Введение в программирование на C# 2.0

0. Введение: Предисловие: версия для печати и PDA

Книга представляет собой пособие по изучению языка программирования C#, который является одним из важных элементов платформы Microsoft .NET. Основные задачи пособия заключаются:

- в ознакомлении с синтаксисом и семантикой языка программирования C# 2.0;
- в описании особенностей архитектуры .NET;
- в формировании навыков разработки приложений в рамках парадигмы объектно-ориентированного программирования.

При описании синтаксиса некоторых языковых конструкций C# 2.0 в книге использовалась нотация Бэкуса-Наура. Формы Бэкуса-Наура (БНФ) традиционно применяются 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. Они 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 нетерминальных и терминальных символов, отображающих структуру определяемого понятия. 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 синтаксиса не ограничено. В данном контексте их может быть один, два, три, ..., много, а может и не быть вовсе.

Материал книги основан на находящейся в открытом доступе литературе по C# и .NET. При изучении C# следует иметь в виду одно универсальное правило. Изучение нового языка программирования, новой платформы, новой технологии требует прочтения по крайней мере нескольких из ставших на сегодняшний день классическими книг. Часть из этих книг приведена в списке литературы.

В книге содержится большое количество примеров. В ряде случаев это всего лишь документированные фрагменты программного кода. Однако значительная часть приводимых примеров является законченными работающими приложениями. И хотя к книге 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 "волшебные" заклинания, в результате которых в окошке консольного приложения появляются последовательности выводимых символов.

Введение

Обзор .NET. Основные понятия

ПЛАТФОРМА – в контексте информационных технологий – среда, обеспечивающая выполнение программного кода. Платформа определяется характеристиками процессоров, особенностями операционных систем.

Framework – это инфраструктура среды выполнения программ, нечто, определяющее особенности разработки и выполнения программного кода на данной платформе. Предполагает средства организации взаимодействия с операционной системой и прикладными программами, методы доступа к базам данных, средства поддержки распределенных (сетевых) приложений, языки программирования, множества базовых классов, унифицированные интерфейсы пользователя, парадигмы программирования.

Microsoft .NET – платформа.

.NET Framework – инфраструктура платформы Microsoft .NET. Включает следующие основные компоненты: Common Language Runtime (CLR) и .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# 2.0, 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 (управляемого) программного кода.

Управляемый код – программный код, который при своем выполнении способен использовать службы, предоставляемые CLR. Соответственно, неуправляемый код подобной способностью не обладает. Об особенностях управляемого кода можно судить по перечню задач, решение которых возлагается на CLR:

- Управление кодом (загрузка и выполнение).
- Управление памятью при размещении объектов.
- Изоляция памяти приложений.
- Проверка 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 кодами (в том числе и COM-объектами).
- Поддержка сервисов для разработки (профилирование, отладка и т.д.).

Короче, CLR – это набор служб, необходимых для выполнения управляемого кода.

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 нее 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 производит следующие действия:

- находит расположение 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, передает 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 служить основой 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, и в частности:

- встроенные (элементарные) типы, представленные в виде классов (на платформе .NET все 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-протоколов (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) – промежуточный язык платформы Microsoft .NET. Исходные тексты программ для .NET-приложений пишутся на языках программирования, соответствующих спецификации CLS. Для таких языков 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, который содержит информацию о данных, используемых в программе. Фактически это наборы таблиц, которые включают в себя 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;
- обеспечения динамической подсказки (IntelliSence) при разработке программы стандартными инструментальными средствами (Microsoft Visual Studio .NET) на основе метаданных.

Языки, для которых реализован перевод на MSIL:

- Visual Basic,
- Visual C++,
- Visual C# 2.0,

и еще 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 выполняемом 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, определенные в исходном коде; 2) типы, на которые имеются в коде ссылки
IL	Собственно код, который создается 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. Это еще один набор таблиц метаданных, который:

- идентифицирует сборку в виде текстового имени, ее версию, культуру и цифровую сигнатуру (если сборка распределяется 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. 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.

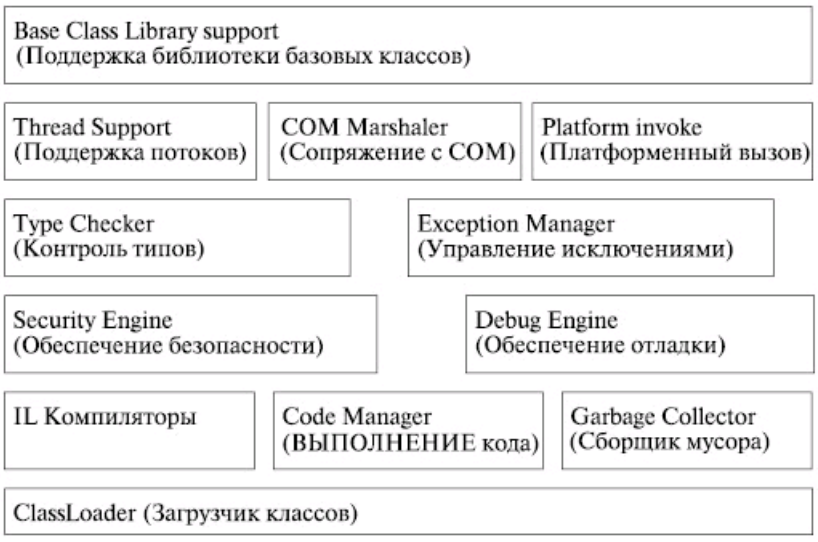
В среде CLR допускается совместная работа и взаимодействие компонентов программного обеспечения, реализованных на различных языках программирования.

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 оказываются приложениями на MSIL (IL).

CLR берет 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).



Структура среды выполнения CLR (основные функциональные элементы среды) представлена на рисунке.

Строгий контроль типов, в частности, предполагает проверку соответствия типа объекта диапазону значений, которые могут быть присвоены данному объекту.

Защита .NET (безопасность) строится поверх системы защиты операционной системы компьютера. Она не дает пользователю или коду делать то, что делать не позволено, и накладывает ограничения на выполнение кода. Например, можно запретить доступ некоторым секциям кода к определенным файлам.

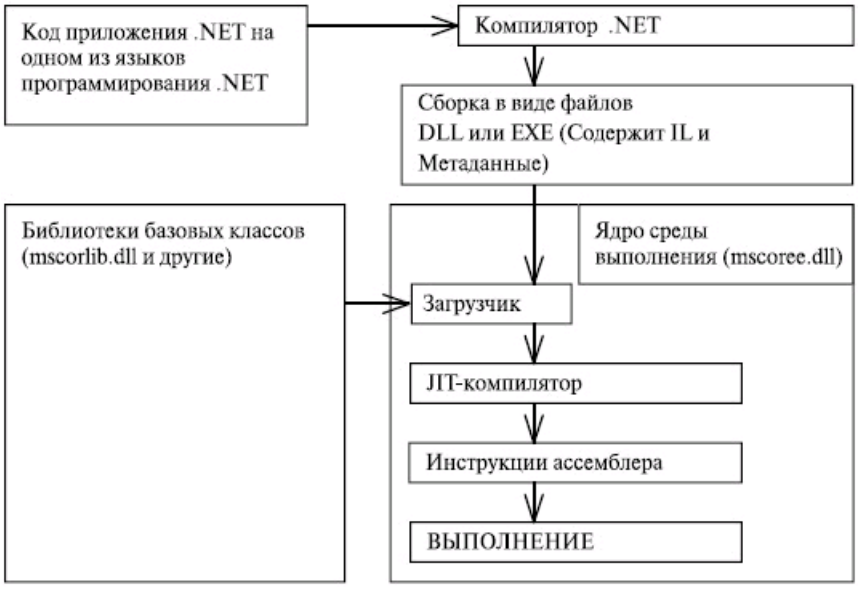
Функциональные блоки CLR Code Manager и Garbage Collector работают совместно: Code Manager обеспечивает размещение объектов в управляемой памяти, Garbage Collector – освобождает управляемую память.

Exception Manager включает следующие компоненты:

- finally handler (обеспечивает передачу управления в блок finally);
- fault handler (включается при возникновении исключения);

- type-filtered handler (обеспечивает выполнение кода соответствующего блока обработки исключения);
- user-filtered handler (выбор альтернативного блока исключения).

Ниже представлена схема выполнения .NET-приложения в среде CLR.



AppDomain (домен приложения) – это логический контейнер сборок, который используется для изоляции приложения в рамках адресного пространства процесса. Код, выполняемый в CLR (CLR-процесс), отделен 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). 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. В приложении, помимо основного домена, может быть создано несколько дополнительных доменов.

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.

GAC (Global Assembly Cache – Общий КЭШ сборок). 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.

CTS – Common Type System (Стандартная Система Типов). Поддерживается всеми языками платформы. В силу того, что .NET основана на парадигме ООП, речь здесь идет 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 ПРЕДОПРЕДЕЛЕННЫХ типов, об их использовании и управлении (правилах преобразования значений). CTS играет важную роль в деле интеграции разноязыких управляемых приложений.

Пространство имен – это способ организации системы типов в единую группу. В рамках .NET существует единая (общеязыковая) библиотека базовых классов. Концепция пространства имен обеспечивает эффективную организацию и навигацию по этой библиотеке. Вне зависимости от языка программирования, доступ к определенным классам обеспечивается за счет их группировки в рамках общих пространств имен.

System	
System.Data	Классы 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, удаленный доступ
System.Runtime.Remoting	
System.Security	Криптография, разрешения
System.Threading	Работа с потоками

System.Web	Работа с web-приложениями

Выполнение неуправляемых исполняемых модулей (обычные Windows-приложения) обеспечивается непосредственно системой Windows. Неуправляемые модули выполняются в среде Windows как "простые" процессы. Процесс характеризуется замкнутым адресным пространством. При этом необходимая для выполнения программы информация (данные) размещается в различных областях адресного пространства процесса, которые называются стеком и кучей. Эти области имеют различное назначение и механизмы управления.

Информация в стеке и куче представляется в виде последовательностей битов с четко определенными свойствами. Такие последовательности называются объектами. Множество неизменяемых свойств, которые могут служить для классификации различных объектов (количество байтов последовательности, формат представления информации в последовательности, операции, определенные над объектом), задают тип объекта.

Таким образом, при выполнении программы в стеке и куче размещаются объекты различных типов.

В стеке размещается информация, необходимая для выполнения программного кода (реализация механизма вызова функций, поддержка операторов управления, сохранение значений локальных переменных). Основным механизмом управления стеком является указатель стека. Освобождение стека происходит в результате перемещения указателя стека, которое связано с выполнением операторов программы.

В куче размещаются используемые в программе данные. Они структурированы и представлены объектами различных типов. Размещение объектов в куче происходит в результате выполнения специального оператора new. Основным механизмом управления кучи является сборщик мусора (Garbage Collector).

Ссылкой (ссылкой на объект) называется объект, который определен на множестве значений, представленных адресами оперативной памяти в рамках процесса. Если значением ссылки является адрес конкретного объекта, ссылка считается определенной. Ссылка на объект не определена, если ее значение не соответствует адресу какого-либо объекта. Неопределенное значение ссылки в программе кодируется специальным обозначением (null). Особых ограничений на месторасположение ссылки не существует. Ссылка может располагаться как в стеке, так и в куче.

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.

Принципы объектно-ориентированного программирования

Имеет смысл воспринимать языки программирования высокого уровня как ОБЫЧНЫЕ ИНОСТРАННЫЕ ЯЗЫКИ.

На английском, французском, китайском и т.д. языках можно сформулировать собственные мысли таким образом, чтобы они были понятны носителю данного иностранного языка.

Языки программирования позволяют аналогичным образом "общаться" с электронными устройствами.

Программист владеет естественным языком.

Для электронного устройства "понятна" специфическая система команд.

Общение на естественном языке пока невозможно для устройств.

Общение на машинном языке неудобно для программиста.

Компромиссным решением является разработка "промежуточного" языка – языка программирования. Она ведется в соответствии с некоторыми представлениями о том, КАК можно описать что-либо на языке программирования таким образом, чтобы это описание, несмотря на специфику "слушателя" было бы максимально простым и понятным для всех участников общения. Эти представления о способах изложения мысли на языке программирования формулируются в виде системы общих принципов, которые формируют парадигму программирования.

За более чем полувековой период развития информационных технологий парадигмы менялись несколько раз. Современные языки программирования разрабатываются в соответствии с парадигмой Объектно-Ориентированного Программирования (ООП), которая основана на нижеследующих принципах.

- **Принцип наследования**. Обобщение и детализация являются основными видами интеллектуальной деятельности в повседневной жизни. Не вдаваясь в детали, можно утверждать, что VW Passat – это АВТОМОБИЛЬ. АВТОМОБИЛИ бывают разных моделей (марок, типов). В частности, АВТОМОБИЛЬ, обладающий определенным набором специфических свойств, может быть идентифицирован как автомобиль Ford Focus. Обобщить частное явление и понять, ЧТО ЭТО. Исходя из общих представлений, сформулировать частный случай. Частное обобщается, общее детализируется. Автомобиль любой марки (VW, Renault, Ford) является детализацией общего понятия АВТОМОБИЛЬ и НАСЛЕДУЕТ наиболее общие свойства (знать бы какие) понятия АВТОМОБИЛЬ. Объект (ФАКТ), для которого не удается найти обобщающего понятия (невозможно понять, ЧТО ЭТО ТАКОЕ), является АРТЕФАКТОМ. Формальный язык, построенный в соответствии с принципами ООП, должен включать средства обобщения и детализации, которые и обеспечивают реализацию принципа наследования.
- **Принцип инкапсуляции**. Окружающий мир воспринимается трехмерным. При описании объекта реального мира закономерными являются вопросы "Что у него ВНУТРИ?", "Как он УСТРОЕН?". Или утверждения "У него внутри ...", "Он устроен ...". Формальный язык, построенный в соответствии с принципами ООП, должен содержать средства, которые позволяли бы описывать "внешний вид", внутреннее устройство, механизмы взаимодействия объекта с окружающим миром.
- **Принцип полиморфизма**. Предполагается, что при выполнении программы можно заставить "работать" (активировать) созданные в программе на основе предварительного описания объекты. При этом объекты реализуют заложенную в них функциональность. Описание множеств объектов (объявление классов объектов) осуществляется в соответствии с принципами наследования и инкапсуляции. Принцип полиморфизма предполагает универсальный механизм реализации функциональности объектов любых классов. Формальный язык, соответствующий принципам ООП, поддерживает стандартные методы активизации объектов. Эти методы просты в реализации и универсальны в применении для любого типа объектов.

Введение в программирование на C# 2.0

9. Лекция: Делегаты и события: версия для печати и PDA
Класс, структура, интерфейс, перечисление, делегат – это все разнообразные категории классов. Каждая категория имеет свои особенности объявления, свое назначение и строго определенную область применения. Об особенностях делегатов в данной лекции

Класс, структура, интерфейс, перечисление, делегат – это все разнообразные категории классов. Каждая категория имеет свои особенности объявления, свое назначение и строго определенную область применения.

Делегаты

Делегат – это 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. В частности, тех которые оказываются видимыми из пространства имен, 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.

Любой класс-делегат наследует `System.MulticastDelegate`. Это обстоятельство определяет многоадресность делегатов: в ходе выполнения приложения объект-делегат способен запоминать ссылки на произвольное количество функций независимо от их статичности или нестатичности и принадлежности классам. Многоадресность обеспечивается внутренним списком, в который заносятся ссылки на функции, соответствующие заданной сигнатуре и спецификации возвращаемого значения.

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 объявления и применения 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 пошел!
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
// классе, 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.
Console.WriteLine(asd.d.Method.ToString());

// 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 делегатов.
// Сюда будем сгружать содержимое Multicast'a.
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 измененным 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!
// Еще 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 встретишь в этом списке!
// 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 чем 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 чем 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);

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

Листинг 9.1.

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.

Два примера иллюстрируют технику применения событий и функциональные возможности 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();

// Создается 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...
// Орел 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;
}
}

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

Листинг 9.2.

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 еще и УВЕДОМЛЯЕТ подписавшийся у нее 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 несет
// информацию о 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 еще и УВЕДОМЛЯЕТ подписавшийся у нее 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 объекты 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 неважно, 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);
```

```
}
}
}
```

Листинг 9.3.

Анонимные методы и делегаты для анонимных методов

Прежде всего, дадим определение АНОНИМНОГО МЕТОДА. В C# 2.0 это фрагмент кода, который оформляется в виде блока и располагается в теле метода или конструктора. Анонимный метод имеет характерный "заголовок", который содержит ключевое слово delegate и список параметров анонимного метода. Внутри блока, представляющего анонимный метод, действуют общепринятые правила областей видимости (scope). Анонимный метод выступает в контексте выражения инициализации делегата и при выполнении содержащего анонимный метод кода НЕ выполняется. В этот момент происходит инициализация делегата, которому все равно, на ЧТО он настроен: непосредственно на метод или на блок операторов в методе (то есть на анонимный метод).

Далее приводится пример объявления анонимного метода, настройки и активизации делегата, в результате которой и происходит выполнение кода анонимного метода:

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

namespace DelegatesForAnonymousMethods
{
delegate int DDD(char op, int key1, int key2);

class DAM
{

public static int DAMstatic(char chKey, int iKey1, int iKey2)
{
// Это простой блок операторов.
// При желании его можно использовать как АНОНИМНЫЙ МЕТОД.
// Для этого всего лишь надо будет дописать выражение инициализации
// делегата. Что и будет сделано в свое время.
{
int ret;
ret = iKey1 % iKey2;
}
//return ret; // Внимание. Здесь ret не определен!

Console.WriteLine("DAMstatic: {0},{1},{2}", chKey, iKey1, iKey2);
return 0;
}

// Делегат как параметр!
// Функция модифицирует значение делегата, передаваемого как ссылка
// на ссылку, добавляя код еще одного анонимного метода,
// и выполняет при этом свою собственную работу.
public int xDAM(ref DDD dKey)
{
Console.WriteLine("Hello from xDAM!");

dKey += delegate(char op, int key1, int key2)
{
Console.WriteLine("this code was from xDAM");
return
((string)(op.ToString()+key1.ToString()+key2.ToString())).GetHashCode();
};

int x = 0;
while (true)
{
x++;
if (x == 100) break;
}

return 0;
}

// И опять делегат как параметр...
// Выполнение набранного кода.
public void goDAM(DDD dKey)
{
int ret;
Console.WriteLine("this is goDAM!");
ret = dKey('+', 5, 5);
Console.WriteLine(ret);
}
}

class Program
{
static void Main(string[] args)
{
DDD d0;
int x;

x = 0;
```

```
// Это АНОНИМНЫЙ МЕТОД.
// Объявление анонимного метода - составная часть выражения инициализации
// делегата. В момент инициализации сам код не выполняется!
d0 = delegate(char op, int key1, int key2)
{
int res;
switch (op)
{
    case '+':
         res = key1 + key2;
         break;
     case '-':
         res = key1 - key2;
         break;
     case '*':
         res = key1 * key2;
         break;
     case ':':
         try
        {
          res = key1 / key2;
         }
         catch
        {
          Console.Write("That was a wrong number! ");
          res = int.MaxValue;
         }
         break;
     default:
         Console.Write("That was illegal symbol of operation! ");
          res = int.MinValue;
         break;
 }

Console.WriteLine(">>>{0}<<<", res);

return res;
};// Конец кода для инициализации делегата (конец анонимного метода).

x++;

// Делегат настраивается на фрагмент кода.
// И вот теперь этот код выполняется...
x = d0('*', 125, 7);
Console.WriteLine(x);
// Даже будучи настроенным на анонимный метод, делегат остается делегатом!
// Вот он настраивается на полноценную статическую функцию...
d0 += new DDD(DAM.DAMstatic);
x = d0(':', 125, 25);
Console.WriteLine(x);
Console.WriteLine("_____________________________________________");

// А вот сам выступает в качестве параметра
//  и передается функции, в которой делегату
// передается ссылка еще на один анонимный метод.
DAM dam = new DAM();
dam.xDAM(ref d0);

x = d0('-', 125, 5);
Console.WriteLine(x);
Console.WriteLine("_____________________________________________");

dam.goDAM(d0);

Console.WriteLine("_____________________________________________");
}
}
}
```

Листинг 9.4.

События и делегаты. Различия

Так в чем же разница между событиями и делегатами в .NET?

В последнем примере предыдущего раздела при объявлении события очевидно его строгое соответствие определенному 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 произойдет. Вернее, ВСЕ 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.

10. Лекция: Атрибуты, сборки, рефлексия: версия для печати и PDA
Рефлексия представляет собой процесс анализа типов (структуры типов) в ходе выполнения приложения (сборки). В .NET рефлексия реализуется свойствами и методами класса System.Type и классов пространства имен System.Reflection

Рефлексия (Reflection) – предоставление выполняемому коду информации о нем самом.

Рефлексия представляет собой процесс анализа типов (структуры типов) в ходе выполнения приложения (сборки). В .NET рефлексия реализуется свойствами и методами класса `System.Type` и классов пространства имен `System.Reflection`.

Пространство имен System.Reflection

Пространство имен `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.

`System.Reflection` определяет типы для организации позднего связывания и динамической загрузки типов.

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

Класс `System.Type` содержит методы, позволяющие получать информацию о типах приложения. Является основой для реализации функциональности пространства имен `System.Reflection` и средством для получения доступа к метаданным.

Использование членов класса `Type` 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 объекты были созданы в результате обращения к одному и тому же типу.

Объект — представитель класса `Type` может представить любой из следующих типов:

- классы;
- типы-значения;
- 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");
```

   Здесь используется операция `typeof`. Операндом этой унарной операции является обозначение класса (в буквальном смысле – имя класса).

```
class MyType
{
:::::
}

:::::
Type t = typeof(MyType);
```

4. 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.

```
// А как этим хозяйством пользоваться – не знаю. В хелпах не нашел.
public delegate bool TypeFilter(Type m, object filterCriteria);
```

5. 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

Сначала – определения.

Раннее (статическое) связывание – деятельность, выполняемая на стадии компиляции и 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 и 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 изощренной системе управления, для реализации которой применяется перечисление `BindingFlags`. В рамках этого перечисления определяются значения флажков, которые управляют процессом динамического связывания в ходе реализации отражения.

Перечисление также применяется для управления методом `GetMethod`.

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 должны 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 `PROP-PUTREF` 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 Type
// 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" все 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 переопределенного 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} // 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 (ArgumentOutOfRange Exception)) throw;
// А вот как реагируем на ArgumentOutOfRangeException.
// Вот так скромненько уведомляем о попытке присвоения запрещенного
// 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);
 }
 }
```

Листинг 10.1.

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;
- и многое другое.

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 ее в соответствии с предопределенными правилами или замыслами разработчика. Различаются:

- предопределенные атрибуты. В .NET реализовано множество атрибутов с предопределенными значениями:

  `DllImport` – для загрузки .dll-файлов;

  `Serializable` – означает возможность сериализации свойств объекта – представителя класса;

  `NonSerialized` – обозначает данные-члены класса как несериализуемые. Карандаши (средство графического представления информации, элемент GDI+) не сериализуются;

- 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 – все та же система типов с наследованием от базового класса `System.Attribute`.

И неспроста! В конце концов, информация, содержащаяся в атрибутах, предназначается для CLR, и она в случае необходимости должна суметь разобраться в этой информации. 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.

Ниже приводится описание членов класса `Attribute`.

Открытые свойства

`TypeId`	При реализации в производном классе получает уникальный идентификатор для этого атрибута `Attribute`

Открытые методы

`Equals`	Переопределен (см. `Object.Equals`)
`GetCustomAttribute`	Перегружен. Извлекает пользовательский атрибут указанного типа, который применен к заданному члену класса
`GetCustomAttributes`	Перегружен. Извлекает массив пользовательских атрибутов указанного типа, которые применены к заданному члену класса
`GetHashCode`	Переопределен. Возвращает хэш-код для этого экземпляра
`GetType` (унаследовано от `Object`)	Возвращает `Type` текущего экземпляра
`IsDefaultAttribute`	При переопределении в производном классе возвращает значение, показывающее, является ли значение этого производного экземпляра значением по умолчанию для производного класса
`IsDefined`	Перегружен. Определяет, применены ли какие-либо пользовательские атрибуты заданного типа к указанному члену класса
`Match`	При переопределении в производном классе возвращает значение, указывающее, является ли этот экземпляр эквивалентным заданному объекту
`ToString` (унаследовано от `Object`)	Возвращает `String`, который представляет текущий `Object`

Защищенные конструкторы

`Attribute`-конструктор	Инициализирует новый экземпляр класса `Attribute`

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 C# для функций финализации используется синтаксис деструктора
`MemberwiseClone` (унаследовано от `Object`)	Создает неполную копию текущего `Object`

Следующий пример является демонстрацией объявления 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;
 }
 }
 }//=================================================================

 // Еще 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 трех 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
 // снабжен соответствующим ранее определенным пользовательским
 // СТАНДАРТНЫМ атрибутом. У класса атрибута AnimalTypeAttribute есть все,
 // 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), которая содержит список описателей методов,
 // объявленных в данном классе. В список будет включена информация
 // о ВСЕХ методах класса: о тех, которые были определены явным
 // образом, и о тех, которые были унаследованы
 // от базовых классов. И по этому списку описателей методов мы
 // пройдем 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 снабжен 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 Color.Red -
  // подняли флажок второй раз.
  if (((ColorTypeAttribute)attr).ColorIs.CompareTo(Color.Red) == 0)
  invokeFlag++;
  }

  // И если случилось счастливое совпадение значений атрибутов метода
  // (Красный Бычок), то 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++;
  }
  }
  }//============================================================
  }
```

Листинг 10.2.

Эта программа демонстрирует одну замечательную особенность синтаксиса объявления атрибутов. При их применении можно использовать сокращенные имена ранее объявленных классов атрибутов.

В программе объявлено два класса атрибутов – наследников класса `Attribute`:

```
public class AnimalTypeAttribute : Attribute...
public class ColorTypeAttribute : Attribute...
```

При объявлении соответствующих экземпляров атрибутов последняя часть имени класса атрибута (`...Attribute`) была опущена.

Вместо

```
  [AnimalTypeAttribute(Animal.Dog)]
```

используется имя

```
  [AnimalType(Animal.Dog)],
```

вместо

```
  [ColorTypeAttribute(Color.Red)]
```

используется

```
  [ColorType(Color.Red)]
```

Транслятор терпимо относится только к этой модификации имени класса атрибута. Любые другие изменения имен атрибутов пресекаются.

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 элементов выполняемого приложения.

Сборки бывают:

- частными (private). Представляют наборы типов, которые могут быть использованы только теми приложениями, где они включены в состав. Располагаются в файлах с расширениями .dll (.exe) и .pdb (program debug Database). Для того чтобы использовать в приложении частную сборку, ее 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 компьютере. Эти сборки устанавливаются не в каталог приложения, а в специальный каталог, называемый Глобальным Кэшем Сборок (Global Assembly Cache – GAC). Кэш на платформе Windows XP усилиями специальной утилиты принимает вид каталога (путь ...\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 ее 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 в состав приложения не пользуются.

Для анализа сборки применяется утилита IlDasm.exe, которую можно подключить к непосредственно вызываемому из среды разработки Visual Studio 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, который содержит 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 хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-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
 {

 // The main entry Point for the application.

 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!");
 }
 }
 }
```

Листинг 10.3.

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
 {
 // Class1 : первая компонента сборки AssemblyForStart.

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:

```
using System;
 using System.Reflection;
 using System.IO;

namespace AssemblyStarter
 {

 // Приложение обеспечивает запуск сборки.

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, что 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 загрузке 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.
 // Все происходит в режиме ПОЗДНЕГО связывания.
 // Поэтому ни о каких ЯВНЫХ упоминаниях
 // имен 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, все 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 входа сборки объявил без параметров.
 mi.Invoke(null,null);
  }
  }
  }
```

Листинг 10.4.

Собрали, запустили. Получилось. Стало быть, ВСЕ ХОРОШО.

В ходе выполнения приложения класс `Assembly` позволяет:

- получать информацию о самой сборке;
- обращаться к членам класса, входящим в сборку;
- загружать другие сборки.

Разбор полетов

В приведенном 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 поймет 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`. Создает экземпляр COM-объекта;
- `CreateInstance`. Создает 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`. Создает 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, что 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" в манифесте сборки с помощью IlDasm.exe. Можно было бы для пущей крутизны кода поле `Version` проинициализировать непосредственно (дело хозяйское).

И тут открываются несколько проблем:

- в манифесте версия задается последовательностью цифр, разделенных 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 четверки чисел. Лишь бы требования формата были соблюдены;
- две одноименных частных сборки с разными версиями в раздел References третьей сборки загрузить не получается. Одноименные файлы с одинаковым расширением в один каталог не помещаются. Невзирая на версии.

Следующий шаг: загружаем сборку в память.

Используется блок `try`, поскольку загружаемую сборку можно и не найти.

При загрузке сборки известно ее расположение (application directory), однако с расширением имени могут возникнуть проблемы. Действует такой алгоритм "поиска" (и называть-то это поиском как-то неудобно):

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 алгоритмы?

Все 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 определенном 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, я что-то не так 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 description for Class1.

public class Class1
 {
 public Class1()
 {

 }

public void f0()
 {
 Console.WriteLine("This is SharedAssembly00.f0()");
 }
 }
 }
```

2. Делаем пару "Открытый/Закрытый ключ".

   Для этого 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\Shared Assembly00\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, содержащем 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`
`{`
`.publickeytoken =`
`(90 8E D8 5E 3E 37 72 08)// ...^>7r.`
`.ver 1:0:1790:37888`
`}`
`::::::::::`
	Манифест сборки общего пользования в GAC содержит такое же значение ключа:
	`908ED85E3E377208`
	Его можно увидеть при исследовани и содержимого GAC (свойства элемента)

	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 этого либо используем утилиту gacutil.exe с ключом `\i` и именем сборки с полным путем 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

Создаем новую сборку, в которой предполагается использовать наше детище. Затем добавляем ссылку на сборку. Не все 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 нехитрое...

```
using System;
 using SharedAssembly00;
 using SharedAssembly01;
 using SharedAssembly02;

namespace AssemblyStarter01
 {

 // Summary description for Class1.

class startClass
 {

 // The main entry Point for the application.

 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);
 }
 }
 }
 }
```

Листинг 10.5.

Итак, подключили общедоступную сборку. Она не копируется, а остается в GAC. При создании клиента были выполнены определенные телодвижения, в результате которых клиент сохраняет информацию о свойствах располагаемых в GAC компонентов. 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

Все, о чем 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 они обречены на бесследное исчезновение, если, конечно, не были предприняты особые усилия по их сохранению на диск.

Для работы с динамическими сборками используется пространство имен `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 возвращаемой метаданными лексемы, которая используется 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

Создание, сохранение, загрузка и выполнение сборки

```
 //-------------------------------------------------------------------
 // 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
 {

 // AssemblyGenerator - класс, реализующий динамическую генерацию сборки.

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");
 // Создается сборка в памяти. В рамках текущего домена приложения.
 // С использованием режима доступа,
 // который задается одним из элементов перечисления:
 // Run - динамическая сборка выполняется, но не сохраняется;
 // RunAndSave - динамическая сборка выполняется и сохраняется;
 // Save - динамическая сборка не выполняется, но сохраняется.
 AssemblyBuilder assembly = currentAppDomain.DefineDynamicAssembly(assmName,
 AssemblyBuilderAccess.Save);
 // Создается однофайловая сборка, в которой имя единственного
 // модуля совпадает с именем самой сборки.
 ModuleBuilder module = assembly.DefineDynamicModule("DynamicAssm",
 "DynamicAssm.dll" );
 // Создается и определяется класс DynamicTest.
 // Метод module.DefineType позволяет
 // встраивать в модуль класс, структуру или интерфейс.
 // Вторым параметром метода идет элемент перечисления.
 // Таким образом создается 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,
 // определенного 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.Assembly Generator::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();
 // Остается его сохранить на диск.
 assembly.Save("DynamicAssm.dll");
 return 0;
 }


static void Main(string[] args)
 {
 // Создается 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");

 // Создается 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? //
}
}
}
```

Листинг 10.6.

Введение в программирование на C# 2.0

11. Лекция: Ввод/вывод: версия для печати и PDA
Применительно к обсуждаемым проблемам ввода/вывода в программах на C#, поток – это последовательность байтов, связанная с конкретными устройствами компьютера (дисками, дисплеями, принтерами, клавиатурами) посредством системы ввода/вывода. Подробнее о потоках - в этой лекции

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 (дисками, дисплеями, принтерами, клавиатурами) 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 обеспечиваются стандартными наборами, как правило, одноименных функций ввода/вывода со стандартным интерфейсом.

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 объем 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, от источников и получателей информации.

Процедуры чтения и записи информации определяется следующим (список неполон!) набором свойств и АБСТРАКТНЫХ методов, объявляемых в этом классе.

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 жестко 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 определенного (`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 определенного (`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 задается 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 ее в поток ЗАПИСАТЬ. Как чтение, так и запись предполагают реализацию определенных 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()` – считывает следующий знак, не изменяя состояние средства чтения или источника знака. Возвращает следующий доступный знак, не считывая его в действительности из потока входных данных;
- `int Read(...)` – несколько одноименных перегруженных 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 максимально возможного количества символов из текущего потока и записывает данные в буфер, начиная с некоторого значения индекса;
- `string ReadLine()` – читает строку символов из текущего потока. Возвращается ссылка на объект типа `string`;
- `string ReadToEnd()` – читает все символы, начиная с текущей позиции символьного 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()` – обеспечивает очистку буферов вывода. Содержимое буферов выводится в выходной поток;
- `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 осуществлять 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.

Еще 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 не известно.

Предопределенные потоки ввода/вывода

Предопределенные потоки ввода/вывода используются для реализации ввода/вывода в рамках консольных приложений. Это еще одна категория потоков.

В классах предопределенных 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 свойства, обеспечивающие доступ к предопределенным потокам. В классе `System.Console` таких потоков три (`In, Out, Error`).

Ниже описываются свойства этих потоков:

- `Standard input stream` – является объектом – представителем класса `TextReader`. По умолчанию поток `In` ориентирован на получение информации с клавиатуры.
- `Standard output stream` – является объектом – представителем класса `TextWriter`. По умолчанию обеспечивает вывод информации на дисплей.
- `Standard error stream` – также является объектом – представителем класса `TextWriter`. И также по умолчанию выводит информацию на дисплей.

В классе `System.Console` также определены следующие СТАТИЧЕСКИЕ (!) функции – члены (вернее, множества СТАТИЧЕСКИХ ПЕРЕГРУЖЕННЫХ функций), обеспечивающие 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, который представляется определенным в классе статическим свойством `Console.Out`. Это свойство предоставляет доступ к объекту – представителю класса `TextWriter`. Таким образом, 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 определенного в классе `Console` метода `ReadLine`

```
Console.ReadLine(...);
```

обеспечивает получение информации через поток, доступ к которому предоставляется статическим свойством `Console.In`. Непосредственный вызов аналогичной функции от имени члена класса `TextReader` выглядит следующим образом:

```
Console.In.ReadLine(...);
```

Функция – представитель класса `Console`, отвечающая за непосредственное направление информации в поток вывода сообщений об ошибках, не предусмотрена.

Вывод информации об ошибках может быть реализован путем непосредственного обращения к `standard error output stream`:

```
Console.Error.WriteLine(...);
```

Если вспомнить, что свойство `Error` возвращает ссылку на объект – представитель класса `TextWriter` и, так же как и свойство `Out`, обеспечивает вывод в окно 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 перенаправления 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 объеме реализующим принципы ООП. В этом языке все 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 класса `Object` (а это ВСЕ типы!) имеет либо унаследованную, либо собственную переопределенную версию метода `ToString()`. Применительно к объектам предопределенного типа из CTS, соответствующие версии методов `ToString()` обеспечивают преобразование значения данного типа к строковому виду.

Все сделано для программиста. Реализация метода преобразования значения в строку в конечном счете оказывается скрытой от программиста. Именно поэтому вывод значений предопределенных типов с использованием функций `Write` и `WriteLine` в программах на языке C# осуществляется так легко и просто. В справочнике 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 предопределенным 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 десятичное целое
`E`	Отображение значения в соответствии с научной нотацией
`F`	Отображение значения как `fixed Point`
`G`	Отображение значения в формате с фиксированной точкой или как десятичное целое. Общий формат
`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 или дополняется нулями справа. При выводе целочисленных значений ограничительная константа игнорируется, если количество преобразуемых символов превышает ее значение. В противном случае преобразуемое значение слева дополняется нулями.

Форматирование используется для преобразования значений "обычных" .NET Framework типов данных в строковое представление, да еще 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 (строки?) 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 (имеется в виду "Настройка языковых параметров, представление чисел, денежных единиц, времени и дат").

```
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 приложения, 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); //Символьное представление целочисленного значения.
Console.WriteLine(3.14); //Символьное представление 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 все происходит само собой. Уже на стадии компиляции при выяснении типа выводимого значения подбирается соответствующий вариант перегруженной функции вывода.

При выводе значения определяется его тип, производится соответствующее стандартное преобразование к символьному виду, которое в виде последовательности символов, соответствующей существующей в операционной системе настройке языковых параметров, представления чисел, времени и дат, 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 (см. о перегрузке 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 определенного 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 определенная 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 заключенные в фигурные скобки целочисленные литералы (например, {3}).

При этом способ указания места состоит в следующем:

- CLR индексирует все параметры метода `WriteLine`, идущие 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 нужно вывести 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 дополнительные сведения 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 предопределенным "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);
```

приведет 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 предопределенных символов форматирования и дополнительных целочисленных значений.

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 или дополняется нулями справа. При выводе целочисленных значений ограничительная константа игнорируется, если количество выводимых символов превышает ее 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);//Integers 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
```

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, которая состоит 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, нуль 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, нуль 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 примере ниже показано применение разделителей секций для форматирования.

Следующий код демонстрирует использование разделителей секций при форматировании строки:

```
// Объявляются переменные с различными значениями (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.
```

Приведенная далее 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)

Демонстрация custom number formatting. Еще один пример:

```
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 идет о получении символьной информации. Символьная информация достаточно просто извлекается из входного потока. Однако это не самая большая составляющая общего объема 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 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 неслучайно употребляется слово "попытка". Не каждая строка символов может быть преобразована к определенному базовому типу. Для успешного преобразования предполагается, что символьная строка содержит символьное представление значения в некотором общепринятом формате в соответствии с соответствующей существующей в операционной системе настройкой языковых параметров, представления чисел, времени и дат. С аналогичной ситуацией мы уже сталкивались 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. Для этого требуется 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. Для этого требуется 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. Для этого требуется FileIOPermissionAccess.Write. Попытка чтения из файла, открытого с помощью Truncate, вызывает исключение

FileAccess enumerations:

Члены перечисления	Описание
Read	Доступ к файлу для чтения. Данные могут быть прочитаны из файла. Для обеспечения возможностей чтения/записи может комбинироваться со значением Write
ReadWrite	Доступ к файлу для чтения/записи
Write	Доступ к файлу для записи. Данные могут быть записаны в файл. Для обеспечения возможностей чтения/записи может комбинироваться со значением Read

```
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----------------------------------------");
 }
 }
 }
```

Листинг 11.1.

Пример перенаправления потоков. В файл можно записать информацию, используя привычные классы и методы!

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

namespace OutFile
 {
 class Program
 {

static void Main(string[] args)
 {
 string buff;

FileStream outFstr, inFstr; // Ссылки на файловые потоки.

 // Ссылка на выходной поток. Свойства и методы,
 // которые обеспечивают запись в...
 StreamWriter swr;

 // Ссылка на входной поток. Свойства и методы,
 // которые обеспечивают чтение из...
 StreamReader sr;

 // Класс Console -  средство управления ПРЕДОПРЕДЕЛЕННЫМ потоком.
 // Сохранили стандартный выходной поток,
 // связанный с окошком консольного приложения.
TextWriter twrConsole = Console.Out;

 // Сохранили стандартный входной поток, связанный с буфером клавиатуры.
 TextReader trConsole = Console.In;

inFstr = new FileStream
               (@"F:\Users\Work\readme.txt", FileMode.Open, FileAccess.Read);
 sr = new StreamReader(inFstr); // Входной поток, связанный с файлом.

outFstr = new FileStream
               (@"F:\Users\Work\txt.txt", FileMode.Create, FileAccess.Write);
 swr = new StreamWriter(outFstr); // Выходной поток, связанный с файлом.
 // А вот мы перенастроили предопределенный входной поток.
 // Он теперь связан не с буфером клавиатуры, а с файлом, открытым для чтения.
 Console.SetIn(sr);
 Console.SetOut(swr);

while (true)
 {
 // Но поинтересоваться в предопределенном потоке относительно
 // конца файла невозможно.
 // Такого для предопределенных потоков просто не предусмотрено.
if (sr.EndOfStream) break;

 // А вот читать - можно.
 buff = Console.ReadLine();
 Console.WriteLine(buff);

 }
Console.SetOut(twrConsole);
 Console.WriteLine("12345");
Console.SetOut(swr);
 Console.WriteLine("12345");
 Console.SetOut(twrConsole);
 Console.WriteLine("67890");
 Console.SetOut(swr);
 Console.WriteLine("67890");

sr.Close();
 inFstr.Close();
 swr.Close();
 outFstr.Close();
 }
 }
 }
```

Листинг 11.2.

Введение в программирование на C# 2.0

12. Лекция: Коллекции. Параметризованные классы: версия для печати и PDA
Списки, очереди, двоичные массивы, хэш-таблицы, словари – все это коллекции. Существуют различные типы (классы) коллекций. Объект – представитель данного класса коллекции характеризуется множеством функциональных признаков, определяющих способы работы с элементами (неважно какого типа), которые образуют данную коллекцию

Обзор

Пространство имен `System.Collections` содержит классы и интерфейсы, которые определяют различные коллекции объектов.

Классы

Класс	Описание
`ArrayList`	Служит для реализации интерфейса `IList` с помощью массива с динамическим изменением размера по требованию
`BitArray`	Управляет компактным массивом двоичных значений, представленных логическими величинами, где значение `true` соответствует `1`, а значение `false` соответствует `0`
`CaseInsensitiveComparerpa`	Проверяет равенство двух объектов без учета регистра строк
`CaseInsensitiveHashCodeProvider`	Предоставляет хэш-код объекта, используя алгоритм хэширования, при котором не учитывается регистр строк
`CollectionBase`	Предоставляет абстрактный (MustInherit в Visual Basic) базовый класс для коллекции со строгим типом
`Comparer`	Проверяет равенство двух объектов с учетом регистра строк
`DictionaryBase`	Предоставляет абстрактный (MustInherit в Visual Basic) базовый класс для коллекции пар "ключ-значение" со строгим типом
`Hashtable`	Предоставляет коллекцию пар "ключ-значение", которые упорядочены по хэш-коду ключа
`Queue`	Предоставляет коллекцию объектов, которая обслуживается по принципу "первым пришел — первым вышел" (FIFO)
`ReadOnlyCollectionBase`	Предоставляет абстрактный (MustInherit в Visual Basic) базовый класс для коллекции со строгим типом, которая доступна только для чтения
`SortedList`	Предоставляет коллекцию пар "ключ-значение", которые упорядочены по ключам. Доступ к парам можно получить по ключу и по индексу
`Stack`	Представляет коллекцию объектов, которая обслуживается по принципу "последним пришел — первым вышел" (LIFO)

Интерфейсы

Интерфейс	Описание
`ICollection`	Определяет размер, перечислители и методы синхронизации для всех коллекций
`IComparer`	Предоставляет другим приложениям метод для сравнения двух объектов
`IDictionary`	Предоставляет коллекцию пар "ключ-значение"
`IDictionaryEnumerator`	Осуществляет нумерацию элементов словаря
`IEnumerable`	Предоставляет перечислитель, который поддерживает простое перемещение по коллекции
`IEnumerator`	Поддерживает простое перемещение по коллекции
`IHashCodeProvider`	Предоставляет хэш-код объекта, используя пользовательскую хэш-функцию
`IList`	Предоставляет коллекцию объектов, к которым можно получить доступ отдельно, по индексу

Структуры

Структура	Описание
`DictionaryEntry`	Определяет в словаре пару "ключ-значение", которая может быть задана или получена

Примеры

ArrayList

Неполный перечень свойств и методов приводится ниже.

Конструктор

`ArrayList`	Перегружен. Инициализирует новый экземпляр класса `ArrayList`

Свойства

`Capacity`	Возвращает или задает число элементов, которое может содержать класс `ArrayList`
`Count`	Возвращает число элементов, которое в действительности хранится в классе `ArrayList`
`IsFixedSize`	Возвращает значение, показывающее, имеет ли класс `ArrayList` фиксированный размер
`IsReadOnly`	Возвращает значение, определяющее, доступен ли класс `ArrayList` только для чтения
`IsSynchronized`	Возвращает значение, определяющее, является ли доступ к классу `ArrayList` синхронизированным (потокобезопасным)
`Item`	Возвращает или задает элемент с указанным индексом. В C# это свойство является индексатором класса `ArrayList`

Методы

`Add`	Добавляет объект в конец класса `ArrayList`
`AddRange`	Добавляет элементы интерфейса `ICollection` в конец класса `ArrayList`
`BinarySearch`	Перегружен. Использует алгоритм двоичного поиска для нахождения определенного элемента в отсортированном классе `ArrayList` или в его части
`Clear`	Удаляет все элементы из класса `ArrayList`

Clone	Создает неполную копию класса ArrayList
Contains	Определяет, принадлежит ли элемент классу ArrayList
CopyTo	Перегружен. Копирует класс ArrayList или его часть в одномерный массив
FixedSize	Перегружен. Возвращает обертку списка фиксированного размера, в которой элементы можно изменять, но нельзя добавлять или удалять
GetEnumerator	Перегружен. Возвращает перечислитель, который может осуществлять просмотр всех элементов класса ArrayList
GetRange	Возвращает класс ArrayList, который представляет собой подмножество элементов в исходном классе ArrayList
IndexOf	Перегружен. Возвращает отсчитываемый от нуля индекс первого найденного элемента в классе ArrayList или в его части
Insert	Вставляет элемент в класс ArrayList по указанному индексу
InsertRange	Вставляет элементы коллекции в класс ArrayList по указанному индексу
LastIndexOf	Перегружен. Возвращает отсчитываемый от нуля индекс последнего найденного элемента в классе ArrayList или в его части
Remove	Удаляет первый найденный объект из класса ArrayList
RemoveAt	Удаляет элемент с указанным индексом из класса ArrayList
RemoveRange	Удаляет диапазон элементов из класса ArrayList
Repeat	Возвращает класс ArrayList, элементы которого являются копиями указанного значения
Reverse	Перегружен. Изменяет порядок элементов в классе ArrayList или в его части на обратный
SetRange	Копирует элементы коллекции в диапазон элементов класса ArrayList
Sort	Перегружен. Сортирует элементы в классе ArrayList или в его части
ToArray	Перегружен. Копирует элементы класса ArrayList в новый массив
TrimToSize	Задает значение емкости, равное действительному количеству элементов в классе ArrayList

```
using System;
using System.Collections;
public class SamplesArrayList  {

public static void Main()  {

// Создается и инициализируется объект ArrayList.
ArrayList myAL = new ArrayList();
myAL.Add("Россия,");
myAL.Add("вперед");
myAL.Add("!");

// Свойства и значения ArrayList.
Console.WriteLine( "myAL" );
Console.WriteLine( "\tCount:    {0}", myAL.Count );
Console.WriteLine( "\tCapacity: {0}", myAL.Capacity );
Console.Write( "\tValues:" );
PrintValues( myAL );
}

public static void PrintValues( IEnumerable myList )
{
// Для эффективной работы с объектом ArrayList
// создается перечислитель...
// Перечислитель обеспечивает перебор элементов.
System.Collections.IEnumerator myEnumerator
         = myList.GetEnumerator(); // Перечислитель для myList
while (myEnumerator.MoveNext())
 Console.Write( "\t{0}", myEnumerator.Current );
}
}
```

Листинг 12.1.

Результат:

```
myAL
     Count:    3
     Capacity: 16
     Values:    Россия ,   вперед    !
```

BitArray

Неполный перечень свойств и методов приводится ниже.

Конструкторы

BitArray	Перегружен. Инициализирует новый экземпляр класса BitArray, для которого могут быть указаны емкость и начальные значения

Открытые свойства

Count	Возвращает число элементов, которое хранится в классе BitArray
IsReadOnly	Возвращает значение, определяющее, доступен ли класс BitArray только для чтения
IsSynchronized	Возвращает значение, определяющее, является ли доступ к классу BitArray синхронизированным (потокобезопасным)
Item	Возвращает или задает значение бита по указанному адресу в классе BitArray В языке C# это свойство является индексатором класса BitArray
Length	Возвращает или задает число элементов в классе BitArray

SyncRoot	Возвращает объект, который может быть использован для синхронизации доступа к классу BitArray

Открытые методы

And	Выполняет поразрядную операцию логического умножения элементов текущего класса BitArray с соответствующими элементами указанного класса BitArray
Clone	Создает неполную копию класса BitArray
CopyTo	Копирует целый класс BitArray в совместимый одномерный массив класса Array, начиная с указанного индекса конечного массива
Get	Возвращает значение бита по указанному адресу в классе BitArray
GetEnumerator	Возвращает перечислитель, который может осуществлять просмотр всех элементов класса BitArray
Not	Преобразовывает все двоичные значения в текущем классе BitArray таким образом, чтобы каждому элементу со значением true было присвоено значение false, а каждому элементу со значением false было присвоено значение true
Or	Выполняет поразрядную операцию логического сложения элементов текущего класса BitArray с соответствующими элементами указанного класса BitArray
Set	Задает указанное значение биту по указанному адресу в классе BitArray
SetAll	Задает определенное значение всем битам в классе BitArray
Xor	Выполняет поразрядную операцию "исключающее ИЛИ" для элементов текущего класса BitArray и соответствующих элементов указанного класса BitArray

Пример использования:

```
  using System;
  using System.Collections;
  public class SamplesBitArray  {

     public static void Main()  {

        // Creates and initializes several BitArrays.
        BitArray myBA1 = new BitArray( 5 );

        BitArray myBA2 = new BitArray( 5, false );

        byte[] myBytes = new byte[5] { 1, 2, 3, 4, 5 };
        BitArray myBA3 = new BitArray( myBytes );

        bool[] myBools = new bool[5] { true, false, true, true, false };
        BitArray myBA4 = new BitArray( myBools );

        int[]  myInts  = new int[5] { 6, 7, 8, 9, 10 };
        BitArray myBA5 = new BitArray( myInts );

        // Displays the properties and values of the BitArrays.
        Console.WriteLine( "myBA1" );
        Console.WriteLine( "\tCount:    {0}", myBA1.Count );
        Console.WriteLine( "\tLength:   {0}", myBA1.Length );
        Console.WriteLine( "\tValues:" );
        PrintValues( myBA1, 8 );

        Console.WriteLine( "myBA2" );
        Console.WriteLine( "\tCount:    {0}", myBA2.Count );
        Console.WriteLine( "\tLength:   {0}", myBA2.Length );
        Console.WriteLine( "\tValues:" );
        PrintValues( myBA2, 8 );

        Console.WriteLine( "myBA3" );
        Console.WriteLine( "\tCount:    {0}", myBA3.Count );
        Console.WriteLine( "\tLength:   {0}", myBA3.Length );
        Console.WriteLine( "\tValues:" );
        PrintValues( myBA3, 8 );

        Console.WriteLine( "myBA4" );
        Console.WriteLine( "\tCount:    {0}", myBA4.Count );
        Console.WriteLine( "\tLength:   {0}", myBA4.Length );
        Console.WriteLine( "\tValues:" );
        PrintValues( myBA4, 8 );
        Console.WriteLine( "myBA5" );
        Console.WriteLine( "\tCount:    {0}", myBA5.Count );
        Console.WriteLine( "\tLength:   {0}", myBA5.Length );
        Console.WriteLine( "\tValues:" );
        PrintValues( myBA5, 8 );
     }


     public static void PrintValues( IEnumerable myList, int myWidth )
    {
     System.Collections.IEnumerator myEnumerator = myList.GetEnumerator();
     int i = myWidth;
     while ( myEnumerator.MoveNext() )
   {
    if ( i <= 0 )
    {
      i = myWidth;
      Console.WriteLine();
     }
```

```
            i--;
            Console.Write( "\t{0}", myEnumerator.Current );
        }
           Console.WriteLine();
        }
    }
```

Листинг 12.2.

Результат:

```
myBA1
    Count:    5
    Length:   5
    Values:
    False    False    False    False    False
myBA2
    Count:    5
    Length:   5
    Values:
    False    False    False    False    False
myBA3
    Count:    40
    Length:   40
    Values:
    True     False    False    False    False    False    False    False
    False    True     False    False    False    False    False    False
    True     True     False    False    False    False    False    False
    False    False    True     False    False    False    False    False
    True     False    True     False    False    False    False    False
myBA4
    Count:    5
    Length:   5
    Values:
    True    False    True    True    False
myBA5
    Count:    160
    Length:   160
    Values:
    False    True     True     False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    True     True     True     False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    True     False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    True     False    False    True     False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    True     False    True     False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
    False    False    False    False    False    False    False    False
```

Листинг 12.3.

Queue

Неполный перечень свойств и методов.

Конструктор	
`Queue`	Перегружен. Инициализирует новый экземпляр класса `Queue`
Открытые свойства	
`Count`	Возвращает число элементов, которое хранится в классе `Queue`
`SyncRoot`	Получает объект, который может быть использован для синхронизации доступа к классу `Queue`
Открытые методы	
`Clear`	Удаляет все объекты из класса `Queue`
`Clone`	Создает поверхностную копию класса `Queue`
`Contains`	Определяет, принадлежит ли элемент классу `Queue`
`CopyTo`	Копирует элементы класса `Queue` в существующий одномерный массив класса `Array` по указанному адресу
`Dequeue`	Удаляет и возвращает объект в начале класса `Queue`
`Enqueue`	Добавляет объект в конец класса `Queue`
`GetEnumerator`	Возвращает перечислитель, который может перемещаться по классу `Queue`
`Peek`	Возвращает объект в начале класса `Queue`, но не удаляет его
`ToArray`	Копирует элементы класса `Queue` в новый массив
`TrimToSize`	Задает значение емкости, равное действительному количеству элементов в классе `Queue`

Пример:

```
using System;
using System.Collections;
public class SamplesQueue
{

   public static void Main()
  {
      // Creates and initializes a new Queue.
      Queue myQ = new Queue();
      myQ.Enqueue("Россия,");
      myQ.Enqueue("вперед");
      myQ.Enqueue("!");

    // Displays the properties and values of the Queue.
    Console.WriteLine( "myQ" );
    Console.WriteLine( "\tCount:    {0}", myQ.Count );
    Console.Write( "\tValues:" );
    PrintValues( myQ );
   }


 public static void PrintValues( IEnumerable myCollection )
  {
    System.Collections.IEnumerator myEnumerator
                                    = myCollection.GetEnumerator();
      while ( myEnumerator.MoveNext() )
         Console.Write( "\t{0}", myEnumerator.Current );
   }
}
```

Листинг 12.4.

Результат:

```
myQ
     Count:    3
     Values:    Россия ,   вперед    !
```

Stack

Открытые конструкторы	
`Stack`	Перегружен. Инициализирует новый экземпляр класса `Stack`
Открытые свойства	
`Count`	Возвращает число элементов, которое хранится в классе `Stack`
`IsSynchronized`	Возвращает значение, определяющее, является ли доступ к классу `Stack` синхронизированным (потокобезопасным)
`SyncRoot`	Возвращает объект, который может быть использован для синхронизации доступа к классу `Stack`
Открытые методы	
`Clear`	Удаляет все объекты из класса `Stack`
`Clone`	Создает неполную копию класса `Stack`
`Contains`	Определяет, принадлежит ли элемент классу `Stack`
`CopyTo`	Копирует элементы класса `Stack` в существующий одномерный массив класса `Array`, начиная с указанного индекса массива
`GetEnumerator`	Интерфейс `IEnumerator` для класса `Stack`
`Peek`	Возвращает самый верхний объект класса `Stack`, но не удаляет его
`Pop`	Удаляет и возвращает верхний объект класса `Stack`
`Push`	Вставляет объект в начало класса `Stack`
`Synchronized`	Возвращает синхронизированную (потокобезопасную) обертку класса `Stack`
`ToArray`	Копирует элементы класса `Stack` в новый массив

Класс `Stack` допускает в качестве действительного значение "пустая ссылка", а также допускает наличие повторяющихся элементов.

Ниже приведен пример создания и инициализации класса `Stack` и способ вывода его значений:

```
using System;
using System.Collections;
public class SamplesStack  {

 public static void Main()
 {

// Creates and initializes a new Stack.
     Stack myStack = new Stack();
     myStack.Push("Hello");
     myStack.Push("world");
     myStack.Push("!");

     // Displays the properties and values of the Stack.
     Console.WriteLine( "myStack" );
     Console.WriteLine( "\tCount:    {0}", myStack.Count );
     Console.Write( "\tValues:" );
     PrintValues( myStack );
  }
```

```
  public static void PrintValues( IEnumerable myCollection )
   {
   System.Collections.IEnumerator myEnumerator = myCollection.GetEnumerator();
    while ( myEnumerator.MoveNext() )
    Console.Write( "\t{0}", myEnumerator.Current );
    Console.WriteLine();
   }
 }
```

Листинг 12.5.

Результат:

```
myStack
     Count:    3
     Values:    !    hello    Hello
```

Перечислитель

Наследует интерфейс `IEnumerator`, который является основным для всех перечислителей. Поддерживает простое перемещение по коллекции.

Открытые свойства	
`Current`	Возвращает текущий элемент коллекции
Открытые методы	
`MoveNext`	Перемещает перечислитель на следующий элемент коллекции
`Reset`	Устанавливает перечислитель в исходное положение перед первым элементом коллекции

Перечислитель позволяет считывать (только считывать!) информацию (данные) из коллекции. Перечислители не используются для изменения содержания коллекции. Для этого применяются специфические методы данной коллекции (`Enqueue, Dequeue, Push, Pop`).

Вновь созданный перечислитель размещается перед первым элементом коллекции. Метод `Reset` возвращает перечислитель обратно в положение перед первым элементом коллекции.

В этом положении обращение к свойству `Current` приводит к возникновению исключения. Поэтому необходимо вызвать метод `MoveNext`, чтобы переместить перечислитель на первый элемент коллекции до считывания значения свойства `Current`.

Свойство `Current` не меняет своего значения (возвращает ссылку на один и тот же член коллекции), пока не будет вызван метод `MoveNext` или `Reset`.

Метод `MoveNext` обеспечивает изменение значения свойства `Current`.

Завершение перемещения по коллекции приводит к установке перечислителя после последнего элемента коллекции. При этом вызов метода `MoveNext` возвращает значение `false`. Если последний вызов метода `MoveNext` вернул значение `false`, обращение к свойству `Current` приводит к возникновению исключения. Последовательный вызов методов `Reset` и `MoveNext` приводит к перемещению перечислителя на первый элемент коллекции.

Перечислитель действителен до тех пор, пока в коллекцию не вносятся изменения. Если в коллекцию вносятся изменения (добавляются или удаляются элементы коллекции), перечислитель становится недействительным, а следующий вызов методов `MoveNext` или `Reset` приводит к возникновению исключения `InvalidOperationException`.

После изменения коллекции в промежутке между вызовом метода `MoveNext` и новым обращением к свойству `Current`, свойство `Current` возвращает текущий элемент коллекции, даже если перечислитель уже недействителен.

Перечислитель не имеет монопольного доступа к коллекции, поэтому перечисление в коллекции не является потокобезопасной операцией. Даже при синхронизации коллекции другие потоки могут изменить ее, что приводит к созданию исключения при перечислении. Чтобы обеспечить потокобезопасность при перечислении, можно либо заблокировать коллекцию на все время перечисления, либо перехватывать исключения, которые возникают в результате изменений, внесенных другими потоками.

Введение в программирование на C# 2.0

13. Лекция: Шаблоны: версия для печати и PDA
В данной лекции рассказывается о мощном средстве языка C# - шаблонах. Шаблонный класс (и функция) обеспечивают стандартную реализацию какой-либо функциональности для подстановочного класса

Ранее рассмотренные классы (`ArrayList, Queue, Stack`) могут поддерживать коллекции произвольных типов — благодаря тому, что любой класс является производным (либо наследует его) класса `object`. Именно поэтому и можно организовать коллекцию объектов производного типа. Для стека и очереди это всего лишь экземпляры класса `object`.

Например, `Stack` сохраняет свои данные в массиве `object`, и два его метода, `Push` и `Pop`, используют `object` для приема и возвращения данных:

```
public class Stack{
object[] items;
int count;
public void Push(object item) {...}
public object Pop() {...}
}
```

В этом заключается гибкость и универсальность классов коллекций, и в этом же заключаются основные проблемы, связанные с использованием коллекций.

В коллекции можно разместить объекты произвольного типа. После извлечения объекта из коллекции (например, методом `Pop`) полученное значение (ссылка) должна быть явно приведена к соответствующему типу:

```
Class XXX
{
:::::
}

Stack stack = new Stack();
stack.Push(new XXX());
XXX c = (XXX)stack.Pop();
```

Если значение типа-значения (например, `int`) передается в метод `Push`, оно автоматически упаковывается. При извлечении значения оно должно быть распаковано с явным приведением типа:

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

Такие операции упаковки/распаковки увеличивают непроизводительные издержки, поскольку приводят к динамическим перераспределениям памяти и динамическим проверкам типов.

Следующая проблема заключается в невозможности задания типа данных, помещаемых в коллекцию. Объект – представитель класса `XXX` может быть помещен в стек и после извлечения преобразован в другой тип:

```
Stack stack = new Stack();
stack.Push(new XXX());
string s = (string)stack.Pop();
```

Приведенный здесь код не вызывает возражений транслятора, хотя и является примером некорректного использования класса `Stack`.

Проблема проявится лишь во время выполнения кода, о чем станет известно после генерации исключения `InvalidCastException`.

Таким образом шаблоны позволяют избежать вышеперечисленных трудностей.

Общее представление

В C# можно объявлять шаблоны классов и шаблоны функций. Шаблоны классов и шаблоны функций – две разные вещи, которые в общем случае могут сосуществовать в рамках одного и того же объявления.

Объявление любого шаблона предполагает использование специальных обозначений, которые называются параметрами шаблона. Традиционно для этого применяются однобуквенные обозначения: `A, ..., K, L, M, N, ..., T, ..., Z`.

При объявлении шаблонного класса или функции параметры шаблона заменяются на конкретные имена классов.

Шаблонный класс – это класс, который строится на основе ранее объявленного шаблона для определенного класса (этот класс мы будем называть подстановочным классом). Шаблонный класс (и функция) обеспечивают стандартную реализацию какой-либо функциональности для подстановочного класса — например, построения специализированной коллекции для множества объектов данного типа.

Параметры шаблонов позволяют параметризовать шаблоны классов, структур, интерфейсов, делегатов и функций классами, для которых эти шаблоны создавались с целью расширения их функциональности.

Ниже приводится пример объявления шаблона класса `Stack` с параметром типа `<T>`.

Параметр `T` в объявлении рассматриваемого шаблонного класса обозначает место подстановки реального типа. Он используется для обозначения типа элемента для внутреннего массива элементов, типа параметра метода `Push` и типа результата для метода `Pop`:

```
public class Stack<T>
{
T[] items;
```

```
int count;
public void Push(T item) {...}
public T Pop() {...}
}
```

Вместо преобразований в и из класса `object`, параметризованные коллекции (`Stack<T>` в том числе) принимают на сохранение значения типа, для которого они созданы, и сохраняют данные этого типа без дополнительных преобразований. При использовании шаблонного класса `Stack<T>`, вместо `T` подставляется реальный тип. В следующем примере `int` задается как аргумент типа (type argument) для `T`:

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

Тип `Stack<int>` называют составным или шаблонным типом (constructed type). При объявлении этого типа каждое вхождение `T` заменяется аргументом типа `int`.

Когда объект – представитель класса `Stack<int>` создается, массив `items` оказывается объектом типа `int[]`, а не `object[]`.

Методы `Push` и `Pop` класса `Stack<int>` также оперируют значениями `int`, при этом помещение в стек значений другого типа приводит к ошибкам компиляции (а не ошибкам времени выполнения).

Также не требуется явного приведения извлеченных из коллекции значений к их исходному типу.

Шаблоны обеспечивают строгий контроль типов. Это означает, например, что помещение `int` в стек объектов `XXX` является ошибкой.

Точно так же, как `Stack<int>` ограничен возможностью работы только со значениями типа `int`, так и `Stack<XXX>` ограничен объектами типа `XXX`.

А компиляция последних двух строк следующего примера выдаст ошибку:

```
Stack<XXX> stack = new Stack<XXX>();
stack.Push(new XXX());
XXX xxx = stack.Pop();
stack.Push(3);                // Ошибка несоответствия типов
int x = stack.Pop();     // Ошибка несоответствия типов
```

Объявление шаблона может содержать любое количество параметров типа. В приведенном выше примере в `Stack<T>` есть только один параметр типа, но шаблонный класс `Dictionary` должен иметь два параметра типа: один для подстановочного типа ключей, другой для подстановочного типа значений:

```
public class Dictionary<K,V>
{
public void Add(K key, V value) {...}
public V this[K key] {...}
}
```

При объявлении шаблонного класса на основе шаблона `Dictionary<K,V>` должны быть заданы два аргумента типа:

```
Dictionary<string,XXX> dict = new Dictionary<string,XXX>();

// В словарь добавляется объект типа XXX,
// проиндексированный строкой x1
dict.Add("x1", new XXX());

// Извлечение из словаря элемента, проиндексированного строкой "x1"
XXX xxx = dict["x1"];
```

Пример использования шаблонов: сортировка.

Старая задача, новые решения с использованием предопределенных шаблонов классов и интерфейсов...

```
using System;
 using System.Collections;
 using System.Collections.Generic;

namespace PatternArrays
 {

 // Данные для массива элементов.
 // Подлежат сортировке в составе шаблонного массива методом Sort.
class Points
 {
 public int x;
 public int y;

public Points(int key1, int key2)
 {
     x = key1;
     y = key2;
 }

 // Вычисляется расстояние от начала координат.
 public int R
 {
     get
    {
         return (int)(Math.Sqrt(x * x + y * y));
     }
```

```
 }
 }


 // ...ШАБЛОННЫЙ КОМПАРЕР на основе шаблона интерфейса...


class myComparer : IComparer<Points>
 {

 // Предлагаемый ШАБЛОННЫЙ метод сравнения возвращает разность расстояний
 // двух точек (вычисляется по теореме Пифагора) от начала координат
 // -  точки с координатами (0,0). Чем ближе точки к началу координат
 // -  тем они меньше. Не требуется никаких явных приведений
типа.
 // Шаблон настроен на работу с классом Points.


int IComparer<Points>.Compare(Points p1, Points p2)
 {
     return (p1.R - p2.R);
 }
 }
 // После реализации соответствующего ШАБЛОННОГО интерфейса объект-
 // КОМПАРЕР обеспечивает реализацию стандартного алгоритма сортировки.

class Class1
 {
 static void Main(string[] args)
 {
     // Объект - генератор "случайных" чисел.
    Random rnd = new Random();

    int i;
     // Очередь Points.
     Queue<Points> qP = new Queue<Points>();
     // Шаблонный перечислитель. Предназначен для обслуживания
     // шаблонной очереди элементов класса Points.
     IEnumerator<Points> enP;
     // Сортировка поддерживается классом Array.
     Points[] pp;

    // Создали Компарер, способный сравнивать пары
     // объектов - представителей класса Points.
    myComparer c = new myComparer();

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

    // Проинициализировали массив объектов - представителей класса Points.
    for (i = 0; i < 10; i++)
     {
         qP.Enqueue(new Points(rnd.Next(0, 10), rnd.Next(0, 10)));
     }

    enP = ((IEnumerable<Points>)(qP)).GetEnumerator();
     for (i = 0; enP.MoveNext(); i++)
     {
         Console.WriteLine("{0}: {1},{2}", i, enP.Current.x, enP.Current.y);
     }

    // Сортируются элементы массива типа Points, который формируется на
   // основе шаблонной очереди.
   // Условием успешной сортировки элементов массива является реализация
   // интерфейса IComparer. Если Компарер не сумеет справиться с
   // поставленной задачей - будет возбуждено исключение.
   // На основе очереди построили массив.
    pp = qP.ToArray();

    // А саму очередь можно почистить!
     qP.Clear();

    try
    {
         Array.Sort<Points>(pp, c);
     }
     catch (Exception ex)
     {
         Console.WriteLine(ex);
     }

    // Сортировка произведена, очередь восстановлена.
     for (i = 0; i < 10; i++) qP.Enqueue(pp[i]);

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

    enP = ((IEnumerable<Points>)(qP)).GetEnumerator();
     for (i = 0; enP.MoveNext(); i++)
     {
         Console.WriteLine("{0}: {1},{2}", i, enP.Current.x, enP.Current.y);
     }
```

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

Листинг 13.1.

Шаблоны классов и функций. Ограничения параметра типа

До настоящего момента обсуждались вопросы построения шаблонных классов на основе предопределенных (включенных в состав библиотеки классов) шаблонов классов. В этом разделе на простом примере обсуждается техника объявления собственных шаблонов классов и шаблонов функций.

Шаблонный класс обеспечивает стандартную реализацию дополнительной функциональности на основе ранее объявленных подстановочных классов.

Эта дополнительная функциональность может накладывать дополнительные ограничения на подстановочный класс. Например, для успешной работы объекта шаблонного класса подстановочный класс должен наследовать определенному интерфейсу. Иначе функциональность шаблонного класса просто невозможно будет реализовать.

Для формирования ограничений на подстановочные классы в C# используется механизм ограничителей параметров шаблона — он вводится при объявлении шаблона с помощью ключевого слова where, за которым могут располагаться имя параметра типа и список типов класса или интерфейса либо конструктор – ограничение `new()`:

```
using System;
 using System.Collections;
 using System.Collections.Generic;

namespace PatternArrays
 {

 //========== Это заголовок шаблона класса W ==========
 // Шаблон класса своими руками. T – параметр шаблона.
 // Шаблонный класс -  это класс-шаблон, который детализируется
 // подстановочным классом.
 // При создании шаблонного класса вхождения параметра шаблона
 // (в данном случае это T) замещаются именем подстановочного
 // класса. Разработчик шаблона класса может выдвигать требования
 // относительно характеристик подстановочного класса.
 // Для этого используются специальные языковые конструкции,
 // называемые ОГРАНИЧИТЕЛЯМИ ПАРАМЕТРА ШАБЛОНА.
 // ОГРАНИЧИТЕЛЬ ПАРАМЕТРА ШАБЛОНА формулирует требования для
 // подстановочного класса.
 class W<T> where T: IComparable, new()
 //                         Ограничитель параметра шаблона new() – особый
 //                         ограничитель.
 //                   Во-первых, в списке ограничителей шаблона
 //                   он всегда последний.
 //                   Во-вторых, этот ограничитель НЕ ограничивает.
 //                 Он ОБЕСПЕЧИВАЕТ обязательное выполнение явно
 //                 заданного конструктора умолчания для
 //                 подстановочного класса в шаблонном
 //                 классе. Это единственный способ заставить
 //                 выполниться конструктор умолчания
 //                 подстановочного класса при создании
 //                 объекта шаблонного класса.
 //                 Ограничитель подстановочного класса.
 //                 Шаблонный класс строится на основе шаблона и множества
 //                 подстановочных классов, которыми замещаются параметры
 //                 шаблона. Таким образом ограничители подстановочного
 //                 класса формулируют требования по поводу "родословной"
 //                 подстановочного класса.
 //                 В данном случае претендент на замещение параметра T
 //                 в шаблоне W обязательно должен наследовать интерфейс
 //                 IComparable.
 {

 // Вот место, которое предназначено объекту подстановочного класса.
 // Объект - представитель шаблонного класса включает объект,
 // представляющий подстановочный класс.
public T t;

 // Конструктор шаблона.
 // Вот по какой схеме производится встраивание объекта-представителя
 // подстановочного класса. Всего лишь для того, чтобы эта схема
 // построения объекта - представителя шаблонного класса работала,
 // в объявлении шаблона должен присутствовать ограничитель параметра
 // шаблона new(). Его отсутствие приводит к возникновению ошибки
 // компиляции. С каких это пор необходимое требование стали
 // называть ограничением?
  public W()
  {
      t = new T();
  }

 // Сравнение объектов в шаблоне. Обращение к функциям сравнения
 // регламентировано стандартными интерфейсами.
 // Полиморфизм через интерфейсы в действии.
  public int wCompare(T t)
```

```
 {
   return ((IComparable)this.t).CompareTo(t);
  }

 // А вот замечательный шаблон функции.
 // Он реализован в рамках класса-шаблона W.
 // Эта функция предназначена для формирования шаблонных очередей
 // из входных массивов объектов - представителей подстановочного
 // класса, представленного параметром шаблона Z.
 // Между прочим, такое обозначение параметра ничуть не хуже любого
 // другого. Более того, если бы здесь было использовано старое
 // обозначение параметра, транслятор выступил бы с предупреждением
 // по поводу того, что две разных сущности (параметр шаблона для
 // шаблона класса и параметр шаблона для параметра функции)
 // в рамках одного и того же объявления имеют одинаковые обозначения.
  public void QueueFormer<Z>(Queue<Z> queue, params Z[] values)
  {
      foreach (Z z in values)
      {
          queue.Enqueue(z);
     }
  }
 }

 //=================================================================
 // Вот классы-кандидаты на подстановку в шаблон.
 // Первый класс подходит, а второй -  не подходит!
 // Все решается на этапе трансляции.
 //=================================================================
  class xPoints : IComparable
 {
     // Объект-генератор "случайных" чисел.
    static Random rnd = new Random();

    public int x;
     public int y;

    public xPoints()
     {
         x = rnd.Next(0, 100);
         y = rnd.Next(0, 100);
     }

     // Ничто не может помешать классу иметь
      // различные версии конструкторов!
      public xPoints(int x, int y)
      {
          this.x = x;
          this.y = y;
      }

    // Вычисляется расстояние от начала координат.
     public int R
    {
         get
        {
             return (int)(Math.Sqrt(x * x + y * y));
         }
     }

    // После реализации соответствующего интерфейса объект-КОМПАРЕР
    // обеспечивает реализацию алгоритма сравнения.

    public int CompareTo(object p)
     {
         return (this.R - ((xPoints)p).R);
     }
 }

class yPoints
 {
     // Объект-генератор "случайных" чисел.
    static Random rnd = new Random();

    public int x;
     public int y;

    public yPoints()
     {
         x = rnd.Next(0, 100);
         y = rnd.Next(0, 100);
     }

    // Шаблон функции в рамках объявления "обычного" класса.
    // Функция предназначена для формирования шаблонных магазинов
    // из входных массивов объектов - представителей подстановочного
    // класса, представленного параметром шаблона T.
    public void  StackFormer<T>(Stack<T> stack, params T[] values)
     {
         foreach (T t in values)
```

```
        {
            stack.Push(t);
        }
    }
}

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

       W<xPoints> xw0 = new W<xPoints>();
        W<xPoints> xw1 = new W<xPoints>();

// Объекты - представители шаблонного класса можно сравнивать
// в результате
 // реализации интерфейса IComparable.
       if (xw0.wCompare(xw1.t) == 0) Console.WriteLine("Yes");
        else                         Console.WriteLine("No");

// В силу ограничений параметра шаблона T, следующий код
// для подстановочного
// класса в принципе нереализуем.============================

   //W<yPoints> yw0 = new W<yPoints>();
   //W<yPoints> yw1 = new W<yPoints>();

   //if (yw0.ww(yw1.t) == 0) Console.WriteLine("Yes");
   //else                    Console.WriteLine("No");
        //=====================================================

   // Демонстрация использования шаблона функции.
   // На основе подстановочного класса сформировали
   // шаблонную функцию для подстановочного класса,
   // которая обслуживает шаблонные очереди, формируемые на основе
   // предопределенного шаблона класса Queue<...>.
    Queue<xPoints> xpQueue = new Queue<xPoints>();
    xw1.QueueFormer<xPoints>(xpQueue,
                                    new xPoints(0, 9),
                                    new xPoints(1, 8),
                                    new xPoints(2, 7),
                                    new xPoints(3, 6),
                                    new xPoints(4, 5),
                                    new xPoints(5, 4),
                                    new xPoints(6, 3),
                                    new xPoints(7, 2),
                                    new xPoints(8, 1),
                                    new xPoints(9, 0)
                            );

   // Шаблоны классов и шаблоны функций концептуально не связаны.
   // В C# это самостоятельные и независимые конструкции.
   // Шаблон функции может быть объявлен где угодно -
   // в шаблоне класса и в рамках объявления "обычного" класса.
   // При объявлении шаблона функции ограничения на свойства
   // подстановочного класса, представленного параметром шаблона,
   // не упоминаются.
   // В силу ограничений на параметр шаблона класс yPoints в принципе
   // не может быть использован для построения шаблонного класса
   // на основе шаблона class W<T>.
   // Однако этот же самый класс может быть использован для построения
   // шаблонной функции в шаблонном классе, созданном на основе
   // подстановочного класса xPoints!
   Queue<yPoints> ypQueue = new Queue<yPoints>();
    xw1.QueueFormer<yPoints>(ypQueue,
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints()
                            );

// А вот применение шаблона функции, объявленного в "обычном" классе.
// Создали объект класса, содержащего шаблон функции
// по обслуживанию очередей.
yPoints yp = new yPoints();

// Ссоздали шаблонный стек и воспользовались шаблоном функции,
// объявленной yPoints в классе.
Stack<xPoints> xpStack = new Stack<xPoints>();
yp.StackFormer<xPoints>(xpStack,
                                new xPoints(),
                                new xPoints(),
                                new xPoints(),
```

```
                                    new xPoints(),
                                    new xPoints(),
                                    new xPoints(),
                                    new xPoints(),
                                    new xPoints(),
                                    new xPoints(),
                                    new xPoints()
                                   );


    }
 }
 }
```

Листинг 13.2.

Пример использования шаблонов: сортировка

Старая задача, новые решения с использованием предопределенных шаблонов классов и интерфейсов...

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace PatternArrays
{

// Данные для массива элементов.
// Подлежат сортировке в составе шаблонного массива методом Sort.

class Points
{
public int x;
public int y;

public Points(int key1, int key2)
{
     x = key1;
     y = key2;
 }

 // Вычисляется расстояние от начала координат.
 public int R
 {
     get
    {
         return (int)(Math.Sqrt(x * x + y * y));
     }
 }
 }


 // ...ШАБЛОННЫЙ КОМПАРЕР на основе шаблона интерфейса...


class myComparer : IComparer<Points>
 {
 // Предлагаемый ШАБЛОННЫЙ метод сравнения возвращает разность расстояний
 // двух точек (вычисляется по теореме Пифагора) от начала координат
 // - точки с координатами (0,0). Чем ближе точки к началу координат
 // - тем они меньше. Не требуется никаких явных приведений типа.
 // Шаблон настроен на работу с классом Points.


int IComparer<Points>.Compare(Points p1, Points p2)
 {
     return (p1.R - p2.R);
 }
 }


 // После реализации соответствующего ШАБЛОННОГО интерфейса
 // объект-КОМПАРЕР обеспечивает реализацию стандартного алгоритма
 // сортировки.


class Class1
 {
 static void Main(string[] args)
 {
     // Объект-генератор "случайных" чисел.
    Random rnd = new Random();

    int i;
     // Очередь Points.
     Queue<Points> qP = new Queue<Points>();
     // Шаблонный перечислитель. Предназначен для обслуживания
     // шаблонной очереди элементов класса Points.
     IEnumerator<Points> enP;
```

```
     // Сортировка поддерживается классом Array.
     Points[] pp;

    // Создали Компарер, способный сравнивать пары
     // объектов - представителей класса Points.
    myComparer c = new myComparer();

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

    // Проинициализировали массив объектов - представителей класса Points.
    for (i = 0; i < 10; i++)
     {
         qP.Enqueue(new Points(rnd.Next(0, 10), rnd.Next(0, 10)));
     }

    enP = ((IEnumerable<Points>)(qP)).GetEnumerator();
     for (i = 0; enP.MoveNext(); i++)
     {
         Console.WriteLine("{0}: {1},{2}", i, enP.Current.x, enP.Current.y);
     }

    // Сортируются элементы массива типа Points, который формируется на
    // основе шаблонной очереди.
    // Условием успешной сортировки элементов массива является
    // реализация интерфейса IComparer. Если Компарер не сумеет
    // справиться с поставленной задачей – будет возбуждено исключение.
    // На основе очереди построили массив.

    pp = qP.ToArray();

    // А саму очередь можно почистить!
     qP.Clear();

    try
    {
         Array.Sort<Points>(pp, c);
     }
     catch (Exception ex)
     {
         Console.WriteLine(ex);
     }
    // Сортировка произведена, очередь восстановлена.
     for (i = 0; i < 10; i++) qP.Enqueue(pp[i]);

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

    enP = ((IEnumerable<Points>)(qP)).GetEnumerator();
     for (i = 0; enP.MoveNext(); i++)
     {
         Console.WriteLine("{0}: {1},{2}", i, enP.Current.x, enP.Current.y);
     }

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

Листинг 13.3.

Шаблоны классов и функций. Ограничения параметра типа

До настоящего момента обсуждались вопросы построения шаблонных классов на основе предопределенных (включенных в состав библиотеки классов) шаблонов классов. В этом разделе на простом примере обсуждается техника объявления собственных шаблонов классов и шаблонов функций.

Шаблонный класс обеспечивает стандартную реализацию дополнительной функциональности на основе ранее объявленных подстановочных классов.

Эта дополнительная функциональность может накладывать дополнительные ограничения на подстановочный класс. Например, для успешной работы объекта шаблонного класса подстановочный класс должен наследовать определенному интерфейсу. Иначе функциональность шаблонного класса просто невозможно будет реализовать.

Для формирования ограничений на подстановочные классы в C# используется механизм ограничителей параметров шаблона — он вводится при объявлении шаблона с помощью ключевого слова where, за которым могут располагаться имя параметра типа и список типов класса или интерфейса либо конструктор-ограничение `new()`.

```
using System;
 using System.Collections;
 using System.Collections.Generic;
namespace PatternArrays
 {

 //========== Это заголовок шаблона класса W ==========
 // Шаблон класса своими руками. T -  параметр шаблона.
 // Шаблонный класс -  это класс-шаблон, который детализируется
 // подстановочным классом.
 // При создании шаблонного класса вхождения параметра шаблона
 // (в данном случае это T) замещаются именем подстановочного
 // класса. Разработчик шаблона класса может выдвигать требования
 // относительно характеристик подстановочного класса.
```

```
 // Для этого используются специальные языковые конструкции,
 // называемые ОГРАНИЧИТЕЛЯМИ ПАРАМЕТРА ШАБЛОНА.
 // ОГРАНИЧИТЕЛЬ ПАРАМЕТРА ШАБЛОНА формулирует требования для
 // подстановочного класса.
 class W<T> where T: IComparable, new()
 //                 Ограничитель параметра шаблона new() - особый
 //                 ограничитель.
 //                 Во-первых, в списке ограничителей шаблона
 //                 он всегда последний.
 //                 Во-вторых, этот ограничитель НЕ ограничивает.
 //                 Он ОБЕСПЕЧИВАЕТ обязательное выполнение явно
 //                 заданного конструктора умолчания для
 //                 подстановочного класса в шаблонном
 //                 классе. Это единственный способ заставить
 //                 выполниться конструктор умолчания
 //                 подстановочного класса при создании
 //                 объекта шаблонного класса.
 //                 Ограничитель подстановочного класса.
 //                 Шаблонный класс строится на основе шаблона и множества
 //                 подстановочных классов, которыми замещаются параметры
 //                 шаблона. Таким образом ограничители подстановочного
 //                 класса формулируют требования по поводу "родословной"
 //                 подстановочного класса.
 //                 В данном случае претендент на замещение параметра T
 //                 в шаблоне W обязательно должен наследовать интерфейс
 //                 IComparable.
 {
 // Вот место, которое предназначено объекту подстановочного класса.
 // Объект - представитель шаблонного класса включает объект,
 // представляющий подстановочный класс.
public T t;

 // Конструктор шаблона.
 // Вот по какой схеме производится встраивание объекта - представителя
 // подстановочного класса. Всего лишь для того, чтобы эта схема
 // построения объекта - представителя шаблонного класса работала,
 // в объявлении шаблона должен присутствовать ограничитель параметра
 // шаблона new(). Его отсутствие приводит к возникновению ошибки
 // компиляции. С каких это пор необходимое требование стали называть
 // ограничением?
  public W()
  {
      t = new T();
  }

 // Сравнение объектов в шаблоне. Обращение к функциям сравнения
 // регламентировано стандартными интерфейсами.
 // Полиморфизм через интерфейсы в действии.
  public int wCompare(T t)
 {
   return ((IComparable)this.t).CompareTo(t);
  }

 // А вот замечательный шаблон функции.
 // Он реализован в рамках класса-шаблона W.
 // Эта функция предназначена для формирования шаблонных очередей
 // из входных массивов объектов - представителей подстановочного
 // класса, представленного параметром шаблона Z.
 // Между прочим, такое обозначение параметра ничуть не хуже любого
 // другого.
 // Более того, если бы здесь было использовано старое обозначение
 // параметра, транслятор выступил бы с предупреждением по поводу того, что
 // две разных сущности (параметр шаблона для шаблона класса и параметр
 // шаблона для параметра функции) в рамках одного и того же объявления
 // имеют одинаковые обозначения.
  public void QueueFormer<Z>(Queue<Z> queue, params Z[] values)
  {
      foreach (Z z in values)
      {
          queue.Enqueue(z);
     }
  }
 }

 //=============================================================
 // Вот классы-кандидаты на подстановку в шаблон.
 // Первый класс подходит, а второй -  не подходит!
 // Все решается на этапе трансляции.
 //=============================================================
  class xPoints : IComparable
 {
     // Объект-генератор "случайных" чисел.
    static Random rnd = new Random();

    public int x;
     public int y;

    public xPoints()
     {
         x = rnd.Next(0, 100);
```

```
        y = rnd.Next(0, 100);
    }

    // Ничто не может помешать классу иметь
     // различные версии конструкторов!
     public xPoints(int x, int y)
     {
         this.x = x;
         this.y = y;
     }

   // Вычисляется расстояние от начала координат.
    public int R
   {
        get
       {
            return (int)(Math.Sqrt(x * x + y * y));
        }
    }

   // После реализации соответствующего интерфейса объект-КОМПАРЕР
   // обеспечивает реализацию алгоритма сравнения.

   public int CompareTo(object p)
    {
        return (this.R - ((xPoints)p).R);
    }
}

class yPoints
 {
    // Объект-генератор "случайных" чисел.
   static Random rnd = new Random();

   public int x;
    public int y;

   public yPoints()
    {
        x = rnd.Next(0, 100);
        y = rnd.Next(0, 100);
    }

    // Шаблон функции в рамках объявления "обычного" класса.
    // Функция предназначена для формирования шаблонных магазинов
    // из входных массивов объектов - представителей подстановочного
    // класса, представленного параметром шаблона T.
   public void  StackFormer<T>(Stack<T> stack, params T[] values)
    {
        foreach (T t in values)
        {
            stack.Push(t);
        }
    }
}
//=============================================================
class Class1
{
    static void Main(string[] args)
    {

       W<xPoints> xw0 = new W<xPoints>();
        W<xPoints> xw1 = new W<xPoints>();

// Объекты - представители шаблонного класса можно сравнивать
// в результате реализации интерфейса IComparable.
       if (xw0.wCompare(xw1.t) == 0) Console.WriteLine("Yes");
        else                          Console.WriteLine("No");

 // В силу ограничений параметра шаблона T, следующий код для
 // подстановочного класса в принципе нереализуем.=================

   //W<yPoints> yw0 = new W<yPoints>();
    //W<yPoints> yw1 = new W<yPoints>();

   //if (yw0.ww(yw1.t) == 0) Console.WriteLine("Yes");
    //else                    Console.WriteLine("No");
        //=====================================================

   // Демонстрация использования шаблона функции.
    // На основе подстановочного класса сформировали
    // шаблонную функцию для подстановочного класса,
    // которая обслуживает шаблонные очереди, формируемые на основе
    // предопределенного шаблона класса Queue<...>.
    Queue<xPoints> xpQueue = new Queue<xPoints>();
    xw1.QueueFormer<xPoints>(xpQueue,
                                    new xPoints(0, 9),
                                    new xPoints(1, 8),
                                    new xPoints(2, 7),
                                    new xPoints(3, 6),
```

```
                                    new xPoints(4, 5),
                                    new xPoints(5, 4),
                                    new xPoints(6, 3),
                                    new xPoints(7, 2),
                                    new xPoints(8, 1),
                                    new xPoints(9, 0)
                                );

        // Шаблоны классов и шаблоны функций концептуально не связаны.
        // В C# это самостоятельные и независимые конструкции.
        // Шаблон функции может быть объявлен где угодно -
        // в шаблоне класса и в рамках объявления "обычного" класса.
        // При объявлении шаблона функции ограничения на свойства
        // подстановочного класса, представленного параметром шаблона,
        // не упоминаются.
        // В силу ограничений на параметр шаблона класс yPoints в принципе
        // не может быть использован для построения шаблонного класса
        // на основе шаблона class W<T>.
        // Однако этот же самый класс может быть использован для построения
        // шаблонной функции в шаблонном классе, созданном на основе
        // подстановочного класса xPoints!
        Queue<yPoints> ypQueue = new Queue<yPoints>();
    xw1.QueueFormer<yPoints>(ypQueue,
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints(),
                                    new yPoints()
                                );

  // А вот применение шаблона функции, объявленного в "обычном" классе.
  // Создали объект класса, содержащего шаблон функции
  // по обслуживанию очередей.
 yPoints yp = new yPoints();

  // Ссоздали шаблонный стек и воспользовались шаблоном функции,
  // объявленной yPoints в классе.
  Stack<xPoints> xpStack = new Stack<xPoints>();
  yp.StackFormer<xPoints>(xpStack,
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints(),
                                  new xPoints()
                                 );


    }
  }
  }
```

Листинг 13.4.

Nullable-типы

Nullable-типы (простые Nullable-типы) представляют собой расширения простых типов. Их объявления принадлежат пространству имен `System.Nullable`.

Это шаблонные типы, то есть типы, построенные в результате детализации шаблонов. Шаблон `Nullable<>` используется для расширения простых типов, которые по своей сути являются структурами. Для обозначения `Nullable` шаблонных (построенных на основе шаблона) типов используются две нотации:

```
 Nullable<Int32> val;        // Полная нотация.
  Nullable<int> val;        // Полная нотация. Еще один вариант.

 Int32? val;        // Сокращенная нотация.
  int?   val;        // Сокращенная нотация. Еще один вариант.
```

Шаблон расширяет диапазон возможностей шаблонных типов.

Такие типы имеют "обычный" для своих типов-аналогов диапазон значений, который дополняется null-значением. Кроме того, шаблонный тип получает дополнительные функциональные возможности в виде методов и свойств, что объясняется необходимостью работы с null значением.

Например, переменной типа `Nullable<Int32>` (следует говорить "Nullable of Int32") может быть присвоено значение в диапазоне от `-2147483648` до `2147483647`. Этой переменной также может быть присвоено значение `null`.

У Nullable-типов к тому же шире диапазон функциональных возможностей.

Переменной типа `Nullable<bool>` могут быть присвоены значения `true, false, null`.

Возможность присвоения такого значения переменной арифметического или Boolean типа может оказаться полезным при работе с базами данных и разработке типов (классов или структур), содержащих поля, которым может быть НЕ присвоено НИКАКОГО значения.

Пример объявления и применения nullable-значений приводится ниже:

```
class NullableExample
 {
     static void Main()
     {
// Вопросительный знак в объявлении - признак nullab'ости переменной.
         int? num = null;
// Здесь мы обращаемся к свойству, которое проверяет наличие "непустого"
// значения у переменной num.
         if (num.HasValue == true)
         {
             System.Console.WriteLine("num = " + num.Value);
         }
         else
        {
             System.Console.WriteLine("num = Null");
         }

// Простой двойник (типа int) - переменная y устанавливается в 0.
// И это безопасный способ инициализации "простого" двойника.

int y = num.GetValueOrDefault();

// Можно, конечно, попытаться присвоить значение "напрямую",
// однако при этом сохраняется опасность того, что nullable в
// данный момент (num.Value - свойство, позволяющее получить
// значение num) проинициализировано пустым значением. Тип int этого
// не вынесет - будет возбуждено исключение. Поэтому присвоение
// значения и сопровождается такими предосторожностями.
        try
        {
             y = num.Value;
         }
         catch (System.InvalidOperationException e)
         {
             System.Console.WriteLine(e.Message);
         }
     }
}
```

Листинг 13.5.

Результат выполнения программки:

```
num = Null
Nullable object must have a value.
```

Обзор Nullable-типов

`Nullable` обладают следующими характеристиками:

- Переменной `Nullable` типа-значения может быть присвоено значение из "базового" диапазона значений или `null`.
- шаблон `System.Nullable<T>` не может быть применен для расширения ссылочных типов. (Ссылочные типы и без того поддерживают `null` значение).
- Синтаксис допускает сокращенную форму для объявления `Nullable` типа: `T?` (`T` — обозначение типа-значения) и `System.Nullable<T>` —две эквивалентные формы для обозначения типа.
- Операция присвоения выглядит аналогично операции присвоения типов-значений, например `int? x = 10;` или `double? d = 4.108`.
- Свойство `System.Nullable.GetValueOrDefault` обеспечивает возвращение присвоенного переменной значения, либо значения, которое соответствует значению, присваиваемому по умолчанию переменной типа-значения. Это свойство позволяет организовать безопасное взаимодействие между объектами – представителями типов `System.Nullable<T>` и `T`. Пример использования свойства:

  ```
  int j = x.GetValueOrDefault();
  ```

- применение `read-only` свойств `HasValue` и `Value` также позволяет взаимодействовать объектам типа `System.Nullable<T>` и `T` в рамках одного предложения. Свойство используется для проверки непустого значения перед присвоением. `HasValue` возвращает `true`, если переменная содержит значение, либо `false`, это `null`. Свойство `Value` возвращает значение, если таковое было присвоено. В противном случае возбуждается исключение `System.InvalidOperationException`:

  ```
  if(x.HasValue) j = x.Value;
  ```

- предопределенное значение для объектов `Nullable` типа может быть определено с помощью ранее описанных свойств. Применительно к значению по умолчанию свойство `HasValue` возвращает `false`. Значение свойства `Value` НЕ ОПРЕДЕЛЕНО.
- объявленная в шаблоне операторная функция `??` работает следующим образом: если `Nullable` объекту присвоено непустое значение, операторная функция возвращает это значение. В противном случае возвращается значение второго операнда операторной функции. Например:

  ```
  int? x = null; int y = x ?? -1;
  ```

- вложенные конструкции при объявлении `Nullable` типов не допускаются:

```
Nullable<Nullable<int>> n;
```

- диапазон значений `Nullable<T>` типа соответствует диапазону значений простого типа `T`, и включает еще одно дополнительное (пустое) значение `null`. Синтаксис C# допускает две эквивалентные формы объявления `Nullable`-типа:

```
System.Nullable<T> variable
T? variable
```

- здесь `T` обозначает "основной" тип шаблонного типа. Тип `T` должен быть типом-значением, возможно, что структурой. Тип `T` не может быть типом-ссылкой.
- Любой тип-значение может быть основой для шаблонного nullable типа. Например:

```
int? i = 10;
 double? d1 = 3.14;
 bool? flag = null;
 char? letter = 'a';
 int?[] arr = new int?[10];
```

Nullable Types. Члены класса

Прежде всего, пара свойств:

```
HasValue
```

Свойство `HasValue` является свойством типа `bool`. Оно принимает значение `true`, если переменная имеет непустое значение (значение, отличное от `null`):

```
Value
```

Тип свойства `Value` соответствует "основному" типу. Если значение свойства `HasValue` является `true`, свойство `Value` принимает соответствующее значение. Если свойство `HasValue` принимает значение `false`, попытка определения значения свойства `Value` завершается генерацией исключения `InvalidOperationException`.

В следующем примере свойство `HasValue` позволяет без генерации исключения (если это возможно) определить (и распечатать) значение `nullable` переменной типа `int?`:

```
int? x = 10;
 if (x.HasValue)
 {
     System.Console.WriteLine(x.Value);
 }
 else
 {
     System.Console.WriteLine("Undefined");
 }
```

Проверку значения можно также произвести следующим образом:

```
int? y = 10;
 if (y != null)
 {
     System.Console.WriteLine(y.Value);
 }
 else
 {
     System.Console.WriteLine("Undefined");
 }
```

Явное преобразование

Особенности явного приведения типов. Значение `Nullable` может быть явным образом приведено к `regular type`. При этом можно использовать как выражение явного приведения к типу, так и свойство `Value`. Например:

```
int? n = null;

//int m1 = n;     // Будет зафиксирована ошибка на этапе компиляции.
int m2 = (int)n;  // Compiles, but will create an exception if x is null.
int m3 = n.Value; // Compiles, but will create an exception if x is null.
```

Возможно также объявление user-defined операций приведения между двумя типами. Это же преобразование будет применяться и для Nullable-вариантов этих типов.

Неявное преобразование

Переменной `nullable`-типа может быть присвоено пустое (`null`) значение:

```
int? n1 = null;
```

Значение "основного" типа допускает неявное преобразование к соответствующему `nullable`-типу.

```
int? n2;
 n2 = 10;  // Implicit conversion.
```

Операции

Предопределенные одноместные и двуместные операции, а также user-defined операции (операторные функции), которые существуют для value types-могут применяться для Nullable типов. Эти операторные функции produces null value, если значение операндов null. В противном случае значения операндов используются для определения значения. Например:

```
int? a = 10;
 int? b = null;

 a++;          // Increment by 1, now a is 11.
 a = a * 10;   // Multiply by 10, now a is 110.
 a = a + b;    // Add b, now a is null.
```

Основное правило, по которому реализуются операции сравнения значений Nullable типов, состоит в следующем. Результат сравнения двух Nullable-операндов всегда будет false, если хотя бы один из операндов имеет значение null. Например:

```
int? num1 = 10;
 int? num2 = null;
 if (num1 >= num2)
 {
     System.Console.WriteLine("num1 is greater than or equal to num1");
 }
 else
 {
     // num1 НЕ МЕНЬШЕ num2
 }
```

Заключение в части else условного оператора НЕКОРРЕКТНО, поскольку num2 является равным null, то есть пустым и поэтому не содержит никакого значения.

Операция ??

Операция ?? определяет то значение, которое должно присваиваться переменной "основного" типа в случае, когда значением переменной nullable типа является null:

```
int? c = null;
 // d = c, если только c не равно null. В этом случае d = -1.
 int d = c ?? -1;
```

Эта операция позволяет строить сложные выражения. Например:

```
int? e = null;
 int? f = null;
 // g = e или f, если только e и f одновременно не равны  null.
 // В этом случае g = -1.
 int g =  e ?? f ?? -1;
```

Тип bool?

Область значений bool? Nullable-типа состоит из трех значений: true, false, null.

Поэтому значения данного типа неприменимы в операторах управления выполнением кода — условных операторах цикла — if, for, while. Попытка применения этого типа в вышеперечисленных операторах пресекается уже на стадии компиляции с кодом Compiler Error CS0266:

```
bool? b = null;
 if (b) // Error CS0266.
 {
 }
```

Ну не сделали под этот тип новых операторов, а для старых непонятно, каким же образом интерпретировать значение null в контексте условия.

Nullable Booleans могут быть явно преобразованы к типу bool , однако, если операнд все же окажется пустым (null), возбуждение InvalidOperationException неизбежно. Поэтому непосредственно перед применением преобразования следует прибегать к помощи HasValue-свойства.

Для Nullable-типа, основанного на типе bool, определены операторные функции

```
 bool? operator &(bool? x, bool? y)
 bool? operator |(bool? x, bool? y)
```

Таблица истинности трехзначной логики прилагается:

x	y	x&y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

Введение в программирование на C# 2.0

14. Лекция: Совмещение управляемого и неуправляемого кодов: версия для печати и PDA

.NET появилась не на пустом месте. Вновь разрабатываемый управляемый код вынужден взаимодействовать с существующим неуправляемым программным кодом. Поэтому на платформе .NET предусмотрены различные сценарии установления взаимодействия между управляемым и неуправляемым кодами. О них рассказанно в этой лекции

Программный код, выполняющийся под управлением CLR, называется управляемым кодом.

Программный код, выполняющийся вне среды выполнения CLR, называется неуправляемым кодом.

Примеры неуправляемого программного кода:

- функции Win32 API;
- компоненты COM;
- интерфейсы ActiveX.

.NET появилась не на пустом месте. Вновь разрабатываемый управляемый код вынужден взаимодействовать с существующим неуправляемым программным кодом. Поэтому на платформе .NET предусмотрены различные сценарии установления взаимодействия между управляемым и неуправляемым кодами. Microsoft .NET Framework обеспечивает взаимодействие с компонентами COM, службами COM+, внешними библиотеками типов и многими службами операционной системы.

CLR скрывает имеющиеся в управляемой и неуправляемой моделях различия. Проблемы, связанные с типами данных, механизмами обработки ошибок и т. д. в управляемой и неуправляемой моделях, решаются CLR "незаметно" как для вызывающей стороны (клиента), так и для вызываемой стороны (сервера). Таким образом, организация взаимодействия между управляемым и неуправляемым кодом выглядит проще, чем могло быть...

C++ .NET. Совмещение управляемого и неуправляемого кодов

Реализованные в .NET языки программирования позволяют создавать управляемый код, который может взаимодействовать с неуправляемыми библиотеками Win32 и компонентами на основе модели компонентных объектов Microsoft (COM).

Язык программирования C++ .NET является единственным, который позволяет создавать как управляемый, так и неуправляемый код. Это дает возможность не только использовать неуправляемый библиотечный код, но и смешивать управляемый и неуправляемый коды в одном приложении.

В приводимых ниже примерах кроме языка программирования C# будет использоваться язык C++.

Управляемый код. Осознать разницу

Первый шаг тривиален. Создается C++ Win32 Console Project, то есть Неуправляемое Консольное Приложение на C++. Последняя версия Visual Studio .NET 2005 позволяет в широких пределах смешивать управляемый и неуправляемый коды.

```
  // Транслятору задали опцию /clr.
  // Эта опция преобразует все объявляемые в программе типы.
  // От имени объекта элементарного типа (например, int) можно
  // вызвать методы базового класса object.
  // Но только вызываются они лишь из фрагментов управляемого
кода.
  // Иначе – сообщение транслятора:
  // ... managed type or function cannot be used in an unmanaged function

  #include "stdafx.h"

  #include <iostream>
  using namespace std;

  #using <mscorlib.dll> // Ядро CLR
using namespace System;

class uClass; // Предварительное неполное объявление класса.
 class mClass; // Предварительное неполное объявление класса.

  #pragma managed
class mClass
 {
 public:
 uClass *ucP;
mClass *mcP;
int x;

mClass()
   {
  // Только в неуправляемом коде!
  // cout << "Ha-Ha-Ha";

Console::WriteLine("mClass");
  // Легко посмотрели значение непроинициализированной переменной.
  Console::WriteLine(x.ToString());
  }
  ~mClass()
    {
           Console::WriteLine("~mClass");
    }
```

```
void mcF0()
 {
  Console::WriteLine("mcF0()");
 }

mClass* mcGenerator();

 };
 #pragma unmanaged

class uClass
 {
 public:
 uClass *ucP;
mClass *mcP;
int x;

uClass()
   {
    // Только в управляемом коде!
    //Console::WriteLine("Ha-Ha-Ha");
    printf("uClass\n");
   }
 ~uClass()
   {
    printf("~uClass\n");
   }

void ucF0()
 {
  cout << "ucF0()\n";
 }

uClass* ucGenerator();
 };
 // Судя по всему, функция Управляемого
 // класса может быть НеУправляемой!
 mClass* mClass::mcGenerator()
 {
  //x.ToString();
  //Console::WriteLine("Ha-Ha-Ha");
  cout << "Ha-Ha-Ha from unmanaged function of managed class!" << endl;
  ucP = new uClass();
  ucP->ucF0();
  delete ucP;
  return new mClass();
 }

 #pragma managed
 // А сделать Управляемой функцию НеУправляемого класса невозможно.
 // Прагма managed для функции - члена неуправляемого класса игнорируется.
uClass* uClass::ucGenerator()
 {
  cout << "Ha-Ha-Ha from function of unmanaged class!" << endl;
  //Console::WriteLine("Ha-Ha-Ha");
  //x.ToString();
  mcP = new mClass();
  mcP->mcF0();
  delete mcP;
  return new uClass();
 }
 #pragma unmanaged

int _tmain(int argc, _TCHAR* argv[])
 {
 void *xPoint;
 int x = 125;

 // Только не смешивать!
 //Console::WriteLine("Ha-Ha-Ha");

mClass *mc = new mClass();
 mc->mcF0();
 xPoint = mc->mcGenerator();
 delete (mClass*)xPoint;
 delete mc;

uClass *uc = new uClass();
 uc->ucF0();
 xPoint = uc->ucGenerator();
 delete (uClass*)xPoint;
 delete uc;

return 0;
 }
```

Листинг 14.1.

Управляемая библиотека

Создание кода управляемой библиотеки тривиально. Для этого в Visual Studio предусмотрены специальные опции. Новый проект создается как библиотека классов (Class Library). Сборка автоматически получает расширение .dll.

C# основывается на парадигме объектно-ориентированного программирования, поэтому библиотека классов представляет собой все то же объявление класса.

Методы – члены класса составляют основу функциональности библиотеки. Наличие пары явно определяемых конструкторов – вынужденная мера. Это требование со стороны модуля на C++, использующего библиотеку. Там невозможно создать объект без явным образом объявленных конструкторов умолчания и копирования:

```
using System;

namespace CSLib00
 {
 public class Class1
 {
 // Явное объявление конструктора без параметров.
 public Class1()
 {
 }

 // Явное объявление конструктора копирования.
 // Конструктор с одним параметром – ссылкой на
 // объект - представитель собственного класса.
 // Именно такие конструкторы называются конструкторами
 // копирования.
 public Class1(Class1 cKey)
 {
 }

 // Реализация функциональности. Метод - член класса Class1 Summ.
 public int Summ(int key1, int key2)
 {
 Console.WriteLine("this is Summ from CSLib00.dll");
 return (key1 + key2);
 }
 }
 }
```

Управляемая библиотека в управляемом коде

Использование управляемой библиотеки в управляемом коде тривиально. Главная проблема, которая возникает при разработке приложения, использующего код управляемой библиотеки, – добавить ссылку (Add Reference) на библиотечную сборку. Для этого нужно указать месторасположение сборки в соответствующем диалоговом окне.

После этого Visual Studio копирует сборку в директорию, в которой располагается разрабатываемый код. При согласовании пространств имен (оператор using или использование полного имени с точечной нотацией) библиотечного модуля и модуля клиента, библиотечные классы и методы готовы к использованию в коде клиента:

```
using System;

namespace CSClient
 {
 class Program
 {
 static void Main(string[] args)
 {
 // Здесь используется точечная нотация.
 // Явное указание принадлежности имени класса
 // пространству имен CSLib00.
CSLib00.Class1 c1 = new CSLib00.Class1();
 // Создали объект, затем вызываем метод!
 Console.WriteLine("the res = {0}", c1.Summ(1, 2));
 }
 }
 }
```

Управляемая библиотека в неуправляемом коде

Особенности разработки неуправляемого программного кода, использующего управляемую библиотеку, состоят в следующем:

- неуправляемый код транслируется с опцией /clr;
- добавляется ссылка на библиотечную сборку;
- определяется управляемая функция, в теле которой с помощью оператора gcnew создается объект объявленного управляемой библиотеке класса, от имени которого вызывается библиотечная функция.

```
 #include "stdafx.h"
 #using <mscorlib.dll>
 using namespace System;
 using namespace CSLib00;

 #pragma managed
void managedLibStarter()
 {
  Class1 cl1 = gcnew Class1();
  Console::WriteLine("{0}",cl1.Summ(1,2));
```

```
}
#pragma unmanaged

int _tmain(int argc, _TCHAR* argv[])
{
 printf("QWERTY\n");
 managedLibStarter();
 return 0;
}
```

Вызов неуправляемых функций из управляемого модуля

Сначала определение.

Платформный вызов — это служба, которая позволяет управляемому программному коду вызывать неуправляемые функции, реализованные в библиотеках динамической компоновки (DLL), например, функции библиотек Win32 API.

Служба платформного вызова обеспечивает поиск и вызов экспортируемой функции и в случае необходимости обеспечивает маршалинг ее параметров (передачу параметров через границы процесса).

Платформный вызов позволяет управлять значительной частью операционной системы посредством вызова функций Win32 API и других библиотек DLL. В дополнение к Win32 API, через платформный вызов доступны многие другие интерфейсы API и DLL.

Вызов неуправляемых функций средствами платформного вызова предполагает следующие действия:

- идентификация вызываемой функции в библиотеке DLL;
- создание класса для сохранения вызываемой функции;
- объявление прототипа (!) вызываемой функции в управляемом коде;
- вызов функции.

При вызове неуправляемой функции платформный вызов выполняет следующую последовательность действий:

- определяется местонахождение DLL-библиотеки, содержащей функцию;
- DLL загружается в память;
- определяется месторасположение вызываемой функции (адрес функции в памяти);
- значения параметров функции заносятся в стек;
- в случае необходимости осуществляется маршалинг данных;
- осуществляется передача управления неуправляемой функции.

В лучших традициях .NET определение местонахождения и загрузка DLL, а также определение местоположения адреса функции в памяти, выполняется только при первом вызове функции.

Идентификация вызываемой функции

Идентификация функции DLL предполагает:

- указание имени вызываемой функции;
- указание имени содержащей ее библиотеки DLL (имени файла).

Например, идентификация функции `MessageBox` включает имя (`MessageBox`) и имя файла (User32.dll, User32, или user32). Программный интерфейс приложений Microsoft Windows (Win32 API) может содержать несколько версий функции. Например, версии функций, которые работают с символами и строками: для 1-байтовых символов – ANSI и для 2-байтовых символов – Unicode.

Так `MessageBoxA` является точкой входа для функции `MessageBox` версии ANSI, а `MessageBoxW` — точкой входа для аналогичной функции версии Unicode.

Имена функций, располагаемых в определенной библиотеке DLL, например, user32.dll, можно получить с помощью ряда инструментов в режиме командной строки. Например, для получения имен функций можно использовать вызовы

```
dumpbin –exports user32.dll
```

или

```
link –dump –exports user32.dll.
```

Создание класса для размещения библиотечной функции

Эта деятельность называется упаковкой неуправляемой функции в обертку управляемого класса.

На самом деле неуправляемые функции можно и не обертывать. Но это удобно, поскольку процесс определения функций DLL может быть трудоемким и представлять собой источник ошибок.

Для обертывания можно использовать существующий класс, можно создать отдельный класс для каждой неуправляемой функции либо создать один класс для всего набора используемых в приложении неуправляемых функций. В классе-обертке для каждой вызываемой библиотечной функции должен быть определен собственный статический метод. Это определение может включать дополнительную информацию, например характеристику символьного набора или информацию о соглашении о вызовах, которое используется при передаче параметров функции. Если эта информация отсутствует, используются соглашения, принятые по умолчанию.

После упаковки функции в обертку ее методы можно вызывать точно так же, как и методы любой другой статической функции. Платформный вызов обеспечивает передачу управления библиотечной функции автоматически.

Прототипы в управляемом коде

Для обращения к неуправляемой функции DLL из управляемого программного кода требуется знать имя функции и имя библиотеки DLL, которая ее экспортирует.

Располагая этой информацией, можно начинать создавать управляемое определение для неуправляемой функции, реализованной в DLL.

При этом с помощью атрибутов можно управлять порядком создания, платформным вызовом функции и маршалингом данных в функцию и из нее (передачу значений параметров и получение возвращаемого значения):

```
using System.Runtime.InteropServices;
 [DllImport("user32.dll")]
 public static extern int MessageBox(int hWnd,
          String text,
          String caption,
          uint type);
```

Поведение управляемого программного кода при его выполнении задается значениями полей атрибутов. Служба платформного вызова работает в соответствии с набором стандартных значений различных полей атрибутов, которые хранятся как метаданные в сборке.

В следующей таблице представлен полный набор полей атрибутов, имеющих отношение к платформному вызову. Для каждого поля в таблице имеется стандартное значение и ссылка на сведения относительно использования этих полей для определения неуправляемых функций DLL.

Управление службой платформного вызова сводится к применению полей атрибутов. Знание правил применения полей атрибутов – основа управления службой платформного вызова!

Поле	Описание
`BestFitMapping`	Отключает наилучшее соответствие
`CallingConvention`	Задает соглашение о вызовах, которое должно использоваться при передаче аргументов методов. По умолчанию используется WinAPI, что соответствует __ stdcall для 32-разрядных платформ на основе процессора Intel
`CharSet`	Управляет передачей имен и задает способ маршалинга строковых аргументов в функцию. Стандартное значение — `CharSet.Ansi`
`EntryPoint`	Задает точку входа DLL для вызова
`ExactSpelling`	Указывает, должна ли быть изменена точка входа в соответствии с символьным набором. Стандартное значение варьируется в зависимости от языка программирования
`PreserveSig`	Указывает, должна ли управляемая подпись метода быть преобразована в неуправляемую подпись, которая возвращает значение `HRESULT` и для возвращаемого значения имеет дополнительный аргумент `[out, retval]`. По умолчанию используется значение `true` (подпись не должна преобразовываться)
`SetLastError`	Позволяет вызывающему объекту для определения факта ошибки при выполнении метода использовать API-функцию `Marshal.GetLastWin32 Error`. В Visual Basic по умолчанию используется значение true; в C# и C++ — значение `false`
`ThrowOnUnmappableChar`	Управляет возникновением исключения при появлении несопоставимого символа Unicode, который преобразуется в символ ANSI "`?`"

Указание точки входа

Точка входа определяет расположение функции в DLL. В управляемом проекте исходное имя или порядковый номер точки входа функции назначения определяет эту функцию в границах взаимодействия. Можно также сопоставить точку входа с другим именем, фактически переименовывая функцию.

Для переименования функции DLL возможны следующие причины:

- чтобы избежать использования имен API-функций, чувствительных к регистру знаков;
- чтобы привести имена в соответствие с существующими стандартами именования;
- чтобы сделать возможным вызов функций, принимающих данные разных типов (объявляя несколько версий одной и той же функции DLL);
- чтобы упростить применение API-интерфейсов, которые содержат функции версий для ANSI и Unicode.

Переименование функции в C#

Для задания функции DLL по имени или порядковому номеру можно использовать поле `DllImportAttribute.EntryPoint`. Если имя функции в определении метода совпадает с именем точки входа в DLL, явно задавать функцию с помощью поля `EntryPoint` не требуется. В противном случае, чтобы указать имя или порядковый номер, следует использовать одну из следующих форм атрибута:

```
  [DllImport("dllname", EntryPoint="Functionname")]
  [DllImport("dllname", EntryPoint="#123")]
```

При этом порядковому номеру должен предшествовать знак '#'.

Ниже приводится пример переименования функции. Имя функции `MessageBoxA` заменяется на `MsgBox` с помощью поля `EntryPoint`. Неуправляемая функция `MessageBoxA`, которая является точкой входа, в управляемом коде представляется под именем `MsgBox`:

```
using System.Runtime.InteropServices;

public class Win32
 {
     [DllImport("user32.dll", EntryPoint="MessageBoxA")]
     public static extern int MsgBox(int hWnd,
                String text,
                String caption,
                uint type);
 }
```

Указание набора знаков

Поле `DllImportAttribute.CharSet` играет двойную роль:

- управляет маршалингом строк;
- определяет порядок нахождения платформным вызовом имен функций в DLL.

Некоторые API экспортируют две версии функций, которые принимают строковые аргументы: узкую (ANSI) и широкую (Unicode). Например, Win32 API для функции `MessageBox` содержит следующие имена точек входа:

- `MessageBoxA` — обеспечивает форматирование с использованием 1-байтовых символов ANSI, на что указывает буква "`A`", добавляемая в конец имени точки входа. При вызовах MessageBoxA маршалинг строк всегда выполняется в формате ANSI, который обычно применяется на платформах Windows 98 и Windows 95;
- `MessageBoxW` — обеспечивает форматирование с использованием 2-байтовых символов Unicode, на что указывает буква "`W`", добавляемая в конец имени точки входа. При вызовах `MessageBoxW` маршалинг строк всегда выполняется в формате Unicode, который обычно применяется на платформах Windows NT, Windows 2000 и Windows XP.

Маршалинг строк и совпадение имен

Поле `CharSet` может принимать следующие значения:

- `CharSet.Ansi` (стандартное значение).

  При этом:

  1. Платформный вызов выполняет маршалинг строк из их управляемого формата Unicode в формат ANSI.
  2. Если значение поля `DllImportAttribute.ExactSpelling` оказывается установленным в `true`, как это принято по умолчанию в Visual Basic .NET, платформный вызов ищет только указанное имя. Например, если указано `MessageBox`, платформный вызов ищет именно `MessageBox`. Если он не может найти точное посимвольное совпадение, возникает исключение при выполнении вызова функции.

  Когда значение поля `ExactSpelling` установлено в `false`, как это принято по умолчанию в C# и управляемых расширениях для C++, платформный вызов сначала ищет точный псевдоним (`MessageBox`), а затем добавленное имя (`MessageBoxA`), если точный псевдоним не найден.

  Следует учитывать, что механизм совпадения имен в формате ANSI отличается от механизма совпадения имен в формате Unicode.

- `CharSet.Unicode`.

  При этом платформный вызов обеспечивает маршалинг строк путем копирования строки из их управляемого формата в формат Unicode.

  Если же при этом значение поля `ExactSpelling` оказывается равным `true`, действует принцип совпадения имен: платформный вызов ищет только указанное имя. Например, если указано `MessageBox`, платформный вызов ищет именно `MessageBox`. Если он не может найти точное посимвольное совпадение, возникает сбой.

  А если значение поля `ExactSpelling` устанавливается в `false`, как это принято по умолчанию в C#, платформный вызов сначала ищет добавленное имя (`MessageBoxW`), а затем – точный псевдоним (`MessageBox`), если добавленное имя не найдено.

- `CharSet.Auto`.

  При этом платформный вызов выбирает между форматами ANSI и Unicode во время выполнения в соответствии с платформой назначения.

Пример. Указание набора символов в C#

Поле `DllImportAttribute.CharSet` определяет набор символов как базовый набор ANSI или Unicode. Набор символов определяет режим выполнения маршалинга строковых аргументов. Для указания набора символов применяется один из следующих вариантов атрибута:

```
[DllImport("dllname", CharSet=CharSet.Ansi)]
[DllImport("dllname", CharSet=CharSet.Unicode)]
[DllImport("dllname", CharSet=CharSet.Auto)]
```

В следующем примере показаны три управляемых определения функции `MessageBox` с атрибутами, задающими наборы символов. В первом определении, в котором значение поля `CharSet` не задано, по умолчанию принимается набор символов ANSI:

```
[DllImport("user32.dll")]
    public static extern int MessageBoxA(int hWnd, String text,
      String caption, uint type);

[DllImport("user32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBoxW(int hWnd, String text,
      String caption, uint type);

[DllImport("user32.dll", CharSet=CharSet.Auto)]
    public static extern int MessageBox(int hWnd, String text,
      String caption, uint type);
```

Примеры платформного вызова. MessageBox, Beep, PlaySound

Под ПЛАТФОРМОЙ НАЗНАЧЕНИЯ понимается платформа, которой предназначена закодированная в атрибутах информация. Естественно, это .NET.

В других платформах службы платформного вызова нет. Никакая другая платформа просто не сумеет прочитать и понять значение полей атрибутов.

Ниже демонстрируется определение и вызов функции `MessageBox` из библиотеки `User32.dll`. В качестве аргумента передается простая строка. Значение для атрибута поля `DllImportAttribute.CharSet` установлено в `Auto`. Это позволяет платформе назначения самостоятельно определять размер символов и выбирать маршалинг строк:

```
using System.Runtime.InteropServices;

public class Win32 {
      [DllImport("user32.dll", CharSet=CharSet.Auto)]
      public static extern int MessageBox(int hWnd, String text,
                  String caption, uint type);
 }

public class HelloWorld {
     public static void Main() {
      Win32.MessageBox(0, "Hello World", "Platform Invoke Sample", 0);
     }
 }
```

Информация о функции `Beep` в хелпах по C# отсутствует. Такой функции в .NET НЕТ. Однако способы заставить C#-приложение "пропищать" все же существуют (передача функции `Console.Writeline` параметра с соответствующей escape-последовательностью не в счет).

Итак, функция

```
Beep
```

обеспечивает генерацию simple tones (простых звуков) на спикере. Функция выполняется синхронно; она не возвращает управления до тех пор, пока не завершится звучание.

```
BOOL Beep(
   DWORD dwFreq,
   DWORD dwDuration
 );
```

Параметры

```
 dwFreq
```

Частотная характеристика звука в герцах. Диапазон значений этого параметра ограничен следующими величинами `37–32,767` (`0x25– 0x7FFF`).

В Windows Me/98/95 функция Beep этот параметр игнорирует.

```
dwDuration
```

Продолжительность звучания в миллисекундах.

В Windows Me/98/95 функция Beep этот параметр игнорирует.

Возвращаемое значение

Здесь речь идет о неуправляемой функции, судя по всему, реализованной в C++. Так вот, при успешном выполнении функция возвращает ненулевое значение. В C++ такие значения соответствуют значению ИСТИНА.

В противном случае возвращается нулевое значение, то есть ЛОЖЬ:

```
using System;
 using System.Threading;
 using System.Runtime.InteropServices;

namespace BeepByWin32API
 {

 // Import a Beep() API function.

class Class1
 {
 // Таким образом получаем возможность использования импортируемых
 // функций в приложении .NET.

 [DllImport("kernel32.dll")]
 private static extern bool Beep(int frec, int dur);
 // Импортируется функция PlaySound().
 [DllImport("winmm.dll")]
 public static extern bool PlaySound(string pszSound,
                                     int hmod,
                                    int fdwSound);
 // Константы, необходимые для использования PlaySound...
 public const int SND_FILENAME = 0x00020000;
 // За конкретное значение параметра я не ручаюсь. Подобрал сам.
 // Играет - и ладно...
 public const int SND_ASYNC = 0x0010;//0x0001;


 // The main entry Point for the application.

 static void Main(string[] args)
 {
 int i;
 // Извлекаем звук посредством escape-последовательности.
 Console.WriteLine("Пропищим..." + "\a\a\a\a\a\a\a\a\a\a");
```

```
 Console.WriteLine("И еще...");
 for (i = 0; i < 10; i++) {Console.Write("\a"); Thread.Sleep(1000);}

Console.WriteLine("Нажми Any Key для продолжения опытов...");
 Console.ReadLine();
 // Извлекаем звук посредством обращения к функции API Beep().
 // Frequency of the sound, in hertz.
 // This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).
 // Duration of the sound, in milliseconds.
 Beep(800,200);
 Beep(500,1000);
 Beep(100,500);
 Beep(1000,2000);
 Beep(0x25,2000);
 Beep(0x7FFF,2000);

Console.WriteLine("Нажми Any Key для продолжения опытов...");
 Console.ReadLine();
for (i = 0; i < 10; i++)
 {
 PlaySound("Trumpet1.wav", 0, SND_FILENAME|SND_ASYNC);
 }

 }
 }
 }
```

Листинг 14.2.

Введение в программирование на C# 2.0

15. Лекция: Потоки: версия для печати и PDA
В этой лекции рассказывается о принципах управления потоками в языке C#

Процесс – объект, который создается Операционной Системой для приложения (не для приложения .NET) в момент его запуска. Характеризуется собственным адресным пространством, которое напрямую недоступно другим процессам.

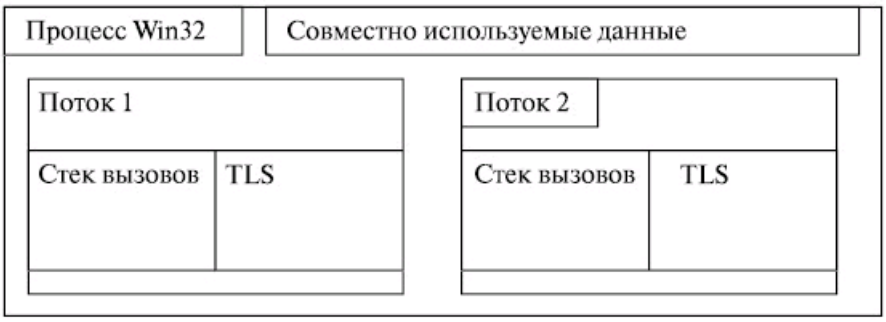
Поток. В рамках процесса создаются потоки (один – первичный – создается всегда). Это последовательность выполняемых команд процессора. В приложении может быть несколько потоков (первичный поток и дополнительные потоки).

Потоки в процессе разделяют совместно используемые данные и имеют собственные стеки вызовов и локальную память потока (Thread Local Storage – TLS). TLS потока содержит информацию о ресурсах, используемых потоком, о регистрах, состоянии памяти, выполняемой инструкции процессора.

Общее управление выполнением потоков осуществляется на уровне ОС. Нет ничего, кроме выполняющихся потоков, которыми руководит управляющий поток ОС. В этом управляющем потоке принимается решение относительно того, на протяжении какого времени будет выполняться данный поток. Управляющий поток ОС прерывает выполнение текущего потока, состояние прерванного потока фиксируется в TLS, после чего управление передается другому потоку.

В приложении все потоки выполняются в рамках общего адресного пространства. И хотя организация взаимодействия между потоками, выполняемыми в общем процессе, проще, при этом все равно требуется специальный инструментарий (критические секции, мьютексы и семафоры) и остается 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.



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 Интернета, базам данных, передача данных по сети;
- одновременное выполнение нескольких задач, имеющих различный приоритет.

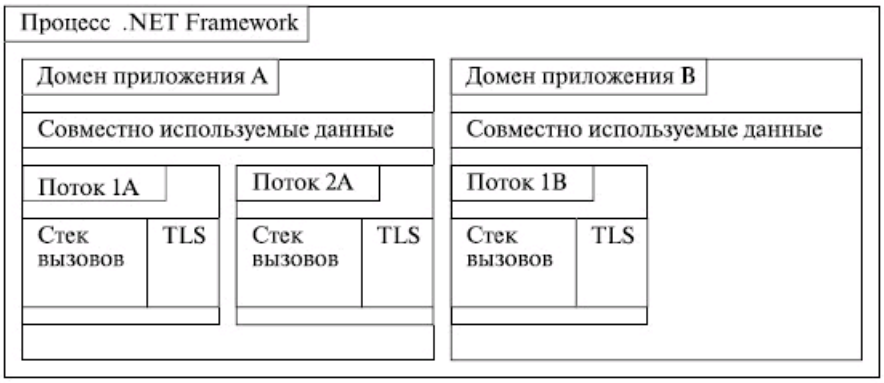
Виды многопоточности:

- Переключательная многопоточность. Основа – резидентные программы. Программа размещалась в памяти компьютера вплоть до перезагрузки системы, и управление ей передавалось каким-либо заранее согласованным способом (предопределенной 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 определенный 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.

Домен приложения

Выполнение приложений .NET начинается с запуска .NET Framework. Это процесс со своими потоками, специальными атрибутами и правилами взаимодействия с другими процессами. Приложения .NET выполняются в ОДНОМ процессе. Для этих приложений процесс .NET Framework играет роль, аналогичную роли операционной системы при обеспечении выполнения процессов.

Выполняемые в процессе .NET Framework .NET-приложения также изолируются друг от друга. Средством изоляции .NET-приложений являются домены приложений.



Домен приложения изолирует (на логическом уровне) выполняемые приложения и используемые в его рамках ресурсы. В процессе .NET Framework может выполняться множества доменов приложений. При этом в рамках одного домена может выполняться множество потоков.

Функциональность и основные свойства домена приложения реализуются в классе `System.AppDomain`.

Методы класса System.AppDomain	Описание
`CreateDomain()`	Статический. Создает новый домен приложения
`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()`	Создает объект заданного типа, определенного в указанной сборке
`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

Таким образом, множество выполняемых .NET-приложений ограничены объектами – представителями класса `System.AppDomain`.

Домен обеспечивает выполнение приложения. Приложение может "провести инспекцию пограничной службы", то есть получить доступ к объекту-домену.

Пример. Приложение-инспектор. Получает ссылку на текущий домен приложения (свой собственный домен) и исследует некоторые свойства домена:

```
using System;
 namespace Domains_00
 {
 class Class1
 {
 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`;
- как результат выполнения метода `System.Threading.Thread.GetDomain()`:

В рамках процесса (процесса .NET Framework) может быть создано множество доменов (объектов-доменов), причем новые домены можно создавать непосредственно в ходе выполнения .NET-приложения. Порождение новых доменов обеспечивается четверкой статических перегруженных методов – членов класса AppDomain с именем `CreateDomain`.

- Создается новый домен приложения с именем, заданным строкой-параметром:

```
public static AppDomain CreateDomain(string);
```

- Создается новый домен приложения с именем, заданным строкой-параметром, в рамках существующей политики безопасности:

```
public static AppDomain CreateDomain(string, Evidence);
```

- Создается новый домен приложения с использованием the specified name, evidence, and application domain setup information:

```
public static AppDomain CreateDomain(string, Evidence, AppDomainSetup);
```

- Создается новый домен приложения с именем, заданным строкой-параметром, в рамках существующей политики безопасности, указанием базового каталога, в котором проводит поиск распознаватель сборок, указанием базового каталога для поиска закрытых сборок, флагом управления загрузкой сборки в домен приложения:

```
public static AppDomain CreateDomain(string, Evidence, string, string, bool);
```

А кто в домене живет?

Прилагаемый пример демонстрирует методы анализа процесса, домена и потоков (в .NET потоки и сборки также представлены объектами соответствующих классов).

Здесь показаны потоки в процессе, а также сборки, которые выполняются в домене приложения:

```
using System;
 using System.Windows.Forms;

 // Это пространство имен требуется для работы с классом Assembly.
 using System.Reflection;
 // Это пространство имен 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 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!
 }
 }
 }
```

Листинг 15.1.

Обзор пространства имен 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

Класс Thread. Общая характеристика

Thread-класс представляет управляемые потоки: создает потоки и управляет ими — устанавливает приоритет и статус потоков. Это объектная оболочка вокруг определенного 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 определенного 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, который может включать дополнительную специфичную для данного приложения информацию.

Именование потока

Потоки рождаются безымянными. Это означает, что у объекта, представляющего поток, свойство Name имеет значение null. Ничего страшного. Главный поток все 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);

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

Листинг 15.2.

Игры с потоками

Но сначала о том, что должно происходить в потоке. Выполнение текущего потока предполагает выполнение программного кода.

Что это за код? Это код приложения. Это код статических и нестатических методов – членов классов, которые объявляются или просто используются в приложении.

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.

Поскольку первичный поток создается 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 – это забота программиста. Здесь все 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 передается 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 здесь 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.

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 расположить этот код в правильном месте. И чтобы он выполнился в правильное время:

```
using System;
using System.Threading;

namespace ThreadApp_1
{
// Рабочий класс. Делает себе свое 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("{0,10}\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);
// При создании потока не обязательно использовать делегат.
// Возможен и такой вариант. Главное — это сигнатура функции.
// th1 = new Thread(w1.DoItEasy);
th0.Start();
th1.Start();
}
}
}
```

Листинг 15.3.

Важно! Первичный поток ничем не лучше любых других потоков приложения. Он может скоропостижно завершиться раньше всех им же порожденных 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 определенные интервалы времени. Все зависит от выбора перегруженного варианта метода. Планировщик потоков смотрит на поток и принимает решение относительно того, можно ли продолжить выполнение усыпленного 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 учетом 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 миллисекунд:
Thread.Sleep(new TimeSpan(1,2,3,4,5));
```

Значение параметра, представленное выражением

```
System.Threading.Timeout.Infinite
```

позволяет усыпить поток на неопределенное 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...");
 }
 }
```

Листинг 15.4.

И всегда надо помнить: приложение выполняется до тех пор, пока не будет выполнен последний оператор последнего потока. И неважно, 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 не отстраненные от выполнения потоки оказались завершены и некому запустить отстраненный поток – приложение в буквальном смысле "зависает". Операторы в потоке могут выполняться, а выполнить их невозможно по причине отстранения потока от выполнения. Вот приложение и зависает...

```
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(25);
 }

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.");
 }
 }
```

Ко всему прочему следует иметь в виду, что метод `Suspend()` уже устарел...

"System.Threading.Thread.Suspend() is obsolete: Thread.Suspend has been deprecated. Please use other classes in System.Threading such as Monitor, Mutex, Event, and Semaphore to synchronize thread or project resources..."

Не следует использовать методы `Suspend` и `Resume` для синхронизации активности потоков. Просто в принципе ничего нельзя будет сделать, когда код в выполняемом потоке будет приостановлен с помощью метода `Suspend()`. Одним из нежелательных последствий подобного устранения от выполнения может оказаться взаимная блокировка потоков. Deadlocks can occur very easily.

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 передается оператору `return`:

```
for (x=0;;x++)
 {
 if (x==max)
 return; // Все. 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 путем вызова метода `ResetAbort()`.
  2. При перехвате исключения CLR обеспечивает выполнение операторов блоков `finally`, которые выполняются все в том же потоке.

  Таким образом, остановка потока путем 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.");
 }
 }
```

Листинг 15.5.

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 свое 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 осознать 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()` не будет переведен в состояние `Running`.

В дополнение к вышеперечисленным состояниям существует также `Background` – состояние, которое указывает, выполняется ли поток на фоне или на переднем плане.

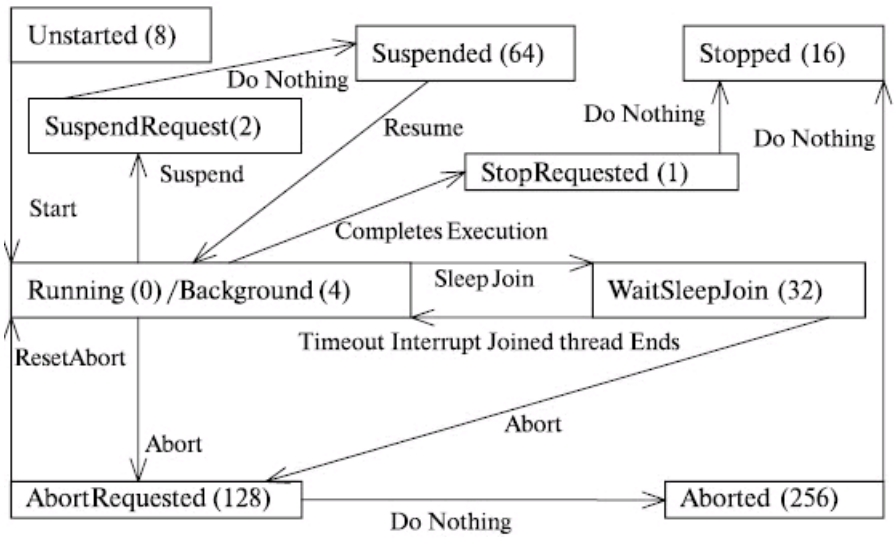
Свойство `Thread.ThreadState` потока содержит текущее состояние потока. Для определения текущего состояния потока в приложении можно использовать битовые маски. Пример условного выражения:

```
if((myThread.ThreadState & (ThreadState.Stopped | ThreadState.Unstarted))==0) {...}
```

Члены перечисления:

Имя члена	Описание	Значение
`Running`	Поток был запущен, он не заблокирован, и нет задерживающегося объекта `ThreadAbortException`	`0`
`StopRequested`	Поток запрашивается на остановку. Это только для внутреннего использования	`1`
`SuspendRequested`	Запрашивается приостановка работы потока	`2`
`Background`	Поток выполняется как фоновый, что является противоположным к приоритетному потоку. Это состояние контролируется заданием свойства `Thread.IsBackground`	`4`
`Unstarted`	Метод `Thread.Start` не был вызван для потока	`8`
`Stopped`	Поток остановлен	`16`
`WaitSleepJoin`	Поток заблокирован в результате вызова к методам `Wait, Sleep` или `Join`	`32`
`Suspended`	Работа потока была приостановлена	`64`
`AbortRequested`	Метод `Thread.Abort` был вызван для потока, но поток еще не получил задерживающийся объект `System.Threading.ThreadAbortException`, который будет пытаться завершить поток	`128`
`Aborted`	Поток находится в `Stopped`-состоянии	`256`

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 БОЛЕЕ ЧЕМ В 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 в таблице допустимых состояний:

	AR	Ab	Back	U	S	R	W	St	SusR	StopR
Abort Requested	—	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 WaitSleep	N	N	N	N	N	—	N	N	N	N
Join	Y	N	Y	N	Y	N	—	N	Y	N
Stopped	N	N	Y	N	N	N	N	—	Y	N
Suspend Requested	Y	N	Y	N	N	N	Y	N	—	N
Stop Requested	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` распознается 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");
 }
 }
 }
```

Листинг 15.6.

Приоритет потока

Задается значениями перечисления `ThreadPriority`. Эти значения используются при планировке очередности выполнения потоков в ПРОЦЕССЕ.

Приоритет потока определяет относительный приоритет потоков.

Каждый поток имеет собственный приоритет. Изначально он задается 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 очередность 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! Все равно возвращаемое значение стартовой функции при существующем механизме запуска потока (функция `Start`) некому перехватывать, а стандартный жесткий механизм предопределенных 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 создается 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
 // получает доступ ко всей необходимой ранее подготовленной информации.
 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 передается 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 ждет.");
 t.Join();
 Console.WriteLine
     ("Сообщение из первичного потока: Вторичный поток отработал.");
 Console.WriteLine
     ("Сообщение из первичного потока: Главный поток остановлен. ");
```

```
}
}
```

Листинг 15.7.

Контроль вторичных потоков. Callback-методы

Первичный поток создал и запустил вторичный поток для решения определенной задачи. Вторичный поток выполнил поставленную задачу и... завершил свою работу. Возможно, что от результатов работы вторичного потока зависит дальнейшая работа приложения. Возможно, что до завершения выполнения вторичного потока первичному потоку вообще нечего делать и он приостановлен в результате выполнения метода `Join`.

Проблема заключается в том, КОГДА и КАКИМ ОБРАЗОМ о проделанной работе станет известно в первичном потоке.

Для анализа результата выполнения вторичного потока можно использовать метод класса, который обеспечивает запуск вторичного потока. Соответствующим образом настроенный делегат также может быть передан в качестве параметра конструктору вторичного потока. Вызывать метод класса, запустившего вторичный поток, можно будет по выполнении работ во ВТОРИЧНОМ потоке.

Таким образом, функция, контролирующая завершение работы вторичного потока, сама будет выполняться во ВТОРИЧНОМ потоке! Если при этом для дальнейшей работы первичного потока (который в этот момент, возможно, находится в состоянии ожидания) необходима информация о результатах проделанной вторичным потоком работы, контролирующая функция справится с этой задачей за счет того, что она имеет доступ ко всем данным и методам своего класса. И неважно, в каком потоке она при этом выполнялась:

```
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 входную информацию и настраивает
 // callback delegate.
 public WorkThread(string text, int number,
CallbackMethod callbackDelegate)
 {
 entryInformation = text;
 value = number;
 callback = callbackDelegate;
 }

 // Метод, обеспечивающий выполнение 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 СВОЕМ потоке.
 }
 }

 // Класс-делегат задает 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 ждет.");
 t.Join();
 Console.WriteLine("Вторичный поток отработал. Главный поток остановлен.");
 }

 // Callback-метод, естественно, соответствует сигнатуре
 // callback класса делегата.
 public static void ResultCallback(int lineCount)
 {
 Console.WriteLine("Вторичный поток обработал {0} строк.", lineCount);
 }
 }
```

Листинг 15.8.

Callback-метод – метод – член класса, запустившего вторичный поток. Этот метод запускается "в качестве уведомления" о том, что вторичный поток "завершил выполнение своей миссии". Особенность Callback-метода заключается в том, что он выполняется в "чужом" потоке.

Организация взаимодействия потоков

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();

 // Еще раз о методе Join(): Выполнение главного потока приостановлено.
 StartClass.th0.Join();
 StartClass.th1.Join();

 // Потому последнее слово остается 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. Ее выполняет функция - член
 // класса StartClass. Этой функции в силу своего определения
 // известно ВСЕ 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.");
 }
 }
 }
```

Листинг 15.9.

Посредством общедоступных (public) свойств

Следующий вариант организации взаимодействия между потоками основан на использовании общедоступных свойств. От предыдущего примера отличается тем, что доступ к закрытому счетчику `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();

 // Еще раз о методе Join(): Выполнение главного потока приостановлено.
 StartClass.th0.Join();
 StartClass.th1.Join();

 // Потому последнее слово остается 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. Ее выполняет функция - член
 // класса StartClass. Этой функции в силу своего определения
 // известно ВСЕ 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.");
 }
 }
 }
```

Листинг 15.10.

Посредством общедоступных очередей

`Queue` – класс, который представляет коллекцию объектов (`objects`), работающую по принципу "первым пришел, первым ушел" (first in, first out).

`Stack` – класс, который представляет коллекцию объектов (`objects`), работающую по принципу "последним пришел, первым ушел" (last in, first out).

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 и реализующих интерфейс доступа к этим 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();
 // Еще раз о методе Join():
 // Выполнение главного потока приостановлено до завершения
 // выполнения вторичных потоков.
 StartClass.th0.Join();
 StartClass.th1.Join();
 // Потому последнее слово остается 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. Ее выполняет функция - член
 // класса StartClass. Этой функции в силу своего определения
 // известно ВСЕ 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.");
 }
 }
 }
```

Листинг 15.11.

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, размещенных в очереди. Разумеется, это идеальная ситуация. Изменение соответствующих значений может привести к тому, что обработчик данных опустошит очередь и завершит работу раньше, чем 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 именно сейчас этот ресурс используется другим потоком.

Тупик – взаимная блокировка потоков:

поток `A` захватывает ресурс a и не может получить доступа к ресурсу `b`, занятому потоком `B`, который может быть освобожден потоком только по получению доступа к ресурсу `a`.

Пример взаимодействующих потоков рассматривается далее:

```
 // 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.
using System;
 using System.Threading;

using System.Collections;
 namespace CommunicatingThreadsQueue
 {

 // Модифицированный вариант очереди – очередь с флажком.
 // Захвативший эту очередь поток объявляет очередь "закрытой".

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 - 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();
 // Еще раз о методе Join():
 // Выполнение главного потока приостановлено до завершения
 // выполнения вторичного потока загрузки 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 приостановлено
 // до завершения выполнения вторичных потоков.
 StartClass.th1.Join();
 StartClass.th2.Join();

 // Последнее слово остается 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. Ее выполняет функция - член
 // класса StartClass. Этой функции в силу своего определения
 // известно ВСЕ 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.");
```

```
  }
  }
  }
```

Листинг 15.12.

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 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 изменен.

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.

Очередь как объект синхронизации

Очередь — 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 серьезную опасность — и не столько для самого объекта очереди в смысле возможного искажения сохраняемой в ней информации, сколько для самого приложения. Класс `Queue` взаимодействует с окружением через интерфейсы, а 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 своем 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, который выполняет 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, которая связана с объектом
  // 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 синхронизирована она или нет, а соответствующий код (критические секции кода) защищать все 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.
```

Стек как объект синхронизации

`Stack` – класс, который представляет коллекцию объектов, обслуживаемую по принципу "последним пришел — первым вышел".

Список всех членов этого типа представлен в разделе "Stack-члены".

```
public class Stack : ICollection, IEnumerable, ICloneable
```

Открытые статические (Shared в Visual Basic) члены этого типа могут использоваться для многопоточных операций. Потокобезопасность членов экземпляра не гарантируется.

Для обеспечения потокобезопасности операций с классом `Stack`, все они должны выполняться с помощью обертки, возвращенной методом `Synchronized`.

Перечисление в коллекции в действительности не является потокобезопасной процедурой. Даже при синхронизации коллекции другие потоки могут изменить ее, что приводит к созданию исключения при перечислении. Чтобы обеспечить потокобезопасность при перечислении, можно либо заблокировать коллекцию на все время перечисления, либо перехватывать исключения, которые возникают в результате изменений, внесенных другими потоками.

Синхронизация работы потоков

Организация критических секций

```
 // Пара потоков "наперегонки" заполняет одну очередь.
 // Эти потоки синхронизуются посредством критических 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.


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. Ее выполняет функция - член
// класса StartClass. Этой функции в силу своего определения
// известно ВСЕ 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.");
}
}
}
```

Листинг 15.13.

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 ее.

Таким образом, монитор гарантирует, что никакой другой поток не сможет обратиться к коду, выполняемому потоком, который захватил 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 остается 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`-методы 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 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 непосредственно в классе, членом которого он является.

```
  // 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.

 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();
  }
  }
  }
```

Листинг 15.14.

Mutex

Когда двум или более потокам нужно произвести доступ к разделяемому ресурсу одновременно, системе необходим механизм синхронизации для того, чтобы гарантировать использование ресурса только одним процессом. Класс `Mutex` — это примитив синхронизации, который предоставляет эксклюзивный доступ к разделяемому ресурсу только для одного процесса. Если поток получает семафор, второй поток, желающий получить этот семафор, приостанавливается до тех пор, пока первый поток не освободит семафор.

Можно использовать метод `WaitHandle.WaitOne` для запроса на владение семафором. Поток, владеющий семафором, может запрашивать его в повторяющихся вызовах `Wait` без блокировки выполнения. Однако поток должен вызвать метод `ReleaseMutex` соответственное количество раз для того, чтобы прекратить владеть семафором. Если поток завершается нормально во время владения семафором, состояние семафора задается сигнальным, и следующий ожидающий поток становится владельцем семафора. Если нет потоков, владеющих семафором, его состояние является сигнальным.

Открытые конструкторы	
`Mutex` - конструктор	Перегружен. Инициализирует новый экземпляр класса `Mutex`
Открытые свойства	
`Handle` (унаследовано от `WaitHandle`)	Получает или задает собственный дескриптор операционной системы
Открытые методы	
`Close` (унаследовано от `WaitHandle`)	При переопределении в производном классе освобождает все ресурсы, занимаемые текущим объектом `WaitHandle`
`CreateObjRef` (унаследовано от `MarshalByRefObject`)	Создает объект, который содержит всю необходимую информацию для разработки прокси-сервера, используемого для коммуникации с удаленными объектами
`Equals` (унаследовано от `Object`)	Перегружен. Определяет, равны ли два экземпляра `Object`
`GetHashCode` (унаследовано от `Object`)	Служит хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-таблице
`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
`ReleaseMutex`	Освобождает объект `Mutex` один раз
`ToString` (унаследовано от `Object`)	Возвращает `String`, который представляет текущий `Object`
`WaitOne` (унаследовано от `WaitHandle`)	Перегружен. В случае переопределения в производном классе, блокирует текущий поток до получения сигнала текущим объектом `WaitHandle`

Защищенные методы	
`Dispose` (унаследовано от `WaitHandle`)	При переопределении в производном классе отключает неуправляемые ресурсы, используемые `WaitHandle`, и по возможности освобождает управляемые ресурсы
`Finalize` (унаследовано от `WaitHandle`)	Переопределен. Освобождает ресурсы, удерживаемые текущим экземпляром. В языках C# и C++ для функций финализации используется синтаксис деструктора
`MemberwiseClone` (унаследовано от `Object`)	Создает неполную копию текущего объекта `object`

Многопоточное приложение. Способы синхронизации

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
 {
 public Mutex mutex;

public TextPresentation()
 {
  mutex = new Mutex();
 }

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) {
 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.ReleaseMutex();//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();
 }
 }
 }
```

Листинг 15.15.

Рекомендации по недопущению блокировок потоков

- Соблюдать определенный порядок при выделении ресурсов.
- При освобождении выделенных ресурсов придерживаться обратного (reverse) порядка.
- Минимизировать время неопределенного 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 удается 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 захватывается ресурс.

Введение в программирование на C# 2.0

16. Лекция: Форма: версия для печати и PDA

В этой лекции рассказывается о формах. Форма — это класс. 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

Форма — это класс. 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`.

Объект-представитель класса `Form` — это окно, которое появляется в Windows- приложении. 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 подключены и реализованы функции обработки событий.

Этот класс можно использовать как основу для создания различных вариантов окошек:

- стандартных;
- инструментальных;
- всплывающих;
- borderless;
- диалоговых.
- Создается "окно", а класс называется формой – поскольку в окне приложения 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 создается после установки в `true` свойства `IsMdiContainer`.

Используя доступные в классе Формы свойства, можно определять внешний вид, размер, цвет и особенности управления создаваемого диалога:

- свойство `Text` позволяет специфицировать надпись в заголовке окна приложения;
- свойства `Size` и `DesktopLocation` позволяют определять размеры и положение окна в момент его появления на экране монитора;
- свойство `ForeColor` позволяет изменить предопределенный foreground цвет всех расположенных на форме элементов управления;
- свойства `FormBorderStyle`, `MinimizeBox` и `MaximizeBox` позволяют изменять размеры окна приложения во время его выполнения.

Методы класса обеспечивают управление формой:

- например, метод `ShowDialog` обеспечивает представление формы как модального диалога;
- метод `Show` показывает форму как немодальный диалог;
- метод `SetDesktopLocation` используется для позиционирования формы на поверхности Рабочего стола.

Форма может использоваться как стартовый класс приложения. 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 проста 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 description for Form1.

public class Form1 : System.Windows.Forms.Form
{

// Required designer variable.

private System.ComponentModel.Container components = null;

public Form1()
{
//
// Required for Windows Form Designer support
//
InitializeComponent();
//
// TODO: Add any constructor code after InitializeComponent call
//
}
// Clean up any resources being used.

protected override void Dispose( bool disposing )
```

```
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

private void InitializeComponent()
{
this.components = new System.ComponentModel.Container();
this.Size = new System.Drawing.Size(300,300);
this.Text = "Form1";
}
#endregion

// The main entry Point for the application.

static void Main()
{
Application.Run(new Form1());
}
}
}
```

Листинг 16.1.

Главная проблема – метод `InitializeComponent()`. Это зона ответственности мастера приложений. Категорически 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.

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 данную 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.Close()` – но только для активной формы. Если закрывать неактивную форму, событие не произойдет! 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 отменен, 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" (освобождения ресурсов) 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 на его поверхности элементам управления. Это обеспечивается за счет 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. Здесь могут быть полосы прокрутки.

Каждый контейнер поддерживает набор элементов, который состоит из всех вложенных в него элементов управления. У контейнера имеется свойство `Count`, которое возвращает количество элементов управления, свойство [ИНДЕКСАТОР], методы `Add` и `Remove`.

К числу контейнеров относятся:

- `Form...` Без комментариев!
- `Panel`. Элемент управления, содержащий другие элементы управления. Может быть использован для группировки множества элементов управления, как, например, группа для множества `RadioButton`. По умолчанию внешний вид панели НЕ ИМЕЕТ рамок. Нет у ПАНЕЛИ и заголовка. Рамки можно заказать. Для этого в классе панели используется свойство `BorderStyle`. ПАНЕЛЬ является производным классом от `ScrollableControl`, а это значит, что можно заказать для нее собственные полосы прокрутки путем изменения значения соответствующего свойства.
- `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`, они представлены `TabPage`-объектами, которые могут добавляться через свойство `Controls`. Каждая `TabPage`-страница подобна листку записной книжки: страница с закладкой. 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` и `Control.MouseMove`. Следует иметь в виду, что при взаимодействии пользователя со страницей генерируются одни события, при взаимодействии с "корешком" элемента управления – 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

Это 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 счет 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

Провайдер дополнительных свойств – компонент (не элемент управления, а именно компонент!), который "предоставляет" свойства другим компонентам. Провайдеры дополнительных свойств обеспечивают дополнительные возможности при реализации элементов управления.

К числу провайдеров относятся:

- `ToolTipProvider`;
- `HelpProvider`;
- `ErrorProvider`.

Когда `ToolTip Component` на этапе разработки добавляется к форме, все остальные компоненты, размещенные 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`, может быть предоставлено для каждого элемента управления данной формы. Однако не надо обольщаться. Множество свойств элементов управления при этом не изменяется. При выполнении приложения дополнительное свойство остается недоступным через данный конкретный элемент управления.

В следующем примере кода форма была построена с кнопочкой, которая была названа `MyButton`, и элементом управления `ToolTip`, названным `MyToolTip` и "представляющим" кнопочке свойство `ToolTip`.

```
// И вот так просто и наивно значение свойства кнопочки не получить,
// поскольку это свойство для данного элемента управления – не родное!
string myString;
myString = MyButton.ToolTip;
```

Подобное обращение приведет к ошибке еще на стадии компиляции. А вот как надо обращаться к этому самому дополнительному свойству, установленному для кнопочки `MyBotton`, для получения ранее назначенного этому элементу управления совета (просто форменное надувательство):

```
string myString;
myString = MyToolTip.GetToolTip(MyButton); // Объект MyToolTip предоставляет
// ToolTip (совет!) для элемента управления MyButton.
```

Провайдеры дополнительных свойств являются классами, а это означает, что у них имеются собственные свойства, методы и даже события.

Следующий программный код демонстрирует использование провайдеров.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace Rolls01
{

// Summary description for numPointsForm.

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!"
                         };
 }


// Clean up any resources being used.

protected override void Dispose( bool disposing )
{
 if( disposing )
 {
 if(components != null)
 {
 components.Dispose();
 }
}
 base.Dispose( disposing );
}

#region Windows Form Designer generated code

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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;
 // Обращение к Error-провайдеру.
 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();
}
}
}
```

Листинг 16.2.

`ToolTipProvider`, не имея собственного внешнего представления, тем не менее обеспечивает визуализацию дополнительной информации, которая в соответствии с замыслом разработчика формы предоставляется пользователю приложения. Поэтому среди свойств объекта-представителя `ToolTipProvider` имеются такие свойства, связанные с формой подачи дополнительной информации, как:

- `BackColor` — задает цвет фона выводимой информации;
- `ForeColor` — задает цвет текста выводимой информации;
- `ToolTipIcon` — сопутствующая пиктограмма;
- `ToolTipTitle` — дополнительный заголовок;
- `IsBalloon` — совет размещается в рамочке, в стиле прямой речи в комиксах.

Для `ToolTip` провайдера объявляются два события:

- `Draw` — это событие возникает при условии установления в "true" свойства `OwnerDraw` (особая тема для исследования),
- `Popup` — возникает непосредственно в момент "появления" совета.

Следующий фрагмент программного кода демонстрирует вариант обработки события `Popup`:

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

namespace WindowsApplication2
{
public partial class Form1 : Form
{
Random rnd;
float rp;

// Массив советов!
string[] tipsForPlusButton =
{
  "Эта кнопочка используется для вычисления суммы пары слагаемых.",
  "Жми на эту кнопочку, дружок!",
  "Не робей, эта кнопочка специально для тебя...",
  "Нажимать с осторожностью.",
  "Кнопка ПЛЮС"
};

// Специально объявляемые заголовки для выдаваемых советов.
string[] tipsTitles =
{
 "Страшные случаи из жизни",
 "Совет по заполнению первого окна",
 "Совет по заполнению второго окна",
 "Сумма"
};
```

```
public Form1()
{
    rnd = new Random();
    InitializeComponent();
}


// При загрузке формы элементы управления "получают" собственные советы.
private void Form1_Load(object sender, EventArgs e)
{
    this.xToolTip.SetToolTip
        (this.plusButton, tipsForPlusButton[0]);
    this.xToolTip.SetToolTip
        (
         this.textBox1,
         "Текстовое окно для ввода значения первого слагаемого"
         );
    this.xToolTip.SetToolTip
        (
         this.textBox2,
         "Текстовое окно для ввода значения второго слагаемого"
        );
    this.xToolTip.SetToolTip
        (
         this.textBox3,
         "Текстовое окно для вывода значения суммы слагаемых"
        );

    this.textBox1.Text = "0";
    this.textBox2.Text = "0";
}

// При нажатии кнопочки в провайдере заменяется совет.
private void plusButton_Click(object sender, EventArgs e)
{
    x = int.Parse(this.textBox1.Text) + int.Parse(this.textBox2.Text);
    this.textBox3.Text = x.ToString();
    this.xToolTip.SetToolTip
        (
         this.plusButton,
         tipsForPlusButton[rnd.Next(0, tipsForPlusButton.Length)]
        );
}

// В обработчике события можно определить элемент управления,
// для которого выдается совет. При этом изменяется заголовок совета.
private void xToolTip_Popup(object sender, PopupEventArgs e)
{
    if (e.AssociatedControl.Name.Equals("plusButton"))
    {
        xToolTip.ToolTipTitle = tipsTitles[0];
        this.Text = "tip for plusButton";
    }
    else if (e.AssociatedControl.Name.Equals("textBox1"))
    {
        xToolTip.ToolTipTitle = tipsTitles[1];
    }
    else if (e.AssociatedControl.Name.Equals("textBox2"))
    {
        xToolTip.ToolTipTitle = tipsTitles[2];
    }
    else if (e.AssociatedControl.Name.Equals("textBox3"))
    {
        xToolTip.ToolTipTitle = tipsTitles[3];
    }
}
}
}
```

Листинг 16.3.

Validating и Validated элементов управления

Предполагается, что свойство `CausesValidation` элементов управления, для которых будет проводиться проверка, установлено в `true`. Это позволяет отработать обработчику событие `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 корректное 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, отследить 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 еще 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 все... ",
 ""
 };


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

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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 нажимать не надо!");
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

 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, "Ну целое же...");
 }
 }
 }
 }
```

Листинг 16.4.

Управление посредством сообщений

Когда при программировании Windows-приложений еще не использовались элементы MFC, когда еще не было карт сообщений, оконная процедура `WinMain` определялась явным образом, и в ней содержался ЦИКЛ, и можно было наблюдать устройство механизма непрерывного "прослушивания" и интерпретациии перехватываемых сообщений системы, передаваемых данному Windows-приложению.

При этом становилось очевидным, что вся работа приложения фактически сводится к установлению соответствия (с использованием простого оператора выбора) между распознанным в этом цикле сообщением и соответствующей функцией-обработчиком.

Естественно, при этом производился вызов соответствующей функции с возможной передачей этой функции параметров. А попадание в этот цикл обеспечивалось достаточно тривиальной стандартной последовательностью операторов.

С появлением MFC этот цикл при помощи достаточно простой стандартной серии 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 путем 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 НЕСЕТ 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, который предназначен 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.

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, который настроен на метод, обеспечивающий реакцию на данное событие. Когда создается 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. 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. предопределенный 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;
}

}
//=============================================================
//Базовый класс действующих в приложении объектов.
//Содержит ВСЕ необходимое для того, чтобы объекты производных классов
// могли адекватно реагировать на заложенные в базовом классе события.

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 происходит 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!
//=====================================================================


// Объявление первого производного класса.
// Надо сделать дополнительные усилия для того, чтобы объекты этого класса
// стали бы реагировать на события.

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());
}
}
// Второй класс чуть сложнее.
// Снабжен конструктором, который позволяет классу
// самостоятельно "подписаться" на "интересующее" его событие.

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 перестает
// получать уведомление о событии.
BaseClass.xEvent -= new System.EventHandler(one.MyFun);
// После чего повторно запускается цикл прослушивания,
// который прекращает выполняться после повторного
// получения отрицательного значения.
BaseClass.XScaner();
Console.WriteLine("2_____________________________");
}
}
}
```

Листинг 16.5.

Класс Application

`Public sealed class Application` – класс, закрытый для наследования.

Предоставляет СТАТИЧЕСКИЕ (и только статические!) методы и свойства для общего управления приложением, предоставляет 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 неперехваченного 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 неперехваченного 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 Windows сообщений для работы с сообщениями.

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
Свойство HWnd. Get, set . Дескриптор окна (window handle)	`public IntPtr HWnd (get; set;)`
Свойство Msg. Get, set. ID сообщения	`public int Msg {get; set;}`
Свойство WParam. Get, set. Поле WParam сообщения. Значение этого поля зависит от конкретного сообщения. Поле WParam обычно используется для фиксирования небольших фрагментов информации, например, значений флагов	`public IntPtr WParam {get; set;}`
Свойство LParam property. Get, set. Поле LParam зависит от типа сообщения. Значением поля может быть ссылка на объект (object)	`public IntPtr Lparam {get; set;}`
Свойство Result. Специфицирует значение, которое возвращает Windows после перехвата сообщения	`public IntPtr Result {get; set;}`
Create method. Создает новое сообщение — представитель структуры Message	`public static Message Create ( IntPtr hWnd, int msg, IntPtr wparam, IntPtr lparam);`
operator ==. Сравнивает два сообщения на предмет определения их идентичности	`public static Boolean operator ==( Message left, Message right);`
operator !=. Сравнивает два сообщения на предмет определения их различия	`public static Boolean operator !=( Message left, Message righ);`
Equals метод. Сравнивает два сообщения для определения их эквивалентности	`public override bool Equals(object o)`
GetHashCode method. Вычисляет hash код дескриптора (handle) сообщения	`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 бита — 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.

Например, объекты IntPtr используются в классе System.IO.FileStream для работы с дескрипторами файлов.

Существует еще 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 description for Class1.

class Class1
{

// The main entry Point for the application.


static void Main(string[] args)
{
Application.AddMessageFilter(new MyMessageFilter());
Application.Run();
}
}
}
```

Листинг 16.6.

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 description for Class1.

class Class1
{

// The main entry Point for the application.

static void Main(string[] args)
{
if (Application.MessageLoop) Console.WriteLine("Yes!");
else Console.WriteLine("No!");

Application.Run(new MyForm());
}
}
}
```

Листинг 16.7.

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-метод 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 переопределен. При его переопределении следует иметь в виду, что для адекватного поведения приложения в среде Windows необходимо обеспечить вызов базовой версии метода WndProc. Все, что не было обработано в рамках переопределенного 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 передается 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;

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 переопределенная 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);
}
}

}
}
```

Листинг 16.8.

Контекст приложения

Создается простейшее Windows-приложение с единственной формой. Точка входа в приложение – статическая функция `Main` – располагается непосредственно в классе формы. Здесь форма создается, инициализируется, показывается:

```
// Этот явно избыточный набор пространств имен 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 description for Form1.

public class Form1 : System.Windows.Forms.Form
{

// Required designer variable.

private System.ComponentModel.Container components = null;

public Form1()
{
//
// Required for Windows Form Designer support
//
InitializeComponent();

//
// TODO: Add any constructor code after InitializeComponent call
//
}


// Clean up any resources being used.

protected override void Dispose( bool disposing )
{
if( disposing )
{
if (components != null)
{
components.Dispose();
}
}
base.Dispose( disposing );
}

#region Windows Form Designer generated code

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

private void InitializeComponent()
{
this.components = new System.ComponentModel.Container();
this.Size = new System.Drawing.Size(300,300);
this.Text = "Form1";
}
#endregion

// The main entry Point for the application.

static void Main()
{
Application.Run(new Form1());
}
}
}
```

Листинг 16.9.

Приложение, в задачи которого входит поддержка нескольких одновременно существующих (и, возможно, взаимодействующих) форм, создается 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 имен. 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

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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 - контекста приложения на событие
// закрытия формы.
this.Closed+=new EventHandler(appContext.OnFormClosed);
}
#endregion

// The main entry Point for the application.
// Точка входа в приложение перенесена в класс
// контекста приложения.

//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 - счетчик открытых форм уменьшается.
// Приложение сохраняет работоспособность до тех пор,
// пока значение счетчика не окажется равным 0.
// После этого - Exit().
formCount--;
if (formCount == 0)
{
Application.Exit();
}
}

static void Main(string[] args)
{
// Создается объект - представитель класса MyApplicationContext,
// который берет на себя функции управления приложением.
MyApplicationContext context = new MyApplicationContext();
// Приложение запускается для объекта контекста приложения.
Application.Run(context);
}
}
}
```

Листинг 16.10.

Введение в программирование на C# 2.0

17. Лекция: GDI+: версия для печати и PDA

GDI+ (Graphic Device Interface+ — Интерфейс Графических Устройств) — это подсистема 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. Взаимодействию GDI и C# посвящена эта лекция.

GDI+ - это набор программных средств, которые используются 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.

Graphics

Класс, который ИНКАПСУЛИРУЕТ поверхность рисования GDI+. Для этого класса не определен ни один конструктор. Видимо, успех в деле ручного конструирования инкапсулятора поверхностей рисования (еще вопрос, сколько разновидностей таких поверхностей) представляется проблематичным.

Конкретный объект – представитель класса Graphics предоставляется в виде ссылки методами-обработчиками событий либо создается в ходе выполнения ряда методов применительно к конкретным объектам, обладающим "поверхностями рисования" (клиентская область формы, кнопки, панели, битовая матрица):

```
Bitmap bmp;
Pen greedPen;
:::::
greedPen = new Pen(Color.FromArgb(50, 0, 0, 255), 1);
// Новая битовая карта под новый размер клиентской области формы.
bmp = new Bitmap(this.ClientSize.Width, this.ClientSize.Height);
Graphics gr = Graphics.FromImage(bmp);
gr.DrawLine(this.greedPen, 0, 0, 100, 100);
gr.Dispose();
```

Ниже представлен список членов класса.

Clip	Получает или задает объект Region, ограничивающий область рисования данного объекта Graphics
ClipBounds	Получает структуру RectangleF, которая заключает в себе вырезанную область данного объекта Graphics
CompositingMode	Получает значение, задающее порядок рисования сложных изображений в данном объекте Graphics
CompositingQuality	Получает или задает качество отображения сложных изображений, которые выводятся в данном объекте Graphics
DpiX	Получает горизонтальное разрешение данного объекта Graphics
DpiY	Получает вертикальное разрешение данного объекта Graphics
InterpolationMode	Получает или задает режим вставки, связанный с данным объектом Graphics
IsClipEmpty	Получает значение, которое указывает, является ли вырезанная область данного объекта Graphics пустой
IsVisibleClipEmpty	Получает значение, которое указывает, является ли видимая вырезанная область данного объекта Graphics пустой
PageScale	Получает или задает масштабирование между универсальными единицами и единицами страницы для данного объекта Graphics
PageUnit	Получает или задает единицу измерения для координат страницы данного объекта Graphics
PixelOffsetMode	Получает или задает значение, которое задает порядок смещения точек во время отображения данного объекта Graphics
RenderingOrigin	Получает или задает исходное заполнение данного объекта Graphics для сглаживания цветовых переходов и для штриховки
SmoothingMode	Получает или задает качество заполнения для данного объекта Graphics
TextContrast	Получает или задает значение коррекции яркости для отображения текста
TextRenderingHint	Получает или задает режим заполнения для текста, связанного с данным объектом Graphics
Transform	Получает или задает универсальное преобразование для данного объекта Graphics
VisibleClipBounds	Получает или задает рабочий прямоугольник видимой вырезанной области данного объекта Graphics

Открытые методы

AddMetafileComment	Добавляет комментарий к текущему объекту Metafile
BeginContainer	Перегружен. Сохраняет графический контейнер, содержащий текущее состояние данного объекта Graphics, а затем открывает и использует новый графический контейнер
Clear	Очищает всю поверхность рисования и выполняет заливку поверхности указанным цветом фона
CreateObjRef	Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для коммуникации с удаленными объектами
Dispose	Освобождает все ресурсы, используемые данным объектом Graphics
DrawArc	Перегружен. Рисует дугу, которая является частью эллипса, заданного парой координат, шириной и высотой
DrawBezier	Перегружен. Строит кривую Безье, определяемую четырьмя структурами Point
DrawBeziers	Перегружен. Формирует набор кривых Безье из массива структур Point
DrawClosedCurve	Перегружен. Строит замкнутую фундаментальную кривую, определяемую массивом структур Point
DrawCurve	Перегружен. Строит замкнутую фундаментальную кривую через точки указанного массива структур Point

DrawEllipse	Перегружен. Формирует эллипс, который определяется ограничивающим прямоугольником, заданным с помощью пары координат — ширины и высоты
DrawIcon	Перегружен. Формирует изображение, которое представлено указанным объектом Icon, расположенным по указанным координатам
DrawIconUnstretched	Формирует изображение, представленное указанным объектом Icon, не масштабируя его
DrawImage	Перегружен. Рисует заданный объект Image в заданном месте, используя исходный размер
DrawImageUnscaled	Перегружен. Рисует заданное изображение, используя его исходный фактический размер, в расположении, заданном парой координат
DrawLine	Перегружен. Проводит линию, соединяющую две точки, определенные парами координат
DrawLines	Перегружен. Формирует набор сегментов линии, которые соединяют массив структур Point
DrawPath	Рисует объект GraphicsPath
DrawPie	Перегружен. Рисует сектор, определенный эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями
DrawPolygon	Перегружен. Рисует многоугольник, определяемый массивом структур Point
DrawRectangle	Перегружен. Рисует прямоугольник, который определен парой координат, шириной и высотой
DrawRectangles	Перегружен. Рисует набор прямоугольников, определяемых структурой Rectangle
DrawString	Перегружен. Создает текстовую строку в заданном месте с указанными объектами Brush и Font
EndContainer	Закрывает текущий графический контейнер и восстанавливает состояние данного объекта Graphics, которое было сохранено при вызове метода BeginContainer
EnumerateMetafile	Перегружен. Отправляет записи указанного объекта Metafile по отдельности методу обратного вызова, который отображает их в заданной точке
Equals	Перегружен. Определяет, равны ли два экземпляра Object
ExcludeClip	Перегружен. Обновляет вырезанную область данного объекта Graphics, чтобы исключить из нее часть, определенную структурой Rectangle
FillClosedCurve	Перегружен. Заполняет замкнутую фундаментальную кривую, определяемую массивом структур Point
FillEllipse	Перегружен. Заполняет внутреннюю часть эллипса, который определяется ограничивающим прямоугольником, заданным с помощью пары координат — ширины и высоты
FillPath	Заполняет внутреннюю часть объекта GraphicsPath
FillPie	Перегружен. Заполняет внутреннюю часть сектора, определенного эллипсом, который задан парой координат, шириной, высотой и двумя радиальными линиями
FillPolygon	Перегружен. Заполняет внутреннюю часть многоугольника, определенного массивом точек, заданных структурами Point
FillRectangle	Перегружен. Заполняет внутреннюю часть прямоугольника, который определен парой координат, шириной и высотой
FillRectangles	Перегружен. Заполняет внутреннюю часть набора прямоугольников, определяемого структурами Rectangle
FillRegion	Заполняет внутреннюю часть объекта Region
Flush	Перегружен. Вызывает принудительное выполнение всех отложенных графических операций и немедленно возвращается, не дожидаясь их окончания
FromHdc	Статический. Перегружен. Создает новый объект Graphics из указанного дескриптора для контекста устройства
FromHdcInternal	Статический. Внутренний метод. Не используется
FromHwnd	Статический. Создает новый объект Graphics из указанного дескриптора для окна
FromHwndInternal	Статический. Внутренний метод. Не используется
FromImage	Статический. Создает новый объект Graphics из заданного объекта Image
GetHalftonePalette	Статический. Получает дескриптор текущей полутоновой палитры Windows
GetHashCode	Служит хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-таблице
GetHdc	Получает дескриптор контекста устройства, связанный с данным объектом Graphics
GetLifetimeService	Извлекает служебный объект текущего срока действия, который управляет средствами срока действия данного экземпляра
GetNearestColor	Получает цвет, ближайший к указанной структуре Color
GetType	Возвращает Type текущего экземпляра
InitializeLifetimeService	Получает служебный объект срока действия, для управления средствами срока действия данного экземпляра
IntersectClip	Перегружен. Обновляет вырезанную область данного объекта Graphics, включая в нее пересечение текущей вырезанной области и указанной структуры Rectangle
IsVisible	Перегружен. Указывает, содержится ли точка, заданная с помощью пары координат, в видимой вырезанной области данного объекта Graphics
MeasureCharacterRanges	Получает массив объектов Region, каждый из которых связывает диапазон позиций символов в рамках указанной строки
MeasureString	Перегружен. Измеряет указанную строку в процессе ее создания с помощью заданного объекта Font
MultiplyTransform	Перегружен. Умножает универсальное преобразование данного объекта Graphics на преобразование указанного объекта Matrix
ReleaseHdc	Освобождает дескриптор контекста устройства, полученный в результате предыдущего вызова метода GetHdc данного объекта Graphics
ReleaseHdcInternal	Внутренний метод. Не используется
ResetClip	Сбрасывает вырезанную область данного объекта Graphics и делает ее бесконечной
ResetTransform	Сбрасывает матрицу универсального преобразования данного объекта Graphics и делает ее единичной матрицей

Restore	Восстанавливает состояние данного объекта Graphics, возвращая его к состоянию объекта GraphicsState
RotateTransform	Перегружен. Применяет заданное вращение к матрице преобразования данного объекта Graphics
Save	Сохраняет текущее состояние данного объекта Graphics и связывает сохраненное состояние с объектом GraphicsState
ScaleTransform	Перегружен. Применяет указанную операцию масштабирования к матрице преобразования данного объекта Graphics путем ее добавления к матрице преобразования объекта
SetClip	Перегружен. Задает в качестве вырезанной области данного объекта Graphics свойство Clip указанного объекта Graphics
ToString	Возвращает String, который представляет текущий Object
TransformPoints	Перегружен. Преобразует массив точек из одного координатного пространства в другое, используя текущее универсальное преобразование и преобразование страницы данного объекта Graphics
TranslateClip	Перегружен. Переводит вырезанную область данного объекта Graphics в указанном объеме в горизонтальном и вертикальном направлениях.
TranslateTransform	Перегружен. Добавляет заданный перевод к матрице преобразования данного объекта Graphics

Защищенные методы

Finalize	Переопределен. См. Object.Finalize. В языках C# и C++ для функций финализации используется синтаксис деструктора
MemberwiseClone	Создает неполную копию текущего Object

Битовая карта как поверхность для рисования

Приводимое ниже приложение демонстрирует технику рисования на невидимых виртуальных поверхностях – в буквальном смысле в оперативной памяти.

Приложение моделирует случайное блуждание множества однородных частиц. Несмотря на интенсивный вывод графической информации, удается избежать эффекта мигания, который возникает при непосредственной модификации внешнего вида элементов управления.

Демонстрация предварительно нарисованной в памяти картинки происходит при помощи элемента управления типа PictureBox, который благодаря свойству объекта Image обеспечивает быстрое отображение выводимой графической информации.

Итак, данное Windows-приложение включает объявление трех классов:

- класса xPoint, объекты которого периодически изменяют собственное положение в пределах клиентской области окна;
- класса cForm, который обладает свойствами сохранения постоянного соотношения собственных размеров и размеров клиентской области, и обеспечивает отображение множества блуждающих объектов – представителей класса xPoint;
- класса Program, который в соответствии с предопределенными алгоритмами обеспечивает создание и выполнение приложения.

Класс Program

Создается автоматически. Без комментариев.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace cForm
{
    static class Program
    {

        // The main entry point for the application.

        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new cForm());
        }
    }
}
```

Класс xPoint

Объявление класса блуждающих объектов:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Text;
namespace cForm
{
class xPoint
{
// Собственный статический генератор случайных чисел.
// Используется в конструкторе.
public static Random rnd = new Random();
```

```
// Цвет объекта.
public Color xColor;

// Текущая позиция. Относительные значения координат X и Y.
public Point p;

// Обеспечение перемещения. Текущие значения координат могут
// изменяться как по X, так и по Y.
public Point direction;

// Счетчик циклов сохранения выбранного направления.
// Объект на протяжении фиксированного интервала времени может
// сохранять ранее избранное направление движения.
// В течение n.X "тиков" для X в течение n.Y "тиков" для Y.
public Point n;

public xPoint(int X, int Y, int Xn, int Yn)
{
n = new Point(rnd.Next(0, Xn), rnd.Next(0, Yn));
direction = new Point(rnd.Next(-1, 2), rnd.Next(-1, 2));
p = new Point(X, Y);
xColor =
  Color.FromArgb(150, rnd.Next(0, 256), rnd.Next(0, 256), rnd.Next(0, 256));
}

// Статический метод. Еще вопрос, где следовало размещать его
// объявление... Определение нового положения амебы.
// Просматривается ВЕСЬ список.
// Ее перемещение ограничивается физическими размерами клиентской области
// окна приложения и определяется по двум координатам (X и Y).
// В каждый момент она может оставаться на месте, либо изменить
//  свое положение на один "шаг" по каждой из осей координат
// ("вверх" или "вниз" по оси Y и "вперед" или "назад" по оси X).
public static void setNextPosition(cForm cf, xPoint[] pt)
{
int i;
float xy;

// Итак, определение текущей позиции объекта.
// Просмотр массива...
for (i = 0; i < pt.Length; i++)
{
// Сначала разбираемся со значением координаты X.
// Вычисляем возможную позицию после перемещения объекта.
xy = (float)((pt[i].p.X + pt[i].direction.X) * cf.rPointGet);
// Вполне возможно, что это вполне подходящая позиция.
// И надо всего лишь проверить две вещи:
// 1. не выскочит ли объект за пределы клиентской области окна
// приложения,
// 2. не настало ли время поменять направление движения.
if (
xy < 0 || xy > cf.ClientSize.Width        // 1.
||
pt[i].n.X > cf.NX                 // 2.
)
{
pt[i].n.X = 0; // Обнулили счетчик циклов сохранения выбранного направления.
// Процедура изменения направления перемещения по оси X.
// На ближайшую перспективу объект может переместиться
// вперед:            pt[i].direction.X == 1,
// назад:             pt[i].direction.X == -1,
// остаться на месте: pt[i].direction.X == 0.
// Главное — это не выйти за пределы клиентской области окна приложения.
    pt[i].direction.X = xPoint.rnd.Next(-1, 2);
    xy = (float)((pt[i].p.X + pt[i].direction.X) * cf.rPointGet);
    if (xy >= 0 && xy <= cf.ClientSize.Width)
    {
        // Направление выбрано, перемещение произведено.
        pt[i].p.X += pt[i].direction.X;
    }
    else
    {
        // Выбранное направление движения приводит к выходу объекта
        // за пределы клиентской области окна приложения.
        // На ближайшие cf.NX тактов объект остается неподвижен по оси X.
        pt[i].direction.X = 0;
    }
}
else
{
// Осуществили очередное перемещение по оси X.
pt[i].p.X += pt[i].direction.X;
pt[i].n.X++;
}

xy = (float)((pt[i].p.Y + pt[i].direction.Y) * cf.rPointGet);
// Вполне возможно, что это вполне подходящая позиция.
// И надо всего лишь проверить две вещи:
// 1. не выскочит ли объект за пределы клиентской области
// окна приложения,
```

```
// 2. не настало ли время поменять направление движения.
if (
xy < 0 || xy > cf.ClientSize.Height // 1.
||
pt[i].n.Y > cf.NY                   // 2.
)
{
pt[i].n.Y = 0; // Обнулили счетчик циклов сохранения выбранного направления.
// Процедура изменения направления перемещения по оси Y.
// На ближайшую перспективу объект может переместиться
// вверх:             pt[i].direction.Y == 1,
// вниз:              pt[i].direction.Y == -1,
// остаться на месте: pt[i].direction.Y == 0.
// Главное — это не выйти за пределы клиентской области окна приложения.
pt[i].direction.Y = xPoint.rnd.Next(-1, 2);
xy = (float)((pt[i].p.Y + pt[i].direction.Y) * cf.rPointGet);
if (xy >= 0 && xy <= cf.ClientSize.Height)
{
    // Направление выбрано, перемещение произведено.
    pt[i].p.Y += pt[i].direction.Y;
}
else
{
    // Выбранное направление движения приводит к выходу объекта
    // за пределы клиентской области окна приложения.
    // На ближайшие cf.NY тактов объект остается неподвижен по оси Y.
    pt[i].direction.Y = 0;
}
}
else
{
// Осуществили очередное перемещение по оси Y.
pt[i].p.Y += pt[i].direction.Y;
pt[i].n.Y++;
}
}
}
}
}
```

Листинг 17.1.

Класс cForm

Partial-класс. Содержит объявление класса формы приложения. Первая часть объявления в основном создается автоматически. Следует обратить внимание на настройку событий:

```
namespace cForm
{
partial class cForm
{

// Required designer variable.

private System.ComponentModel.IContainer components = null;

// Clean up any resources being used.
// <param name="disposing">true if managed resources should be disposed;
// otherwise, false.</param>
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}

#region Windows Form Designer generated code


// Required method for Designer support - do not modify
// the contents of this method with the code editor.

private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.mainPictureBox = new System.Windows.Forms.PictureBox();
    this.mainTimer = new System.Windows.Forms.Timer(this.components);
  ((System.ComponentModel.ISupportInitialize)(this.mainPictureBox)).
                        BeginInit();
    this.SuspendLayout();
    //
    // mainPictureBox
    //
    this.mainPictureBox.BackColor = System.Drawing.Color.White;
    this.mainPictureBox.BorderStyle =
                  System.Windows.Forms.BorderStyle.FixedSingle;
    this.mainPictureBox.Dock = System.Windows.Forms.DockStyle.Fill;
    this.mainPictureBox.Location = new System.Drawing.Point(0, 0);
```

```
    this.mainPictureBox.Name = "mainPictureBox";
    this.mainPictureBox.Size = new System.Drawing.Size(569, 538);
    this.mainPictureBox.TabIndex = 0;
    this.mainPictureBox.TabStop = false;
    //
    // mainTimer
    //
    this.mainTimer.Interval = 250;
    this.mainTimer.Tick += new System.EventHandler(this.mainTimer_Tick);
    //
    // cForm
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(569, 538);
    this.Controls.Add(this.mainPictureBox);
    this.Name = "cForm";
    this.Text = "cForm";
this.Paint +=
            new System.Windows.Forms.PaintEventHandler(this.bForm00_Paint);
this.Layout +=
new System.Windows.Forms.LayoutEventHandler(this.bForm00_Layout);
((System.ComponentModel.ISupportInitialize)(this.mainPictureBox)).EndInit();
this.ResumeLayout(false);
}

#endregion

private System.Windows.Forms.PictureBox mainPictureBox;

// Определение неизменяемых характеристик рамки окна приложения.
private void GetBordersGabarits()
{
    this.borderHeight = this.Height - this.ClientSize.Height;
    this.borderWidth = this.Width - this.ClientSize.Width;
}

// Определение предельных габаритов окна приложения.
private void SetMinMaxSize()
{
 this.MinimumSize =
   new System.Drawing.Size(minWidth + borderWidth, minHeight + borderHeight);
 this.MaximumSize =
   new System.Drawing.Size(maxWidth + borderWidth, maxHeight + borderHeight);
}
}
}
```

Листинг 17.2.

Вторая часть объявления. Здесь сосредоточены методы управления поведением популяции амеб, методы масштабирования внешнего вида приложения и отображения информации.

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

namespace cForm
{
public partial class cForm : Form
{
// Средства для отображения объектов на плоскости.
Color backColor;
Pen greedPen;
Pen pen0;
SolidBrush br;

// Действующие лица. Массив объектов xPoint.
xPoint[] pt;

// Количество точек.
const int nPT = 10000;

// Синхронизаторы изменения направления перемещения объектов.
public int NX, NY;

// Предельные характеристики размеров клиентской области.
const int minHeight = 100;
const int minWidth = 100;
const int maxHeight = 950;
const int maxWidth = 950;

// Средство масштабирования. Всеобщая единица измерения.
private double rPoint;

// Абсолютные неизменяемые характеристики рамки окна приложения.
```

```
int borderHeight;
int borderWidth;

// Область рисования. Изображение формируется в памяти.
// На "поверхности" битовой карты
// (объект - представитель класса Graphics,
// создается в результате выполнения метода FromImage),
// средствами векторной графики (методы класса Graphics)
// рисуется картинка.
// Изображение проецируется на поверхность элемента управления
// mainPictureBox
// путем модификации свойства Image этого элемента управления.
private Bitmap bmp;

// Таймер. Фактически представляет собой делегат,
// который через фиксированные интервалы
// времени активизирует соответствующее событие
// (передает управление методу mainTimer_Tick).
private System.Windows.Forms.Timer mainTimer;

// Конструктор формы.
public cForm()
{
int i;

// Цвет фона картинки. Используется при перерисовке изображения.
backColor = Color.FromArgb(255,0,0,0);

// Средства для рисования.
greedPen = new Pen(Color.FromArgb(50, 0, 0, 255), 1);
pen0 = new Pen(Color.FromArgb(255,0,0,0),1);
br = new SolidBrush(Color.FromArgb(0, 0, 0, 0));

// Обязательно смотреть код начальной инициализации элементов формы!
InitializeComponent();

// Определение неизменяемых характеристик окна приложения (формы).
GetBordersGabarits();

// Начальная коррекция размеров клиентской области окна приложения.
// "Прямой" вызов метода - обработчика события изменения размеров
// окна приложения.
bForm00_Layout(null, null);

// Определение предельных габаритов окна приложения.
SetMinMaxSize();

// Значения синхронизаторов.
// Неплохо приспособить специальные элементы управления,
// которые позволяли бы изменять эти значения в интерактивном режиме!
NX = 50;
NY = 50;

// Инициализация популяции.
// Все располагаются в одном месте – в центре клиентской области окна.
// Одинаковые значения двух последних параметров приводят к тому, что
// члены популяции синхронно изменяют направление движения.
pt = new xPoint[nPT];
for (i = 0; i < pt.Length; i++)
{
pt[i] = new xPoint(
  (int)(this.ClientSize.Width  / (2*rPoint)),
  (int)(this.ClientSize.Height / (2*rPoint)),
   0, //NX, // xPoint.rnd.Next(0, NX+1), // А такие значения нарушают
   0  //NY  // xPoint.rnd.Next(0, NY+1) // синхронность поведения объектов.
   );
}

// Процесс пошел!
mainTimer.Start();
}

// Определение относительной величины rPoint реализовано как свойство.
// Замечательная особенность свойства!
// Оно имеет явную спецификацию возвращаемого значения.
// И точно такого же типа должен быть единственный параметр value!
private Size rPointSet
{
set
{
Size s = value;
rPoint = Math.Sqrt(2.0*s.Width*s.Height) * 0.002;
}
}

// А вот перегрузить свойство нельзя! Невидимый параметр value
// и возвращаемое значение свойства должны быть одного типа.
// Именно поэтому МНЕ пришлось вводить пару свойств.
public double rPointGet
{
```

```
get
{
return rPoint;
}
}

// Обработчик события перерисовки формы.
// В списке параметров ссылка на объект - инициализатор события,
// а также ссылка на объект – носитель информации об инкапсуляторе
// данной поверхности рисования.
// Способ получения этой информации простой – через свойство Graphics:
//      Graphics gr = e.Graphics;
// И знай рисуй себе!
// Однако эта ссылка нам не понадобится, поскольку мы рисуем
// не на ПОВЕРХНОСТИ ЭЛЕМЕНТА УПРАВЛЕНИЯ,
// а в памяти, на ПОВЕРХНОСТИ БИТОВОЙ КАРТЫ.
private void bForm00_Paint(object sender, PaintEventArgs e)
{
xRedraw();
}

// Событие Layout возникает при удалении или добавлении дочерних
// элементов управления,
// изменении границ элемента управления и других изменениях,
// в том числе и изменении размеров формы, которые могут повлиять
// на макет элемента.
// Рекомендуется использовать именно это событие,
// а не события ..._Resize или ..._SizeChanged. Как сказано
// в руководстве, событие Layout возникает в ответ на события
// ..._Resize и ..._SizeChanged,
// а также в других ситуациях, когда может понадобиться применение макета.
private void bForm00_Layout(object sender, LayoutEventArgs e)
{
// Использование сразу ДВУХ величин – длины и ширины клиентской
// области окна!
// Вот оптимальное решение проблемы поддержки ПОСТОЯННОГО соотношения
// характеристик клиентской области
// (среднее арифметическое для поддержки равенства сторон).
// ПЛЮС поправка на размеры рамки окна приложения.

Size = new System.Drawing.Size
    (
    (ClientSize.Width + ClientSize.Height)/2 + borderWidth,
    (ClientSize.Width + ClientSize.Height)/2 + borderHeight
    );

// После коррекции размеров окна – устанавливаем значение величины
// rPoint, которое производится через обращение к свойству rPointSet.
rPointSet = this.ClientSize;

// Новая битовая карта под новый размер клиентской области.
bmp = new Bitmap(this.ClientSize.Width, this.ClientSize.Height);

// Объявляется недопустимой конкретная область элемента управления
// (здесь форма) и происходит отправка соответствующего сообщения
// (Paint) элементу управления (форме). При этом значение
// параметра (true)
// объявляет недопустимыми назначенные элементу управления
// дочерние элементы.
// Для каждого элемента автоматически формируется
// объект-представитель
// класса Graphics, инкапсулирующий поверхность для рисования.
// Для активизации процесса перерисовки соответствующего
// элемента управления требуется объявить метод – обработчик события
// (в нашем случае это bForm00_Paint).
this.Invalidate(false);
}

// Обработчик события таймера:
// всем амебам надлежит переопределить собственные координаты и
// перерисовать картинку! Здесь также можно было бы объявить
// недействительной и требующей перерисовки поверхность формы.
// Это просто: this.Invalidate(false);
// Однако никто не помешает нам сделать все напрямую, через метод xRedraw.
private void mainTimer_Tick(object sender, EventArgs e)
{
xPoint.setNextPosition(this, pt);
xRedraw();
}

// Перерисовка.
private void xRedraw()
{
// Поверхность для рисования на битовой карте!
Graphics gr = Graphics.FromImage(bmp);

// Очистить поверхность и закрасить ее в ненавязчивый черный цвет!
gr.Clear(backColor);
```

```
// Нарисовать решетку!
DrawGreed(gr);

// Нарисовать объекты!
foreach (xPoint xp in pt)
{
// Сообщили кисти цвет объекта...
br.Color = xp.xColor;
// Закрасили объект.
gr.FillEllipse(br,
               (float)((xp.p.X * rPoint) - rPoint),
               (float)((xp.p.Y * rPoint) - rPoint),
               (float)(2 * rPoint),
               (float)(2 * rPoint));
// Сообщили перу цвет объекта...
pen0.Color = xp.xColor;
// Нарисовали контур амебы...
gr.DrawEllipse(pen0,
               (float)((xp.p.X * rPoint) - rPoint),
               (float)((xp.p.Y * rPoint) - rPoint),
               (float)(2 * rPoint),
               (float)(2 * rPoint));
}

// Рисование закончено. Подготовить объект-поверхность к удалению.
gr.Dispose();

// Отобразить картинку!
// Это всего лишь модификация свойства Image элемента управления PictureBox.
this.mainPictureBox.Image = bmp;
}

// На поверхности битовой карты (параметр gr) рисуется координатная сетка.
private void DrawGreed(Graphics gr)
{
float i;
float w = (float)(this.ClientSize.Width/25.0F);
float h = (float)(this.ClientSize.Height/25.0F);
float W = this.ClientSize.Width;
float H = this.ClientSize.Height;

for (i = h; i < H; i += h)
{
gr.DrawLine(this.greedPen, 0, i, W, i);
}

for (i = w; i < W; i += w)
{
gr.DrawLine(this.greedPen, i, 0, i, H);
}
}
}
}
```

Листинг 17.3.

Запустили приложение, порадовались отсутствию утомляющего глаза мигания, характерного для непосредственного рисования на поверхности элемента управления... подумали над созданием элементов управления для изменения синхронности перемещения членов амебной популяции.

GraphicsPath

`GraphicsPath` – класс, представляющий последовательность соединенных линий и кривых. 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 задается 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`
`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 конструирования 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 хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например, в хэш-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, которые окружают 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

Класс Region (Область) предназначается для создания объектов, которые описывают внутреннюю часть графической формы 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 применяет 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 пикселей, 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
DisposeEquals	Освобождает все ресурсы, используемые объектом Region
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, которая представляет прямоугольник, ограничивающий объект Region на поверхности рисунка объекта Graphics
GetHashCode (унаследовано от Object)	Служит хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-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, который представляет 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
{

// Объявляется класс - наследник базового класса Form.
// Наша форма, конечно же, основывается на обычной классической
// прямоугольной, масштабируемой форме.
```

```
// Прежде всего это означает, что у нее 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 каемочкой.

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
// конфигурации нашей формы.
int bw;

// Ничем не примечательный конструктор...
public RoundForm()
{
       InitializeComponent();
}

#region Windows Form Designer generated code

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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

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, определяющий 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 переопределенной виртуальной функции...
protected override void OnPaint(PaintEventArgs e)
{
// Определяем габариты формы.
SetSize();
// Получаем ссылку на графический контекст.
Graphics gr = this.CreateGraphics();
// Карандаш для отрисовки каемочки.
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 определения.
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 создаем 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 – оно прекрасно.
this.Region = reg;
// Рисуем каемочку...
gr.DrawEllipse(pen,0,0,this.ClientSize.Width,this.ClientSize.Height);
// Освобождаем занятые ресурсы.
gr.Dispose();
}
}
}
```

Листинг 17.4.

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 удается частично использовать функциональности 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 клиентской области элемента управления. Член этого класса объект типа `Graphics` – обеспечивает формирование представления 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 четких границ. В обоих случаях основанием для классификации оказывается объем 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 варианте 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 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.

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 передается 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
	//
}
}
}
```

Листинг 17.5.

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 посвящен разработке составного элемента управления, который представляет 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, связанным 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 параметров используется для идентификации нажатой кнопки. При нажатии на кнопку происходит изменение состояния этой кнопки, которое кодируется посредством текстового значения (свойства `Text`), отображаемого на поверхности кнопки.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Windows.Forms;

namespace By2
{

// Summary description for ButtonsControl.

public class ButtonsControl : System.Windows.Forms.UserControl
{

// Required designer variable.
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();
}

}


// Clean up any resources being used.

protected override void Dispose( bool disposing )
{
if( disposing )
{
       if(components != null)
       {
        components.Dispose();
       }
}
base.Dispose( disposing );
}

#region Component Designer generated code

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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();
}
```

```
  }
  }
```

Листинг 17.6.

Введение в программирование на C# 2.0

18. Лекция: Основы ADO .NET: версия для печати и PDA

ADO .NET (ActiveX Data Objects .NET) является набором классов, реализующих программные интерфейсы для облегчения подключения к базам данных из приложения независимо от особенностей реализации конкретной системы управления базами данных и от структуры самой базы данных, а также независимо от места расположения этой самой базы — в частности, в распределенной среде (клиент-серверное приложение) на стороне сервера. Работу C# с ADO обсуждает данная лекция

ADO .NET (ActiveX Data Objects .NET) является набором классов, реализующих программные интерфейсы для облегчения подключения к базам данных из приложения независимо от особенностей реализации конкретной системы управления базами данных и от структуры самой базы данных, а также независимо от места расположения этой самой базы — в частности, в распределенной среде (клиент-серверное приложение) на стороне сервера.

ADO .NET широко используется совместно с технологией web-программирования с использованием объектов ASP .NET для доступа к расположенным на сервере базам данных со стороны клиента.

Особенность изложения материала этой главы заключается в следующем.

Решение даже самой простой задачи, связанной с данными, предполагает использование множества разнообразных объектов – представителей классов ADO .NET, которые находятся между собой в достаточно сложных взаимоотношениях. Из-за этих отношений строго последовательное описание элементов ADO .NET представляется весьма проблематичным. С какого бы элемента ни начиналось описание, всегда предполагается предварительное представление о множестве других элементов.

По этой причине часто упоминание и даже примеры использования некоторых классов предшествуют их подробному описанию.

Реляционные базы данных. Основные понятия

Ниже обсуждаются наиболее общие понятия, связанные с представлением реляционной базы данных с точки зрения программиста, использующего ADO .NET.

Детали реализации конкретной базы данных в рамках данной СУБД не обсуждаются. ADO .NET для того и используется, чтобы максимально скрыть детали реализации конкретной базы и предоставить программисту набор стандартных классов, интерфейсов, программных средств, которые превращают процесс создания приложения в стандартизированный технологический процесс. Таким образом, с точки зрения .NET:

столбец (поле, атрибут) —

- характеризуется определенным 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 определенного типа,
- состоит из множества элементов столбцов-строк,
- каждая строка таблицы УНИКАЛЬНА;

первичный ключ таблицы —

- непустое множество столбцов таблицы (возможно, 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 еще одна гарантия уникальности строки в таблице;

внешний ключ таблицы —

- непустое множество столбцов таблицы (возможно, состоящее из одного столбца), соответствующие значения (комбинации значений) которых в строках таблицы соответствуют первичному или дополнительному ключу другой таблицы,
- обеспечивает логическую связь между таблицами.

Работа с базами данных

Работа с БД на уровне приложения .NET – это работа:

- с множествами объявлений классов, которые содержат объявления унаследованных методов и свойств, предназначенных для решения задачи извлечения информации из базы данных;
- с множеством объектов-представителей классов, которые обеспечивают работу с базами данных;
- с множеством значений и свойств конкретных объектов, отражающих специфику структуры конкретной базы данных.

Функциональные особенности этой сложной системы взаимодействующих классов обеспечивают ЕДИНООБРАЗНУЮ работу с базами данных независимо от системы управления базой и ее реализации.

Конечно же, при написании программы, взаимодействующей с базами данных, программист все может сделать своими руками. Правда, в силу сложности задачи (много всяких деталек придется вытачивать), времени на разработку такого приложения может потребоваться достаточно много.

Для программиста – разработчика приложения принципиальной становится информация о логической организации (структуре таблиц, отношениях и ограничениях) данной конкретной базы данных — то есть о том, как эту базу видит приложение.

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.

А потому желательно:

- понимать принципы организации и взаимодействия классов, которые обеспечивают работу с базой данных;
- представлять структуру, назначение и принципы работы соответствующих объектов;
- знать, для чего и как применять различные детали при организации взаимодействия с базами данных;
- уметь создавать компоненты ADO .NET заданной конфигурации с использованием вспомогательных средств (волшебников), предоставляемых в рамках Visual Studio .NET.

Доступ к отсоединенным данным

В приложениях, работающих с базами данных, до недавних пор применялся доступ к данным через постоянное соединение с источником данных. Приложение открывало соединение с базой данных и не закрывало его по крайней мере до завершения работы с источником данных. В это время соединение с источником поддерживалось постоянно.

Недостатки такого подхода стали выявляться после появления приложений, ориентированных на Интернет.

Соединения с базой данных требуют выделения системных ресурсов, и если база данных располагается 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.

Факт плохого масштабирования приложений с постоянным соединением известен давно. Соединение с парой клиентов обслуживается приложением хорошо, 10 клиентов обслуживаются хуже, 100 – много хуже...

В ADO .NET используется другая модель доступа – доступ к отсоединенным данным. При этом соединение устанавливается лишь на то время, которое необходимо для проведения определенной 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.

ADO .NET. Доступ к данным

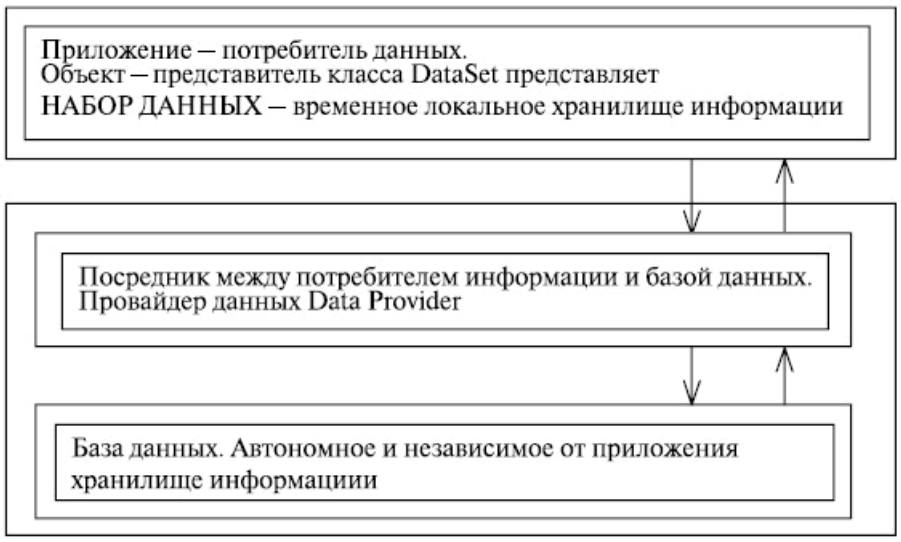
Предполагается, что к моменту написания приложения соответствующая база данных уже создана.

Объектная модель ADO .NET реализует отсоединенный доступ к данным. При этом в Visual Studio .NET существует множество ВСТРОЕННЫХ мастеров и дизайнеров, которые позволяют реализовать механизмы доступа к БД еще на этапе разработки программного кода.

С другой стороны, задача получения доступа к данным может быть решена непосредственно во время выполнения приложения.

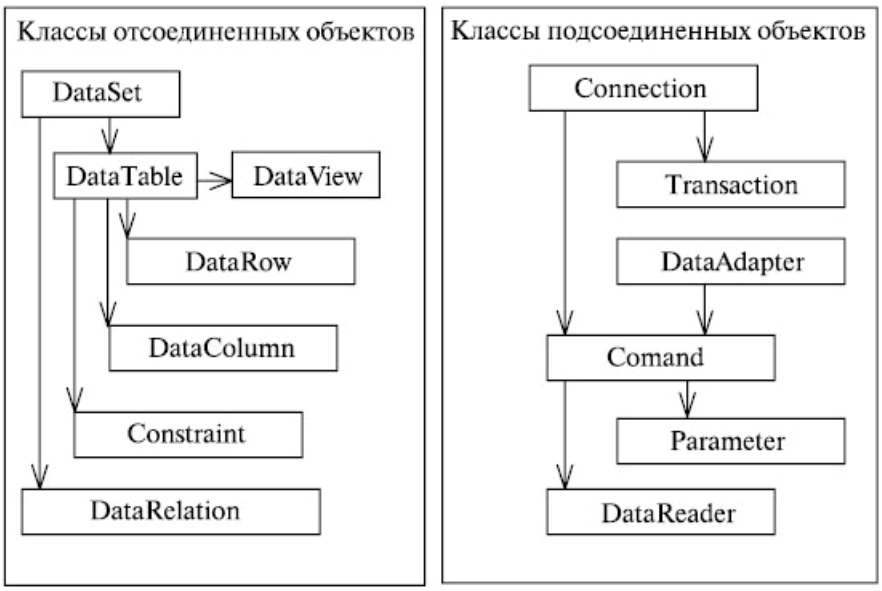
Концепция доступа к данным в ADO .NET основана на использовании двух компонентов:

- НАБОРА ДАННЫХ (представляется объектом класса `DataSet`) со стороны клиента. Это локальное временное хранилище данных;
- ПРОВАЙДЕРА ДАННЫХ (представляется объектом класса `DataProvider`). Это посредник, обеспечивающий взаимодействие приложения и базы данных со стороны базы данных (в распределенных приложениях – со стороны сервера).



ADO .NET. Объектная модель

Объектная модель ADO .NET предполагает существование (при написании приложения для работы с базой данных — использование) двух множеств классов, выполняющих четко определенные задачи при работе с базой данных:



Классы подсоединенных объектов обеспечивают установление соединения с базой данных и управление базой со стороны приложения; классы отсоединенных объектов обеспечивают сохранение, использование и преобразование полученной от базы данных информации на стороне приложения.

Далее рассматриваются классы отсоединенных объектов объектной модели ADO .NET. Их подробному описанию посвящаются следующие разделы пособия. При этом классы отсоединенных объектов могут быть самостоятельно использованы в приложении наряду с обычными компонентами и элементами управления, даже если в приложении и не предполагается организовывать работу с базами данных.

DataTable

Каждый объект `DataTable` представляет одну таблицу базы данных. 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 перехватывать и в случае необходимости отменять изменения состояния таблицы данных.

1. Изменения строк.
   - `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 )
     ```

   Пример программного кода для объекта – представителя класса `DataTable`:

```
using System;
using System.Data;

namespace DataRowsApplication00
{
class DataTester
{
DataTable custTable;

public void DTBuild()
{
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);
}

public void RowsAdd(int id)
{
int x;
// Добавляем строки.
for (x = 0; x < id; x++)
{
    custTable.Rows.Add(new object[] { x, string.Format("customer{0}", x) });
}
// Фиксируются все изменения, которые были произведены над таблицей
// со времени последнего вызова AcceptChanges.
custTable.AcceptChanges();
}

public void RowsChange()
{
// Изменяем значение поля name во всех строках.
// Все имена убираются, а на их место
// подставляется буквосочетание, состоящее из
// префикса vip и старого значения строки каждого клиента.
// Была строка customer5 - стала vip customer5.

foreach (DataRow row in custTable.Rows)
{
    row["name"] = string.Format("vip {0}", row["id"]);
}

}

// И после 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);
}
}
class Program
{
 static void Main(string[] args)
 {
  DataTester dt = new DataTester();
   dt.DTBuild();
   dt.RowsAdd(10);
   dt.RowsChange();
  }
 }
}
```

Листинг 18.1.

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`	`Get`. Объект-представитель класса `DataColumn` с изменённым значением
`ProposedValue`	`Gets, sets`. Новое значение для поля в строке
`Row`	Строка, содержащая запись с изменяемым (измененным) значением

Аналогичный пример. Только теперь программируется реакция на модификацию столбца (поля), а не строки:

```
using System;
using System.Data;

namespace DataColumnsApplication00
{
class DataTester
{
DataTable custTable;

public void DTBuild()
{
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"] };

// Добавление события ColumnChanging handler для таблицы.
custTable.ColumnChanging +=
             new DataColumnChangeEventHandler(Column_Changing);
// Добавление события ColumnChanged event handler для таблицы.
custTable.ColumnChanged +=
            new DataColumnChangeEventHandler(Column_Changed);
}

public void RowsAdd(int id)
{
int x;
// Добавляем строки.
for (x = 0; x < id; x++)
{
    custTable.Rows.Add(new object[] { x, string.Format("customer{0}", x) });
}
// Фиксируются все изменения, которые были произведены над таблицей
// со времени последнего вызова AcceptChanges.
custTable.AcceptChanges();
}

public void RowsChange()
{
// Изменяем значение поля name во всех строках.
foreach (DataRow row in custTable.Rows)
{
    row["name"] = string.Format("vip{0}", row["id"]);
}
}

// И после вмешательства результаты становятся известны
// обработчику события Column_Changing. А толку-то...
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);
}

// Аналогично устроен обработчик Column_Changed.
    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);
}
}
class Program
{
 static void Main(string[] args)
 {
```

```
          DataTester dt = new DataTester();
           dt.DTBuild();
           dt.RowsAdd(10);
           dt.RowsChange();
          }
         }
        }
```

Листинг 18.2.

DataColumns

`DataColumnCollection` задает 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.

Объект `DataColumn` представляет тип колонки в `DataTable`. Это стандартный блок, предназначенный для построения схемы `DataTable`.

Каждый объект `DataColumn` как элемент схемы характеризуется собственным типом, определяющим тип значений, которые `DataColumn` содержит.

Если объект `DataTable` создается как отсоединенное хранилище информации, представляющее таблицу базы данных, тип столбца объекта-таблицы должен соответствовать типу столбца таблицы в базе данных.

DataRows

СОДЕРЖИМОЕ таблицы (непосредственно данные) задается набором `DataRows` – это конкретное множество строчек таблицы, каждая из которых является объектом – представителем класса `DataRow`.

Его методы и свойства представлены в таблице.

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 объект `DataTable`, содержащий данную строку

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 хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-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, сделанных за текущий сеанс редактирования базы.

Посредством набора Rows реализуется возможность ссылки на любую запись таблицы. К любой записи можно обратиться напрямую, и поэтому 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 ее 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() еще не вызывался
Deleted	Строка была удалена из таблицы, но метод AcceptChanges() еще не вызывался
Modified	Некоторые из полей строки были изменены, но метод AcceptChanges() еще не вызывался
Detached	Строка НЕ ЯВЛЯЕТСЯ ЭЛЕМЕНТОМ КОЛЛЕКЦИИ DataRows. Ее создали от имени таблицы, но не подключили

Пример. Создание таблицы, работа с записями

```
using System;
using System.Data;

namespace Rolls01
{

// Работа с таблицей:
// определение структуры таблицы,
// сборка записи (строки таблицы),
// добавление новой записи в таблицу,
// индексация записей,
// выбор значения поля в строке,
// изменение записи.

public class RollsData
{
       public DataTable rolls;
       int rollIndex;
```

```
public RollsData()
{
rollIndex = 0;
// Создается 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 создается 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;
        // Созданная и заполненная строка
        // подсоединяется к таблице.
         rolls.Rows.Add(dr);
   }

   // Выборка 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. Скорость 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;
        }

  }
  }
```

Листинг 18.3.

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 объекте `DataSet` реализуется полный набор элементов для управления данными, включая сами таблицы, ограничения и отношения между таблицами.

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 и 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`.

Используя этот объект, можно представлять содержащиеся в таблице данные в соответствии с тем или иным порядком сортировки, а также организовать различные варианты фильтрации данных.

`DataView` предоставляет динамический взгляд на содержимое таблицы в зависимости от установленного в таблице порядка представления и вносимых в таблицы изменений.

Функционально реализация `DataView` отличается от метода `Select`, определенного 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 по 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 создается (независимый!) вьюер, который связывается с таблицей.

Итак, имеем

```
DataTable tbl = new DataTable("XXX"); // Объявлен и определен 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 также осуществляется посредством 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();
}
```

Листинг 18.4.

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();
}
}
```

Листинг 18.5.

В примере демонстрируется взаимодействие автономной таблицы и "заточенного" под нее вьюера. Записи в таблицу можно добавлять как непосредственно, так и через вьюер. При этом необходимо совершать некоторое количество дополнительных действий. Через вьюер же организуется поиск записей.

```
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;   // Счетчик 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

// Required method for Designer support - do not modify
// the contents of this method with the code editor.

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()
{

// Создается 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 задается множество первичных ключей.
keyCol[0]= c1;
// Свойству объекта t передается 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 проводится сортировка,
//        |   |
//        |   | имя столбца, значения которого сортируются,
//        |   |   |
//        |   |   |  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 последовательного автоматического
// обновления таблицы. Все 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 предыдущей 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();
}
}
}
```

Листинг 18.6.

DataSet

В рамках отсоединенной модели ADO .NET объект `DataSet` становится важным элементом технологии отсоединенного доступа. Объект-представитель `DataSet` ПРЕДСТАВЛЯЕТ МНОЖЕСТВО ТАБЛИЦ.

Для успешного решения задачи представления в DataSet'е есть все необходимое. Его функциональные возможности позволяют загрузить в локальное хранилище на стороне приложения данные из любого допустимого для ADO .NET источника: SQL Server, Microsoft Access, XML-файл.

В числе данных – членов этого класса имеется набор Tables. 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. DataSet – это находящийся в памяти объект ADO .NET, используемый в приложении для представления данных; он определяет согласованную реляционную модель базы данных, которая не зависит от источника содержащихся в нем данных. Степень полноты модели определяется задачами, которые решает приложение.

Объект DataSet может представлять абсолютно точную модель базы данных, и в таком случае эта модель должна будет включать полный набор структурных элементов базы данных, включая таблицы, содержащие данные, с учетом установленных ограничений и отношений между таблицами.

Содержащуюся в объекте DataSet информацию можно изменять независимо от источника данных (от самой БД). Соответствующие значения формируются непосредственно в программе и добавляются в таблицы.

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 отсоединенных объектов – хранилищ информации.

Класс DataSet – класс не абстрактный и не интерфейс. Это значит, что существует множество вариантов построения отсоединенных хранилищ.

На основе базового класса DataSet можно определять производные классы определенной конфигурации, которая соответствует структуре базы данных.

Можно также создать объект – представитель класса DataSet оригинальной конфигурации и добавить непосредственно к этому объекту все необходимые составляющие в виде таблиц (объектов – представителей класса Table) соответствующей структуры и множества отношений Relation.

Объект – представитель класса DataSet и сам по себе, без сопутствующего окружения, представляет определенную 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 ее хранения!). Не случайно DataSet представляет ОТСОЕДИНЕННЫЕ данные.

На DataSet работают все ранее перечисленные компоненты ADO .NET.

В свою очередь, в приложении, обеспечивающем взаимодействие с базой данных, объект DataSet функционирует исключительно за счет объекта DataAdapter, который обслуживает DataSet.

При этом DataAdapter является центральным компонентом архитектуры отсоединенного доступа.

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. Все определяется решаемой с помощью данного DataSet задачей.

При всем 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 базы данных, является достаточно сложным и трудоемким. 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 ускорять процесс воспроизведения структуры БД средствами языка программирования. В конечном счете появляется все то же объявление класса, над которым также можно медитировать и (осторожно!) изменять его структуру.

Объект – представитель данного класса DataSet обеспечивает представление в памяти компьютера 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 хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-таблице
GetService (унаследовано от MarshalByValueComponent)	Получает реализацию объекта IServiceProvider
GetType (унаследовано от Object)	Возвращает Type текущего экземпляра
GetXml	Возвращает XML-представления 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 полете

```
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 определенной таблицы
// метод NewRow() порождает строку
// (последовательность значений соответствующего типа).
// Для непосредственного
// редактирования вновь созданной строки запоминается ее ссылка.
// Работать со строкой через эту ссылку проще, чем с массивом
// строк таблицы.
DataRow myRow = myDataSet.Tables[0].NewRow();

// ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

// Остается 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 Rowскласса DataTable.
myDataSet.Tables[0].Rows.Add(myRow);
```

Листинг 18.7.

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)
{
// Создается объект DataSet.
DataSet ds = new DataSet("myDataSet");
```

```
// Создается 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 задается множество первичных ключей.
keyCol[0]= c1;
// Свойству объекта t передается 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 еще 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 чем проблема,
// описание возможной ошибки (в свободной форме!) 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. Все 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 все, 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 измененные 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();
}
}
}
}
}
```

Листинг 18.8.

Подсоединенные объекты 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 передает 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 для 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 составных 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) создается 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";
```

Листинг 18.9.

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`	`Get`. Возвращает объект `IContainer`, который содержит объект `Component`
`Database`	`string`	`Gets` текущей базы данных или базы, которая использовалась после установления соединения

DataSource	string	Get. Имя сервера или имя файла-источника данных. Все зависит от того, с каким хранилищем информации ведется работа. Серверное хранилище данных (SQL Server, Oracle) – имя компьютера, 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	Get. Строка с информацией о версии сервера, с которым было установлено соединение
Site	string	Get или set. Объект ISite с информацией о функциональных возможностях узла
State	string	Gets текущее состояние соединения

Текущее состояние соединения кодируется как элемент перечисления ConnestionState. Список возможных значений представлен ниже.

Имя члена	Описание	Value
Broken	Соединение с источником данных разорвано.Подобное может случиться только после того, как соединение было установлено. В этом случае соединение может быть либо закрыто, либо повторно открыто	16
Closed	Соединение закрыто	0
Connecting	Идет процесс подключения (значение зарезервировано)	2
Executing	Соединение находится в процессе выполнения команды (значение зарезервировано.)	4
Fetching	Объект соединения занят выборкой данных (значение зарезервировано)	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 конструирования прокси-сервера, используемого для коммуникации с удаленными объектами
Dispose (унаследовано от Component)	Перегружен. Освобождает ресурсы, используемые объектом Component
EnlistDistributed	Зачисляет в указанную транзакцию в качестве
Transaction	распределенной транзакции
Equals (унаследовано от Object)	Перегружен. Определяет, равны ли два экземпляра Object
GetHashCode (унаследовано от Object)	Служит хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-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
ReleaseObjectPoolOleDb Connection	Статический. Означает, что пул объектов 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	Это событие сопровождает процесс удаления объекта
InfoMessage	Некоторые СУБД (SQL Server) поддерживают механизм информационных сообщений. Это событие происходит при отправке провайдером некоторых сообщений
StateChange	Возникает при изменении состояния соединения

Подключение к БД на этапе разработки приложения

Для продолжения экспериментов воспользуемся базой-примером "Северные ветры".

Для внешнего окружения – это всего лишь файл с расширением .mdb.

В нашем случае –

```
E:\Program Files\Microsoft Office\Office10\Samples\Борей.mdb.
```

Ближайшей задачей будет разработка простого Windows-приложения, обеспечивающего просмотр содержащейся в базе информации о клиентах и их заказах.

Внешний вид формы приложения будет представлен несколькими текстовыми полями, в которых будет отображаться информация о клиенте, а также элементом `DataGrid`, где будет отображаться информация о связанных с клиентом заказах.

Один из возможных вариантов создания соединения средствами встроенных мастеров предполагает следующую последовательность шагов в рамках Visual Studio .NET 2005. Непосредственно перед началом работ по созданию приложения для наглядности рекомендуется держать открытым окно с информацией о базе данных (пункты меню Data.Show Data Sources):

- для нового приложения выполняются действия, связанные с созданием объекта – представителя класса `DataSet` (`Data, Add New Data Source`). При этом осуществляются действия по установлению и тестированию соединения с базой данных, требуется ответить на вопрос по поводу возможности копирования информации из базы в директорию приложения (речь идет о локальной копии базы);
- с использованием инструмента "Add Connection" объявляется тип источника данных (Microsoft Access Database File (OLE DB)) и определяется имя файла базы данных (файл "Борей.mdb"). Волшебник предоставляет возможность непосредственного тестирования устанавливаемого соединения;
- в результате создается объект – представитель класса `DataSet`, построенный и настроенный применительно к данному приложению для работы с базой данных. О сложности этой конструкции можно судить по объему программного кода, подсоединяемого к проекту;
- для исследования и редактирования его свойств `DataSet` предусмотрено средство `DataSet Designer`;
- свидетельством успешного установления соединения является возможность выполнения действия "Edit DataSet with designer", в результате которого в окошке NorthWinds.xsd визуализируется полная схема базы данных, включая таблицы и отношения между ними.

При этом в кодах приложения размещается соответствующий программный код, который обеспечивает создание объекта соединения соответствующего типа и с определенными параметрами.

Объект – представитель класса `DataSet` можно расположить на форме в виде компоненты. Объявление класса (включая строку соединения) можно будет попытаться проанализировать (файл имеет объем около 9000 строк), открыв файл БорейDataSet.Designer.sc.

Таким образом, процедура установления соединения с базой данных на этапе создания приложения целиком обеспечивается средствами "волшебников" в рамках работы по созданию приложения.

Продолжение разработки. Простые шаги

Продолжение работы по созданию простого приложения для работы с базой данных также не предусматривает непосредственной работы с программным кодом.

После создания объекта `DataSet` и трансляции кода приложения на панели инструментов появляется новая вкладка, предоставляющая возможность работы с автоматически объявленными классами – адаптерами таблиц. После чего работа по созданию приложения сводится к нескольким достаточно простым "волшебным" манипуляциям:

- создается объект – представитель класса `BindingSource`, свойству `DataSource` которого присваивается ссылка на ранее созданный объект `DataSet`, а свойству `DataMember` – значение, связанное с определенной в базе данных таблицей "Клиенты";
- это действие сопровождается созданием объекта – адаптера таблицы, условное обозначение которого появляется на панели компонентов формы, что делает адаптер доступным для возможной модификации и настройки;
- получение информации из базы данных обеспечивается при помощи запросов к базе, которые также достаточно просто построить, запустив соответствующий "волшебник". Мышиный клик по пиктограмме адаптера на панели, вызов генератора запроса, далее – в соответствии с замыслом приложения и сценарием генератора. После создания запроса на панели компонентов формы появляется пиктограмма, обозначающая ранее построенный объект-представитель класса `DataSet`, а непосредственно на форме – инструментальная панель с элементом, который обеспечивает выполнение запроса, в результате которого через соответствующий адаптер таблицы производится заполнение объекта `DataSet`'а;
- для решения поставленной задачи необходимо дважды запустить генератор запросов для заполнения таблиц "клиенты" и "заказы". Генератор запускается "от существующего" адаптера таблицы. Процесс создания второго запроса сопровождается появлением второго адаптера таблицы. При этом на форме появляются две инструментальные панели, обеспечивающие загрузку информации при выполнении приложения;
- информация о клиентах и заказах размещается в элементах управления типа `TextBox` (о клиентах) и элементе управления `DataGrid` (о заказах). Эти элементы размещаются на поверхности формы с последующей их привязкой к элементам `DataSet`'а, при этом мастер создает объекты – представители класса `BindingSource`;
- для обеспечения навигации по данным используется комбинированный элемент управления `BindingNavigator`, который настраивается на один из объектов – представителей класса `BindingSource` (к моменту настройки навигатора таких объектов в приложении – два).

В результате получаем приложение, которое обеспечивает просмотр содержимого базы данных.

... и еще более простые шаги

На самом деле разработка простой формы для работы с базой данных требует еще меньше усилий. Можно совсем ничего не делать и получить готовую форму.

После создания объекта-представителя класса `DataSet` надо всего лишь "перетащить" на форму из окна Data Sources пиктограмму соответствующей таблицы базы данных. В случае с базой данных "Борей" – это пиктограмма таблицы "Клиенты". При этом автоматически к коду приложения добавляются соответствующие классы и элемент управления для навигации по таблице "Клиенты". Для визуализации множества записей, связанных с заказами клиентов, следует проделать следующие манипуляции:

- "Раскрыть" в окне Data Sources пиктограмму, обозначающую таблицу "Клиенты". При этом становятся видимыми пиктограммы, отображающие столбцы таблицы, и пиктограмма связанной с таблицей "Клиенты" таблицы "Заказы".
- На форму следует перетащить эту самую пиктограмму. В результате получаем форму с парой объектов `DataGrid`, в которых можно наблюдать согласованные множества записей. Для каждого клиента наблюдаем множество заказов.

Разработчику приложения остается упражняться в вариантах перетаскивания пиктограмм (перетаскивать можно пиктограммы отдельных полей таблицы) и медитировать над полученным кодом.

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 трудоемкая. 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 создает объект соединения, устанавливает соединение с базой, использует его и, не разрывая соединения с базой, передает 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 остается свободных соединений – создается и настраивается новый объект, который после употребления приложением также передается 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 завершении 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 для исследования схемы базы

Установив соединение с базой данных, можно исследовать ее схему. Эта возможность обеспечивается объявленным в классе соединения методом `GetOleDbSchemaTable`.

Управление процессом сбора информации о схеме базы обеспечивается двумя параметрами метода.

Первый параметр представлен статическими свойствами класса `OleDbSchemaGuid`.

Второй параметр – массивом (кортежем) объектов, каждый из которых в зависимости от занимаемой позиции в кортеже может принимать определенное 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. 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, который определен в каталоге и принадлежит 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, которые используются ссылочными ограничениями, уникальными ограничениями и утверждениями, определенными в каталоге и принадлежащими указанному пользователю
`Constraint_Table_Usage`	Статический. Возвращает таблицы, которые используются 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 хэш-функцией для конкретного типа, пригоден для использования в алгоритмах хэширования и структурах данных, например в хэш-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'
```

- запросы изменения или Data Definition Language queries. Предназначены для изменения структуры базы данных

```
CREATE TABLE myTable
                       (
                       Field1 int NOT NULL
                       Field2 varchar()
                       )
```

2. 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

Команда – объект, представляющий один из двух классов: либо класс `OleDbCommand`, либо класс `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 применен объект – представитель класса `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 Language) 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)`.

Объект представлен двумя классами – `SqlCommand` и `OleDb Command`. Позволяет исполнять команды на БД и при этом использует установленное соединение. Исполняемые команды могут быть представлены:

- хранимыми процедурами;
- командами SQL;
- операторами, возвращающими целые таблицы.

Объекта – 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 метода `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();
}
```

Листинг 18.10.

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 хранимых 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 на 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 создается 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 остается вручную задать свойства `Connection, CommandText, CommandType` (для определения типа команды, которая задается в свойстве `CommandText`). При этом свойство `CommandType` может принимать одно из трех значений: (1) `Text` – значение свойства `CommandText` воспринимается как текст команды SQL. При этом возможна последовательность допустимых операторов, разделенных 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 имен таблиц, возможно, состоящий из одного элемента. При выполнении команды возвращаются все строки и столбцы этих таблиц;
- размещение (путем перетаскивания) хранимой процедуры из окна Server Explorer в окно дизайнера. Объект "Команда" соответствующего типа при этом создается 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` – применяется 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 строки, независимо от общего количества выбранных строк.
- `ExecuteReader` – обеспечивает выбор строк из таблицы БД. Возвращает неизменяемый объект `DataReader`, который допускает последовательный однонаправленный просмотр извлеченных данных без использования объекта `DataAdapter`.

Кроме того, класс `SqlCommand` поддерживает еще один метод:

- `ExecuteXmlReader` – обеспечивает выбор строк из таблицы БД в формате XML. Возвращает неизменяемый объект `XMLReader`, который допускает последовательный однонаправленный просмотр извлеченных данных.

Назначение всех четырех методов – ИСПОЛНЕНИЕ НА ИСТОЧНИКЕ ДАННЫХ команды, представленной объектом команды.

Parameter

Данные из базы выбираются в результате 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 определенной 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 путем 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 путем присваивания свойству `Parameters` списка (вполне возможно, 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 чем 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 возможных длин строкового и двоичного параметров).

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 счет 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` или `InputOutput` (если параметр используется в процедуре как для получения, так и для отправления значений). Очевидно, что тип параметра должен соответствовать ожидаемому возвращаемому значению.

После выполнения процедуры можно прочитать значение возвращаемого параметра.

- Непосредственный перехват возвращаемого значения сохраняемой процедурой.

Для этого следует создать параметр с `Direction`-свойством, установленным в `ReturnValue`. Такой параметр должен быть первым в списке параметров.

При этом тип параметра должен соответствовать ожидаемому возвращаемому значению.

Предложения SQL – `Update, Insert, and Delete` возвращают целочисленное значение, соответствующее количеству записей, на которые повлияло выполнение данного предложения.

Это значение может быть получено как возвращаемое значение метода `ExecuteNonQuery`.

Следующий пример демонстрирует, как получить возвращаемое значение, возвращаемое хранимой процедурой `CountAuthors`. В этом случае предполагается, что первый параметр списка параметров конфигурируется как возвращаемый:

```
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 подключенный 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 создать нельзя. Этим DataReader отличается от других компонент провайдера данных.

Объект DataReader создается в результате обращения к одному из вариантов метода ExecuteReader объекта Command (SqlCommand.Execute Reader возвращает ссылку на SqlDataReader, OleDbCommand.Execute Reader 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. Остается 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 поэтому 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 ОТСОЕДИНЕННЫМ данным" – заслуга объекта DataReader. Дело в том, что получение данных приложением из базы данных все 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, образовавшегося 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. Все 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 отсоединенный 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 объекта OleDbCommand под именем myOleDb Command. 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 ее в переменную Customer
Customer = myDataReader.GetString(CustomerID);
```

DataAdapter

`DataAdapter` – составная часть провайдера данных. То есть подсоединенная 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 счет:

- 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`:

- с помощью окна Server Explorer;
- 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.

Транзакция

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 модели 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:

- Инициализация 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 изменение состояния источника данных. В противном случае (блок catch), от имени объекта – представителя класса `Transaction` вызывается метод `Rollback()`, который отменяет ранее произведенные 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();
 // Создается объект транзакции.
 xTransaction = xConnection.BeginTransaction();
 // Транзакция фиксируется в командах.
 xCommand1.Transaction = xTransaction;
 xCommand2.Transaction = xTransaction;
 // Выполнение команд.
 xCommand1.ExecuteNonQuery();
 xCommand2.ExecuteNonQuery();
 // Если ВСЕ ХОРОШО и мы все еще здесь – ПРИНЯТЬ ТРАНЗАКЦИЮ!
 xTransaction.Commit();
}
catch
{
 // Если возникли осложнения – отменяем транзакцию.
 xTransaction. Rollback();
}
finally
{
 // В любом случае соединение закрывается.
 xConnection.Close();
}
```

Введение в программирование на C# 2.0

1. Лекция: Программа. Сборка. Класс: версия для печати и PDA
В данной лекции рассматриваются основные понятия языка, а также принципы работы Microsoft .NET

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

В общем случае, программист создает файл, содержащий объявления классов, который подается на вход компилятору. Результат компиляции представляется транслятором в виде сборки. В принципе сборка может быть двух видов (здесь все зависит от замысла разработчика кода):

- 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.

Пространство имен

.NET Framework располагает большим набором полезных функций. Каждая из них является членом какого-либо класса. Классы группируются по пространствам имен. Это означает, что в общем случае имя класса может иметь сложную структуру — состоять из последовательности имен, разделенных между собой точками. Последнее имя в этой последовательности собственно и является именем класса. Классы, имена которых различаются лишь последними членами (собственно именами классов) последовательностей, считаются принадлежащими одному пространству имен.

Средством "навигации" по пространствам имен, а точнее, средством, которое позволяет сокращать имена классов, является оператор

```
using <ИмяПространстваИмен>;
```

В приложении может объявляться собственное пространство имен, а также могут использоваться ранее объявленные пространства.

В процессе построения сборки транслятор должен знать расположение сборок с заявленными для использования пространствами имен. Расположение части сборок известно изначально. Расположение всех остальных требуемых сборок указывается явно (непосредственно в Visual Studio при работе над проектом открыть окно Solution Explorer, выбрать пункт References, далее Add Reference... – там надо задать или выбрать соответствующий .DLL- или .EXE-файл).

В частности, сборка, которая содержит классы, сгруппированные в пространстве имен System, располагается в файле mscorlib.dll.

Наиболее часто используемое пространство имен – `System`. Расположение соответствующей сборки известно. Если не использовать оператор

```
using System;
```

корректное обращение к функции `WriteLine(...)` – члену класса `Console` выглядело бы следующим образом:

```
System.Console.WriteLine("Ha-Ha-Ha!"); // 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 на его основе) находит ее код, который и используется при выполнении сборки.

Класс и Структура. Первое приближение

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.

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 объявлений и конструкций, облегчающих процедуру объявления.

С точки зрения синтаксиса, между объявлениями классов и структур существуют незначительные различия (ключевые слова `struct` и `class`, в структуре не допускается объявлений членов класса со спецификаторами доступа `protected` и `protected internal`, при объявлении структуры не допускается объявление конструктора без параметров), часть из которых будет рассмотрены далее.

Основное их различие состоит в том, что класс и структура принадлежат к двум различным категориям типов – типов-ссылок и типов-значений.

В этом разделе обсуждаются основные правила объявления классов.

Объявление класса состоит из нескольких элементов:

- Объявление атрибутов – необязательный элемент объявления.
- Модификаторы (в том числе модификаторы прав доступа) – необязательный элемент объявления.
- Partial (спецификатор разделения объявления класса) – необязательный элемент объявления.
- Class (struct для структуры).
- Имя класса.
- Имена предков (класса и интерфейсов) – необязательный элемент объявления.
- Тело класса (структуры).

Атрибуты – 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 и многое другое.

Модификаторы `new, abstract, sealed, static` обсуждаются дальше. Модификаторы прав доступа обеспечивают реализацию 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, которые доступны из методов классов, объявляемых в рамках сборки, где содержится объявление данного класса
`protected internal`	Обозначение для членов класса, доступных в рамках объявляемого класса, из методов производных классов, а также доступных из методов классов, которые объявлены в рамках сборки, содержащей объявление данного класса
`private`	Обозначение для членов класса, доступных в рамках объявляемого класса

Спецификатор разделения объявления класса partial позволяет разбивать код объявления класса на несколько частей, каждая из которых размещается в собственном файле. Если объявление класса занимает большое количество строк, его размещение по нескольким файлам может существенно облегчить работу над программным кодом, его документирование и модификацию. Транслятор способен восстановить полное объявление класса. Спецификатор partial может быть использован при объявлении классов, структур и интерфейсов.

Сочетание ключевого слова `class` (`struct, interface`) и имени объявляемого класса (структуры или интерфейса) задает имя типа.

Конструкции

```
:имя класса
(при объявлении класса)
:список имен интерфейсов
(при объявлении структуры или класса)
:имя класса, список имен интерфейсов
(при объявлении класса)
```

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

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

namespace qwe // Объявление собственного пространства имен. 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 имеет развитого механизма
наследования.
// 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 имен. Конец.
```

Введение в программирование на C# 2.0

Литература: версия для печати

Учебники к курсу

1. Марченко А. Л.
   Основы программирования на C# 2.0
   БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2007
2. Кариев Ч.А.
   Разработка Windows-приложений на основе Visual C#
   БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий - ИНТУИТ.ру, 2007

Список литературы

Внимание! Внешние ссылки могут не работать. Пожалуйста, ищите необходимую информацию в Сети (WWW).

1. **Библиотека MSDN на русском языке**
   http://msdn.microsoft.com/library/rus/
2. Жарков В
   Самоучитель Жаркова по анимации и мультипликации в Visual C# .NET
   Жарков Пресс, 2004. 432 с
3. Либерти Д
   Программирование на C#
   Символ-Плюс, 2003. 688 с
4. Майо Дж
   C#. Искусство программирования
   ДиаСофт, 2002. 656 с
5. Микелсен К
   Язык программирования C#. Лекции и упражнения
   ДиаСофт, 2002. 656 с
6. Петзольд Ч
   Программирование для Microsoft Windows на C#. В 2 томах
   М.: Русская редакция, 2002
7. Петзольд Ч
   Программирование в тональности C#
   М.: Русская редакция, 2004. 512 с
8. **Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# .NET. Учебный курс**
   СПб.: Русская Редакция. 512 с
9. Рихтер Д
   Программирование на платформе Microsoft .NET Framework
   СПб.: Русская редакция, 2005. 512 с
10. Робисон У
    C# без лишних слов
    ДМК, 2002. 352 с
11. Троелсен Э
    C# и платформа .NET. Библиотека программиста
    СПб., 2004. 796 с
12. Шилдт Г
    C#. Учебный курс
    СПб., 2002. 511 с
13. Шилдт Г
    Полный справочник по C#
    Вильямс, 2004, 752 с
14. Фролов А., Фролов Г
    Язык C#. Самоучитель
    М.: Диалог-МИФИ, 2002. 560 с

Введение в программирование на C# 2.0

2. Лекция: Система типов: версия для печати и PDA

Язык программирования предполагает наличие правил построения корректных предложений. В свою очередь, предложения строятся из выражений. Основной характеристикой выражения является значение этого выражения. Можно утверждать, что выполнение программы состоит из вычисления значений выражений, которые образуют предложения программы. Синтаксис языка не ограничивает сложности выражения. Выражение может состоять из очень большого количества более простых (элементарных) выражений.

Значение выражения характеризуется типом (типом выражения). Выражение сложной структуры может сочетать элементарные выражения различных типов. При вычислении значений таких выражений требуются дополнительные усилия по преобразованию значений одного типа в значения другого типа. Большая часть этой работы производится автоматически, в ряде случаев от программиста требуются дополнительные усилия по разработке алгоритмов преобразования.

При разработке кода любой сложности знание правил кодирования, вычисления и преобразования значений различных типов является условием создания правильно выполняемой программы.

В разделах этой главы обсуждаются система типов и основные характеристики категорий типов, литералы которые являются основным средством кодирования значений в выражениях, проблемы преобразования значений в выражениях, операторы объявления и связанная с ними проблема области действия имени.

Категории типов

Система типов включает несколько категорий типов:

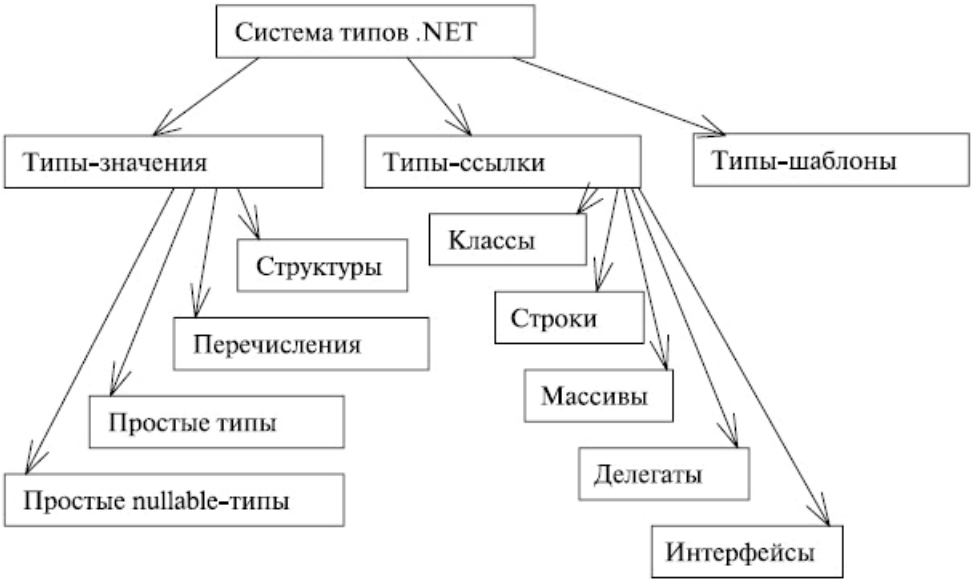
- типы значений (типы-значения),
- ссылочные типы (типы-ссылки),
- параметризованные типы (типы-шаблоны).

Схема типов представлена ниже.

Простые (элементарные) типы – это типы, имя и основные свойства которых известны компилятору. Относительно элементарных типов компилятору не требуется никакой дополнительной информации. Свойства и функциональность этих типов известны.

Среди простых типов различаются:

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



Некоторые характеристики простых (элементарных) типов отражены в следующей таблице. Используемые в .NET языки программирования основываются на общей системе типов. Между именами простых типов в C# и именами типов, объявленных в Framework Class Library, существует взаимно однозначное соответствие. Смысл точечной нотации в графе "Соответствует FCL-типу" состоит в явном указании пространства имен, содержащем объявление соответствующего типа:

`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

Ниже представлены основные отличия ссылочных типов и типов-значений.

	Типы-значения	Типы-ссылки
Объект представлен	непосредственно значением	ссылкой в стеке или куче
Объект располагается	в стеке или куче	в куче
Значение по умолчанию	0, false, '\0', null	ссылка имеет значение null
При выполнении операции присваивания копируется	значение	ссылка

В C# объявление любой структуры и класса основывается на объявлении предопределенного 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.

Все типы (типы-значения и типы-ссылки), за исключением простых типов-значений и пары предопределенных ссылочных типов (string и object), должны определяться (если уже не были ранее специально определены) программистами в рамках объявлений. Подлежащие объявлению типы называются производными типами.

В разных CLS-языках типам, удовлетворяющим CLS-спецификации, будут соответствовать одни и те же элементарные типы.

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.

Параметризованные типы занимают особое место в системе типов и обсуждаются позже.

object и string: предопределенные ссылочные типы

Итак, к ссылочным типам относятся:

- классы;
- интерфейсы;
- массивы;
- делегаты.

Для каждой категории ссылочных типов существуют собственные правила объявления. Объявления классов вводятся ключевым словом class. Правила объявления классов (как и правила объявления других ссылочных типов, а также типов-значений, объявляемых с помощью ключевого слова struct) позволяют объявлять неограниченное множество разнообразных ссылочных типов и структур.

Среди множества классов выделяют предопределенные ссылочные типы object и string, которым соответствуют FCL-типы System.Object и System.String. Свойства, функциональность и особенности применения этих типов рассматриваются ниже.

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 значением типа short;
- дополнительным суфиксом. Суфиксы 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

Это заключенные в одинарные кавычки вводимые с клавиатуры одиночные символы: 'X', 'p', 'Q', '7', а также 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 – какой символ будет здесь – не ясно.

Символьные Escape-последовательности

Следующие заключенные 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).

`\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, заключенных в двойные кавычки.

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

`Verbatim string` представляется при помощи символа `@`, который располагается непосредственно перед строковым литералом, заключенным 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 счет их дублирования:

```
...@"""Focus"""
```

имеет значение

```
"Focus"
```

Строковые литералы являются литералами типа `string`.

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 типа-значения в методе класса приводит к созданию в памяти объекта соответствующего типа, возможно, проинициализированного определенным значением. Это значение может быть задано в виде литерала соответствующего типа или 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) или как псевдоним типа (имя типа, как оно объявляется в C#). Соответствующее выражение инициализации может быть представлено литералом или выражением более сложной структуры.

Эквивалентные формы записи операторов определения переменных элементарных типов-значений:

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

Эквивалентные формы записи операторов определения и инициализации переменных типа значения:

```
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 b.
a=b+b; // Инициализация a.
```

Константы

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

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

Операции и выражения

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

Эти операции задают 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 определенного подмножества операций и имеет собственное графическое представление.

Общие характеристики используемых в C# операций представлены ниже.

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

На основе элементарных (первичных) выражений с использованием этих самых операций и дополнительных разделителей в виде открывающихся и закрывающихся скобочек формируются выражения все 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

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

Приведение типов – один из аспектов безопасности языка.

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 значение типа `sbyte, byte, short, ushort, int, uint, long, ulong`. При этом в программе все должно быть устроено таким образом, чтобы логика преобразования значений одного типа к другому типу была бы понятной, а результаты этих преобразований – предсказуемы. В одном выражении могут быть сгруппированы операнды различных типов. Однако возможность подобного "смешения" при определении значения выражения приводит к необходимости применения дополнительных усилий по приведению значений операндов к "общему типу".

Иногда приведение значения к другому типу происходит автоматически. Такие преобразования называются неявными.

Но в ряде случаев преобразование требует дополнительного внимания со стороны программиста, который должен явным образом указывать необходимость преобразования, используя выражения приведения типа или обращаясь к специальным методам преобразования, определенным в классе `System.Convert`, которые обеспечивают преобразование значения одного типа к значению другого (в том числе значения строкового типа к значениям базовых типов).

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.

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.

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` — как результат деления отрицательного числа на ноль.

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

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

object. Характеристики типа

Всеобщий базовый тип. Обязательная составляющая любого типа в .NET. Функциональные характеристики типа `System.Object` приводятся в таблице.

Конструктор

`Object`	Создает и инициализирует объект типа `Object`

Общедоступные (public) методы

`Equals`	Обеспечивает сравнение объектов
`GetHashCode`	Обеспечивает реализацию алгоритма хэширования для значений объектов
`GetType`	Для любого объекта создает объект типа Type, содержащий информацию о структуре типа данного объекта
`ReferenceEquals`	Проверка эквивалентности ссылок. Статический
`ToString`	Возвращает объект типа `String` с описанием данного объекта

Защищенные (protected) методы

Имя	Описание
`Finalize`	Реализует процедуру уничтожения объекта.
`MemberwiseClone`	Создает копию текущего объекта.

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);
}
}
```

Листинг 2.1.

В результате выполнения программы выводится такой результат:

```
The result of action (1000000*1000000/1000000) is -727
>>>>> 12345678901234600000,0000000000
>>>>> 12345680000000000000,0000000000
No! 0,400000005960464
No! 1,00000001490116
No! 1,00000001490116
Yes! 0,0000000, 0,0000100
```

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`) дает `NaN` (Not a Number).

checked и unchecked. Контроль за переполнением

Причиной некорректных результатов выполнения арифметических операций является особенность представления значений арифметических типов.

Арифметические типы имеют ограниченные размеры. Поэтому любая арифметическая операция может привести к переполнению. По умолчанию в C# переполнение, возникающее при выполнении операций, никак не контролируется. Возможный неверный результат вычисления остается всего лишь результатом выполнения операции, и никого не касается, КАК эта операция выполнялась.

Механизм контроля за переполнением, возникающим при выполнении арифметических операций, обеспечивается ключевыми словами `checked` (включить контроль за переполнением) и `unchecked` (отключить контроль за переполнением), которые используются в составе выражений. Конструкции управления контролем за переполнением имеют две формы:

- операторную, которая обеспечивает контроль над выполнением одного выражения:

```
:::::
short x = 32767;
short y = 32767;
short z = 0;

try
{
 z = checked(x + unchecked(x+y));
}
catch (System.OverflowException e)
{
Console.Writeline("Переполнение при выполнении сложения");
}
return z;
:::::
```

  При этом контролируемое выражение может быть произвольной формы и сложности и может содержать другие вхождения как контролируемых, так и неконтролируемых выражений;

- блочную, которая обеспечивает контроль над выполнением операций в блоке операторов:

```
:::::
short x = 32767;
short y = 32767;
short z = 0, w = 0;

try
{
unchecked
{
 w = x+y;
}

checked
{
 z = x+w;
}

}
catch (System.OverflowException e)
{
 Console.Writeline("Переполнение при выполнении сложения");
}

return z;
:::::
```

Естественно, контролируемые блоки при этом также могут быть произвольной сложности.

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, определенных 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.

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 нулем. 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 хэш-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
`GetTypeCodeType`	Возвращает базовый тип `TypeCode` для этого экземпляра
`GetUnderlying`	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 имен. 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 имен (проблемы с обращением к одноименным объектам) позволяет избежать принятая в языке дисциплина именования. В каждом языке программирования она своя.

Избежать конфликта имен в C# позволяет такая синтаксическая конструкция, как блок операторов. Блок – это множество предложений (возможно пустое), заключенное 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 ее создания. Обращение к объявляемой сущности (переменной или константе) "выше" точки ее 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. Одноименные объекты во вложенных областях конфликтуют.

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

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

В теле класса не допускается объявления одноименных функций — членов (методов класса) с пустыми списками параметров. Также не допускаются объявления одноименных функций — членов (методов класса) со списками параметров, у которых совпадают типы параметров.

В списках параметров имена параметров не важны. 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;
}
}
```

Листинг 2.2.

Введение в программирование на C# 2.0

3. Лекция: Управляющие операторы и методы: версия для печати и PDA
В языке программирования C# существуют специальные операторы, которые в зависимости от вычисляемых значений выражений позволяют управлять ходом выполнения программы, эта лекция рассказывает именно о них

Основным средством реализации функциональности класса являются методы. Методы строятся из предложений-операторов, в ходе выполнения которых производятся вычисления составляющих операторы выражений. В языке программирования C# существуют специальные операторы, которые в зависимости от вычисляемых значений выражений позволяют управлять ходом выполнения программы.

В этой главе обсуждаются управляющие операторы и проблемы, связанные с определением и вызовом методов.

Управляющие операторы

Управляющие операторы применяются в рамках методов. Они определяют последовательность выполнения операторов в программе и являются основным средством реализации алгоритмов.

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 еще один оператор.

В силу того, что в C# отсутствуют предопределенные алгоритмы преобразования значений к булевскому типу, условное выражение должно быть выражением типа `bool` – переменной, константой или 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 оператор `if ... else ...` включает ДВЕ части.

```
if (...)
 {
if (...)
 {

 }
else
```

```
  {
  }

  }
 else
  {

  }
```

переписывается как

```
 if (...) if (...)
  {

  }
 else
  {

  }
 else
  {

  }
```

Главное – не перепутать соответствующие части оператора `if ... else ...` .

Еще 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. Одиночный оператор в C# — 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 определена 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 ::=
        СписокРазделенныхМеток [СписокОператоров] ОператорТерминатор
 СписокРазделенныхМеток ::= [СписокРазделенныхМеток] 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; // default – равноправный элемент switch-оператора.
 // может располагаться в любом месте switch-блока, поскольку
```

```
// НЕ БЛОКИРУЕТ (!!!) выполнение операторов всех нижележащих CaseЭлементов.
case 100: ... break;
}
::::::::::
switch (val)
{
default: // default НЕ БЛОКИРУЕТ выполнение операторов
// всех нижележащих CaseЭлементов.
// Операторы, входящие в CaseЭлемент под меткой 100, выполняются
// ПРИ ЛЮБЫХ значениях переменной val, отличных от 1 и 10.
case 100: ... break;
case 1: ... break;
case 10: ... break;
}
```

Подтверждение равноправия `CaseЭлемента` с меткой `default`. При выполнении менять значение `switchВыражения`. Осознать происходящее:

```
using System;


namespace OperatorsTest
{
class Program
{
static void Main(string[] args)
{
int i = 1; // = 500; = 125; = 100; = 250;
switch (i)
{
case 15:
default:
case 100:
Console.WriteLine("default: Ha-Ha-Ha");
break;
case 1:
case 2:
Console.WriteLine("1: 2: Ha-Ha-Ha");
break;
case 125:
Console.WriteLine("125: Ha-Ha-Ha! Ha-Ha-Ha");
break;
case 250:
Console.WriteLine("250: Ha-Ha-Ha! Ha-Ha-Ha");
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 разделенных 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; // Терминатор.
case 1:
XXX += 125;
break; // Терминатор.
```

```
default:
return XXX; // Терминатор.
}
```

while

```
ОператорWHILE ::= while (УсловиеПродолжения) Оператор
УсловиеПродолжения ::= БулевоВыражение
```

Здесь

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

Правило выполнения этого итеративного оператора 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 (еще 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

```
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 разделенную запятыми последовательность выражений.

Следует иметь в виду, что оператор объявления также строится на основе списка выражений (выражений объявления), состоящих из спецификаторов типа, имен 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 имен 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 передается следующему за `Оператором FOREACH` оператору (разумеется, если таковые имеются).

Область видимости имен 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.

Обычно об этом операторе говорится много нехороших слов как о главном разрушителе структурированного программного кода, и его описание сопровождается рекомендациями к его НЕИСПОЛЬЗОВАНИЮ.

Операторы

```
break;
```

и

```
continue;
```

применяются 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).

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

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

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[]
```

Листинг 3.1.

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

Класс, объявление которого содержит только объявления статических методов, называется статическим классом и может объявляться со спецификатором `static`:

```
public static class XXX
{
```

```
static int f1(int x)
 {
  return 1;
 }

static int f2(int x)
 {
 return 2;
 }
}
```

Вызов метода

Выражение вызова метода — это всего лишь выражение, составная часть оператора (предложения C#). Оно располагается в блоке операторов — в теле метода. Выражению вызова метода может предшествовать выражение, определяющее принадлежность метода классу или объекту.

Статические методы определяют функциональность класса. Для статических методов, принадлежащих другому классу, этим выражением может быть имя класса, содержащего объявление данного метода. Таким образом, статические методы вызываются от имени класса, в котором они были объявлены.

Нестатические методы определяют поведение конкретных объектов-представителей класса и потому вызываются "от имени" объекта-представителя класса, содержащего объявление вызываемого метода. Указание на конкретный объект — это всего лишь выражение, обеспечивающее ссылку на данный объект. Таковым действительно может быть имя объекта, выражение `this` (которое вообще может быть опущено), выражение индексации и другие выражения ссылочного типа.

Выражение вызова метода можно считать точкой вызова метода. В точке вызова управление передается вызываемому методу. После выполнения последнего оператора в теле вызываемого метода управление возвращается к точке вызова. Если метод возвращает значения, выражение вызова принимает соответствующее значение. В момент получения возвращаемого значения точка вызова становится точкой возврата.

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 все 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 узнает о результатах деятельности вызываемого 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 схема 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, и метод, в котором она произошла, реагирует на нее 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 написать 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 ее отличие.
public string xMessage;

// Кроме того, в поле ее 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 дополнительную
// специальную строчку для кодирования информации
// об исключении.
public string MyMessage;

// Кроме того, в поле ее 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 ВСЕ 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 ВСЕ ХОРОШО - будет выполнен оператор Console.WriteLine.
 // Потом операторы блока finally, затем операторы за
 // 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.
// Еще один вариант построения последнего рубежа
// выглядит так:
// catch
//{
//  Операторы обработки - сюда. Здесь в любом случае код,
//  реализующий алгоритм последнего перехвата.

    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!");
}
}
```

Листинг 3.2.

Работа с входным потоком. Предварительная информация

Информация, поступающая из входного потока, представляет собой символьные последовательности. Естественно, что область применения вводимой информации существенно ограничена. Арифметические вычисления требуют значений соответствующего (арифметического) типа.

Следующий пример демонстрирует способы преобразования поступающей через входной поток (ввод с клавиатуры) информации с целью получения значений арифметического типа:

```
using System;

namespace InOutTest00
{
public class InputTester
{

// Ввод арифметических значений типов int и float.
// Осуществляется в рамках бесконечного цикла, поскольку
// предполагает возможные ошибки при вводе.
// Выход из бесконечного цикла
// происходит только после прочтения из входного потока
// последовательности символов, которая может быть
// преобразована в значения типа int или float.
public int intIput()
{
string inputStr;
int val = 0;
for ( ; ; )
{
Console.Write("integer value, please >>> ");
// Строка символов читается с клавиатуры.
inputStr = Console.ReadLine();
try
{
// Попытка преобразования строки символов к типу int.
// В случае неудачи возникает исключение.
val = int.Parse(inputStr);
}
catch (Exception e)
{
// Перехват исключения.
// Сообщили об ошибке ввода.
Console.WriteLine("Input Error. {0} isn't integer", inputStr);
// Собственно информация об исключении.
Console.WriteLine("{0}", e.ToString());
// Повторяем попытку ввода символьной строки.
continue;
}
// Успешная попытка преобразования строки символов к типу int.
// Выход из цикла.
return val;
}

}

// Ввод значений типа float.
// Выход из бесконечного цикла
// происходит только после прочтения из входного потока
// последовательности символов, которая может быть
// преобразована в значение типа float.
// На правильность ввода влияют настройки языковых параметров,
// представления чисел, денежных единиц, времени и дат.


public float floatIput()
{
string inputStr;
float val;
for ( ; ; )
{
Console.Write("float value, please >>> ");
inputStr = Console.ReadLine();
try
{
// Приведение строки символов к значению типа float
// с использованием статических методов класса Convert.
val = System.Convert.ToSingle(inputStr);
}
catch (Exception e)
```

```
{
Console.WriteLine("Input Error!");
Console.WriteLine("{0}", e.ToString());
continue;
}
return val;
}
}
}

class Program
{
// Точка входа.
static void Main(string[] args)
{
// Создается объект – представитель класса InputTester.
InputTester it = new InputTester();
// Этот объект используется для корректного ВВОДА с клавиатуры:
// значений типа int...
int iVal = it.intIput();
// значений типа float...
float fVal = it.floatIput();
}
}
}
```

Листинг 3.3.

Перегрузка методов

Имя метода и список типов параметров метода являются его важной характеристикой и называются СИГНАТУРОЙ метода. В C# методы, объявляемые в классе, идентифицируются по сигнатуре. Эта особенность языка позволяет объявлять в классе множество одноименных методов. Такие методы называются перегруженными, а деятельность по их объявлению – перегрузкой.

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

```
// Класс содержит объявление четырех 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).

Передача по значению – БЕЗ спецификаторов (для типов-значений этот способ предполагается по умолчанию). Параметр представляет собой локальную копию значения в методе. В теле метода это означенная переменная, которую можно использовать

в методе наряду с переменными, непосредственно объявленными в этом методе. При этом изменение значения параметра НЕ влияет на значение параметра в выражении вызова.

Для организации передачи по ссылке параметра типа значения требуется явная спецификация `ref`. Для ссылочных типов передача параметра по ссылке предполагается по умолчанию. Спецификатор `ref` в этом случае не требуется, поскольку другого способа передачи параметра для ссылочных типов просто нет.

При передаче значения по ссылке также может использоваться спецификатор `out`. Этим спецификатором обозначают параметры, которым в методе присваиваются значения. Наличие в вызываемом методе выражения, обеспечивающего присвоение значения параметру со спецификатором `out`, обязательно.

Выражение вызова метода приобретает свое значение непосредственно по возвращении из вызываемого метода. Этим значением можно воспользоваться только в точке возврата.

Переданные методу "для означивания" параметры со спецификатором `out` сохраняют свои значения за точкой возврата.

Спецификация способа передачи параметра является основанием 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
  // обязательно 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 параметра типа значения спецификаторы out-ref
// неразличимы.
//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.

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

Листинг 3.4.

Ссылка и ссылка на ссылку как параметры

На самом деле ВСЕ параметры ВСЕГДА передаются по значению. Это означает, что в области активации создается копия ЗНАЧЕНИЯ параметра. При этом важно, ЧТО копируется в эту область. Для ссылочных типов возможны два варианта:

- можно копировать адрес объекта (ссылка как параметр),
- можно копировать 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
{

// Ссылка и ссылка на ссылку.
class WorkClass
{
public int x;

// Варианты конструкторов.
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

// Еще 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;
 }
```

```
 }
}
```

Листинг 3.5.

В результате выполнения программы в окошко консольного приложения выводится информация следующего содержания:

```
on start: 0
in f0: 1
after f0: 1
in f1: 2
after f1: 1
in f2: 10
after f2: 10
in f3: 3
after f3: 3
w0 != null
w1 == null
```

Сравнение значений ссылок

Существуют 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`);
- четвертого не дано. Отношение "больше-меньше" в условиях управляемой памяти не имеет никакого смысла.

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 на конкретные места хранения значений. Определение свойства в C# предполагает описание способов чтения и записи значений. Алгоритмы чтения и записи значений реализуются в виде блоков операторов, которые называются get accessor и set accessor.

Наличие accessor'ов определяет доступность свойства для чтения и записи. При обращении к значению свойства активизируется механизм чтения (управление передается в get accessor), при изменении значения активизируется механизм записи (управление передается в 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;
}
}
```

Листинг 3.6.

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 рекомендуется реализовывать как метод.

Main в классе. Точка входа

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

В сборке можно помещать несколько классов. Каждый класс располагает собственным набором методов. В каждом классе могут находиться одноименные методы. В следующем примере объявляются три класса в одном пространстве имен. В каждом классе объявляется независимая точка входа и три (!) СТАТИЧЕСКИЕ функции `Main` (возможно и такое). Здесь 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
  ```

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

Транслятор создаст сборку, в которой будет обеспечена передача управления соответствующей функции `Main` (одной из трех!):

```
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;
}

}

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

Листинг 3.7.

Структура также может иметь свою точку входа!

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

Введение в программирование на C# 2.0

4. Лекция: Объекты: версия для печати и PDA
В соответствии с принципами объектно-ориентированного программирования решение поставленной задачи сводится к разработке модели (объявлению класса) и созданию экземпляров (объектов), представляющих реализацию этой модели.В этой лекции обсуждаются проблемы, связанные с созданием и последующим уничтожением объектов

В соответствии с принципами объектно-ориентированного программирования решение поставленной задачи сводится к разработке модели (объявлению класса) и созданию экземпляров (объектов), представляющих реализацию этой модели.

В этой главе обсуждаются проблемы, связанные с созданием и последующим уничтожением объектов.

Создание объекта. Конструктор

Конструктором называется множество операторов кода, которому передается управление при создании объекта. Синтаксис объявления конструктора аналогичен объявлению метода — те же спецификаторы доступа, имя, список параметров. Особенности конструктора заключаются в том, что:

- конструктор НЕ ИМЕЕТ НИКАКОГО возвращаемого спецификатора, 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 передается посредством выполнения специальной операции `new`.

Операция new

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

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

new также используется для обращения к конструкторам объектов типа значений:

```
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;
```

Конструктор БЕЗ ПАРАМЕТРОВ (конструктор умолчания) обеспечивает инициализацию переменной предопределенным значением. Со списком предопределенных значений, которыми инициализируются объекты предопределенных 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 типов — значений конструкторов с параметрами в принципе нет!

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

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

Сколько конструкторов может иметь 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);
 }
}
```

Листинг 4.1.

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 берет 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 перестает заниматься построением собственных версий конструкторов. Отныне в классе нет больше конструктора умолчания. Теперь все 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 вариантов конструкторов, которые могут потребоваться для решения поставленной задачи. 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 конструкторе, деструкторе, методе, свойстве

Нестатический метод класса определяет индивидуальное поведение объекта и поэтому вызывается "от имени" данного объекта. В большинстве объектно-ориентированных языков программирования реализована возможность непосредственного обращения из нестатического метода (а также свойства) к объекту, который обеспечил выполнение кода данного метода или свойства. Аналогичная возможность обращения к объекту реализуется в теле нестатических конструкторов и деструктора.

В C# связь метода (свойства, конструктора, деструктора) с объектом обеспечивается первичным выражением (элементарным выражением языка) `this`. Это выражение не требует никакого дополнительного объявления, его тип соответствует классу, содержащему объявление данного метода. Он может использоваться в теле любого нестатического метода, конструктора и деструктора (о деструкторах — ниже) в любом классе или структуре, если только они содержат соответствующие объявления.

По своей сути первичное выражение `this` в данном контексте является ссылкой на объект, обеспечивающий выполнение данного кода.

Если метод возвращает ссылку на объект, который вызвал данный метод, `this` можно использовать в качестве возвращаемого значения оператора return.

Для метода `ToString` все ссылки на объекты, представляющие данный класс, неразличимы. Следующий пример подтверждает тот факт, что `this` в нестатическом методе является всего лишь ссылкой на объект — представитель данного класса.

```
using System;

namespace XXX
{
class TEST
{
int x;
int y;
public string str;

public TEST(int xKey, int yKey)
{
// В конструкторе this – это всегда ссылка
// на создаваемый объект
this.x = xKey;
y = yKey;
str = "Ha-Ha-Ha";
}

public TEST f1(string strKey)
{
this.str = this.str + strKey + this.ToString();
// Здесь можно будет посмотреть,
// что собой представляет значение ссылки на объект.
return this; // А вот здесь без выражения this просто
// ничего не получается.
}
}

class Program
{
static void Main(string[] args)
{
TEST t, tM;
t = new TEST(10, 10);
Console.WriteLine(t.str);
tM = t.f1(" from ");
t = new TEST(100, 100);
Console.WriteLine(t.str);
Console.WriteLine(t.ToString());
Console.WriteLine(tM.str);
Console.WriteLine(tM.ToString());
}
}
}
```

Листинг 4.2.

this в заголовке конструктора

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

Ситуация: в структуре или классе объявляются различные версии конструкторов, специализирующиеся на создании объектов разной модификации. При этом все объявленные конструкторы вынуждены выполнять один и тот же общий перечень регламентных работ по инициализации объектов.

Для сокращения размера кода можно оставить ОДИН конструктор, выполняющий ВСЮ "черновую" обязательную работу по созданию объектов. Тонкую настройку объектов можно производить после выполнения кода конструктора, непосредственно вызывая соответствующие методы-члены. При этом методы вызываются именно ПОСЛЕ того, как отработает конструктор.

Можно также определить множество специализированных конструкторов, после выполнения которых следует вызывать дополнительный метод инициализации для общей "достройки" объекта.

И то и другое — 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.

Вот так `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
// еще 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++)
 {
// Здесь тоже заблаговременно  позаботились о значениях.
// Хорошо.
 }
Console.WriteLine("Point2D({0}) is here!", pKey.ToString());
 }
}
```

Листинг 4.3.

Уничтожение объектов в управляемой памяти. Деструктор

Время начала работы сборщика мусора неизвестно. Сборка мусора предотвращает утечки памяти в управляемой динамически распределяемой памяти, программист освобождается от ответственности за проблемы, связанные с распределением этого ресурса.

Однако сборщик мусора не способен освобождать другие типы ресурсов, например, соединения с базами данных и серверами (надо разъединять) и открытые файлы (надо закрывать).

Если соответствующие фрагменты кода, обеспечивающие освобождение этих ресурсов, разместить в специальном блоке операторов с заголовком, состоящем из имени класса с префиксом '~' и пустым списком параметров (в деструкторе), то время их освобождения совпадет со временем выполнения кода данного деструктора, который запускается при разрушении объекта сборщиком мусора. Однако время наступления этого события для конкретного объекта остается неопределенным. Порядок передачи управления деструкторам различных объектов в куче во время очистки памяти также неизвестен.

Простой пример, иллюстрирующий принципы функционирования управляемой памяти и работы деструктора:

```
using System;

namespace MemoryTest01
{

// Объекты этого класса регистрируют деятельность конструктора
// при создании объекта и деструктора при его уничтожении.
 public class MemElem
 {
  public static long AllElem = 0;
  long N = 0;
public MemElem(long key)
{
AllElem++;
N = key;
Console.WriteLine("{0} element was created. {1} in memory!", N, AllElem);
}


// Объявление деструктора.
~MemElem()
{
AllElem--;
Console.WriteLine("{0} was destroyed by GC. {1} in memory!", N, AllElem);
}
}

class Program
{

// Порождение объектов-представителей класса MemElem.
// Партиями по 50 штук.
// Периодичность активности GC неизвестна.

static void Main(string[] args)
 {
    MemElem mem;
    string s;
    long N = 0;
    int i = 0;
```

```
    for ( ; ; )
    {
     Console.WriteLine("_____________________________");
     Console.Write("x for terminate >> ");
     s = Console.ReadLine();
     if (s.Equals("x")) break;
     else N += i;
     for (i = 0; i < 50; i++)
     {
       mem = new MemElem(N+i);
     }
   }
  }
 }
 }
```

Листинг 4.4.

Для объектов, связанных с неуправляемыми ресурсами (объект содержит дескриптор файла, который в данный момент может быть открыт), рекомендуется определять специальные методы освобождения этих ресурсов вне зависимости от того, используется или нет данный объект.

Класс GC

Однако, несмотря на вышеуказанные особенности управляемой памяти, в .NET в известных пределах можно управлять сборкой мусора.

В составе библиотеки классов присутствует класс GC, в котором реализованы методы управления сборщиком мусора.

Public-свойства	
`MaxGeneration`	Максимальное количество поддерживаемых в управляющей памяти поколений. Статическое
Public-методы	
`AddMemoryPressure`	Уведомление среды выполнения о резервировании большого объема неуправляемой памяти, который необходимо учесть при планировании работы сборщика мусора. Статический
`Collect`	Перегруженный. Активизирует процесс сборки мусора. Сборка происходит в отдельном потоке. Поэтому время начала деятельности сборщика остается неопределенным. Статический
`CollectionCount`	Определяет общее количество проходов сборщика для данного поколения объектов. Статический
`Equals`	Определение эквивалентности объектов.
`GetGeneration`	Перегруженный. Возвращает значение номера поколения, содержащего данный объект. Статический
`GetTotalMemory`	Возвращает количество байт, занятых под объекты в управляющей памяти. В зависимости от значения параметра типа `bool` учитываются (или не учитываются) результаты деятельности сборщика мусора в момент выполнения метода. Сборщик мусора работает в теневом потоке, и в принципе можно немножко подождать результатов его работы. Статический
`KeepAlive`	Ссылается на указанный объект, делая его недоступным для сборщика мусора с момента начала текущей программы до вызова этого метода. Статический
`RemoveMemoryPressure`	Информирует среду выполнения об освобождении области неуправляемой памяти. Эта информация может быть полезна сборщику мусора для планирования работы. Статический
`SuppressFinalize`	Метод обеспечивает уведомление сборщика мусора о том, что данный объект (представляется ссылкой в параметре метода) не подлежит удалению. Статический
`ReRegisterForFinalize`	Сначала защищаем объект от GC путем вызова метода `SuppressFinalize`. А теперь снимаем с объекта эту защиту. Статический
`WaitForPendingFinalizers`	Приостанавливает текущий поток до тех пор, пока поток, обрабатывающий очередь финализаторов, не обработает всю очередь. Статический

Далее демонстрируются методы управления сборкой мусора. Используется модифицированный класс `MemElem`:

```
using System;

namespace GCExample
{
// Объекты этого класса регистрируют деятельность конструктора
// при создании объекта и деструктора при его уничтожении.

public class MemElem
{
 public static long AllElem = 0;
 long N = 0;
 public MemElem(long key)
 {
 AllElem++;
 N = key;
 Console.WriteLine("{0} was created. {1} in memory!", N, AllElem);
 }

~MemElem()
{
AllElem--;
Console.WriteLine("{0} was destroyed. {1} in memory!", N, AllElem);

// Смотрим общее количество байтов, занимаемых объектами в куче.
// Значение параметра (false) свидетельствует о том, что эта информация
// требуется незамедлительно, вне зависимости от того, работает ли в данный
// момент в теневом потоке сборщик мусора или нет.
// А в этом случае значения true мы готовы подождать окончания работы
// сборщика мусора, если он в данный момент разбирает завалы.
// Заменить false на true и осознать разницу!
```

```
Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
}
}

class GCTestClass
{
private const long maxGarbage = 50;

static void Main()
{
// Можно посмотреть максимальное количество поколений
// (maximum number of generations),
// которое в данный момент поддерживается
// системой сборки мусора.
Console.WriteLine("The highest generation is {0}", GC.MaxGeneration);

// Создан объект - представитель класса GCTestClass.
GCTestClass gct = new GCTestClass();

// Начали забивать память всяким хламом.
gct.MakeSomeGarbage();

// Можно посмотреть, в каком "поколении" расположился объект.
Console.WriteLine("gct is in Generation: {0}", GC.GetGeneration(gct));

// Активизировали сборщик мусора для объектов генерации 0.
// Время начала сборки не определено.
GC.Collect(0);
// Смотрим, в каком поколении располагается данный объект. Переместился.
Console.WriteLine("gct is in Generation: {0}", GC.GetGeneration(gct));

// Perform a collection of all generations up to and including 2.
// Сборка мусора во всех поколениях, включая второе.
// Время начала сборки не определено.
GC.Collect(2);

// А в каком поколении gct сейчас? Ну надо же... опять переместился.
Console.WriteLine("gct is in Generation: {0}", GC.GetGeneration(gct));
// И как дела с памятью?
Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
Console.Read();
}


// Метод, который всего лишь забивает память всякой ненужной ерундой.
void MakeSomeGarbage()
{
// В данном случае в качестве мусора используются объекты -
// представители двух замечательных классов...
//Version vt;
//Object ob;
MemElem mem;
for (int i = 0; i < maxGarbage; i++)
{
//vt = new Version();
//ob = new Object();
mem = new MemElem(i);
Console.WriteLine("mem is in Generation: {0}", GC.GetGeneration(mem));
}
}
}
}
```

Листинг 4.5.

Деструктор и метод Finalize

Проблема освобождения неуправляемых ресурсов в .NET решается за счет применения метода `Finalize`, который наследуется любым классом от `object`. По умолчанию метод `Object.Finalize`

```
void Finalize()...
```

не делает ничего. Если же требуется, чтобы сборщик мусора при уничтожении объекта выполнил завершающие операции (закрыл ранее открытый файл, разорвал соединение с базой данных), то в соответствующем классе этот метод должен быть переопределен. В нем должен быть размещен соответствующий программный код, обеспечивающий освобождение ресурсов.

В .NET метод `Finalize` занимает особое место, и его использование регламентируется следующими ограничениями:

- предполагается, что управление этому методу передается непосредственно сборщиком мусора при удалении данного объекта;
- НЕ РЕКОМЕНДУЕТСЯ вызывать метод `Finalize` из методов, непосредственно не связанных с уничтожением объекта, тем более из методов других классов. Поэтому метод `Finalize` должен объявляться со спецификатором `protected`. При этом единственным корректным вызовом метода считается вызов метода `Finalize` для базового класса с использованием нотации `base (...base.Finalize();...)`;
- в методе `Finalize` должны освобождаться только те ресурсы, которыми владеет уничтожаемый объект.

Транслятор C# знает об особом статусе метода `Finalize` и о проблемах, связанных с его использованием. Соответствующее объявление метода в классе сопровождается предупреждением (warning), которое гласит:

```
Introgucing a 'Finalize' method can interfere with destructor invocation.
Did you intend  to declare a destructor?
```

То есть не лучше ли заменить этот метод деструктором? Дело в том, что для C# транслятора следующие строки кода являются эквивалентными:

```
~MyClass() // объявление деструктора.
{
  // Здесь реализуются алгоритмы освобождения ресурсов.
}


protected override void Finalize() // Объявление метода финализации.
{
  try
  {
    // Здесь реализуются алгоритмы освобождения ресурсов.
  }
  finally
  {
    base.Finalize(); // Вызов Finalize для базового класса.
  }
}
```

и если в классе объявляется деструктор (деструктор всегда предпочтительнее), то присутствие метода

```
 protected override void Finalize()...
```

в том же классе воспринимается как ошибка.

Таким образом, освобождение ресурсов в программе на C# возлагается на `Finalize`, точнее, на деструктор, который запускается непосредственно GC.

Совмещение уничтожения объекта с освобождением ресурсов не может рассматриваться как оптимальное решение. Время начала работы сборщика мусора остается неопределенным даже после его активизации, а немедленное освобождение ресурса может оказаться решающим для дальнейшего выполнения приложения.

Поэтому в .NET предусмотрены альтернативные (детерминированные) способы освобождения неуправляемых ресурсов, которые будут рассмотрены ниже.

Введение в программирование на C# 2.0

5. Лекция: Массивы: версия для печати и PDA
В этой лекции обсуждаются массивы. Функциональность класса массива, категории массивов, синтаксис объявления, инициализация, применение массивов

В этой главе обсуждаются массивы. Функциональность класса массива, категории массивов, синтаксис объявления, инициализация, применение массивов.

Массив. Объявление

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

Существует несколько способов создания 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),
- элементы массива ВСЕГДА однотипны, располагаются в непрерывной области памяти,
- доступ к элементу массива осуществляется по индексу, при этом допускается случайный доступ.

Одной из характеристик массива является ранг или размерность массива. 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. При этом составляющая массива сама может быть либо массивом, либо элементом массива:

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

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

Листинг 5.1.

`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 ограничена.

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 намека 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 (зубчатые, зазубренные; неровно оторванные; пьяные; находящиеся под влиянием наркотиков) массивы.

Инициализация массивов

Массив мало объявить, его нужно еще 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 определенным значением.

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 сокращенной 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};
```

К моменту создания массива должна быть определена его основная характеристика – количество составляющих данный массив элементов. 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[x][125];
// Внимание! Для jagged такое определение недопустимо!
// Нет никаких ограничений  на размеры составляющих массивов.
// Для каждой составляющей требуется ЯВНАЯ спецификация.
```

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

Количество составляющих массив элементов — это не всегда ВСЕ образующие массив элементы!

При определении такой конструкции, как МАССИВ МАССИВОВ, инициализирующая запись должна содержать ПОЛНУЮ информацию о характеристиках первой составляющей массива, то есть об общем количестве составляющих данный массив элементов массивов. 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 и наглядно представлено одним оператором.

При этом само объявление оператора может содержать лишь частичную инициализацию, а может и не содержать ее 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 его объявлении.

```
ИнициализацияКопированием ::= СпецификацияСоставляющей
СпецификацияСоставляющей ::= ИмяМассива СпискомИндексныхВыражений
СпискомИндексныхВыражений ::= [СпискомИндексныхВыражений] [Индексное Выражение]
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[0][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 определенный массив q.
```

В следующем примере одномерный массив `myArray` строится из элементов типа `int`.

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

Операция `new` используется для создания массива и инициализации его элементов предопределенными 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 спецификатор 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 делать:

```
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.
```

А вот объявление одномерного массива, состоящего из трех 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 одномерным 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 еще 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-массива, 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 еще один пример, где строится массив `myArray`, элементами которого являются массивы. Каждый из составляющих имеет собственные размеры.

```
using System;
public class ArrayTest
{
public static void Main()
 {
int[][] myArray = new int[2][];
myArray[0] = new int[5] {1,3,5,7,9};
myArray[1] = new int[4] {2,4,6,8};

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
```

Value Type и Reference Type. Два типа массивов

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 типом-значением или типом-ссылкой.

Если это тип-значение, результатом выполнения оператора будет массив, содержащий 10 объектов `MyType` с предопределенными значениями.

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.

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 следующие 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), создает массив элементов заданного типа и определенной размерности:

```
public int GetLength (int dimension);
```

Возвращает количество элементов заданной параметром 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( "Вышел ", 0, 0 );
myArray.SetValue( "Месяц, ", 0, 1 );
myArray.SetValue( "из", 0, 2 );
myArray.SetValue( "тумана,", 0, 3 );
myArray.SetValue( "вынул", 1, 0 );
myArray.SetValue( "ножик", 1, 1 );
myArray.SetValue( "из", 1, 2 );
myArray.SetValue( "кармана.", 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 ) );
}
}
```

Элементы массива массивов (jagged array) на одном и том же уровне 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.

Массивы как параметры

В качестве параметра методу всегда можно передать ссылку на массив — в частности, ссылку на ОДНОМЕРНЫЙ 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 определенное (соответствующее значению первой размерности) количество проинициализированных ссылок на составляющие данный одномерный массив элементы.

Этот принцип действует во всех случаях, в том числе и при вызове 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 создается консольное приложение (пусть под именем `ConLine`, в файле `ConLine.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);
```

Передаваемый в качестве параметра массив может быть предварительно проинициализирован:

```
// 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);
 }
}
```

Еще 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}});
 }
}
```

Спецификатор params

Неопределенное (переменное) количество (ОДНОТИПНЫХ!) параметров или список параметров переменной длины передается в функцию в виде ссылки на одномерный массив. Эта ссылка в списке параметров функции должна быть последним элементом списка параметров. Ссылке должен предшествовать спецификатор `params`.

В выражении вызова метода со списком параметров, члены списка могут присутствовать либо в виде списка однотипных значений (этот список преобразуется в массив значений), либо в виде ссылки на ОДНОМЕРНЫЙ массив значений определенного типа:

```
using System;

class Class1
{

// Методы, принимающие списки параметров переменной длины.
// Их объявление. Единственный параметр способен принимать
// массивы значений переменной длины.
static void f1(params int[] m)
{
Console.WriteLine(m.Length.ToString());
}

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


static void Main(string[] args)
{
// Выражения вызова. На основе списков однотипных
// значений формируются массивы.
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}});
}
}
```

Листинг 5.2.

Введение в программирование на C# 2.0

6. Лекция: Перегруженные операции: версия для печати и PDA

Основная конструкция C# – объявление класса. Класс есть тип. Тип характеризуется неизменяемым набором свойств и методов. Для предопределенных типов определены множества операций, которые кодируются с использованием множества определенных в языке символов операций. Мощнейшее средство языка C# перегрузка операций обсуждается в этой лекции

Основная конструкция C# – объявление класса. Класс есть тип. Тип характеризуется неизменяемым набором свойств и методов. Для предопределенных типов определены множества операций, которые кодируются с использованием множества определенных в языке символов операций.

Язык позволяет строить сложные выражения с использованием этих операций, причем результат выполнения (определения результирующего значения) зависит от типа элементарных выражений, составляющих сложное выражение. Например, операция сложения для целочисленных значений определяется и выполняется иначе, нежели сложение для чисел с плавающей точкой.

На основе объявляемых в программе классов в программе могут создаваться новые объекты. Эти объекты, в принципе, ничем не отличаются от других объектов, в том числе от объектов — представителей предопределенных 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 типов C# предоставляет те же возможности, что и для выражений предопределенных типов. Главная проблема заключается в том, что алгоритм вычисления значения выражения, представленного операндами вновь объявляемого типа в сочетании с символами операций, применяемых с операндами данного типа, неизвестен. Правила вычисления значения для такого выражения должны быть специальным образом определены программистом при определении класса. Эти правила (алгоритм вычисления значения выражения) задаются при определении функций специального вида, выражения вызова которых внешне напоминают выражения, построенные на основе операндов предопределенного типа и соответствующих символов операций.

Перегрузка операций

Перегрузка операций в C# является способом объявления семантики новых операций, которые обозначаются принятыми в C# символами 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 (перегружены) подобным образом. Некоторые операции могут перегружаться с ограничениями.

В таблице приводится соответствующая информация относительно перегрузки различных категорий символов операций.

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

При этом при перегрузке операций для любого нового типа выполняются следующие правила:

- префиксные операции `++` и `--` перегружаются парами;
- операции сравнения перегружаются парами: если перегружается операция `==`, также должна перегружаться операция `!=`. То же самое относится к парам `<` и `>`, `<=` и `>=`.

Операторная функция. Объявление

Перегрузка операций основывается на следующих принципах:

- определяется статическая операторная функция особого вида (синтаксис объявления рассматривается ниже), в заголовке которой указывается символ переопределяемой операции;
- в теле функции реализуется соответствующий алгоритм, определяющий семантику операторной функции (семантику новой операции);
- выражение вызова операторной функции имитирует внешний вид выражения, построенного на основе соответствующего символа операции;
- при трансляции выражения, включающего символы операций, производится анализ типов операндов данного выражения, на основе которого идентифицируется соответствующая операторная функция, которой и обеспечивается передача управления (возможно, что при этом автоматически реконструируется соответствующее выражение вызова операторной 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
              ::= ;
```

Листинг 6.1.

Унарные операторные функции. Пример объявления и вызова

```
using System;

class Point2D
{
float x,y;

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

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


// Перегруженные операции обязаны возвращать значения!
// Должны объявляться как public и static.
// При этом префиксная и постфиксная формы операций ++ и --,
// в отличие от оригинальных операций, семантически НЕ различаются.
// Каждая из этих операций может быть объявлена либо как префиксная:
public static Point2D operator ++ (Point2D par)
{
par.x++;
par.y++;
return par;
}

// либо 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;
}

// Что на самом деле есть криво!
```

```
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.
}
}
```

Листинг 6.2.

Бинарные операции. Пример объявления и вызова

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 по аналогичной схеме. Хотя не существует никаких ограничений на тип возвращаемого значения, в силу специфики применения (обычно в условных выражениях операторов управления) операций сравнения все же имеет смысл определять их как операторные функции, возвращающие значения `true` и `false`:

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

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

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

В программе можно поинтересоваться непосредственно значением некоторого поля объекта:

```
Point2D p1 = new Point2D(GetVal(), GetVal());
 ::::::::::
// Это логическое выражение!
if ((p1.x).Equals(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 `true` типа `bool` для обозначения факта `true` и возвращает `false` в противном случае.

Классы, включающие объявления подобных операций (операторных функций), могут быть использованы в структуре операторов `if, do, while, for` в качестве условных выражений.

При этом, если в классе была определена операция `true`, в том же классе должна быть объявлена операция `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. В частности, степень удаления от точки
// с координатами (0,0).
public static bool operator true (Point2D par)
{
if (Math.Sqrt(par.x*par.x + par.y*par.y) < 10.0) return true;
else                                             return false;
}

public static bool operator false (Point2D par)
{
double r = (Math.Sqrt(par.x*par.x + par.y*par.y));
if (r > 10.0 || r.Equals(10.0))                return false;
else                                          return true;
}
```

Определение операций. Конъюнкция и дизъюнкция

После того как объявлены операторы `true` и `false`, могут быть объявлены операторные функции — конъюнкция и дизъюнкция. Эти функции работают по следующему принципу:

- после оценки истинности (критерий оценки задается при объявлении операций `true` и `false`) операндов конъюнкции или дизъюнкции функция возвращает ссылку на один из операндов либо на вновь создаваемый в функции объект;
- в соответствии с ранее объявленными операциями `true` и `false`, операторные конъюнкция и дизъюнкция возвращают логическое значение, соответствующее отобранному или вновь созданному объекту:

```
public static Point2D operator | (Point2D par1, Point2D par2)
{
if (par1) return par1; // Определить "правильность" объекта par1 можем!
if (par2) return par2; // Определить "правильность" объекта par2 можем!
return new Point2D(10.0F, 10.0F); // Вернули ссылку на
                                  // новый "неправильный" объект.
}

public static Point2D operator & (Point2D par1, Point2D par2)
{
if (par1 && par2) return par1; // Вернули ссылку на один из "правильных"
                               // объектов.
else return new Point2D(10.0F, 10.0F); // Вернули ссылку на
                               // новый "неправильный" объект.
}
```

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 условия:

- тип возвращаемого значения и типы каждого из параметров данной операторной функции должны быть типа `T`. Операторные функции `operator &` и `operator |`, определенные 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:

1. Операция `x && y` представляется в виде выражения, построенного на основе трехместной 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`.

2. Операция `x || y` представляется в виде выражения, построенного на основе трехместной 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, а 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 еще много чего...
public Point2D():this(0.0F,0.0F)
{
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 (Math.Sqrt(par.x*par.x + par.y*par.y) < 10.0) return true;
else                                             return false;
}

public static bool operator false (Point2D par)
{
double r = (Math.Sqrt(par.x*par.x + par.y*par.y));
if (r > 10.0 || r.Equals(10.0))                 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(10.0F, 10.0F);
}

public static Point2D operator & (Point2D par1, Point2D par2)
{
if (par1 && par2) return par1;
else return new Point2D(10.0F, 10.0F);
}
}

// Стартовый класс.


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 у 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 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...");
}
}
}
}
}
```

Листинг 6.3.

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;
}

// Внимание! Объявление индексатора.
// Индексатор НЕ является операторной функцией.
// Он объявляется как НЕСТАТИЧЕСКОЕ множество операторов
// со стандартным заголовком. При его объявлении используется
// ключевое слово this.
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 еще одного (перегруженного!) индексатора.
// В качестве значения для индексации (параметра 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 использующие конкатенацию строк.
wmp0 = mpArr["1"+"2"+"5"];
}
}
}
```

Листинг 6.4.

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 на соответствующее преобразование.

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. Существует мнение,
// что об этом обстоятельстве программисту следует напоминать всякий раз,
// когда он в программном коде собирается применить данное преобразование.
public static explicit operator Point2D(Point3D p3d)
{
   Point2D p2d = new Point2D();
   p2d.x = p3d.x;
   p2d.y = p3d.y;
   return p2d;
}

}
// Тестовый класс. Здесь все 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 будет потеряна в случае присвоения значения типа Point3D
 // значению типа Point2D.
  // Ключевое слово explicit в объявлении соответствующего
  // метода преобразования вынуждает программиста подтверждать,
  // что он в курсе возможных последствий этого преобразования.
  p2d = (Point2D)p3d;
}
}
```

Листинг 6.5.

Введение в программирование на C# 2.0

7. Лекция: Наследование и полиморфизм: версия для печати и PDA
Наследование и полиморфизм являются одними из принципов ООП. О том как они реализованы в C# в этой лекции

Наследование является одним из принципов ООП.

Основы классификации и реализация механизмов повторного использования и модификации кода. Базовый класс задает 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 ВСЕ, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем наследством.

Добавляем в базовый класс `private`-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 `private` данным — членам и функциям — членам класса `A`. Косвенное влияние на такие данные-члены и функции — члены – лишь через `public`-функции класса `A`.

Следует иметь в виду еще одно важное обстоятельство.

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

`Public` ... `private`

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

Используем еще 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!
 }
```

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

И еще 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 имен (поэтому и классы в нашем пространстве имен были объявлены со спецификаторами доступа `public`). Но в пространстве имен 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.

Таким образом, передача управления конструктору базового класса осуществляется посредством конструкции

```
...(...):base(...){...},
```

которая располагается в объявлении конструктора класса-наследника между заголовком конструктора и телом. После ключевого слова `base` в скобках располагается список значений параметров конструктора базового класса. Очевидно, что выбор соответствующего конструктора определяется типом значений в списке (возможно, пустом) параметров:

```
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
{
```

```
// Явные обращения к конструкторам базового класса.
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);
}
}
```

Листинг 7.1.

Для создания объектов в программе можно применять конструкторы трех степеней защиты:

`public` – при создании объектов в рамках данного пространства имен, в методах любого класса — члена данного пространства имен;

`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);
```

С этого момента в производном классе нет больше конструктора умолчания. Теперь все зависит от соответствия оператора определения объекта построенному нами конструктору.

Объявим в производном классе оба варианта конструкторов. И опять все хорошо. Конструктор умолчания базового класса (тот, который строится транслятором) продолжает исправно выполнять свою работу.

Проблемы в производном классе 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 начнет 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 применялись при объявлении членов базового класса. Это касается как методов, так и данных — членов объявляемых классов.

В этом случае соответствующие члены базового класса считаются переопределенными.

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 слово `new` выступает в качестве модификатора, который используется для явного обозначения переопределяемых членов базового класса:

```
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 переопределенным 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 переопределенные члены базового класса не видны.
y0.fun1();
y0.q = 100;
y0.fun0();
y0.q0 = 125;
 }
}
}
```

Листинг 7.2.

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

Дело в том, что переопределенные члены базового класса при "взгляде" на производный класс "извне" не видны. В производном классе они замещаются переопределенными членами, и непосредственный доступ к этим членам возможен только из функций — членов базового и производного классов. При этом для обращения к переопределенному 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.

Объявление класса может содержать вложенное (или внедренное) объявление класса. C# допускает и такие объявления:

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

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 защищенных 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 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, остается доступным в производном классе.

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

- константа, поле, свойство или внедренный в класс тип или структура скрывают ВСЕ одноименные члены базового класса;
- внедренный в класс или структуру метод скрывает в базовом классе свойства, поля, типы, обозначенные тем же самым идентификатором, а также одноименные методы с той же самой сигнатурой;
- объявляемые в производном классе индексаторы скрывают одноименные индексаторы в базовом классе с аналогичной сигнатурой.

Одновременное использование в производном классе члена базового класса и его переопределенного потомка является ошибкой.

Полное квалифицированное имя. Примеры использования

В этом примере и базовый 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 все-таки можно обратиться к переопределенному 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 член базового класса
public static void Main()
 {
// Доступ к переопределенному 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 переопределенного 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 переопределенный вложенный класс:
C s1 = new C();
// Полное имя используется для доступа к классу, вложенному в базовый:
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 которой дальнейшее совершенствование и переопределение возможностей класса в деле решения специфических задач окажется нежелательным (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 переопределенной функции.
::::::::::
return 0;
}
}
```

Такой код никому не нужен, и C# позволяет избегать подобных 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 объекта:

```
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 переопределенная абстрактная функция!
y0.f0();
}
```

Еще пример:

```
using System;
namespace Interface01
{
// Абстрактный класс.
abstract class aX1
{
public int xVal;
// Его конструкторы могут использоваться при построении
// объектов классов-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 переопределенную функцию. В базовом классе это
// абстрактная функция.
}
}
}
```

Листинг 7.3.

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, fun0()");
}

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

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

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

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

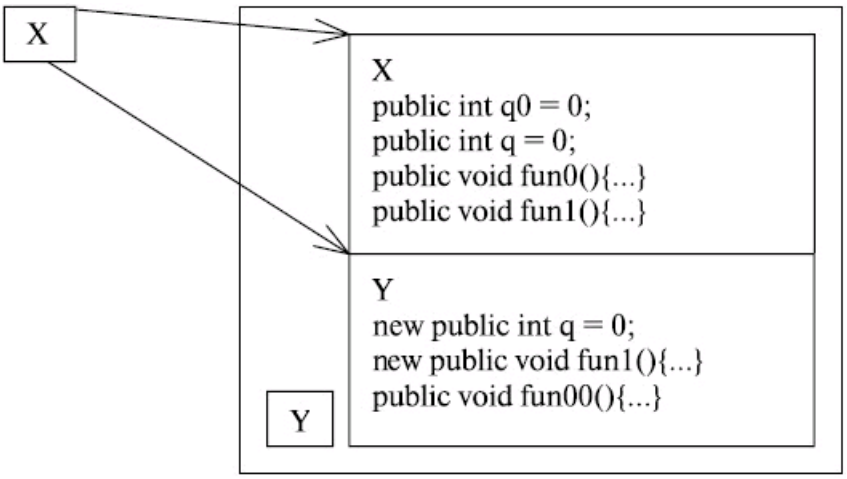
```
{
X x = null; // Просто ссылка!
// Объекты - представители производных классов-наследников.
Y y = new Y(); y.fun0(); y.fun00(); y.fun1();
Z z = new Z(); z.fun0(); z.fun00(); z.fun1();
// Настройка базовой ссылки.
x = y; x.fun0(); x.fun1(); x.q = 100; x.q0 = 125;
x = z; x.fun0(); x.fun1(); x.q = 100;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

}
```

Листинг 7.4.

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 из 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

При преобразованиях ссылочных типов используются операции преобразования `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),
// удается 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
// удается 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("!!!!!");
}
}
}
}
}
```

Листинг 7.5.

Boxing и Unboxing. Приведение к типу object

Любой тип .NET строится на основе базового типа (наследует) `object`. Это означает, что:

- методы базового типа `object` доступны к выполнению любым (производным) типом,
- любой объект (независимо от типа) может быть преобразован к типу `object` и обратно. Деятельность по приведению объекта к типу object называется `Boxing`. Обратное преобразование называют `Unboxing`.

Примеры вызова методов базового класса:

```
int i = 125;
i.ToString();
i.Equals(100);
```

Примеры преобразований:

```
int i = 125;  // Объявление и инициализация переменной типа int.
object o = i; // Неявное приведение к типу object в результате присвоения.
object q = (object)i; // Явное приведение к типу object.

int x = (int)o; // Пример Unboxing'a.
```

И еще пример кода на ту же тему:

```
using System;

namespace Boxing02
{
class TestClass
{
public int t;
public string str;
// Конструктор класса принимает параметры в упакованном виде.
public TestClass(object oT, object oStr)
{
t = (int)oT; // unboxing посредством явного преобразования.
str = oStr as string; // unboxing в исполнении операции as.
}
}

class Program
{
static void Main(string[] args)
{
TestClass tc;
// При создании объекта пакуем параметры.
tc = new TestClass((object)0, (object)(((int)0).ToString()));
Console.WriteLine("tc.t == {0}, tc.str == {1}", tc.t, tc.str);
}
}
}
```

Виртуальные функции. Принцип полиморфизма

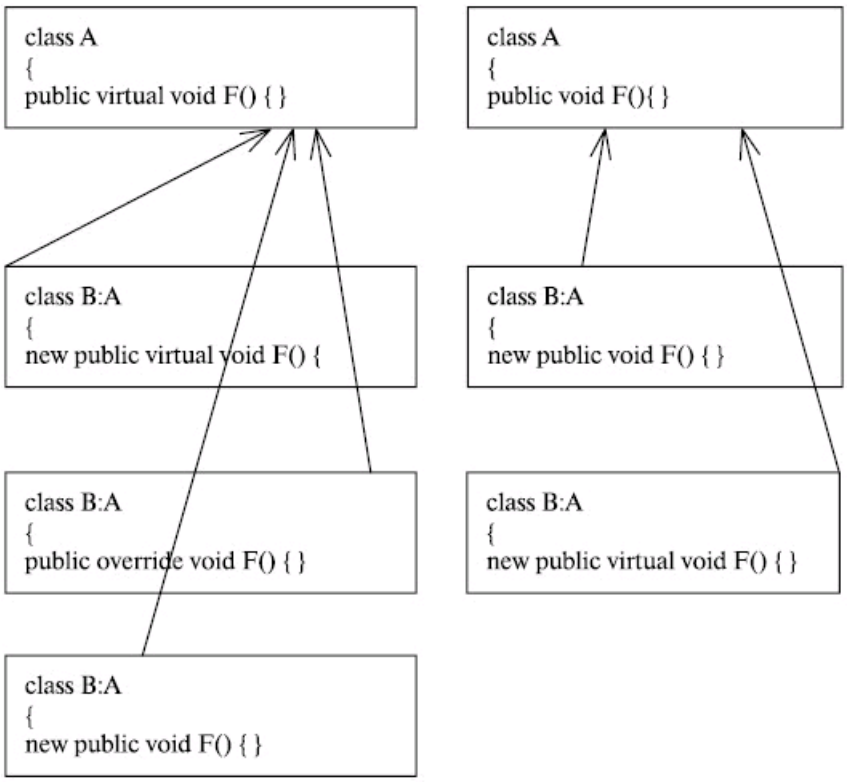
Основа реализации принципа полиморфизма – наследование. Ссылка на объект базового класса, настроенная на объект производного, может обеспечить выполнение методов ПРОИЗВОДНОГО класса, которые НЕ БЫЛИ ОБЪЯВЛЕНЫ В БАЗОВОМ КЛАССЕ. При реализации принципа полиморфизма происходят вещи, которые не укладываются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

Одноименные функции с одинаковой сигнатурой могут объявляться как в базовом, так и в производном классах. Между этими функциями может быть установлено отношение замещения:

функция ПРОИЗВОДНОГО класса ЗАМЕЩАЕТ функцию БАЗОВОГО класса.

Основное условие отношения замещения заключается в том, что замещаемая функция базового класса должна при этом дополнительно специфицироваться спецификатором `virtual`.

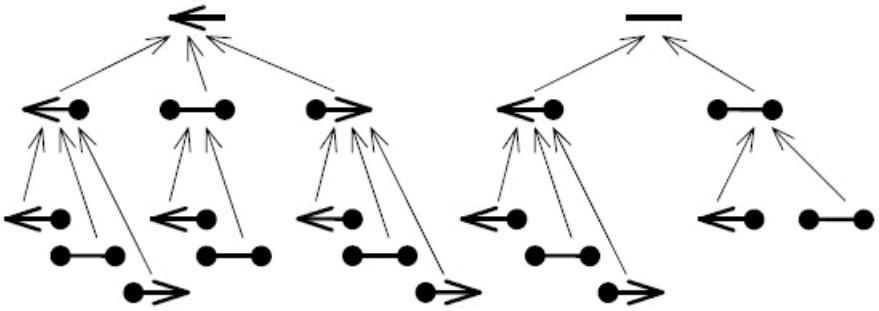
Отношение замещения между функциями базового и производного класса устанавливается, если соответствующая (одноименная функция с соответствующей сигнатурой) функция производного класса специфицируется дополнительным спецификатором `override`.



Введем следующие обозначения и построим схему наследования.

`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 трех 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 */
}
}
}
```

Листинг 7.6.

в котором становится ВСЕ понятно, если ПОДРОБНО нарисовать схемы классов, структуру объекта — представителя класса `D` и посмотреть, КАКАЯ ссылка ЗА КАКОЕ место этот самый объект удерживает.

Введение в программирование на C# 2.0

8. Лекция: Интерфейсы: версия для печати и PDA
Интерфейсы фактически те же самые абстрактные классы, не содержащие объявлений данных-членов и объявлений обычных функций. О них рассказанно в этой лекции

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.

Объявление интерфейса

```
using System;

namespace Interface01
{

// Интерфейсы.
// Этот интерфейс характеризуется уникальными
// именами объявленных в нем методов.
interface Ix
{
void IxF0(int xKey);
void IxF1();
}

// Пара интерфейсов, содержащих объявления одноименных 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 имен 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();
}
}
}
```

Листинг 8.1.

Преимущества программирования с использованием интерфейсов проявляются в том случае, когда ОДНИ И ТЕ ЖЕ ИНТЕРФЕЙСЫ наследуются РАЗНЫМИ классами. При этом имеет место еще один вариант полиморфности: объект, представленный ссылкой-интерфейсом, способен проявлять различную функциональность в зависимости от реализации функций наследуемого интерфейса в классе данного объекта. И здесь все 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.
// Для классов-наследников интерфейсы представляют собой всего лишь
// базовые классы. Поэтому в соответствующих функциях сравнения
// допускается использование операции as.
//_____________________________________________________________.
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 будет 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 вернет отрицательного значения по причине
// несоответствия типов операндов.
  Console.WriteLine("__________y2==x2__________");
   if ((y2 as ICompare0).Eq(x2))
   Console.WriteLine("Yes!");
   else
   Console.WriteLine("No!");
}
}
}
```

Листинг 8.2.

Реализация сортировки в массиве. Интерфейс IComparable

Независимо от типа образующих массив элементов, массив элементов данного типа – это особый класс, производный от базового класса `Array`.

Класс `Array` является основой для любого массива и для любого массива предоставляет стандартный набор методов 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 определенным критерием сравнения.

При этом критерии для сравнения и алгоритмы сравнения могут задаваться либо с помощью массивов ключей, либо реализацией интерфейса `IComparable`, который специфицирует характеристики (тип возвращаемого значения, имя метода и список параметров) методов, обеспечивающих реализацию данного алгоритма. Далее будут рассмотрены различные варианты метода `Sort`.

- Вариант метода `Sort` сортирует полное множество элементов одноразмерного массива с использованием алгоритма, который реализуется методами, специфицированными стандартным интерфейсом сравнения:

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

Реализованный на основе интерфейса алгоритм способен распознавать значения элементов массива и сравнивать эти элементы между собой, если это элементы предопределенных типов.

Таким образом, никаких проблем не существует, если надо сравнить и переупорядочить массивы элементов, для которых известно, КАК СРАВНИВАТЬ значения составляющих массив элементов. 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:

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

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 в этом случае алгоритм сравнения способен обеспечить необходимое сравнение.

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.

```
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 четырех 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.

Для сортировки (изменения порядка 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, который позволяет организовать сортировку подмножества элементов целевого массива, начиная с элемента, заданного вторым параметром метода, и включая количество элементов, заданное 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 интерфейса сравнения:

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

  // Сортировка элементов с использованием стандартного Компарера.
  // iArr – одномерный массив целочисленных значений.
  // System.Collections.Comparer.DefaultInvariant – свойство,
  // которое обеспечивает ссылку на объект "стандартный Компарер",
  // реализующий предопределенные алгоритмы сравнения.
  try
  {
  System.Array.Sort(iArr,System.Collections.Comparer.DefaultInvariant);
  }
  catch (Exception ex)
  {
  Console.WriteLine(ex.ToString());
  }
  ```

`System.Collections.Comparer.DefaultInvariant` – свойство, которое обеспечивает доступ к объекту, реализующему обобщенный метод сравнения.

Для любого класса на основе интерфейса `IComparer` можно создать собственный класс-КОМПАРЕР, в котором можно будет реализовать специфический для данного класса сортируемых элементов метод сравнения (interface implemented by each element of the Array).

```
using System;
using System.Collections;

namespace SortArrays
{
// 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 Points не знает,
// как воспринимать и каким образом сравнивать между собой
// объекты-представители класса Points. Объекты p0 и p1
// со значениями координат (7,5) и (5,7) – кто из них "больше"?
class Points
{
// Конечно, можно говорить о введении аксиом для установления отношения
// порядка между элементами множества объектов – представителей класса
// Points и о реализации соответствующих методов сравнения;
// также можно было бы позаботиться при объявлении класса о том,
// чтобы просто не допускать к стандартной сортировке
// объекты классов, в которых не определено
// отношение порядка. Однако в .NET используется другой подход.
public int x;
public int y;
public Points (int key1, int key2)
{
 x = key1;
 y = key2;
```

```
}
// Вычисляется расстояние от начала координат.
public int R
{
get
{
 return (int)(Math.Sqrt(x*x+y*y));
}
}
}

// Предполагается, что отношение порядка над элементами
// множества объектов – представителей соответствующего
// класса может быть введено в любой момент. Для этого
// достаточно иметь четкое представление о критериях
// установления порядка на множестве объектов – представителей
// данного класса и иметь доступ к значениям соответствующих
// полей объектов.
// Стандартные библиотечные средства .NET позволяют
// реализовать (доопределить) соответствующие интерфейсы
// (заготовки) для реализации методов сравнения.
// Ниже объявляется вариант процедуры сравнения,
// использование которого позволяет реализовать стандартные
// алгоритмы сортировки, применяемые в классе Array.
// ...КОМПАРЕР...

class myComparer: IComparer
{
// Предлагаемый метод сравнения возвращает разность расстояний
// двух точек (вычисляется по теореме Пифагора) от начала координат
// – точки с координатами (0,0). Чем ближе точки к началу координат
// – тем они меньше.

int IComparer.Compare(object obj1, object obj2)
{
    return ((Points)obj1).R - ((Points)obj2).R;
}
}

// После реализации соответствующего интерфейса объект-КОМПАРЕР
// обеспечивает реализацию стандартного алгоритма сортировки.
class Class1
{

static void Main(string[] args)
{
    // Объект-генератор "случайных" чисел.
   Random rnd = new Random();

   int i;
    // Массив Points.
    Points[] itArr = new Points[10];

   // Создали Компарер, способный сравнивать пары
   // объектов-представителей класса Points.
   myComparer c = new myComparer();

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

   // Проинициализировали массив объектов – представителей класса Points.
   for (i = 0; i < 10; i++)
   {
      itArr[i] = new Points(rnd.Next(0,10),rnd.Next(0,10));
   }

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

   // Сортируются элементы массива типа Points.
   // 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,c);
   }
   catch (Exception ex)
   {
        Console.WriteLine(ex);
   }

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

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

```
}
}
}
```

Листинг 8.3.

Интерфейс IDisposable. Освобождение ресурсов

Совмещение освобождения ресурсов с удалением объекта-владельца данного ресурса называется недетерминированным освобождением. Зависящее от деятельности сборщика мусора недетерминированное освобождение ресурсов не всегда является оптимальным решением:

- время начала очередного цикла работы GC обычно неизвестно;
- будет ли удален соответствующий объект в случае незамедлительной активизации GC;
- что делать, если ресурс необходимо освободить незамедлительно, а объект должен быть сохранен.

Для детерминированного освобождения неуправляемых ресурсов может быть использован какой-либо специальный метод. Этот метод желательно сделать универсальным, по крайней мере, в смысле его объявления и вызова.

В этом случае для любого вновь объявляемого класса можно предположить существование метода с предопределенным именем и сигнатурой (спецификацией типа возвращаемого значения и списком параметров), который можно было бы стандартным образом вызывать непосредственно от имени объекта-владельца ресурса вне зависимости от активности системы очистки памяти.

В .NET с этой целью используется интерфейс `IDisposable` с методом `Dispose`.

```
public interface IDisposable
{
 void Dispose();
}
```

Таким образом, для реализации механизма освобождения ресурсов достаточно наследовать интерфейс `IDisposable` и обеспечить реализацию метода `Dispose`. После этого деятельность по освобождению ресурсов сводится к теперь стандартному выражению вызова метода `Dispose`.

Следующие правила определяют в общих чертах рекомендации по использованию метода `Dispose`.

В методе `Dispose` освобождаются любые ресурсы, которыми владеет объект данного типа и которые можно освободить.

Если для освобождения ресурсов, которыми владеет объект, не был вызван метод `Dispose`, неуправляемые ресурсы должны освобождаться в методе `Finalize`.

Метод `Dispose` может дублировать действия по освобождению ресурсов, предпринимаемые в деструкторе (`Finalize`) при уничтожении объекта. Для этого можно предусмотреть систему булевых переменных, связанных с состоянием ресурса, или вызов для данного объекта метода `GC.SuppressFinalize`, который для этого объекта запретит выполнение кода деструктора, соответствующего в C# коду метода `Finalize`. Означает ли это, что `GC` уничтожит объект, не передавая при этом управление деструктору?

Метод `Dispose` должен освобождать все ресурсы, удерживаемые данным объектом и любым объектом, которым владеет данный объект.

Следует обеспечить возможность многократного вызова метода. При этом желательно обойтись без генерации исключений. `Dispose` просто не должен пытаться повторно освобождать ранее освобожденные ресурсы.

```
using System;

namespace TestD00
{
public class MemElem : IDisposable
{
bool dFlag;
int i;
public MemElem(int iKey)
{
dFlag = true;
i = iKey;
}

~MemElem() // Деструктор (он же Finalizer)
{
Console.WriteLine("{0} disposed : {1}!", i, (dFlag == true) ? "yes" : "no");
// Если ресурсы не освобождены – вызывается Dispose.
if (!dFlag) Dispose();
}
// Детерминированное управление ресурсом:
// имитация активности.
public void Dispose()
{
if (dFlag==true)
{
 Console.WriteLine("Dispose is here!");
 dFlag = false;
}
else
{
 Console.WriteLine("Dispose was here!");
}
}
}

class Program
```

```
{
static void Main(string[] args)
{
IDisposable d;
MemElem mmm = null;
int i;
for (i = 0; i < 10; i++)
{
MemElem m = new MemElem(i);
if (i == 5)
{
 d = m as IDisposable;
 if (d!=null) d.Dispose();
 mmm = m;
 GC.SuppressFinalize(mmm); // И закрыли для кода деструктора.
}
}

Console.WriteLine("-----0-----");
GC.Collect();
Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
// Заменить false на true. Осознать разницу.
Console.WriteLine("-----1-----");
// А теперь вновь разрешили его удалить...
// Пока существует хотя бы одна ссылка – GC все равно не будет
// уничтожать этот объект.
if (mmm != null) GC.ReRegisterForFinalize(mmm);
mmm = null; // Вот потеряли ссылку на объект.
// Теперь GC может уничтожить объект. Если разрешена финализация -
// будет выполнен код деструктора. Возможно, что отработает метод
// Dispose. Если финализация запрещена -
// код деструктора останется невыполненным.
Console.WriteLine("-----2-----");
GC.Collect();
// Заменить false на true. Осознать разницу.
Console.WriteLine("Total Memory: {0}", GC.GetTotalMemory(false));
Console.WriteLine("-----3-----");
}
}
}
```

Листинг 8.4.