы

CreateInstance	Перегружен. Находит тип в этой сборке и создает его экземпляр, используя абстрактный метод.
CreateQualifiedName	Статический. Создает тип, задаваемый отображаемым именем его сборки.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetAssembly	Статический. Возвращает сборку, в которой определяется заданный класс.
GetCallingAssembly	Статический. Возвращает Assembly метода, который вызывает текущий метод выполнения.
GetCustomAttributes	Перегружен. Возвращает пользовательские атрибуты для этой сборки.
GetEntryAssembly	Статический. Возвращает процесс, исполняемый в домене приложения по умолчанию. В других доменах приложений это первый исполняемый процесс, который был выполнен AppDomain.ExecuteAssembly.
GetExecutingAssembly	Статический. Возвращает Assembly, из которой исполняется текущий код.
GetExportedTypes	Возвращает экспортируемые типы, определенные в этой сборке.
GetFile	Возвращает объект FileStream для указанного файла из таблицы файлов манифеста данной сборки.
GetFiles	Перегружен. Возвращает файлы в таблице файлов манифеста сборки.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetLoadedModules	Перегружен. Возвращает все загруженные модули, являющиеся частью этой сборки.
GetManifestResourceInfo	Возвращает информацию о способе сохранения данного ресурса.
GetManifestResourceNames	Возвращает имена всех ресурсов в этой сборке.
GetManifestResourceStream	Перегружен. Загружает указанный ресурс манифеста из сборки.
GetModule	Возвращает указанный модуль этой сборки.
GetModules	Перегружен. Возвращает все модули, являющиеся частью этой сборки.
GetName	Перегружен. Возвращает AssemblyName для этой сборки.
GetObjectData	Возвращает сведения сериализации со всеми данными, необходимыми для повторного создания этой сборки.
GetReferencedAssemblies	Возвращает объекты AssemblyName для всех сборок, на которые ссылается данная сборка.
GetSatelliteAssembly	Перегружен. Возвращает сопутствующую сборку.
GetType	Перегружен. Возвращает объект Type, предоставляющий указанный тип.
GetTypes	Возвращает типы, определенные в этой сборке.
IsDefined	Показывает, определен ли пользовательский атрибут, заданный указанным значением Type.
Load	Статический. Перегружен. Загружает сборку.
LoadFile	Статический. Перегружен. Загружает содержимое файла сборки.
LoadFrom	Статический. Перегружен. Загружает сборку.
LoadModule	Перегружен. Загружает внутренний модуль этой сборки.
LoadWithPartialName	Статический. Перегружен. Загружает сборку из папки приложения или из глобального кэша сборок, используя частичное имя.
ToString	Переопределен. Возвращает полное имя сборки, также называемое отображаемым именем.

Открытые события

ModuleResolve	Возникает, когда загрузчик классов общеязыковой среды выполнения не может обработать ссылку на внутренний модуль сборки, используя обычные средства.

Защищенные методы

Finalize (унаследовано от Object)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

Класс сборки в действии

Исследование свойств и областей применения класса Assembly начинаем с создания тестовой однофайловой сборки AssemblyForStart в рамках проекта Class Library.

Первая сборка Operators00.exe:

```
using System;
namespace Operators00
{
public class xPoint
{
float x, y;
xPoint()
{
x = 0.0F;
y = 0.0F;
}

public xPoint(float xKey, float yKey):this()
{
x = xKey;
y = yKey;
}

public static bool operator true(xPoint xp)
{
if (xp.x != 0.0F && xp.y != 0.0F) return true;
else return false;
}

public static bool operator false(xPoint xp)
{
if (xp.x == 0.0F || xp.y == 0.0F) return false;
else return true;
}


public static xPoint operator | (xPoint key1, xPoint key2)
{
if (key1) return key1;
if (key2) return key2;
return new xPoint();
}

public static xPoint operator & (xPoint key1, xPoint key2)
{
if (key1 && key2) return new xPoint(1.0F, 1.0F);
return new xPoint();
}


public void Hello()
{
Console.WriteLine("Hello! Point {0},{1} is here!",this.x,this.y);
}

}

class Class1
{
```

```
/// <summary>
/// The main entry Point for the application.
/// </summary>
[STAThread]
static void Main() // У точки входа пустой список параметров.
// Я пока не сумел ей передать через метод Invoke массива строк.
{
xPoint xp0 = new xPoint(1.0F, 1.0F);
xPoint xp1 = new xPoint(1.0F, 1.0F);

if (xp0 || xp1) Console.WriteLine("xp0 || xp1 is true!");
else Console.WriteLine("xp0 || xp1 is false!");

}
}
}
```

Вторая сборка AssemblyForStart.dll. В примере она так и не запускалась. Используется только для тестирования стандартных средств загрузки сборок-библиотек классов.

```
using System;
namespace AssemblyForStart
{
/// <summary>
/// Class1 : первая компонента сборки AssemblyForStart.
/// </summary>
public class Class1
{
public Class1()
{
Console.WriteLine("This is constructor Class1()");
}

public void fC1()
{
Console.WriteLine("This is fC1()");
}
}

}
```

А вот полигон AssemblyStarter.exe. В примере демонстрируется техника ПОЗДНЕГО СВЯЗЫВАНИЯ. Именно поэтому код, который выполняется после загрузки сборки не содержит в явном виде информации об используемых в приложении типах. Транслятор действует в строгом соответствии с синтаксисом языка C# и просто не поймёт пожелания "создать объект-представитель класса … который будет объявлен в сборке, которую предполагается загрузить в ходе выполнения приложения".

```
using System;
using System.Reflection;
using System.IO;

namespace AssemblyStarter
{
/// <summary>
/// Приложение обеспечивает запуск сборки.
/// </summary>
class Class1
{

static void Main(string[] args)
{
// Сборка может быть вызвана непосредственно по имени
// (строковый литерал с дружественным именем сборки).
// Ничего особенного. Просто имя без всяких там расширений.
// Главная проблема заключается в том, сборки должны предварительно
```

```
// включаться в раздел References (Ссылки).
// Кроме того, информация о загружаемой сборке может быть представлена
// в виде объекта - представителя класса AssemblyName, ссылка на который
// также может быть передана в качестве аргумента методу Assembly.Load().
// Вот здесь как раз и происходит формирование этого самого объекта.
AssemblyName asmName = new AssemblyName();
asmName.Name = "AssemblyForStart";
// Версию подсмотрели в манифесте сборки с помощью IlDasm.exe.
Version v = new Version("1.0.1790.25124");
// Можно было бы для пущей крутизны кода поле Version проинициализировать
// непосредственно (дело хозяйское):
// asmName.Version = new Version(«1:0:1790:25124»);
asmName.Version = v;

// Ссылка на объект-представитель класса Assembly.
//
Assembly asm = null;
try
{
// Загрузка сборки по «дружественному имени».
//asm = Assembly.Load(«AssemblyForStart»);
// Путь и полное имя при загрузки частной сборки не имеют значения.
// Соответствующие файлы должны располагаться непосредственно в каталоге приложения.
//asm = Assembly.Load
//(@"D:\Users\WORK\Cs\AssemblyTest\AssemblyForStart\bin\Debug\AssemblyForStart.dll");
//asm = Assembly.Load(asmName);
// Если сборку организовать в виде исполняемого модуля и «запихнуть» в каталог
// вручную - загрузится и такая сборка.
asm = Assembly.Load("Operators00");
}
catch(FileNotFoundException e)
{
Console.WriteLine("We have a problem:" + e.Message);
}

// Итак, решено. Загрузили сборку, содержащую объявление класса xPoint.
// Если сборка загрузилась - с ней надо что-то делать. Ясное дело,
// надо выполнять программный код сборки.
// Первый вариант выполнения. В сборке содержатся объявления двух классов:
// класса xPoint и класса Class1. Мы воспользуемся объявлением класса xPoint,
// построим соответствующий объект-представитель этого класса, после чего
// будем вызывать нестатические методы-члены этого класса.
// Всё происходит в режиме ПОЗДНЕГО связывания. Поэтому ни о каких ЯВНЫХ упоминаниях
// имён типов не может быть речи. Единственное явное упоминание - это упоминание
// имени метода.
object[] ps = {25,25};
Type[] types = asm.GetTypes();
// Здесь используется класс Activator!
object obj = Activator.CreateInstance(types[0],ps);

MethodInfo mi = types[0].GetMethod("Hello");
mi.Invoke(obj,null);

// Второй вариант выполнения. Воспользуемся тем обстоятельством, что загружаемая
// сборка не является библиотекой классов, а является обычной выполнимой сборкой
// с явным образом обозначенной точкой входа - СТАТИЧЕСКОЙ функцией Main(), которая
// является членом класса Class1.
mi = asm.EntryPoint;
// Вот всё получилось! Единственное, что я не сделал, так это не смог передать
// в точку входа загруженной сборки массива строк-параметров
// (смотреть на точку входа данной сборки).
// Ну не удалось. Потому и в сборке Operators точку сборки объявил без параметров.
mi.Invoke(null,null);

}
}
}
```

Собрали, запустили. Получилось. Стало быть, ВСЁ ХОРОШО.
В ходе выполнения приложения класс Assembly позволяет:

- получать информацию о самой сборке,
- обращаться к членам класса, входящим в сборку, загружать другие сборки.

Разбор полётов

В приведённом выше примере демонстрируется техника ПОЗДНЕГО СВЯЗЫВАНИЯ. Именно поэтому код, который выполняется после загрузки сборки не содержит в явном виде информации об используемых в приложении типах. Транслятор действует в строгом соответствии с синтаксисом языка C# и просто не поймёт пожелания "создать объект-представитель класса … который будет объявлен в сборке, которую предполагается загрузить в ходе выполнения приложения".

Класс System.Activator

Класс Activator - главное средство, обеспечивающее позднее связывание.

Содержит методы, позволяющие создавать объекты на основе информации о типах, получаемой непосредственно в ходе выполнения приложения, а также получать ссылки на существующие объекты. Далее приводится список перегруженных статических методов класса:

- CreateComInstanceFrom. Создаёт экземпляр COM объекта.
- CreateInstance. Создаёт объект-представитель specified типа, используя при этом наиболее подходящий по списку параметров конструктор (best matches the specified parameters).

Пример

```
ObjectHandle hdlSample;
IMyExtenderInterface myExtenderInterface;
string argOne = "Value of argOne";
int argTwo = 7;
object[] args = {argOne, argTwo};
// Uses the UrlAttribute to create a remote object.
object[] activationAttributes =
{new UrlAttribute("http://localhost:9000/MySampleService")};
// Activates an object for this client.
// You must supply a valid fully qualified assembly name here.
hdlSample = Activator.CreateInstance(
"Assembly text name, Version, Culture, PublicKeyToken",
"samplenamespace.sampleclass",
true,
BindingFlags.Instance|BindingFlags.Public,
null,
args,
null,
activationAttributes,
null);
myExtenderInterface = (IMyExtenderInterface)hdlSample.Unwrap();
Console.WriteLine(myExtenderInterface.SampleMethod("Bill"));
```

- CreateInstanceFrom. Создаёт объект-представитель типа, специфицированного по имени. Имя специфицируется на основе имени сборки. При этом используется подходящий по списку параметров конструктор (the constructor that best matches the specified parameters).

Пример

```
ObjectHandle hdlSample;
IMyExtenderInterface myExtenderInterface;
object[] activationAttributes = {new SynchronizationAttribute()};
// Assumes that SampleAssembly.dll exists in the same directory as this assembly.
hdlSample = Activator.CreateInstanceFrom(
"SampleAssembly.dll",
"SampleNamespace.SampleClass",
activationAttributes);
// Assumes that the SampleClass implements an interface provided by
// this application.
myExtenderInterface = (IMyExtenderInterface)hdlSample.Unwrap();
```

```
Console.WriteLine(myExtenderInterface.SampleMethod("Bill"));
```

- GetObject - Overloaded. Создает прокси (заместителя, представителя) для непосредственного запуска активизированного сервером объекта или XML сервиса.

Сборка может быть вызвана непосредственно по имени (строковый литерал с дружественным именем сборки).

Ничего особенного. Просто имя без всяких там расширений. Главная проблема заключается в том, частные сборки должны предварительно включаться в раздел References (Ссылки).

Кроме того, информация о загружаемой сборке может быть представлена в виде объекта - представителя класса AssemblyName, ссылка на который также может быть передана в качестве аргумента методу Assembly.Load().

Версия сборки

Эта характеристика имеется у каждой сборки, несмотря на то, что частной сборке она и ни к чему.

Версию можно "подсмотреть" подсмотрели в манифесте сборки с помощью IlDasm.exe. Можно было бы для пущей крутизны кода поле Version проинициализировать непосредственно (дело хозяйское):

И тут открываются несколько приколов:

- В манифесте версия задаётся последовательностью цифр, разделённых между собой двоеточием. Однако при формировании поля Version эти цифры следует разделять точкой.
- Если сборка уникальна и среда разработки приложения ничего не знает о других версиях данной сборки - в поле Version можно спокойно забивать любые четвёрки чисел. Лишь бы требования формата были соблюдены.
- Две одноименных частные сборки с разными версиями в раздел References третьей сборки загрузить не получается. Одноимённые файлы с одинаковым расширением в один каталог не помещаются. Не взирая на версии.

Следующий шаг: загружаем сборку в память

Используется блок try, поскольку загружаемую сборку можно и не найти.

При загрузке сборки известно её расположение (application directory), однако с расширением имени могут возникнуть проблемы. Действует такой алгоритм "поиска" (и называть-то это поиском как-то не удобно):

Среда выполнения .NET пытается обнаружить файл с расширением .dll.

В случае неудачи среда выполнения .NET пытается обнаружить файл с расширением .exe.

В случае неудачи – предпринимаются ДРУГИЕ алгоритмы поиска.

Файл конфигурации приложения

Так что за ДРУГИЕ-такие алгоритмы?

А всё зависит от файла конфигурации приложения. В этом файле можно явным образом сформулировать особенности приложения. В частности, явным образом указать место расположения загружаемой сборки.

Файл конфигурации – это текстовый файл со странным именем <ИмяФайла.Расширение>.config, в котором размещаются в строго определённом порядке теги, прописанные на языке XML.

Среда выполнения .NET умеет читать XML.

Утверждается, что если расположить частные сборки в других подкаталогах приложения, нежели bin\Debug, то с помощью файла конфигурации эти сборки исполняющая среда БЕЗ ТРУДА обнаружит. Однако не находит. Может быть, чего криво делаю?

Файл AssemblyStarter.exe.config:

```
<configuration>
<runtime>
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1" >
<probing privatePath="XXX\YYY"/>
</assemblyBinding>
</runtime>
```

```
</configuration>
```

Общедоступная сборка

Строгое имя общедоступной сборки стоит из:

- дружественного текстового имени и “культурной” информации,
- идентификатора версии,
- пары Открытый/Закрытый ключ,
- цифровой подписи

Делаем общую сборку

1. Сначала – ЧАСТНАЯ сборка.

```
using System;
namespace SharedAssembly00
{
/// <summary>
/// Summary description for Class1.
/// </summary>
public class Class1
{
public Class1()
{

}

public void f0()
{
Console.WriteLine(“This is SharedAssembly00.f0()”);
}
}
}
```

2. Делаем пару Открытый/Закрытый ключ

Для чего в Visual Studio .NET 2003 Command Prompt командной строкой вызываем утилиту-генератор ключей

```
D:\...>sn –k theKey.cnk
```

3. В файле сборки AssemblyInfo.cs (таковой имеется в каждом проекте, ранее не использовался) дописываем в качестве значения ранее пустого атрибута AssemblyKeyFile полный путь к созданному утилитой sn файлу (в одну строчку):

```
[assembly:
AssemblyKeyFile(@”D:\Users\WORK\Cs\AssemblyTest\SharedAssembly00\theKey.snk”)]
```

4. Компилируем сборку и наблюдаем манифест сборки, в котором появляется открытый ключ. Открытый ключ размещается в манифесте сборки. Закрытый ключ хранится в модуле сборки, содержащим манифест, однако в манифест не включается. Этот ключ используется для создания цифровой подписи, которая помещается в сборку. Во время выполнения сборки среда выполнения проверяет соответствие маркера открытого ключа сборки, запрашиваемой клиентом (приложением, запускающим эту сборку) с маркером открытого ключа самой сборки общего пользования из GAC. Такая проверка гарантирует, что клиент получает именно ту сборку, которую он заказывал.

Клиентское приложение	Сборка общего пользования, установленная в GAC
В манифесте клиента имеется ссылка на внешнюю сборку общего пользования. Маркер открытого ключа этой сборки отмечен тегом : :::::::::: .assembly extern SharedAssembly00	Манифест сборки общего пользования в GAC содержит такое же значение ключа: 908ED85E3E377208 Его можно увидеть при исследовании содержимого GAC (свойства элемента)

{
.publickeytoken =
(90 8E D8 5E 3E 37 72 08) // ...^>7r.
.ver 1:0:1790:37888
}
::::::::::
	А закрытый ключ сборки общего пользования совместно с открытым ключом используется для создания цифровой подписи сборки и хранится вместе с подписью в самой сборке.

5. Размещаем общедоступную сборку в GAC. Для чего либо используем утилиту gacutil.exe с ключом \i и именем сборки с полным путём в качестве второго параметра. Либо просто перетаскиваем мышкой файл сборки в каталог, содержащий GAC. В случае успеха наблюдаем состояние GAC.

Сборка там!

Игры со сборками из GAC

Создаём новую сборку, в которой предполагается использовать наше детище. Затем добавляем ссылку на сборку. Не всё так просто. Сначала надо отыскать соответствующую .dll'ку. Наличие ключа в манифесте сборки приводит к тому, что сборка (в отличие от частных сборок) не будет копироваться в каталог запускающей сборки. Вместо этого исполняющая среда будет обращаться в GAC. Дальше – проще.

Объявленные в сборке классы оказываются доступны запускающей сборке. Набрали аж 3 сборки общего пользования. Дело не хитрое…

```
using System;
using SharedAssembly00;
using SharedAssembly01;
using SharedAssembly02;

namespace AssemblyStarter01
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class startClass
{
/// <summary>
/// The main entry Point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
try
{
SharedAssembly00.Class1 c001 = new SharedAssembly00.Class1(); c001.f0();
SharedAssembly01.Class1 c011 = new SharedAssembly01.Class1(); c011.f0();
SharedAssembly02.Class1 c021 = new SharedAssembly02.Class1(); c021.f0();
}
catch(TypeLoadException e)
{
Console.WriteLine("We are the problem: " + e.Message);
}
}
}
}
```

Итак, подключили общедоступную сборку. Она не копируется, а остаётся в GAC. При создании клиента были выполнены определённые телодвижения, в результате которых клиент сохраняет информацию о свойствах располагаемых в GAC компонентах. Свойства этих компонент можно посмотреть после подсоединения данного элемента из GAC к References клиента. В частности, там есть свойство Copy local (по умолчанию установленное в false). Это означает, что соответствующая компонента из GAC клиентом не копируется. Общую сборку можно превратить в частную сборку, если это свойство установить в true.

Динамические сборки

Всё, о чём писалось до этого момента – суть СТАТИЧЕСКИЕ СБОРКИ. Статические сборки существуют в виде файлов на диске или других носителях. В случае необходимости загружаются в оперативную память и выполняются благодаря функциональным возможностям класса Assembly.

Динамические сборки создаются непосредственно в ходе выполнения приложения (статической сборки) и существуют в оперативной памяти. По завершении выполнения приложения обречены на бесследное исчезновение, если, конечно, не были предприняты особые усилия по их сохранению на диск.

Для работы с динамическими сборками используется пространство имён System.Reflection.Emit.

Emit – излучать, испускать, выпускать (деньги).

Это множество типов позволяет создавать и выполнять динамические сборки, а также ДОБАВЛЯТЬ НОВЫЕ типы и члены в загруженные в оперативную память сборки.

Пространство имен System.Reflection.Emit содержит классы, позволяющие компилятору или инструментальным средствам создавать метаданные и инструкции промежуточного языка MSIL и при необходимости формировать на диске PE-файл. Эти классы предназначены в первую очередь для обработчиков сценариев и компиляторов.

Список классов, структур и перечислений, входящих в пространство, прилагается.

Классы

Класс	Описание
AssemblyBuilder	Определяет и представляет динамическую сборку.
ConstructorBuilder	Определяет и представляет конструктор динамического класса.
CustomAttributeBuilder	Помогает в построении пользовательских атрибутов.
EnumBuilder	Описывает и предоставляет тип перечисления.
EventBuilder	Определяет события для класса.
FieldBuilder	Определяет и предоставляет поле. Этот класс не наследуется.
ILGenerator	Создает инструкции промежуточного языка MSIL.
LocalBuilder	Представляет локальную переменную внутри метода или конструктора.
MethodBuilder	Определяет и предоставляет метод (или конструктор) для динамического класса.
MethodRental	Позволяет быстро менять реализацию основного текста сообщения метода, задающего метод класса.
ModuleBuilder	Определяет и представляет модуль. Получает экземпляр класса ModuleBuilder с помощью вызова метода DefineDynamicModule.
OpCodes	Содержит поля, предоставляющие инструкции промежуточного языка MSIL для эмиссии членами класса ILGenerator (например, методом Emit).
ParameterBuilder	Создает или связывает информацию о параметрах.
PropertyBuilder	Определяет свойства для типа.
SignatureHelper	Обеспечивает методы построения подписей.
TypeBuilder	Определяет и создает новые экземпляры классов во время выполнения.
UnmanagedMarshal	Представляет класс, описывающий способ маршалирования поля из управляемого в неуправляемый код. Этот класс не наследуется.

Структуры

Структура	Описание
EventToken	Предоставляет Token, возвращаемый метаданными для представления события.
FieldToken	Структура FieldToken является объектным представлением лексемы, представляющей поле.
Label	Представляет метку в потоке инструкций. Label используется вместе с классом ILGenerator.
MethodToken	Структура MethodToken является объектным представлением лексемы, представляющей метод.
OpCode	Описывает инструкцию промежуточного языка MSIL.
ParameterToken	Структура ParameterToken является закрытым представлением возвращаемого метаданными лексемы, используемого для представления параметра.
PropertyToken	Структура PropertyToken является закрытым представлением возвращаемого метаданными маркера Token, используемого для представления свойства.

SignatureToken	Предоставляет Token, возвращенный метаданными для представления подписи.
StringToken	Предоставляет лексему, которая предоставляет строку.
TypeToken	Представляет маркер Token, который возвращается метаданными, чтобы представить тип.

Перечисления

Перечисление	Описание
AssemblyBuilderAccess	Определяет режимы доступа для динамической сборки.
FlowControl	Описывает, каким образом инструкция меняет поток команд управления.
OpCodeType	Описывает типы инструкций промежуточного языка MSIL.
OperandType	Описывает тип операнда инструкции промежуточного языка MSIL.
PackingSize	Задает один из двух факторов, определяющих выравнивание занимаемой полями памяти при маршаллинге типа.
PEFileKinds	Задает тип переносимого исполняемого PE-файла.
StackBehaviour	Описывает, как значения помещаются в стек или выводятся из стека.

Динамическая сборка: создание, сохранение, загрузка, выполнение

```
///-------------------------------------------------------------------------------
/// Вот такой класс в составе однофайловой сборки DynamicAssm
/// предполагается построить в ходе выполнения сборки DynamicAssemblyGenerator.
///
/// public class DynamicTest
/// {
/// private string messageString;
/// // Конструктор
/// DynamicTest(string strKey)
/// {
/// messageString = strKey;
/// }
///
/// // Методы
/// public void ShowMessageString()
/// {
/// System.Console.WriteLine("the value of messageString is {0}...", messageString);
/// }
///
/// public string GetMessageString()
/// {
/// return messageString;
/// }
///
///}
///-------------------------------------------------------------------------------

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Threading;

namespace DynamicAssemblyGenerator
{
/// <summary>
/// AssemblyGenerator - класс, реализующий динамическую генерацию сборки.
/// </summary>
class AssemblyGenerator
{

public string XXX;
public string ZZZ()
{
return XXX;
}

public int CreateAssm(AppDomain currentAppDomain)
{
// Создание сборки начинается с присвоения ей имени и номера версии.
```

```
// Для этого используется класс AssemblyName.
// Определяется имя и версия создаваемой сборки.
AssemblyName assmName = new AssemblyName();
assmName.Name = "DynamicAssm";
assmName.Version = new Version("1.0.0.0");

// Создаётся сборка в памяти. В рамках текущего домена приложения.
// С использованием режима доступа, который задаётся одним из элементов перечисления:
// Run - динамическая сборка выполняется, но не сохраняется.
// RunAndSave - динамическая сборка выполняется и сохраняется.
// Save - динамическая сборка не выполненяется, но сохраняется.
AssemblyBuilder assembly = currentAppDomain.DefineDynamicAssembly(assmName,
AssemblyBuilderAccess.Save);
// Создаётся однофайловая сборка, в которой имя единственного модуля совпадает с
именем самой сборки.
ModuleBuilder module = assembly.DefineDynamicModule("DynamicAssm",
«DynamicAssm.dll» );
// Создаётся и определяется класс DynamicTest класс. Метод module.DefineType позволяет
// встраивать в модуль класс, структуру или интерфейс.
// Вторым параметром метода идёт элемент перечисления. Таким образом создаётся объект-
заготовка
// класса, который далее дополняется полями, свойствами, методами...
TypeBuilder dynamicTestClass = module.DefineType("DynamicAssm.DynamicTest",
TypeAttributes.Public);
// Объявляется данное-член класса DynamicTest.
// Предполагается объявить "private string messageString;"
FieldBuilder messageStringField = dynamicTestClass.DefineField("messageString",
Type.GetType("System.String"),
FieldAttributes.Private);
// Объекты для генерации элементов класса. В данном конкретном случае используются при
генерации:
ILGenerator bodyConstructorIL; // - тела конструктора.
ILGenerator methodIL; // - тела метода.

// Объявляется конструктор._____________________________________________________
// Предполагается объявить «DynamicTest(string strKey)...»
Type[] constructorArgs = new Type[1];
constructorArgs[0] = Type.GetType("System.String");
ConstructorBuilder constructor = dynamicTestClass.DefineConstructor(
MethodAttributes.Public,
CallingConventions.Standard,
constructorArgs);
// Тело конструктора. Представляет собой IL код,
// который встраивается в тело конструктора посредством метода Emit,
// определённого в классе ILGenerator (см. Объекты для генерации элементов класса).
// Метод Emit в качестве параметров использует перечисление OpCodes (коды операций),
// которые определяют допустимые команды IL.
bodyConstructorIL = constructor.GetILGenerator();
bodyConstructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = Type.GetType("System.Object");
ConstructorInfo greatConstructor = objectClass.GetConstructor(new Type[0]);
bodyConstructorIL.Emit(OpCodes.Call, greatConstructor);
bodyConstructorIL.Emit(OpCodes.Ldarg_0);
bodyConstructorIL.Emit(OpCodes.Ldarg_1);
bodyConstructorIL.Emit(OpCodes.Stfld,messageStringField);
bodyConstructorIL.Emit(OpCodes.Ret);
// Конец объявления конструктора._______________________________________________

// Объявление метода public string GetMessageString()___________________________
MethodBuilder GetMessageStringMethod = dynamicTestClass.DefineMethod(
"GetMessageString",
MethodAttributes.Public,
Type.GetType("System.Sring"),
null);

// IL_0000: ldarg.0
// IL_0001: ldfld string DynamicAssemblyGenerator.AssemblyGenerator::XXX
// IL_0006: stloc.0
// IL_0007: br.s IL_0009
// IL_0009: ldloc.0
// IL_000a: ret
```

```
//System.Reflection.Emit.Label label = new Label();

// Тело метода...
methodIL = GetMessageStringMethod.GetILGenerator();
methodIL.Emit(OpCodes.Ldarg_0);
methodIL.Emit(OpCodes.Ldfld,messageStringField);
methodIL.Emit(OpCodes.Ret);
// Конец объявления метода public string GetMessageString()_____________________________

// Объявление метода public string ShowMessageString()_____________________________
MethodBuilder ShowMessageStringMethod = dynamicTestClass.DefineMethod(
“ShowMessageString”,
MethodAttributes.Public,
null,
null);
// Тело метода...
methodIL = ShowMessageStringMethod.GetILGenerator();
methodIL.EmitWriteLine(“This is ShowMessageStringMethod...”);
methodIL.Emit(OpCodes.Ret);
// Конец объявления метода public string ShowMessageString()_____________________________
// Вот и завершили динамическое объявление класса.
dynamicTestClass.CreateType();

// Остаётся его сохранить на диск.
assembly.Save(“DynamicAssm.dll”);

return 0;
}


static void Main(string[] args)
{
// Создаётся и сохраняется динамическая сборка.
AssemblyGenerator ag = new AssemblyGenerator();
ag.CreateAssm(AppDomain.CurrentDomain);

// Для наглядности! создаются НОВЫЕ объекты и заново добывается
// ссылка на текущий домен приложения.
// Теперь - дело техники. Надо загрузить и выполнить сборку.
// Делали. Умеем!
AppDomain currentAppDomain = Thread.GetDomain();
AssemblyGenerator assmGenerator = new AssemblyGenerator();
assmGenerator.CreateAssm(currentAppDomain);

// Загружаем сборку.
Assembly assm = Assembly.Load(“DynamicAssm”);
// Объект класса Type для класса DynamicTest.
Type t = assm.GetType(“DynamicAssm.DynamicTest”);

// Создаётся объект класса DynamicTest и вызывается конструктор с параметрами.
object[] argsX = new object[1];
argsX[0] = “Yes, yes, yes-s-s-s!”;
object obj = Activator.CreateInstance(t, argsX);

MethodInfo mi;
// «От имени» объекта-представителя класса DynamicTest
// вызывается метод ShowMessageString.
mi = t.GetMethod(“ShowMessageString”);
mi.Invoke(obj,null);

// «От имени» объекта-представителя класса DynamicTest
// вызывается метод GetMessageString.
// Этот метод возвращает строку, которая перехватывается
// и выводится в окне консольного приложения.
mi = t.GetMethod(“GetMessageString”);
//!!!//mi.Invoke(obj,null);//Этот метод не вызывается. Криво объявился? //
}
}
}
```

Ввод-вывод

Базовые операции

В общем случае понятие ПОТОК это абстрактное понятие, которое обозначает динамическую изменяющуюся во времени последовательность чего-либо.

Применительно к обсуждаемым проблемам ввода-вывода в программах на C#, поток - это последовательность байтов, связанная с конкретными устройствами компьютера (диски, дисплей, принтер, клавиатура) посредством системы ввода-вывода.

Система ввода-вывода обеспечивает для программиста стандартные и не зависящие от физического устройства средства представления информации и управления потоками ввода-вывода. Действия по организации ввода-вывода обеспечивается стандартными наборами, как правило, одноименных функций ввода-вывода со стандартным интерфейсом (спецификацией возвращаемого значения и списком параметров).

Функции, обеспечивающие взаимодействие с различными устройствами ввода-вывода, объявляются в различных классах. Вопрос “ОТКУДА ВЫЗВАТЬ функцию” часто становится более важным, чем вопрос “КАК ПОСТРОИТЬ выражение вызова функции”.

Потоки: байтовые, символьные, двоичные

Большинство устройств, предназначенных для выполнения операций ввода-вывода, являются байт-ориентированными. Этим и объясняется тот факт, что на самом низком уровне все операции ввода-вывода манипулируют с байтами в рамках байтовых потоков.

С другой стороны, значительный объём работ, для которых, собственно и используется вычислительная техника, предполагает работу с символами, а не с байтами (заполнение экранной формы, вывод информации в наглядном и легко читаемом виде, текстовые редакторы).

Символьно-ориентированные потоки, предназначенные для манипулирования с символами, а не с байтами, являются потоками ввода-вывода более высокого уровня. В рамках Framework.NET определены соответствующие классы, которые при реализации операций ввода-вывода обеспечивают автоматическое преобразование данных типа byte в данные типа char и обратно.

В дополнение к байтовым и символьным потокам в C# определены два класса, реализующих механизмы считывания и записи информации непосредственно в двоичном представлении (потоки BynaryReader и BynaryWriter).

Общая характеристика классов потоков

Основные особенности и правила работы с устройствами ввода-вывода в современных языках высокого уровня описываются в рамках классов потоков. Для языков платформы .NET местом описания самых общих свойств потоков является класс System.IO.Stream.

Назначение этого класса заключается в объявлении общего стандартного набора операций (стандартного интерфейса), обеспечивающих работу с устройствами ввода-вывода, независимо от их конкретной реализации источников и получателей информации.

Процедуры чтения и записи информации определяется следующим (список неполон!) набором свойств и АБСТРАКТНЫХ методов, объявляемых в этом классе.

Здесь и ниже понятие АБСТРАКТНЫЙ означает отсутствие КОНКРЕТНОЙ реализации данного метода (об этом позже!). Классы, содержащие абстрактные методы, также называются абстрактными. Основная задача абстрактного класса – задать план для построения, общее направление реализации конкретных (неабстрактных) классов.

В рамках Framework.NET, независимо от характеристик того или иного устройства ввода-вывода, программист ВСЕГДА может узнать:

- можно ли читать из потока – bool CanRead (если можно – значение должно быть установлено в true),

- можно ли писать в поток – bool CanWrite (если можно – значение должно быть установлено в true),
- можно ли задать в потоке текущую позицию – bool CanSeek (если последовательность, в которой производится чтение-запись не является жёстко детерминированной и возможно позиционирование в потоке - значение должно быть установлено в true),
- позицию текущего элемента потока – long Position (возможность позиционирования в потоке предполагает возможность программного изменения значения этого свойства),
- общее количество символов потока (длину потока) – long Length.

В соответствии с общими принципами реализации операций ввода-вывода, для потока предусмотрен набор методов, позволяющих реализовать:
- чтение байта из потока с возвращением целочисленного представления СЛЕДУЮЩЕГО ДОСТУПНОГО байта в потоке ввода – int ReadByte(),
- чтение определённого (count) количества байтов из потока и размещение их в массиве buff, начиная с элемента buff[index], с возвращением количества успешно прочитанных байтов – int Read(byte[] buff, int index, int count),
- запись в поток одного байта – void WriteByte(byte b),
- запись в поток определённого (count) количества байтов из массива buff, начиная с элемента buff[index], с возвращением количества успешно записанных байтов – int Write(byte[] buff, int index, int count),
- позиционирование в потоке – long Seek (long index, SeekOrigin origin) (позиция текущего байта в потоке задаётся значением смещения index относительно позиции origin),
- для буферизованных потоков принципиальна операция флэширования (записи содержимого буфера потока на физическое устройство) – void Flush(),
- закрытие потока – void Close().

Множество классов потоков ввода-вывода в Framework.NET основывается (наследует свойства и интерфейсы) на абстрактном классе Stream. При этом классы конкретных потоков обеспечивают собственную реализацию интерфейсов этого абстрактного класса.

Наследниками класса Stream являются, в частности, три класса байтовых потоков:
- BufferedStream – обеспечивает буферизацию байтового потока. Как правило, буферизованные потоки являются более производительными по сравнению с небуферизованными,
- FileStream – байтовый поток, обеспечивающий файловые операции ввода-вывода,
- MemoryStream – байтовый поток, использующий в качестве источника и хранилища информации оперативную память.

Манипуляции с потоками предполагают НАПРАВЛЕННОСТЬ производимых действий. Информацию из потока можно ПРОЧИТАТЬ, а можно её в поток ЗАПИСАТЬ. Как чтение, так и запись, предполагают реализацию определённых механизмов байтового обмена с устройствами ввода-вывода.

Свойства и методы, объявляемые в соответствующих классах, определяют специфику потоков, используемых для чтения и записи:
- TextReader,
- TextWriter.

Эти классы являются абстрактными. Это означает, что они не “привязаны” ни к какому конкретному потоку. Они лишь определяют интерфейс (набор методов), который позволяет организовать чтение и запись информации для любого потока.

В частности, в этих классах определены следующие методы, определяющие базовые механизмы символьного ввода-вывода. Для класса TextReader:
- int Peek() - Reads the next character without changing the state of the reader or the character source. Returns the next available character without actually reading it from the input stream.
- int Read(…) - Overloaded. Несколько одноименных функций с одним и тем же именем. Читает значения из входного потока. Вариант int Read() предполагает чтение из потока одного символа с возвращением его целочисленного эквивалента или -1 при достижении конца потока. Вариант int Read (char[] buff, int index, int count) и его полный аналог int ReadBlock(char[] buff,

int index, int count) обеспечивает прочтение a maximum of count characters из текущего потока и записывает данные в buffer, начиная at index.
- string ReadLine() – Читает строку символов из текущего потока. Возвращается ссылка на объект типа string.
- string ReadToEnd() – Читает все символы, начиная с текущей позициисимвольного потока, определяемого объектом класса TextReader и возвращает результат как ссылка на объект типа string.
- void Close() – Закрывает поток ввода.

Для класса TextWriter, в свою очередь, определяется:
- множество перегруженных вариантов функции void Write(…) со значениями параметров, позволяющих записывать символьное представление значений базовых типов (смотреть список базовых типов) и массивов значений (в том числе и массивов строк).
- void Flush() - Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream. Очистка буфера вывода с предварительным выводом в поток вывода (носитель данных) содержимого буфера.
- void Close() – Закрывает поток вывода.

Эти классы являются базовыми для классов:
- StreamReader – содержит свойства и методы, обеспечивающие считывание СИМВОЛОВ из байтового потока,
- StreamWriter – содержит свойства и методы, обеспечивающие запись СИМВОЛОВ в байтовый поток.

Вышеуказанные классы включают методы своих “предков” и позволяют реализовать процессы чтения-записи непосредственно из конкретных байтовых потоков. Работа по организации ввода-вывода с использованием этих классов предполагает определение соответствующих объектов, “ответственных” за реализацию операций ввода-вывода с явным указанием потоков, которые предполагается при этом использовать.

Ещё одна пара классов потоков обеспечивает механизмы символьного ввода-вывода для строк:
- StringReader,
- StringWriter.

В этом случае источником и хранилищем множества символов является символьная строка.

Интересно заметить, что у всех ранее перечисленных классов имеются методы, обеспечивающие закрытие потоков и не определены методы обеспечивающие открытие соответствующего потока. Потоки открываются в момент создания объекта-представителя соответствующего класса. Наличие функции, обеспечивающей явное закрытие потока принципиально. Оно связано с особенностями выполнения управляемых модулей в Framework.NET. Время начала работы сборщика мусора заранее неизвестно.

Предопределённые потоки ввода-вывода

Ещё одна категория потоков. Предопределённые потоки ввода-вывода используются для реализации ввода-вывода в рамках консольных приложений.

В классах предопределённых потоков в отличие от ранее рассмотренных потоков явным образом задаются источник (откуда) и место размещения (куда) информации.

Основные свойства и методы этих потоков определены в классе System.Console. Класс Console является достаточно сложной конструкцией.

В рамках этого класса определены данные-члены, обеспечивающие реализацию предопределённых потоков. В классе System.Console таких потоков три (In, Out, Error).

Ниже описываются свойства этих потоков:
- Standard input stream - член класса Console In (в общепринятой нотации Console.In). Является объектом-представителем класса TextReader. По умолчанию поток In ориентирован на получение информации с клавиатуры,

- Standard output stream - поток вывода Console.Out. Является объектом-представителем класса TextWriter. По умолчанию обеспечивает вывод информации на дисплей.
- Standard error stream – поток вывода сообщений об ошибках Console.Error. Также является объектом-представителем класса TextWriter. И также по умолчанию выводит информацию на дисплей.

В классе System.Console также определены следующие функции-члены (вернее, множества ПЕРЕГРУЖЕННЫХ функций), обеспечивающих процедуры ввода-вывода:
- Write и WriteLine,
- Read и ReadLine.

При этом функции-члены класса Console ПО УМОЛЧАНИЮ обеспечивают связь с конкретным потоком. Например, при вызове метода

```
Console.WriteLine(…); // Список параметров опущен
```

информация направляется в поток Console.Out. Класс Console также обеспечивает вызов соответствующей функции-члена от имени объекта-представителя класса TextWriter. Непосредственное обращение к члену класса Console (то есть, к потоку), отвечающего за текстовый вывод, имеет следующую форму вызова:

```
Console.Out.WriteLine(…);
```

Вызов определённого в классе Console метода ReadLine

```
Console.ReadLine(…);
```

обеспечивает получение информации через поток Console.In. Непосредственный вызов аналогичной функции от имени члена класса Console, представляющего класс TextReader, выглядит следующим образом:

```
Console.In.ReadLine(…);
```

Функции-представителя класса Console, отвечающей за непосредственное направление информации в поток вывода сообщений об ошибках, не предусмотрено.

Вывод информации об ошибках может быть реализован путём непосредственного обращения к потоку:

```
Console.Error.WriteLine(…);
```

Если вспомнить, что объект Error является представителем того же самого класса TextWriter и, так же как и объект Out связан с окном консольного приложения, разделение сообщений, связанных с нормальным ходом выполнения приложения и сообщений об ошибках становится более чем странным.

Информация, выводимая приложением в “штатном” режиме и сообщения об ошибках одинаково направляются в окно приложения на экран консоли. Однако в классе Console реализована возможность перенаправления соответствующих потоков. Например, стандартный поток вывода можно перенаправить в файл, а поток сообщений об ошибках оставить без изменений направленным в окно приложения. Результат очевиден.

Процедуры перенаправления потоков осуществляются с помощью методов void SetIn(TextReader in), void SetOut(TextWriter out), void SetErr(TextWriter err) и будут рассмотрены ниже.

Функция ToString()

C# является языком, строго и в полном объёме реализующим принципы ООП. В этом языке всё построено на классах и нет ничего, что бы ни соответствовало принципам объектно-ориентированного программирования. Любой элементарный тип является наследником общего класса Object, реализующего, в частности, метод String ToString(), формирующий в виде human-readable текстовой строки описание тех или иных характеристик типа и значений объектов-представителей

данного типа. Любой тип-наследник типа Object (а это ВСЕ типы!) имеет унаследованную, либо собственную переопределённую версию метода ToString(). Применительно к объектам предопределённого типа из CTS, соответствующие версии методов ToString() обеспечивают преобразование значения данного типа к строковому виду.

Всё сделано для программиста. Реализация метода преобразования значения в строку, в конечном счёте, оказывается скрытой от программиста. Именно поэтому вывод значений предопределённых типов с использованием функций Write и WriteLine в программах на языке C# осуществляется так легко и просто. В справочниках по поводу этих функций так и сказано:

The text representation of value is produced by calling Type.ToString.

Эта функция имеет перегруженный вариант, использующий параметр типа string для указания желаемого формата представления. Множество значений этого параметра ограничено предопределённым списком символов форматирования (представлены ниже), возможно, в сочетании с целочисленными значениями.

Символ форматирования	Описание
C	Отображение значения как валюты с использованием принятого по соглашению символа
D	Отображение значения как decimal integer
E	Отображение значения в соответствии с научной нотацией
F	Отображение значения как fixed Point
G	Display the number as a fixed-Point or integer, depending on which is the most compact
N	Применение запятой для разделения порядков
X	Отображение значения в шестнадцатеричной нотации

Непосредственно за символом форматирования может быть расположена целочисленная ограничительная константа, которая в зависимости от типа выводимого значения может определять количество выводимых знаков после точки, либо общее количество выводимых символов. При этом дробная часть действительных значений округляется, либо дополняется нулями справа. При выводе целочисленных значений ограничительная константа игнорируется, если количество преобразуемых символов превышает её значение. В противном случае преобразуемое значение слева дополняется нулями.

Форматирование используется для преобразования значений “обычных” .NET Framework типов данных в строковое представление да ещё в соответствии с каким-либо общепринятым форматом. Например, целочисленное значение 100 можно представить в общепринятом формате currency для отображения валюты. Для этого можно использовать метод ToString() с использованием символа (стоки?) форматирования (“C”). В результате может быть получена строка вида “$100.00”. И это при условии, что установки операционной системы, компьютера, на котором производится выполнение данного программного кода, соответствуют U.S. English specified as the current culture.

```
int MyInt = 100;
String MyString = MyInt.ToString(“C”);
Console.WriteLine(MyString);
```

Консольный ввод-вывод. Функции-члены класса Console

При обсуждении процедур ввода-вывода следует иметь в виду одно важное обстоятельство. Независимо от типа выводимого значения, в конечном результате выводится, СИМВОЛЬНОЕ ПРЕДСТАВЛЕНИЕ значения. Это становится очевидным при выводе информации в окно представления, что и обеспечивают по умолчанию методы Console.Write и Console.WriteLine.

WriteLine отличается тем, что завершает свою работу обязательным выводом Escape-последовательности line feed/carriage return.

```
Console.WriteLine(“The Captain is on the board!”); // Вывод строки.
Console.WriteLine(314); // Вывод символьного представления целочисленного значения.
Console.WriteLine(3.14); // Вывод символьного представления значения типа float.
```

Выводимая символьная строка может содержать Escape-последовательности, которые при выводе информации в окно представления позволяют создавать различные “специальные эффекты”.

Методы с одним параметром достаточно просты. Практически всё происходит само собой. Уже на стадии компиляции при выяснении типа выводимого значения подбирается соответствующий вариант перегруженной функции вывода. становится При выводе значения определяется его тип, производится соответствующее стандартное преобразование к символьному виду, которое в виде последовательности символов и выводится в окно представления.

Для облегчения процесса программирования ввода-вывода в C# также используются варианты функций Write и WriteLine с несколькими параметрами.

Эти методы называются ПЕРЕГРУЖАЕМЫМИ (смотреть перегрузку методов). Для программиста C# это означает возможность вызова этих функций с различными параметрами. Можно предположить, что различным вариантам списков параметров функции могут соответствовать различные варианты функций вывода.

Ниже представлен вариант метода WriteLine с тремя параметрами. Во всех случаях, когда количество параметров превышает 1, первый параметр обязан быть символьной строкой.

```
Console.WriteLine(“qwerty”,314,3.14);
```

Чтобы понять, как выполняется такой оператор, следует иметь в виду, что всякий параметр метода является выражением определённого типа, которое в процессе выполнения метода может вычисляться для определения значения соответствующего выражения. Таким образом, при выполнении метода WriteLine должны быть определены значения его параметров, после чего в окне приложения должна появиться определённая последовательность символов. Ниже представлен результат выполнения выражения вызова функции:

```
qwerty
```

Значения второго и третьего параметров не выводятся.

Дело в том, что первый строковый параметр выражений вызова функций Write и WriteLine используется как управляющий шаблон для представления выводимой информации. Значения следующих за строковым параметром выражений будут выводиться в окно представления лишь в том случае, если первый параметр-строка будет явно указывать места расположения выводимых значений, соответствующих этим параметрам. Явное указание обеспечивается маркерами выводимых значений, которые в самом простом случае представляют собой заключённые в фигурные скобки целочисленные литералы (например, {3}).

При этом способ указания места состоит в следующем:

- CLR индексирует все параметры метода WriteLine, следующие за первым параметром-строкой. При этом второй по порядку параметр получает индекс 0, следующий за ним – 1, и т.д. до конца списка параметров.
- В произвольных местах параметра-шаблона размещаются маркеры выводимых значений.
- Значение маркера должно соответствовать индексу параметра, значение которого должно выводиться на экран.
- Значение целочисленного литерала маркера не должно превышать максимального значения индекса параметра.

Таким образом, оператор

```
Console.WriteLine(“The sum of {0} and {1} is {2}”,314,3.14,314+3.14);
```

обеспечивает вывод следующей строки:

```
The sum of 314 and 3.14 is 317.3
```

В последнем операнде выражения вызова WriteLine при вычислении значения выражения используется неявное приведение типа. При вычислении значения суммы операнд типа int без потери значения приводится к типу float. Безопасные преобразования типов проводятся автоматически.

Несмотря на явную абсурдность выводимых утверждений, операторы

```
Console.WriteLine("The sum of {0} and {1} is {0}",314,3.14,314+3.14);
Console.WriteLine("The sum of {2} and {1} is {0}",314,3.14,314+3.14);
```

также отработают вполне корректно.

Консольный вывод. Форматирование

Помимо индекса параметра маркер выводимого значения может содержать дополнительную информацию относительно формата представления выводимой информации. Выводимые значения преобразуются к символьному представлению, которое, в свою очередь, при выводе в окно приложения может быть дополнительно преобразовано в соответствии с предопределённым "сценарием преобразования". Вся необходимая для дополнительного форматирования информация размещается непосредственно в маркерах и отделяется запятой от индекса маркера.

Таким образом, в операторах вывода можно определить область позиционирования выводимого значения. Например, результатом выполнения следующего оператора вывода:

```
Console.WriteLine(«***{0,10}***,3.14);
```

будет следующая строка:

```
***3.14 ***
```

А выполнение такого оператора:

```
Console.WriteLine(«***{0,-10}***,3.14);
```

приведёт к следующему результату:

```
*** 3.14***
```

Кроме того, в маркерах вывода могут также размещаться дополнительные строки форматирования (FormatString). При этом маркер приобретает достаточно сложную структуру, внешний вид которой в общем случае можно представить следующим образом (M – значение индекса, N - область позиционирования):

```
{M,N:FormatString}
```

, либо

```
{M:FormatString}
```

, если не указывается значение области позиционирования.

Сама же строка форматирования аналогична ранее рассмотренной строке-параметру метода ToString и является комбинацией предопределённых символов форматирования и дополнительных целочисленных значений.

Символ форматирования	Описание
C	Отображение значения как валюты с использованием принятого по соглашению символа
D	Отображение значения как decimal integer
E	Отображение значения в соответствии с научной нотацией
F	Отображение значения как fixed Point
G	Display the number as a fixed-Point or integer, depending on which is the most compact
N	Применение запятой для разделения порядков
X	Отображение значения в шестнадцатеричной нотации

Непосредственно за символом форматирования может быть расположена целочисленная ограничительная константа, которая в зависимости от типа выводимого значения может определять количество выводимых знаков после точки, либо общее количество выводимых символов. При этом дробная часть действительных значений округляется, либо дополняется нулями справа. При выводе целочисленных значений ограничительная константа игнорируется, если

количество выводимых символов превышает её значение. В противном случае выводимое значение слева дополняется нулями.

Следующие примеры иллюстрируют варианты применения маркеров со строками форматирования:

```
Console.WriteLine("Integer fotmating - {0:D3},{1:D5}",12345, 12);
Console.WriteLine("Currency formatting – {0:C},{1:C5}", 99.9, 999.9);
Console.WriteLine("Exponential formatting – {0:E}", 1234.5);
Console.WriteLine("Fixed Point formatting – {0:F3}", 1234.56789);
Console.WriteLine("General formatting – {0:G}", 1234.56789);
Console.WriteLine("Number formatting – {0:N}", 1234567.89);
Console.WriteLine("Hexadecimal formatting – {0:X7}", 12345); // Integer types only!
```

В результате выполнения этих операторов в окно консольного приложения будут выведены следующие строки

```
Integer fotmating – 12345,00012
Currency formatting – $99.90,$999.90000
Exponential formatting – 1.234500E+003
Fixed Point formatting – 1234.568
General formatting – 1234.56789
Number formatting – 1,234,567.89
Hexadecimal formatting – 0003039
```

Функции вывода. Нестандартное (custom) форматирование значений.

В маркерах выражений вызова функций вывода могут также размещаться спецификаторы (custom format strings), реализующие возможности расширенного форматирования.

В приведенной ниже таблице представлены символы, используемые для создания настраиваемых строк числовых форматов.

Следует иметь в виду, что на выходные строки, создаваемые с помощью некоторых из этих знаков, влияют настройки компонента "Язык и региональные стандарты" панели управления объекта NumberFormatInfo, связанного с текущим потоком. Результаты будут различными на компьютерах с разными параметрами культуры.

Знак формата	Имя	Описание
0	Знак-заместитель нуля	Цифра, расположенная в соответствующей позиции форматируемого значения будет скопирована в выходную строку, если в этой позиции в строке формата присутствует "0". Позиции крайних знаков «0» определяют знаки, всегда включаемые в выходную строку. Строка «00» приводит к округлению значения до ближайшего знака, предшествующего разделителю, если используется исключение из округления нуля. Например, в результате форматирования числа 34,5 с помощью строки «00» будет получена строка "35".
#	Заместитель цифры	Цифра, расположенная в соответствующей позиции форматируемого значения будет скопирована в выходную строку, если в этой позиции в строке формата присутствует знак "#". В противном случае в эту позицию ничего не записывается. Обратите внимание, что ноль не будет отображен, если он не является значащей цифрой, даже если это единственный знак строки. Ноль отображается, только если он является значащей цифрой форматируемого значения. Строка формата "##" приводит к округлению значения до ближайшего знака, предшествующего разделителю, если используется исключение из округления нуля. Например, в результате форматирования числа 34,5 с помощью строки "##" будет получена строка «35».
.	Разделитель	Первый знак "." определяет расположение разделителя целой и дробной частей, дополнительные знаки "." игнорируются. Отображаемый разделитель целой и дробной частей определяется свойством NumberDecimalSeparator объекта NumberFormatInfo.
,	Разделитель тысяч	Знак "," применяется в двух случаях. Во-первых, если

		знак “,” расположен в строке формата между знаками-заместителями (0 или #) и слева от разделителя целой и дробной частей, то в выходной строке между группами из трех цифр в целой части числа будет вставлен разделитель тысяч. Отображаемый разделитель целой и дробной частей определяется свойством NumberGroupSeparator текущего объекта NumberFormatInfo. Во-вторых, если строка формата содержит один или несколько знаков “,” сразу после разделителя целой и дробной частей, число будет разделено на 1000 столько раз, сколько раз знак “,” встречается в строке формата. Например, после форматирования строки “0,,” значение 100000000 будет преобразовано в “100”. Применение этого знака для масштабирования не включает в строку разделитель тысяч. Таким образом, чтобы разделить число на миллион и вставить разделители тысяч следует использовать строку формата “#,##0,,”.
%	Заместитель процентов	При использовании этого знака число перед форматированием будет умножено на 100. В соответствующую позицию выходной строки будет вставлен знак “%”. Знак процента определяется текущим классом NumberFormatInfo.
E0 E+0 E-0 e0 e+0 e-0	Научная нотация	Если в строке формата присутствует один из знаков “E”, “E+”, “E-”, “e”, “e+” или “e-”, за которым следует по крайней мере один знак “0”, число представляется в научной нотации; между числом и экспонентой вставляется знак “E” или “e”. Минимальная длина экспоненты определяется количеством нулей, расположенных за знаком формата. Знаки “E+” и “e+” устанавливают обязательное отображение знака “плюс” или “минус” перед экспонентой. Знаки “E”, “e”, “E-” и “e-” устанавливают отображение знака только для отрицательных чисел.
\	Escape-знак	В языке C# и управляемых расширениях C++ знак, следующий в строке формата за обратной косой чертой, воспринимается как escape-последовательность. Этот знак используется с обычными последовательностями форматирования (например, \n — новая строка). Чтобы использовать обратную косую черту как знак, в некоторых языках ее необходимо удвоить. В противном случае она будет интерпретирована компилятором как escape-последовательность. Чтобы отобразить обратную косую черту, используйте строку “\\”. Обратите внимание, что escape-знак не поддерживается в Visual Basic, однако объект ControlChars обладает некоторой функциональностью.
'ABC' "ABC"	Строка букв	Знаки, заключенные в одинарные или двойные кавычки, копируются в выходную строку без форматирования.
;	Разделитель секций	Знак “;” служит для разделения секций положительных, отрицательных чисел и нулей в строке формата.
Другие	Все остальные знаки	Все остальные знаки копируются в выходную строку в соответствующие позиции.

Строки формата с фиксированной запятой (не содержащие подстрок “E0”, “E+0”, “E-0”, “e0”, “e+0” или “e-0”) ОКРУГЛЯЮТ значение с точностью, заданной количеством знаков-заместителей справа от разделителя целой и дробной частей.

Если в строке формата нет разделителя (точки), число округляется до ближайшего целого значения.

Если в целой части числа больше цифр, чем знаков-заместителей цифр, лишние знаки копируются в выходную строку перед первым знаком-заместителем цифры.

К строке может быть применено различное форматирование в зависимости от того, является ли значение положительным, отрицательным или нулевым. Для этого следует создать строку формата, состоящую из трех секций, разделенных точкой с запятой.

- Одна секция. Строка формата используется для всех значений.
- Две секции. Первая секция форматирует положительные и нулевые значения, вторая — отрицательные. Если число форматируется как отрицательное и становится нулем в результате округления в соответствии с форматированием, ноль форматируется в соответствии со строкой первой секции.

- Три секции. Первая секция форматирует положительные значения, вторая — отрицательные, третья — нули. Вторая секция может быть пустой (две точки с запятой рядом), в этом случае первая секция будет использоваться для форматирования нулевых значений. Если число форматируется как ненулевое и становится нулем в результате округления в соответствии с форматированием, ноль форматируется в соответствии со строкой третьей секции.

При генерации выходной строки в этом типе форматирования не учитывается предыдущее форматирование. Например, при использовании разделителей секций отрицательные значения всегда отображаются без знака "минус". Чтобы конечное форматированное значение содержало знак "минус", его следует явным образом включить в настраиваемый указатель формата. В следующем примере демонстрируется применение разделителей секций для форматирования.

Следующий код демонстрирует использование разделителей секций при форматировании строки.

```
// Объявляются переменные с различными значения (положительное, отрицательное, нулевое)
double MyPos = 19.95, MyNeg = -19.95, MyZero = 0.0;
string MyString = MyPos.ToString("$#,##0.00;($#,##0.00);Zero");
// In the U.S. English culture, MyString has the value: $19.95.
MyString = MyNeg.ToString("$#,##0.00;($#,##0.00);Zero");
// In the U.S. English culture, MyString has the value: ($19.95).
MyString = MyZero.ToString("$#,##0.00;($#,##0.00);Zero");
// In the U.S. English culture, MyString has the value: Zero.
```

Следующая таблица иллюстрирует варианты представления выводимых значений в зависимости от используемых строк форматирования. Предполагается, что форматирование проводится в рамках ToString метода, а значения в столбце Формат соответствуют используемой строке форматирования.

В столбце Data type указывается тип форматируемого значения. В столбце Значение – значение, подлежащее форматированию. В столбце Вывод отображается результат форматирования строки для U.S. English параметров культуры.

Формат	Data type	Значение	Вывод
#####	double	123	123
00000	double	123	00123
(###) ### - ####	double	1234567890	(123) 456 - 7800
#.##	double	1.2	1.2
0.00	double	1.2	1.20
00.00	double	1.2	01.20
#,#	double	1234567890	1,234,567,890
#,,	double	1234567890	12345
#,,,	double	1234567890	1
#,##0,,	double	1234567890	1,235
#0.##%	double	0.095	9.5%
0.###E+0	double	95000	9.5E+4
0.###E+000	double	95000	9.5E+004
0.###E-000	double	95000	9.5E004
[##-##-##]	double	123456	[12-34-56]
##;(##)	double	1234	1234
##;(##)	double	-1234	(1234)

И ещё один пример, демонструрующий custom number formatting.

```
Double myDouble = 1234567890;
String myString = myDouble.ToString( "(# ##) ### - ####" );
// The value of myString is "(123) 456 – 7890".
int MyInt = 42;
MyString = MyInt.ToString( "My Number \n= #" );
// In the U.S. English culture, MyString has the value:
// «My Number
// = 42».
```

Консольный ввод. Преобразование значений

Следует иметь в виду, что чтение информации из входного потока класса Console связано с получением из буфера клавиатуры СИМВОЛЬНЫХ последовательностей и, в большинстве случаев, предполагает дальнейшие преобразования этих последовательностей к соответствующему типу значений.

Консольный ввод предполагает использование уже известных статических(!) функций-членов класса Console

- Read() - Читает следующий знак из стандартного входного потока.

```
public static int Read();
```

Возвращаемое значение:
Следующий знак из входного потока или значение «-1», если знаков больше нет.

Метод не будет завершен до окончания операции чтения, например, при нажатии клавиши “Ввод”. При наличии данных входной поток содержит ввод пользователя и зависящую от окружения последовательность знаков перехода на новую строку.

- ReadLine() - Считывает следующую строку символов из стандартного входного потока.

```
public static string ReadLine();
```

Возвращаемое значение:
Следующая строка из входного потока или пустая ссылка, если знаков больше нет.

Строка определяется как последовательность символов, завершаемая парой escape-символов carriage return line feed (“\r\n”) - (hexadecimal 0x000d), (hexadecimal 0x000a). При этом возвращаемая строка эти символы не включает.

В любом случае речь идёт о получении символьной информации. Символьная информация достаточно просто извлекается из входного потока. Однако это не самая большая составляющая общего объёма обрабатываемой информации. Как правило, содержимое входного потока приходится приводить к одному из базовых типов.

В .NET FCL реализован процесс преобразования информации в рамках Общей Системы Типов (CTS).

.NET Framework Class Library включает класс System.Conwert, в котором реализовано множество функций-членов, предназначенных для выполнения ЗАДАЧИ ПРЕОБРАЗОВАНИЯ ЗНАЧЕНИЙ ОДНОГО базового ТИПА В ЗНАЧЕНИЯ ДРУГОГО базового ТИПА. В частности, в этом классе содержится множество функций (по нескольку функций на каждый базовый тип), обеспечивающих по попытку преобразования символьных строк в значения базовых типов.

Кроме того, множество классов, входящих в CTS и FCL располагают вариантами функции Parse(), основное назначение которой - ПОПЫТКА преобразования строк символов (в частности, получаемых в результате выполнения методов консольного ввода) в значения соответствующих типов.

При обсуждении функций преобразования не случайно употребляется слово “попытка”. Не каждая строка символов может быть преобразована к определённому базовому типу. Для успешного преобразования предполагается, что символьная строка содержит символьное представление значения в некотором общепринятом формате. С аналогичной ситуацией мы уже встречались при обсуждении понятия литералов. В случае успешного преобразования функции возвращают результат преобразования в виде значения соответствующего типа.

Если же значение одного типа не может быть преобразовано к значению другого типа, преобразующая функция ВЫРАБАТЫВАЕТ ИСКЛЮЧЕНИЕ, с помощью которого CLR (то есть, среда выполнения!) уведомляется о неудачной попытке преобразования.

Файловый ввод-вывод

FileMode enumeration:

Описывает, каким образом операционная система должна открывать файл.

Имя члена	Описание
Append	Открывается существующий файл, и выполняется поиск конца файла, или же создается новый файл. FileMode.Append можно использовать только вместе с FileAccess.Write. Любая попытка чтения заканчивается неудачей и создает исключение ArgumentException.
Create	Описывает, что операционная система должна создавать новый файл. Если файл уже существует, он будет переписан. Это требует FileIOPermissionAccess.Write и FileIOPermissionAccess.Append. System.IO.FileMode.Create эквивалентно следующему запросу: если файл не существует, использовать CreateNew; в противном случае использовать Truncate.
CreateNew	Описывает, что операционная система должна создать новый файл. Это требует FileIOPermissionAccess.Write. Если файл уже существует, создается исключение IOException.
Open	Описывает, что операционная система должна открыть существующий файл. Возможность открыть данный файл зависит от значения, задаваемого FileAccess. Если данный файл не существует, создается исключение System.IO.FileNotFoundException.
OpenOrCreate	Указывает, что операционная система должна открыть файл, если он существует, в противном случае должен быть создан новый файл. Если файл открыт с помощью FileAccess.Read, требуется FileIOPermissionAccess.Read. Если файл имеет доступ FileAccess.ReadWrite и данный файл существует, требуется FileIOPermissionAccess.Write. Если файл имеет доступ FileAccess.ReadWrite и файл не существует, в дополнение к Read и Write требуется FileIOPermissionAccess.Append .
Truncate	Описывает, что операционная система должна открыть существующий файл. После открытия должно быть выполнено усечение файла таким образом, чтобы его размер стал равен нулю. Это требует FileIOPermissionAccess.Write. Попытка чтения из файла, открытого с помощью Truncate, вызывает исключение.

FileAccess enumerations:

Члены перечисления	Описание
Read	Read access to the file. Data can be read from the file. Combine with Write for read/write access.
ReadWrite	Read and write access to the file. Data can be written to and read from the file.
Write	Write access to the file. Data can be written to the file. Combine with Read for read/write access.

```
using System;
using System.IO;

namespace fstream00
{
class Class1
{
static string[] str = {
“1234567890”,
“qwertyuiop”,
“asdfghjkl”,
“zxcvbnm”,   };

static void Main(string[] args)
{
int i;
// Полное имя файла.
string filename = @”D:\Users\WORK\Cs\fstream00\test.txt”;
string xLine = “”;
char[] buff = new char[128];
for (i = 0; i < 128; i++) buff[i] = (char)25;
// Запись в файл.
FileStream fstr = new FileStream(filename,
FileMode.Create,
FileAccess.Write);
BufferedStream buffStream = new BufferedStream(fstr);
```

```
StreamWriter streamWr = new StreamWriter(buffStream);
for (i = 0; i < str.Length; i++)
 {
streamWr.WriteLine(str[i]);
 }

streamWr.Flush();
streamWr.Close();

Console.WriteLine("----------------------------------------------------------------");

fstr = new FileStream(filename,
FileMode.Open,
FileAccess.Read);

StreamReader streamRd = new StreamReader(fstr);
for ( ; xLine != null; )
{
xLine = streamRd.ReadLine();
Console.WriteLine(xLine);
}


Console.WriteLine("----------------------------------------------------------------");

fstr.Seek(0, SeekOrigin.Begin);
streamRd.Read(buff,0,10);
Console.WriteLine(new string(buff));
Console.WriteLine("1----------------------------------------------------------------");

streamRd.Read(buff,0,20);
Console.WriteLine(new string(buff));
Console.WriteLine("2----------------------------------------------------------------");

Console.WriteLine(streamRd.Read(buff,0,15));
i = (int)fstr.Seek(-20, SeekOrigin.Current);
Console.WriteLine(streamRd.ReadLine());

Console.WriteLine("3----------------------------------------------------------------");

}
}
}
```

Потоки

Процесс, поток, домен

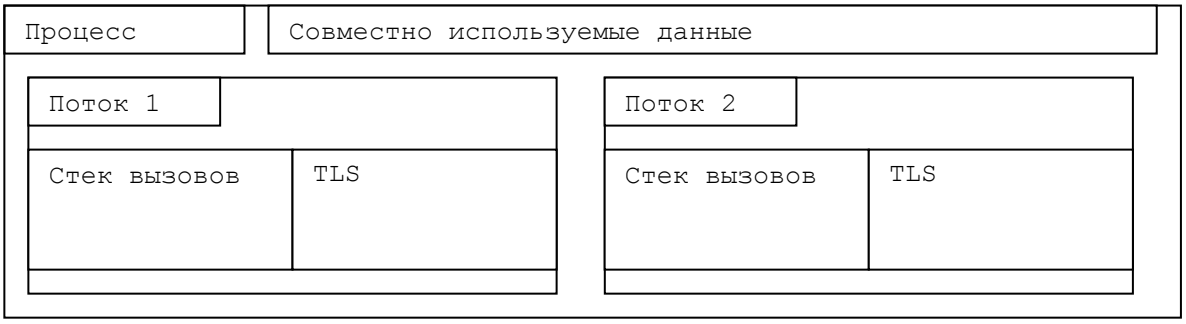
Процесс – объект, который создаётся Операционной Системой для каждого приложения в момент его запуска. Характеризуется собственным адресным пространством, которое напрямую недоступно другим процессам.

Поток. В рамках процесса создаются потоки (один – первичный создаётся всегда). Это последовательность выполняемых команд процессора. В приложении может быть несколько потоков (первичный поток и дополнительные потоки).

Потоки в процессе разделяют совместно используемые данные и имеют собственные стеки вызовов и локальную память потока (Thread Local Storage - TLS).

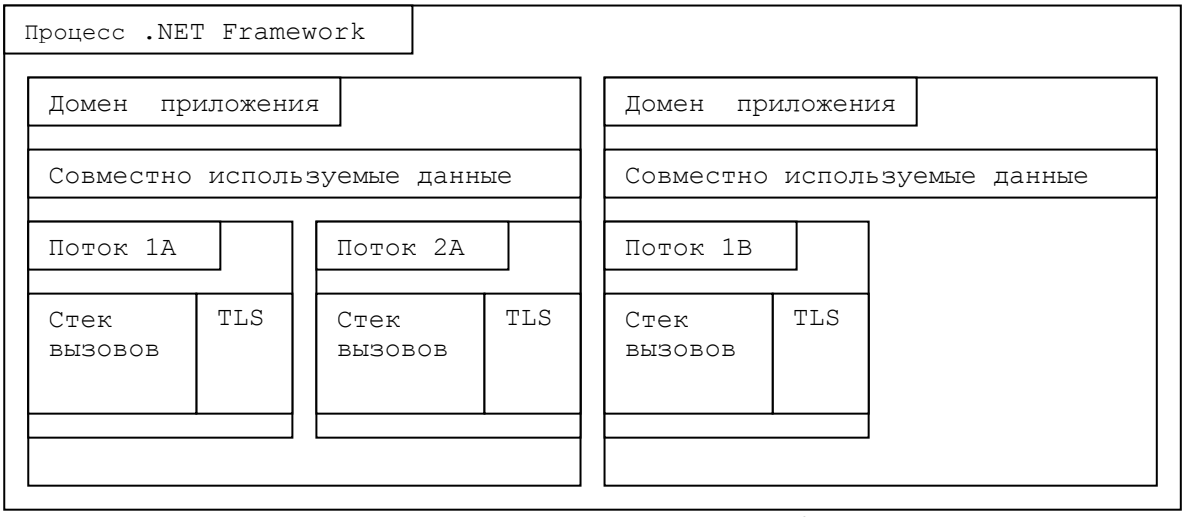
Главное – это то, что все потоки выполняются в рамках общего адресного пространства (в рамках одного процесса). Нет ничего, кроме выполняющихся потоков, которыми руководит управляющий поток ОС. Взаимодействие между одними потоками проще, если эти потоки выполняются в рамках одного процесса. При этом всё равно требуется специальный инструментарий (критические секции, мьютексы и семафоры) и остаётся множество проблем, решение которых требует внимания со стороны программиста.

Для обеспечения взаимодействия потоков в разных процессах используются каналы, обмен информацией через которые обеспечивается специальными системными средствами. Общее управление выполнением потоков осуществляется на уровне ОС.



Домен приложения

Выполнение приложений .NET всегда начинается с запуска .NET Framework. Это процесс со своими потоками, специальными атрибутами и правилами взаимодействия с другими процессами.



Домен приложения – это объект класса System.AppDomain.

Домен приложения полностью изолирует используемые в его рамках ресурсы (совместно используемые данные) как от других доменов того же самого процесса, так, естественно, от доменов приложения других процессов. Домены приложения одного процесса не могут совместно использовать никакие данные за исключением случаев, когда используется протокол удалённого доступа к данным .NET.

В одном процессе могут выполняться множества доменов приложений. При этом в рамках одного домена может выполняться множество потоков.

Методы класса System.AppDomain	Описание
CreateDomain()	Статический. Overloaded. Creates a new application domain.
GetCurrentThreadId()	Статический. Возвращает Id текущего потока.
Unload()	Статический. Для выгрузки из процесса указанного домена приложения.
BaseDirectory	Свойство. Возвращает базовый каталог, который используется РАСПОЗНАВАТЕЛЕМ для поиска нужных приложению сборок.
CreateInstance()	Overloaded. Creates a new instance of a specified type defined in a specified assembly.
ExecuteAssembly()	Запускает на выполнение сборку, имя которой было указано в качестве параметра.
GetAssemblies()	Возвращает список сборок, загруженных в текущий домен приложения.
Load()	Загружает сборку в текущий домен приложения.

Пример. Способы получения ссылки на текущий домен приложения и некоторые свойства доменов.

```
using System;
namespace Domains_00
{
class Class1
{
[STAThread]
static void Main(string[] args)
{
AppDomain appD1 = AppDomain.CurrentDomain;
AppDomain appD2 = Threading.Thread.GetDomain();

if (AppDomain.ReferenceEquals(appD1,appD2))
Console.WriteLine("The same! {0}, {1}.",appD1.FriendlyName,appD2.FriendlyName);
else
Console.WriteLine("Different! {0}, {1}.",appD1.FriendlyName,appD2.FriendlyName);
if (null != appD1.DynamicDirectory)
Console.WriteLine("{0}",appD1.DynamicDirectory);
else
Console.WriteLine("No DynamicDirectory.");

if (null != appD1.RelativeSearchPath)
Console.WriteLine("{0}",appD1.RelativeSearchPath);
else
Console.WriteLine("No RelativeSearchPath.");
}
}
}
```

Таким образом, ссылка на текущий домен может быть получена:

- как значение статического свойства System.AppDomain.CurrentDomain
- либо как результат выполнения метода System.Threading.Thread.GetDomain()

В рамках ОБЫЧНОГО (Win32) процесса может быть создано множество доменов. Причём новые домены можно создавать непосредственно в ходе выполнения .NET приложения. Порождение новых доменов обеспечивается четвёркой статических перегруженных методов-членов класса AppDomain с именем CreateDomain.

Создаётся новый домен приложения с именем, заданным строкой-параметром:

```
public static AppDomain CreateDomain(string);
```

Создаётся новый домен приложения с именем, заданным строкой-параметром, в рамках существующей политики безопасности:

```
public static AppDomain CreateDomain(string, Evidence);
```

Создаётся новый домен приложения using the specified name, evidence, and application domain setup information:

```
public static AppDomain CreateDomain(string, Evidence, AppDomainSetup);
```

Создаётся новый домен приложения the given name, using evidence, application base path, relative search path, and a parameter that specifies whether a shadow copy of an assembly is to be loaded into the application domain:

```
public static AppDomain CreateDomain(string, Evidence, string, string, bool);
```

Обзор пространства имён System.Threading

В этом пространстве объявляются типы, которые используются для создания многопоточных приложений: работа с потоком, средства синхронизации доступа к общим данным, примитивный вариант класса Timer… Много всего!

Тип	Назначение
Interlocked	Синхронизация доступа к общим данным.
Monitor	Синхронизация потоковых объектов при помощи блокировок и управления ожиданием.
Mutex	Синхронизация ПРОЦЕССОВ.
Thread	Собственно класс потока, работающего в среде выполнения .NET. В текущем домене приложения с помощью этого класса создаются новые потоки.
ThreadPool	Класс, предоставляющий средства управления набором взаимосвязанных потоков.
ThreadStart	Класс-делегат для метода, который должен быть выполнен перед запуском потока.
Timer	Вариант класса-делегата, который обеспечивает передачу управления некоторой функции-члену (неважно какого класса!) в указанное время. Сама процедура ожидания выполняется потоком в пуле потоков.
TimerCallback	Класс-делегат для объектов класса Timer.
WaitHandle	Объекты-представители этого класса являются объектами синхронизации (обеспечивают многократное ожидание).
WaitCallback	Делегат, представляющий методы для рабочих элементов (объектов) класса ThreadPool

Многопоточность

МНОГОПОТОЧНАЯ ОС. Прежде всего, операционная система должна допускать параллельную (псевдопараллельную) работу нескольких программ.

Многопоточное приложение: отдельные компоненты работают одновременно (псевдоодновременно), не мешая друг другу.

Случаи использования многопоточности:

- выполнение длительных процедур, ходом выполнения которых надо управлять,
- функциональное разделение программного кода: пользовательский интерфейс – функции обработки информации,
- обращение к серверам и службам Инернета, базам данных, передача данных по сети,
- одновременное выполнение нескольких задач, имеющих различный приоритет.

Виды многопоточности

- Переключательная многопоточность. Основа – резидентные программы. Программа размещалась в памяти компа вплоть до перезагрузки системы и управление ей

передавалось каким-либо заранее согласованным способом (предопределённой комбинацией клавиш на клавиатуре).

- Совместная многопоточность. Передача управления от одной программы другой. При этом возвращение управления – это проблема выполняемой программы. Возможность блокировки, при которой аварийно завершаются ВСЕ программы.
- Вытесняющая многопоточность. ОС централизованно выделяет всем запущенным приложениям определённый квант времени для выполнения в соответствии с приоритетом приложения. Реальная возможность работы нескольких приложений в ПСЕВДОПАРАЛЛЕЛЬНОМ режиме. “Зависание” одного приложения не является крахом для всей системы и оставшихся приложений.

А кто в домене живёт…

Прилагаемый пример демонструрует методы анализа процесса, домена и потоков.

Показывает потоки в процессе, а также сборки, которые выполняются в домене приложения.

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

// Это пространство имён требуется для работы с классом Assembly.
using System.Reflection;
// Это пространство имён требуется для работы с классом Process.
using System.Diagnostics;

namespace AppDomain1
{
class MyAppDomain
{
public static void ShowThreads()
{
Process proc = Process.GetCurrentProcess();
foreach(ProcessThread aPhysicalThread in proc.Threads)
{
Console.WriteLine(aPhysicalThread.Id.ToString() + ":" + aPhysicalThread.ThreadState);
}
}

public static void ShowAssemblies()
{
// Получили ссылку на домен.
AppDomain ad = AppDomain.CurrentDomain;
// В рамках домена может быть множество сборок.
// Можно получить список список сборок домена.
Assembly[] loadedAssemblies = ad.GetAssemblies();
// У домена имеется FriendlyName, которое ему присваивается
// при создании. При этом у него даже нет доступного конструктора.
Console.WriteLine("Assemblies in {0} domain:", ad.FriendlyName);
foreach(Assembly assembly in loadedAssemblies)
{
Console.WriteLine(assembly.FullName);
}
}

static void Main(string[] args)
{
Console.WriteLine("========================");
// MessageBox.Show("XXX"); - Это всего лишь вызов метода класса MessageBox.
// Вызов выполняется лишь потому, что в домен с самого начала
// загружается сборка System.Windows.Forms.dll.
MessageBox.Show("XXX");
ShowThreads();
ShowAssemblies();
Console.WriteLine("========================");
// Даже в таком простом приложении в рамках домена приложения живут три сборки!
}
}

}
```

Класс Thread. Общая характеристика

Thread класс представляет управляемые потоки. Создаёт потоки и управляет ими: устанавливает приоритет и статус потоков. Это объектная оболочка вокруг определённого этапа выполнения программы внутри домена приложения.

Статические члены класса Thread	Назначение
CurrentThread	Свойство. Только для чтения. Возвращает ссылку на поток, выполняемый в настоящее время.
GetData() SetData()	Обслуживание слота текущего потока.
GetDomain() GetDomainID()	Получение ссылки на домен приложения (на ID домена), в рамках которого работает указанный поток.
Sleep()	Блокировка выполнения потока на определенное время.

Нестатические члены	Назначение
IsAlive	Свойство. Если поток запущен, то true
IsBackground	Свойство. Работа в фоновом режиме. GC работает как фоновый поток.
Name	Свойство. Дружественное текстовое имя потока. Если поток никак не назван – значение свойства установлено в null. Поток может быть поименован единожды. Попытка переименования потока возбуждает исключение.
Priority	Свойство. Значение приоритета потока. Область значений – значения перечисления ThreadPriority.
ThreadState	Свойство. Состояние потока. Область значений – значения перечисления ThreadState.
Interrupt()	Прерывание работы текущего потока.
Join()	Ожидание появления другого потока (или определённого промежутка времени) с последующим завершением.
Resume()	Возобновление выполнения потока после приостановки.
Start()	Начало выполнения ранее созданного потока, представленного делегатом класса ThreadStart.
Suspend()	Приостановка выполнения потока.
Abort()	Завершение выполнения потока посредством генерации исключения TreadAbortException в останавливаемом потоке. Это исключение следует перехватывать для продолжения выполнения оставшихся потоков приложения. Перегруженный вариант метода содержит параметр типа object. An object that contains application-specific information, such as state, which can be used by the thread being aborted.

Именование потока

Потоки рождаются безымянными. Это означает, что у объекта, представляющего поток, свойство Name имеет значение null. Ничего страшного. Главный поток всё равно изначально поименовать некому. То же самое и с остальными потоками. Однако никто не может помешать потоку поинтересоваться своим именем. И получить имя. Несмотря на то, что это свойство при выполнении приложения играет вспомогательную роль, повторное переименование потоков недопустимо. Повторное изменение значения свойства Name приводит к возбуждению исключения.

```
using System;
using System.Threading;

namespace ThreadApp_1
{

class StartClass
{
static void Main(string[] args)
{//================================================================
int i = 0;
bool isNamed = false;
do
{
```

```
try
{
if (Thread.CurrentThread.Name == null)
{
Console.Write("Get the name for current Thread > ");
Thread.CurrentThread.Name = Console.ReadLine();
}
else
{
Console.WriteLine("Current Thread : {0}.", Thread.CurrentThread.Name);
if (!isNamed)
{
Console.Write("Rename it. Please...");
Thread.CurrentThread.Name = Console.ReadLine();
}
}
}
catch (InvalidOperationException e)
{
Console.WriteLine("{0}:{1}",e,e.Message);
isNamed = true;
}

i++;
}
while (i < 10);

}//=====================================================================
}
}
```

Игры с потоками

Но сначала о том, что должно происходить в потоке. Выполнение текущего потока предполагает выполнение программного кода.

Откуда этот код? От функций, естественно. От статических и нестатических методов-членов класса. И запустить поток можно единственным способом – указав точку входа потока - метод, к выполнению операторов которого должен приступить запускаемый поток.

Точкой входа ПЕРВИЧНОГО потока являются СТАТИЧЕСКИЕ функции Main или WinMain. Точнее, первый оператор метода.

Точка входа ВТОРИЧНОГО потока назначается при создании потока.

А дальше – как получится. Поток выполняется оператор за оператором. Со всеми циклами, заходами в вызываемые функции, в блоки свойств, в операторы конструкторов… И так продолжается до тех пор, пока не возникнет:

- неперехваченное исключение,
- не будет достигнут конец цепочки операторов (последний оператор в функциях Main или WinMain),
- не будет отработан вызов функции, обеспечивающей прекращение выполнения потока.

В силу того, что первичный поток создаётся и запускается автоматически (без какого-либо особого участия со стороны программиста), то и заботиться в случае простого однопоточного приложения не о чем.

При создании многопоточного приложения забота о создании дополнительных потоков – забота программиста. Здесь всё надо делать своими руками.

Деятельность по созданию потока предполагает три этапа:

- определение метода, который будет играть роль точки входа в поток,
- создание объекта-представителя специального класса-делегата (ThreadStart class), который настраивается на точку входа в поток,
- создание объекта-представителя класса потока. При создании объекта потока конструктору потока передаётся в качестве параметра ссылка на делегата, настроенного на точку входа.

Замечание. Точкой входа в поток не может быть конструктор, поскольку не существует делегатов, которые могли бы настраиваться на конструкторы.

Характеристики точки входа дополнительного потока

Сигнатура точки входа в поток определяется характеристиками класса-делегата.

```
public delegate void ThreadStart();
```

Класс-делегат С ПУСТЫМ СПИСКОМ параметров! Очевидно, что точка входа обязана соответствовать этой спецификации. У функции, представляющей точку входа должен быть ТОТ ЖЕ САМЫЙ список параметров! То есть, ПУСТОЙ список параметров!

Пустой список параметров функции, представляющей точку входа потока – это не самое страшное ограничение! Если учесть то обстоятельство, что создаваемый поток не является первичным потоком, то это означает, что вся необходимая входная информация может быть получена заранее и представлена в классе в доступном для функций-членов данного класса виде, то для функции, представляющей точку входа не составит особого труда эту информацию получить! А зато можно обойтись минимальным набором функций, обслуживающих данный поток.

Запуск вторичных потоков

Пример очень простой. Это всего лишь запуск пары потоков. Вторичные потоки запускаются последовательно из главного потока. А уже последовательность выполнения этих потоков определяется планировщиком.

Вторичный поток также вправе поинтересоваться о собственном имени. Надо всего лишь в правильном месте расположить этот код. И чтобы он выполнился в правильное время.

```
using System;
using System.Threading;

namespace ThreadApp_1
{
// Рабочий класс. Делает себе своё ДЕЛО...
class Worker
{
int allTimes;
int n;
// Конструктор умолчания...
public Worker()
{
n = 0;
allTimes = 0;
}
// Конструктор с параметрами...
public Worker(int nKey, int tKey)
{
n = nKey;
allTimes = tKey;
}

// Тело рабочей функции...
public void DoItEasy()
{//====================================
int i;
for (i = 0; i < allTimes; i++)
{
if (n == 0)
Console.Write("{0,25}\r",i);
else
Console.Write("{0,-25}\r",i);
}
Console.WriteLine("\nWorker was here!");
}//====================================
}

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

```
      {
Worker w0 = new Worker(0,100000);
Worker w1 = new Worker(1,100000);
ThreadStart t0, t1;
t0 = new ThreadStart(w0.DoItEasy);
t1 = new ThreadStart(w1.DoItEasy);
Thread th0, th1;
th0 = new Thread(t0);
th1 = new Thread(t1);

th0.Start();
th1.Start();
      }
}
}
```

Важно! Первичный поток ничем не лучше любых других потоков приложения. Он может скоропостижно завершиться раньше всех им же порождённых потоков! Приложение же завершается после выполнения ПОСЛЕДНЕЙ команды в ПОСЛЕДНЕМ выполняемом потоке. Неважно в каком.

Приостановка выполнения потока

Обеспечивается статическим методом Sleep(). Метод статический – это значит, что всегда производится не “усыпление”, а “САМОусыпление” выполняемого в данный момент потока. Выполнение методов текущего потока блокируется на определённые интервалы времени. Всё зависит от выбора перегруженного варианта метода. Планировщик потоков смотрит на поток и принимает решение относительно того, можно ли продолжить выполнение усыплённого потока.

В самом простом случае целочисленный параметр определяет временной интервал блокировки потока в миллисекундах.

Если значение параметра установлено в 0, поток будет остановлен до того момента, пока не будет предоставлен очередной интервал для выполнения операторов потока.

Если значение интервала задано с помощью объекта класса TimeSpan, то момент, когда может быть возобновлено выполнение потока, определяется с учётом закодированной в этом объекте информации.

```
// Поток заснул на 1 час, 2 минуты, 3 секунды:
Thread.Sleep(new TimeSpan(1,2,3));
:::::
// Поток заснул на 1 день, 2 часа, 3 минуты, 4 секунды, 5 милисекунд:
Thread.Sleep(new TimeSpan(1,2,3,4,5));
```

Значение параметра, представленное выражением

```
System.Threading.Timeout.Infinite
```

позволяет усыпить поток на неопределённое время. А разбудить поток при этом можно с помощью метода Interrupt(), который в этом случае вызывается из другого потока.

```
using System;
using System.Threading;

class Worker
{
int allTimes;
// Конструктор с параметрами...
public Worker(int tKey)
{
allTimes = tKey;
}

// Тело рабочей функции...
public void DoItEasy()
{//====================================
int i;
```

```
for (i = 0; i < allTimes; i++)
{
Console.Write("{0}\r",i);
if (i == 5000)
{
try
{
Console.WriteLine("\nThread go to sleep!");
Thread.Sleep(System.Threading.Timeout.Infinite);
}
catch (ThreadInterruptedException e)
{
Console.WriteLine("{0}, {1}",e,e.Message);
}
}
}
Console.WriteLine("\nWorker was here!");
}//==================================
}

class StartClass
{
static void Main(string[] args)
{
Worker w0 = new Worker(10000);
ThreadStart t0;
t0 = new ThreadStart(w0.DoItEasy);
Thread th0;
th0 = new Thread(t0);
th0.Start();
Thread.Sleep(10000);
if (th0.ThreadState.Equals(ThreadState.WaitSleepJoin)) th0.Interrupt();
Console.WriteLine("MainThread was here...");
}
}
```

И всегда надо помнить: приложение выполняется до тех пор, пока не будет выполнен последний оператор последнего потока. И не важно, выполняются ли при этом потоки, "спят", либо просто заблокированы.

Отстранение потока от выполнения

Обеспечивается нестатическим методом Suspend(). Поток входит в "кому", из которой его можно вывести, вызвав метод Resume(). Этот вызов, естественно, должен исходить из другого потока. Если все неотстранённые от выполнения потоки оказались завершены и некому запустить отстранённый поток – приложение в буквальном смысле "зависает". Операторы в потоке могут выполняться, а выполнить их невозможно по причине отстранения потока от выполнения.

```
using System;
using System.Threading;


public class ThreadWork
{
public static void DoWork()
{
for(int i=0; i<10; i++)
{
Console.WriteLine("Thread - working.");
Thread.Sleep(10);
}

Console.WriteLine("Thread - still alive and working.");
Console.WriteLine("Thread - finished working.");
}
}

class ThreadAbortTest
```

```
{
public static void Main()
{
ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
Thread myThread = new Thread(myThreadDelegate);
myThread.Start();
Thread.Sleep(10);
Console.WriteLine("Main - aborting my thread.");
myThread.Suspend();
Console.WriteLine("Main ending.");
}
}
```

Завершение потоков

- Первый вариант остановки потока тривиален. Поток завершается после выполнения ПОСЛЕДНЕГО оператора выполняемой цепочки операторов. Допустим, в ходе выполнения условного оператора значение некоторой переменной сравнивается с фиксированным значением и в случае совпадения значений управление передаётся оператору return:

```
for (x=0;;x++)
{
if (x==max)
return; // Всё. Этот оператор оказался последним.
else
 {
 ::::::::::
 }
}
```

- Поток может быть остановлен в результате выполнения метода Abort(). Эта остановка является достаточно сложным делом.

1. При выполнении этого метода происходит активация исключения ThreadAbortException. Естественно, это исключение может быть перехвачено в соответствующем блоке catch. Во время обработки исключения допустимо выполнение самых разных действий, которые осуществляются в этом самом "остановленном" потоке. В том числе, возможна и реанимация остановленного потока путём вызова метода ResetAbort().
2. При перехвате исключения CLR обеспечивает выполнение операторов блоков finally, которые выполняются всё в том же потоке.

Таким образом, остановка потока путём вызова метода Abort не может рассматриваться как НЕМЕДЛЕННАЯ остановка выполнения потока.

```
using System;
using System.Threading;

public class ThreadWork
{
public static void DoWork()
{
int i;

try
{
 for(i=0; i<100; i++)
  {
    //4. Вот скромненько так работает...
    //   Поспит немножко - а потом опять поработает.
    //   Take Your time! 100 раз прокрутиться надо
    //   вторичному потоку до нормального завершения.
    Console.WriteLine("Thread - working {0}.", i);
    Thread.Sleep(10);
   }
}
catch(ThreadAbortException e)
{
 //6.
```

```
 //- Ну дела! А где это мы...
 Console.WriteLine("Thread - caught ThreadAbortException - resetting.");
 Console.WriteLine("Exception message: {0}", e.Message);
 //  (Голос сверху)
 //- Вы находитесь в блоке обработки исключения, связанного с
 //  непредвиденным завершением потока.
 //- Понятно… Значит, не успели. "Наверху" сочли нашу деятельность
 //  Нецелесообразной и не дали (потоку) завершить до конца начатое дело!
 Thread.ResetAbort();
 //   (Перехватывают исключение и отменяют остановку потока)
 //   Будем завершать дела. Но будем делать это как положено,
 //   а не в аварийном порядке. Нам указали на дверь, но мы
 //   уходим достойно!
 //   (Комментарии постороннего)
 //   А чтобы стал понятен альтернативный исход - надо
 //   закомментировать строку с оператором отмены остановки потока.
}
finally
{
 //7.
 //- Вот где бы мы остались, если бы не удалось отменить
 // остановку потока! finally блок... Отстой!
 Console.WriteLine("Thread - in finally statement.");
}

//8.
// - А вот преждевременный, но достойный уход.
//   Мы не довели дело до конца только потому, что нам не дали
//   сделать этого. Обстоятельства бывают выше. Уходим достойно.
Console.WriteLine("Thread - still alive and working.");
Console.WriteLine("Thread - finished working.");
}
}

class ThreadAbortTest
{
public static void Main()
{
//1. Мероприятия по организации вторичного потока!
ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
Thread myThread = new Thread(myThreadDelegate);
//2. Вторичный поток стартовал!
myThread.Start();

//3. А вот первичный поток - самоусыпился!
//   и пока первичный поток спит, вторичный поток - работает!
Thread.Sleep(50);

//5. Но вот первичный поток проснулся - и первое, что он
//   делает - это прерывает вторичный поток!
Console.WriteLine("Main - aborting my thread.");
myThread.Abort();

//9. А в столицах тоже все дела посворачивали...
Console.WriteLine("Main ending.");
}
}
```

Метод Join()

Несколько потоков выполняются “параллельно” в соответствии с предпочтениями планировщика потоков. Нестатический метод Join() позволяет изменить последовательность выполнения потоков многопоточного приложения. Метод Join() выполняется в одном из потоков по отношению к другому потоку.

В результате выполнения этого метода данный текущий поток немедленно блокируется до тех пор, пока не завершит своё выполнение поток, по отношению к которому был вызван метод Join.

Перегруженный вариант метода имеет целочисленный аргумент, который воспринимается как временной интервал. В этом случае выполнение текущего

потока может быть возобновлено по истечении этого периода времени до завершения этого потока.

```
using System;
using System.Threading;

public class ThreadWork
{
public static void DoWork()
{
for(int i=0; i<10; i++)
{
Console.WriteLine("Thread - working.");
Thread.Sleep(10);
}

Console.WriteLine("Thread - finished working.");
}
}

class ThreadTest
{
public static void Main()
{
ThreadStart myThreadDelegate = new ThreadStart(ThreadWork.DoWork);
Thread myThread = new Thread(myThreadDelegate);
myThread.Start();
Thread.Sleep(100);
myThread.Join(); // Закомментировать вызов метода и почувствовать разницу.
Console.WriteLine("Main ending.");
}
}
```

Состояния потока (перечисление ThreadState)

Класс ThreadState определяет набор всех возможных состояний выполнения для потока. После создания потока и до завершения он находится по крайней мере в одном из состояний. Потоки, созданные в общеязыковой среде выполнения, изначально находятся в состоянии Unstarted, в то время как внешние потоки, приходящие в среду выполнения, находятся уже в состоянии Running. Потоки с состоянием Unstarted переходят в состояние Running при вызове метода Start. Не все комбинации значений ThreadState являются допустимыми; например, поток не может быть одновременно в состояниях Aborted и Unstarted.

В следующей таблице перечислены действия, вызывающие смену состояния.

Действие	Состояние Потока
Поток создается в среде CLR.	Unstarted
Поток вызывает метод Start.	Running
Поток начинает выполнение.	Running
Поток вызывает метод Sleep.	WaitSleepJoin
Поток вызывает метод Wait для другого объекта.	WaitSleepJoin
Поток вызывает метод Join для другого потока.	WaitSleepJoin
Другой поток вызывает метод Interrupt.	Running
Другой поток вызывает метод Suspend.	SuspendRequested
Поток отвечает на запрос метода Suspend.	Suspended
Другой поток вызывает метод Resume.	Running
Другой поток вызывает метод Abort.	AbortRequested
Поток отвечает на запрос метода Abort.	Stopped
Поток завершен.	Stopped

Начальное состояние потока (если это не главный поток), в котором он оказывается непосредственно после его создания – Unstarted. В этом состоянии он пребывает до тех пор, пока вызовом метода Start() он не будет переведён в состояние Running.

В дополнение к вышеперечисленным состояниям, существует также Background состояние, которое указывает, выполняется ли поток на фоне или на переднем плане.

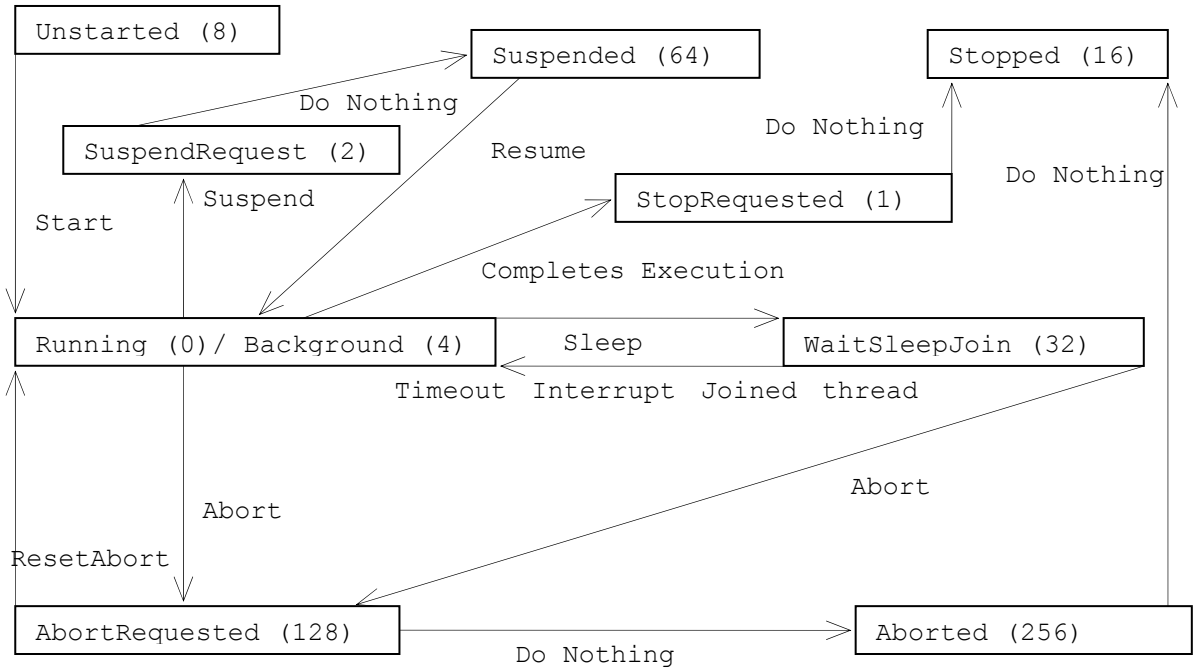
Свойство Thread.ThreadState потока содержит текущее состояние потока. Для определения текущего состояния потока в приложении можно использовать битовые маски. Пример условного выражения:

```
if((myThread.ThreadState & (ThreadState.Stopped | ThreadState.Unstarted)) == 0) {…}
```

Члены перечисления:

Имя члена	Описание	Значение
Aborted	Поток находится в Stopped состоянии.	256
AbortRequested	Метод Thread.Abort был вызван для потока, но поток еще не получил задерживающийся объект System.Threading.ThreadAbortException, который будет пытаться завершить поток.	128
Background	Поток выполняется как фоновый поток, что является противоположным к приоритетному потоку. Это состояние контролируется заданием свойства Thread.IsBackground.	4
Running	Поток был запущен, он не заблокирован, и нет задерживающегося объекта ThreadAbortException.	0
Stopped	Процесс остановлен.	16
StopRequested	Поток запрашивается на остановку. Это только для внутреннего использования.	1
Suspended	Работа потока была приостановлена.	64
SuspendRequested	Запрашивается приостановка работы потока.	2
Unstarted	Метод Thread.Start не был вызван для потока.	8
WaitSleepJoin	Поток заблокирован в результате вызова к методам Wait, Sleep или Join.	32

Ниже приводится диаграмма состояний потока.



Одновременное пребывание потока в различных состояниях

В условиях многопоточного приложения разные потоки могут переводить друг друга в разные состояния. Таким образом, поток может находиться одновременно в БОЛЕЕ ЧЕМ ОДНОМ Состоянии.

Например, если поток блокирован в результате вызова метода Wait, а другой поток вызвал по отношению к блокированному потоку метод Abort, то

блокированный поток окажется в одно и то же время в состояниях WaitSleepJoin и AbortRequested.

В этом случае, как только поток выйдет из состояния WaitSleepJoin (в котором он оказался в результате выполнения метода Wait), ему будет предъявлено исключение ThreadAbortException, связанное с началом процедуры aborting.

С другой стороны, не все сочетания значений ThreadState допустимы. Например, поток не может одновременно находиться в состояниях Aborted и Unstarted. Перевод потока из одного состояния в другое несовместимое с ним состояние, а также повторная попытка перевода потока в одно и то же состояние (пара потоков один за другим применяют метод Resume() к одному и тому же потоку), может привести к генерации исключения. Поэтому операторы, связанные с управлением потоками следует размещать в блоках try.

Информация о возможности одновременного пребывания потока в нескольких состояниях представлена таблице допустимых состояний:

	AR	Ab	Back	U	S	R	W	St	SusR	StopR
AbortRequested	--	N	Y	Y	Y	N	Y	N	Y	N
Aborted	N	--	Y	N	N	N	N	N	N	N
Background	Y	Y	--	Y	Y	N	Y	Y	Y	N
Unstarted	Y	N	Y	--	N	N	N	N	N	N
Suspended	Y	N	Y	N	--	N	Y	N	N	N
Running	N	N	N	N	N	--	N	N	N	N
WaitSleepJoin	Y	N	Y	N	Y	N	--	N	Y	N
Stopped	N	N	Y	N	N	N	N	--	Y	N
SuspendRequested	Y	N	Y	N	N	N	Y	N	--	N
StopRequested	N	N	N	N	N	N	N	N	N	--

Фоновый поток

Потоки выполняются:

- в обычном режиме (Foreground threads) и
- в фоновом режиме (Background threads).

Состояние Background state распознаётся по значению свойства IsBackground, которое указывает на режим выполнения потока: background или foreground.

Любой Foreground поток можно перевести в фоновый режим, установив значение свойства IsBackground в true.

Завершение Background потока не влияет на завершение приложения в целом.

Завершение последнего Foreground потока приводит к завершению приложения, независимо от состояния потоков, выполняемых в фоновом режиме.

Ниже в примере один из потоков переводится в фоновый режим. Изменяя значения переменных, определяющих характеристики циклов, можно проследить за поведением потоков.

```
using System;
using System.Threading;

namespace ThreadApp_1
{

class Worker
{
int allTimes;
public Worker(int tKey)
{
allTimes = tKey;
}
// Тело рабочей функции...
public void DoItEasy()
{//====================================
int i;
for (i = 0; i < allTimes; i++)
{
Console.WriteLine("Back thread >>>> {0}\r",i);
}
```

```
Console.WriteLine("\nBackground thread was here!");
}//===================================
}

class StartClass
{
static void Main(string[] args)
{
long i;
Worker w0 = new Worker(100000);
ThreadStart t0;
t0 = new ThreadStart(w0.DoItEasy);
Thread th0;
th0 = new Thread(t0);
th0.IsBackground = true;
th0.Start();
for (i = 0; i < 100 ; i++)
{
Console.WriteLine("Fore thread: {0}", i);
}

Console.WriteLine("Foreground thread ended");
}
}
}
```

Приоритет потока

Задаётся значениями перечисления ThreadPriority. Эти значения используются при планировке очерёдности выполнения потоков в ПРОЦЕССЕ.

Приоритет потока определяет относительный приоритет потоков.

Каждый поток имеет собственный приоритет. Изначально он задаётся как Normal priority.

Алгоритм планировки выполнения потока позволяет системе определить последовательность выполнения потоков. Операционная система может также корректировать приоритет потока динамически, переводя поток из состояния foreground в background.

Значение приоритета не влияет на состояние потока. Система планирует последовательность выполнения потоков на основе информации о состоянии потока.

Highest	Потоки с низшим уровнем приоритета выполняются лишь после того, как в процессе будет завершено выполнение потоков с более высоким приоритетом.	Приоритет процесса
AboveNormal		
Normal		
BelowNormal		
Lowest		

Приоритет потока – это относительная величина. Прежде всего, система планирует очерёдность выполнения ПРОЦЕССА. В рамках выполняемого процесса определяется последовательность выполнения потоков.

Передача данных во вторичный поток

Делегат-представитель класса-делегата ThreadStart обеспечивает запуск вторичных потоков. Это элемент СТАНДАРТНОГО механизма поддержки вторичных потоков. Именно этим и объясняется главная особенность этого делегата: настраиваемые с его помощью стартовые функции потоков НЕ имеют параметров и не возвращают значений. Это означает, что невозможно осуществить запуск потока с помощью метода, имеющего параметры, а также получить какое-либо значение при завершении стартовой функции потока.

Ну и ладно! Всё равно возвращаемое значение стартовой функции при существующем механизме запуска потока (функция Start) некому перехватывать, а стандартный жёсткий механизм предопределённых параметров (как у функции Main) ничуть не лучше его полного отсутствия.

Если, конечно, существуют простые средства передачи данных в поток.

Так вот такие средства существуют.

Дело в том, что вторичный поток строится на основе методов конкретного класса. Это означает, что сначала создаётся объект-представитель класса,

затем объект потока, с настроенным на стартовую функцию делегатом, после чего поток стандартным образом запускается.

При этих условиях задача передачи данных потоку может быть возложена на конструкторы класса. На списки их параметров никаких особых ограничений не накладывается. В конструкторе могут быть реализованы самые сложные алгоритмы подготовки данных. Таким образом, “место для битвы” может быть подготовлено задолго до начала выполнения потока.

```
using System;
using System.Threading;

// Класс WorkThread содержит всю необходимую для выполнения
// данной задачи информацию, а также и соответствующий метод.
public class WorkThread
{
// State information used in the task.
private string entryInformation;
private int value;

// Конструктор получает всю необходимую информацию
// через параметры.
public WorkThread(string text, int number)
{
entryInformation = text;
value = number;
}

// Рабочий метод потока непосредственно после своего запуска
// ЛЕГКО прочитывает всю необходимую информацию.
public void ThreadProc()
{
Console.WriteLine(entryInformation, value);
}
}

// Точка входа приложения.
//
public class Example
{
public static void Main()
{
// Подготовка к запуску вторичного потока предполагает создание
// объекта класса потока. В этот момент вся необходимая для работы потока
// информация передаётся через параметры конструктора.
// Здесь переданы необходимые детали, которые будут составлены
// стандартным образом в строку методом WriteLine.
WorkThread tws = new WorkThread(“This report displays the number {0}.”, 125);
// Создали объект потока, затем его запустили.
Thread t = new Thread(new ThreadStart(tws.ThreadProc));
t.Start();
Console.WriteLine(«Первичный поток поработал. Теперь ждёт первой звезды...»);
t.Join();
Console.WriteLine(«Вторичный поток отработал. Главный поток остановлен. Всё...»);
}
}
```

Извлечение значений (данных) с помощью Callback методов

А для анализа результата выполнения вторичного потока можно использовать метод класса, отвечающего за запуск вторичного потока. Соответствующим образом настроенный делегат также может быть передан в качестве параметра конструктору, отвечающему за организацию вторичного потока. Вызывать метод класса, запустившего вторичный поток, можно будет по окончании работы во ВТОРИЧНОМ потоке.

Важно! ЭТА ФУНКЦИЯ сама будет выполняться во ВТОРИЧНОМ потоке! Однако если главный поток ещё не завершён, а остаётся в ожидании результатов (например, в результате выполнения метода Join), это обстоятельство не является проблемой.

Будучи сама членом класса, эта функция-член имеет доступ ко всем данным и методам своего класса. И неважно, в каком в каком потоке она выполняется.

```
using System;
using System.Threading;

// Класс WorkThread включает необходимую информацию,
// метод и делегат для вызова метода, который запускается
// после выполнения задачи.
public class WorkThread
{
// Входная информация.
private string entryInformation;
private int value;
// Ссылка на объект-представитель класса-делегата, с помощью которого
// вызывается метод обратного вызова. Сам класс-делегат объявляется позже.
private CallbackMethod callback;

// Конструктор получает кходную информацию и настраивает
// callback delegate.
public WorkThread(string text, int number,
CallbackMethod callbackDelegate)
{
entryInformation = text;
value = number;
callback = callbackDelegate;
}

// Метод, обеспечивающий выплнение поставленной задачи:
// составляет строку и после дополнительной проверки настройки
// callback делегата обеспечивает вызов метода.
public void ThreadProc()
{
Console.WriteLine(entryInformation, value);
if (callback != null)
callback(1); // Вот, вызвал ЧУЖОЙ МЕТОД в СВОЁМ потоке.
}
}

// Класс-делегат задаёт сигнатуру callback методу.
//
public delegate void CallbackMethod(int lineCount);
// Entry Point for the example.
//
public class Example
{
public static void Main()
{
// Supply the state information required by the task.
WorkThread tws = new WorkThread("This report displays the number {0}.",
125,
new CallbackMethod(ResultCallback));
Thread t = new Thread(new ThreadStart(tws.ThreadProc));
t.Start();
Console.WriteLine(«Первичный поток поработал. Теперь ждёт первой звезды...»);
t.Join();
Console.WriteLine(«Вторичный поток отработал. Главный поток остановлен. Всё...»);
}

// Callback метод, естественно, соответствует сигнатуре callback класса делегата.
public static void ResultCallback(int lineCount)
{
Console.WriteLine("Вторичный поток обработал {0} строк.", lineCount);
}
}
```

Callback метод – метод-член класса, запустившего вторичный поток. Этот метод запускается "в качестве уведомления" о том, что вторичный поток "завершил выполнение своей миссии". Главная проблема Callback метода заключается в том, что он выполняется в "чужом" потоке.

Организация взаимодействия потоков

1. Посредством общедоступных (public) данных

В предлагаемом примере вторичные потоки взаимодействуют между собой через общедоступный объект, располагаемый в общей памяти потоков. Данные для обработки подготавливаются соответствующими конструкторами классов, отвечающих за выполнение вторичных потоков. Конструктор класса, выполняемого во вторичном потоке, также отвечает за организацию уведомления главного потока о результатах взаимосвязанной деятельности вторичных потоков.

```
using System;
using System.Threading;

namespace CommunicatingThreadsData
{

public delegate void CallBackFromStartClass (long param);
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
public long lVal;
public CommonData(long key)
{
lVal = key;
}
}


// Классы Worker и Inspector: основа взаимодействующих потоков.
class Worker
{
CommonData cd;
// Конструктор...
public Worker(ref CommonData rCDKey)
{
cd = rCDKey;
}

public void startWorker()
{
DoIt(ref cd);
}

// Тело рабочей функции...
public void DoIt(ref CommonData cData)
{//====================================
for (;;)
{
cData.lVal++; // Изменили значение…
Console.Write(«{0,25}\r»,cData.lVal); // Сообщили о результатах.
}
}//====================================
}

class Inspector
{
long stopVal;
CommonData cd;
CallBackFromStartClass callBack;

// Конструктор... Подготовка делегата для запуска CallBack метода.
public Inspector(ref CommonData rCDKey, long key, CallBackFromStartClass cbKey)
{
stopVal = key;
cd = rCDKey;
callBack = cbKey;
}

public void startInspector()
```

```
{
measureIt(ref cd);
}


// Тело рабочей функции...
public void measureIt(ref CommonData cData)
{//====================================
for (;;)
{
if (cData.lVal < stopVal)
{
Thread.Sleep(100);
Console.WriteLine("\n{0,-25}",cData.lVal);
}
else
callBack(cData.lVal);
}
}//====================================
}


class StartClass
{

static Thread th0, th1;
static CommonData cd;
static long result = 0;


static void Main(string[] args)
{

StartClass.cd = new CommonData(0);
// Конструкторы классов Worker и Inspector несут дополнительную нагрузку.
// Они обеспечивают необходимыми значениями методы,
// выполняемые во вторичных потоках.

Worker work;
// До начала выполнения потока вся необходимая информация доступна методу.
work = new Worker(ref cd);
Inspector insp;
// На инспектора возложена дополнительная обязанность вызова функции-терминатора.
// Для этого используется специально определяемый и настраиваемый делегат.
insp = new Inspector(ref cd,
50000,
new CallBackFromStartClass(StartClass.StopMain));

// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart. Поэтому они не имеют параметров.
ThreadStart t0, t1;
t0 = new ThreadStart(work.startWorker);
t1 = new ThreadStart(insp.startInspector);

// Созданы вторичные потоки.
StartClass.th0 = new Thread(t0);
StartClass.th1 = new Thread(t1);

// Запущены вторичные потоки.
StartClass.th0.Start();
StartClass.th1.Start();

// Ещё раз о методе Join(): Выполнение главного потока приостановлено.
StartClass.th0.Join();
StartClass.th1.Join();

// Потому последнее слово остаётся за главным потоком приложения.
Console.WriteLine("Main(): All stoped at {0}. Bye.», result);

}
```

```
// Функция-член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public static void StopMain(long key)
{
Console.WriteLine("StopMain: All stoped at {0}...", key);
// Остановка рабочих потоков. Её выполняет функция-член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЁ о вторичных потоках. Но выполняется она
// в ЧУЖОМ (вторичном) потоке. Поэтому:
// 1. надо предпринять особые дополнительные усилия для того чтобы
// результат работы потоков оказался доступен в главном потоке.
/*StartClass.*/result = key;
// 2. очень важна последовательность остановки потоков,
StartClass.th0.Abort();
StartClass.th1.Abort();

// Этот оператор не выполняется! Поток, в котором выполняется
// метод-член класса StartClass StopMain() остановлен.
Console.WriteLine(«StopMain(): bye.»);
}
}
}
```

2. Посредством общедоступных (public) свойств

Следующий вариант организации взаимодействия между потоками основан на использовании общедоступных свойств. От предыдущего примера отличается тем, что доступ к закрытому счётчику lVal в соответствии с принципами инкапсуляции осуществляется через свойство с блоками get (акцессор) и set (мутатор).

```
using System;
using System.Threading;

namespace CommunicatingThreadsData
{
public delegate void CallBackFromStartClass (long param);
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
private long lVal;
public long lValProp
{
get
{
return lVal;
}
set
{
lVal = value;
}
}

public CommonData(long key)
{
lVal = key;
}
}


// Классы Worker и Inspector: основа взаимодействующих потоков.
class Worker
{
CommonData cd;
// Конструктор умолчания...
public Worker(ref CommonData rCDKey)
{
cd = rCDKey;
}
public void startWorker()
{
```

```
DoIt(ref cd);
}

// Тело рабочей функции...
public void DoIt(ref CommonData cData)
{//====================================
for (;;)
{
cData.lValProp++;
Console.Write("{0,25}\r",cData.lValProp);
}
}//====================================
}

class Inspector
{
long stopVal;
CommonData cd;
CallBackFromStartClass callBack;

// Конструктор...
public Inspector(ref CommonData rCDKey, long key, CallBackFromStartClass cbKey)
{
stopVal = key;
cd = rCDKey;
callBack = cbKey;
}

public void startInspector()
{
measureIt(ref cd);
}


// Тело рабочей функции...
public void measureIt(ref CommonData cData)
{//====================================
for (;;)
{
if (cData.lValProp < stopVal)
{
Thread.Sleep(100);
Console.WriteLine("\n{0,-25}",cData.lValProp);
}
else
callBack(cData.lValProp);
}
}//====================================
}


class StartClass
{
static Thread th0, th1;
static CommonData cd;
static long result = 0;

static void Main(string[] args)
{
StartClass.cd = new CommonData(0);
// Конструкторы классов Worker и Inspector несут дополнительную нагрузку.
// Они обеспечивают необходимыми значениями методы,
// выполняемые во вторичных потоках.

Worker work;
// До начала выполнения потока вся необходимая информация доступна методу.
work = new Worker(ref cd);
Inspector insp;
// На инспектора возложена дополнительная обязанность вызова функции-терминатора.
// Для этого используется специально определяемый и настраиваемый делегат.
insp = new Inspector(ref cd, 50000, new CallBackFromStartClass(StartClass.StopMain));
```

```
// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart. Поэтому они не имеют параметров.
ThreadStart t0, t1;
t0 = new ThreadStart(work.startWorker);
t1 = new ThreadStart(insp.startInspector);

// Созданы вторичные потоки.
StartClass.th0 = new Thread(t0);
StartClass.th1 = new Thread(t1);

// Запущены вторичные потоки.
StartClass.th0.Start();
StartClass.th1.Start();

// Ещё раз о методе Join(): Выполнение главного потока приостановлено.
StartClass.th0.Join();
StartClass.th1.Join();

// Потому последнее слово остаётся за главным потоком приложения.
Console.WriteLine("Main(): All stoped at {0}. Bye.», result);
}

// Функция-член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public static void StopMain(long key)
{
Console.WriteLine("StopMain: All stoped at {0}...", key);
// Остановка рабочих потоков. Её выполняет функция-член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЁ о вторичных потоках. Но выполняется она
// в ЧУЖОМ (вторичном) потоке. Поэтому:
// 1. надо предпринять особые дополнительные усилия для того чтобы
// результат работы потоков оказался доступен в главном потоке.
/*StartClass.*/result = key;
// 2. очень важна последовательность остановки потоков,
StartClass.th0.Abort();
StartClass.th1.Abort();
// Этот оператор не выполняется! Поток, в котором выполняется
// метод-член класса StartClass StopMain() остановлен.

Console.WriteLine("StopMain(): bye.");
}
}
}
```

3. Посредством общедоступных очередей

Взаимодействующие потоки выполняют поставленную перед ними задачу. При этом один поток обеспечивает генерацию данных, а второй – обработку получаемых данных. Если при этом время, необходимое для генерации данных и время обработки данных различаются, проблема взаимодействия потоков может быть решена посредством очереди данных, которая в этом случае играет роль буфера между двумя потоками. Первый поток размещает данные в очередь "с одного конца", абсолютно не интересуясь успехами потока-обработчика. Второй поток извлекает данные из очереди "с другого конца" руководствуясь исключительно состоянием этой очереди. Отсутствие данных в очереди для потока-обработчика означает успешное выполнение поставленной задачи или сигнал к самоусыплению.

Организация работы потоков по этой схеме предполагает:

- выделение обрабатываемых данных в отдельный класс,
- создание общедоступного объекта-представителя класса "Очередь" с интерфейсом, обеспечивающим размещение и извлечение данных,
- разработку классов, содержащих методы генерации и обработки данных, реализующих интерфейс доступа к данным,
- разработку методов обратного вызова, сообщающих о результатах выполнения задач генерации и обработки данных,
- создание и запуск потока, обеспечивающего генерацию данных и размещение данных в очереди,

- создание и запуск потока, обеспечивающего извлечение данных из очереди и обработку данных.

Пример

```
using System;
using System.Threading;


using System.Collections;
namespace CommunicatingThreadsQueue
{

public delegate void CallBackFromStartClass (string param);
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
private int iVal;
public int iValProp
{
get
{
return iVal;
}
set
{
iVal = value;
}
}

public CommonData(int key)
{
iVal = key;
}
}


// Классы Receiver и Sender: основа взаимодействующих потоков.
class Receiver
{
Queue cdQueue;
CallBackFromStartClass callBack;

// Конструктор умолчания...
public Receiver(ref Queue queueKey, CallBackFromStartClass cbKey)
{
cdQueue = queueKey;
callBack = cbKey;
}

public void startReceiver()
{
DoIt();
}

// Тело рабочей функции...
public void DoIt()
{//====================================
CommonData cd = null;
while (true)
{

Console.WriteLine("Receiver. notifications in queue: {0}",cdQueue.Count);
if (cdQueue.Count > 0)
{
cd = (CommonData)cdQueue.Dequeue();
if (cd == null)
Console.WriteLine("?????");
else
{
Console.WriteLine("Process started ({0}).", cd.iValProp);
```

```
// Выбрать какой-нибудь из способов обработки полученного уведомления.
// Заснуть на соответствующее количество тиков.
//Thread.Sleep(cd.iValProp);
// Заняться элементарной арифметикой. С усыплением потока.
while (cd.iValProp != 0)
{
cd.iValProp--;
Thread.Sleep(cd.iValProp);
Console.WriteLine("process:{0}",cd.iValProp);
}
}
}
else
callBack("Receiver");

Thread.Sleep(100);
}
}//====================================
}

class Sender
{
Random rnd;
int stopVal;
Queue cdQueue;
CallBackFromStartClass callBack;

// Конструктор...
public Sender(ref Queue queueKey, int key, CallBackFromStartClass cbKey)
{
rnd = new Random(key);
stopVal = key;
cdQueue = queueKey;
callBack = cbKey;
}

public void startSender()
{
sendIt();
}


// Тело рабочей функции...
public void sendIt()
{//====================================

while (true)
{
if (stopVal > 0)
{
// Размещение в очереди нового члена со случайными характеристиками.
cdQueue.Enqueue(new CommonData(rnd.Next(0,stopVal)));
stopVal--;
}
else
callBack("Sender");

Console.WriteLine("Sender. in queue:{0}, the rest of notifications:{1}.",
cdQueue.Count, stopVal);
Thread.Sleep(100);
}
}//====================================
}


class StartClass
{
static Thread th0, th1;
static Queue NotificationQueue;
static string[] report = new string[2];
```

```
static void Main(string[] args)
{
StartClass.NotificationQueue = new Queue();
// Конструкторы классов Receiver и Sender несут дополнительную нагрузку.
// Они обеспечивают необходимыми значениями методы,
// выполняемые во вторичных потоках.
Sender sender;
// По окончании работы отправитель вызывает функцию-терминатор.
// Для этого используется специально определяемый и настраиваемый делегат.
sender = new Sender(ref NotificationQueue,
100,
new CallBackFromStartClass(StartClass.StopMain));

Receiver receiver;
// Выбрав всю очередь получатель вызывает функцию-терминатор.
receiver = new Receiver(ref NotificationQueue,
new CallBackFromStartClass(StartClass.StopMain));
// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart. Поэтому они не имеют параметров.
ThreadStart t0, t1;
t0 = new ThreadStart(sender.startSender);
t1 = new ThreadStart(receiver.startReceiver);

// Созданы вторичные потоки.
StartClass.th0 = new Thread(t0);
StartClass.th1 = new Thread(t1);

// Запущены вторичные потоки.
StartClass.th0.Start();
StartClass.th1.Start();

// Ещё раз о методе Join():
// Выполнение главного потока приостановлено до завершения
// выполнения вторичных потоков.
StartClass.th0.Join();
StartClass.th1.Join();

// Потому последнее слово остаётся за главным потоком приложения.
Console.WriteLine("Main(): " + report[0] + "..." + report[1] + "... Bye.");

}

// Функция-член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public static void StopMain(string param)
{
Console.WriteLine("StopMain: " + param);
// Остановка рабочих потоков. Её выполняет функция-член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЁ о вторичных потоках. Но выполняется она
// в ЧУЖИХ (вторичных) потоках.
if (param.Equals("Sender"))
{ report[0] = "Sender all did.";
StartClass.th0.Abort();
      }

if (param.Equals("Receiver"))
{     report[1] = "Receiver all did.";
StartClass.th1.Abort();
      }

// Этот оператор не выполняется! Поток, в котором выполняется
// метод-член класса StartClass StopMain() остановлен.
Console.WriteLine("StopMain(): bye.");
}
}
}
```

Состязание потоков

В ранее рассмотренном примере временные задержки при генерации и обработке данных подобраны таким образом, что обработчик завершает свою деятельность

последним. Таким образом, обеспечивается обработка ВСЕГО множества данных, размещённых в очереди. Разумеется, это идеальная ситуация. Изменение соответствующих значений может привести к тому, что обработчик данных опустошит очередь и завершит работу до того, как генератор данных разместит в очереди все данные.

Таким образом, результаты выполнения программы оказываются зависимыми от обстоятельств, никаким образом не связанных с поставленной задачей.

Подобная ситуация хорошо известна как “Race conditions” – состязание потоков и должна учитываться при реализации многопоточных приложений. Результаты работы потока-обработчика не должны зависеть от быстродействия потока-генератора.

Блокировки и тупики

Блокировка выполнения потока возникает при совместном использовании потоками нескольких ресурсов. В условиях, когда выполнение потоков явным образом не управляется, поток в нужный момент может не получить доступа к требуемому ресурсу, поскольку в данный момент этот ресурс используется другим потоком.

Тупик – взаимная блокировка потоков:

Поток A захватывает ресурс a и не может получить доступа к ресурсу b, который занят потоком B, и может быть им освобождён только по получению доступа к ресурсу a.

В приводимом ниже примере отсутствует ВЗАИМНАЯ блокировка потоков, поскольку описание изощрённых “самодельных” способов преодоления тупика не входит в задачу.

```
/// <summary>
/// Взаимодействующие потоки разделяют общие ресурсы - пару очередей.
/// Для успешной работы каждый поток должен последовательно получить доступ
/// к каждой из очередей. Из одной очереди взять в, другую положить.
/// Поток оказывается заблокирован, когда одна из очередей оказывается
/// занятой другим потоком.
/// </summary>


using System;
using System.Threading;


using System.Collections;
namespace CommunicatingThreadsQueue
{
/// <summary>
/// Модифицированный вариант очереди - очередь с флажком.
/// Захвативший эту очередь поток объявляет очередь «закрытой».
/// </summary>
public class myQueue: Queue
{
private bool isFree;
public bool IsFree
{
get
{
return isFree;
}
set
{
isFree = value;
}
}

public object myDequeue()
{
if (IsFree) {IsFree = false; return base.Dequeue();}
else return null;
}
```

```
public bool myEnqueue(object obj)
{
if (IsFree == true) {base.Enqueue(obj); return true;}
else return false;
}

}


public delegate void CallBackFromStartClass (string param);
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
private int iVal;
public int iValProp
{
get
{
return iVal;
}
set
{
iVal = value;
}
}

public CommonData(int key)
{
iVal = key;
}
}

// Классы Receiver и Sender: основа взаимодействующих потоков.
class Receiver
{
myQueue cdQueue0;
myQueue cdQueue1;
CallBackFromStartClass callBack;
int threadIndex;

// Конструктор...
public Receiver(ref myQueue queueKey0,
ref myQueue queueKey1,
CallBackFromStartClass cbKey,
int iKey)
{
threadIndex = iKey;
if (threadIndex == 0)
{
cdQueue0 = queueKey0;
cdQueue1 = queueKey1;
}
else
{
cdQueue1 = queueKey0;
cdQueue0 = queueKey1;
}


callBack = cbKey;
}

public void startReceiver()
{
DoIt();
}

// Тело рабочей функции...
public void DoIt()
{//====================================
```

```
CommonData cd = null;
while (true)
{//================================================================

if (cdQueue0.Count > 0)
{//=========================================================
while (true)
       {
cd = (CommonData)cdQueue0.myDequeue();
if (cd != null) break;
Console.WriteLine("»> Receiver{0} is blocked.", threadIndex);
       }


// Временная задержка "на обработку" полученного блока информации
// влияет на частоту и продолжительность блокировок.
Thread.Sleep(cd.iValProp*100);
// И это не ВЗАИМНАЯ блокировка потоков.
// "Обработали" блок - открыли очередь.
// И только потом предпринимается попытка
// обращения к очереди оппонента.
cdQueue0.IsFree = true;

//Записали результат во вторую очередь.
while (cdQueue1.myEnqueue(cd) == false)
       {
Console.WriteLine("<< Receiver{0} is blocked.", threadIndex);
       }


// А вот если пытаться освободить захваченную потоком очередь
// в этом месте – то взаимной блокировки потоков не избежать!
// cdQueue0.IsFree = true;

// Сообщили о состоянии очередей.
Console.WriteLine("Receiver{0}...{1}>{2}",threadIndex.ToString(),
cdQueue0.Count,cdQueue1.Count);

       }//=========================================================
else
       {//=========================================================
cdQueue0.IsFree = true;
callBack(string.Format("Receiver{0}",threadIndex.ToString()));
       }//=========================================================

}//================================================================
}//====================================
}

class Sender
{
Random rnd;
int stopVal;
myQueue cdQueue0;
myQueue cdQueue1;
CallBackFromStartClass callBack;

// Конструктор...
public Sender(ref myQueue queueKey0,
              ref myQueue queueKey1,
              int key,
              CallBackFromStartClass cbKey)
{
rnd = new Random(key);
stopVal = key;
cdQueue0 = queueKey0;
cdQueue1 = queueKey1;
callBack = cbKey;
}

public void startSender()
```

```
{
sendIt();
}


// Тело рабочей функции...
public void sendIt()
{//====================================

cdQueue0.IsFree = false;
cdQueue1.IsFree = false;

while (true)
{
if (stopVal > 0)
      {
// Размещение в очереди нового члена со случайными характеристиками.
cdQueue0.Enqueue(new CommonData(rnd.Next(0,stopVal)));
cdQueue1.Enqueue(new CommonData(rnd.Next(0,stopVal)));
stopVal--;
      }
else
      {
cdQueue0.IsFree = true;
cdQueue1.IsFree = true;
callBack("Sender");
      }

Console.WriteLine("Sender.  the  rest  of  notifications:{0},  notifications  in
queue:{1},{2}.", stopVal, cdQueue0.Count, cdQueue1.Count);
}


}//====================================
}


class StartClass
{

static Thread th0, th1, th2;
static myQueue NotificationQueue0;
static myQueue NotificationQueue1;
static string[] report = new string[3];

static void Main(string[] args)
{

StartClass.NotificationQueue0 = new myQueue();
StartClass.NotificationQueue1 = new myQueue();

// Конструкторы классов Receiver и Sender несут дополнительную нагрузку.
// Они обеспечивают необходимыми значениями методы,
// выполняемые во вторичных потоках.

Sender sender;
// По окончании работы отправитель вызывает функцию-терминатор.
// Для этого используется специально определяемый и настраиваемый делегат.
sender = new Sender(ref NotificationQueue0,
ref NotificationQueue1,
10, new CallBackFromStartClass(StartClass.StopMain));

Receiver receiver0;
// Выбрав всю очередь получатель вызывает функцию-терминатор.
receiver0 = new Receiver(ref NotificationQueue0,
ref NotificationQueue1,
new CallBackFromStartClass(StartClass.StopMain),0);

Receiver receiver1;
// Выбрав всю очередь получатель вызывает функцию-терминатор.
receiver1 = new Receiver(ref NotificationQueue0,
```

```
ref NotificationQueue1,
new CallBackFromStartClass(StartClass.StopMain),1);


// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart. Поэтому они не имеют параметров.
ThreadStart t0, t1, t2;
t0 = new ThreadStart(sender.startSender);
t1 = new ThreadStart(receiver0.startReceiver);
t2 = new ThreadStart(receiver1.startReceiver);

// Созданы вторичные потоки.
StartClass.th0 = new Thread(t0);
StartClass.th1 = new Thread(t1);
StartClass.th2 = new Thread(t2);

// Запущены вторичные потоки.
StartClass.th0.Start();
// Ещё раз о методе Join():
// Выполнение главного потока приостановлено до завершения
// выполнения вторичного потока закрузки очередей.
// Потоки получателей пока отдыхают.
StartClass.th0.Join();


// Отработал поток загрузчика.
// Очередь получателей.
StartClass.th1.Start();
StartClass.th2.Start();

// Метод Join():
// Выполнение главного потока опять остановлено
// приостановлено до завершения выполнения вторичных потоков.
StartClass.th1.Join();
StartClass.th2.Join();

// Последнее слово остаётся за главным потоком приложения.
// Но только после того как отработают терминаторы.
Console.WriteLine("Main(): "+report[0]+"..."+report[1]+"..."+report[2]+"... Bye.");

}

// Функция-член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public static void StopMain(string param)
{
Console.WriteLine(«StopMain: « + param);
// Остановка рабочих потоков. Её выполняет функция-член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЁ о вторичных потоках. Но выполняется она
// в ЧУЖИХ (вторичных) потоках.
if (param.Equals("Sender"))
{
report[0] = "Sender all did.";
StartClass.th0.Abort();
}

if (param.Equals("Receiver0"))
{
report[1] = "Receiver0 all did.";
StartClass.th1.Abort();
}

if (param.Equals("Receiver1"))
{
report[2] = "Receiver1 all did.";
StartClass.th2.Abort();
}


// Этот оператор не выполняется! Поток, в котором выполняется
// метод-член класса StartClass StopMain() остановлен.
```

```
Console.WriteLine("StopMain(): bye.");
}
}
}
```

Очереди. Основа интерфейса взаимодействия

Queue – класс, представляющий коллекцию объектов (objects), работающую по принципу "первый пришёл, первый ушёл" (first-in, first-out).

Stack - класс, представляющий коллекцию объектов (objects), работающую по принципу "последний пришёл, первый ушёл" (last-in, first-out).

Безопасность данных и критические секции кода

Некоторое значение, связанное с конкретным объектом, подвергается воздействию (изменению, преобразованию) со стороны потока. Это означает, что по отношению к объекту (значению объекта) применяется некоторая фиксированная последовательность операторов, в результате которой происходит КОРРЕКТНОЕ изменение состояния объекта или его значения.

В многопоточном приложении один и тот же объект может быть подвергнут одновременному "параллельному" воздействию со стороны нескольких потоков. Подобное воздействие представляет опасность для объекта и его значения, поскольку в этом случае порядок применения операторов из нескольких потоков (пусть даже и содержащих одни и те же операторы) неизбежно будет изменён.

В многопоточном программировании последовательности операторов, составляющих поток и при неконтролируемом доступе к объекту, возможно, приводящих к некорректному изменению состояния объекта, называются критическими секциями кода.

Управление последовательностью доступа потоков к объекту называют синхронизацией потоков.

Сам же объект называют объектом синхронизации.

Типичными средствами синхронизации потоков являются:

- критические секции,
- мониторы,
- мьютексы.

Пример организации многопоточного приложения

Приводимый ниже пример является многопоточным приложением, пара дополнительных потоков которого получают доступ к одному и тому же объекту (объекту синхронизации). Результаты воздействия образующих потоки операторов наглядно проявляются на экране консольного приложения.

```
using System;
using System.Threading;

namespace threads12
{
class TextPresentation
{
Mutex mutex = new Mutex(false);
public void showText(string text)
{
int i;
// Объект синхронизации в данном конкретном случае -
// представитель класса TextPresentation. Для его обозначения используется
// первичное выражение this._____________________________________________
//1. Блокировка кода монитором (начало)// Monitor.Enter(this);
//2. Критическая секция кода (начало)// lock(this)
//2. Критическая секция кода (начало)// {
mutex.WaitOne();//3.Блокировка кода мьютексом (начало)//
Console.WriteLine("\n" + (char)31 + (char)31 + (char)31 + (char)31);
for (i = 0; i < 250; i++)
      {
Console.Write(text);
      }
Console.WriteLine("\n" + (char)30 + (char)30 + (char)30 + (char)30);
mutex.Close();//3.Блокировка кода мьютексом (конец)//
```

```
//2. Критическая секция кода (конец)// }
//1. Блокировка кода монитором (конец)
// Monitor.Exit(this);
}
}

class threadsRunners
{

public static TextPresentation tp = new TextPresentation();
public static void Runner1()
{
Console.WriteLine("thread_1 run!");
Console.WriteLine("thread_1 - calling TextPresentation.showText");
tp.showText("*");
Console.WriteLine("thread_1 stop!");
}


public static void Runner2()
{
Console.WriteLine("thread_2 run!");
Console.WriteLine("thread_2 - calling TextPresentation.showText");
tp.showText("|");
Console.WriteLine("thread_2 stop!");
}

static void Main(string[] args)
{
ThreadStart runner1 = new ThreadStart(Runner1);
ThreadStart runner2 = new ThreadStart(Runner2);

Thread th1 = new Thread(runner1);
Thread th2 = new Thread(runner2);

th1.Start();
th2.Start();

}
}
}
```

Очередь как объект синхронизации

Достаточно сложное образование с множеством свойств и методов, предназначенное для упорядоченного размещения объектов (все дети класса object). Одновременное воздействие на очередь со стороны кода нескольких потоков представляет серьёзную опасность. И не только для самого объекта очереди в смысле возможного искажения сохраняемой в ней информации, сколько для самого приложения. Класс Queue взаимодействует с окружением через интерфейсы, неупорядоченное воздействие на объект очереди через эти интерфейсы возбуждает исключения. Очередь располагает специальными средствами, позволяющими защитить объект от неупорядоченного воздействия со стороны множества потоков. Назначение некоторых средств и их применение очевидно, как использовать другие средства – пока неясно (я пометил их вопросительным знаком).

В класс входят методы и свойства:

- множество вариантов конструкторов - Queue(…),
- методы, обеспечивающие загрузку и выгрузку данных (объектов-представителей классов-наследников класса object) – void Enqueue(object), object Dequeue(),
- методы поиска – bool Contains(object),
- методы предъявления – object Peek() (возвращает объект из начала очереди, не удаляя его из очереди),
- свойство Count – сохраняет информацию о количестве объектов в очереди,
- свойство SyncRoot – предоставляет ссылку на ОБЪЕКТ СИНХРОНИЗАЦИИ, который используется при синхронизации потоков многопоточного приложения,

- свойство IsSynchronized (?) – предоставляет информацию о том, синхронизирован ли объект для работы в многопоточном приложении. Это всего лишь значение объявленной в классе Queue булевской переменной,
- статический метод Synchronized (?) – создающий синхронизированную оболочку вокруг объекта очереди.

Примеры использования очередей в приложении приводятся ниже, а пока – вопросы, связанные с взаимодействием объекта очереди с потоками многопоточного приложения.

Перебор элементов очереди посредством оператора цикла foreach – самое “опасное” для очереди занятие в условиях многопоточного приложения. И причина всех неприятностей заключается во внутреннем устройстве и особенностях реализации цикла foreach, который при своём выполнении использует множество функций интерфейса очереди:

```
Queue myCollection = new Queue();
::::::::::::::::::::::::::::::::

// Перебор элементов очереди – критическая секция кода.
foreach ( Object item in myCollection )
 {
 ::::::::::::::::::::::::::::::::::::
 }
```

Возможный способ преодоления опасной ситуации – защита кода критической секцией. Суть защиты сводится к следующему. Поток, выполняющий собственный код при “подходе” к критической секции, связанной с конкретным объектом синхронизации, блокируется, если ЭТУ или ДРУГУЮ связанную с данным объектом синхронизации критическую секцию в данное время выполняет другой поток.

```
Queue myCollection = new Queue();
::::::::::::::::::::::::::::::::::::
lock( myCollection.SyncRoot )
{// Критическая секция, связанная с объетом
 // синхронизации, полученным от очереди
 // myCollection обозначена…
foreach ( Object item in myCollection )
 {
 ::::::::::::::::::::::::::::::::::::
 }
}
```

Пример синхронизации объекта очереди. Видно, как создавать синхронизированную оболочку вокруг несинхронизированной очереди, как узнавать о том, синхронизирована она или нет, НО ЗАЧЕМ ДЕЛАТЬ ЭТО – не сказано и не показано. Дело в том, что synchronized она или нет, а соответствующий код (критические секции кода) защищать всё равно надо!

```
using System;
using System.Collections;
public class SamplesQueue {

public static void Main() {
// Creates and initializes a new Queue.
Queue myQ = new Queue();
myQ.Enqueue( “The” );
myQ.Enqueue( “quick” );
myQ.Enqueue( “brown” );
myQ.Enqueue( “fox” );

// Creates a synchronized wrapper around the Queue.
Queue mySyncdQ = Queue.Synchronized( myQ );

// Displays the sychronization status of both Queues.
Console.WriteLine(“myQ is {0}.”,
myQ.IsSynchronized ? “synchronized” : “not synchronized” );
Console.WriteLine( “mySyncdQ is {0}.”,
mySyncdQ.IsSynchronized ? “synchronized” : “not synchronized” );
 }
```

```
}
/*
This code produces the following output.
myQ is not synchronized.
mySyncdQ is synchronized.
*/
```

Синхронизация работы потоков при работе с общими ресурсами

1. Организация критических секций

```
/// <summary>
/// Пара потоков «наперегонки» заполняет одну очередь.
/// Эти потоки синхронизуются посредством критичесих секций кода,
/// связанных с разделяемым ресурсом - общей очередью.
/// Третий поток читает из этой очереди.
/// Этот поток синхронизуется посредством монитора.
/// Методы Enter(...) и Exit(...) обеспечивают вход в критическую секцию кода,
/// связанную с конкретным разделяемым объектом и тем самым блокируют
/// одновременное выполнение какого-либо связанного с данным ресурсом кода в другом
потоке.
/// Толчея потоков сопровождается генерацией исключений.
/// </summary>

using System;
using System.Threading;
using System.Collections;

namespace CommunicatingThreadsQueue
{//-------------------------------------------------------------------------------

public delegate void CallBackFromStartClass (string param);

//================================================================================
// Данные. Предмет и основа взаимодействия двух потоков.
class CommonData
{
private int iVal;
public int iValProp
{
get{return iVal;}
set{iVal = value;}
}

public CommonData(int key)
{
iVal = key;
}
}
// Классы Sender и Receiver: основа взаимодействующих потоков.
class Sender
{
Queue cdQueue;
CallBackFromStartClass callBack;
int threadIndex;

// Конструктор...
public Sender(ref Queue queueKey, CallBackFromStartClass cbKey,   int iKey)
{
cdQueue = queueKey;
callBack = cbKey;
threadIndex = iKey;
}

public void startSender()
{
DoIt();
}
```

```
// Тело рабочей функции...
public void DoIt()
{//================================================================
Console.WriteLine("Sender{0}.DoIt()", threadIndex);
int i;

for (i = 0; i < 100; i++)
{//================================================================
try
{
lock(cdQueue.SyncRoot)
{//________________________________________________________________
Console.WriteLine("Sender{0}.", threadIndex);
cdQueue.Enqueue(new CommonData(i));
Console.WriteLine(">> Sender{0} >> {1}.", threadIndex,cdQueue.Count);
foreach(CommonData cd in cdQueue)
{
Console.Write("\rS{0}:{1} ", threadIndex, cd.iValProp);
}
Console.WriteLine("__ Sender{0} __", threadIndex);
}//________________________________________________________________
}
catch (ThreadAbortException e)
{
Console.WriteLine("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
Console.WriteLine("AbortException from Sender{0}.", threadIndex);
Console.WriteLine(e.ToString());
Console.WriteLine("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
}
catch (Exception e)
{
Console.WriteLine("____________________________________________________");
Console.WriteLine("Exception from Sender{0}.", threadIndex);
Console.WriteLine(e.ToString());
Console.WriteLine("____________________________________________________");
callBack(threadIndex.ToString());
}
}//================================================================
callBack(string.Format("Sender{0}",threadIndex.ToString()));
}//================================================================
}

class Receiver
{
Queue cdQueue;
CallBackFromStartClass callBack;
int threadIndex;

// Конструктор...
public Receiver(ref Queue queueKey, CallBackFromStartClass cbKey, int iKey)
{
cdQueue = queueKey;
callBack = cbKey;
threadIndex = iKey;
}

public void startReceiver()
{
DoIt();
}

// Тело рабочей функции...
public void DoIt()
{
Console.WriteLine("Receiver.DoIt()");
int i = 0;
CommonData cd;
while (i < 200)
{
try
{
```

```
Monitor.Enter(cdQueue.SyncRoot);
Console.WriteLine("Receiver.");
if (cdQueue.Count > 0)
{
cd = (CommonData)cdQueue.Dequeue();
Console.WriteLine("Receiver.current:{0},in queue:{1}.", cd.iValProp,cdQueue.Count);
foreach(CommonData cdW in cdQueue)
{
Console.Write("\rR:{0}.", cdW.iValProp);
}
Console.WriteLine("__ Receiver __");
i++;
}
Monitor.Exit(cdQueue.SyncRoot);
}
catch (ThreadAbortException e)
{
Console.WriteLine("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
Console.WriteLine("AbortException from Receiver.");
Console.WriteLine(e.ToString());
Console.WriteLine("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
}
catch (Exception e)
{
Console.WriteLine("_____________________________________________________");
Console.WriteLine("Exception from Receiver.");
Console.WriteLine(e.ToString());
Console.WriteLine("_____________________________________________________");
callBack(threadIndex.ToString());
}
}
callBack("Receiver");
}
}


class StartClass
{

Thread th0, th1, th2;
Queue queueX;
string[] report = new string[3];
ThreadStart t0, t1, t2;
Sender sender0;
Sender sender1;
Receiver receiver;

static void Main(string[] args)
{
StartClass sc = new StartClass();
sc.go();
}

void go()
{
// Простая очередь.
// queueX = new Queue();

// Синхронизированная очередь. Строится на основе простой очереди.
// Свойство синхронизированности дополнительно устанавливается в true
// посредством метода Synchronized.
queueX = Queue.Synchronized(new Queue());
// Но на самом деле никакой разницы между двумя версиями очереди
// (между несинхронизированной очередью и синхронизирванной оболочкой вокруг
// несинхронизированной очереди) мною замечено не было. И в том и в другом
// случае соответствующий код, который обеспечивает перебор элементов очереди
// должен быть закрыт посредством lock блока, с явным указанием ссылки на
// объект синхронизации.
sender0 = new Sender(ref queueX, new CallBackFromStartClass(StopMain), 0);
sender1 = new Sender(ref queueX, new CallBackFromStartClass(StopMain), 1);
receiver = new Receiver(ref queueX, new CallBackFromStartClass(StopMain), 2);
```

```
// Стартовые функции потоков должны соответствовать сигнатуре
// класса делегата ThreadStart. Поэтому они не имеют параметров.
t0 = new ThreadStart(sender0.startSender);
t1 = new ThreadStart(sender1.startSender);
t2 = new ThreadStart(receiver.startReceiver);

// Созданы вторичные потоки.
th0 = new Thread(t0);
th1 = new Thread(t1);
th2 = new Thread(t2);

th0.Start();
th1.Start();
th2.Start();

th0.Join();
th1.Join();
th2.Join();

Console.WriteLine
("Main(): " + report[0] + "..." + report[1] + "..." + report[2] + «... Bye.»);
}

// Функция-член класса StartClass выполняется во ВТОРИЧНОМ потоке!
public void StopMain(string param)
{
Console.WriteLine(«StopMain: « + param);
// Остановка рабочих потоков. Её выполняет функция-член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЁ о вторичных потоках. Но выполняется она
// в ЧУЖИХ (вторичных) потоках.
if (param.Equals("Sender0"))
{
report[0] = "Sender0 all did.";
th0.Abort();
}

if (param.Equals("Sender1"))
{
report[1] = "Sender1 all did.";
th1.Abort();
}

if (param.Equals("Receiver"))
{
report[2] = "Receiver all did.";
th2.Abort();
}

if (param.Equals("0"))
{
th1.Abort();
th2.Abort();
th0.Abort();
}

if (param.Equals("1"))
{
th0.Abort();
th2.Abort();
th1.Abort();
}

if (param.Equals("2"))
{
th0.Abort();
th1.Abort();
th2.Abort();
}

// Этот оператор не выполняется! Поток, в котором выполняется
```

```
// метод-член класса StartClass StopMain() остановлен.
Console.WriteLine("StopMain(): bye.");
}
}
}//-------------------------------------------------------------------------------
```

2. Специальные возможности мониторов

Класс Monitor управляет доступом к коду с использованием объекта синхронизации. Объект синхронизации предоставляет возможности для ограничения доступа к блоку кода, в общем случае обозначаемого как критическая секция.

Поток выполняет операторы. Выполнение оператора, обеспечивающего захват с помощью монитора объекта синхронизации, закрывает критическую секцию кода.

Другие потоки, выполняющие данную последовательность операторов, не могут продвинуться дальше оператора захвата монитором объекта синхронизации и переходят в состояние ожидания до тех пор, пока поток, захвативший с помощью монитора критическую секцию, кода не освободит её.

Таким образом, монитор гарантирует, что никакой другой поток не сможет обратиться к коду, выполняемому потоком, захватившим с помощью монитора и данного объекта синхронизации критическую секцию кода, пока она не будет освобождена, если только потоки не выполняют данную секцию кода с использованием другого объекта синхронизации.

Следующая таблица описывает действия, которые могут быть предприняты потоками при взаимодействии с монитором:

Действие	Описание
Enter, TryEnter	Закрытие секции с помощью объекта синхронизации. Это действие также обозначает начало критической секции. Никакие другие потоки не могут войти в заблокированную критическую секцию, если только они не используют другой объект синхронизации.
Exit	Освобождает блокировку критической секции кода. Также обозначает конец критической секции, связанной с данным объектом синхронизации.
Wait	Поток переходит в состояние ожидания, предоставляя тем самым другим потокам возможность выполнения других критических секций кода, связанных с данным объектом синхронизации. В состоянии ожидания поток остаётся до тех пор, пока на выходе из другой секции, связанной с данным объектом синхронизации, другой поток не выполнит на мониторе действия Pulse (PulseAll), которые означают изменение состояния объекта синхронизации и обеспечивают выход потока из состояния ожидания на входе в критическую секцию.
Pulse (signal), PulseAll	Посылает сигнал ожидающим потокам. Сигнал служит уведомлением ожидающему потоку, что состояние объекта синхронизации изменилось, и что владелец объекта готов его освободить. Находящийся в состоянии ожидания поток фактически находится в очереди для получения доступа к объекту синхронизации.

Enter и Exit иетоды используются для обозначения начала или конца критической секции. Если критическая секция представляет собой "непрерывное" множество инструкций, закрытие кода посредством метода Enter гарантирует, что только один единичный поток сможет выполнять код, закрытый объектом синхронизации.

Рекомендуется размещать эти инструкции в try block и помещать Exit instruction в finally блоке.

Все эти возможности обычно используются для синхронизации доступа к статическим и нестатическим методам класса. Нестатический метод блокируется посредством объекта синхронизации. Статический объект блокируется с использованием типа (класса, членом которого данный метод является).

```
/// <summary>
/// Синхронизация потоков с использованием класса монитора.
/// Синхронизация потоков с использованием класса монитора.
```

```
/// Монитор защищает очередь от параллельного вторжения со стороны
/// взаимодействующих потоков из разных фрагментов кода.
/// Однако монитор не может защитить потоки от взаимной блокировки.
/// Поток просыпается, делает свою работу, будит конкурента, засыпает сам.
/// К тому моменту, как поток будит конкурента, конкурент должен спать.
/// Активизация незаснувшего потока не имеет никаких последствий.
/// Если работающий поток разбудит не успевший заснуть поток - возникает
/// тупиковая ситуация. Оба потока оказываются погруженными в сон.
/// В этом случае имеет смысл использовать перегруженный вариант метода
/// Wait - с указанием временного интервала.
/// </summary>

using System;
using System.Threading;
using System.Collections;

namespace MonitorCS1
{
class MonitorApplication
{
const int MAX_LOOP_TIME = 100;
Queue xQueue;


public MonitorApplication()
{
xQueue = new Queue();
}


public void FirstThread()
{
int counter = 0;
while(counter < MAX_LOOP_TIME)
{
Console.WriteLine("Thread_1___");
counter++;
Console.WriteLine("Thread_1...{0}", counter);


try
{
//Push element.
xQueue.Enqueue(counter);

foreach(int ctr in xQueue)
{
Console.WriteLine(":::Thread_1:::{0}", ctr);
}
}
catch (Exception ex)
{
Console.WriteLine(ex.ToString());

}

//Release the waiting thread. Применяется к конкурирующему потоку.
lock(xQueue){Monitor.Pulse(xQueue);}
Console.WriteLine(">1 Wait<");
//Wait, if the queue is busy. Применяется к текущему потоку.
// Собственное погружение в состояние ожидания.
lock(xQueue){Monitor.Wait(xQueue,1000);}
Console.WriteLine("!1 Work!");

}
Console.WriteLine("*****1 Finish*****");
lock(xQueue) {Monitor.Pulse(xQueue);}

}

public void SecondThread()
```

```
{
int counter = 0;
while(counter < MAX_LOOP_TIME)
{
//Release the waiting thread. Применяется к конкурирующему потоку.
lock(xQueue){Monitor.Pulse(xQueue);}
Console.WriteLine(«>2 Wait<»);
// Собственное погружение в состояние ожидания.
lock(xQueue){Monitor.Wait(xQueue,1000);}

Console.WriteLine("!2 Work!");
Console.WriteLine("Thread_2___");

try
{
foreach(int ctr in xQueue)
{
Console.WriteLine(":::Thread_2:::{0}", ctr);
}

//Pop element.
counter = (int)xQueue.Dequeue();
}
catch (Exception ex)
{

counter = MAX_LOOP_TIME;
Console.WriteLine(ex.ToString());
}

Console.WriteLine("Thread_2...{0}",counter);

}
Console.WriteLine("*****2 Finish*****");
lock(xQueue) {Monitor.Pulse(xQueue);}
}

static void Main(string[] args)
{
// Create the MonitorApplication object.
MonitorApplication test = new MonitorApplication();
Thread tFirst = new Thread(new ThreadStart(test.FirstThread));
// Вторичные потоки созданы!
Thread tSecond = new Thread(new ThreadStart(test.SecondThread));
//Start threads.
tFirst.Start();
tSecond.Start();
// Ждать завершения выполнения вторичных потоков.
tFirst.Join();
tSecond.Join();
}
}
}
```

Рекомендации по недопущению блокировок потоков

- Соблюдать определённый порядок при выделении ресурсов.
- При освобождении выделенных ресурсов придерживаться обратного (reverse) порядка.
- Минимизировать время неопределённого ожидания выделяемого ресурса.
- Не захватывать ресурсы без необходимости и при первой возможности освобождать захваченные ресурсы.
- Захватывать ресурс только в случае крайней необходимости.
- В случае если ресурс не удаётся захватить, повторную попытку его захвата производить только после освобождения ранее захваченных ресурсов.
- Максимально упрощать структуру задачи, решение которой требует захвата ресурсов. Чем проще задача – тем на меньший период времени захватывается ресурс.

Форма

Класс Form

Форма является представлением окна, которое появляется в Windows приложении. Это класс, который можно использовать как основу для создания различных вариантов окошек: стандартных, инструментальных, всплывающих, borderless, диалоговых, and floating windows.

Создаётся "окно", а класс называется формой – поскольку в окошке можно разместить элементы управления, обеспечивающие интерактивное взаимодействие приложения и пользователя (заполните форму, please).

Известна особая категория форм – формы (MDI) – формы с многодокументным интерфейсом (the multiple document interface).

Эти формы могут содержать другие формы, которые в этом случае называются MDI child forms. MDI форма создаётся после установки в true свойства IsMdiContainer.

Форма это класс, включающий свойства, методы и события.

Используя доступные в классе Формы свойства, можно определять внешний вид, размер, цвет, и особенности управления создаваемого окна или диалога.

Свойство Text позволяет специфицировать надпись of the window in the title bar.

Свойства Size и DesktopLocation позволяют определять размеры и положение окна в момент его появления на экране монитора.

Свойство ForeColor позволяет изменить предопределённый foreground color всех элементов управления, placed on the form.

Свойства FormBorderStyle, MinimizeBox, and MaximizeBox позволяют to control whether the form can be minimized, maximized, or resized во время выполнения приложения.

Методы класса обеспечивают управление формой.

Например, метод ShowDialog обеспечивает представление формы как модального dialog box.

Метод Show показывает форму как немодальный dialog box.

Метод SetDesktopLocation используется для позиционирования формы на поверхности desktop.

Форма предназначена для реализации интерфейса пользователя приложения. Содержит большой набор свойств, методов, событий для реализации различных вариантов пользовательского интерфейса. Является окном и наследует классу Control.

Это означает, что объект-представитель класса Form поддерживает механизмы управления, реализованные на основе обмена сообщениями Windows.

Структура сообщений и стандартные механизмы управления здесь не рассматриваются. Достаточно знать, что класс формы содержит объявление множества событий, для которых на основе стандартного интерфейса (сигнатуры) могут быть реализованы и легко подключены функции обработки событий.

Форма может использоваться как the starting class в приложении. При этом класс формы должен содержать точку входа – статический метод Main. В теле этого метода обычно размещается код, обеспечивающий создание и формирование внешнего вида формы.

Обычно заготовка формы "пишется" мастером. Пример кода простой заготовки окна прилагается. Форма настолько, что после некоторой медитации может быть воспроизведена вручную.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace FormByWizard
{
/// <summary>
/// Summary description for Form1.
/// </summary>
```

```
public class Form1 : System.Windows.Forms.Form
{
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
//
// Required for Windows Form Designer support
//
InitializeComponent();
//
// TODO: Add any constructor code after InitializeComponent call
//
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

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

Главная проблема – метод InitializeComponent(). Это зона ответственности мастера приложений и не рекомендуется в этом методе делать что-либо самостоятельно. Во избежание потери всего того, что там может быть построено, поскольку мастер приложения может изменить содержимое тела метода в соответствии с изменениями внешнего вида приложения.

В этом классе куча всяких методов. Интересно посмотреть на внушительный список обработчиков событий.

Важна строчка в теле функции Main

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

В принципе эта строчка и отвечает за создание, “запуск” в потоке приложения и возможное появление на экране дисплея формы.

Форма: управление и события жизненного цикла

Управление жизненным циклом и отображением форм осуществляется следующими методами:

```
Form.Show(),
Form.ShowDialog(),
Form.Activate(),
Form.Hide(),
Form.Close().
```

Имеет смысл рассмотреть события, связанные с созданием, функционированием и уничтожением формы. К их числу относятся:

- Load. Генерируется ОДИН РАЗ, непосредственно после первого вызова метода Form.Show() или Form.ShowDialog(). Это событие можно использовать для первоначальной инициализации переменных и для подготовки формы к работе. Сколько в приложении форм, столько раз будет генерироваться это событие. Назначение максимальных и минимальных размеров формы – для этого более подходящего места, нежели обработчик события OnLoad не найти!
- Activated. Многократно генерируется в течение жизни формы. Когда Windows активизирует форму. Связано с получением и потерей фокуса. Все необходимые мероприятия выполняются здесь. Методы Form.Show(), Form.ShowDialog(), Form.Activate() (передача фокуса, реализованная программно!) способствуют этому. Передача фокуса элементу управления (кнопка – это тоже окно) сопровождается автоматическим изменением цвета элемента управления.
- VisibleChanged. Генерируется всякий раз при изменении свойства Visible формы. Когда она становится видимой или невидимой. Событию способствуют методы Form.Show(), Form.ShowDialog(), Form.Hide(), Form.Close().
- Deactivated. Возникает при потере фокуса формой в результате взаимодействия с пользовательским интерфейсом либо в результате вызова методов Form.Hide() или Form.Cloze() – но только для активной формы. Если закрывать неактивную форму, событие не произойдёт! Сказано, что Activated и Deactivated возбуждаются только при перемещении фокуса в пределах приложения. При переключении с одного приложения на другое эти события не генерируются.
- Closing. Непосредственно перед закрытием формы. В этот момент процесс закрытия формы может быть приостановлен и вообще отменён, чему способствует размещаемый в теле обработчика события следующий программный код: e.Cancel = true; // e - событие типа CancelEventArgs.
- Closed. Уже после закрытия формы. Назад пути нет. В обработчике этого события размещается любой код для “очистки” после закрытия формы.

Форма: контейнер как элемент управления

Container controls – место для группировки элементов пользовательского интерфейса. И это не простая группировка! Контейнер может управлять доступом к размещаемым не его поверхности элементам управления. Это обеспечивается за счёт свойства Enable, которое будучи установленным в false, закрывает доступ к размещаемым в контейнере компонентам. “Поверхность” контейнера у некоторых его разновидностей не ограничивается непосредственно видимой областью. Могут быть полосы прокрутки.

Каждый контейнер поддерживает набор элементов, который состоит из всех вложенных в него элементов управления. У контейнера имеются свойства Count, которое возвращает количество элементов управления, свойство [ИНДЕКСАТОР], методы Add и Remove.

К числу контейнеров относятся:

- Form… Без комментариев!
- Panel. Элемент управления, содержащий другие элеменгты управления. You can use a Panel to для группировки множества элементов управления, как например, группа of RadioButton controls. The Panel control is displayed по умолчанию БЕЗ всяческих рамок. Нет у ПАНЕЛИ и заголовка. Рамки можно заказать. You can

provide a standard or three-dimensional border using the BorderStyle property to distinguish the area of the panel from other areas on the form. Panel является производным классом от the ScrollableControl class, а это значит, что можно заказать и полосы прокрутки, путём изменения значения соответствующего свойства. И всё, что располагается на панели за пределами видимости, при условии установки свойства AutoScroll в true, any controls located within the Panel (but outside of its visible region), can be scrolled to with the scroll bars provided.

- GroupBox. Предполагается для размещения радиокнопок и прочих переключателей. Рисует рамочку вокруг элементов управления, которые группируются по какому-либо признаку. В отличие от панели имеет заголовок, который, впрочем, можно не использовать. Не имеет полос прокрутки.
- TabControl. Содержит tab pages, которые представлены TabPage объектами, которые могут добавляться через свойство Controls. Каждая TabPage-страница подобна листку записной книжки: страница с закладкой. И в зависимости от того, подцеплены ли tab pages к объекту-представителю класса TabControl, будут генерироваться следующие события: Control.Click, Control.DoubleClick, Control.MouseDown, Control.MouseUp, Control.MouseHover, Control.MouseEnter, Control.MouseLeave and Control.MouseMove. If there is at least one TabPage in the collection, and the user interacts with the tab control’s header (where the TabPage names appear), the TabControl raises the appropriate event. However, if the user interaction is within the ClientRectangle of the tab page, the TabPage raises the appropriate event. От взаимодействия со страницей генерируются одни события, от взаимодействия с корешком – другие.

Разница между элементами управления и компонентами.

Элемент управления имеет видимое представление и непосредственно используется в процессе управления формой. Компоненты (таймер, провайдеры дополнительных свойств) видимого представления не имеют. И поэтому в управлении формой участвуют опосредованно.

Свойства элементов управления. Anchor и Dock

Средства, которые обеспечивают стыковку и фиксацию элементов управления в условиях изменяемых размеров контейнера.

Anchor позволяет автоматически выдерживать постоянное расстояние между границами элемента управления, для которого оно определено, и границами контейнера. И это обеспечивается за счёт автоматического изменения размеров “зацепленного” элемента управления вслед за изменениями размеров контейнера. Таким образом обеспечивается реакция элемента управления на изменения контейнера.

Dock – стыковка. Прикрепление элемента управления к границе контейнера. Dock – свойство элемента управления, а не контейнера. Центральный прямоугольник приводит к тому, что элемент управления заполнит всю поверхность контейнера.

Extender providers. Провайдеры дополнительных свойств

An extender provider - компонент, который предоставляет свойства другим компонентам. Это делается с целью расширения множества свойств (элементов управления).

```
ToolTipProvider,
HelpProvider,
ErrorProvider.
```

Например, когда a ToolTip Component (Windows Forms) компонент добавляется к форме, все остальные компоненты, размещённые на этой форме, получают новое свойство, которое даже можно просматривать и устанавливать (редактировать) в окне Properties непосредственно в процессе разработки формы. И это новое свойство, называемое ToolTip, вызывается для каждого элемента управления данной формы.

Не надо обольщаться. Множество свойств элементов управления не изменилось. Дополнительное свойство предоставляется ВСЕМ элементам отдельным компонентом.

На этапе разработки, это свойство появляется в Properties window for the component that is being modified.

Однако при выполнении (at run time), это свойство не может быть доступно (accessed) непосредственно через данный конкретный элемент управления.

В следующем примере кода форма была построена с кнопочкой, которая была названа MyButton и элементол управления ToolTip, названным MyToolTip, which provides a ToolTip property.

```
// И вот так просто и наивно значение свойства кнопочки не получить,
// поскольку это свойство не родное!
// This is an example of code that is NOT CORRECT!
string myString;
myString = MyButton.ToolTip;
```

Подобное обращение пиведёт к ошибке ещё на стадии компиляции. А вот как надо обращаться к этому самому дополнительному свойству, установленному для кнопочки (просто форменное надувательство):

```
string myString;
myString = MyToolTip.GetToolTip(MyButton);
```

Провайдеры дополнительных свойств являются классами, а это означает, что у них имеются собственные свойства и методы.

Пример…

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace Rolls01
{
/// <summary>
/// Summary description for numPointsForm.
/// </summary>
public class numPointsForm : System.Windows.Forms.Form
{
 Form1 f1;
 int nPoints;
 private System.Windows.Forms.TextBox numBox;
 private System.Windows.Forms.Button button1;
 private System.ComponentModel.IContainer components;

 private System.Windows.Forms.ToolTip toolTip;
 // Ссылка на объект-представитель класса ErrorProvider
 private System.Windows.Forms.ErrorProvider errorProvider;

 private string[] errMess;

 public numPointsForm(Form1 f1Key)
 {
  f1 = f1Key;
  InitializeComponent();
  errMess = new string[] {
                             "Больше двух тараканов!",
                             "Целое и больше двух!"
                         };
 }

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
 if( disposing )
 {
 if(components != null)
```

```
 {
 components.Dispose();
 }
}
 base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
 this.components = new System.ComponentModel.Container();
 this.numBox = new System.Windows.Forms.TextBox();
 this.button1 = new System.Windows.Forms.Button();
 this.toolTip = new System.Windows.Forms.ToolTip(this.components);
 this.SuspendLayout();
 //
 // numBox
 //
 this.numBox.Location = new System.Drawing.Point(8, 11);
 this.numBox.Name = "numBox";
 this.numBox.Size = new System.Drawing.Size(184, 20);
 this.numBox.TabIndex = 0;
 this.numBox.Text = "";
 this.numBox.Validating +=
 new System.ComponentModel.CancelEventHandler(this.numBox_Validating);
 //
 // button1
 //
 this.button1.Location = new System.Drawing.Point(208, 8);
 this.button1.Name = "button1";
 this.button1.TabIndex = 1;
 this.button1.Text = "OK";
 this.button1.Click += new System.EventHandler(this.button1_Click);
 //
 // numPointsForm
 //
 this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(288, 45);
this.Controls.Add(this.button1);
this.Controls.Add(this.numBox);
this.MaximizeBox = false;
this.MinimizeBox = false;
this.Name = "numPointsForm";
this.Text = "numPointsForm";
this.Load += new System.EventHandler(this.numPointsForm_Load);
this.ResumeLayout(false);
}
#endregion

private void numBox_Validating
             (object sender, System.ComponentModel.CancelEventArgs e)
{
int x;
try
{
 x = int.Parse(numBox.Text);
 if (x <= 1)
 {
 numBox.Text = nPoints.ToString();
 e.Cancel = true;
 errorProvider.SetError(numBox, this.errMess[0]);
 }
else
{
 nPoints = x;
}
}
```

```
catch (Exception e2)
{
numBox.Text = nPoints.ToString();
e.Cancel = true;
// Обращение к Error провайдеру.
errorProvider.SetError(numBox, this.errMess[1]);
}
}

private void numPointsForm_Load(object sender, System.EventArgs e)
{
nPoints = f1.nPoints;
numBox.Text = nPoints.ToString();
toolTip.SetToolTip(numBox,"Количество тараканчиков. Больше двух.");
errorProvider = new ErrorProvider();
}

private void button1_Click(object sender, System.EventArgs e)
{
f1.nPoints = nPoints;
this.Close();
}
}
}
```

Validating и Validated элементов управления

Предполагается, что свойство CausesValidation элементов управления, для которых будет проводиться проверка, установлено в true. Это позволяет отработать обработчику события Validating, которое возникает в момент потери фокуса элементом управления. У обработчика события Validating имеется аргумент объект-представитель класса CancelEventArgs обычно с именем e. У него есть поле Cancel, которое в случае ошибки можно установить в true, что приводит к возвращению фокуса.

Validated генерируется после Validating. Разница между ними заключается в следующем.

Validating активизируется для данного элемента управления непосредственно после потери фокуса. Перехват этого события позволяет, например, оперативно проверить правильность заполнения данного поля ввода и в случае некорректного заполнения вернуть фокус в это поле. При этом можно предпринять некоторые шаги по коррекции неправильного значения. Например, если в поле ввода должна располагаться последовательность символов, преобразуемая к целочисленному значению, а туда была записана “qwerty”, то можно восстановить последнее корректрое значение или вписать туда строку “0”.

Validated активизируется при попытке закрытия формы. В обработчике этого события обычно располагается код, который позволяет осуществить проверку корректности заполнения всей формы в целом. Например, отсутствие значений в текстовых полях, которые обязательно должны быть заполнены.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace PropertiesProviders
{
public class Form1 : System.Windows.Forms.Form
 {
private System.Windows.Forms.Button BigButton;
private System.Windows.Forms.ToolTip toolTip01;
private System.ComponentModel.IContainer components;

private System.Windows.Forms.Button RedButton;
private System.Windows.Forms.ErrorProvider errorProvider1;

int nTip = 0;
string[] Tips = {
```

```
"Не торопись...",
"Попробуй ещё раз... " ,
"Зри в корень... "
       };

int nErr = 0;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.HelpProvider helpProvider1;
string[] ErrMess = {
"Не надо было этого делать... ",
"Какого хрена... ",
"Ну всё... ",
 ""
 };


public Form1()
{
InitializeComponent();
}

protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
this.components = new System.ComponentModel.Container();
this.BigButton = new System.Windows.Forms.Button();
this.toolTip01 = new System.Windows.Forms.ToolTip(this.components);
this.RedButton = new System.Windows.Forms.Button();
this.errorProvider1 = new System.Windows.Forms.ErrorProvider();
this.textBox1 = new System.Windows.Forms.TextBox();
this.helpProvider1 = new System.Windows.Forms.HelpProvider();
this.SuspendLayout();
 //
// BigButton
 //
this.BigButton.Location = new System.Drawing.Point(8, 40);
this.BigButton.Name = "BigButton";
this.BigButton.TabIndex = 0;
this.BigButton.Text = "BigButton";
this.toolTip01.SetToolTip(this.BigButton, "Жми на эту кнопку, дружок!");
this.BigButton.Click += new System.EventHandler(this.BigButton_Click);
 //
// RedButton
 //
this.RedButton.Location = new System.Drawing.Point(112, 40);
this.RedButton.Name = "RedButton";
this.RedButton.Size = new System.Drawing.Size(80, 23);
this.RedButton.TabIndex = 1;
this.RedButton.Text = "RedButton";
this.toolTip01.SetToolTip(this.RedButton, "А на эту кнопку не надо нажимать!");
this.RedButton.Click += new System.EventHandler(this.RedButton_Click);
 //
// errorProvider1
 //
this.errorProvider1.ContainerControl = this;
```

```
 //
// textBox1
 //
this.helpProvider1.SetHelpString(this.textBox1, "int values only...");
this.textBox1.Location = new System.Drawing.Point(272, 40);
this.textBox1.Name = "textBox1";
this.helpProvider1.SetShowHelp(this.textBox1, true);
this.textBox1.TabIndex = 2;
this.textBox1.Text = "";
this.textBox1.Validating +=
new System.ComponentModel.CancelEventHandler(this.textBox1_Validating);
 //
// Form1
 //
this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);
this.ClientSize = new System.Drawing.Size(536, 168);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.RedButton);
this.Controls.Add(this.BigButton);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
 }
#endregion
[STAThread]
static void Main()
 {
Application.Run(new Form1());
 }

private void BigButton_Click(object sender, System.EventArgs e)
 {
this.toolTip01.SetToolTip(this.BigButton, Tips[nTip]);
if (nTip < 2) nTip++;
else nTip = 0;
 }

private void RedButton_Click(object sender, System.EventArgs e)
 {
errorProvider1.SetError(RedButton, ErrMess[nErr]);
if (nErr < 3) nErr++;
else nErr=0;
 }

private void textBox1_Validating
(object sender, System.ComponentModel.CancelEventArgs e)
 {
int val = 0;
try
 {
if (this.textBox1.Text != "")
val = int.Parse(textBox1.Text);
errorProvider1.SetError(textBox1, "");
 }
catch
 {
e.Cancel = true;
errorProvider1.SetError(textBox1, "Целое, блин...");
 }
}
}
}
```

Управление посредством сообщений

Когда ещё при программировании Windows-приложений не прибегали к помощи MFC, когда ещё не было карт сообщений, оконная процедура WinMain определялась явным образом, и в ней содержался ЦИКЛ, который обеспечивал непрерывное "прослушивание" системы и интерпретацию перехватываемых сообщений.

И вся работа приложения фактически сводилась к тому, что между распознанным в этом цикле сообщением и соответствующей функцией приложения устанавливалось посредством тривиального оператора выбора взаимнооднозначное соответствие.

Естественно, при этом производился вызов соответствующей функции с возможной передачей этой функции параметров. При этом попадание в этот цикл обеспечивалось достаточно тривиальной стандартной последовательностью операторов. С появлением MFC этот цикл достаточно простой стандартной серией макроопределений скрывался от разработчика приложения за картой сообщений.

Обсуждение реальных механизмов работы приложения рядовыми программистами не предполагалось.

Обеспечить реакцию приложения на одно из множества стандартных сообщений (событие), приходящих от операционной системы можно было путём простой модификации соответствующего макроопределения, добавляя к этому макроопределению указатель (ссылку, делегат, событие) на функцию-обработчик события.

Стандартный делегат

Существует соглашение, по которому обработчик событий в .NET Framework не возвращает значений (тип возвращаемого значения void) и принимает два параметра.

Первый параметр – ссылка на источник события (объект-издатель), второй параметр – ссылка на объект, производный от класса EventArgs. Сам по себе базовый класс НЕ НЕСЁТ никакой “полезной” информации о конкретных событиях. Назначение данного класса заключается в поддержке универсального стандартного механизма реализации событий (в том числе и передачи параметров). Забота о представлении информации о событии возлагается на разработчика производного класса.

Делегат EventHandler

Представляет метод со стандартной сигнатурой, предназначенный для обработки события, не содержащего дополнительной информации.

Объявляется следующим образом:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Параметры

```
object sender //Представляет ссылку на объект-источник события.
EventArgs e   // Таким образом кодируется информация о событии.
```

Замечание

Модель событий в .NET Framework основывается на механизме ссылок на функции (events – разновидности класса-делегата), которые обеспечивают стандартную стыковку события с обработчиком. Для возбуждения события необходимы два элемента:

Класс-носитель информации о событии. Должен наследовать от базового класса EventArgs.

Делегат, настроенный на метод, обеспечивающий реакцию на данное событие. Когда создаётся делегат-представитель класса-делегата EventHandler, прежде всего, определяется соответствующий метод, выполнение которого обеспечивает реакцию на событие.

Таким образом, для реализации перехвата события достаточно использовать:

1. для идентификации события. Базовый класс EventArgs, если уведомление о произошедшем событии не связано с генерацией дополнительной информации или производный от данного класса класс, если необходимо передавать дополнительную информацию, связанную с событием,
2. предопределённый класс EventHandler для реализации ссылки на метод-обработчик события.

Пример

```
using System;
namespace Events00
{
```

```
// Однопоточное приложение, в котором для реализации механизма
// реакции на события используется стандартный класс-делегат
// System.EventHandler.
// Объявляется класс события, производный от System.EventArgs.
//=====================================================================
class EventPosition:System.EventArgs
{
// Дополнительная информация о событии.
public int X;

// Конструктор...
public EventPosition(int key)
{
X = key;
}

}
//=====================================================================

/// <summary>
///Базовый класс действующих в приложении объектов.
///Содержит ВСЁ необходимое для того, чтобы объекты производных классов
/// могли адекватно реагировать на заложенные в базовом классе события.
/// </summary>
class BaseClass
{
// Ссылка на это событие идентифицируется как xEvent.
// Это «стандартное» событие.
// xEvent стыкуется со стандартным классом-делегатом System.EventHandler.
public static event System.EventHandler xEvent;

// Статическая переменная-носитель дополнительной информации.
static int xPosition = 0;
// Статическая функция. Это модель процесса непрерывного сканирования.
// Аналог цикла обработки сообщений приложения.
public static void XScaner()
{
while (true)
{//====================================================================
while(true)
{//====================================================================
// Источником события является вводимая с клавиатуры
// последовательность символов, соответствующая целочисленному
// значению 50. При получении этого значения просходит уведомление
// подписанных на событие объектов.============================
try
 {
Console.WriteLine("new xPosition, please: ");
xPosition = Int32.Parse(Console.ReadLine());
break;
 }
catch
 {
Console.WriteLine("Incorrect value of xPosition!");
 }

}//====================================================================
if (xPosition < 0) break; // Отрицательные значения являются сигналом
// к прекращению выполнения , а при получении 50 - возбуждается событие!
if (xPosition == 50) xEvent(new BaseClass(), new EventPosition(xPosition));
}//====================================================================
}
}

//=====================================================================
// Важное обстоятельство! В этом приложении событие возбуждается
// ПРЕДКОМ, а реагирует на него объект-представитель класса ПОТОМКА!
//=====================================================================

/// <summary>
/// Объявление первого производного класса.
```

```
/// Надо сделать дополнительные усилия для того чтобы объекты этого класса
/// стали бы реагировать на события.
/// </summary>
class ClassONE:BaseClass
{
public void MyFun(object obj, System.EventArgs ev)
{
Console.Write("{0} - ",this.ToString());
Console.WriteLine
("{0}:YES! {1}",((BaseClass)obj).ToString(),((EventPosition)ev).X.ToString());
}
}

/// <summary>
/// Второй класс чуть сложнее. Снабжён конструктором, который позволяет классу
/// самостоятельно «поднисаться» на «интересующее» его событие.
/// </summary>
class ClassTWO:BaseClass
{
public ClassTWO()
{
BaseClass.xEvent += new System.EventHandler(this.MyFun);
}

public void MyFun(object obj, System.EventArgs ev)
{
Console.Write("{0} - ",this.ToString());
Console.WriteLine
("{0}:YES! {1}", ((BaseClass)obj).ToString(), ((EventPosition)ev).X.ToString());
}
}


//=============================================================================
class mainClass
{
static void Main(string[] args)
{
Console.WriteLine(«0_____»);
// Создали первый объект и подписали его на получение события.
ClassONE one = new ClassONE();
BaseClass.xEvent += new System.EventHandler(one.MyFun);
// Второй объект подписался сам.
ClassTWO two = new ClassTWO();
// Запустили цикл прослушивания базового класса.
BaseClass.XScaner();
// При получении отрицательного значения цикл обработки
// сообщений прерывается.
Console.WriteLine(«1_____»);
// Объект - представитель класса ClassONE перестаёт
// получать уведоомление о событии.
BaseClass.xEvent -= new System.EventHandler(one.MyFun);
// После чего повторно запусткается цикл прослушивания,
// который прекращает выполняться после повторного
// получения отрицательного значения.
BaseClass.XScaner();
Console.WriteLine("2_____");
}
}
//=============================================================================
}
```

Класс Application

public sealed class Application – класс, закрытый для наследования.

Предоставляет СТАТИЧЕСКИЕ (и только статические!) методы и свойства для общего управления приложением. редоставляет статические методы и свойства для управления приложением, в том числе:

- методы для запуска и остановки приложения,
- методы для запуска и остановки потоков в рамках приложения,

- методы для обработки сообщений Windows,
- свойства для получения сведений о приложении.

Открытые свойства

AllowQuit	Получает значение, определяющее, может ли вызывающий объект выйти из этого приложения.
CommonAppDataPath	Получает путь для данных приложения, являющихся общими для всех пользователей.
CommonAppDataRegistry	Получает раздел реестра для данных приложения, являющихся общими для всех пользователей.
CompanyName	Получает название компании, связанное с приложением.
CurrentCulture	Получает или задает данные о культурной среде для текущего потока.
CurrentInputLanguage	Получает или задает текущий язык ввода для текущего потока.
ExecutablePath	Получает путь для исполняемого файла, запустившего приложение, включая исполняемое имя.
LocalUserAppDataPath	Получает путь для данных приложения местного, не перемещающегося, пользователя.
MessageLoop	Получает значение, указывающее, существует ли цикл обработки сообщений в данном потоке.
ProductName	Получает название продукта, связанное с данным приложением.
ProductVersion	Получает версию продукта, связанную с данным приложением.
SafeTopLevelCaptionFormat	Получает или задает строку формата для использования в заголовках окон верхнего уровня, когда они отображаются с предупреждающим объявлением.
StartupPath	Получает путь для исполняемого файла, запустившего приложение, не включая исполняемое имя.
UserAppDataPath	Получает путь для данных приложения пользователя.
UserAppDataRegistry	Получает раздел реестра для данных приложения пользователя.

Открытые методы

AddMessageFilter	Добавляет фильтр сообщений для контроля за сообщениями Windows во время их направления к местам назначения.
DoEvents	Обрабатывает все сообщения Windows, в данный момент находящиеся в очереди сообщений.
EnableVisualStyles	Включите визуальные стили Windows XP для приложения.
Exit	Сообщает всем прокачкам сообщений, что следует завершить работу, а после обработки сообщений закрывает все окна приложения.
ExitThread	Выходит из цикла обработки сообщений в текущем потоке и закрывает все окна в потоке.
OleRequired	Инициализирует OLE в текущем потоке.
OnThreadException	Вызывает событие ThreadException.
RemoveMessageFilter	Удаляет фильтр сообщений из прокачки сообщений приложения.
Run	Перегружен. Запускает стандартный цикл обработки сообщений приложения в текущем потоке.

Открытые события

ApplicationExit	Происходит при закрытии приложения.
Idle	Происходит, когда приложение заканчивает обработку и собирается перейти в состояние незанятости.
ThreadException	Возникает при выдаче не перехваченного исключения потока.
ThreadExit	Происходит при закрытии потока. Перед закрытием главного потока для приложения вызывается данное событие, за которым следует событие ApplicationExit.

IMessageFilter позволяет остановить вызов события или выполнить специальные операции до вызова обработчика событий.

Класс имеет свойства CurrentCulture и CurrentInputLanguage, чтобы получать или задавать сведения о культурной среде для текущего потока.

События класса Application

ApplicationExit	Статическое. Происходит при закрытии приложения.
Idle	Статическое. Происходит, когда приложение заканчивает обработку и

	собирается перейти в состояние незанятости.
ThreadException	Статическое. Возникает при выдаче не перехваченного исключения потока.
ThreadExit	Статическое. Происходит при закрытии потока. Перед закрытием главного потока для приложения вызывается данное событие, за которым следует событие ApplicationExit.

Итак, класс Application располагает методами для запуска и останова ПОТОКОВ и ПРИЛОЖЕНИЙ, а также для обработки Windows messages.

Вызов методов Run обеспечивает выполнение цикла обработки сообщений (an application message loop) в текущем потоке, а также, возможно, делает видимой соответствующую форму.

Вызов методов Exit и ExitThread приводит к остановке цикла обработки сообщений.

Вызов DoEvents позволяет активизировать обработку сообщений практически из любого места выполняемого программного кода. Например, во время выполнения операторов цикла.

Вызов AddMessageFilter обеспечивает добавление фильтра сообщений to the application message pump для monitor Windows messages.

Интерфейс IMessageFilter позволяет реализовывать специальные алгоритмы непосредственно перед вызовом обработчика сообщения.

Класс статический и объектов-представителей этого класса создать невозможно!

Windows message

Прежде всего, Message – это СТРУКТУРА, представляющая в .NET сообщения Windows. Те самые, которые адресуются приложению и используются системой как средство уведомления выполняющихся в Windows приложений.

Эта структура используется также для формирования собственных сообщений, которые могут формироваться “в обход системы” и передаваться для последующей их обработки оконным процедурам приложений. Это стандартный интерфейс обмена информацией между приложениями. Важно, чтобы приложение “понимало” смысл происходящего.

Объект-представитель Message structure, может быть создан с использованием метода Create (создать – не означает отправить).

Список членов Message structure

Члены	Объявление
HWnd property. Gets or sets the window handle of the message.	public IntPtr HWnd(get; set;)
Msg property. Gets or sets the ID number for the message.	public int Msg {get; set;}
WParam property. Gets or sets the WParam field of the message. Значение этого поля зависит от конкретного сообщения. Use the WParam field to get information that is important to handling the message. Это поле обычно используется для фиксирования small pieces of information, например, значений флагов.	public IntPtr WParam {get; set;}
LParam property. Gets or sets the LParam field of the message. The value of this field depends on the message. Use the LParam field to get information that is important to handling the message. This field is typically used to store an object if it is needed by the message.	public IntPtr Lparam {get; set;}
Result property. Specifies the value that is returned to Windows in response to handling the message.	public IntPtr Result {get; set;}
Create method. Создает новую структуру Message.	public static Message Create (IntPtr hWnd, int msg, IntPtr wparam, IntPtr lparam);
Equality operator. Сравнивает два сообщения на предмет определения их идентичности.	public static Boolean operator ==(Message left, Message right);
Inequality operator. Сравнивает два сообщения на	public static Boolean

предмет определения их различия.	operator !=(Message left, Message right);
Equals метод. Compares two message structures to determine if they are equal.	public override bool Equals(object o)
GetHashCode method. Gets the hash code of the message handle.	public override int GetHashCode();

При этом IntPtr – это platform-specific тип, который используется для представления указателей или дескрипторов. Предназначен для представления целочисленных величин, размеры которых зависят от характеристик платформы (is platform-specific). То есть ожидается, что объект этого типа будет иметь размер 32 бита на 32-разрядных аппаратных средствах и операционных системах, и 64 битах на аппаратных средствах на 64 бита и операционных системах.

Тип IntPtr может использоваться языками, которые поддерживают механизм указателей, и как общее средство для обращения к данным между языками, которые поддерживают и не поддерживают указатели.

Объект IntPtr может быть также использован для поддержки дескрипторов.

For example, instances of IntPtr are used extensively in the System.IO.FileStream class to hold file handles.

Существует ещё один экзотический тип – UIntPtr, который в отличие от IntPtr не является CLS-compliant типом.

Только IntPtr тип используется в common language runtime.

UIntPtr тип разработан в основном для поддержки архитектурной симметрии (to maintain architectural symmetry) с IntPtr типом.

Примеры перехвата сообщений

В данном приложении очередь сообщений запускается без использования дополнительных классов. Выполняется в режиме отладки. Причина, по которой приложение не реагирует на сообщения системы при запуске в обычном режиме, в настоящее время мне не известна.

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

namespace MessageLoop01
{
// A message filter.
public class MyMessageFilter : IMessageFilter
{
long nMess = 0;
public bool PreFilterMessage(ref Message m)
{
nMess++;
Console.WriteLine
(“Processing the messages: {0} - {1}: {2}”,m.Msg,
nMess,
Application.MessageLoop);
return false;
}
}

/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
/// <summary>
/// The main entry Point for the application.
/// </summary>

static void Main(string[] args)
{
Application.AddMessageFilter(new MyMessageFilter());
Application.Run();
}
}
}
```

В следующем примере объект-представитель класса Form связан с оконной процедурой, в рамках которой и запускается цикл обработки сообщений. Замечательное сочетание консольного приложения с окном формы.

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

namespace MessageLoop01
{

// A message filter.
public class MyMessageFilter : IMessageFilter
{
long nMess = 0;
public bool PreFilterMessage(ref Message m)
{
nMess++;
Console.WriteLine
(“Processing the messages: {0} - {1}:{2} > {3}”,m.Msg,m.LParam,m.WParam,nMess);
return false;
}
}

class MyForm:Form
{
public MyForm()
{

if (Application.MessageLoop) Console.WriteLine(“Yes!”);
else Console.WriteLine(“No!”);

Application.AddMessageFilter(new MyMessageFilter());

}
}

/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1
{
/// <summary>
/// The main entry Point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
if (Application.MessageLoop) Console.WriteLine(“Yes!”);
else Console.WriteLine(“No!”);

Application.Run(new MyForm());
}
}
}
```

Метод WndProc

В классах - наследниках класса Control (в классах Form, Кнопка, Список, …) объявляется виртуальный метод WndProc, который обеспечивает обработку передаваемых приложению сообщений Windows. Переопределение этого метода позволяет задавать специфику поведения создаваемого элемента управления, включая и саму форму.

```
protected virtual void WndProc(ref Message m);
Parameters
m
The Windows Message to process.
```

Все сообщения после PreProcessMessage метода поступают к WndProc методу. В сущности, ничего не изменилось! WndProc method в точности соответствует WindowProc функции.

Пример переопределения WndProc

Реакция приложения на передаваемые в оконную процедуру WndProc сообщения Windows определяет поведение приложения. Метод WndProc в приложении может быть переопределён. При его переопределении следует иметь в виду, что для адекватного поведения приложения в среде Windows необходимо обеспечить вызов базовой версии метода WndProc. Всё, что не было обработано в рамках переопределённого метода WndProc, должно обрабатываться в методе базового класса.

Ниже демонстрируется пример переопределения метода WndProc. В результате переопределения приложение “научилось” идентифицировать и реагировать на системное сообщение WM_ACTIVATEAPP, которое передаётся каждому приложению при переводе приложения из активного состояния в пассивное и обратно.

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

namespace csTempWindowsApplication1
{
public class Form1 : System.Windows.Forms.Form
{
// Здесь определяется константа, которая содержится в «windows.h» header file.
private const int WM_ACTIVATEAPP = 0x001C;
// Флаг активности приложения.
private bool appActive = true;

[STAThread]
static void Main()
 {
Application.Run(new Form1());
 }

//=====================================================================
public Form1()
{
// Задание свойств формы.
this.Size = new System.Drawing.Size(300,300);
this.Text = “Form1”;
this.Font = new System.Drawing.Font(“Microsoft Sans Serif”,
18F,
System.Drawing.FontStyle.Bold,
System.Drawing.GraphicsUnit.Point,
((System.Byte)(0))
 );
}
//=====================================================================

//=====================================================================
// Вот переопределённая оконная процедура.
protected override void WndProc(ref Message m)
{
// Listen for operating system messages.
switch (m.Msg)
{
// Сообщение под кодовым названием WM_ACTIVATEAPP occurs when the application
// becomes the active application or becomes inactive.
case WM_ACTIVATEAPP:
// The WParam value identifies what is occurring.
appActive = (((int)m.WParam != 0));

// Invalidate to get new text painted.
this.Invalidate();
break;
}
base.WndProc(ref m);
```

```
}
//=====================================================================

//=====================================================================
protected override void OnPaint(PaintEventArgs e)
{
// Стиль отрисовки текста определяется состоянием приложения.
if (appActive)
{
e.Graphics.FillRectangle(SystemBrushes.ActiveCaption,20,20,260,50);
e.Graphics.DrawString("Application is active",
this.Font, SystemBrushes.ActiveCaptionText, 20,20);
}
else
{
e.Graphics.FillRectangle(SystemBrushes.InactiveCaption,20,20,260,50);
e.Graphics.DrawString("Application is Inactive",
this.Font, SystemBrushes.ActiveCaptionText, 20,20);
}
}
//=====================================================================

}
}
```

Контекст приложения

Создаётся простейшее Windows приложение. С единственной формой. Точка входа в приложение – статическая функция Main располагается непосредственно в классе формы. Здесь форма создаётся, инициализируется, показывается.

```
// Этот явно избыточный набор пространств имён формируется
// Visual Studio по умолчанию.
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace xForm
{
/// <summary>
/// Summary description for Form1.
/// </summary>
public class Form1 : System.Windows.Forms.Form
{
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
//
// Required for Windows Form Designer support
//
InitializeComponent();

//
// TODO: Add any constructor code after InitializeComponent call
//
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
if( disposing )
{
```

```
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

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

Приложение, в задачи которого входит поддержка нескольких одновременно существующих (и, возможно, взаимодействующих) форм, создаётся с использованием дополнительного класса – класса контекста приложения. В этом классе обычно и объявляется Функция Main.

ApplicationContext (контекст приложения) специфицирует и объединяет контекстную информацию о потоках приложения.

Это класс, который позволяет собрать в единый модуль основные элементы приложения. Перечень членов класса представлен ниже. Их количество невелико, но вполне достаточно для централизованного формирования и запуска всех элементов приложения.

Пара конструкторов, которые обеспечивают инициализацию объекта. Один из них в качестве параметра использует ссылку на объект-представитель класса формы.

Свойство MainForm, определяющее главную форму данного контекста приложения.

Общедоступные методы Dispose (Overloaded. Releases the resources used by the ApplicationContext), Equals (inherited from Object), ExitThread (Terminates the message loop of the thread), GetHashCode (inherited from Object), GetType (inherited from Object), ToString (inherited from Object).

Событие ThreadExit, которое происходит когда в результате выполнение метода ExitThread прекращает выполнение цикл обработки сообщений.

Protected методы Dispose (Releases the resources used by the ApplicationContext), ExitThreadCore (Terminates the message loop of the thread), Finalize, MemberwiseClone, OnMainFormClosed.

Можно обойтись без контекста приложения, однако программный код, создаваемый с использованием объекта-представителя класса контекста приложения позволяет инкапсулировать детали реализации конкретных форм приложения. Прилагаемый фрагмент приложения обеспечивает создание пары форм-представителей класса MyForm и отслеживает присутствие хотя бы одной формы. После закрытия обеих форм приложение завершает выполнение.

```
// По сравнению с предыдущим примером сократили количество
// используемых пространств имён. Оставили только необходимые.
using System;
using System.ComponentModel;
using System.Windows.Forms;

namespace xApplicationContext
{
```

```
public class xForm : System.Windows.Forms.Form
{//============================================================================

private System.ComponentModel.Container components = null;
private MyApplicationContext appContext;

// Модифицировали конструктор.
// 1. Формы украшены названиями.
// 2. В соответствии с замыслом, о факте закрытия формы
// должно быть известно объекту-контексту приложения,
// который осуществляет общее руководство приложением.
public xForm(string keyText, MyApplicationContext contextKey)
{
this.Text = keyText;
appContext = contextKey;
InitializeComponent();
}

protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
this.components = new System.ComponentModel.Container();
this.Size = new System.Drawing.Size(300,300);
// Объект-контекст приложения должен отреагировать на
// факт (событие) закрытия формы. Здесь производится
// подписка объекта-контекст приложения на событие
// закрытия формы.
this.Closed+=new EventHandler(appContext.OnFormClosed);
}
#endregion
/// <summary>
/// The main entry Point for the application.
/// Точка входа в приложение перенесена в класс
/// контекста приложения.
/// </summary>
//[STAThread]
//static void Main()
//{
//     Application.Run(new xForm());
//}
}//============================================================================


// The class that handles the creation of the application windows
public class MyApplicationContext : ApplicationContext
{//============================================================================

private int formCount;
private xForm form1;
private xForm form2;


// Конструктор Контекста приложения.
private MyApplicationContext()
{
```

```
formCount = 0;
// Create both application forms and handle the Closed event
// to know when both forms are closed.
form1 = new xForm("Form 0",this);
formCount++;

form2 = new xForm("Form 1",this);
formCount++;

form1.Show();
form2.Show();
}


public void OnFormClosed(object sender, EventArgs e)
{
// Форма закрывается - счётчик открытых форм уменьшается.
// Приложение сохраняет работоспособность до тех пор,
// пока значение счётчика не окажется равным 0.
// После этого - Exit().

formCount--;
if (formCount == 0)
{
Application.Exit();
}
}

[STAThread]
static void Main(string[] args)
{
// Создаётся объект-представитель класса MyApplicationContext,
// который берёт на себя функции кправления приложением.
MyApplicationContext context = new MyApplicationContext();
// Приложение запускается для объекта контекста приложения.
Application.Run(context);
}
}//============================================================================

}
```

Общие сведения о GDI+

GDI+ (Интерфейс Графических Устройств) — это подистема Microsoft Windows XP, обеспечивающая вывод графической информации на экраны и принтеры. GDI+ является преемником GDI, интерфейса графических устройств, включаемого в ранние версии Windows. Интерфейс GDI+ изолирует приложение от особенностей конкретного графического оборудования. Такая изоляция позволяет разработчикам создавать аппаратно-независимые приложения.

Это набор программных средств, которые используются в в .NET.

GDI+, позволяют создателям приложений выводить данные на экран или на принтер без необходимости обеспечивать работу с определенными типами устройств отображения. Для отображения информации программисту достаточно вызывать нужные методы классов GDI+. При этом автоматически учитываются типы определенных устройств и выполняются вызовы соответствующих драйверов.

GraphicsPath

GraphicsPath – класс, представляющий последовательность соединенных линий и кривых. Класс не наследуется. Список членов класса представляется ниже.

Приложения используют контуры для отображения очертаний фигур, заполнения внутренних областей фигур, создания областей отсечения.

Контур может состоять из любого числа фигур (контуров). Каждая фигура либо составлена из последовательности линий и кривых, либо является геометрическим примитивом. Начальная точка фигуры – первая точка в последовательности соединенных линий и кривых. Конечная точка – последняя точка в последовательности.

Фигура, состоящая из последовательности соединенных линий и кривых (чьи начальная и конечная точки могут совпадать), является разомкнутой фигурой,

если она не замкнута явно. Фигура может быть замкнута явно с помощью метода CloseFigure, который замыкает фигуру путем соединения конечной и начальной точек линией. Фигура, состоящая из геометрического примитива, является замкнутой.

Для целей заполнения и отсечения (например, если контур визуализируется с помощью метода Graphics.FillPath) все разомкнутые фигуры замыкаются путем добавления линии от первой точки фигуры к последней.

Новая фигура начинается неявно, когда создается контур или фигура замыкается. Новая фигура создается явно, когда вызывается метод StartFigure.

Когда к контуру добавляется геометрический примитив, то он добавляет фигуру, содержащую геометрический примитив, а также неявно начинает новую фигуру. Следовательно, в контуре всегда существует текущая фигура. Когда линии и кривые добавляются к контуру, то добавляется неявная линия, соединяющая конечную точку текущей фигуры с начальной точкой новых линий и кривых, чтобы сформировать последовательность соединенных линий и кривых.

У фигуры есть направление, определяющее, как отрезки прямых и кривых следуют от начальной точки к конечной. Направление определяется либо порядком добавления линий и кривых к фигуре, либо геометрическим примитивом. Направление используется для определения внутренних областей контура для целей отсечения и заполнения. О структуре и назначении классов можно судить по списку членов класса, который представлен ниже.

Конструкторы

GraphicsPath - конструктор	Инициализирует новый экземпляр класса GraphicsPath с перечислением FillMode из Alternate.

Свойства

FillMode	Получает или задает перечисление FillMode, определяющее, как заполняются внутренние области фигур в объекте GraphicsPath.
PathData	Получает объект PathData, инкапсулирующий массивы точек (points) и типов (types) для объекта GraphicsPath.
PathPoints	Получает точки в контуре.
PathTypes	Получает типы соответствующих точек в массиве PathPoints.
PointCount	Получает число элементов в массиве PathPoints или PathTypes.

Методы

AddArc	Присоединяет дугу эллипса к текущей фигуре.
AddBezier	Добавляет в текущую фигуру кривую Безье третьего порядка.
AddBeziers	Добавляет в текущую фигуру последовательность соединенных кривых Безье третьего порядка.
AddClosedCurve	Добавляет замкнутую кривую к данному контуру. Используется кривая фундаментального сплайна, поскольку кривая проходит через все точки массива.
AddCurve	Добавляет в текущую фигуру кривую сплайна. Используется кривая фундаментального сплайна, поскольку кривая проходит через все точки массива.
AddEllipse	Добавляет эллипс к текущему пути.
AddLine	Добавляет отрезок прямой к объекту GraphicsPath.
AddLines	Добавляет последовательность соединенных отрезков прямых в конец объекта GraphicsPath.
AddPath	Добавляет указанный объект GraphicsPath к данному контуру.
AddPie	Добавляет контур сектора к данному контуру.
AddPolygon	Добавляет многоугольник к данному контуру.
AddRectangle	Добавляет прямоугольник к данному контуру.
AddRectangles	Добавляет последовательность прямоугольников к данному контуру.
AddString	Добавляет строку текста в данный путь.
ClearMarkers	Удаляет все маркеры из данного контура.
Clone	Создает точную копию данного контура.
CloseAllFigures	Замыкает все незамкнутые фигуры в данном контуре и открывает новую фигуру. Каждая незамкнутая фигура замыкается путем соединения ее начальной и конечной точек линией.
CloseFigure	Замыкает текущую фигуру и открывает новую. Если текущая фигура содержит последовательность соединенных линий и кривых, то метод замыкает ее путем соединения начальной и конечной точек линией.

CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
Dispose	Освобождает все ресурсы, используемые объектом GraphicsPath.
Equals (унаследовано от Object)	Определяет, равны ли два экземпляра Object.
Flatten	Преобразует каждую кривую в данном контуре в последовательность соединенных отрезков прямых.
GetBounds	Возвращает прямоугольник, ограничивающий объект GraphicsPath.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetLastPoint	Получает последнюю точку массива PathPoints объекта GraphicsPath.
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
IsOutlineVisible	Указывает, содержится ли определенная точка внутри (под) контура объекта GraphicsPath при отображении его с помощью указанного объекта Pen.
IsVisible	Определяет, содержится ли указанная точка в объекте GraphicsPath.
Reset	Очищает массивы PathPoints и PathTypes и устанавливает FillMode в Alternate.
Reverse	Изменяет порядок точек в массиве PathPoints объекта GraphicsPath на противоположный.
SetMarkers	Устанавливает маркер на объекте GraphicsPath.
StartFigure	Открывает новую фигуру, не замыкая при этом текущую. Все последующие точки, добавляемые к контуру, добавляются к новой фигуре.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.
Transform	Применяет матрицу преобразования к объекту GraphicsPath.
Warp	Применяет преобразование перекоса, определяемое прямоугольником и параллелограммом, к объекту GraphicsPath.
Widen	Заменяет данный контур кривыми, окружающими область, заполняемую при отображении контура указанным пером.

Защищенные методы

Finalize	Переопределен. См. Object.Finalize. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

Region

Описывает внутреннюю часть графической формы, состоящей из прямоугольников и фигур, составленных из замкнутых линий. Этот класс не наследуется.

Область является масштабируемой. Приложение может использовать области для фиксации выходных данных операций рисования. Диспетчер окон использует области для определения области изображения окон. Эти области называются вырезанными. Приложение может также использовать области в операциях проверки наличия данных, например пересечения точки или прямоугольника с областью. Приложение может заполнять область с помощью объекта Brush.

Множество пикселов, входящих в состав региона, может состоять из нескольких несмежных участков.

Список членов класса представляется ниже.

Открытые конструкторы

Region - конструктор	Инициализирует новый объект Region.

Открытые методы

Clone	Создает точную копию объекта Region.
Complement	Обновляет объект Region, чтобы включить часть указанной структуры RectangleF, не пересекающуюся с объектом Region.
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
Dispose	Освобождает все ресурсы, используемые объектом Region.
Equals	
Exclude	Обновляет объект Region, чтобы включить часть его внутренней части, не пересекающуюся с указанной структурой Rectangle.
FromHrgn	Инициализирует новый объект Region из дескриптора указанной существующей области GDI.
GetBounds	Возвращает структуру RectangleF, представляющую прямоугольник, ограничивающий объект Region на поверхности рисунка объекта Graphics.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetHrgn	Возвращает дескриптор Windows для объекта Region в указанном графическом контексте.
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetRegionData	Возвращает объект RegionData, представляющий данные, описывающие объект Region.
GetRegionScans	Возвращает массив структур RectangleF, аппроксимирующих объект Region.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
Intersect	Заменяет объект Region на его пересечение с указанным объектом Region.
IsEmpty	Проверяет, имеет ли объект Region пустую внутреннюю часть на указанной поверхности рисунка.
IsInfinite	Проверяет, имеет ли объект Region пустую внутреннюю часть на указанной поверхности рисунка.
IsVisible	Проверяет, содержится ли указанный прямоугольник в объекте Region.
MakeEmpty	Инициализирует объект Region для пустой внутренней части.
MakeInfinite	Инициализирует объект Region для бесконечной внутренней части.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.
Transform	Преобразует этот объект Region с помощью указанного объекта Matrix.
Translate	Смещает координаты объекта Region на указанную величину.
Union	Заменяет объект Region на его объединение с указанным объектом GraphicsPath.
Xor	Заменяет объект Region на разность объединения и его пересечения с указанным объектом GraphicsPath.

Защищенные методы

Finalize	Переопределен. См. Object.Finalize.

	В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

Применение классов GraphicsPath и Region. Круглая форма

```
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace StrangeControls
{
/// <summary>
/// Объявляется класс-наследник базового класса Form.
/// Наша форма, конечно же, основывается на обычной классической
/// прямоугольной, масштабируемой форме.
/// Прежде всего, это означает,что у неё имеется свойство Size,
/// значения которого используются для определения размеров
/// составляющих нашу форму элементов.
/// Наша форма состоит из двух элементов:
/// Прямоугольника-заголовка (здесь должны размещаться основные элементы управления
формы).
/// Эллипсовидной клиентской области с голубой каёмочкой.
/// </summary>
public class RoundForm : System.Windows.Forms.Form
{

private System.ComponentModel.Container components = null;
// Объект headSize используется для управления размерами формы.
private Size headSize;
// bw - (border width) это переменная,
// сохраняющая метрические характеристики рамки.
// В дальнейшем это значение используется для определения
// размеров нашей формы.
int bw;

// Ничем не примечательный конструктор...
public RoundForm()
{
       InitializeComponent();
}


#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
       //
       // RoundForm
       //
       this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
       this.ClientSize = new System.Drawing.Size(292, 273);
       this.Cursor = System.Windows.Forms.Cursors.Cross;
       this.MinimumSize = new System.Drawing.Size(100, 100);
       this.Name = "RoundForm";
       this.Text = "RoundForm";
       this.Resize += new System.EventHandler(this.RoundForm_Resize);

}
#endregion

/// <summary>
/// Clean up any resources being used.
```

```
/// </summary>
protected override void Dispose( bool disposing )
{
	if( disposing )
	{
		if(components != null)
		{
			components.Dispose();
		}
	}
	base.Dispose( disposing );
}


// Обработчик события, связанного с изменением формы окна обеспечивает
// вызов метода, в котором определяются основные характеристики
// составляющих формы.
private void RoundForm_Resize(object sender, System.EventArgs e)
{
SetSize();
}

// Код, определяюший габариты ФОРМЫ - весь здесь...
// Это определение метрических характеристик ФОРМЫ.
protected void SetSize()
{
 // Всего ничего...
 // На основе соответствующей "классической" формы
 // определяются размеры прямоугольной составляющей (заголовка формы).
 int w = this.Size.Width;
 bw = (int)((w - this.ClientSize.Width)/2);
 int h = this.Size.Height;
 int bh = h - this.ClientSize.Height - bw;
 headSize = new Size(w,bh);
}

// В рамках переопределённой виртуальной функции...
protected override void OnPaint(PaintEventArgs e)
{
// Определяем габариты формы.
SetSize();
// Получаем ссылку на графический контекст.
Graphics gr = this.CreateGraphics();
// Карандаш для отрисовки каёмочки.
Pen pen = new Pen(SystemColors.Desktop,bw);
// Объект, поддерживающий контуры формы.
GraphicsPath gp = new GraphicsPath();
// Контуры формы образуются прямоугольником
//(его характеристики совпадают с заголовком формы).
// класс GraphicsPath не имеет метода, который позволял
// бы непосредственно подсоединять прямоугольные области.
// Поэтому структура Rectangle предварительно определяется.
// С эллипсом проще. Он подсоединяется "на лету",
// без предварительного опредедления.
Rectangle rect = new Rectangle(0,0,headSize.Width,headSize.Height);
gp.AddRectangle(rect);
gp.AddEllipse(bw,headSize.Height,this.ClientSize.Width,this.ClientSize.Height);
// Сформировали объект, поддерживающий контуры формы.
// И на его основе создаём объект, который описывает внутреннюю часть
// графической формы.
// В нашем случае она состоит из прямоугольника и эллипса.
Region reg = new Region(gp);
// У любой формы имеется свойство Region.
// У нашей формы - он прекрасен.
this.Region = reg;
// Рисуем каёмочку...
gr.DrawEllipse(pen,0,0,this.ClientSize.Width,this.ClientSize.Height);
// Освобождаем занятые ресурсы.
gr.Dispose();
}
```

```
}
}
```

Собственные элементы управления

В этом разделе обсуждаются некоторые аспекты проблемы построения собственных элементов управления.

Известно, по крайней мере, три возможных подхода к разработке новых элементов управления:

- объединение стандартных элементов управления в группы (составные элементы управления),
- объявление новых классов, наследующих от существующих элементов управления,
- написание новых элементов “с нуля”.

Разработка составных элементов управления предполагает объявление класса, производного от класса UserControl и использование мастера UserControl, для добавления вложенных элементов управления с последующей настройкой образующих элементов.

Новый элемент управления может быть построен на основе класса-наследника какого-либо из существующих элементов управления. В этом случае в новом классе удаётся частично использовать функциональные возможности ранее объявленного класса, возможно, сохраняя при этом внешний вид элемента. Например, можно объявить собственный вариант класса кнопки, который будет наследовать классу Button.

Написание нового элемента “с нуля” отличается от предыдущего варианта разработки выбором базового класса. В этом случае основываются на классе Control, который не предоставляет потомкам даже элементарного графического интерфейса. Процесс визуализации в этом случае обеспечивается переопределяемым обработчиком события Paint. При этом переопределяется виртуальный метод базового класса OnPaint с единственным аргументом типа PaintEventArgs, который содержит информацию о клиентсой области элемента управления. Член этого класса объект типа Graphics обеспечивает формирование представление элемента управления. Второй член класса - объект типа ClipRectangle описывает доступную клентскую область элемента управления.

Следует отметить, что между двумя последними способами определения элементов управления не существует чётких границ. В обоих случаях основанием для классификации оказывается объём работы по доопределению и переопределению методов и свойств вновь создаваемого класса элементов управления.

В рассматриваемом ниже примере определения собственного элемента управления используется объект-представитель класса ImageList. Объекты этого класса предназначаются для сохранения рисунков, которые могут отображаться другими элементами управления. В общем случае этот компонент позволяет написать код для унифицированного каталога рисунков. В нашем случае он используется для изменения внешнего вида элемента управления. К каждому рисунку можно получить доступ с помощью значения индекса этого рисунка. Отображаемые рисунки имеют один и тот же формат и размер, устанавливаемый в свойстве ImageSize. Таким образом, на основе данного свойства может быть реализован эффект масштабирования элемента управления в смысле изменения его видимых размеров в случае изменения клиентских размеров формы.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Data;
using System.Windows.Forms;

namespace StrangeControls
{
/// <summary>
/// Summary description for RoundButton.
/// Хотя класс RoundButton и объявляется на основе базового
/// класса Button, внешнее представление его
/// объектов-представителей реализуется в переопределяемом
/// методе OnPaint с использованием класса ImageList.
```

```
/// </summary>
public class RoundButton : Button
{
ImageList imgList;
// Внешнее представление элемента управления меняется
// в зависимости от состояния элемента. Это состояние
// зависит от конкретных событий, происходящих с элементом
// управления.
// Индекс, который используется при изменении внешнего
// представления элемента управления.
// Внешний вид элемента управления определяется значением
// переменной indexB, которая меняет значение в рамках
// переопределяемых обработчиков событий.
// Функциональность данного элемента управления задана
// в классе формы, в виде стандартных (непереопределяемых) обработчиков.

int indexB;

public RoundButton():base()
{
       indexB = 0;
       imgList = new ImageList();
       imgList.Images.Add(Image.FromFile(@"jpg0.jpg"));
       imgList.Images.Add(Image.FromFile(@"jpg1.jpg"));
       imgList.Images.Add(Image.FromFile(@"jpg2.jpg"));
       imgList.Images.Add(Image.FromFile(@"jpg3.jpg"));
       this.ImageList=imgList;

}

// Код, управляющий изменением внешнего вида элемента.
// Сюда передаётся управление в результате выполнения метода
// Refresh().
protected override void OnPaint(PaintEventArgs e)
{
       GraphicsPath gp = new GraphicsPath();
    gp.AddEllipse(10,10,50,50);
    Region reg = new Region(gp);
    // Свойству Region присваивается новое значение - новый регион!
       this.Region = reg;
    e.Graphics.DrawImage(ImageList.Images[indexB],10,10,50,50);
}

protected override void OnClick(EventArgs e)
{
       indexB = 0;
       this.Refresh();
       base.OnClick (e);
}

protected override void OnMouseDown(MouseEventArgs e)
{
       indexB = 1;
       this.Refresh();
       base.OnMouseDown (e);
}

protected override void OnGotFocus(EventArgs e)
{
       indexB = 0;
       this.Refresh();
       base.OnGotFocus (e);
}

protected override void OnLostFocus(EventArgs e)
{
       indexB = 0;
       this.Refresh();
       base.OnLostFocus(e);
}
```

```
protected override void OnMouseEnter(EventArgs e)
{
       indexB = 2;
       this.Refresh();
       base.OnMouseEnter (e);
}

protected override void OnMouseLeave(EventArgs e)
{
       indexB = 3;
       this.Refresh();
       base.OnMouseLeave (e);
}

private void InitializeComponent()
{
       //
       // RoundButton
       //
}
}
}
```

Ниже приводится фрагмент кода формы (включая назначение и определение метода-обработчика события), использующей данный элемент управления.

```
private void InitializeComponent()
{
::::::::::::::::::::
//
// roundButton1
//
this.roundButton1.Cursor = System.Windows.Forms.Cursors.Cross;
this.roundButton1.Location = new System.Drawing.Point(136, 8);
this.roundButton1.Name = "roundButton1";
this.roundButton1.Size = new System.Drawing.Size(72, 72);
this.roundButton1.TabIndex = 2;
this.roundButton1.Text = "roundButton1";
this.roundButton1.Click += new System.EventHandler(this.roundButton_Click);
::::::::::::::::::::
}
```

Сам обработчик события в форме объявлен следующим образом:

```
private void roundButton_Click(object sender, System.EventArgs e)
{
 if (imgListindex < imgList.Images.Count-1) imgListindex++;
 else                                       imgListindex = 0;

 pictureBox1.Image = imgList.Images[imgListindex];
}
```

Следующий фрагмент кода посвящён разработке составного элемента управления, представляющего собой “батарею” из 16 одинаковых кнопочек, располагаемых на одной панели. Несмотря на внушительное количество составляющих элементов, перед нами один элемент управления с общим единым обработчиком события, связанного с нажатием на одну из 16 кнопочек. Естественно, что реализация единого обработчика требует дополнительных усилий: каждой кнопке назначается один и тот же стандартный обработчик с двумя параметрами. Один из которых используется для идентификации нажатой кнопки. При нажатии на кнопку происходит изменение состояния этой кнопки, которое кодируется посредством текстового значения (свойства Text), отображаемого на поверхности кнпки.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
```

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

namespace By2
{
/// <summary>
/// Summary description for ButtonsControl.
/// </summary>
public class ButtonsControl : System.Windows.Forms.UserControl
{
/// <summary>
/// Required designer variable.
/// </summary>

private Form1 f1;
private ArrayList bArrList;
private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.Button button16;
private System.Windows.Forms.Button button15;
private System.Windows.Forms.Button button14;
private System.Windows.Forms.Button button13;
private System.Windows.Forms.Button button12;
private System.Windows.Forms.Button button11;
private System.Windows.Forms.Button button10;
private System.Windows.Forms.Button button9;
private System.Windows.Forms.Button button8;
private System.Windows.Forms.Button button7;
private System.Windows.Forms.Button button6;
private System.Windows.Forms.Button button5;
private System.Windows.Forms.Button button4;
private System.Windows.Forms.Button button3;
private System.Windows.Forms.Button button2;
private System.Windows.Forms.Button button1;
public ArrayList parameterList;

public void SetParent(Form1 fKey)
{
f1 = fKey;
}

private System.ComponentModel.Container components = null;

public ButtonsControl()
{
InitializeComponent();
bArrList = new ArrayList();
bArrList.Add(button16);
bArrList.Add(button15);
bArrList.Add(button14);
bArrList.Add(button13);
bArrList.Add(button12);
bArrList.Add(button11);
bArrList.Add(button10);
bArrList.Add(button9);
bArrList.Add(button8);
bArrList.Add(button7);
bArrList.Add(button6);
bArrList.Add(button5);
bArrList.Add(button4);
bArrList.Add(button3);
bArrList.Add(button2);
bArrList.Add(button1);
parameterList = new ArrayList();
}

private void SetParameterList()
{
int i, I = bArrList.Count ;
parameterList.Clear();
for (i = 0; i < I;  i++)
{
```

```
parameterList.Add(int.Parse(((Button)bArrList[i]).Text));
}
f1.formDecimalValue(parameterList);
}

public void SetButtons(ArrayList lst)
{
int i, I = lst.Count, J = bArrList.Count;

for (i = 0; i < J; i++)
{
((Button)bArrList[i]).Text = "0";
}

for (i = 0; i < I; i++)
{
((Button)bArrList[i]).Text = ((int)lst[i]).ToString();
}

}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
if( disposing )
{
       if(components != null)
       {
              components.Dispose();
       }
}
base.Dispose( disposing );
}

#region Component Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
this.panel1 = new System.Windows.Forms.Panel();
this.button16 = new System.Windows.Forms.Button();
this.button15 = new System.Windows.Forms.Button();
this.button14 = new System.Windows.Forms.Button();
this.button13 = new System.Windows.Forms.Button();
this.button12 = new System.Windows.Forms.Button();
this.button11 = new System.Windows.Forms.Button();
this.button10 = new System.Windows.Forms.Button();
this.button9 = new System.Windows.Forms.Button();
this.button8 = new System.Windows.Forms.Button();
this.button7 = new System.Windows.Forms.Button();
this.button6 = new System.Windows.Forms.Button();
this.button5 = new System.Windows.Forms.Button();
this.button4 = new System.Windows.Forms.Button();
this.button3 = new System.Windows.Forms.Button();
this.button2 = new System.Windows.Forms.Button();
this.button1 = new System.Windows.Forms.Button();
this.panel1.SuspendLayout();
this.SuspendLayout();
//
// panel1
//
this.panel1.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;
this.panel1.Controls.Add(this.button16);
this.panel1.Controls.Add(this.button15);
this.panel1.Controls.Add(this.button14);
this.panel1.Controls.Add(this.button13);
this.panel1.Controls.Add(this.button12);
```

```
this.panel1.Controls.Add(this.button11);
this.panel1.Controls.Add(this.button10);
this.panel1.Controls.Add(this.button9);
this.panel1.Controls.Add(this.button8);
this.panel1.Controls.Add(this.button7);
this.panel1.Controls.Add(this.button6);
this.panel1.Controls.Add(this.button5);
this.panel1.Controls.Add(this.button4);
this.panel1.Controls.Add(this.button3);
this.panel1.Controls.Add(this.button2);
this.panel1.Controls.Add(this.button1);
this.panel1.Location = new System.Drawing.Point(8, 8);
this.panel1.Name = "panel1";
this.panel1.Size = new System.Drawing.Size(648, 48);
this.panel1.TabIndex = 0;
//
// button16
//
this.button16.Location = new System.Drawing.Point(605, 12);
this.button16.Name = "button16";
this.button16.Size = new System.Drawing.Size(32, 23);
this.button16.TabIndex = 146;
this.button16.Text = "0";
this.button16.Click += new System.EventHandler(this.buttonX_Click);
//
// button15
//
this.button15.Location = new System.Drawing.Point(565, 12);
this.button15.Name = "button15";
this.button15.Size = new System.Drawing.Size(32, 23);
this.button15.TabIndex = 145;
this.button15.Text = "0";
this.button15.Click += new System.EventHandler(this.buttonX_Click);
//
// button14
//
this.button14.Location = new System.Drawing.Point(525, 12);
this.button14.Name = "button14";
this.button14.Size = new System.Drawing.Size(32, 23);
this.button14.TabIndex = 144;
this.button14.Text = "0";
this.button14.Click += new System.EventHandler(this.buttonX_Click);
//
// button13
//
this.button13.Location = new System.Drawing.Point(485, 12);
this.button13.Name = "button13";
this.button13.Size = new System.Drawing.Size(32, 23);
this.button13.TabIndex = 143;
this.button13.Text = "0";
this.button13.Click += new System.EventHandler(this.buttonX_Click);
//
// button12
//
this.button12.Location = new System.Drawing.Point(445, 12);
this.button12.Name = "button12";
this.button12.Size = new System.Drawing.Size(32, 23);
this.button12.TabIndex = 142;
this.button12.Text = "0";
this.button12.Click += new System.EventHandler(this.buttonX_Click);
//
// button11
//
this.button11.Location = new System.Drawing.Point(405, 12);
this.button11.Name = "button11";
this.button11.Size = new System.Drawing.Size(32, 23);
this.button11.TabIndex = 141;
this.button11.Text = "0";
this.button11.Click += new System.EventHandler(this.buttonX_Click);
//
// button10
```

```
//
this.button10.Location = new System.Drawing.Point(365, 12);
this.button10.Name = "button10";
this.button10.Size = new System.Drawing.Size(32, 23);
this.button10.TabIndex = 140;
this.button10.Text = "0";
this.button10.Click += new System.EventHandler(this.buttonX_Click);
//
// button9
//
this.button9.Location = new System.Drawing.Point(325, 12);
this.button9.Name = "button9";
this.button9.Size = new System.Drawing.Size(32, 23);
this.button9.TabIndex = 139;
this.button9.Text = "0";
this.button9.Click += new System.EventHandler(this.buttonX_Click);
//
// button8
//
this.button8.Location = new System.Drawing.Point(285, 12);
this.button8.Name = "button8";
this.button8.Size = new System.Drawing.Size(32, 23);
this.button8.TabIndex = 138;
this.button8.Text = "0";
this.button8.Click += new System.EventHandler(this.buttonX_Click);
//
// button7
//
this.button7.Location = new System.Drawing.Point(245, 12);
this.button7.Name = "button7";
this.button7.Size = new System.Drawing.Size(32, 23);
this.button7.TabIndex = 137;
this.button7.Text = "0";
this.button7.Click += new System.EventHandler(this.buttonX_Click);
//
// button6
//
this.button6.Location = new System.Drawing.Point(205, 12);
this.button6.Name = "button6";
this.button6.Size = new System.Drawing.Size(32, 23);
this.button6.TabIndex = 136;
this.button6.Text = "0";
this.button6.Click += new System.EventHandler(this.buttonX_Click);
//
// button5
//
this.button5.Location = new System.Drawing.Point(165, 12);
this.button5.Name = "button5";
this.button5.Size = new System.Drawing.Size(32, 23);
this.button5.TabIndex = 135;
this.button5.Text = "0";
this.button5.Click += new System.EventHandler(this.buttonX_Click);
//
// button4
//
this.button4.Location = new System.Drawing.Point(125, 12);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(32, 23);
this.button4.TabIndex = 134;
this.button4.Text = "0";
this.button4.Click += new System.EventHandler(this.buttonX_Click);
//
// button3
//
this.button3.Location = new System.Drawing.Point(85, 12);
this.button3.Name = "button3";
this.button3.Size = new System.Drawing.Size(32, 23);
this.button3.TabIndex = 133;
this.button3.Text = "0";
this.button3.Click += new System.EventHandler(this.buttonX_Click);
//
```

```
// button2
//
this.button2.Location = new System.Drawing.Point(45, 12);
this.button2.Name = "button2";
this.button2.Size = new System.Drawing.Size(32, 23);
this.button2.TabIndex = 132;
this.button2.Text = "0";
this.button2.Click += new System.EventHandler(this.buttonX_Click);
//
// button1
//
this.button1.Location = new System.Drawing.Point(5, 12);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(32, 23);
this.button1.TabIndex = 131;
this.button1.Text = "0";
this.button1.Click += new System.EventHandler(this.buttonX_Click);
//
// ButtonsControl
//
this.Controls.Add(this.panel1);
this.Name = "ButtonsControl";
this.Size = new System.Drawing.Size(664, 64);
this.panel1.ResumeLayout(false);
this.ResumeLayout(false);

}
#endregion



private void buttonX_Click(object sender, System.EventArgs e)
{
if (((Button)sender).Text.Equals("0")) ((Button)sender).Text = "1";
else                                   ((Button)sender).Text = "0";

SetParameterList();
}

}
}
```

Основы ADO.NET

Реляционные базы данных. Основные понятия

Столбец (поле, атрибут):

- характеризуется определённым типом (данных),
- множество значений столбца являются значениями одного типа.

Строка (запись, кортеж):

- характеризуется кортежем атрибутов,
- состоит из упорядоченного множества значений (кортежа) атрибутов.

Таблица:

- набор данных, представляющих объекты определённого типа,
- состоит из множества столбцов*строк,
- каждая строка таблицы УНИКАЛЬНА.

Первичный ключ таблицы:

- непустое множество столбцов таблицы (возможно, что состоящее из одного столбца), соответствующие значения (комбинации значений) которых в строках таблицы обеспечивают уникальность каждой строки в данной таблице.

Дополнительный ключ таблицы:

- а Бог его знает, зачем ещё одна гарантия уникальности строки в таблице.

Внешний ключ таблицы:

- непустое множество столбцов таблицы (возможно, что состоящее из одного столбца), соответствующие значения (комбинации значений) которых в строках таблицы соответствуют первичного или дополнительного ключа другой таблицы,
- обеспечивает логическую связь между таблицами.

Разъединённый доступ к данным (Доступ к отсоединенным данным)

В клиент-серверных приложениях обычно применяется доступ к данным через постоянное соединение с источником данных. Таким образом, приложение открывает соединение с базой данных и не закрывает его, по крайней мере, до завершения работы с источником данных. В это время соединение с источником поддерживается постоянно.

Недостатки такого подхода стали выявляться после распространения приложений, ориентированных на Интернет.

Соединения с базой данных требуют выделения системных ресурсов, и если база на сервере, то при большом количестве клиентов это может быть критично для сервера. Хотя постоянное соединение и позволяет немного ускорить работу приложения, общий убыток от растраты системных ресурсов преимущество в скорости выполнения приложения сводит на нет.

Плохое масштабирование приложений с постоянным соединением известно давно. Соединение с парой клиентов обслуживается приложением хорошо, 10 клиентов обслуживаются хуже, 100 – намного хуже…

В ADO.NET используется другая модель доступа – доступ к отсоединённым данным. При этом соединение устанавливается лишь на то время, которое необходимо для проведения определённой операции над базой данных.

Модель доступа – модель компромиссная. В ряде случаев она проигрывает по производительности традиционной модели. И для этих случаев рекомендуется вместо ADO.NET использовать ADO.

Предварительные замечания

Работа с БД на уровне приложения – это работа:

- с множеством объектов-представителей классов,
- с множествами объявлений унаследованных методов и свойств, предназначенных для решения поставленных задач,
- с множеством объявлений данных-членов, отражающих специфику структуры конкретной базы данных.

Функциональные особенности этой сложной системы взаимодействующих классов обеспечивают единообразную работу с множеством баз данных различных типов.

Деятельность программиста-разработчика приложений для работы с базами данных в основе своей ничем не отличается от того, что было раньше. Те же объявления классов и интерфейсов. И, конечно же, всё можно сделать своими

руками. Правда, в силу сложности задачи (много всяких деталек придётся вытачивать) времени на разработку может потребоваться многовато.

А потому, желательно:

- понимать принципы функционирования системы,
- представлять структуру, назначение и принципы функционирования компонентов,
- знать, для чего и как применять различные детали при решении поставленных задач,
- уметь создавать компоненты ADO.NET с использованием вспомогательных средств (волшебников),
- помнить, что все компоненты ADO.NET суть объекты-представители классов (пусть даже и не самой простой структуры),
- понимать закономерности их конструкции и быть готовым к модификации классов.

ADO.NET. Доступ к данным

Объектная модель ADO.NET реализует доступ к данным. При этом в Visual Studio.NET существует множество ВСТРОЕННЫХ мастеров и дизайнеров, которые позволяют реализовать механизмы доступа к БД ещё на этапе разработки программного кода.

С другой стороны, эта задача может быть решена непосредственно во время выполнения приложения.

Команда – некий объект, который, естественно, является представителем одного из двух классов - либо класса OleDbCommand, либо класса SqlCommand. Основное назначение объекта Команда - выполнение различных действий над Базой Данных (ИСТОЧНИКЕ ДАННЫХ) при использовании ОТКРЫТОГО СОЕДИНЕНИЯ. Сами же действия обычно кодируются оператором SQL или хранимой процедурой. Закодированная информация фиксируется с использованием объектов-представителей класса Parameter, специально разработанных для “записи” кодируемой в команде информации.

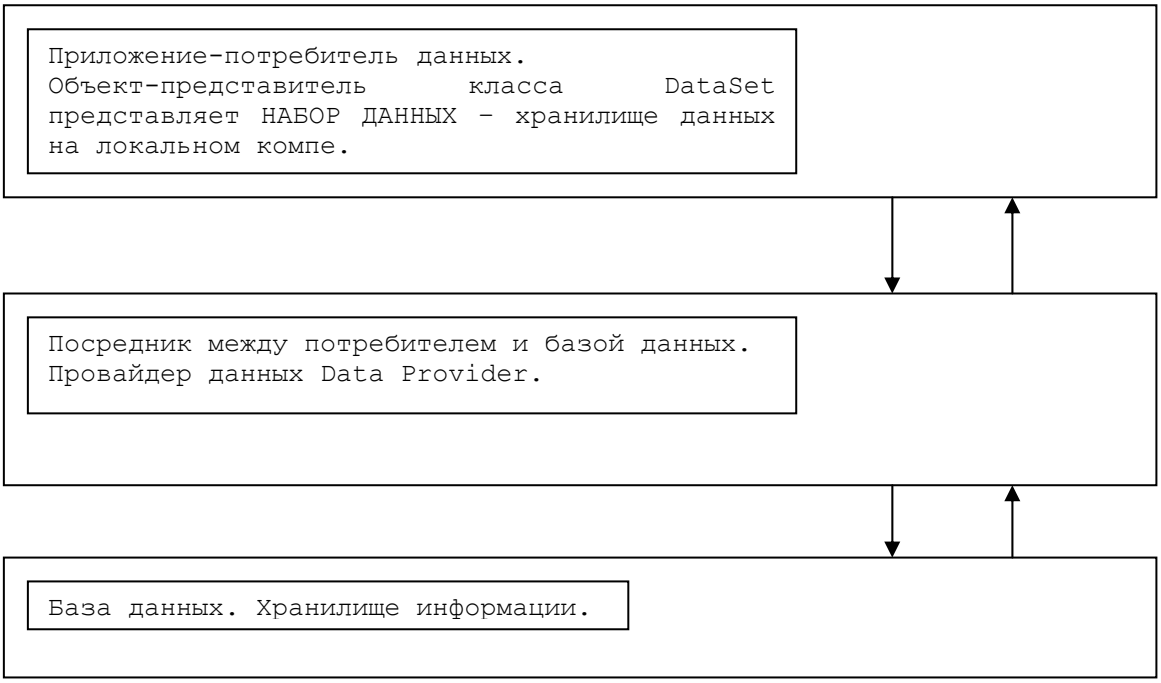
То есть, после установления соединения с БД для изменения состояния этой базы может быть создан, соответствующим образом настроен и применён объект-представитель класса …Command.

ADO.NET. Архитектура

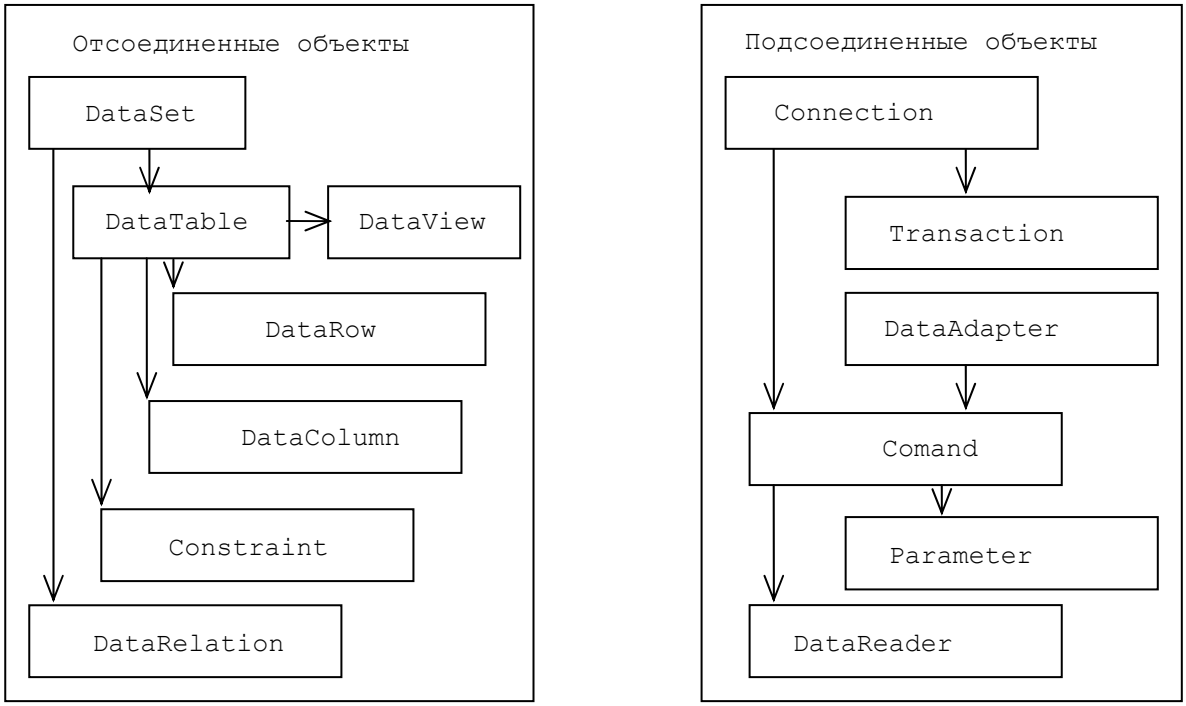
Доступ к данным в ADO.NET основан на использовании двух компонентов:

НАБОРА ДАННЫХ (представляется объектом класса DataSet) со стороны клиента. Это хранилище данных на локальном компе.

ПРОВАЙДЕРА ДАННЫХ (представляется объектом класса DataProvider) со стороны сервера. Это посредник, обеспечивающий взаимодействие приложения и базы данных.



ADO.NET. Объектная модель



Отсоединенные объекты объектной модели ADO.NET. Им посвящаются следующие главы пособия.

DataTable

Каждый объект DataTable представляет одну таблицу. Таблица в каждый конкретный момент своего существования характеризуется:

- СХЕМОЙ таблицы,
- СОДЕРЖИМЫМ таблицы (информацией).

При этом СХЕМА таблицы (структура объекта DataTable) определяется двумя наборами:

- множеством столбцов таблицы (набор DataColumns, состоящий из множества объектов DataColumn),
- множеством ограничений таблицы (набор Constraints, состоящий из множества объектов Constraint).

События класса DataTable

В классе определены четыре события, которые позволяют отслеживать и в случае необходимости отменять значения данных.

1. Изменения строк. Позволяют отслеживать и при необходимости отменять изменения данных.

- DataRowChanging – изменения вносятся в строку таблицы.

Объявление соответствующего обработчика события имеет вид:

```
private static void Row_Changing( object sender, DataRowChangeEventArgs e )
```

- DataRowChanged – изменения внесены в строку таблицы.

Объявление соответствующего обработчика события имеет вид:

```
private static void Row_Changed( object sender, DataRowChangeEventArgs e )
```

```
private static void DataTableRowChange()
{
    DataTable custTable = new DataTable("Customers");
    // Добавляем столбики.
    custTable.Columns.Add( "id", typeof(int) );
    custTable.Columns.Add( "name", typeof(string) );

    // Определяем первичный ключ.
    custTable.Columns[ "id" ].Unique = true;
    custTable.PrimaryKey = new DataColumn[] { custTable.Columns["id"] };

    // Добавляем RowChanging event handler для нашей таблицы.
    custTable.RowChanging += new DataRowChangeEventHandler( Row_Changing );
    // Добавляем a RowChanged event handler для нашей таблицы.
    custTable.RowChanged += new DataRowChangeEventHandler( Row_Changed );

    // Добавляем строки.
    for( int id=1; id<=10; id++ )
    {
     custTable.Rows.Add(
      new object[] { id, string.Format("customer{0}", id)}
                    );
    }

    custTable.AcceptChanges();

// Изменяем значение поля name во всех строках.
// Скромненько так все имена убираются, а на их место
// подставляется странное буквосочетание, состоящее из
// префикса vip и значения поля id каждого клиента.
// Была customer129 - стала vip129.

foreach( DataRow row in custTable.Rows )
    {
        row["name"] = string.Format( "vip{0}", row["id"] );
    }

}

// И после такого вмешательства результаты становятся известны
// обработчику события. Row_Changing. А толку-то…
private static void Row_Changing( object sender, DataRowChangeEventArgs e )
{
    Console.WriteLine( "Row_Changing Event: name={0}; action={1}",
        e.Row["name"], e.Action );
}
```

```
// Аналогично устроен обработчик. Row_Changed.
private static void Row_Changed( object sender, DataRowChangeEventArgs e )
{
    Console.WriteLine( "Row_Changed Event: name={0}; action={1}",
        e.Row["name"], e.Action );
}
```

Параметр обработчика события DataRowChangeEventArgs обладает двумя свойствами (Action и Row), которые позволяют определить изменяемую строку и выполняемое над строкой действие. Действие кодируется значениями специального перечисления:

```
enum RowDataAction
{
 Add,
 Change,
 Delete,
 Commit,
 Rollback,
 Nothing
}
```

2. Изменения полей (элементов в строках таблицы)

- DataColumnChanging – изменения вносятся в поле строки данных.
  Объявление соответствующего обработчика события имеет вид:

```
private static void Column_Changing( object sender, DataColumnChangeEventArgs e )
```

- DataColumnChanged – изменения были внесены в поле строки данных.
  Объявление соответствующего обработчика события имеет вид:

```
private static void Column_Changed( object sender, DataColumnChangeEventArgs e )
```

Параметр обработчика события DataColumnChangeEventArgs e обладает тремя свойствами.

Свойство	Описание
Column	Gets the DataColumn with a changing value.
ProposedValue	Gets or sets the proposed new value for the column. Новое значение для поля в строке.
Row	Строка, содержащая запись с изменяемым (изменённым) значением.

Пример аналогичного содержания. Только теперь реакция на модификацию столбца (поля), а не строки.

```
private static void DataTableColumnChanging()
{
    DataTable custTable = new DataTable("Customers");
    // add columns
    custTable.Columns.Add( "id", typeof(int) );
    custTable.Columns.Add( "name", typeof(string) );

    // set PrimaryKey
    custTable.Columns[ "id" ].Unique = true;
    custTable.PrimaryKey = new DataColumn[] { custTable.Columns["id"] };

    // add a ColumnChanging event handler for the table.
    custTable.ColumnChanging += new DataColumnChangeEventHandler( Column_Changing );
    // add a ColumnChanged event handler for the table.
    custTable.ColumnChanged += new DataColumnChangeEventHandler( Column_Changed );

    // add ten rows
    for( int id=1; id<=10; id++ )
    {
        custTable.Rows.Add(
            new object[] { id, string.Format("customer{0}", id)} );
    }
```

```
    custTable.AcceptChanges();

    // Изменяем значение поля name во всех строках.
    foreach( DataRow row in custTable.Rows )
    {
        row["name"] = string.Format( "vip{0}", row["id"] );
    }

}

private static void Column_Changing( object sender, DataColumnChangeEventArgs e )
{
Console.WriteLine
("Column_Changing Event: name={0}; Column={1}; proposed name={2}",
        e.Row["name"],
        e.Column.ColumnName,
        e.ProposedValue );
}

private static void Column_Changed( object sender, DataColumnChangeEventArgs e )
{
Console.WriteLine
("Column_Changed Event: name={0}; Column={1}; proposed name={2}",
        e.Row["name"],
        e.Column.ColumnName,
        e.ProposedValue );
}
```

DataColumns

Прежде всего!

DataColumnCollection задаёт схему таблицы, определяя тип данных каждой колонки.

В классе DataTable объявлено get-свойство DataColumns.

С помощью которого может быть получена коллекция принадлежащих таблице столбцов.

```
public DataColumnCollection Columns {get;}
```

Возвращается коллекция объектов-представителей класса DataColumn таблицы. Если у объекта-таблицы нет столбцов, возвращается null.

Объекты-представители класса DataColumn образуют набор DataColumns, который является обязательным элементом каждого объекта-представителя класса DataTable. Эти объекты соответствуют столбцам таблицы, представленной объектом-представителем класса DataTable. Объект DataColumn содержит информацию о структуре столбца (метаданные). Например, у этого объекта имеется свойство, Type, описывающее тип данных столбца. Также имеются свойства ReadOnly, Unique, Default, AutoIncrement, которые, в частности, позволяют ограничить диапазон допустимых значений поля и определить порядок генерации значений для новых данных.

Представляет тип колонки в DataTable. Это стандартный блок, предназначенный для построения схемы DataTable.

Каждый DataColumn как элемент схемы характеризуется собственным типом, который определяет тип значений, которые DataColumn содержит. Если объект DataTable создаётся как отсоединённое хранилище информации, представляющее таблицу базы данных, тип столбца объекта-таблицы должен соответствовать типу столбца таблицы в базе данных.

DataRows

СОДЕРЖИМОЕ таблицы (данные) задаётся набором DataRows – это конкретное множество строчек таблицы, каждая из которых является объектом-представителем класса DataRow.

Его методы и свойства представлены в таблице.

Открытые свойства

HasErrors	Возвращает значение, показывающее, есть ли ошибки в строке.
Item	Перегружен. Возвращает или задает данные, сохраненные в указанном столбце. В языке C# это свойство является индексатором класса DataRow.
ItemArray	Возвращает или задает все значения для этой строки с помощью массива.
RowError	Возвращает или задает пользовательское описание ошибки для строки.
RowState	Возвращает текущее состояние строки по отношению к DataRowCollection.
Table	Возвращает DataTable, для которой эта строка имеет схему.

Открытые методы

AcceptChanges	Сохраняет все изменения, сделанные с этой строкой со времени последнего вызова AcceptChanges.
BeginEdit	Начинает операцию редактирования объекта DataRow.
CancelEdit	Отменяет текущее редактирование строки.
ClearErrors	Удаляет ошибки в строке, включая RowError и ошибки, установленные SetColumnError.
Delete	Удаляет DataRow.
EndEdit	Прекращает редактирование строки.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetChildRows	Перегружен. Возвращает дочерние строки DataRow.
GetColumnError	Перегружен. Возвращает описание ошибки для столбца.
GetColumnsInError	Возвращает массив столбцов, имеющих ошибки.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetParentRow	Перегружен. Возвращает родительскую строку DataRow.
GetParentRows	Перегружен. Возвращает родительские строки DataRow.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
HasVersion	Возвращает значение, показывающее, существует ли указанная версия.
IsNull	Перегружен. Возвращает значение, показывающее, содержит ли нулевое значение указанный столбец.
RejectChanges	Отменяет все значения, выполненные со строкой после последнего вызова AcceptChanges.
SetColumnError	Перегружен. Устанавливает описание ошибки для столбца.
SetParentRow	Перегружен. Устанавливает родительскую строку DataRow.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.

Защищенные методы

Finalize (унаследовано от Object)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

SetNull	Устанавливает значение указанного DataColumn на нулевое.

Элементы этого набора являются объектами класса DataRow. В этом классе обеспечивается несколько вариантов реализации свойства Item, которые обеспечивают навигацию по множеству записей объекта DataTable, и сохранение текущих изменений данных, которые сделаны за текущий сеанс редактирования базы.

Посредством набора Rows реализуется возможность ссылки на любую запись таблицы. К любой записи можно обратиться напрямую, и потому не нужны методы позиционирования и перемещения по записям таблицы.

В примере используются различные варианты индексации. По множеству строк позиционирование проводится по целочисленному значению индекса. Выбор записи в строке производится по строковому значению, которое соответствует имени столбца.

Пример

```
private void PrintValues(DataTable myTable)
{
// Для каждой строки, которая входит в состав коллекции строк объекта таблицы…
foreach(DataRow myRow in myTable.Rows)
{
// Для каждой ячейки (столбца) в строке…
      foreach(DataColumn myCol in myTable.Columns)
      {
          // Выдать на консоль её значение!
          Console.WriteLine(myRow[myCol]);
       }
    }
 }
```

Изменение данных в DataTable и состояние строки таблицы

Основной контроль за изменениями данных в таблице возлагается на строки-объекты класса DataRow.

Для строки определены несколько состояний, которые объявлены в перечислении RowState. Контроль за сохраняемой в строках таблицы информацией обеспечивается посредством определения состояния строки, которое обеспечивается одноименным (RowState) свойством-членом класса DataRow.

Состояние	Описание
Unchanged	Строка не изменялась со времени загрузки при помощи метода Fill(), либо с момента вызова метода AcceptChanges().
Added	Строка была добавлена в таблицу, но метод AcceptChanges() ещё не вызывался.
Deleted	Строка была удалена из таблицы, но метод AcceptChanges() ещё не вызывался.
Modified	Некоторые из полей строки были изменены, но метод AcceptChanges() ещё не вызывался.
Detached	Строка НЕ ЯВЛЯТСЯ ЭЛЕМЕНТОМ КОЛЛЕКЦИИ DataRows. Её создали от имени таблицы, но не подключили

Пример. Создание таблицы, работа с записями

```
using System;
using System.Data;

namespace Rolls01
{
/// <summary>
/// Работа с таблицей:
/// определение структуры таблицы,
/// сборка записи (строки таблицы),
/// добавление новой записи в таблицу,
/// индексация записей,
/// выбор значения поля в строке,
```

```
/// изменение записи.
/// </summary>
public class RollsData
{
       public DataTable rolls;
       int rollIndex;

    public RollsData()
    {
    rollIndex = 0;

    // Создаётся объект таблица.
    rolls = new DataTable("Rolls");

    // Задаются и подсоединяются столбики, которые определяют тип таблицы.
    // Ключевой столбец.
    DataColumn dc = rolls.Columns.Add("nRoll",typeof(Int32));
    dc.AllowDBNull = false;
    dc.Unique = true;
    //rolls.PrimaryKey = dc;

    // Прочие столбцы, значения которых определяют физические
    // характеристики объектов.
    rolls.Columns.Add("Victim",typeof(Int32));
    // Значения координат.
    rolls.Columns.Add("X", typeof(Single));
    rolls.Columns.Add("Y", typeof(Single));
    // Старые значения координат.
    rolls.Columns.Add("lastX", typeof(Single));
    rolls.Columns.Add("lastY", typeof(Single));
    // Составляющие цвета.
    rolls.Columns.Add("Alpha", typeof(Int32));
    rolls.Columns.Add("Red", typeof(Int32));
    rolls.Columns.Add("Green", typeof(Int32));
    rolls.Columns.Add("Blue", typeof(Int32));
    }

       // Добавление записи в таблицу.
       public void AddRow(int key,
                          int victim,
                          float x,
                          float y,
                          float lastX,
                          float lastY,
                          int alpha,
                          int red,
                          int green,
                          int blue)
       {
              // Новая строка создаётся от имени таблицы,
              // тип которой определяется множеством ранее
              // добавленных к таблице столбцов. Подобным образом
              // созданная строка содержит кортеж ячеек
              // соответствующего типа.

              DataRow dr = rolls.NewRow();

              // Заполнение ячеек строки.
              dr["nRoll"] = key;
              dr["Victim"] = victim;
              dr["X"] = x;
              dr["Y"] = y;
              dr["lastX"] = lastX;
              dr["lastY"] = lastY;
              dr["Alpha"] = alpha;
              dr["Red"] = red;
              dr["Green"] = green;
              dr["Blue"] = blue;

              // Созданая и заполненная строка
              // подсоединяется к таблице.
```

```
rolls.Rows.Add(dr);
}

// Выбока значений очередной строки таблицы.
// Ничего особенного. Для доступа к записи (строке) используются
// выражения индексации по отношению к множеству Rows.
// Для доступа к полю выбранной записи используются
 // "индексаторы-идентификаторы".
public void NextRow(ref rPoint p)
{
p.index = (int)rolls.Rows[rollIndex]["nRoll"];
 p.victim = (int)rolls.Rows[rollIndex]["Victim"];

 p.p.X =   (float)rolls.Rows[rollIndex]["X"];
  p.p.Y =   (float)rolls.Rows[rollIndex]["Y"];

  p.lastXdirection  =  (float)rolls.Rows[rollIndex]["lastX"];
  p.lastYdirection  =  (float)rolls.Rows[rollIndex]["lastY"];

  p.c.alpha =   (int)rolls.Rows[rollIndex]["Alpha"];
  p.c.red =   (int)rolls.Rows[rollIndex]["Red"];
  p.c.green =   (int)rolls.Rows[rollIndex]["Green"];
  p.c.blue =   (int)rolls.Rows[rollIndex]["Blue"];

  p.cp.alpha = p.c.alpha - 50;
  p.cp.red = p.c.red - 10;
  p.cp.green = p.c.green - 10;
  p.cp.blue = p.c.blue - 10;

   rollIndex++; // Изменили значение индекса строки.

  if (rollIndex == rolls.Rows.Count) rollIndex = 0;
}

 // Та же выборка, но в параметрах дополнительно указан индекс записи.
public void GetRow(ref rPoint p, int key)
{
   p.index = (int)rolls.Rows[key]["nRoll"];
   p.victim = (int)rolls.Rows[key]["Victim"];

   p.p.X =   (float)rolls.Rows[key]["X"];
   p.p.Y =   (float)rolls.Rows[key]["Y"];

   p.lastXdirection  =  (float)rolls.Rows[key]["lastX"];
   p.lastYdirection  =  (float)rolls.Rows[key]["lastY"];

   p.c.alpha =   (int)rolls.Rows[key]["Alpha"];
   p.c.red =     (int)rolls.Rows[key]["Red"];
   p.c.green =   (int)rolls.Rows[key]["Green"];
   p.c.blue =    (int)rolls.Rows[key]["Blue"];

   p.cp.alpha = p.c.alpha - 50;
   p.cp.red = p.c.red - 10;
   p.cp.green = p.c.green - 10;
   p.cp.blue = p.c.blue - 10;

   if (rollIndex == rolls.Rows.Count) rollIndex = 0;
}

// Изменяется значение координат и статуса точки.
 // Значение порядкового номера объекта-параметра используется
 // в качестве первого индексатора, имя столбца – в
 // качестве второго. Скотрость выполнения операции присваивания
 // значения ячейке оставляет желать лучшего.
public void SetXYStInRow(rPoint p)
{
       rolls.Rows[p.index]["X"] = p.p.X;
       rolls.Rows[p.index]["Y"] = p.p.Y;

       rolls.Rows[p.index]["lastX"] = p.lastXdirection;
       rolls.Rows[p.index]["lastY"] = p.lastYdirection;
```

```
            rolls.Rows[p.index]["Victim"] = p.victim;
        }


        public void ReSetRowIndex()
        {
            rollIndex = 0;
        }

}
}
```

Relations

В классе DataSet определяется свойство Relations - набор объектов-представителей класса DataRelations. Каждый такой объект определяет связи между составляющими объект DataSet объектами DataTable (таблицами). Если в DataSet более одного набора DataTable, набор DataRelations будет содержать несколько объектов типа DataRelation. Каждый объект определяет связи между таблицами – DataTable. Таким образом, в DataSet реализован полный набор для управления данными, включая сами таблицы, ограничения и отношения между таблиц.

Constraints

Объекты-представители класса Constraint в наборе Constraints объекта DataTable позволяет задать на множестве объектов DataTable различные ограничения. Например, можно создать объект Constraint, гарантирующий, что значение поля или нескольких полей будут уникальны в пределах DataTable.

DataView

Объекты-представители класса DataView НЕ ПРЕДНАЗНАЧЕНЫ для организации визуализации объектов DataTable.

Их назначение – простой последовательный доступ к строкам таблицы. Объекты DataView являются средством перебора записей таблицы. При обращении ЧЕРЕЗ объект DataView к таблице получают данные, которые хранятся в этой таблице.

DataView нельзя рассматривать как таблицу. DataView не может обеспечить представление таблиц. Также DataView не может обеспечить исключения и добавления столбцов. Таким образом, DataView НЕ является средством преобразования исходной информации, зафиксированной в таблице.

После создания объекта DataView и его настройки на конкретную таблицу, появляется возможность перебора записей, их фильтрации, поиска, сортировки.

DataView предоставляет средства динамического представления набора данных, к которому можно применить различные вырианты сортировки и фильтрации на основе критериев, обеспечиваемых базой данных.

Класс DataView обладает большим набором свойств, методов и событий, что позволяет с помощью объекта-представителя класса DataView создавать различные представления данных, содержащихся в DataTable, a capability that is often used in data-binding applications.

Используя этот объект, можно представлять содержащиеся в таблице данные в соответствии с тем или иным порядком сортировки, а также можно организовать фильтрацию данных by row state или основанных на выражениях фильтрации.

A DataView предоставляет динамический взгляд на содержимое таблицы в зависимости от установленного в таблице порядка представления и вносимых в таблицы изменений.

Это отличается от метода Select, определённого в DataTable, который возвращает массив DataRow (строк).

Для управления установками представления для всех таблиц, входящих в DataSet, используется объект-представитель класса DataViewManager.

DataViewManager предоставляет удобный способ управления параметрами настройки представления умолчанию для каждой таблицы.

Примеры использования DataView

Для организации просмотра информации, сохраняемой объектом-представителем класса DataTable через объект-представитель класса DataView, этот объект необходимо связать с таблицей.

Таким образом, в приложении создаётся (независимый!) вьюер, который связывается с таблицей.

Итак, имеем:

```
DataTable tbl = new DataTable("XXX"); // Объявлен и определён объект таблица.
DataView vie; // Ссылка на вьюер.


vie = new DataView(); // Создали…
vie.Table = tbl; // Привязали таблицу к вьюеру.

// Можно и так…

vie = new DataView(tbl); // Создали и сразу привязали…
```

Управление вьюером осуществляется посредством и различных достаточно простых манипуляций, включая изменение свойств объекта.
Например, сортировка включается следующим образом:

```
// Предполагается наличие кнопочных переключателей,
// в зависимости от состояния которых в свойстве Sort
// вьюера выставляется в качестве значения имя того или
// иного столбца. Результат сортировки становится виден
// непосредственно после передачи информации объекту,
// отвечающему за демонстрацию таблицы (rpDataGreed).

if (rBN.Checked) rd.view.Sort = "nRoll";
if (rBX.Checked) rd.view.Sort = "X";
if (rBY.Checked) rd.view.Sort = "Y";

this.rpDataGrid.DataSource = rd.view;
```

Следующий пример кода демонстрирует возможности поиска, реализуемые объектом-представителем класса DataView. Сортировка обеспечивается вариантами методов Find и FindRows, которые способны в различной реализации воспринимать отдельные значения или массивы значений.

Поиск информации проводится по столбцам, предварительно перечисленным в свойстве Sort. При неудовлетворительном результате поиска метод Find возвращает отрицательное значение, метод FindRows – нулевое.

В случае успеха метода Sort возвращается индекс первой найденной записи. По этому индексу можно получить непосредственный доступ к записи.

```
:::::
// Выставили значение индекса
int findIndex = -1;

:::::
// Поиск в строке по полю "nRoll" (целочисленный столбец)
rd.view.Sort = "nRoll";
try
{
// Проверка на соответствие типа.
int.Parse(this.findTtextBox.Text);
// Сам поиск.
findIndex = rd.view.Find(this.findTtextBox.Text);
}
catch (Exception e1)
{
this.findTtextBox.Text = "Integer value expected...";
}
```

```
}
:::::
// Проверка результатов.
if (findIndex == -1)
{
this.findTtextBox.Text = "Row not found: " + this.findTtextBox.Text;
}
else
{
this.findTtextBox.Text =
"Yes :" + rd.view[findIndex]["nRoll"].ToString() +
","     + rd.view[findIndex]["X"].ToString() +
","     + rd.view[findIndex]["Y"].ToString();
}
:::::
```

Применение метода FindRows. В случае успешного завершения поиска возвращается массив записей, элементы которого могут быть выбраны посредством цикла foreach.

```
:::::
// Массив для получения результатов поиска
DataRowView[] rows;

:::::
// Поиск в строке по полю "nRoll" (целочисленный столбец)
rd.view.Sort = "nRoll";
try
{
// Проверка на соответствие типа.
int.Parse(this.findTtextBox.Text);
// Сам поиск. Возвращается массив rows.
rows = rd.view.FindRows(this.findTtextBox.Text);
}
catch (Exception e1)
{
this.findTtextBox.Text = "Integer value expected...";
}
}
:::::
// Проверка результатов.
if (rows.Length == 0)
{
this.findTtextBox.Text = "No rows found: " + this.findTtextBox.Text;
}
else
{
 foreach (DataRowView row in rows)
{
this.findTtextBox.Text =
row["nRoll"].ToString() +
","     + row["X"].ToString() +
","     + row["Y"].ToString();
}
}
:::::
```

В примере демонстрируется взаимодействие таблицы и заточенного под неё вьюера. Таблица автономна, не включена с DataSet и не связана с базой данных. Записи в таблицу можно добавлять как непосредственно, так и через вьюер. При этом необходимо совершать некоторое количество дополнительных действий. Через него же организуется поиск записей. Также просматриваются мотивы организации сортировок.

```
using System;
using System.Data;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
```

```
namespace Lights01
{
public class DataViewForm : System.Windows.Forms.Form
{
private System.ComponentModel.Container components = null;

DataTable dt;          // Таблица.
DataColumn c1, c2;     // Столбцы таблцы.
DataRow dr;            // Строка таблицы.
DataView dv;           // Вьюер таблицы.
DataRowView rv;        // Вьюер строки таблицы.

int currentCounter;    // Счётчик текущей строки для вьюера таблицы.

private System.Windows.Forms.DataGrid dG;
private System.Windows.Forms.DataGrid dGforTable;
private System.Windows.Forms.Button buttPrev;
private System.Windows.Forms.Button buttFirst;
private System.Windows.Forms.Button buttLast;
private System.Windows.Forms.Button buttonFind;
private System.Windows.Forms.TextBox demoTextBox;
private System.Windows.Forms.TextBox findTextBox;
private System.Windows.Forms.Button buttonAdd;
private System.Windows.Forms.Button buttonAcc;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.GroupBox groupBox2;
private System.Windows.Forms.Button buttNext;

public DataViewForm()
{
InitializeComponent();
CreateTable();
dG.DataSource = dv;
dGforTable.DataSource = dt;
currentCounter = 0;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();

}

protected override void Dispose( bool disposing )
{
if( disposing )
{
       if(components != null)
       {
              components.Dispose();
       }
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
this.dG = new System.Windows.Forms.DataGrid();
this.demoTextBox = new System.Windows.Forms.TextBox();
this.buttPrev = new System.Windows.Forms.Button();
this.buttNext = new System.Windows.Forms.Button();
this.buttFirst = new System.Windows.Forms.Button();
this.buttLast = new System.Windows.Forms.Button();
this.findTextBox = new System.Windows.Forms.TextBox();
this.buttonFind = new System.Windows.Forms.Button();
this.buttonAdd = new System.Windows.Forms.Button();
this.dGforTable = new System.Windows.Forms.DataGrid();
this.buttonAcc = new System.Windows.Forms.Button();
```

```
this.groupBox1 = new System.Windows.Forms.GroupBox();
this.groupBox2 = new System.Windows.Forms.GroupBox();
((System.ComponentModel.ISupportInitialize)(this.dG)).BeginInit();
((System.ComponentModel.ISupportInitialize)(this.dGforTable)).BeginInit();
this.groupBox1.SuspendLayout();
this.groupBox2.SuspendLayout();
this.SuspendLayout();
//
// dG
//
this.dG.DataMember = "";
this.dG.HeaderForeColor = System.Drawing.SystemColors.ControlText;
this.dG.Location = new System.Drawing.Point(8, 80);
this.dG.Name = "dG";
this.dG.Size = new System.Drawing.Size(280, 128);
this.dG.TabIndex = 0;
this.dG.MouseDown += new System.Windows.Forms.MouseEventHandler(this.dG_MouseDown);
//
// demoTextBox
//
this.demoTextBox.Location = new System.Drawing.Point(152, 48);
this.demoTextBox.Name = "demoTextBox";
this.demoTextBox.Size = new System.Drawing.Size(128, 20);
this.demoTextBox.TabIndex = 1;
this.demoTextBox.Text = "";
//
// buttPrev
//
this.buttPrev.Location = new System.Drawing.Point(189, 16);
this.buttPrev.Name = "buttPrev";
this.buttPrev.Size = new System.Drawing.Size(25, 23);
this.buttPrev.TabIndex = 2;
this.buttPrev.Text = "<-";
this.buttPrev.Click += new System.EventHandler(this.buttPrev_Click);
//
// buttNext
//
this.buttNext.Location = new System.Drawing.Point(221, 16);
this.buttNext.Name = "buttNext";
this.buttNext.Size = new System.Drawing.Size(25, 23);
this.buttNext.TabIndex = 3;
this.buttNext.Text = "->";
this.buttNext.Click += new System.EventHandler(this.buttNext_Click);
//
// buttFirst
//
this.buttFirst.Location = new System.Drawing.Point(157, 16);
this.buttFirst.Name = "buttFirst";
this.buttFirst.Size = new System.Drawing.Size(25, 23);
this.buttFirst.TabIndex = 4;
this.buttFirst.Text = "<<";
this.buttFirst.Click += new System.EventHandler(this.buttFirst_Click);
//
// buttLast
//
this.buttLast.Location = new System.Drawing.Point(253, 16);
this.buttLast.Name = "buttLast";
this.buttLast.Size = new System.Drawing.Size(25, 23);
this.buttLast.TabIndex = 5;
this.buttLast.Text = ">>";
this.buttLast.Click += new System.EventHandler(this.buttLast_Click);
//
// findTextBox
//
this.findTextBox.Location = new System.Drawing.Point(8, 48);
this.findTextBox.Name = "findTextBox";
this.findTextBox.Size = new System.Drawing.Size(128, 20);
this.findTextBox.TabIndex = 6;
this.findTextBox.Text = "";
//
// buttonFind
```

```
//
this.buttonFind.Location = new System.Drawing.Point(88, 16);
this.buttonFind.Name = "buttonFind";
this.buttonFind.Size = new System.Drawing.Size(48, 23);
this.buttonFind.TabIndex = 7;
this.buttonFind.Text = "Find";
this.buttonFind.Click += new System.EventHandler(this.buttonFind_Click);
//
// buttonAdd
//
this.buttonAdd.Location = new System.Drawing.Point(8, 16);
this.buttonAdd.Name = "buttonAdd";
this.buttonAdd.Size = new System.Drawing.Size(40, 23);
this.buttonAdd.TabIndex = 8;
this.buttonAdd.Text = "Add";
this.buttonAdd.Click += new System.EventHandler(this.buttonAdd_Click);
//
// dGforTable
//
this.dGforTable.DataMember = "";
this.dGforTable.HeaderForeColor = System.Drawing.SystemColors.ControlText;
this.dGforTable.Location = new System.Drawing.Point(8, 24);
this.dGforTable.Name = "dGforTable";
this.dGforTable.Size = new System.Drawing.Size(272, 120);
this.dGforTable.TabIndex = 9;
//
// buttonAcc
//
this.buttonAcc.Location = new System.Drawing.Point(8, 152);
this.buttonAcc.Name = "buttonAcc";
this.buttonAcc.Size = new System.Drawing.Size(40, 23);
this.buttonAcc.TabIndex = 10;
this.buttonAcc.Text = "Acc";
this.buttonAcc.Click += new System.EventHandler(this.buttonAcc_Click);
//
// groupBox1
//
this.groupBox1.Controls.Add(this.buttonAcc);
this.groupBox1.Controls.Add(this.dGforTable);
this.groupBox1.Location = new System.Drawing.Point(6, 8);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(298, 184);
this.groupBox1.TabIndex = 11;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Таблица как она есть  ";
//
// groupBox2
//
this.groupBox2.Controls.Add(this.buttPrev);
this.groupBox2.Controls.Add(this.buttonFind);
this.groupBox2.Controls.Add(this.buttFirst);
this.groupBox2.Controls.Add(this.buttLast);
this.groupBox2.Controls.Add(this.demoTextBox);
this.groupBox2.Controls.Add(this.buttNext);
this.groupBox2.Controls.Add(this.dG);
this.groupBox2.Controls.Add(this.buttonAdd);
this.groupBox2.Controls.Add(this.findTextBox);
this.groupBox2.Location = new System.Drawing.Point(8, 200);
this.groupBox2.Name = "groupBox2";
this.groupBox2.Size = new System.Drawing.Size(296, 216);
this.groupBox2.TabIndex = 12;
this.groupBox2.TabStop = false;
this.groupBox2.Text = "Вид через вьюер";
//
// DataViewForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(312, 421);
this.Controls.Add(this.groupBox2);
this.Controls.Add(this.groupBox1);
this.Name = "DataViewForm";
```

```
this.Text = "DataViewForm";
((System.ComponentModel.ISupportInitialize)(this.dG)).EndInit();
((System.ComponentModel.ISupportInitialize)(this.dGforTable)).EndInit();
this.groupBox1.ResumeLayout(false);
this.groupBox2.ResumeLayout(false);
this.ResumeLayout(false);

}
#endregion

private void CreateTable()
{

// Создаётся таблица.
dt = new DataTable("Items");

// Столбцы таблицы...
// Имя первого столбца - id, тип значения - System.Int32.
c1 = new DataColumn("id", Type.GetType("System.Int32"));
c1.AutoIncrement=true;
// Имя второго столбца - Item, тип значения - System.Int32.
c2 = new DataColumn("Item", Type.GetType("System.Int32"));


// К таблице добавляются объекты-столбцы...
dt.Columns.Add(c1);
dt.Columns.Add(c2);


// А вот массив столбцов (здесь он из одного элемента)
// для организации первичного ключа (множества первичных ключей).
DataColumn[] keyCol= new DataColumn[1];

// И вот, собственно, как в таблице задаётся множество первичных ключей.
keyCol[0]= c1;
// Свойству объекта t передаётся массив, содержащий столбцы, которые
// формируемая таблица t будет воспринимать как первичные ключи.
dt.PrimaryKey=keyCol;

// В таблицу добавляется 10 rows.
for(int i = 0; i <10;i++)
{
       dr=dt.NewRow();
       dr["Item"]= i;
       dt.Rows.Add(dr);
}

// Принять изменения.
// Так производится обновление таблицы.
// Сведения о новых изменениях и добавлениях будут фиксироваться
// вплоть до нового обновления.
dt.AcceptChanges();

// Здесь мы применим специализированный конструктор, который
// задаст значения свойств Table, RowFilter, Sort, RowStateFilter
// объекта DataView в двух операторах кода...
//dv = new DataView(dt); // Вместо этого...
// Определение того, что доступно через объект-представитель DataView.
// Задавать можно в виде битовой суммы значений. И не все значения сразу!
// Сумма всех значений - противоречивое сочетание!
// А можно ли делать это по отдельности?
DataViewRowState dvrs = DataViewRowState.Added |
                                DataViewRowState.CurrentRows |
                                DataViewRowState.Deleted |
                                DataViewRowState.ModifiedCurrent |

                                //DataViewRowState.ModifiedOriginal |
                                //DataViewRowState.OriginalRows |
                                //DataViewRowState.None |
// Записи не отображаются.
                                DataViewRowState.Unchanged;
```

```
// Вот такое хитрое объявление...
//                 Таблица
//                  |   Значение, относительно которого будет вестись сортировка
//                  |   |   Имя столбца, по значениям которого производится сортировка
//                  |   |   |      составленное значение DataViewRowState.
//                  |   |   |      |
dv = new DataView(dt, "", "Item", dvrs);
}

private void buttNext_Click(object sender, System.EventArgs e)
{
if (currentCounter+1 < dv.Count) currentCounter++;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();
}


private void buttPrev_Click(object sender, System.EventArgs e)
{
if (currentCounter-1 >= 0) currentCounter--;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();
}

private void buttFirst_Click(object sender, System.EventArgs e)
{
currentCounter = 0;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();
}

private void buttLast_Click(object sender, System.EventArgs e)
{
currentCounter =dv.Count - 1;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();
}

private void dG_MouseDown(object sender, System.Windows.Forms.MouseEventArgs e)
{
currentCounter = dG.CurrentRowIndex;
rv = dv[currentCounter];
demoTextBox.Text = rv["Item"].ToString();
}

// Реализация поиска требует специального опеделения порядка
// сортировки строк, который должен задаваться в конструкторе.
private void buttonFind_Click(object sender, System.EventArgs e)
{
int findIndex = -1;

// Сначала проверяем строку на соответствие формату отыскиваемого значения.
// В нашем случае строка должна преобразовываться в целочисленное значение.
try
{
      int.Parse(findTextBox.Text);
}
catch
{
      findTextBox.Text = "Неправильно задан номер...";
      return;
}

findIndex = dv.Find(findTextBox.Text);

if (findIndex >= 0)
{
      currentCounter = findIndex;
      rv = dv[currentCounter];
      demoTextBox.Text = rv["Item"].ToString();
}
```

```
else
{
       findTextBox.Text = "Не нашли.";
}
}

private void buttonAdd_Click(object sender, System.EventArgs e)
{
// При создании новой записи средствами Вьюера таблицы,
// связанный с ним Вьюер строки переходит в состояние rv.IsNew.
// При этом в действиях этих объектов есть своя логика.
// И если туда не вмешиваться, при создании очередной записи
// предыдущая запись считается принятой и включается в таблицу автоматически.
// Контролируя состояния Вьюера строки (rv.IsEdit || rv.IsNew),
// мы можем предотвратить процесс поаледовательного автоматического
// обновления таблицы. Всё под контролем.
if (rv.IsEdit || rv.IsNew) return;
rv = dv.AddNew();
rv["Item"] = dv.Count-1;
}

private void buttonAcc_Click(object sender, System.EventArgs e)
{
// И вот мы вмешались в процесс.
// Добавление новой записи в таблицу становится возможным лишь
// после явного завершения редактирования прпредыдущей записи.
// Без этого попытки создания новой записи блокируются.
// Завершить редактирование.
rv.EndEdit();
// Принять изменения.
// Так производится обновление таблицы.
// Сведения о новых изменениях и добавлениях будут фиксироваться
// вплоть до нового обновления.
dt.AcceptChanges();
}
}
}
```

DataSet

DataSet - это находящийся в памяти объект ADO.NET, используемый для представления данных; он определяет согласованную реляционную модель программирования, не зависящую от источника содержащихся в нем данных. Объект DataSet представляет полный набор данных, включая таблицы, содержащие данные в определенном порядке с учетом установленных ограничений, а также отношения между таблицами.

В рамках отсоединённой модели ADO.NET объект DataSet становится важным элементом при поддержке разъединенных, распределенных сценариев данных с ADO.NET. Его большие функциональные возможности позволяют загрузить в хранилище данные из любого допустимого для ADO.NET источника: SQL Server, Microsoft Access, XML-файл. Содержащуюся в нём информацию можно изменять независимо от источника данных (от самой БД). Соответствующие значения формируются непосредственно в программе и добавляются в таблицы, связанные с хранилищем.

При работе с базой данных данные могут собираться из разных таблиц, локальное представление которых обеспечивается различными объектами представителями классов DataSet. В классе DataSet определено множество перегруженных методов Merge, которые позволяют объединять содержимое нескольких объектов DataSet.

Любой объект-представитель класса DataSet позволяет организовать чтение и запись содержимого (теоретически – информации из базы) в файл или область памяти. При этом можно читать и сохранять:

- только содержимое объекта (собственно информацию из базы),
- только структуру объекта-представителя класса DataSet,
- полный образ DataSet (содержимое и структуру).

Таким образом, DataSet является основой для построения различных вариантов отсоединённых объектов-хранилищ информации.

Класс DataSet – класс не абстрактный и не интерфейс. Это значит, что существует множество вариантов построения отсоединённых хранилищ.

На основе базового класса DataSet можно определять производные классы определённой конфигурации, которая соответсвует структуре базы данных.

Можно также создать объект-представитель класса DataSet. И добавить непосредственно к этому объекту все необходимые составляющие в виде таблиц-объектов-представителей класса Table соответствующей структуры и множества отношений Relation.

Объект DataSet изолирован от источников данных. Его назначение – представление в рамках приложения необходимого для решения поставленной задачи набора данных, включая таблицы и ограничения. Объект DataSet способен хранить информацию из нескольких источников. Он существует и функционирует исключительно за счёт объекта DataAdapter, который обслуживает DataSet.

Итак, это центральный компонент архитектуры доступа, основанный на отсоединённых данных в приложении представляет данные. На него работают все ранее перечисленные компоненты ADO.NET.

Объект-представитель DataSet ПРЕДСТАВЛЯЕТ ТАБЛИЦЫ. Таблицы любой конфигурации (в рамках допустимого). И для этого в DataSet'е есть всё необходимое. В числе его данных-членов имеется набор Tables. Таким образом, объект DataSet может содержать таблицы, количество которых ограничивается лишь возможностями набора Tables. Для каждой таблицы-элемента набора Tables может быть (и, естественно, должна быть) определена структура таблицы. В случае, когда приложение взаимодействует с реальной базой данных, структура таблиц в DataSet'е должна соответствовать структуре таблиц в базе данных.

Но этот объект и сам по себе, без сопутствующего окружения, представляет определённую ценность. Дело в том, что информация, представляемая в приложении в виде таблиц, НЕ ОБЯЗЯТЕЛЬНО должна иметь внешний источник в виде реальной базы данных. Ничто не мешает программисту обеспечить в приложении чтение обычного "плоского" файла или даже "накопить" необходимую информацию посредством интерактивного взаимодействия с пользователем, используя при этом обычный диалог. В конце концов, база данных – это один из возможных способов ОРГАНИЗАЦИИ информации (а не только её хранения!). Не случайно DataSet – это ОТСОЕДИНЁННЫЕ данные!

А если всё же требуется представление данных из реальной БД – DataAdapter поможет.

Структура класса DataSet

База Данных характеризуется множеством таблиц и множеством отношений между таблицами.

DataSet (как объявление класса) включает:

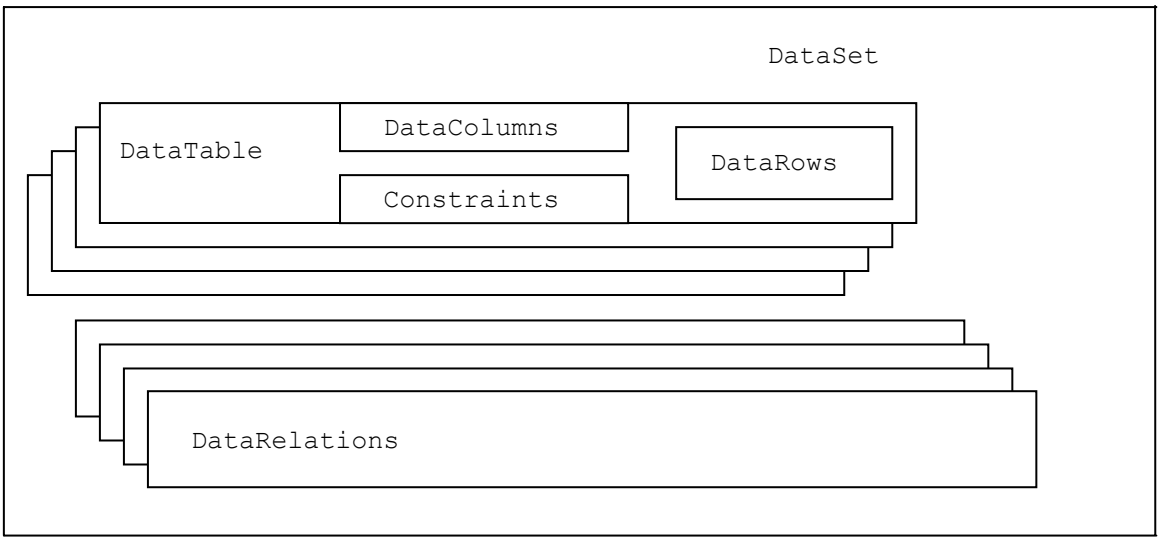
- набор (возможно, что пустой) объявлений классов DataTable (фактически это описание структуры составляющих данный КЛАСС DataSet таблиц),
- набор объявлений классов DataRelations, который обеспечивает установку связей между разными таблицами в рамках данного DataSet.

Структура DataSet может в точности повторять структуру БД и содержать полный перечень таблиц и отношений, а может быть частичной копией БД и содержать, соответственно, лишь подмножество таблиц и подмножество отношений. Всё определяется решаемой с помощью данного DataSet задачей.

При всём при этом, следует иметь в виду, что DataSet, структура которого полностью соответствует структуре БД (ИДЕАЛЬНАЯ DataSet), никогда не помешает решению поставленной задачи. Даже если будет содержать излишнюю информацию.

Процесс объявления (построения) класса DataSet, который ПОДОБЕН структуре некоторой базе данных, является достаточно сложным и трудоёмким. Класс-структурная копия БД содержит множество стандартных и трудных для ручного воспроизведения объявлений. Как правило, его объявление строится с использованием специальных средств-волшебников, которые позволяют оптимизировать и ускорить процесс воспроизведения структуры БД средствами языка программирования. В конечном счёте, появляется всё то же объявление класса, над которым также можно медитировать и (осторожно!) изменять его структуру.

Объект-представитель данного класса DataSet обеспечивает представление представление в памяти компа фрагмента данной БД. Этот объект является локальным представлением (фрагмента) БД.



Члены класса DataSet представлены ниже.

Открытые конструкторы

DataSet - конструктор	Перегружен. Инициализирует новый экземпляр класса DataSet.

Открытые свойства

CaseSensitive	Возвращает или задает значение, определяющее, учитывается ли регистр при сравнении строк в объектах DataTable.
Container (унаследовано от MarshalByValueComponent)	Получает контейнер для компонента.
DataSetName	Возвращает или задает имя текущего DataSet.
DefaultViewManager	Возвращает новое представление данных класса DataSet для осуществления фильтрации, поиска или перехода с помощью настраиваемого класса DataViewManager.
DesignMode (унаследовано от MarshalByValueComponent)	Получает значение, указывающее, находится ли компонент в настоящий момент в режиме разработки.
EnforceConstraints	Возвращает или задает значение, определяющее соблюдение правил ограничения при попытке совершения операции обновления.
ExtendedProperties	Возвращает коллекцию настраиваемых данных пользователя, связанных с DataSet.
HasErrors	Возвращает значение, определяющее наличие ошибок в любом из объектов DataTable в классе DataSet.
Locale	Возвращает или задает сведения о языке, используемые для сравнения строк таблицы.
Namespace	Возвращает или задает пространство имен класса DataSet.
Prefix	Возвращает или задает префикс XML, который является псевдонимом пространства имен класса DataSet.
Relations	Возвращает коллекцию соотношений, связывающих таблицы и позволяющих переходить от родительских таблиц к дочерним.
Site	Переопределен. Возвращает или задает тип System.ComponentModel.ISite для класса DataSet.
Tables	Возвращает коллекцию таблиц класса DataSet.

Открытые методы

AcceptChanges	Сохраняет все изменения, внесенные в класс DataSet после его загрузки или после последнего вызова метода AcceptChanges.
Clear	Удаляет из класса DataSet любые данные путем удаления всех строк во всех таблицах.

Clone	Копирует структуру класса DataSet, включая все схемы, соотношения и ограничения объекта DataTable. Данные не копируются.
Copy	Копирует структуру и данные для класса DataSet.
Dispose (унаследовано от MarshalByValueComponent)	Перегружен. Освобождает ресурсы, использовавшиеся объектом MarshalByValueComponent.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetChanges	Перегружен. Возвращает копию класса DataSet, содержащую все изменения, внесенные после его последней загрузки или после вызова метода AcceptChanges.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetService (унаследовано от MarshalByValueComponent)	Получает реализацию объекта IServiceProvider.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
GetXml	Возвращает XML-представление данных, хранящихся в классе DataSet.
GetXmlSchema	Возвращает XSD-схему для XML-представление данных, хранящихся в классе DataSet.
HasChanges	Перегружен. Возвращает значение, определяющее наличие изменений в классе DataSet, включая добавление, удаление или изменение строк.
InferXmlSchema	Перегружен. Применяет XML-схему к классу DataSet.
Merge	Перегружен. Осуществляет слияние указанного класса DataSet, DataTable или массива объектов DataRow с текущим объектом DataSet или DataTable.
ReadXml	Перегружен. Считывает XML-схему и данные в DataSet.
ReadXmlSchema	Перегружен. Считывает XML-схему в DataSet.
RejectChanges	Отменяет все изменения, внесенные в класс DataSet после его создания или после последнего вызова метода DataSet.AcceptChanges.
Reset	Сбрасывает DataSet в исходное состояние. Для восстановления исходного состояния класса DataSet необходимо переопределить метод Reset в подклассах.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.
WriteXml	Перегружен. Записывает XML-данные и по возможности схемы из DataSet.
WriteXmlSchema	Перегружен. Записывает структуру класса DataSet в виде XML-схемы.

Открытые события

Disposed (унаследовано от MarshalByValueComponent)	Добавляет обработчик событий, чтобы воспринимать событие Disposed на компоненте.
MergeFailed	Возникает, если значения первичного ключа конечного и основного объектов DataRow совпадают, а свойство EnforceConstraints имеет значение true.

Защищенные конструкторы

DataSet - конструктор	Перегружен. Инициализирует новый экземпляр класса DataSet.

Защищенные свойства

Events (унаследовано от MarshalByValueComponent)	Получает список обработчиков событий, которые подключены к этому компоненту.

Защищенные методы

Dispose (унаследовано от MarshalByValueComponent)	Перегружен. Освобождает ресурсы, использовавшиеся объектом MarshalByValueComponent.
Finalize (унаследовано от Object)	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации

	используется синтаксис деструктора.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.
OnPropertyChanging	Вызывает событие OnPropertyChanging.
OnRemoveRelation	Возникает при удалении объекта DataRelation из DataTable.
OnRemoveTable	Возникает при удалении объекта DataTable из DataSet.
RaisePropertyChanging	Посылает уведомление об изменении указанного свойства DataSet.
ShouldSerializeRelations	Возвращает значение, определяющее необходимость сохранения значения свойства Relations.
ShouldSerializeTables	Возвращает значение, определяющее необходимость сохранения значения свойства Tables.

Явные реализации интерфейса

System.ComponentModel.IListSource.ContainsListCollection

DataSet в свободном полёте

```
//=============================================================================

DataSet myDataSet = new DataSet(); // Пустой объект представитель класса DataSet.
DataTable myTable = new DataTable(); // Пустая таблица создана.
myDataSet.Tables.Add(myTable); // И подсоединена к объекту класса DataSet.
// Определение структуры таблицы. Это мероприятие можно было
// провести и до присоединения таблицы.
DataColumn shipColumn = new DataColumn("Ships");
myDataSet.Tables[0].Columns.Add(shipColumn);

// Прочие столбцы подсоединяются аналогичным образом.
// Таким образом формируются поля данных таблицы.
// Внимание! После того, как определена структура таблицы,
// то есть определены ВСЕ СТОЛБЦЫ таблицы, от имени этой конкретной
// таблицы порождается объект-строка. Этот объект сразу располагается
// непосредственно в таблице. Для каждой определённой таблицы
// метод NewRow() порождает строку
// (последовательность значений соответствующего типа). Для непосредственного
// редактирования вновь созданной строки запоминается её ссылка. Работать
// со строкой через эту ссылку проще, чем с массивом строк таблицы.
DataRow myRow = myDataSet.Tables[0].NewRow();

// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

// Остаётся заполнить строку таблицы содержательной информацией.
// При этом может быть использован любой источник данных.
// В данном примере предполагается наличие объекта типа ArrayList
// с именем ShipCollection.
for (int i = 0; i < ShipCollection.Count; i++)
{
myRow.Item[Counter] = ShipCollection[i];
}

// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
// Заполненный объект-представитель класса DataRow добавляется к набору Rows
// класса DataTable.
myDataSet.Tables[0].Rows.Add(myRow);

//=============================================================================
```

Естественно, что данная последовательность действий может повторяться сколь угодно много раз, пока не будут созданы и не заполнены все члены объекта DataSet.

Далее будет рассмотрен пример использования объекта DataSet для прочтения информации из текстового файла. При этом используется метод класса string Split(), который обеспечивает выделение из исходной строки подстрок с их последующим преобразованием в символьные массивы. Критерием выделения подстроки является массив символов-разделителей или целочисленное значение, определяющее максимальную длину подстроки.

Применение класса DataSet

```
using System;
using System.Data;

namespace mergeTest
{
class Class1
{
static void Main(string[] args)
{
// Создаётся объект DataSet.
DataSet ds = new DataSet("myDataSet");

// Создаётся таблица.
DataTable t = new DataTable("Items");

// Столбцы таблицы - это особые объекты.
// Имя первого столбца - id, тип значения - System.Int32.
DataColumn c1 = new DataColumn("id", Type.GetType("System.Int32"));
c1.AutoIncrement=true;
// Имя второго столбца - Item, тип значения - System.Int32.
DataColumn c2 = new DataColumn("Item", Type.GetType("System.Int32"));


// Сборка объекта DataSet:
// Добавляются объекты-столбцы...
t.Columns.Add(c1);
t.Columns.Add(c2);


// А вот массив столбцов (здесь он из одного элемента)
// для организации первичного ключа (множества первичных ключей).
DataColumn[] keyCol= new DataColumn[1];

// И вот, собственно, как в таблице задаётся множество первичных ключей.
keyCol[0]= c1;
// Свойству объекта t передаётся массив, содержащий столбцы, которые
// формируемая таблица t будет воспринимать как первичные ключи.
t.PrimaryKey=keyCol;
// А что с этими ключами будет t делать? А это нас в данный момент
// не касается. Очевидно, что методы, которые обеспечивают контроль
// над информацией в соответствии со значениями ключей уже где-то
// "зашиты" в классе DataTable. Как и когда они будут выполняться -
// не наше дело. Наше дело - указать на столбцы, которые для данной
// таблицы будут ключевыми. Что мы и сделали.

// Таблица подсоединяется к объекту ds представителю класса DataSet.
ds.Tables.Add(t);

DataRow r;

// В таблицу, которая уже присоединена к
// объекту ds DataSet добавляется 10 rows.
for(int i = 0; i <10;i++)
{
r=t.NewRow();
r["Item"]= i;
t.Rows.Add(r);
}

// Принять изменения.
// Так производится обновление DataSet'a.
// Сведения о новых изменениях и добавлениях будут фиксироваться
// вплоть до нового обновления.
ds.AcceptChanges();
PrintValues(ds, "Original values");

// Изменение значения в первых двух строках.
t.Rows[0]["Item"]= 50;
```

```
t.Rows[1]["Item"]= 111;
t.Rows[2]["Item"]= 111;

// Добавление ещё одной строки.
// Судя по всему, значение первого столбца устанавливается автоматически.
// Это ключевое поле со значением порядкового номера строки.
r=t.NewRow();
r["Item"]=74;
t.Rows.Add(r);


// Объявляем ссылку для создания временного DataSet.
DataSet xSet;

// ДЕКЛАРАЦИЯ О НАМЕРЕНИЯХ КОНТРОЛЯ ЗА КОРРЕКТНОСТЬЮ ЗНАЧЕНИЙ СТРОКИ.
// Вот так добавляется свойство, содержащее строку для описания ошибки в значении.
// Наш DataSet содержит одну строку с описанием.
// Это всего лишь указание на то обстоятельство, что МЫ САМИ обязались осуществлять
// некоторую деятельность по проверке чего-либо. Чтобы не забыть, в чём проблема,
// описание возможной ошибки (в свободной форме!) и добавляем в свойства строки,
// значения которой требуют проверки.
t.Rows[0].RowError= "over 100 (ЙЦУКЕН!)";
t.Rows[1].RowError= "over 100 (Stupid ERROR!)";
t.Rows[2].RowError= "over 100 (Ну и дела!)";
// Но одно дело декларировать намерения, а другое – осуществлять контроль.
// Проблема проверки корректности значения – наша личная проблема.
// Однако о наших намерениях контроля за значениями становится известно
// объекту-представителю DataSet!

PrintValues(ds, "Modified and New Values");

// Мы вроде бы согласились проводить контроль значений.
// Даже декларировали некий принцип проверки.
// Однако, ничего само собой не происходит.
// Так вот,
//
// ЕСЛИ В ТАБЛИЦУ БЫЛИ ДОБАВЛЕНЫ СТРОКИ ИЛИ ИЗМЕНЕНЫ ЗНАЧЕНИЯ СТРОК
// И
// МЫ ОБЯЗАЛИСЬ КОНТРОЛИРОВАТЬ ЗНАЧЕНИЯ СТРОК В ТАБЛИЦЕ
//
// , то самое время организовать эту проверку...
// Критерий правильности значений, естественно, наш!
// Алгоритмы проверки - тоже НАШИ!
// Единственное, чем нам может помочь ADO.NET - это выделить
// подмножество строк таблицы,
// которые были добавлены или модифицированы со времени последнего
// обновления нашего объекта - представителя DataSet'a,

if(ds.HasChanges(DataRowState.Modified | DataRowState.Added)& ds.HasErrors)
{
// И для этого мы воспользуемся методом, который позволяет обеспечить выделение
// подмножества добавленных и модифицированных строк в новый объект DataSet'a.
// Use GetChanges to extract subset.
xSet = ds.GetChanges(DataRowState.Modified|DataRowState.Added);
PrintValues(xSet, "Subset values");
// Insert code to reconcile errors. In this case, we'll reject changes.
// Вот, собственно, код проверки. Всё делается своими руками.
foreach(DataTable xTable in xSet.Tables)
{
if (xTable.HasErrors)
{
foreach(DataRow xRow in xTable.Rows)
{
// Выделенное подмножество проверяем на наличие
// ошибочного значения (для нас всё, что больше 100 - уже ошибка!)
Console.Write(xRow["Item"] + "  ");
if((int)xRow["Item",DataRowVersion.Current ]> 100)
{
       // Находим ошибку в строке, сообщаем о ней,
       Console.WriteLine("Error! - " + xRow.RowError);
       // Возвращаем старое значение...
```

```
        xRow.RejectChanges();
        // Отменяем значение свойства-уведомителя о возможных
        // ошибках для данной строки...
        xRow.ClearErrors();
}
else
Console.WriteLine("OK.");
}
}
}

PrintValues(xSet, "Reconciled subset values");

// Сливаем изменённые и откорректированные строки в основной объект - DataSet
// Merge changes back to first DataSet.
ds.Merge(xSet);

PrintValues(ds, "Merged Values");
}
}

// А это всего лишь вывод содержимого DataSet'a.
private static void PrintValues(DataSet ds, string label)
{
Console.WriteLine("\n" + label);
foreach(DataTable t in ds.Tables)
{
Console.WriteLine("TableName: " + t.TableName);
foreach(DataRow r in t.Rows)
{
foreach(DataColumn c in t.Columns)
{
Console.Write("\t " + r[c] );
}
Console.WriteLine();
}

}

}
}
}
```

Подсоединенные объекты объектной модели ADO.NET. Провайдеры

Поставщик данных для приложения (Провайдер) – объект, предназначенный для обеспечения взаимодействия приложения с хранилищем информации (базами данных).

Естественно, приложению нет никакого дела до того, где хранится и как извлекается потребляемая приложением информация. Для приложения источником данных является тот, кто передаёт данные приложению. И как сам этот источник эту информацию добывает – никого не касается.

Источник данных (Data Provider) – это набор взаимосвязанных компонентов, обеспечивающих доступ к данным. Функциональность и само существование провайдера обеспечивается набором классов, специально дла этой цели разработанных.

ADO.NET поддерживает два типа источников данных, соответственно, два множества классов:

- SQL Managed Provider (SQL Server.NET Data Provider) - для работы с Microsoft SQL Server 7.0 и выше. Работает по специальному протоколу, называемому TabularData Stream (TDS) и не использует ни ADO, ни ODBC, ни какую-либо еще технологию. Ориентированный специально на MS SQL Server, протокол позволяет увеличить скорость передачи данных и тем самым повысить общую производительность приложения.
- ADO Managed Provider (OleDb.NET Data Provider) - для всех остальных баз данных. Обеспечивает работу с произвольными базами данных. Однако за счет универсальности есть проигрыш по сравнению с SQL Server Provider, так что при работе с SQL Server рекомендовано использовать специализированные классы.

В следующих разделах приводится описание соствных элементов провайдера.

Connection

Объект-представитель класса Connection представляет соединение с источником (базой) данных и обеспечивает подключение к базе данных. Visual Studio.NET поддерживает два класса:

- SQLConnection (обеспечивает подключение к SQL Server 7.0 и выше),
- OleDbConnection (обеспечивает подключение к прочим вариантам БД).

Компонента Connection (независимо от того, представителем какого класса она является) имеет свойство ConnectionString, в котором фиксируется вся необходимая для установления соединения с БД информация. Кроме того, поддерживается ряд методов, позволяющих обрабатывать данные с применением транзакций.

Свойства объекта Connection позволяют:

- задавать реквизиты пользователя,
- указывать расположение источника данных.

Методы объекта позволяют управлять соединением с источником данных.

В процессе соединения с помощью объекта-представителя класса OleDbConnection (аналогично SQLConnection) создаётся и инициализируется соответствующий объект с использованием одного из вариантов конструктора и строки соединения.

Формирование строки и последовательность действий при инициализации объекта соединения – дело техники. Главное – это чтобы свойство ConnectionString в результате получило бы ссылку на строку символов, содержащую необходимую для установления соединения информацию.

```
// Объявили и определили объект соединения.
private System.Data.OleDb.OleDbConnection oleDbConnection1;
this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();

::::::::::

// Настроили объект соединения.
// Для наглядности необходимая для установления соединения
// информация представлена серией строк.
oleDbConnection1.ConnectionString =
@"Jet OLEDB:Global Partial Bulk Ops=2;" +
@"Jet OLEDB:Registry Path=;" +
@"Jet OLEDB:Database Locking Mode=1;" +
@"Data Source=""F:\Users\Work\CS\DB.BD\DBTests\Lights.mdb"";" +
@"Jet OLEDB:Engine Type=5;" +
@"Jet OLEDB:Global Bulk Transactions=1;" +
@"Provider=""Microsoft.Jet.OLEDB.4.0"";" + // Поставщик
@"Jet OLEDB:System database=;" +
@"Jet OLEDB:SFP=False;" +
@"persist security info=False;" +
@"Extended Properties=;" +
@"Mode=Share Deny None;" +
@"Jet OLEDB:Create System Database=False;" +
@"Jet OLEDB:Don't Copy Locale on Compact=False;" +
@"Jet OLEDB:Compact Without Replica Repair=False;" +
@"User ID=Admin;" +
@"Jet OLEDB:Encrypt Database=False";
```

Свойства, методы и события класса OleDbConnection

Свойства.

ConnectionString	string	Строка, определяющая способ подключения объекта к источнику данных.
ConnectionTimeout	Int32	Интервал времени, в течение которого объект пытается установить соединение с источником данных (только для чтения).
Container	string	Gets the IContainer that contains the Component.

Database	string	Gets текущей базы данных или базы, которая использовалась после установления соединения.
DataSource	string	Gets the server name or file name of the data source. Всё зависит от того, с каким хранилищем информации ведётся работа. Серверное хранилище данных (SQL Server, Oracle) – имя компа, выступающего в роли сервера. Файловые БД (Access) – имя файла.
Provider	string	Gets имя OLE DB провайдера, которое было объявлено в "Provider= …" clause строки соединения.
ServerVersion	string	Gets a string containing the version of the server to which the client is connected.
Site	string	Gets or sets the ISite of the Component.
State	string	Gets текущее состояние соединения.

Текущее состояние соединения кодируется как элемент перечисления ConnestionState. Список возможных значений представлен ниже.

Имя члена	Описание	Value
Broken	The connection to the data source is broken. Подобное может случиться только после того, как соединение было установлено. A connection in this state may be closed and then re-opened. (This value is reserved for future versions of the product.)	16
Closed	Соединение закрыто.	0
Connecting	The connection object is connecting to the data source. (This value is reserved for future versions of the product.)	2
Executing	The connection object в процессе выполнения команды. (This value is reserved for future versions of the product.)	4
Fetching	Объект соединения занят выборкой данных. (This value is reserved for future versions of the product.)	8
Open	Соединение открыто.	1

Открытые методы

BeginTransaction	Перегружен. Начинает транзакцию базы данных.
ChangeDatabase	Изменяет текущую базу данных для открытого OleDbConnection.
Close	Закрывает подключение к источнику данных. Это рекомендуемый метод закрытия любого открытого подключения.
CreateCommand	Создает и возвращает объект OleDbCommand, связанный с OleDbConnection.
CreateObjRef (унаследовано от MarshalByRefObject)	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами.
Dispose (унаследовано от Component)	Перегружен. Освобождает ресурсы, используемые объектом Component.
EnlistDistributedTransaction	Зачисляет в указанную транзакцию в качестве распределенной транзакции.
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object.
GetHashCode (унаследовано от Object)	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.
GetLifetimeService (унаследовано от MarshalByRefObject)	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра.
GetOleDbSchemaTable	Возвращает сведения схемы из источника данных так же, как указано в GUID, и после применения указанных ограничений.
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра.
InitializeLifetimeService (унаследовано от MarshalByRefObject)	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра.
Open	Открывает подключение к базе данных со значениями свойств, определяемыми ConnectionString.
ReleaseObjectPool	Статический. Означает, что пул объектов OleDbConnection может быть освобожден, когда последнее основное подключение будет освобождено.
ToString (унаследовано от Object)	Возвращает String, который представляет текущий Object.

Защищенные методы

Dispose	Перегружен. Переопределен. Освобождает ресурсы, используемые объектом OleDbConnection.
Finalize (унаследовано от Component)	Переопределен. Освобождает неуправляемые ресурсы и выполняет другие действия по очистке, перед тем как пространство, которое использует Component, будет восстановлено сборщиком мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
GetService (унаследовано от Component)	Возвращает объект, представляющий службу, которую предоставляет Component или его Container.
MemberwiseClone (унаследовано от Object)	Создает неполную копию текущего Object.

События

Disposed	Adds an event handler to listen to the Disposed event on the component.
InfoMessage	Некоторые СУБД (SQL Server) поддерживают механизм информационных сообщений. Это событие occurs when the provider sends a warning or an informational message.
StateChange	Occurs when the state of the connection changes.

```
// Объявили ссылку…
private System.Data.OleDb.OleDbConnection RollsConnection;
// Определили объект…
this.RollsConnection = new System.Data.OleDb.OleDbConnection();
// Настроили строкой соединения…
this.RollsConnection.ConnectionString =
                        @"Jet OLEDB:Global Partial Bulk Ops=2;"
                        +"Jet OLEDB:Registry Path=;"
                        +"Jet OLEDB:Database Locking Mode=1;"
                        +@"Data Source=""F:\Users\WORK\RollsBase\Rolls.mdb"""
                        +";Jet OLEDB:Engine Type=5;"
                        +@"Provider=""Microsoft.Jet.OLEDB.4.0"";"
                        +"Jet OLEDB:System database=;"
                        +"Jet OLEDB:SFP=False;"
                        +"persist security info=False;"
                        +"Extended Properties=;"
                        +@"Mode=""ReadWrite|Share Deny None"";"
                        +"Jet OLEDB:Encrypt Database=False;"
                        +"Jet OLEDB:Create System Database=False;"
                        +"Jet OLEDB:Don't Copy Locale on Compact=False;"
                        +"Jet OLEDB:Compact Without Replica Repair=False;"
                        +"User ID=Admin;"
                        +"Jet OLEDB:Global Bulk Transactions=1";
```

Подключение к БД на этапе разработки приложения

Средствами MS Access создаётся простая база данных. После определения общей структуры базы, основных свойств её полей, база заполняется.

Для внешнего окружения - это всего лишь файл с расширением .mdb (в нашем случае F:\SharpUser\CS.book\Rolls.db).

Для того чтобы приложение работало с данным файлом как с базой данных, необходимо создать соединение с базой Rolls.

Создание соединения средствами встроенных мастеров предполагает следующую последовательность шагов в рамках Visual Studio.NET:

- открывается вкладка Server Explorer,
- с использованием инструмента “Connect to DataBase” открывается окошко “Data Link Properties”, которое предоставляет интерфейс для настройки соединения с базой данных. В окошке четыре вкладки: Provider, Connection, Advanced, All,
- на вкладке Provider выбирается провайдер соответствующего типа. Критерии выбора (пока) остаются неизвестными. Главное, что далеко не все провайдеры могут подключать нашу базу. Вкладки Connection, Advanced и All используются для настройки свойств соединения. В зависимости от типа провайдера в

качестве параметров могут быть указаны имя сервера, имя пользователя, пароль и т.д.,

- факт возможного успешного подключения проверяется на вкладке Connection с помощью кнопки “Test Connection”, которая запускает соответствующие методы тестирования. Провайдер Microsoft Jet 4.0 OLE DB Provider (при условии, что User name на вкладке Connection == Admin) успешно прошёл тестирование. На нём пока и остановимся.

Новое соединение появляется в окне Server Explorer’а в виде пиктограммы с раскрывающимся списком, который содержит информацию о свойствах соединения. После этого соединение надо переместить (“перетащить”) в окно дизайнера – прямо на форму. Возможно, что при этом придётся ответить на вопрос по поводу включения пароля в “connection string”. Всё.

Деятельность по добавлению к приложению соединения средствами мастера соединений на этом завершается. При этом в кодах приложения размещается соответствующий программный код, который обеспечивает создание объекта соединения соответствующего типа и с определёнными параметрами.

Внимательный анализ и медитация над этим программным кодом делает вполне возможной деятельность по самостоятельному созданию соединений в приложении без участия мастера соединений.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data; // Пространство имён для работы с базами данных.

namespace ADO01
{
/// <summary>
/// Summary description for Form1.
/// </summary>
public class Form1 : System.Windows.Forms.Form
{
// Объект, представляющий соответствующее соединение.
private System.Data.OleDb.OleDbConnection oleDbConnection1;
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
InitializeComponent();
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>

// this.oleDbConnection1.ConnectionString в InitializeComponent
// - это такая строка соединения. Все характеристики соединения
```

```
// в одном "@-quoted string literals"! В исходниках – вся информация
// размещается в одной строке.
private void InitializeComponent()
{
this.oleDbConnection1 = new System.Data.OleDb.OleDbConnection();
//
// oleDbConnection1
//
this.oleDbConnection1.ConnectionString = @"Jet OLEDB:Global Partial Bulk Ops=2; Jet
OLEDB:Registry Path=;Jet OLEDB:Database Locking Mode=1; Data
Source=""F:\SharpUser\CS.book\Rolls.db"";  Jet  OLEDB:Engine
Type=5;Provider=""Microsoft.Jet.OLEDB.4.0"";Jet OLEDB:System database=;Jet
OLEDB:SFP=False;persist security info=False;Extended Properties=;Mode=Share Deny
None;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Create System Database=False;Jet
OLEDB:Don't Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica
Repair=False;User ID=Admin;Jet OLEDB:Global Bulk Transactions=1";
//
// Form1
//
this.AutoScaleBaseSize = new System.Drawing.Size(6, 15);
this.ClientSize = new System.Drawing.Size(292, 268);
this.Name = "Form1";
this.Text = "Form1";
}
#endregion
/// <summary>
/// The main entry Point for the application.
/// </summary>
[STAThread]
static void Main()
{
Application.Run(new Form1());
}
}
}
```

Если подобная конструкция (соединение с БД) конструируется вручную, следует помнить о механизме конкатенации символьных строк и о том, что объём информации в строке соединения может быть минимизирован. Судя по всему, такой строки вполне достаточно:

```
this.oleDbConnection1.ConnectionString =
                                        @"Provider=Microsoft.Jet.OLEDB.4.0;" +
                                         "Data Source=" +
                                        @"F:\SharpUser\CS.book\Rolls.db";
```

Конечно же, следует упомянуть такую замечательную разновидность формы, как форма, которая строится с использованием помощника Data Form Wizard. Волшебник затачивает форму для работы с базой с минимальным необходимым набором элементов управления под конкретную базу. Со всем необходимым набором объектов, используемых для установления соединения.

Ручная сборка объекта Connection

В любом случае, СОЕДИНЕНИЕ - это:

- объект-представитель класса System.Data.OleDb.OleDbConnection, который объявляется в классе формы.

```
private System.Data.OleDb.OleDbConnection rollsConnection;
```

- , который создаётся, если сборка с помощью волшебника – в методе InitializeComponent(), если вручную – то (наверное) в конструкторе формы

```
this.rollsConnection = new System.Data.OleDb.OleDbConnection();
```

- после чего свойство ConnectionString данного объекта инициируется посредством строки соединения (имя поставщика, его версия, путь (возможно относительный) к базе)

```
this.rollsConnection.ConnectionString = @"Jet OLEDB:Global Partial Bulk Ops=2;Jet OLEDB:Registry Path=;Jet OLEDB:Database Locking Mode=1;Data Source=""F:\SharpUser\CS.book\Rolls.db\Rolls.mdb"";Jet OLEDB:Engine Type=5;Provider=""Microsoft.Jet.OLEDB.4.0"";Jet OLEDB:System database=;Jet OLEDB:SFP=False;persist security info=False;Extended Properties=;Mode=Share Deny None;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Create System Database=False;Jet OLEDB:Don't Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica Repair=False;User ID=Admin;Jet OLEDB:Global Bulk Transactions=1";
```

Следует также упомянуть альтернативный вариант конструктора, в котором строка соединения передаётся в виде параметра конструктору. Что-нибудь в таком стиле:

```
string cs = @"Jet OLEDB:Global Partial Bulk Ops=2;Jet OLEDB:Registry Path=;Jet OLEDB:Database Locking Mode=1;Data Source=""F:\SharpUser\CS.book\Rolls.db\Rolls.mdb"";Jet OLEDB:Engine Type=5;Provider=""Microsoft.Jet.OLEDB.4.0"";Jet OLEDB:System database=;Jet OLEDB:SFP=False;persist security info=False;Extended Properties=;Mode=Share Deny None;Jet OLEDB:Encrypt Database=False;Jet OLEDB:Create System Database=False;Jet OLEDB:Don't Copy Locale on Compact=False;Jet OLEDB:Compact Without Replica Repair=False;User ID=Admin;Jet OLEDB:Global Bulk Transactions=1";

this.rollsConnection = new System.Data.OleDb.OleDbConnection(sc);
```

Созданное таким образом соединение открывается,

```
rollsConnection.Open();
```

а затем закрывается

```
rollsConnection.Close();
```

Имитация отсоединенности. Пул соединений

В ADO.NET официально заявлено, что в приложении используются ОТСОЕДИНЕННЫЕ компоненты.

Присоединились, каким-то образом получили требуемую информацию, отсоединились… Но есть проблема. Открытие и закрытие соединения с БД операция трудоёмкая. Поддерживать соединение постоянно – плохо. Обрывать и восстанавливать соединение всякий раз по мере необходимости – тоже, получается плохо. Компромиссное решение – ПУЛ соединений. Место, где сохраняются установленные и неиспользуемые в данный момент соединения.

Приложение создаёт объект соединения, устанавливает соединение с базой, использует его и, не разрывая соединения с базой, передаёт его в ПУЛ соединений. В дальнейшем, по мере необходимости, объект соединения используется из пула. Если несколько приложений стремятся одновременно получить доступ к БД и в пуле не остаётся свободных соединений – создаётся и настраивается новый объект. Который после употребления приложением также передаётся в пул. Таким образом поддерживается несколько готовых к использованию соединений, общее количество которых (по крайней мере, теоретически) оказывается меньше общего количества приложений, работающих с базой в данный момент.

Пул включается по умолчанию. И при этом выполнение оператора

```
rollsConnection.Close();
```

Приводит НЕ К РАЗРЫВУ соединения с базой, а к передаче этого соединения в пул соединений. Для повторного использования. Запретить размещение объекта соединения в пуле можно, указав в строке соединения для OLE DB.NET атрибут

```
OLE DB Services = -4
```

При котором провайдер OLE DB.NET не будет помещать соединение в пул при закрытии, а будет всякий раз его разрывать и устанавливать заново.

Пул соединений освобождается при выполнении метода Dispose. Этот метод вызывается сборщиком мусора при звершении приложения. В теле этого метода обеспечивается вызов метода Close. Таким образом, даже незакрытое соединение при завершении приложения закрывается. Явный вызов этого метода также приводит к разрыву соединения, освобождению занимаемых ресурсов и подготовке объекта к уничтожению сборщиком мусора.

На самом деле, если класс предоставляет метод Dispose(), именно его, а не Close() следует вызывать для освобождения занимаемых объектом ресурсов.

Применение объекта соединения

Ну, хорошо. Разобрались с соединением. Научились его открывать и "закрывать".

А как же его использовать…

Например, установив соединение с базой данных, можно исследовать её схему. Эта возможность обеспечивается объявленным в классе соединения методом GetOleDbSchemaTable. Ранее этот метод приводился в списке методов класса соединения. Управление процессом сбора информации о схеме базы обеспечивается двумя параметрами метода.

Первый параметр представлен статическими свойствами класса OleDbSchemaGuid.

Второй параметр – массивом (кортежем) объектов, каждый их которых в зависимости от занимаемой позиции в кортеже может принимать определённое фиксированное значение, либо иметь значение null.

Класс OleDbSchemaGuid. Определение схемы базы
Namespace: System.Data.OleDb

При вызове метода GetOleDbSchemaTable значение-параметр типа OleDbSchemaGuid обеспечивает возвращение типа схемы таблицы.

Информация о структуре класса представлена ниже.

Следует иметь в виду, что не все провайдеры полностью поддерживают предоставляемые классом возможности по определению схемы таблицы.

Члены класса OleDbSchemaGuid.

Открытые конструкторы

OleDbSchemaGuid - конструктор	Инициализирует новый экземпляр класса OleDbSchemaGuid.

Открытые поля

Assertions	Статический. Возвращает утверждения, определенные в каталоге, владельцем которого является указанный пользователь.
Catalogs	Статический. Возвращает физические атрибуты, связанные с каталогами, доступными из источника данных. Возвращает утверждения, определенные в каталоге и принадлежащие указанному пользователю.
Character_Sets	Статический. Возвращает наборы символов, определенные в каталоге и доступные указанному пользователю.
Check_Constraints	Статический. Возвращает ограничения проверки, определенные в каталоге и принадлежащие указанному пользователю.
Check_Constraints_By_Table	Статический. Возвращает ограничения проверки, определенные в каталоге и принадлежащие указанному пользователю.
Collations	Статический. Статический. Возвращает сравнения знаков, определенные в каталоге и доступные указанному пользователю.
Columns	Статический. Возвращает столбцы таблиц (включая представления), определенные в каталоге и доступные указанному пользователю.
Column_Domain_Usage	Статический. Возвращает столбцы, определенные в каталоге и зависящие от домена, определенного в каталоге и принадлежащего указанному пользователю.
Column_Privileges	Статический. Возвращает привилегии для столбцов таблиц, определенные в каталоге и доступные указанному пользователю или предоставленные им.
Constraint_Column_Usage	Статический. Возвращает столбцы, используемые ссылочными ограничениями, уникальными ограничениями и

	утверждениями, определенными в каталоге и принадлежащими указанному пользователю.
Constraint_Table_Usage	Статический. Возвращает таблицы, используемые ссылочными ограничениями, уникальными ограничениями и утверждениями, определенными в каталоге и принадлежащими указанному пользователю.
DbInfoLiterals	Статический. Возвращает список литералов, используемых в текстовых командах и специфичных для конкретного поставщика.
Foreign_Keys	Статический. Возвращает столбцы внешнего ключа, определенные в каталоге данным пользователем.
Indexes	Статический. Возвращает индексы, определенные в каталоге и принадлежащие указанному пользователю.
Key_Column_Usage	Статический. Возвращает столбцы, определенные в каталоге и ограниченные как ключи данным пользователем.
Primary_Keys	Статический. Возвращает столбцы первичного ключа, определенные в каталоге данным пользователем.
Procedures	Статический. Возвращает процедуры, определенные в каталоге и принадлежащие указанному пользователю.
Procedure_Columns	Статический. Возвращает сведения о столбцах наборов строк, возвращаемых процедурами.
Procedure_Parameters	Статический. Возвращает сведения о параметрах и кодах возврата процедур.
Provider_Types	Статический. Возвращает основные типы данных, поддерживаемые поставщиком данных .NET Framework для OLE DB.
Referential_Constraints	Статический. Возвращает ссылочные ограничения, определенные в каталоге и принадлежащие указанному пользователю.
Schemata	Статический. Возвращает объекты схемы, принадлежащие указанному пользователю.
Sql_Languages	Статический. Возвращает уровни соответствия, параметры и диалекты, поддерживаемые данными обработки с помощью реализации SQL.
Statistics	Статический. Возвращает статистические данные, определенные в каталоге и принадлежащие указанному пользователю.
Tables	Статический. Возвращает таблицы (включая представления), определенные в каталоге и доступные указанному пользователю.
Tables_Info	Статический. Возвращает таблицы (включая представления), доступные указанному пользователю.
Table_Constraints	Статический. Возвращает табличные ограничения, определенные в каталоге и принадлежащие указанному пользователю.
Table_Privileges	Статический. Возвращает привилегии для таблиц, определенные в каталоге и доступные указанному пользователю или предоставленные им.
Table_Statistics	Статический. Описывает доступный набор статистических данных по таблицам для поставщика.
Translations	Статический. Возвращает переводы знаков, определенные в каталоге и доступные указанному пользователю.
Trustee	Статический. Определяет доверенные объекты, заданные в источнике данных.
Usage_Privileges	Статический. Возвращает привилегии USAGE для объектов, определенные в каталоге и доступные указанному пользователю или предоставленные им.
Views	Статический. Возвращает представления, определенные в каталоге и доступные указанному пользователю.
View_Column_Usage	Статический. Возвращает столбцы, от которых зависят просматриваемые таблицы, определенные в каталоге и принадлежащие данному пользователю.
View_Table_Usage	Статический. Возвращает таблицы, от которых зависят просматриваемые таблицы, определенные в каталоге и принадлежащие данному пользователю.

Открытые методы

Equals	Перегружен. Определяет, равны ли два экземпляра Object.
GetHashCode	Служит хеш-функцией для конкретного типа, пригоден для использования в алгоритмах хеширования и структурах данных, например в хеш-таблице.

GetType	Возвращает Type текущего экземпляра.
ToString	Возвращает String, который представляет текущий Object.
Защищенные методы	
Finalize	Переопределен. Позволяет объекту Object попытаться освободить ресурсы и выполнить другие завершающие операции, перед тем как объект Object будет уничтожен в процессе сборки мусора. В языках C# и C++ для функций финализации используется синтаксис деструктора.
MemberwiseClone	Создает неполную копию текущего Object.

Ниже представлен фрагмент кода, позволяющий прочитать информацию о схеме базы.

```
// Применение метода GetOleDbSchemaTable.
// Получение информации о схеме базы данных.
// При этом можно использовать параметр Restrictions,
// с помощью которого можно фильтровать возвращаемые сведения
// о схеме.
private void schemaButton_Click(object sender, System.EventArgs e)
{
// Точной информацией о том, что собой представляют остальные
// члены множества Restrictions пока не располагаю.
object[] r;
r = new object[] {null, null, null, "TABLE"};
oleDbConnection.Open();
DataTable tbl;
tbl=oleDbConnection.GetOleDbSchemaTable(System.Data.OleDb.OleDbSchemaGuid.Tables, r);
rpDataGrid.DataSource = tbl;
oleDbConnection1.Close();
}
```

Таким образом получается информация о схеме базы, связь с которой устанавливается через объект соединения.

Но основная область применения объекта соединения – в составе команды.

Отступление о запросах

1. Запросы, которые не возвращают записей (action query или КОМАНДНЫЕ ЗАПРОСЫ). Различаются:

запросы обновления или Data Manupulation Language queries. Предназначаются для изменения содержимого базы данных

```
UPDATE Customers
Set    CompanyName = 'NewHappyName'
WHERE  CustomerID = '007'

INSERT INTO Customers (CustomerID, CompanyName)
VALUES                ('007', 'NewHappyCustomer')

DELETE FROM Customers
WHERE  CustomerID = '007'
```

2. запросы изменения или Data Definition Language queries. Предназначены для изменения структуры базы данных

```
CREATE TABLE myTable
                   (
                     Field1 int NOT NULL
                     Field2 varchar()
                    )
```

3. Запросы, возвращающие значения из базы данных. Ниже представлены три примера запросов.

Возвращает значения полей для всех записей, представленных в таблице Customers.

```
SELECT
       CustomerID,
       CompanyName,
       ContactName,
       Phone
FROM
       Customers
```

Возвращает значения полей для записей, представленных в таблице Customers, у которых значение поля Phone равно строке ‘333-2233’.

```
SELECT
       CustomerID,
       CompanyName,
       ContactName
FROM
       Customers
WHERE
       Phone = ‘222-3322’
```

Параметризованный запрос. Множество возвращаемых значений зависит от значения параметра, стандартно обозначаемого маркером ‘?’ и замещаемого непосредственно при выполнении запроса.

```
SELECT
       CompanyName,
       ContactName,
       Phone
FROM
       Customers
WHERE
      CustomerID = ?
```

Command

Объект Команда – стартовый стол для запуска непосредственно из приложения команд управления БД, которыми и осуществляется непосредственное управление БД. Команда в приложении обеспечивает взаимодействие приложения с базой данных, позволяя при этом:

- сохранять параметры команд, которые используются для управления БД,
- выполнять специфические команды БД INSERT, UPDATE, DELETE, которые не возвращают значений,
- выполнять команды, возвращающие единственное значение,
- выполнять команды специального языка определения баз данных DataBase Definition Language (DDL), например CREATE TABLE,
- работать с объектом DataAdapter, возвращающим объект DataSet,
- работать с объектом DataReader,
- для класса SqlCommand – работать с потоком XML,
- создавать результирующие наборы, построенные на основе нескольких таблиц или в результате исполнения нескольких операторов.

Объект Command обеспечивает управление источником данных, которое заключается:

- в выполнении DML (Data Manipulation Langauge) запросов – запросов, не возвращающих данные (INSERT, UPDATE, DELETE),
- в выполнении DDL (Data Definition Language) запросов – запросов, которые изменяют структуру Базы Данных (CREATE)
- в выполнении запросов, возвращающих данные через объект DataReader (SELECT).
- Представлен двумя классами – SqlCommand и OleDbCommand. Позволяет исполнять команды на БД и при этом использует установленное соединение. Исполняемые команды могут быть представлены:
- хранимыми процедурами,
- командами SQL,

- операторами, возвращающими целые таблицы.

Объект-представитель класса Command поддерживает два варианта (варианты методов определяются базовым классом) методов:

- ExecuteNonQuery – обеспечивает выполнение команд, не возвращающих данные, например, INSERT, UPDATE, DELETE,
- ExecuteScalar – исполняет запросы к БД, возвращающие единственное значение,
- ExecuteReader – возвращает результирующий набор через объект DataReader.

Доступ к данным в ADO.NET с помощью Data Provider'а осуществляется следующим образом:

- Объект-представитель класса Connection устанавливает соединение между БД и приложением.
- Это соединение становится доступным объектам Command и DataAdapter.
- При этом объект Command позволяет исполнять команды непосредственно над БД.
- Если исполняемая команда возвращает несколько значений, Command открывает доступ к ним через объект DataReader.
- Результаты выполнения команды обрабатываются либо напрямую, с использованием кода приложения, либо через объект DataSet, который заполняется при помощи объекта DataAdapter.
- Для обновления БД применяют также объекты Command и DataAdapter.

Итак, в любом случае, независимо от выбранного поставщика данных, при работе с данными в ADO.NET используем:

- Connection Object – для установки соединения с базой данных,
- Dataset Object – для представления данных на стороне приложения,
- Command Object – для изменения состояния базы.

Способы создания объекта Command:

- С использованием конструкторов и с последующей настройкой объекта (указание строки запроса и объекта Connection),
- Вызов метод CreateCommand объекта Connection.

```
private void readButton_Click(object sender, System.EventArgs e)
{
int i = 0;
this.timer.Stop();
rd.rolls.Clear();
nPoints = 0;
string strSQL =
            "SELECT nRolls,Victim,X,Y,oldX,OldY,Alpha,Red,Green,Blue From RollsTable";
oleDbConnection.Open();
OleDbCommand cmd = new OleDbCommand(strSQL,oleDbConnection);
OleDbDataReader rdr = cmd.ExecuteReader();
while (rdr.Read())
{
rd.AddRow(
int.Parse((rdr["nRolls"]).ToString()),
int.Parse((rdr["Victim"]).ToString()),
float.Parse((rdr["X"]).ToString()),
float.Parse((rdr["Y"]).ToString()),
float.Parse((rdr["oldX"]).ToString()),
float.Parse((rdr["oldY"]).ToString()),
int.Parse((rdr["Alpha"]).ToString()),
int.Parse((rdr["Red"]).ToString()),
int.Parse((rdr["Green"]).ToString()),
int.Parse((rdr["Blue"]).ToString())
);
i++;
}
rdr.Close();
oleDbConnection.Close();
rpDataGrid.DataSource = rd.rolls;
nPoints = i;
this.timer.Start();
}
```

Сведения о хранимых процедурах

Хранимые процедуры – предварительно оттранслированное множество предложений SQL и дополнительных предложений для управления потоком, сохраняемое под именем и обрабатываемое (выполняемое) как одно целое. Хранимые процедуры сохраняются непосредственно в базе данных; их выполнение обеспечивается вызовом со стороны приложения; допускает включение объявляемых пользователем переменных, условий, и других программируемых возможностей.

В хранимыех процедурах могут применяться входные и выходные параметры, сохраняемые процедуры могут возвращать единичные значения и результирующие множества.

Функционально хранимые процедуры аналогичны запросам. Вместе с тем, по сравнению с предложениями SQL, они обладают рядом преимуществ:

- в одной процедуре можно сгруппировать несколько запросов,
- в одной процедуре можно сослаться другие сохраненные процедуры, что упрощает процедуры обращения к БД,
- выполняются быстрее, чем индивидуальные предложения SQL.

Таким образом, хранимые процедуры облегчают работу с базой данных.

Способы создания команд

Известно три способа создания команд для манипулирования данными:

- объявление и создание объекта команды непосредственно в программном коде с последующей настройкой этого объекта вручную. Следующие фрагменты кода демонстрируют этот способ

```
string cs = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
            "Data Source=" +
            @"F:\SharpUser\CS.book\Rolls.db";
OleDbConnection cn = new OleDbConnection(cs);
cn.Open();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = cn; // Объект Соединение цепляется к команде!
```

Либо так:

```
string cs = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
            "Data Source=" +
            @"F:\SharpUser\CS.book\Rolls.db";
OleDbConnection cn = new OleDbConnection(cs);
cn.Open();
OleDbCommand cmd = cn.CreateCommand();// Команда создаётся соединением!
```

- использование инструментария, предоставляемого панелью ToolBox (вкладка Data). Объект соответствующего класса перетаскивается в окно дизайнера с последующей настройкой этого объекта. SqlCommand или OleDbCommand перетаскивается в окно конструктора со вкладки Data, при этом остаётся вручную задать свойства Connection, CommandText, CommandType (для определения типа команды, которая задаётся в свойстве CommandText). При этом свойство CommandType может принимать одно из трёх значений: (1) Text – значение свойства CommandText воспринимается как текст команды SQL. При этом возможна последовательность допустимых операторов, разделённых точкой с запятой, (2) StoredProcedure – значение свойства CommandText воспринимается как имя существующей хранимой процедуры, которая будет исполняться при вызове данной команды, (3) TableDirect – при этом свойство CommandText воспринимается как непустой список имён таблиц, возможно, состоящий из одного элемента. При выполнении команды возвращаются все строки и столбцы этих таблиц.
- размещение (путём перетаскивания) хранимой процедуры из окна Server Explorer в окно дизайнера. Объект Команда соответствующего типа при этом создаётся автоматически на основе любой хранимой процедуры. При этом новый объект ссылается на хранимую процедуру, что позволяет вызывать эту процедуру без дополнительной настройки.

Оба типа объектов Команда (SqlCommand и OleDbCommand) поддерживают три метода, которые позволяют исполнять представляемую команду:

- ExecuteNonQuery – применятся для выполнения команд SQL и хранимых процедур, таких, как INCERT, UPDATE, DELETE. С помощью этого метода также выполняются команды DDL – CREATE, ALTER. Не возвращает никаких значений
- ExecuteScalar – обеспечивает выбор строк из таблицы БД. Возвращает значение первого поля первой стоки, независимо от общего количества выбранных строк.
- ExecuteReader - обеспечивает выбор строк из таблицы БД. Возвращает неизменяемый объект DataReader, который допускает последовательный однонаправленный просмотр извлечённых данных без использования объекта DataAdapter.

Кроме того, класс SqlCommand поддерживает ещё один метод -

- ExecuteXmlReader - обеспечивает выбор строк из таблицы БД в формате XML. Возвращает неизменяемый объект XMLReader, который допускает последовательный однонаправленный просмотр извлечённых данных.

Назначение всех четырёх методов – ИСПОЛНЕНИЕ НА ИСТОЧНИКЕ ДАННЫХ команды, представленной объектом команды.

Parameter

Данные из базы выбираются в разультате выполнения объекта Command. В ADO.NET применяются параметризованные команды. Объект Parameter является средством модификации команд. Представляет свойства и методы, позволяющие определять типы данных и значения параметров.

Настройка команд

Манипулирование данными, которое осуществляется в процессе выполнения команд, естественно, требует определённой информации, которая представляется в команде в виде параметров. При этом характер и содержание управляющей дополнительной информации зависит от конкретного состояния базы на момент выполнения приложения.

Это означает, что настройка команды должна быть проведена непосредственно в процессе выполнения приложения. Именно во время выполнения приложения определяются конкретные значения параметров, проводится соответствующая настройка команд и их выполнение.

В структуре команды предусмотрены так называемые ПОЛЯ ПОДСТАНОВКИ. Объект команды при выполнении приложения настраивается путём присвоения значения параметра полю подстановки. Эти самые поля подстановки реализованы в виде свойства Parameters объекта команды. В зависимости от типа провайдера каждый параметр представляется объектом одного из классов: OleDbParameter или SqlParameter.

Необходимое количество параметров для выполнения той или иной команды зависит от конкретных обстоятельств: каждый параметр команды представляется собственным объектом-параметром. Кроме того, если команда представляется сохраняемой процедурой, то может потребоваться дополнительный объект-параметр.

Во время выполнения приложения объект команды предварительно настраивается путём присваивания свойству Parameters списка (вполне возможно, что состоящего из одного элемента) соответствующих значений параметров – объектов класса Parameters. При выполнении команды эти значения параметров считываются и либо непосредственно подставляются в шаблон оператора языка SQL, либо передаются в качестве параметров хранимой процедуре.

Параметр команды является объектом довольно сложной структуры, что объясняется широким диапазоном различных действий, которые могут быть осуществлены над базой данных.

Кроме того, работа с конкретной СУБД также накладывает свою специфику на правила формирования объекта команды.

Свойства параметров

Parameter является достаточно сложной конструкцией, о чём свидетельствует НЕПОЛНЫЙ список его свойств:

- Value – свойство, предназначенное для непосредственного сохранения значения параметра.
- Direction – свойство объекта-параметра, которое определяет, является ли параметр входным или выходным. Множество возможных значений представляется следующим списком: Input, Output, InputOutput, ReturnValue.
- DbType (не отображается в окне дизайнера) в сочетании с OleDbType (только для объектов типа OleDbParameters) – параметры, используемые для согласования типов данных, принятых в CTS (Common Type System) и типов, используемых в конкретных базах данных.
- DbType (не отображается в окне дизайнера) в сочетании с SQLType (только для объектов типа SqlParameters) - параметры, также используемые для согласования типов данных, принятых в CTS (Common Type System) и типов, используемых в конкретных базах данных,
- ParameterName – свойство, которое обеспечивает обращение к данному элементу списка параметров команды непосредственно по имени, а не по индексу. Разница между этими двумя стилями обращения к параметрам демонстрируется в следующем фрагменте кода:

```
OleDbCommand1.Parameters[0].Value = “OK”;
// В команде, представляемой объектом OleDbCommand1, значение первого
// элемента списка параметров есть строка “OK”.
OleDbCommand1.Parameters[“ParameterOK”].Value = “OK”;
// В команде, представляемой объектом OleDbCommand1, значение элемента
// списка параметров, представленного именем “ParameterOK”,
// есть строка “OK”.
```

- Precision, Scale, Size определяют длину и точность соответствующих параметров. При этом первые два свойства применяются для задания разрядности и длины дробной части значения параметров таких типов как float, double, decimal, последнее свойство используется для указания максимально возможной длины строкового и двоичного параметра.

Установка значений параметров

Следующий пример показывает, как установить параметры перед выполнением команды, представленной хранимой процедурой. Предполагается, что уже была собрана соответствующая последовательность параметров, с именами au_id, au_lname, и au_fname.

```
OleDbCommand1.CommandText = “UpdateAuthor”;
OleDbCommand1.CommandType = System.Data.CommandType.StoredProcedure;
OleDbCommand1.Parameters[”au_id”].Value = listAuthorID.Text;
OleDbCommand1.Parameters[”au_lname”].Value = txtAuthorLName.Text;
OleDbCommand1.Parameters[”au_fname”].Value = txtAuthorFName.Text;
OleDbConnection1.Open();
OleDbCommand1.ExecuteNonQuery();
OleDbConnection1.Close();
```

Получение возвращаемого значения

Сохраняемые процедуры могут обеспечить передачу возвращаемого значения функции приложения, которое обеспечило их вызов. Это передача может быть обеспечена непосредственно параметром при установке свойства Direction параметра в Output или InputOutput, либо за счёт непосредственного возвращения значения сохраняемой процедурой, при котором используется параметр со свойством, установленным в ReturnValue.

Получение значений, возвращаемых сохраняемой процедурой.

- Использование параметра

Для этого следует создать параметр с Direction свойством, установленным в Output or InputOutput (если параметр используется в процедуре как для получения, так и для отправления значений). Очевидно, что тип параметра должен соответствовать ожидаемому возвращаемому значению.

After executing the procedure, read the Value property of the parameter being passed back. Такие вот дела…

- Непосредственный перехват возвращаемого значения сохраняемой процедурой

Для этого следует создать параметр с Direction свойством, установленным в ReturnValue.

Важно! Такой параметр должен быть первым в списке параметров.

При этом тип параметра должен соответствовать ожидаемому возвращаемому значению.

Важно! Предложения SQL - Update, Insert, and Delete возвращают целочисленное значение, соответствующее the number of records affected by the statement.

Это значение может быть получено как возвращаемое значение метода ExecuteNonQuery.

Следующий пример демонстрирует, как получить возвращаемое значение, возвращаемое хранимой процедурой CountAuthors. In this case, it is assumed that the first parameter in the command's Parameters collection is named "retvalue" and that is configured with a direction of ReturnValue.

```
int cntAffectedRecords;
// The CommandText and CommandType properties can be set
// in the Properties window but are shown here for completeness.
oleDbcommand1.CommandText = "CountAuthors";
oleDbCommand1.CommandType = CommandType.StoredProcedure;
oleDbConnection1.Open();
oleDbCommand1.ExecuteNonQuery();
oleDbConnection1.Close();
cntAffectedRecords = (int)(OleDbCommand1.Parameters["retvalue"].Value);
MessageBox.Show("Affected records = " + cntAffectedRecords.ToString());
```

DataReader

Компонента провайдера, объект-представитель (варианта) класса DataReader.

Предоставляет подключённый к источнику данных набор записей, доступный лишь для однонаправленного чтения.

Позволяет просматривать результаты запроса по одной записи за один раз. Для доступа к значениям столбцов используется свойство Item, обеспечивающее доступ к столбцу по его индексу (то есть, ИНДЕКСАТОР!).

При этом метод GetOrdinal объекта-представителя класса DataReader принимает строку с именем столбца и возвращает целое значение, соответствующее индексу столбца.

Непосредственно обращением к конструктору эту компоненту провайдера созать нельзя. Этим DataReader отличается от других компонентов провайдера данных.

Объект DataReader создаётся в результате обращения к одному из вариантов метода ExecuteReader объекта Command (SqlCommand.ExecuteReader возвращает ссылку на SqlDataReader, OleDbCommand.ExecuteReader возвращает ссылку на OleDbDataReader).

То есть, выполняется команда (например, запрос к базе данных), а соответствующий результат получается при обращении к объекту-представителю класса DataReader.

Метод ExecuteReader возвращает множество значений как ОДИН ЕДИНСТВЕННЫЙ ОБЪЕКТ – объект-представитель класса DataReader. Остаётся только прочитать данные.

Выполняется запрос, получается объект-представитель класса DataReader, который позволяет перебирать записи результирующего набора и… ПЕРЕДАВАТЬ НУЖНЫЕ ЗНАЧЕНИЯ КОДУ ПРИЛОЖЕНИЯ.

При этом DataReader обеспечивает чтение непосредственно из базы. И потому требует монопольного доступа к активному соединению. DataReader реализован без излишеств. Только ОДНОНАПРАВЛЕННОЕ чтение! Любые другие варианты его использования невозможны.

```
// Создание объектов DataReader. Пример кода.
System.Data.OleDb.OleDbCommand myOleDbCommand;
System.Data.OleDb.OleDbDataReader myOleDbDataReader;
myOleDbDataReader = myOleDbCommand.ExecuteReader();

System.Data.SqlClient.SqlCommand mySqlCommand;
System.Data.SqlClient.SqlDataReader mySqlDataReader;
mySqlDataReader = mySqlCommand.ExecuteReader();
```

Использование объекта DataReader

Обеспечение так называемого “доступа к ОТСОЕДИНЁННЫМ данным” – заслуга объекта DataReader. Дело в том, что получение данных приложением из базы данных всё равно требует установления соединения. И это соединение должно быть максимально коротким по продолжительности и эффективным. Быстро соединиться, быстро прочитать и запомнить информацию из базы, быстро разъединиться. Именно для этих целей используется в ADO.NET объект DataReader.

После получения ссылки на объект DataReader, можно организовать просмотр записей. Для получения необходимой информации из базы данных этого достаточно.

У объекта DataReader имеется “указатель чтения”, который устанавливается на первую запись результирующего набора записей, который образовался в результате выполнения метода ExecuteReader(). Очередная (в том числе и первая) запись набора становится доступной в результате выполнения метода Read().

В случае успешного выполнения этого метода указатель переводится на следующий элемент результирующей записи, а метод Read() возвращает значение true.

В противном случае метод возвращает значение false. Всё это позволяет реализовать очень простой и эффективный механизм доступа к данным. Например, в рамках цикла while.

Если иметь в виду, что каждая запись состоит из одного и того же количества полей, которые к тому же имеют различные идентификаторы, то, очевидно, что доступ к значению отдельного поля становится возможным через индексатор, значением которого может быть как значение индекса, так и непосредственно обозначающий данное поле идентификатор.

```
while (myDataReader.Read())
{
object myObj0 = myDataReader[5];
object myObj1 = myDataReader[“CustomerID”];
}
```

Важно!

При таком способе доступа значения полей представляются ОБЪЕКТАМИ. Хотя, существует возможность получения от DataReader’а и типизированных значений.

Следует иметь в виду, что DataReader удерживает монопольный доступ к активному соединению. Вот как всегда! Пообещали отсоединённый доступ к данным, а получаем постоянное соединение! Закрывается соединение методом Close()

```
myDataReader.Close();
```

Можно также настроить DataReader таким образом, чтобы закрытие соединения происходило автоматически, без использования команды Close(). Для этого при вызове метода ExecuteReader свойство объекта команды CommandBehavior должно быть выставлено в CloseConnection.

Пример. Выборка столбца таблицы с помощью объекта DataReader. Предполагается наличие объкта OleDbCommand под именем myOleDbCommand. Свойство Connection этого объекта определяет соединение с именем myConnection.

Итак:

```
// Активное соединение открыто.
MyConnection.Open();
System.Data.OleDb.OleDbDataReader myReader = myOleDbCommand.ExecuteReader();
while (myReader.Read())
{
Console.WriteLine(myReader[”Customers”].ToString());
}
myReader.Close();
// Активное соединение закрыто.
MyConnection.Close();
```

Извлечение типизированных данных

Среди множества методов классов DataReader (SqlDataReader и OleDbDataReader) около десятка методов, имена который начинаются с приставки Get… следом за которой – имя какого-то типа. GetInt32, GetBoolean, …

С помощью этих методов от DataReader можно получить и типизированные значения, а не только объекты базового типа!

```
int CustomerID;
string Customer;
// Определить порядковый номер поля 'CustomerID'
CustomerID = myDataReader.GetOrdinal("CustomerID");
// Извлечь строку из этого поля и прописать её в переменную Customer
Customer = myDataReader.GetString(CustomerID);
```

DataAdapter

DataAdapter – составная часть провайдера данных. То есть, подсоединённая компонента объектной модели ADO.NET. Используется для заполнения объекта DataSet и модификации источника данных. Выполняет функции посредника при взаимодействии БД и объекта DataSet.

Обеспечивает связь между источником данных и объектом DataSet. С одной стороны, база данных, с другой - DataSet. Извлечение данных и заполнение объекта DataSet – назначение DataAdapter'а.

Функциональные возможности DataAdapter'а реализуются за счёт:

- метода Fill, который изменяет данные в DataSet. При выполнении метода Fill объект DataAdapter заполняет DataTable или DataSet данными, полученными из БД. После обработки данных, загруженных в память, с помощью метода Update можно записать модифицированные записи в БД,
- метода Update, который позволяет изменять данные в источнике данных с целью достижения обратного соответствия данных в источнике данных по отношению к данным в DataSet.

Фактически, DataAdapter управляет обменом данных и обновлением содержимого источника данных.

DataAdapter представляет набор команд для подключения к базе данных и модификации данных.

Три способа создания DataAdapter:

- Создание с помощью окна Server Explorer.
- Создание с помощью мастера Data Adapter Configuration Wizard.
- Ручное объявление и настройка в коде.

Достойны особого внимания ЧЕТЫРЕ свойства этого класса, фактически представляющие команды БД. Через эти команды объект DataAdapter и воздействует на DataSet и Базу:

- SelectCommand - содержит текст (строку sql) или объект команды, осуществляющей выборку данных из БД. При вызове метода Fill эта команда выполняется и заполняет объект DataTable или объект DataSet,
- InsertCommand - содержит текст (строку sql) или объект команды, осуществляющий вставку строк в таблицу,
- DeleteCommand - содержит текст (строку sql) или объект команды, осуществляющий удаление строки из таблицы,
- UpdateCommand - содержит текст (строку sql) или объект команды, осуществляющий обновление значений в БД.

Transaction

Под транзакцией понимается неделимая с точки зрения воздействия на базу данных последовательность операторов манипулирования данными:

- чтения,
- удаления,
- вставки,
- модификации,

приводящая к одному из двух возможных результатов:

- либо последовательность выполняется, если все операторы правильные,

- либо вся транзакция откатывается, если хотя бы один оператор не может быть успешно выполнен.

Прежде всего, необходимым условием применения транзакций как элементов моделт ADO.NET, является поддержка источником данных (базой данных) концепции транзакции. Обработка транзакций гарантирует целостность информации в базе данных. Таким образом, транзакция переводит базу данных из одного целостного состояния в другое.

При выполнении транзакции система управления базами данных должна придерживаться определенных правил обработки набора команд, входящих в транзакцию. В частности, гарантией правильности и надежности работы системы управления базами данных являются четыре правила, известные как требования ACID.

- Atomicity – неделимость. Транзакция неделима в том смысле, что представляет собой единое целое. Все ее компоненты либо имеют место, либо нет. Не бывает частичной транзакции. Если может быть выполнена лишь часть транзакции, она отклоняется.
- Consistency – согласованность. Транзакция является согласованной, потому что не нарушает бизнес-логику и отношения между элементами данных. Это свойство очень важно при разработке клиент-серверных систем, поскольку в хранилище данных поступает большое количество транзакций от разных систем и объектов. Если хотя бы одна из них нарушит целостность данных, то все остальные могут выдать неверные результаты.
- Isolation – изолированность. Транзакция всегда изолированна, поскольку ее результаты самодостаточны. Они не зависят от предыдущих или последующих транзакций – это свойство называется сериализуемостью и означает, что транзакции в последовательности независимы.
- Durability – устойчивость. Транзакция устойчива. После своего завершения она сохраняется в системе, которую ничто не может вернуть в исходное (до начала транзакции) состояние, т.е. происходит фиксация транзакции, означающая, что ее действие постоянно даже при сбое системы. При этом подразумевается некая форма хранения информации в постоянной памяти как часть транзакции.

Указанные выше правила реализуются непосредственно источником данных. На программиста возлагаются обязанности по созданию эффективных и логически верных алгоритмов обработки данных. Он решает, какие команды должны выполняться как одна транзакция, а какие могут быть разбиты на несколько последовательно выполняемых транзакций.

Работа с транзакцией предполагает следующую последовательность действий:

- инициализацию транзакции

обеспечивается вызовом метода BeginTransaction() от имени объекта Connection, представляющего открытое соединение. В результате выполнения этого метода возвращается ссылка на объект – представитель класса Transaction, который должен быть записан в свойство Transaction всех объектов-команд, которые должны быть задействованы в данной транзакции,

- выполнение команд-участников транзакции с анализом их возвращаемых значений (обычно этот анализ сводится к тривиальному размещению всех операторов, связанных с выполнением транзакции в один try-блок),
- если все команды выполняются удовлетворительно, от имени объекта-представителя класса Transaction вызывается метод Commit(), который подтверждает изманание состояния источника данных. В противном случае (блок catch), от имени объекта-представителя класса Transaction вызывается метод Rollback(), который отменяет ранее произведённые изменения состояния Базы Данных.

Ниже приводится пример исполнения транзакции с помощью объекта соединения класса OleDbConnection с именем xConnection и пары объектов OleDbCommand с именами xCommand1 и xCommand2.

```
System.Data.OleDb.OleDbTransaction xTransaction = null;
try
{
 xConnection.Open();
 // Создаётся объект транзакции.
```

```
 xTransaction = xConnection.BeginTransaction();
 // Транзакция фиксируется в командах.
 xCommand1.Transaction = xTransaction;
 xCommand2.Transaction = xTransaction;
 // Выполнение команд.
 xCommand1.ExecuteNonQuery();
 xCommand2.ExecuteNonQuery();
 // Если ВСЁ ХОРОШО и мы всё ещё здесь – ПРИНЯТЬ ТРАНЗАКЦИЮ!
 xTransaction.Commit();
}
catch
{
 // Если возникли осложнения – отменяем транзакцию.
 xTransaction. Rollback();
}
finally
{
 // По любому соединение закрывается.
 xConnection.Close();
}
```

Литература

1. Петзольд Чарльз Программирование для Microsoft Windows на C# В 2-х томах
2002 г., Русская редакция

2. Петзольд Чарльз Программирование в тональности C#
512 стр., 2004 г., Русская редакция

3. Эндрю Троелсен C# и платформа .NET. Библиотека программиста
796 стр., 2004 г., Питер. Серия: Библиотека программиста.

4. Герберт Шилдт C#. Учебный курс
511 стр., 2002 г., Питер.

5. Герберт Шилдт Полный справочник по C#
752 стр., 2004 г., Вильямс.

6. Джеффри Рихтер Программирование на платформе Microsoft .NET Framework
512 стр., 2005 г., Питер, Русская редакция.

7. В. Жарков Самоучитель Жаркова по анимации и мультипликации в Visual C# .NET
432 стр., 2004 г., Жарков Пресс.

8. Джесс Либерти Программирование на C#
688 стр., 2003 г., Символ-Плюс.

9. Александр Фролов, Григорий Фролов Язык C#. Самоучитель
560 стр., 2002 г., Диалог-МИФИ.

10. Майо Джозеф C#. Искусство программирования
656 стр., 2002 г., ДиаСофт.

11. У. Робисон C# без лишних слов
352 стр., 2002 г., ДМК.

12. Микелсен К. Язык программирования C#. Лекции и упражнения
656 стр., 2002 г., ДиаСофт

13. Разработка WINDOWS-приложений на Microsoft Visual Basic.NET и Microsoft Visual C# .NET. Учебный курс.
512 стр., Русская Редакция

14. Библиотека MSDN на русском языке
http://msdn.microsoft.com/library/rus/

Оглавление