www.phys.nsu.ru

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра автоматизации физико-технических исследований

К. Ю. МОКИН

C#. ПЛАТФОРМА .NET

Учебное пособие

Новосибирск
2009

Мокин К.Ю. C#. Платформа .NET: Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2009. 126 с.

Данное учебное пособие предназначено для студентов занимающихся углубленным изучением информационных технологий и языков программирования. Для изучения представленного материала необходимо знание принципов объектно-ориентированного программирования (ООП) и языка программирования C++, необходим опыт разработки и отладки приложений на C++ в рамках не менее годового практического курса. Опыт использования Visual Studio.

Представленный материал позволяет ознакомиться с идеологией платформы .NET и синтаксисом языка программирования C#. При изложении материала по языку C# будет делаться упор на схожесть и отличительные особенности синтаксических конструкций и идеологии в сравнении с языком C++. В материале так же будет представлен общий взгляд на разработку исполняемых приложений и библиотек и позволит еще раз взглянуть на процесс разработки приложений.

Рецензент
доц., зам. зав. кафедрой АФТИ ФФ НГУ М. Ю. Шадрин

Учебное пособие подготовлено в рамках реализации программы развития НИУ-НГУ на 2009-2018 гг.

© Новосибирский государственный
университет, 2009
© К. Ю. Мокин, 2009

1. ПЛАТФОРМА .NET 5

1.1 Процессор и процессорные инструкции 5
1.2 Роль операционной системы. 6
1.2.1 Процессы и потоки. 6
1.2.2 Менеджер памяти. 8
1.2.3 Запуск приложения. 10
1.3 Языки интерпретаторы. (Scripts) 14
1.4 Языки компиляторы. 16
1.5 Сравнение методик «Интерпретатора» и «Компилятора». 17
1.6 Симбиоз методик построения кода. IL-код. 19
1.7 Сборки (Assemblies) 22
1.8. Языки .NET и библиотеки типов. 25
1.9 Блоки .NET. (CLR, CTS, CLS) 29

1. C#. ОСНОВЫ ИДЕОЛОГИИ. 31

2.1 Среда разработки Microsoft Visual Studio. 32
2.2 Структура кода. Пространства имен. 36
2.3 Встроенные типы данных. 39
2.4 Базовый тип Object. 41
2.5 Структурные и ссылочные типы. 42
2.5.1 Умные указатели. 42
2.5.2 Структурный и ссылочный тип. 46
2.6 Упаковка и распаковка 51
2.7 Автоматическое освобождение памяти. 52
2.8 Освобождение ресурсов и деструкторы. 57

3. C#. СИНТАКСИЧЕСКИЕ КОНСТРУКЦИИ. 59

3.1 Операторы ветвления 59
3.2 Операторы (Арифметические, логические, сравнения, присваивания). 60
3.3 Операторы циклов. 61
3.4 Определение констант. 63
3.5 Пользовательские типы. 63
3.6 Поля. Свойства. Статические данные типов. 66
3.7 Методы и передача параметров. 69

4. C#. ООП 75

4.1 Наследование. 75
4.2 Конструкторы 77
4.3 Полиморфизм. 79
4.4 Запрещение наследования. 84
4.5 Абстрактные методы и классы 85

4.6 Разрывание полиморфизма, свойств и методов. 85
4.7 Интерфейсы. 88

5. C#. ДЕЛЕГАТЫ И СОБЫТИЯ. 97

5.1 Делегаты. 97
5.2 События. 102

6. RTTI. РЕФЛЕКСИЯ. АТРИБУТЫ. 106

6.1 Приведение типов. RTTI. 106
6.2 Рефлексия 111
6.3 Атрибуты. 114
6.4 Использование рефлексии и атрибутов. Пример «Страница свойств». 117

1. Платформа .NET

В данном разделе будет рассмотрен вопрос различных идеологий создания и исполнения программного кода на вычислительных машинах. Будет описан комбинированный подход исполнения бинарного кода, используемый в идеологии .NET. Так же будет короткий экскурс роли операционной системы в запуске приложений.

1.1 Процессор и процессорные инструкции.

Не секрет, что для исполнения любой программы нужен некий вычислительный комплекс, содержащий одним из основных компонентов процессор. При работе приложения, процессор получает на вход последовательность процессорных инструкций и данных и производит необходимые действия. Этот процесс и называется ***исполнением программы***.

Так или иначе, конечным результатом создания программы является последовательность инструкций предназначенных для данной разновидности процессоров. Архитектурой процессора определяется поддерживаемый ***набор процессорных инструкций***. Если различные процессоры поддерживают одинаковый набор инструкций и внешний “железный” интерфейс, то они называются совместимыми, при том что их реальная внутренняя архитектура может отличаться.

Процессорные инструкции в вычислительном комплексе представляются, как и все остальные данные, в бинарном

виде. Именно поэтому последовательность процессорных инструкций называется ***исполняемым бинарным кодом***.

Файл, в котором хранится исполняемый бинарный код называется ***исполняемым файлом*** (как правило, имеет расширение ***.exe*** или ***.dll***).

1.2 Роль операционной системы.

Одной из функций операционной системы (ОС) является запуск исполняемых файлов. В целом, операционная система сама является исполняемым бинарным кодом, т.е. исполняемой программой. В частности, в первых операционных системах (DOS) при запуске исполняемого файла (.exe), исполнение кода операционной системы прекращалось, и на процессор поставлялся поток процессорных инструкций из запущенного файла. После того как исполнение инструкций заканчивалось, управление передавалось обратно ОС. Недостатками такой организации операционной системы были невозможность параллельного исполнения нескольких программ и отсутствие защищенности от сбоев. Возникновение ошибки при исполнении программы приводило к необходимости перезапуска вычислительной машины, и происходила утеря всех несохраненных данных.

Следующие поколения операционных систем стали делить между запущенными программами ресурсы вычислительной машины. Основные ресурсы, которые приходится делить между одновременно исполняемыми приложениями это ***оперативная память*** и ***процессорное время***.

1.2.1 Процессы и потоки.

При запуске исполняемого файла (приложения) операционная система создает ***процесс***. ОС содержит список

запущенных процессов. Каждый процесс содержит поток исполняемых инструкций. ОС по очереди посылает часть процессорных инструкций на исполнении из одного процесса, после чего переключается на следующий, что и называется ***переключением контекстов между процессами***.

Внутри процесса (исполняемого приложения) существуют ***потоки***. Поток - это последовательность исполняемых инструкций внутри приложения(процесса). Логично, что любое приложение содержит как минимум один поток. Если приложение не использует несколько потоков, то оно называется ***однопоточным (single-threaded)***. В тех случаях, когда некоторые операции занимают значительное время, основное приложение ***«зависает»*** и не реагирует на действия пользователя. (Думаю, всем приходилось видеть заголовок «Программа не отвечает», «Application not respond»). Иногда такое сообщение предвещает ошибку, что основной поток приложения ушел в бесконечное выполнение каких-то операций или ожидания чего-либо… В ряде случаев приложение немного ***«повисев»*** продолжает свою корректную работу. При подключении к серверу или длительном алгоритме вычисления внутри приложения бывает необходимым запустить часть исполняемого кода в отдельном потоке. При этом основное приложение не зависает, и как правило сигнализирует пользователю, что процесс идет… Например, при скачивании файла гораздо приятней наблюдать прогресс бар и кол-во скачанных процентов, оставшегося и прошедшего времени и скорости скачивания, нежели лицезреть «зависшее» окошко с надписью «Not Respond». Приложения, которые исполняют параллельно две последовательности процессорных инструкций (потоков), называются ***многопоточными (multi-threaded)***.

Одним из основных предназначений организации процессов и потоков является разделение процессорного

времени между операционной системой и исполняемыми приложениями.

1.2.2 Менеджер памяти.

При совместной работе нескольких приложений существует еще один ресурс, который необходимо разделять между процессами. Это ***пространство памяти***.

Адресное пространство – это максимальный объем памяти, который может быть напрямую использован системой/приложением. Адресное пространство определяется ***разрядностью операционной системы***, а совсем не тем, сколько планок динамической памяти встроено конкретно в вашем ПК. Для 32-разрядной системы в шестнадцатеричном представлении память адресуется в диапазоне [0x00000000] – [0xFFFFFFFF]. (32/4 = 8 шестнадцатеричных символов). Максимально адресуемая память в 32-разрядной системе составляет 4Гб. Было время (<= ~1990 год), когда были 16-разрядные операционные системы, и они могли адресовать памяти меньше, чем сейчас занимает обычная фотография с цифрового фотоаппарата. Когда ввели 32-разрядную систему, казалось, что адресного пространства хватит «навсегда». Но уже существуют 64-разрядные версии операционных систем и начался «болезненный» переход на них с 32-разрядных систем, а почти все ноутбуки и ПК идут в продаже с 3-4Гб оперативной памяти. Теоретически 64-разрядная система может адресовать 16 миллиардов гигабайт памяти. Пока что на практике 128Гб. Сначала появились высокоточные фотографии, потом видео с большим расширением, а возможно в ближайшем будущем наш ждет 3D видео в «хорошем качестве», тогда сегодняшний фильм в 1Гб станет занимать 1 терабайт.

www.phys.nsu.ru

Вернемся к проблеме выделения памяти приложению. При исполнении приложения требуется два вида памяти. ***Программная память*** и ***память данных***.

В программную память загружается и хранится исполняемый бинарный код. Т.к. бинарный код нелинейный и различные ***фрагменты кода*** могут исполняться большое количество раз или параллельно исполняться в нескольких потоках, то его загружают в память и ОС использует этот код как поток процессорных инструкций внутри процессов и потоков.

К памяти данных относится память, выделяемая для хранения переменных и создаваемых в процессе работы данных. Память данных состоит из двух частей: ***стек/stack*** и ***динамическая память (куча/heap)***. Для того чтобы быть хорошим программистом, необходимо четко представлять принцип работы и назначение этих двух видов памяти. Предполагается, что читатель познакомился с этой темой при изучении курса С++. В дальнейшем, при обсуждении области видимости и времени жизни объектов эта тема будет затронута и еще раз кратко пояснен процесс жизни объектов на стеке и в куче.

При запуске приложения операционной системой, необходимо выделить кусок адресного пространства для создаваемого процесса. При работе приложения необходимо выделять память в динамической области памяти (на куче). Если сами процессы будут заниматься выделением памяти, то они могут начать конфликтовать между собой.

В общем случае необходим ***менеджер памяти***, который бы занимался выделением памяти для процессов и контролем того, что приложение пытается обращаться, записывать

данные в память или освобождать память в разрешенном диапазоне адресного пространства.

На практике все немного сложней и описывать точное устройство политики разделения и контроля памяти в этом учебном пособии не будем. Если рассуждать в общем, может существовать несколько менеджеров памяти. Один менеджер памяти операционной системы, а другие менеджеры памяти внутри процессов. Менеджер памяти операционной системы осуществляет выделение диапазона адресного пространства для процесса и организует контроль за тем, чтобы во время исполнения процесса не происходило обращений и записи за пределы выделенного пространства. Менеджер памяти запущенного процесса контролирует выделение памяти в рамках выделенного адресного пространства, отслеживает занятые и свободные области адресного пространства, освобождает память и следит за тем, чтобы не происходило обращений и попыток записи в свободные области адресного пространства.

Если приложение пытается обратиться к памяти чужого процесса и тем более записать туда данные, то операционная система препятствует этому. Если в процессе/приложении произошла ошибка и вылетело ***необрабатываемое исключение***, то операционная система завершит процесс. Это обеспечивает сохранение работоспособности самой операционной системы и других исполняемых приложений.

1.2.3 Запуск приложения.

В самом начале главы о роли операционной системы было написано, что основная задача ОС – это запуск исполняемых файлов. Процессы, потоки, менеджер памяти созданы только для того, чтобы обеспечить параллельную и безопасную

www.phys.nsu.ru

работу нескольких приложений и их взаимодействие между собой.

Основные компоненты, отвечающие за запуск процессов, потоков и управления памятью и организации файловой структуры и доступа к файлам, называются ***ядром операционной системы***.

Другие встроенные в операционную систему компоненты, такие как отображение файловой структуры в «окошках», Far-е и утилиты-команды для копирования файлов - можно назвать интегрированными в ОС сервисами. Т.е. фактически это те же самые обычные приложения, запуск которых ничем не отличается от запуска любых других приложений.

Если рассмотреть запуск файла из окна файловой структуры, то можно описать его следующим образом: По двойному нажатию левой кнопки мыши на файл, происходит поиск ассоциированного в ОС приложения для этого расширения. К примеру, если выбрали *.doc файл, то загружается процесс для msword.exe и параметром передается файл, который следует открыть при загрузке MS Word-а. Если выбрали *.exe файл, то будет запущен процесс и приложение будет загружено и начнется его исполнение. Т.е. окно файловой структуры передает команду запуска операционной системе и на этом возвращается в ожидание последующего пользовательского ввода.

Ранее было сказано, что файлы с исполняемым бинарным кодом бывают двух типов ***.exe*** или ***.dll.*** Exe – это исполняемый файл приложения, при запуске которого в операционной системе создается новый процесс. Dll – это файл динамической библиотеки (Dynamic Link Library). Т.е. это файл, содержащий бинарный код некоторой библиотеки. Если один и тот же функционал или компонент надо использовать в

разных приложениях, то включать один и тот же бинарный код в различные *.exe приложения не обязательно. И может привести к серьезному росту размеров исполняемого файла.

В таком случае системные и общие DLL файлы располагают в путях поиска операционной системы (Например, в папке Windows\System).

При создании библиотек функционала и компонентов, которые будут использовать другие разработчики, так же необходимо создавать DLL файлы. В этом случае DLL файлы будут располагаться рядом с exe файлом приложения или в строго отведенных для этого местах, определяемых разработчиками приложения.

При создании программных расширений (AddIn-ов) для существующих контейнерных приложений так же необходимо создавать DLL файлы, которые будут загружаться в это ***контейнерное приложение***. Например, можно для CAD программы написать DLL-расширение, в котором будет описан объект «звезда». Если из CAD приложения вы загрузите это DLL-расширение, то появится возможность создавать «звезды». Именно поэтому такие исполняемые DLL файлы и называются ***расширениями***, т.к. добавляют в существующее контейнерное приложение новый функционал.

При разработке большого приложения группой разработчиков удобно и целесообразно делить весь проект (Solution/Решение) на отдельные проекты. Где будет один основной *.exe файл с контейнерным приложением, а остальные логические компоненты функционала будут представляться в виде DLL файлов. Например, компонент вывода документов в Word, Excel можно поместить в библиотеку «ReportGenerator.dll» и посадить разрабатывать этот компонент отдельного разработчика. Предварительно

необходимо определить лишь нужные интерфейсы, которые будут доступны из этого DLL-компонента.

По опыту преподавания курсов C++ и C# автор может сказать, что многие студенты не до конца понимают всю значимость DLL файлов. И в лучшем случае воспринимают DLL как исполняемые файлы общих библиотек и общих компонентов. При создании небольших учебных проектов практически никто не использует возможность разбития своего решения (Solution) на логические проекты (Projects). Да, в большинстве случаев проекты небольшие и делить их может быть и не стоит свеч, но автор настоятельно рекомендует это делать, хотя бы для учебных целей и получения опыта и знаний.

В общем случае приложение представляется одним exe исполняемым файлом (контейнерным приложением) и набором dll файлов с реализацией общего или отделенного функционала. Даже при создании одного exe файла с включенной опцией «Dynamic Linkage» для MFC или какой-нибудь еще библиотеки для запуска exe файла необходимо наличие на рабочей машине необходимых dll файлов. Часто они устанавливаются вместе с операционной системой или другими приложениями.

Набор DLL файлов, от которых зависит exe файл, называется «***зависимостями***» или «***dependencices***». В свою очередь DLL файл может зависеть от других DLL файлов.

При загрузке *.exe файла, операционная система создает процесс и перед загрузкой исполняемого файла, необходимо загрузить набор зависимых dll файлов строго по иерархии зависимостей. При этом используется ***механизм поиска dll файлов***. Сначала операционная система ищет нужный файл в текущей рабочей папке (как правило, та из которой запущен

exe файл), а потом перебирает пути поиска заданные в операционной системе или среди зарегистрированных dll компонентов. Если нужный файл не найден, то операционная система выдает ошибку, что приложение не может быть загружено т.к. не найдены нужные компоненты и процесс убивается.

Когда все зависимые DLL и исполняемый файл загрузились, происходит ***позднее динамическое связывание***. Если A.exe использует функцию из B.dll, то при загрузке A.exe в бинарном коде стоит «заглушка», что в этом месте надо позвать метод f(). А при загрузке B.dll определяется по какому адресу загружен метод f(). Позднее динамическое связывание выполняет подстановку реального адреса вызываемого метода вместо «заглушки».

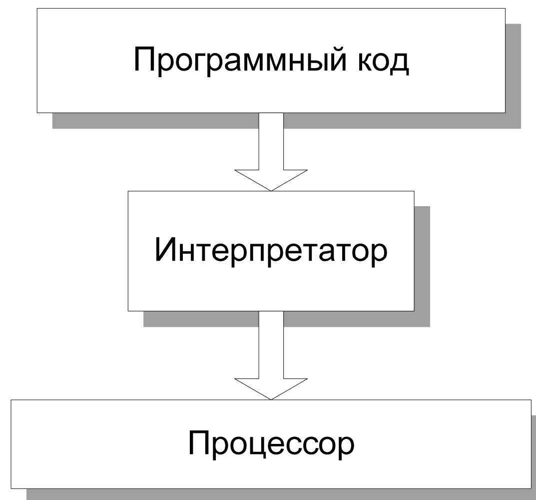
Не зря процессу загрузки приложения состоящего из набора исполняемых бинарных файлов уделено столько времени. В платформе .NET от этого никуда не ушли и более того, все множество библиотек представлены именно dll файлами. Для понимания процесса подключения библиотек хоть в C++, хоть в C# эту тему надо понимать в достаточной мере.

1.3 Языки интерпретаторы. (Scripts)

Язык программирования это средство создания бинарного кода на высокоуровневом синтаксисе. ***Синтаксис языка*** – это набор синтаксических конструкций, который позволяет создать исполняемый бинарный код. Текст программы, заданный с использованием синтаксических конструкций, называется ***программным кодом***.

Существует два основных метода преобразования программного кода в исполняемый бинарный код. ***Интерпретирование*** и ***компилирование***.

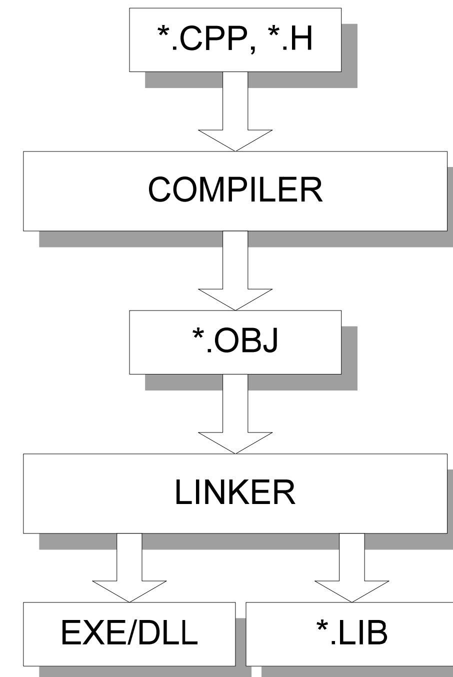
Лучшим пример языков интерпретаторов являются встроенные в приложения скрипты (scripts).



Разбор синтаксических конструкций и их преобразование в бинарный код происходит с участием приложения «Интерпретатор». Фактически, это приложение само является бинарным кодом. Уже в самом этом приложении существуют бинарные фрагменты кода для различных синтаксических конструкций языка. Интерпретатор выделяет данные из программного кода и подставляет их в свои бинарные конструкции и исполняет их на процессоре.

1.4 Языки компиляторы.

Второй, наиболее распространенный, способ перевода программных кодов в бинарный исполняемый код это языки компиляторы. Для преобразования синтаксиса языка используется ***компилятор*** и ***линкер***. На примере языка С++ построении бинарного кода происходит по следующей схеме:



На вход компилятора подаются файлы с исходным кодом. Компилятор производит анализ корректности синтаксических конструкций, условий построения программного кода. После чего преобразует высокоуровневые синтаксические конструкции во фрагменты бинарного кода. Для каждого *.cpp файла создается *.obj файл, который содержит бинарные

фрагменты кода с «заглушками» для вызова методов реализованных в других obj/cpp файлах.

После этого линкер производит объединение фрагментов кода из множества *.obj файлов в один файл и подменяет «заглушки» вызовов функций реальными адресами в DLL/EXE файле. Так же линкер производит lib файл, на основе которого в C++ строятся библиотеки.

1.5 Сравнение методик «Интерпретатора» и «Компилятора».

Методика интерпретатора имеет два существенных недостатка. Проверка корректности исходного кода осуществляется в момент исполнения кода интерпретатором (хотя можно создать и отдельный анализатор синтаксиса). Второй недостаток в скорости исполнения кода т.к. приходится интерпретировать поступающий текстовой поток информации, выделять из него данные и искать нужные для исполнения конструкции.

Методика компилятора лишена вышеописанных недостатков. Код сразу предоставлен в бинарном виде. Большую часть синтаксических ошибок можно выявить на этапе компиляции и линковки приложения.

Вспоминаем первую главу про процессорные инструкции. Набор процессорных инструкций определяется архитектурой процессора. Соответственно код собранный для конкретного процессора будет исполняться только на нем и на совместимых процессорах.

В языке интерпретаторе потенциально возможно написать для каждого архитектурного типа процессора свой интерпретатор. Что даст нам возможность исполнять один и

тот же программный код на вычислительных машинах разной архитектуры.

Более того, компилятор, как несложно догадаться, сам является исполняемым приложением. Его разрабатывает команда программистов. Программистам приходится вникать в различные нюансы архитектуры процессоров для того, чтоб построить оптимальный код. Совместимость бывает не полной. Т.е. один процессор может иметь одну команду-инструкцию, которая на другом процессоре решается в несколько команд-инструкций. При генерации бинарного кода приходится учитывать все это. Использовать пересечение множества команд-инструкций. Либо делать условные вставки времени выполнения. В настройках проекта для компилятора есть множество параметров, которые указывают компилятору «как собрать код». В частности можно выставить галочки «Производить оптимизацию для архитектуры Intel или AMD».

Компиляторы объектно-ориентированных языков являются крайне сложными приложениями. Так разработчикам компилятора приходится разбираться со всеми этапами, от анализа синтаксиса до построения бинарного кода под конкретную “железку”. Как правило производители процессоров, кроме Assemblera (который является архитектурно-зависимым языком по определению т.к. предполагает знание всех команд, реестров и т.п.), предоставляют компилятор для языка C. Программисты частично избавляется от необходимости глубоко вникать в архитектуру процессора. Но производитель процессора вынужден создавать компилятор для этого вида процессоров.

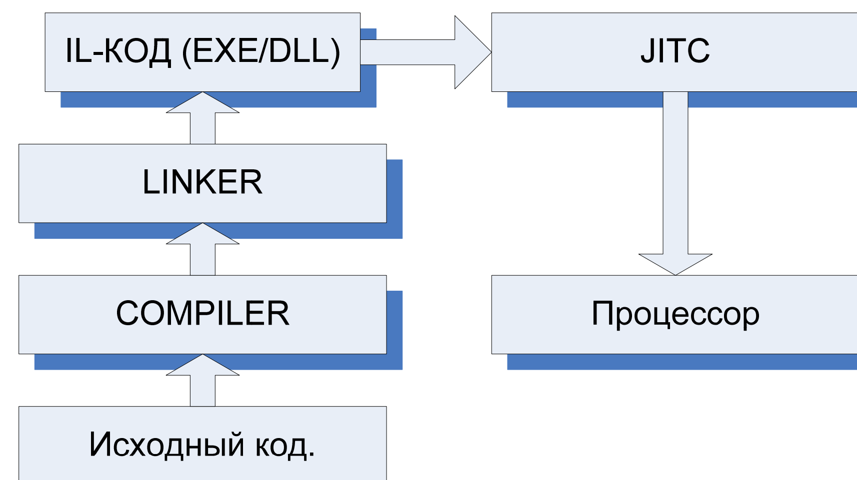
Зависимость от архитектуры “железа” процессора является недостатком методики построения бинарного кода.

1.6 Симбиоз методик построения кода. IL-код.

Для того, чтобы отвязаться от набора конкретных процессорных команд, которые определяются "архитектурой железа", можно ввести понятие ***виртуальной машины***. Это понятие получило широкое распространение с языком Java. Определяется (стандартизуется) набор виртуальных команд. Набор этих виртуальных команд образует исполняемый код для виртуальной машины. Для каждой машины и операционной системы можно написать программу-интерпретатор, которая будет преобразовывать поток виртуальных команд в конкретные процессорные инструкции. Эта концепция была полномасштабно реализована в языке Java. Именно поэтому этот язык получил широкое распространение для создания различных web-приложений и сетевых сервисов т.к. один и тот же код исполнялся на любых машинах с разными операционными системами.

При этом исходный код программы пишется на языке высокого уровня с полноценной поддержкой объектно-ориентированного программирования. Для получения исполняемого кода в виде виртуальных команд для виртуальной машины используются компилятор и линкер. Т.е. разработчики компилятора и линкера для такого языка должны следовать спецификации (стандарту), определенной для виртуальной машины. В этом стандарте все прописано четко и однозначно и нет необходимости вникать в "архитектурную реализацию железа" и нюансы исполнения бинарного кода на совместимых процессорных архитектурах.

Разработчики интерпретатора для конкретной архитектуры машины и операционной системы так же используют эту спецификацию (стандарт). И они уже не привязаны к синтаксису языка высокого уровня. Они преобразуют поток виртуальных команд в процессорные команды.



Рассмотрим еще раз эту модель построения и исполнения кода. Далее будем пользоваться терминологией принятой в технологии .NET.

С использованием высокоуровневых синтаксических конструкций языка создается исходный код. Этот код компилируется и линкуется в ***IL-код. [Intermediate Language]***. Это уже исполняемый код в виде IL инструкций. Т.е. код на промежуточном языке для виртуальной машины .NET. Исполняемые файлы с IL-кодом также имеют расширения *.exe и *.dll.

При запуске этих файлов операционная система создает процесс, и запускается компилятор времени выполнения **JITC** (**Just-In-Time Compiler**). Компилятор времени выполнения из IL-кода создает процессорный код. Из названия компилятора ясно, что происходит это непосредственно в момент исполнения кода. Построенный процессорный код исполняется на реальном "железе" (процессоре).

Вспомним два недостатка методики интерпретатора (или скриптовых языков). Это проверка корректности исходного кода и затраты на перевод синтаксических конструкций в бинарный код.

Проблема проверки синтаксиса, правил построения кода и его связывания полностью решена наличием компилятора и линкера, которые переводят исходный код с высокоуровневых синтаксических конструкций в IL-код. Процесс создания и отладки кода .NET ничем не отличается от привычного процесса создания и отладки кода на языке C++. И в обоих случаях мы получаем исполняемые файлы (exe или dll).

Рассмотрим вторую проблему быстродействия. Следует отметить, что теперь нам надо перевести в процессорный код последовательность IL-инструкций. Это означает, что уже не надо анализировать синтаксические конструкции исходного кода. Именно на это тратится большая часть времени. Т.е. компилятор уже выполнил большую часть работы. Грубо говоря, теперь остается лишь брать IL-инструкцию и подменять ее соответствующим блоком процессорных команд.

Вторая особенность, которая позволяет решить проблему быстродействия - это кэширование откомпилированного кода. Т.е. код единожды переведенный из IL-инструкций в процессорный код запоминается и при повторном исполнении не происходит повторной компиляции. Структура кода приложения такова, что ядро приложения это многократно исполняющийся код (циклы). И в приложении достаточно мало кода, который при длительной работе приложения исполняется только один или несколько раз. Как правило, код критичный к быстродействию – это небольшие алгоритмы и их фрагменты, многократно исполняемые в циклах. Т.е. один и тот же код исполняется миллионы раз.

Методика разработки, отладки и выполнения кода, реализованная в технологии .NET, включает в себя все преимущества методик «интерпретатора» и «компилятора» в чистом виде и лишена всех их недостатков.

1.7 Сборки (Assemblies).

В начале этой главы определим несколько терминов, принятых в .NET.

Исполняемый бинарный код, содержащий процессорные инструкции для исполнений на конкретной архитектуре «железа», называется ***неуправляемым кодом (Unmanaged code)***.

Исполняемый .NET код в виде IL-инструкций называется ***управляемым кодом. (Managed code)***.

Исполняемый exe/dll файл, содержащий неуправляемый код, называется ***Unmanaged exe/dll*** файл.

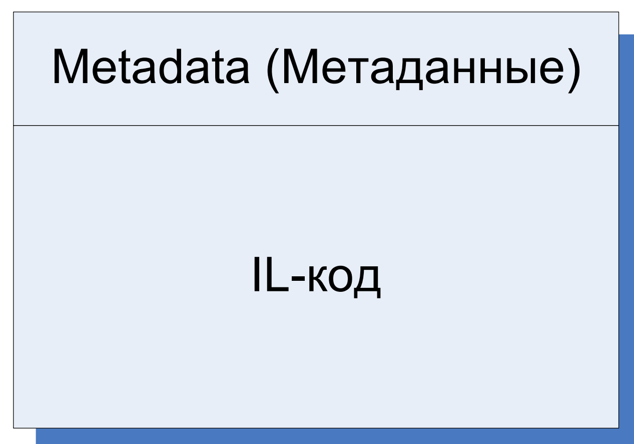
Исполняемый exe/dll файл, содержащий управляемый код в виде IL-инструкций, называется ***Managed exe/dll*** файл.

Так же Managed DLL файл, содержащий библиотеку типов, называется ***сборкой (Assembly)***.

В главе про запуск приложения достаточно подробно обсуждалась модель приложения с одним exe файлом и набором динамических расширений dll файлов. Исполняемый код может быть размешен в нескольких dll файлах, которые могут быть подгружены в процесс по мере необходимости. Для этого unmanaged DLL кроме неуправляемого кода содержат информацию по экспортируемым функциям и

типам. И при загрузке кода в процесс из нескольких исполняемых файлов происходит позднее динамическое связывание. Т.е. вызов функции из кода одного файла подменяется конкретным адресом этой функции после загрузки необходимого DLL файла.

Рассмотрим структуру сборки (Assembly) содержащей управляемый код:



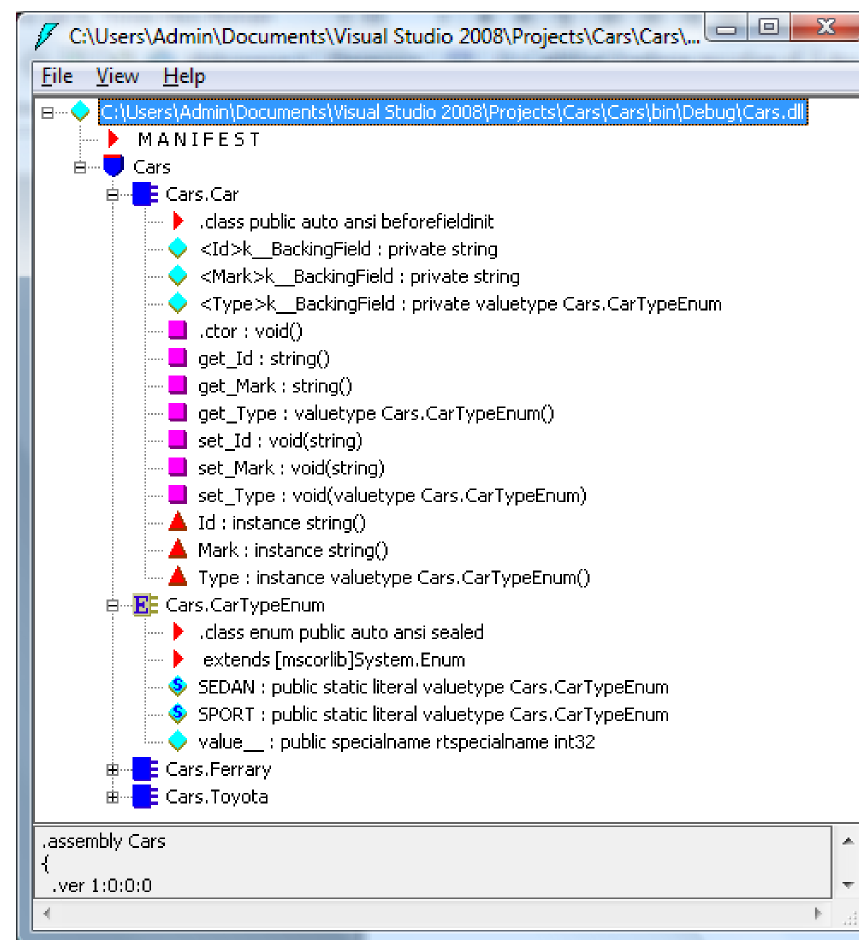
Сборка состоит из метаданных и, собственно, исполняемого IL-кода. То что IL-код это исполняемый код для виртуальной .NET машины, мы уже разобрались, то теперь необходимо разобраться с ролью метаданных.

Метаданные (Metadata) это исчерпывающая информация по всем типам данных содержащихся в сборке. И дополнительная служебная информация по сборке (версия, уникальный ключ, компания разработчик, авторские права и т.п.).

Проводя аналогию с языком C++, где есть .h файлы, которые полностью описывают используемые типы данных и

функции, эта информация полностью сохраняется в метаданных. Используя метаданные можно посмотреть исчерпывающую информацию по типам, содержащихся в сборке (Managed файле).

Для этого есть специальная утилита ILDasm.exe. (Intermediate Language Disaassembler). Она позволяет выбрать управляемый dll или exe файл и просмотреть его содержимое.

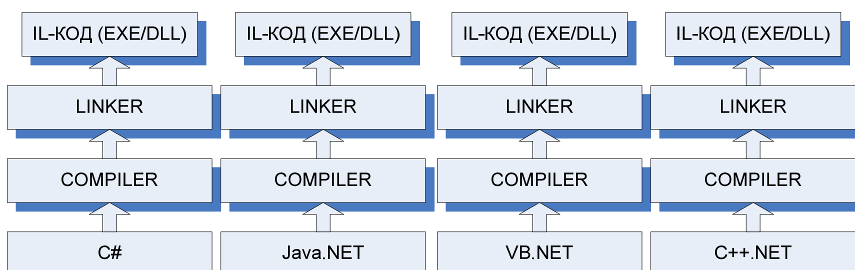


В этой утилите можно видеть имена типов, их свойства, поля, методы, конструкторы. Параметры функций и возвращаемые значения, их типы. По двойному клику на метод или свойство можно просмотреть их IL-код реализации, который очень сильно похож на ассемблер (по сути, он им и является для виртуальной машины).

```
Cars.Car::set_Type : void(valuetype Cars.CarTypeEnum)
Find  Find Next
.method public hidebysig specialname instance void
        set_Type(valuetype Cars.CarTypeEnum 'value') cil managed
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGe
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  stfld      valuetype Cars.CarTypeEnum Cars.Car::'<Type>k__Backin
  IL_0007:  ret
} // end of method Car::set_Type
```

1.8. Языки .NET и библиотеки типов.

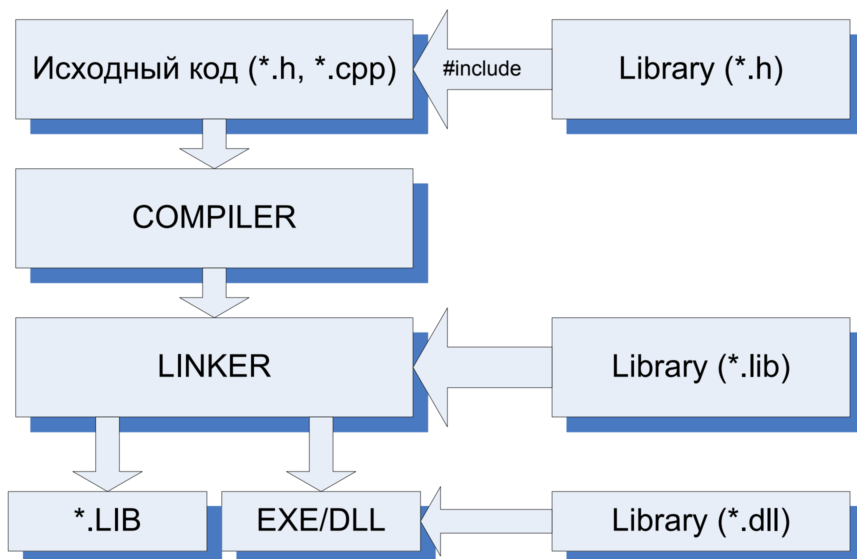
Для получения Managed exe/dll файлов можно использовать синтаксис любого языка. Для этого необходимо реализовать соответствующий компилятор и линкер в IL-код. Практически для всех языков программирования существуют их аналоги, которые получают приставку .NET.



Для платформы .NET разработан специальный язык получивший название ***C#***. На этом языке разрабатываются только «managed» приложения и компоненты. Свое название он получил, видимо, по той причине что его синтаксис очень сильно напоминает синтаксис языка C++. Так же в него органично внесены некоторые удобные концепции и конструкции, позаимствованные из других языков. В дальнейшем мы сконцентрируемся на схожести и различиях между языком C++ и C#.

Но для любого языка необходимы библиотеки. Библиотеки работы с файлами, потоками, строками, массивами, списками, графами, оконным интерфейсом и т.д. Например, для языка C++ можно выделить пару известных библиотек (STL, MFC), кроме непосредственно родных библиотек (ввода/вывода, графики и т.п.). По мере развития языка, он обрастает все большим количеством библиотечных функций и типов, использование которых существенно ускоряет процесс создания приложения. Так же любой язык высокого уровня позволяет

Теперь зададимся вопросом: в каком виде представлены эти библиотеки в C++ и в C#? Как создать свою библиотеку? В языке C++ эту роль выполняют .h файлы, и .lib файлы. (*.h) файлы содержат описание пользовательских типов и функционала, *.lib файлы содержат бинарную реализацию тел функций из .cpp файлов. Заголовочные файлы нужны для создания исходного кода с использованием библиотечных типов данных и функционала. Фактически используются на стадии компиляции. А для линковщика нужны *.lib файлы, чтобы достать из них бинарный код и вставить в создаваемый exe/dll файл. Для работы созданного exe/dll файла (при динамической линковке) понадобится dll файл библиотеки. Т.е. если изобразить графически, то библиотека на C++ это заголовочные .h файлы, набор *.lib файлов и dll файлы.



При использовании сторонних библиотек, вы получаете свой набор .h файлов, *.lib файл и *.dll файл. Соответственно, вы получили свою библиотеку, которую можно использовать в других проектах. На языке C++ было принято раскладывать библиотеки по папкам «***Include***», «***Lib***» и «***Bin***».

В прошлой главе было сказано, что теперь «Managed DLL» (сборка) содержит метаданные, которые полностью описывают типы данных из этого файла и содержат IL-код реализации. Поэтому в .NET языках, и в C# в частности, ***сборка(Assembly) является полноценной библиотекой***.

При разработке кода на языке C# необходимо включить в проект ссылку (***Reference***) на сборку (Managed dll) с библиотекой типов (Class Library). Все публичные «public» типы из этой сборки станут доступны в текущем проекте.

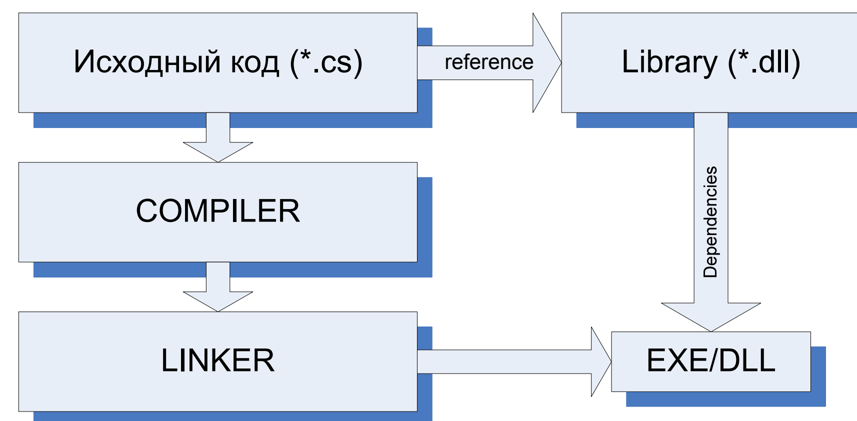


Схема включения библиотек в C# немного прозрачней и проще. Необходимо в проекте поставить ссылки на набор библиотечных dll файлов.

Стоит отметить, что набор библиотечных типов встроенный в .NET огромен. Полностью в базовую библиотеку включен аналог STL (System.Collections), работа с графикой (GDI+), XML, Windows.Forms и многое другое.

Тут стоит рассмотреть один интересный момент. Если на языке C++ вы выпустили коммерческое приложение, то все исходные коды и библиотеки остаются у вас. Пользователям становятся доступны лишь бинарные файлы. И исправить их или использовать что-либо в них крайне затруднительно. При разработке .NET приложений, вы отдаете исполняемые файлы (особенно ваши dll файлы), которые параллельно являются библиотеками. По аналогии с C++, это аналогично передаче набора «*.h» и «*.lib» файлов в третьи руки.

На эту проблему можно смотреть с двух точек. Во-первых, если библиотека большая, то в ней еще надо разобраться. Особенно, если к типам и методам не предусмотрено встроенное документирование. Иногда сами разработчики

начинают путаться в своих библиотеках. Во-вторых, если все же очень сильно не хочется отдавать библиотеку в третьи руки и она представляет какое-то коммерческое «ноу-хау», то можно использовать ***дотфускатор***, который заменит в метаданных смысловые названия типов и методов на случайные наборы слов. В IL-код встроит пустые циклы и операторы ветвления. Фактически это запутывание и обезличивание библиотеки. Использовать ее становится практически невозможно.

Еще одной особенностью того, что сборка является библиотекой, является возможность использования библиотек в коде использующих разные языки. Например, можно написать сборку на VB.NET и включить ее в проект на C#. Поскольку после компиляции в IL-код и создания метаданных, информация на каком языке была разработана сборка теряется. В .NET поддерживается межъязыковое наследование, межъязыковое взаимодействие и межъязыковая обработка исключений.

1.9 Блоки .NET. (CLR, CTS, CLS)

В этой главе мы подведем итог, что же такое платформа .NET. Со всеми ее основными элементами и концепциями в отдельности мы уже в той или иной мере познакомились в предыдущих главах.

Технология .NET состоит из трех основных блоков. CLR (Common Language Runtime), CTS (Common Type System), CLS (Common Language Specification).

CLR – Общая среда выполнения. При установке Framework .NET SDK в операционную систему интегрируется функционал, который отвечает за запуск Managed приложений, компиляции времени выполнения, запускает

особым образом процессы и потоки в операционной системе. Программно реализует виртуальную машину, в силу чего и получил такое название.

CTS – Общая библиотека типов. При установке Framework .NET SDK на персональную машину в специальное место устанавливается обширный набор библиотек в виде dll файлов. Эти библиотеки содержат все от определения примитивных типов (int, double, string…) до специализированных библиотек по работе с графикой, базами данных, взаимодействию с COM и т.д.

CLS – Общая языковая спецификация. Это набор документации (стандартов) для среды .NET. При разработке компиляторов или среды исполнения под другие платформы и операционные системы, этот набор правил и стандартов жестко определяет “виртуальную машину” и определяет способы работы .NET среды в ней.

На этом закончим знакомство с платформой .NET и перейдем к изучению языка C#.

1. C#. Основы идеологии.

Язык C# специально разработан для среды .NET. Синтаксис этого языка похож на синтаксис языков Java и C++. В языках C# и в Java определения классов состоит из одного файла, тогда как в C++ есть заголовочные <*.h> файлы и файлы реализации <*.cpp>.

В целом, синтаксис операторов ветвления и циклов полностью идентичен языку C++. Так же в язык вошло понятие «свойств» (Properties), которые используются в языке Visual Basic.

Язык C# можно рассматривать как симбиоз различных языков программирования, где собраны наиболее удачные синтаксические конструкции и идеологические концепции.

Кратко перечислим основные отличия от языка C++.

- Определение типа (кдасса) происходит в одном (*.cs) файле.
- Из языка убрано понятие указателей. Введено понятие ссылок.
- Автоматическое освобождение памяти. Оператор delete убран из синтаксиса языка. Не надо пунктуально отслеживать жизненный цикл объекта.
- Реализована программная поддержка интерфейсов.
- Все типы данных (в том числе встроенные Int, Double и т.д.) имеют базовый класс Object.
- Убрана возможность множественного наследования по классам. При этом множественное наследование по интерфейсам поддерживается.

- Возможность получения и использование информации по типам данных. Рефлексия. (Reflection).
- В язык встроены специальные конструкции для обработки событий и передачи функций в виде объектов. Делегаты и События (Delegates & Events).
- Поддержка аспектно-ориентированных методик разработки программ. Атрибуты.

Далее, мы рассмотрим синтаксис языка и уделим особое внимание новым идеологическим концепциям и синтаксическим конструкциям.

2.1 Среда разработки Microsoft Visual Studio.

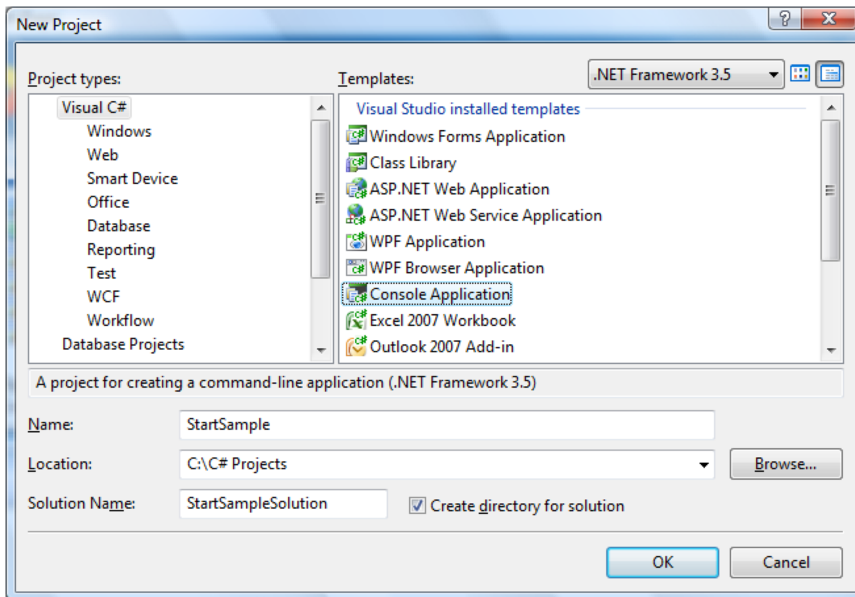
Для разработки приложений на C# в этом учебном пособии мы будем использовать среду разработки Microsoft Visual Studio 2008. В этом учебном пособии не ставится задача обучению разработки кода в Visual Studio. Подразумевается, что читатель познакомился со средой и принципами отладки кода при изучении языка программирования C++. Для языка C# все полностью аналогично. В этой главе лишь будет продемонстрирован минимальный набор шагов для того, чтобы создать новый проект на языке C#.

При первом запуске Visual Studio необходимо выбрать основной язык C#.

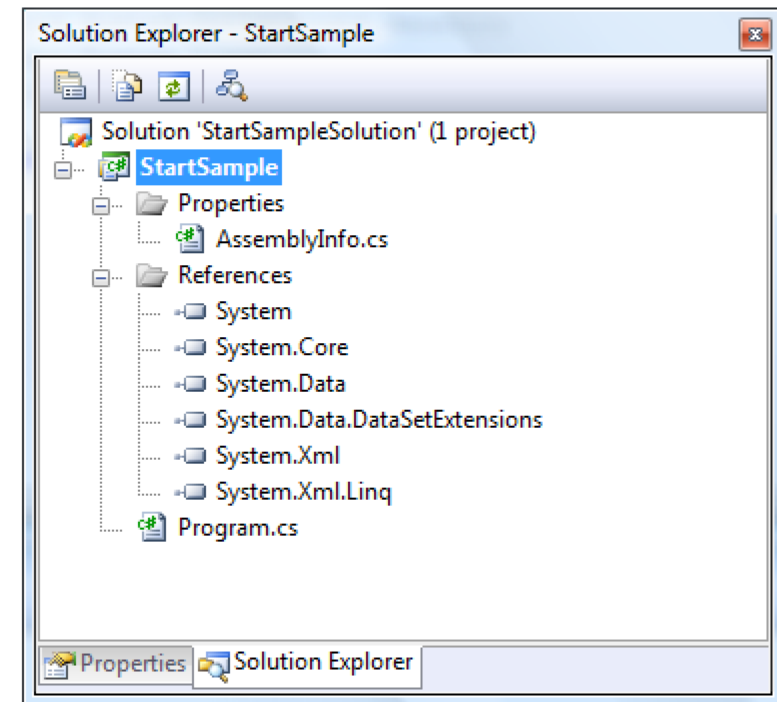
Используя меню «File->New->Project» запускаем окно создания нового проекта. В этом окне предлагается выбрать тип создаваемого приложения. Акцентирую внимание лишь на трех возможных вариантах. Консольное приложение, оконное приложение и библиотека классов. Первые два типа приложений позволяют создавать исполняемы exe приложения. В первом случае, это консольное приложение, во

втором случае, это оконное приложение, использующее библиотеку типов Windows.Forms. Третий тип приложения позволяет создавать динамические библиотеки (dll файлы).

В дальнейшем, будет рассмотрен пример создания решения (Solution) состоящего из нескольких проектов. Но для начала мы будем использовать консольное приложения для изучения основ языка C#. Выбираем тип нового проекта консольное приложение и начинаем знакомство с языком и средой.



Создадим консольный проект «StartSample». Откроем панель решения (Solution) и ознакомимся со структурой проекта. Если панель не открылось сразу после создания проекта, то ее можно вызвать из меню. «View->Solution Explorer».



Рассмотрим структуру проекта. Сразу заострим внимание, что верхним уровнем структуры является не проект, а ***решение (Solution)***. При создании проекта я специально дал ему отличное от проекта имя «StartSampleSolution». Решение может содержать в себе несколько проектов. Пока что в нашем случае в решении только один проект.

Сам проект объединяет набор ***(*.cs) файлов***, набор ***ссылок (References)*** на библиотечные сборки и содержит ***настройки проекта.***

В нашем примере проект ссылается на библиотечные файлы «System», «System.Core», «System.Data» и т.д. В этих библиотеках определяются все базовые и основные типы платформы .NET.

В каждом проекте есть особенный файл «AssemblyInfo.cs». В нем содержится служебная информация по сборке. Через атрибуты сборки прописана информация к сборке, уникальный номер сборки «Guid» и номер ее версии.

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("StartSample")]
[assembly: AssemblyDescription("Sample application")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("StartSample")]
[assembly: AssemblyCopyright("Copyright ©  2009")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]

[assembly: Guid("4ce9bb9a-30a9-4b01-887d-
e521d8fa7649")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Во всех остальных *.cs файлах находится исходный код приложения. Теперь перейдем непосредственно к изучению синтаксиса языка C#.

2.2 Структура кода. Пространства имен.

Рассмотрим код из файла «Program.cs» нашего примера. Допишем туда вывод в консоль.

```
using System;

namespace StartSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            string userInput = Console.ReadLine();
        }
    }
}
```

Как ранее было отмечено, код и декларации и реализации пишутся вместе в одном фале. Язык стал 100% объектным. Теперь все функции должны находится внутри объявления типов (класса). Для консольного приложения в одном из классов должен быть статический метод Main, который является точкой входа в приложение.

В языке C++ управление включаемыми библиотечными и пользовательскими типами осуществлялось через директиву включения (*.h) файлов. (#include). В C#, выполняя подключение через ссылку на сборку (dll файл) в проекте, мы делаем доступными все типы, определенные в библиотечном файле. Так же нам доступны все типы, определенные в своей сборке. В общем случае это сотни и сотни типов (классы, интерфейсы, события). При отсутствии какой-либо системы упорядочивания этих типов, было бы очень проблематично работать в такой системе.

Для упорядочивания множества типов в группы используются ***пространства имен (namespaces)***. Типы в одном пространстве имен образуют логическую группу. Например, все типы для работы с коллекциями (массивами, списками, словарями) лежат в пространстве имен System.Collections. Такой подход позволяет сравнительно быстро ориентироваться в обилии библиотечных типов. Позволяет группировать свои типы в отдельных пространствах имен, чтобы не смешивать их с другими типами. ***Пространства имен можно рекурсивно вкладывать друг в друга***. Т.е. в одном пространстве имен может находится набор типов этого пространства и набор других вложенных в них пространств. Это позволяет внутри групп типов создавать подгруппы. Например, в пространстве System находятся все системные типы, но их очень много и их необходимо группировать.

namespace System
namespace System.Collections
namespace System.Collections.Generic
namespace System.Reflection
namespace System.Media
…

Включим в наш проект новый файл и создадим в нем класс Car. И создадим этот класс внутри пространства имен Technics.Cars.

```
namespace Technics
{
    namespace Cars
    {
        class Car
        {
        }
    }
}
```


Теперь для того, чтобы использовать новый тип данных в Main, нам надо использовать ***полное имя типа***, которое включает путь по пространству имен и имя типа через точку. В нашем случае это Technics.Cars.Car.

```
static void Main(string[] args)
{
    Console.WriteLine("Hello world!");

    Technics.Cars.Car car = new Technics.Cars.Car();

    string userInput = Console.ReadLine();
}
```

Во многих случаях типы спрятаны на один, два, три уровня вглубь пространств имен. Чтобы не писать полное имя типа используется директива ***using***, которая применима только внутри *.cs файла, в котором она использована.

```
using System;
using Technics.Cars;

namespace StartSample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");

            Car car = new Car();

            string userInput = Console.ReadLine();
        }
    }
}
```

Распределение типов по пространствм имен дает возможность решать проблемы пересечения имен типов. Два разных типа могут иметь одно имя, но находится в разных

пространствах имен. В этом случае мы используем директиву using и включаем необходимое пространство. Если мы происпользуем оба пространства в одном cs файле, то разрешить конфликт имен можно будет использованием полного имени файла. Конечно, рекомендуется при создании своих типов данных использовать свое пространство имен и уже в нем делить ваши типы на группы. Пространство имен верхнего уровня можно называть по имени компании, в которой разрабатывается код.

Резюме: В языке C# упорядочивание типов данных осуществляется с помощью пространства имен. (namespaces), которые могут создавать рекурсивную структуру. Для использования нужных типов при реализации пользовательского кода, необходимо в нужном *.cs файле использовать директиву using. Любой тип данных имеет полное имя типа, объединяющий положение в пространстве имн и типов данных. Тип данных можно использовать в коде через полное имя (например, при необходимости разрешения конфликта имен типов).

2.3 Встроенные типы данных.

Как и любой язык, C# имеет встроенный набор типов данных. Типы данных одинаковы для всей платформы .NET, поэтому они совместимы в процессе межъязыкового и межплатформенного взаимодействия. Т.е. типы данных жестко специфицированы в CLS. Не все языки программирования поддерживают некоторые типы данных. Например, типа uint нет в Visual Basic-е. Подмножество типов данных и программных конструкций, которые есть во всех языках, обеспечивают совместимые межъязыковые библиотеки кода.

Приведем краткий обзор типов данных языка C#:

Имя типа	Название в C#	Аналог в C++
System.Byte	byte	char
System.SByte	sbyte	signed char
System.Int16	short	short
System.Int32	int	int (long)
System.Int64	long	_int64
System.UInt16	ushort	unsigned short
System.UInt32	uint	unsigned int
System.UInt64	ulong	unsigned _int64
System.Single	float	float
System.Double	double	double
System.Object	object	void*
System.Char	char	_wchar_t
System.String	string	
System.Boolean	bool	bool

Любой встроенный тип данных представлен соответствующим типом в пространстве имен System. При этом название встроенного типа в C# это сокращенное название соответствующего типа данных. Так же для сравнения представлены аналогичные типы данных в языке C++.

Думаю, что с большинством типов знакомых по C++ никаких вопросов быть не должно. Однако появилось два новых типа object и string.

Как будет обсуждаться далее все типы в .NET, в том числе и встроенные, унаследованы от базового класса object. Поэтому любой объект (в том числе int), это объект типа object. Если проводить сравнение, то в C++ это был указатель на объект неизвестного типа (void*).

И наконец, прям в язык встроена поддержка строк. Это объект string. Строка представлена в формате Unicode. Char это аналог WCHAR-а в C++. На уровне языка встроена однозначность в определении строк и больше не будет огромного множества типов для работы со строками, как исторически получилось в языке C++.

2.4 Базовый тип Object.

Абсолютно все типы данных в .NET унаследованы от базового типа System.Object. Тип System.Object содержит несколько методов.

- bool Equals(object obj);
- string ToString();
- Type GetType();
- int GetHashCode();

Любой объект в C# может быть приведен к строке. Обеспечивается это за счет метода ToString(), который можно перегрузить в наследниках.

Метод Equals позволяет переопределить этот метод и произвести сравнение двух объектов по значениям этих объектов. В базовой реализации этот метод для ссылочных объектов выдает true только в том случае, если this объект и объект для сравнения это один и тот же объект.

Метод GetType() возвращает объект типа Type, который содержит полное описание типа. За счет этой особенности реализовано динамическое приведение типов (RTTI) и рефлексия (Reflection). Этим двум вопросам уделим в дальнейшем особое внимание. Это самый важный метод типа Object.

2.5 Структурные и ссылочные типы.

В этом разделе будут описаны основные различия между структурными и ссылочными типами. Этот материал крайне важен т.к. дает понимание принципов жизненных циклов объектов этих двух типов.

2.5.1 Умные указатели.

В этой главе будет кратко описан принцип работы умных указателей. Для тех, кто знаком с это темой, все равно необходимо ознакомится с ней.

Вернемся к языку C++. Есть два способа создания объектов в памяти. На стеке и в динамической памяти (куче).

Переменные объявленные в теле функций, циклов и операторов ветвления (без оператора new) создаются на стеке. ***Тело реализации*** определяется открывающей и закрывающей скобкой.

```
{
    //Тело 1
    int a = 5;
    {
        //Тело 2
        Car car;
        //Конец тела 2
    }

    {
        //Тело 3
        double d = 11.1;
        //Конец тела 3

    }
     //Конец тела 1
}
```

Переменные в приведенном выше примере будут создаваться и умирать следующим образом. На стеке появится переменная int a, потом будет создан объект Car. При выходе из «Тела 2» объект car будет уничтожен на стеке. Потом при входе в «Тело 3» на стеке будет создана переменная double d, при выходе уничтожена и после этого будет убита переменная a.

Принцип работы стека это выделение памяти под объекты при входе в тело реализации. И при покидании тела реализации происходит уничтожение этих объектов. Другими словами, жизненный цикл объекта на стеке определяется областью видимости, в которой создана эта переменная. Пока существует область «{ Тело }», существуют и объекты.

Объекты создаваемые с ключевым словом new создаются в динамической памяти. Для освобождения этой памяти программист должен освободить этот объект с помощью оператора delete.

```
{
    //Тело 2
    Car* car = new Car();
}
```

При покидании области видимости объект созданный в «куче» остается живым. А на стеке в приведенном выше примере создается и умирает указатель.

В приведенном выше примере после того, как указатель car на стеке был удален, мы получили утечку памяти. Сам объект уже высвободить никто не сможет. В задачи программиста на C++ входит рутинная работа по ручному удалению выделенной памяти с использованием оператора delete. При

этом совершено легко пропустить ветку кода, в которой выделенный на «куче» объект остается потерянным.

В языке C++ для решения этой проблемы был придуман способ для работы с объектами выделенных в куче посредством «***умных указателей***» (Smart Pointers).

Для этого к каждому типу определяется с помощью шаблона новый тип с приставкой Ptr. В нашем примере это CarPtr. Объект этого типа переопределяет операторы присваивания и копирующего конструктора. Внутри себя хранят указатель на реальный объект и счетчик ссылок на этот объект. Работа с объектами идет только с помощью умных указателей. При этом сами указатели больше не нужны и инкапсулированы внутри умных указателей.

```
{
    CarPtr car1 = new Car();
    CarPtr car2 = car1;

    DoSmthWihCar(CarPtr car2);
}
```

В примере выше, на стеке создаются теперь умные указатели. При покидании области видимости они уничтожаются, а значит для них вызывается деструктор. При операции присваивания можно завести общий счетчик ссылок на объект и увеличивать его (операция инкремента). При вызове деструктора производить его уменьшение (операция декремента). Когда у нас будет умирать последняя ссылка на объект, то счетчик обнулится и в этот момент можно автоматически вызвать оператор delete.

Умные указатели нашли очень широкое распространение у разработчиков на C++ т.к. избавляют от необходимости отслеживать все возможные ветвления кода. Даже если

внутри метода `DoSmthWihCar` вылетит исключение, и оно не будет перехвачено, все равно объекты со стека будут уничтожены, будут вызваны их деструкторы и будет убит объект *car* в «куче».

Т.е. фактически в С++ мы имеем три возможности создать объект *car*. На стеке, и двумя способоми в «куче». Будем рассматривать только вариант с умным указателем.

```
{
  Car car; //Объект на стеке
  CarPtr car = new Car();//Ссылка на объект в куче
  CarPtr car; //Пустая ссылка
}
```

Если проанализировать два оставшихся варианта создания на стеке и создания на куче, то они идентичны. Т.е. с точки зрения разработчика все равно как ему создать объект car. Он должен умереть при выходе из области видимости.

Более того, второй вариант обеспечивает безопасность того, что объект создан на куче и если кто-то запомнит на него ссылку, то этот объект не умрет. В пером же случае при выходе за тело реализации, объект car будет убит и если где-то останется на этот объект ссылка, при использовании которой поведение программного кода будет неопределенным, скорее всего произойдет ошибка обращения к памяти. Это достаточно распространенная ошибка на С++ и отловить ее достаточно сложно. Поэтому первый вариант небезопасен и избыточен. И его можно выкинуть.

```
{
  CarPtr car = new Car();//Ссылка на объект в куче
  CarPtr car; //Пустая ссылка
}
```


Но есть типы, для которых не хотелось бы создавать умные указатели. Например, было бы очень неудобно вместо строки:

```
int a = 5;
```

писать конструкцию:

```
IntPtr car = new int(5);
```

Особенностью этих типов является, то что в подавляющем количестве случаев они создаются на стеке в виде локальных переменных, параметров функций. (Вариант того, что это поля объекта другого класса мы пока не рассматриваем).

2.5.2 Структурный и ссылочный тип.

Возвращаемся к языку C#. Все типы (за исключением интерфейсов) в языке C# поделены на два типа. ***Структурные и ссылочные типы***. Фактически существует два типа ***ValueType*** и ***ReferenceType***.

Почти все встроенные типы данных такие как int, double, bool являются наследниками типа ***ValueType*** и следовательно являются структурными типами. Исключением является тип строка (string), который является ссылочным типом, но его поведение очень похоже на объекты структурного типа в силу его реализации.

При определении пользовательских типов данных в C# используются такие же ключевые слова, как и в С++. ***<struct, class, enum>***.

Если в С++ между «struct» и «class» типами была незначительная разница по дефолтному поведению операторов доступа *<public, protected, private>*, то в C# разница коллосальна.

Производные типы от struct и enum являются наследниками типа ValueType. Соотвественно являются ***структурными типами***.

Производные типы от class являются наследниками любого типа кроме ValueType. Соотвественно являются ***ссылочными типами***.

Вернемся к концу предыдущего раздела. ***Структурные типы*** ведут себя точно так же как все типы в языке C++, как это было описано выше. При объявлении переменной в теле реализации, она создается на стеке.

Объекты ссылочного типа можно создавать исключительно в динамической памяти. При этом реально на стеке заводятся ссылки (аналоги умных указателей рассмотренных в предыдущей главе). ***Доступ к объектам ссылочного типа осуществляется только через ссылки***.

В обоих случаях ***указателей нет*** (т.е. реального адреса объекта). В случае со структурными типами он не нужен, в случае ссылочных типов, адрес хранится внутри ссылки и наружу недоступен.

В предыдущей главе мы пришли к варианту, что для сложных типов с использованием умных указателей(ссылок) остается вариант создания типа только в динамической памяти. Как только перестанут существовать ссылки, то объекты могут быть уничтожены.

```
{
  CarPtr car = new Car();//Ссылка на объект в куче
  CarPtr car; //Пустая ссылка
}
```


Эта методика умных указателей и автоматического освобождения занятой памяти, является основной концепцией языка. Приставку «Ptr» для обозначения ссылки убираем.

```
{
  Car car = new Car();//Ссылка на объект в куче
  Car car; //Пустая ссылка
}
```

Первым важным отличием структурного и ссылочного типа является способ создания и жизненного цикла объектов.

Расмотрим структурный тип *Point3d.* И ссылочный тип Car.

```
{
  Car car1 = new Car();//Ссылка на объект в куче
  car1.MaxSpeed = 180; //Доступ к объекту по ссылке
  Car car2; //Пустая ссылка
  car2.MaxSpeed = 200; //Ошибка! объекта нет

  //Объект pnt1 на стеке
  Point3d pnt1 = new Point3d(0,0,0);
  pnt1.X = 20; //Корректно.
  Point3d pnt2; //Объект pnt2 на стеке.
  pnt2.X = 10; //Корректно.
}
```

Т.е. по коду выше видно, что для *car2* создалась пустая ссылка. При попытке обратиться к свойствам вылетит исключение попытки обращения к несуществующему объекту. Однако, для pnt2 создается не ссылка, а объект на стеке. Поэтому обращение к полям возможно.

Такое различное поведение визуально одинаковых синтаксических конструкций вызывает наиболее мучительный процесс привыкания для тех, кто долго работал на C++ и переходит к программированию на языке C#.

Для более быстрой адаптации рекомендуется использовать создание структурных типов только в случаях острой необходимости (класс Point3d и т.п.). Все пользовательские типы стараться создавать через ключевое слово ***class***.

Второе различие между структурными и ссылочными типами заключено в поведении при операциях присваивания.

```
{
  Car car1 = new Car();//Ссылка на объект в куче
  car1.MaxSpeed = 180; //Доступ к объекту по ссылке
  Car car2 = car1;

  //Объект pnt1 на стеке
  Point3d pnt1 = new Point3d(0,0,0);
  Point3d pnt2 = pnt1; //Объект pnt2 на стеке.
}
```

При присваивании ссылок, внутри ссылок, происходит присваивание адреса объекта. Т.е. для случая с типом Car, в нашем примере будут две ссылки, которые будут ссылаться на один и тот же объект.

В случае с классом Point3d, будет два объекта pnt1 и pnt2, при операции присваивания будет произведено побитовое копирование данных их объекта pnt2 в объект pnt1.

При операциях присваивания для структурных типов происходит копирование значения, а для ссылочных типов происходит присваивание ссылкам объектов.

Третье различие в поведении структурных и ссылочных типах проявляется при операциях сравнения.

Для структурных типов происходит сравнение по значению, а для ссылочных типов по адресу объекта.

```
{
  Car car1 = new Car();//Ссылка на объект в куче
  car1.MaxSpeed = 180; //Доступ к объекту по ссылке
  Car car2 = new Car();//Ссылка на объект в куче

  //car1 и car2 это разные объекты
  bool isEqualRef = (car2 == car1); //false
  car2 = car1;
  //car1 и car2 ссылаются на один объект
  isEqualRef = (car2 == car1); //true

  //Объект pnt1 на стеке
  Point3d pnt1 = new Point3d(0,0,0);
  Point3d pnt2 = pnt1; //Объект pnt2 на стеке.

  //pnt1 и pnt2 разные объекты, одинаковые значения
  bool isEqualVal = (pnt1 == pnt2); //true
  pnt1.X = 10;
  //pnt1 и pnt2 разные объекты, разные значения
  isEqualVal = (pnt1 == pnt2); //false
}
```

Резюме: Структурные и ссылочные типы имеют различное поведение при создании объектов, операциях присваивания и сравнения. Объекты ссылочного типа могут быть созданы только в динамической памяти, доступ к этим объектам осуществляется посредством ***ссылок***. Встроеные примитивные типы, перечесления (enums) являются структурными типами. Пользовательские классы (classes), являются ссылочными типами. Пользовательские структуры (structures) являются структурными типами.

2.6 Упаковка и распаковка

В предыдущей главе мы выяснили, что технология .NET поддерживает два вида типов. Структурные и ссылочные. Структурные типы размешаются на стеке или в виде вложенных полей в объектах кучи. В процессе присваивания двух переменных структурного типа происходит копирование по значению (побитовое копирование данных).

Что происходит если необходимо привести структурный тип к ссылочному или наоборот?

```
double d = 10.5;
object obj = d;
```

Данная операция называется упаковкой (boxing). При этом в куче создается объект double, происходит побитовое копирование данных и теперь obj является ссылкой. Очевидно, что операция упаковки не из самых быстрых.

Обратная операция называется распаковкой (unboxing). Т.е. данные из объекта в куче побитово переносятся в объект структурного типа.

```
double d1 = (double)obj;
```

Резюме: При присваивании объектов структурного типа происходит побитовое копирование данных. При присваивании ссылок друг другу, происходит передача адреса от одной ссылке другой на один и тот же объект в динамической памяти. При приведении типов от структурного к ссылочному и наоборот происходит операция упаковки и распаковки.

2.7 Автоматическое освобождение памяти.

Ранее мы обсудили, что доступ к объектам в динамической памяти осуществляется исключительно через ссылки. И эта концепция вытекает из идеи умных указателей, которая призвана решить проблему рутинного удаления неиспользуемых объектов программистами.

В разделе про умные указатели мы сказали, что можно организовать счетчик ссылок на объект и при присваивании ссылке объекта его инкрементировать, при удалении ссылки (в деструктуре или операции переприсваивания) его декрементировать. Но эта концепция в общем случае не работает. Как только появляется набор объектов, которые ссылаются друг на друга циклически через умные указатели, то счетчик ссылок на эти объекты никогда не занулится. И мы получим утечку памяти. Поэтому при использовании умных указателей в C++ приходилось заботится об односторонней структуре ссылок. Ссылки в обратную сторону (например в дереве), делались обычными указателями. Чтобы при потере ссылки на головной элемент дерева, произошло освобождение всех элементов дерева.

Как же организовать автоматическое удаление неиспользуемых объектов в случае, когда могут быть циклические ссылки между объектами?

Опять стоит вспомнить, что сами ссылки могут размешаться как на стеке, так и в динамической памяти в виде полей объектов. Критерий того, что объект не потерян заключается в том, что до него можно добраться по ссылке со стека. Рассмотрим в качетве примера дерево: Если со стека можно добраться до корневого элемента дерева, то значит дерево «живое» и мы можем добраться до всех вложенных потомков child-элементов. Как только мы на стеке потеряли

ссылку на все элементы дерева (в том числе и на корневой элемент), то все объекты стали потерянными. Если мы можем со стека добраться хоть до одного элемента дерева, то мы можем добраться и до его корня.

Т.е. критерием того, что объект «живой» является возможность добраться до него по цепочке ссылок от какой-либо ссылки на стеке.

Всю взаимосвязь по ссылкам можно представить в виде графа. Ссылки на стеке это «точки входа». При уничтожении ссылок на стек, при их присваивании меняется вся общая структура графа. К сожалению, проверять каждый раз весь граф объектов при каждом изменении ссылок, для выявления неиспользуемых объектов и их немедленного уничтожения, нецелесообразно. При огромном числе ссылок, это займет огромные вычислительные ресурсы.

Поэтому созданные и неиспользуемые объекты сразу не уничтожаются, и никаких счетчиков ссылок не делается (так же чтоб не тратить ресурсы). Вместо этого объекты выделяются до тех пор, пока не будет исчерпана квота памяти, выделенная менеджером памяти процесса. Как только квота памяти исчерпана начинает работу ***«Garbage Collector» (сборщик мусора)***.

Он знает все «точки входа» (ссылки на стеке) и проходит по всем объектам и находит все живые объекты. Конкретная реализации стратегии сбора мусора может варьироваться. Одна из них, это поделить память на две области. Выделение памяти производится из одной области, как только место области исчерпано, запускается **GC**. Идя по ссылкам, он переносит живые объекты во вторую область (побитовым копированием). В результате в первой области остаются неиспользуемые объекты и свободная память. При этом при

побитовом переносе «живых» объектов происходит уплотнение памяти. Т.е. избгается проблема дефрагментации, когда менеджер памяти вынужден искать свободные кусочки памяти для ее выделения. (Поскольку объекты в C++ имели постоянное расположение в памяти, и при удалении объектов происходила фрагментация памяти, это осложняло работу менеджера памяти).

После переноса и уплотнения объекттов, происходит трансляция адресов ссылок (т.е. всем найденным ссылкам ставится новый адрес объектов после «переезда»).

Плюс этого метода в том, что нет даже необходимости индивидуально убивать каждый неиспользуемый объект. Просто вся область памяти помечается как свободная (все объекты убиваются «одним движением руки»).

Минус этого метода в том, что надо выполнять побитовое копирование «живых» объектов. Но практика показывает, что мертвых объектов за время работы одного цикла накапливается существенно больше (на порядки) чем живых. Поэтому даже из-за этого убивать каждый объект индивидуально было бы накладно.

Но все равно лишний раз таскать «живые» объекты по памяти накладно. Поэтому придумана концепция ***поколений*** (***Generations***).

Память разделяется на части (поколения). Первое поколение, второе поколение и третье поколение. Выделение новых объектов идет в области памяти первого поколения. При использовании квоты памяти для первого поколения, начинается перенос живых объектов в область памяти второго поколения. Т.е. после сбора мусора память первого поколения снова становится пустой, а все пережившие сборку мусора

объекты уходят во второе поколение. При следующих сборках мусора, опять живые объекты из первого поколения переносятся во второе. И так происходит до тех пор, пока квота памяти на второе поколение не исчерпывается. В этом случае живые объекты переходят в последнее третье поколение. Второе поколение очищается.

Как показывает практика, подавляющее большинство объектов временные и они рождаются и умирают в первом поколении. Объекты со среднем сроком жизни переходят во второе поколение. И их «за зря» по памяти никто не таскает до тех пор, пока не наступает сборка мусора второго поколения. В третье поколение попадают как правило «синглетоны» и объекты выживающие в течении длительного времени. В зависимости от того, с какой скоростью и объемами приложение поглощает память, зависит и частота запуска сборщика мусора. Для приложений, слабо использующих выделение временной памяти, сборщик мусора может вообще ни разу не запускаться. Для активно работающих приложений сборщик мусора может запускаться с частотой раз в час или несколько часов. (При этом сборщик мусора отслеживает простой процессов и может запуститься в фоновом режиме и его работа попросту незаметна). Для интенсивно работающих приложений, потребляющих большое кол-во памяти (например, расчеты по большим графам, которые регулярно перестраиваются по некоторой моделе на сервере или десериализуются из базы, чертежа и т.п.), сборка мусора может проходить с частотой от нескольких минут до часа.

Идея с использованием поколений предотвращает излишние копирование в памяти «живых» объектов.

В C# сборщик мусора представлен классом ***System.GC*** и может использоваться для управления сборкой мусора и

получения информации по занятой и свободной памяти. GC содержит набор статических методов. Мы рассмотрим лишь один метод Collect:

```
GC.Collect();
GC.Collect(generation);
```

Этот метод запускает сборку мусора. Первый метод инициализирует сборку мусора во всех поколениях, второй метод инициализирует сборку в первом, втором и третьех поколениях.

Этот метод можно вызывать перед некоторыми большими и ответственными действиями. Допустим мы хотим снять показания с датчиков по температуре, давлению (и т.п.) поступающих из эксперементной установке по взрыву материала. Показания надо снимать с большой частотой чтобы отследить динамику быстропротекающего процесса. У нас может быть много датчиков. Очевидно, что на поступающую информацию потребуется большой объем памяти. Будет очень обидно, если в середине сбора информации сборщик мусора вдруг решит собрать мусор… Особенно если проведение эксперемента требует больших материальных и временных затрат на подготовку. В этом случае будет целесообразно перед началом эксперимента заставить сборщик мусора собрать всю память.

Резюме: Мы рассмотрели концепцию и алгоритм особождения памяти из-под неиспользуемых «мертвых» объектов. Выяснили что целесообразно запускать сборщик мусора по мере достижения занятой квоты памяти. При этом «выжившие» объекты копируются в памяти, а высвобожденная память с мертвыми объектами разом высвобождается. Именно поэтому адрес объекта скрыт, т.к. не

гарантировано, что он будет постоянным. А работа с объектами осуществляется через ссылки.

2.8 Освобождение ресурсов и деструкторы.

Что делать, если при уничтожении объекта нам надо высвободить какой-либо ресурс? В языке C++ для этих целей предназначался деструктор.

Сейчас в объекте System.Object есть метод Finalize(), который, правда фактически, скрыт и его нельзя переопределить. Но можно в C# написать конструкцию напоминающую деструктор класса в языке C++.

```
namespace Cars
{
    class Car
    {
        ~Car()
        {
            //Освобождаем ресурсы.
        }
    }
}
```

При определении этого метода класс будет помечен как «finalizable» и сборщик мусора перед тем как освободит память вызовет для уничожаемых объектов деструктор (Finalize).

Но вызов этого деструктора произойдет только в тот момент, когда будет активирован процесс сборки мусора. Что делать если нам надо разорвать соединение с сервером в момент, когда ссылка на это соединение теряется из области видимости. Ответ на этот вопрос простой, вручную. Освобождение таких ресурсов должен отслеживать пользователь, исходя из логики своего приложения.

Освобождение важных ресурсов это не менее ответственная задача нежели их получение. Для этого предусмотрен метод Dispose(). И поддерживается специальный интерфейс IDisposable. При необходимости разорвать соединение с сервером, пользователь должен вызвать метод Dispose, в котором будет произведено отключение.

Как синхронизировать работу метода Dispose и деструктора? Для этого в методе Dispose надо вызвать метод сборщика мусора GC.SupressFinalize(thisObject). Вызов этого метода подавляет вызов деструктора. Т.е. если мы вызовем метод Dispose() вручную, то вызов деструктора будет подавлен. Иначе метод Dispose() позовется из деструктора при отчистке памяти сборщиком мусора.

```
class Car : IDisposable
{
    ~Car()
    {
        Dispose();
    }

    public void Dispose()
    {
        //Освобождаем ресурсы
        GC.SuppressFinalize(this);
    }
}
```

3. C#. Синтаксические конструкции.

После знакомства с идеологией языка C# (структурными и ссылочными типами, автоматической сборкой мусора) мы познакомимся с синтаксическими конструкциями языка, определением функций и их параметров, определения свойств (Properties) и пользовательских типов.

3.1 Операторы ветвления.

Реализация операторов ветвления полностью идентична по своему синтаксису реализации в языке C++.

```
if ( logicExpression )
{
    //Тело оператора true
}
else
{
    //Тело оператора else
}
```

Для оператора switch синтаксис полностью повторяет знакомый нам вид по C++.

```
switch (Value)
{
    case VAL1: //Тело
        break;
    case VAL2: //Тело
        break;
    case VAL3:
    case VAL4: //Тело
        break;
    default:
        break; //Тело

}
```


Стоит лишь отметить что теперь для оператора if выражение должно быть строго типа bool. В языке C++ мы могли подставить числовой тип, который трактовался как:

(val != 0) ? true : false

(Кстати, этот условный оператор есть и в C#)

Эта особенность была внесена обдуманно. Возможность подставить числовой тип очень часто приводила к случайным ошибкам, которые компилятор не трактовал как ошибку:

if (intVal = 1)

В этом случае происходило затирание значения в переменной intVal и мы попадали в основное тело условного оператора. А в подавляющем большинстве случаев такая ситуация была опечаткой программиста. if (intVal == 1).

Отличительной особенностью конструкции switch в C#, это возможность использования строковых переменных наряду с числовыми типами и пересечислениями.

3.2 Операторы (Арифметические, логические, сравнения, присваивания).

В этом разделе кратко перечислим операторы языка C#. Поскольку они полностью идентичны по значению операторам C++, не будем заострять внимание на каждом из них по отдельности.

Операторы сравнения			
==	Равно	>	Больше
!=	Не равно	<	Меньше
>=	Больше равно	<=	Меньше равно

Операторы для условий			
&&	Логическое И	!	Не (отрицание)
||	Логическое ИЛИ		
Операторы присваивания X (operator)= Y			
=	Присваивание	+=	Прибавление
*=	Умножение	-=	Вычитание
/=	Деление	%=	Остаток от деления
<<=	Левый сдвиг	>>=	Правый сдвиг
&=	Побитовое И		|= Побитовое Или
Арифметические операторы			
+	Сложение	*	Умножение
-	Вычитание	/	Деление
%	Остаток от деления		
Битовые операторы			
<<	Левый сдвиг	>>	Правый сдвиг
&	Логическое И	|	Логическое ИЛИ
^	XOR (Искл. ИЛИ)		
Унарные операторы			
+	Плюс	~	Bit reversion
-	Минус		
++	Инкремент (пре/пост)	--	Декремент (пре/пост)

3.3 Операторы циклов.

В языке C# существует четыре оператора циклов. Три из них идентичны тем, что присутствовали в C++, и один пришел из языка Visual Basic.

Циклы for, while, do/while

```
for(int i = 0; i < 10; i++)
{  //Тело цикла
}
```



```
while ( logicExpression )
{
    //Тело цикла
}

do
{
    //Тело цикла
} while (logicExpression);
```

В отличие от C++, как и с условными операторами, результатом логического вырожения может быть только результат типа bool.

В C# появился цикл foreach, который позволяет перебирать объекты поддерживающих перечисление (списки, массивы).

```
foreach (T tObj in ListT)
{
    //Тело цикла
}

List<Car> cars = new List<Car>();
//Заполняем список машинами
foreach ( Car car in cars )
{
    //Ускоряем каждую машину в списке
    car.SpeedUp();
}
```

Фактически, конструкция foreach это упрощенная запись цикла for, в котором не надо заводить временную переменную для инкремента и локальную переменную индексатора.

```
for (int i = 0; i < cars.Count; i++)
{
    //Ускоряем каждую машину в списке
    Car car = cars[i];
    car.SpeedUp();
}
```

3.4 Определение констант.

В C# можно определять константы. На уровне класса и на уровне метода. При этом их необходимо инициализировать на месте объявления. Из-за поддержки 100% модели объектного языка, объявить константу вне класса нельзя.

```
class Car
{
    public const int IntConst = 10;

    public void f()
    {
        const string localString = "String";

        Console.WriteLine("{0} {1}",
                IntConst, localString);
    }
}
```

3.5 Пользовательские типы.

Для определения пользовательских типов необходимо воспользоваться ключевыми словами ***class***, ***struct*** или ***enum***. В языке C# появилась новая возможность помечать пользовательский тип модификатором доступа.

```
public class Car
{
  …
}

internal class Car
{
  …
}
```


Что определяет модификатор доступа для типов? При обсуждении библиотечных dll файлов мы выяснили, что при подключении ссылки на dll файл с библиотекой становятся доступны все типы, определенные в этом файле. Но что если часть типов в сборке являются вспомогательными для реализации функционала библиотеки и их необходимо скрыть от пользователей? Для этого и используется модификаторы доступа к типам. Их существует всего два: ***public*** и ***internal***. Если модификатор не указать явно, то по умолчанию будет использован модификатор internal.

Ниже приведен пример определения структурного типа точки трехмерного пространства:

```
public struct Point3d
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

Различия между ключевыми словами struct и class были рассмотрены в теме структурных и ссылочных типов.

Сразу отметим еще пару различий, для структурных типов отсутсвет возможность наследования. Поля классов инициализируются значениями по умолчанию, а поля структурных типов необходимо инициализирвать руками или в конструкторе. При попытке использовать неинициализировнный структурный тип компилятор будет выдавать ошибку.

Так же приведем пример создания пользовательского перечисления. При его определении можно (не обязательно) явно задавать соотвествующее числовое значение. Если

числовые значения не заданы, то нумерация начинается с нуля.

```
public enum UserTypes
{
   ADMINISTRATOR = 1,
   POWER_USER = 3,
   USER = 5,
   GUEST = 7
}
```

При определении перечисления можно указывать перменную какого типа использовать для идентификации типа. По умолчанию это int. Но можно использовать любой целочисленный тип данных (byte, short, int, long + unsigned).

```
public enum UserTypes : byte
{
   …
}
```

Структурный тип, по умолчанию неявно наследуется от типа System.ValueType.

Любой тип перечисления так же является структурным типом, и унаследован от типа System.Enum. Этот тип предоставляет набор статическим методов, которые можны использовать для получения информации по типу. (Список строквых и числовых значений определяемых перечислением, получить тип лежаший в основе перечисления, получить строковое представление переменной)

```
UserTypes userType = UserTypes.POWER_USER;
Enum.GetUnderlyingType(typeof(UserTypes));

string[] names   = Enum.GetNames(typeof(UserTypes));
Array values     = Enum.GetValues(typeof (UserTypes));
string valName   =  Enum.GetName(typeof   (UserTypes),
userType);
```


3.6 Поля. Свойства. Статические данные типов.

В этом разделе мы рассмотрим создание полей и свойств в пользовательских типах.

Сразу необходимо запомнить разницу, которая отличает создание полей и методов в языках C# и C++. ***Модификаторы доступа (public, protected и private) ставятся индивидуально для каждого поля, свойства, метода или события***.

```
public class Car
{
     public double CurrentSpeed = 30;
}
```

В программировании принято инкапсулировать поля классов и делать для них метода доступа (getter и setter). Определим такие поля для нашего примера, а само поле сделаем приватным.

```
public class Car
{
    private double _currentSpeed = 30;

    public double GetCurrentSpeed()
    {
        return _currentSpeed;
    }

    public void SetCurrentSpeed(double speed)
    {
        if ( speed < 0 )
            return;

        _currentSpeed = speed;
    }
}
```

Метода get и set принято писать для инкапсуляции данных типа. Доступ к полю может быть осуществлен только через методы доступа. Это позволяет при написании кода не ошибиться, и случайно не переопределить поле. Более того, если необходимо защитить поле от изменения извне, то метод «set» можно убрать. Еще одним полезным моментом методов доступа является возможность установить току останова для дебагера и отследить в какой момент происходит доступ или изменение поля.

В языке C# введена специальная конструкция, которая позволяет создавать свойства. ***Свойство*** (***Property***) – это конструкция, которая объединяет поле типа и метода доступа для них get и set. Переделаем наш пример.

```
private double _currentSpeed;

public double CurrentSpeed
{
    get { return _currentSpeed; }
    set
    {
        if ( value < 0 )
            return;

        _currentSpeed = value;
    }
}
```

Мы создали свойство CurrentSpeed. Использовать его можно в программе так же как и поле, но в зависимости от того запрашиваем мы значение поля или его устанавливаем будет происходить вызов get или set конструкции. Ключевое слово ***value*** в set является новым устанавливаемым значением того типа, что определен для свойства (в примере double).

Теперь пример использования этого свойства в коде приложения:

```
static void Main(string[] args)
{
    Car car = new Car();
    car.CurrentSpeed = 120;

    Console.WriteLine("Speed {0}", car.CurrentSpeed);
}
```

Если необходимо сделать свойство только на чтение или на запись надо убрать соотвествующую часть get или set в свойстве.

В новых версиях Framework .NET появилась возможность краткой записи свойств, в случае если get и set примитивны.

```
public class Car
{
    public double CurrentSpeed { get; set; }
}
```

Т.е. не надо заводить приватное поле и писать реализацию примитивных get и set, которые по умолчанию будут устанавливать значение и возвращать его. При этом сама приватная переменная будет скрытой и будет инициализирована значением по умолчанию для данного типа.

Значения по умолчанию для полей класса: Все числовые типы инициализируются нулевым значением, для bool по умолчанию идет false, для перечислений первое определенное значение. Для ссылочных типов ссылка будет установлена в null. ***Это правило работает для полей экземпляра объекта, полей класса (статических полей), констант.***

Свойства и поля могут быть статические. Определим в классе Car несколько полей и свойств с максимально разрешимой скорость езды в городе и за его пределами. Конечно, эти поля можно было сделать константами, но в разных странах разрешенная скорость в городе и за ним может меняться. Или может использоваться другая единица измерения скорости.

```
public class Car
{
    public double CurrentSpeed { get; set; }

    private static double _maxCitySpeed = 60;
    public static double MaxCitySpeed
    {
        get { return _maxCitySpeed; }
        set { _maxCitySpeed = value; }
    }

    private static double _maxCountrySpeed = 90;
    public static double MaxCountrySpeed
    {
        get { return _maxCountrySpeed; }
        set { _maxCountrySpeed = value; }
    }
}
```

3.7 Методы и передача параметров.

Аналогично полям и свойствам методы определяются только внутри класса, и для них в индивидуальном порядке используется модификатор доступа.

Создадим для нашего примера с автомобилем метод для ускорения автомобиля. Будем передавать в него изменение скорости. Будем контролировать разрешенную максимальную скорость и следить, чтобы скорость не стала отрицательной. И

в качестве возвращаемого значения будем возвращать «true» в случае если скорость изменилась.

```
public bool SpeedUp(double accValue)
{
    double oldValue = CurrentSpeed;

    if ((CurrentSpeed + accValue) > MaxCitySpeed)
        CurrentSpeed = MaxCitySpeed;
    else
        CurrentSpeed = CurrentSpeed + accValue;

    if (CurrentSpeed < 0)
        CurrentSpeed = 0;

    return (oldValue != CurrentSpeed);
}
```

При необходимости передать несколько параметров указываем их через запятую.

```
public double MakeSumm(double a, double b)
{
    return a + b;
}
```

Если вспомнить C++, то у нас были три возможности передавать параметры по значению, указателю и ссылке. При передаче по значению происходило создание новой переменной на стеке и следовало побитовое копирование значения. При передаче по указателю и по ссылке происходила передача адреса переменной. При передаче по ссылке мы работали с объектом «напрямую», а не через указатель.

Теперь в C# адрес объекта скрыт. Поэтому при передаче в качестве параметра переменной структурного типа происходит его копирование по значению, следовательно создание на стеке новой переменной. При передаче

переменной ссылочного типа, происходит создание на стеке новой ссылки. Если поменяем значение структурного типа или присвоем ссылке новое значение, то при выходе из метода ничего не изменится сверху.

Если необходимо вернуть из метода измененный параметр, то в языке C# для этих целей используется два новых ключевых слова ***out*** и ***ref***. Если взглянуть на проблему передачи параметров, то параметры методов бывают трех типов: ***входные [in]***, ***исходящие [out]*** и ***входящие-исходящие [in-out]***.

[out] параметры отличаются от [in-out], тем что [out] параметры должны быть обязательно инициализированы внутри метода. Т.е. при любом ветвлении тела метода, при выходе из него, [out] параметру должно быть присвоено значение. При передаче [out] параметра в метод его не обязательно инициализировать. Для [in-out] параметра все строго наоборот. Он может не измениться внутри тела метода, но обязан быть инициализирован до его подстановки в метод.

В C# если для параметра не указан тип out или ref, то параметр считается входящим и передается по значению. (для ссылок по значению передается ссылка).

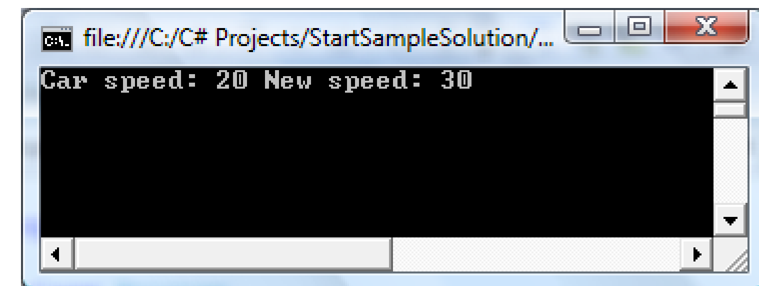
```
public static void SampleMethod( out Car newCar,
                                 ref double speed)
{
    newCar = new Car();
    newCar.CurrentSpeed = speed;

    if (speed < Car.MaxCitySpeed / 2)
        speed += 10;
}
```

Для нашего примера сделаем метод, который должен создать и вернуть ссылку на новый автомобиль и при

исполнении некоторого условия увеличить скорость входящего параметра.

При вызове метода и подстановке параметров, так же необходимо указывать out или ref для соотвествующего параметра.

```
static void Main(string[] args)
{
   //Пустая ссылка
   Car car;

   //Необходимо инициализировать
   double speed = 20;

   //Вызываем метод
   Car.SampleMethod(out car, ref speed);

   //Выводим новые значения
   Console.WriteLine("Car speed: {0} New speed: {1}",
                           car.CurrentSpeed, speed);

   Console.ReadLine();
}
```



Для создания метода класса (статического метода) необходимо между модификатором доступа и типом возвращаемого значения добавить ключевое слово static, как это сделано для метода нашего примера SampleMethod.

Модификаторы доступа: для полей, свойств, методов и событий существует четыре вида модификаторов доступа: ***public***, ***protected***, ***private*** и ***internal***.

Для первых трех модификаторов не должно быть никаких вопросов, т.к. все аналогично языку C++. Модификатор ***internal*** это модификатор public, но помеченное им поле, свойство или метод будут доступны только внутри сборки. При подключении dll файла с библиотекой этот метод не будет виден. Мы уже встречали использование модификатора internal для определения пользовательских типов.

Модификатор internal необходимо рассматривать как еще один уровень инкапсуляции данных. Он скрывает определение типов, полей, свойств и методов внутри одной сборки.

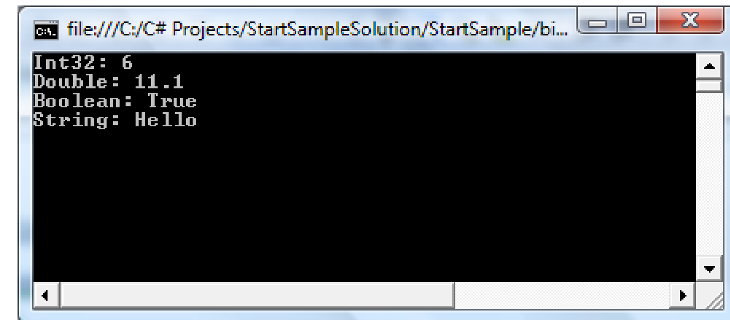
Последнее, что необходимо затронуть в этой теме, это методы с переменным числом параметров. Для создания метода с переменным числом параметров используется ключевое слово ***params***, которое добавляется к последнему параметру-массиву метода.

```
public static void SampleParams
                        (params object[] inObjs)
{
    string str = string.Empty;

    foreach (object obj in inObjs)
    {
        string type = obj.GetType().Name;
        str += String.Format("{0}: {1}\n",
                                  type, obj);
    }

    Console.WriteLine(str);
}
```


При вызове метода мы можем указать любое количество параметров различного типа

```
Car.SampleParams(6, 11.1, true, "Hello");
```

В результате работы нашего приложения мы получим следующий результат.



4. C#. ООП.

В этой главе будут рассмотрены вопросы наследования, полиморфизма, создания абстрактных классов и интерфейсов. Инкапсуляция как один из принципов ООП рассмотрен в предыдущей главе.

Стоит лишь напомнить, что появился еще один уровень инкапсуляции уровня сборок. Если класс или члены помечены модификатором доступа public, то они будут видны в других сборках при включении в качестве библиотеки. При определении их модификатором internal, поведение будет аналогично модификатору public внутри сборки, а за пределами сборки эти члены и типы будут скрыты.

4.1 Наследование.

В языке C# наследование можно проводить только для интерфейсов и классов. Создадим базовый класс фигуры:

```
public class Shape
{
    private Color _color = Color.Red;
    public Color Color
    {
        get { return _color; }
        set { _color = value; }
    }
}
```

Создадим структурный тип Point, который позволит нам задавать вершины наших фигур. Особенностью структурных типов является то, что им нельзя переопределить конструктор по умолчанию. Перед использованием структурного типа все

его поля должны быть инициализированны, для чего мы и создали конструктор.

```
public struct Point
{
    public Point(double x, double y)
    {
        _x = x;
        _y = y;
    }

    private double _x;
    public double X
    {
        get { return _x; }
        set { _x = value; }
    }

    private double _y;
    public double Y
    {
        get { return _y; }
        set { _y = value; }
    }
}
```

Тип Color определен в библиотеке System.Drawning. Для выполнения этого примера необходимо ее подключить в ссылки проекта.

Создадим класс окружности производный от Shape. Как видно из примера в отличии от языка C++ ушло понятие public и protected наследования. Теперь просто указывается базовый класс.

```
public class Circle : Shape
{
   public Point Center { get; set; }
   public double Radius { get; set; }
}
```

В языке C# используется «Single-Inheritance» модель, т.е. у класса может быть только один базовый класс. Это еще одно различие с языком C++. Обсуждать «полезность» поддержки множественного наследования в этой теме мы не будем, но оно вносило в язык много «сложностей» как разработчикам языка, так и в архитектурную модель приложения. При правильном построении архитектуры приложения всегда можно обойтись без множественного наследования.

При этом множественное наследование по интерфейсам поддерживается в языке C#. О чем мы поговорим немного позже.

4.2 Конструкторы.

В контексте обсуждения наследования разберем конструкторы классов и структур. Поскольку наследования по структурным типам нет, то конструктор используется для инициализации полей структуры. Переопределить базовый конструктор по умолчанию (без параметров) нельзя.

Определим для класса Shape конструктор для установки цвета фигуры.

```
public class Shape
{
    public Shape(Color color)
    {
        Color = color;
    }
    …
}
```


При определении конструктора с параметрами в классе, конструктор без параметров становится недоступным. Его необходимо определить явно.

Создадим для класса Circle конструктор, который будет принимать необходимые параметры для создания объекта.

```
public class Circle : Shape
{
    public Circle() : base(Color.Red) {}

    public Circle(Point pos,
                  double radius,
                  Color color) : base(color)
    {
        Center = pos;
        Radius = radius;
    }

    public Circle(Point pos,
                  double radius) : base(Color.Red)
    {
        Center = pos;
        Radius = radius;
    }

}
```

Устанавливаем в конструкторе позицию и радиус окружности. Но поскольку в базовом классе Shape отсутствует конструктор без параметров, то нам необходимо из конструктора Circle вызвать конструктор базового класса с установкой цвета. Для вызова нужного конструктора базового класса используется ключевое слово ***base***.

Для класса можно определить несколько конструкторов с различным набором параметров.

4.3 Полиморфизм.

Создадим в продолжение к нашему примеру еще один класс Line, для того чтобы сделать минимальную иерархию классов, которую мы будем использовать в дальнейшем для примеров.

```
public class Line : Shape
{
    public Line(Point startPos,
                Point endPos,
                Color color)
        : base(color)
    {
        Start = startPos;
        End = endPos;
    }

    public Point Start { get; set; }
    public Point End { get; set; }
}
```

Для справки, кто забыл, полиморфизм это возможность изменять поведение объектов. Если вспомнить основные концепции языка C#, то все типы унаследованы от System.Object. Каждый объект имеет метод ToString(), который можно переопределить. Сделаем это. Для этого необходимо использовать новое ключевое слово ***override***.

```
public class Shape
{
    …
    public override string ToString()
    {
        return string.Format("Цвет: {0}", Color);
    }
}
```



```
public class Circle : Shape
{
   public override string ToString()
   {
      string res = "Окружность:\n";
      res += string.Format("Центр: {0}\n", Center);
      res += string.Format("Радиус: {0}\n", Radius);
      res += base.ToString();

      return res;
   }
}

public class Line : Shape
{
   …
   public override string ToString()
   {
      string res = "Линия:\n";
      res += string.Format("Точка 1: {0}\n", Start);
      res += string.Format("Точка 2: {0}\n", End);
      res += base.ToString();

      return res;
   }
}
```

Для вызова базовой реализации функции используется ключевое слово ***base***. В нашем примере к создаваемой строке, мы добавили базовую реализацию метода ToString() в типе Shape, которая возвразает цвет объекта.

В метод string.Format мы подставляем объекты типа Point. Поэтому необходимо переопределить метод ToString и у структурного типа Point.

```
public override string ToString()
{
    return string.Format("[{0},{1}]", X, Y);
}
```

При переопределении метода ToString() мы уже включили в работу один из проинциипов ООП – полиморфизм. Но метод ToString объявлен в типе System.Object и мы его не создавали, а лишь изменяли его поведение (override).

Создадим в нашем примере метод Draw для реализации классического примера с демонстрацией полиморфизма. В классе Shape реализация будет пустой, а для классов Circle и Line выполним условный процесс рисования в консоль… ☺

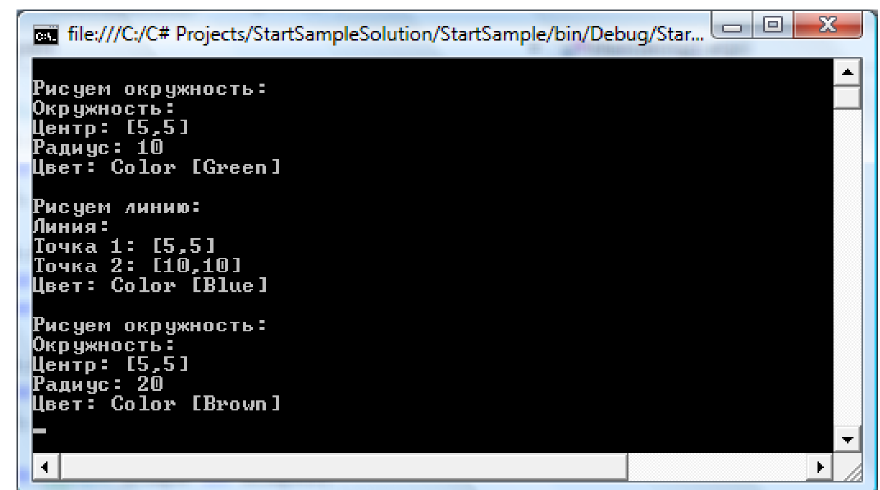
```
public class Shape
{    …
    public virtual void Draw()
    {
    }
}

public class Circle : Shape
{    …
    public override void Draw()
    {
       Console.WriteLine("\nРисуем окружность:");
       Console.WriteLine(this.ToString());
    }
}

 public class Circle : Shape
{    …
    public override void Draw()
    {
       Console.WriteLine("\nРисуем линию:");
       Console.WriteLine(this.ToString());
    }
}
```

Для определения метода, который в дальнейшем можно будет переопределить используется ключевое слово ***virtual*** (аналогично языку C++).

Теперь осталось лишь написать реализацию метода Main:


```
static void Main(string[] args)
{
    List<Shape> shapes = new List<Shape>();
    shapes.Add(new Circle(new Point(5, 5),
        10, Color.Green));

    shapes.Add(new Line(new Point(5, 5),
        new Point(10,10), Color.Blue));

    shapes.Add(new Circle(new Point(5, 5),
        20, Color.Brown));

    foreach(Shape shape in shapes)
    {
        //Вот и полиморфизм :)
        shape.Draw();
    }

    Console.ReadLine();
}
```

Теперь осталось запустить наш пример и посмотреть на результат.



Почему в языке C# при перегрузке метода решили использовать ключевое слово override? Ведь в языке C++ его не было. По началу кажется, что это лишь добавляет лишнюю работу. Но разработчики языка C# старались оградить программистов от всех возможных ошибок.

Допустим, что в базовом библиотечном классе был метод f() помеченный как виртуальный. После этого в классе-наследнике другой программист переопределяет этот метод без использования каких-либо ключевых слов. В процессе разработки приложения первый программист решает, что в метод f() надо пердать параметр, и добавляет его. После этого в языке C++ приложение успешно компилируется, но полиморфизм разорван… в иерархии есть два разных метода f() с разным количеством параметров. В результате эту ошибку надо обнаружить тестерам, после этого найти ее в коде и исправить.

В языке C# эта ситуация выявится на уровне компилятора. В момент компиляции будет выявлено, что пытаемся переопределить метод, которого не существует в базовых классах. Т.е. необходимость использования ключевого слова override дисцеплинирует разработчиков, они явно указывают на то, что необходимо реализовать полиморфное поведение. И исключает ошибку при модификации кода, которая может разорвать цепочку полиморфизма.

Вспомним еще несколько нововведений в языке C#, которые позволяют избежать ошибок. Обязательность использования bool-вых значений в условных операторах и циклах. Это исключает возможные ошибки, когда вместо оператора сравнения «==» ставят оператор присваивания «=».

В языке C# компилятор отслеживает любую попытку использования неинициализированной переменной и выдает

ошибку. В языке C++ невнимательность разработчика, который забыл инициализировать переменную, могла привести к ужасной ситуации, когда в Debug режиме код успешно исполняется, т.к. менеджер памяти инициализирует выделяемую память нулями. А в Release режиме в этой переменной может оказаться любое «мусорное» значение… со всеми печальными последствиями.

4.4 Запрещение наследования.

Если нужно запретить наследование от пользовательского класса, необходимо использовать ключевое слово ***sealed***.

```
public sealed class Circle : Shape {…}
```

При попытке создать наследника от класса Circle на уровне компилятора будет выдана соотвествующая ошибка. Модификатор sealed можер рассматривать как еще один элемент инкапсуляции данных от нежелательного наследования.

Эта же возможность в скрытом виде используется при запрещении наследования при определении структурного типа и перечисления. (struct и enum).

4.5 Абстрактные методы и классы.

Вернемся к нашему примеру. Класс Shape содержит пустую реализацию метода Draw и создавать экземпляры самого класса Shape не имеет смысла. Для создания абстрактных методов и классов необходимо использовать ключевое слово ***abstract***.

```
public abstract class Shape
{   …
    public abstract void Draw();
}
```

4.6 Разрывание полиморфизма, свойств и методов.

В языке C# ключевое слово ***new*** может использоваться для разрывания цепочек полиморфизма, свойств и методов. Рассмотрим цепочку унаследованных классов A B C и D. Для класса C сделаем разрыв метода f().

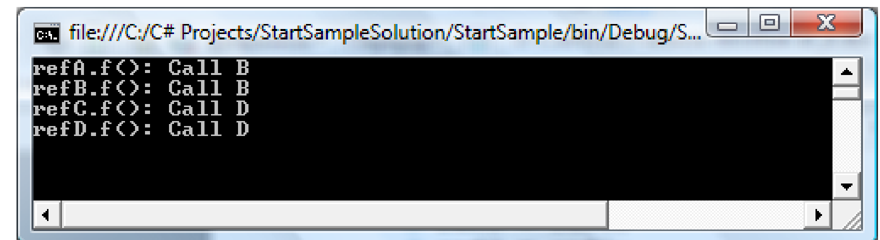
```
public class A
{
    public virtual void f(string callFrom)
    { Console.WriteLine(callFrom + ": Call A"); }
}

public class B : A
{
    public override void f(string callFrom)
    { Console.WriteLine(callFrom + ": Call B"); }
}

public class C : B
{
    public new virtual void f(string callFrom)
    { Console.WriteLine(callFrom + ": Call C"); }
}
```



```
public class D : C
{
    public override void f(string callFrom)
    { Console.WriteLine(callFrom + ": Call D"); }
}
```

Реализуем метод Main следующим образом:

```
static void Main(string[] args)
{
    D refD = new D();

    A refA = refD;
    B refB = refD;
    C refC = refD;

    refA.f("refA.f()");
    refB.f("refB.f()");
    refC.f("refC.f()");
    refD.f("refD.f()");

    Console.ReadLine();
}
```



По результату исполнения примера видно, что слово new разорвало цепочку полиморфизма на две части для классов AB и CD.

Наличие самого полиморфизма для разрывания цепочек не обязательно. Если на каждом уровне иерархии необходимо реализовать свойство или метод с одинаковым именем и

параметрами, то так же можно использовать ключевое слово new.

При объяснении разрыва методов и свойств, многие задают вопрос: А где эту возможность можно происпользовать на практике. Для чего введена эта возможность?

Рассмотрим случай, когда есть две паралельные иерархии. Иерархия объектов и паралельная ей иерархия в базе данных. При создании объекта устанавливается привязка к базе данных через поле _refToDb. Необходимо реализовать такое поведение, чтобы в зависимости от того, к какому типу мы привели объект, используя свойство Db мы получали объект из базы данных приведенный к соотвествующему типу. Создаем свойство Db и используем оператор new для определения этого свойства индивидуально для каждого класса A B и C.

```
public class A
{
    protected object _refToDb;

    public DbA Db
    { get { return _refToDb as DbA; } }
}

public class B : A
{
    public new DbB Db
    { get { return _refToDb as DbB; } }
}

public class C : B
{
    public new DbC Db
    { get { return _refToDb as DbC; } }
}
```


4.7 Интерфейсы.

В язык C# полноценным образом встроена поддержка интерфейсов. В C++ можно было создавать интерфейсы косвенным образом, путем определения класса с набором абстрактных методов.

Напомним, что задача интерфейсов отделить реализацию от декларации. При этом один интерфейс может определять множество различных вариантов реализаций. В базовой библиотеке типов .NET для многих типов или групп типов определена интерфейсная модель. Принято, называть интерфейс с буквы «I» с последующим именем интерфейса. (IName).

Например, все списки, массивы и словари поддерживают интерфейс IList, который позволяет однотипно работать с объектами любого типа, поддерживающих этот интерфейс. Ниже приведем пример определения этого интерфейса.

```
public interface IList
{
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
}
```

Для определения интерфейса используется ключевое слово ***interface.*** В интерфейсе можно определять методы,

свойства, индексаторы и события. При объявлении членов интерфейса не надо использовать модификаторы доступа, т.к. интерфейс по своей сути предполагает публичный (public) доступ к своим членам. Реализацию методов и свойств писать не надо.

В примере выше показано объявление методов, свойств и индексатора в интерфейсе. Создадим класс-оболочку «ListWrp» (Design Paterrn: Оболочка), и реализуем поддержку этого интерфейса путем делегирования вызовов в реальный класс «List». Поскольку интерфейс IList поддерживает интерфейсы ICollection и IEnumerable, нам придется реализовать и их.

```
public class ListWrp : IList
{
    private IList _list = new List<object>();

    #region IList Members

    public int Add(object value)
    { return _list.Add(value); }

    public void Clear()
    { _list.Clear(); }

    public bool Contains(object value)
    { return _list.Contains(value); }

    public int IndexOf(object value)
    { return _list.IndexOf(value); }

    public void Insert(int index, object value)
    { _list.Insert(index, value); }

    public bool IsFixedSize
    { get { return _list.IsFixedSize;  } }

    public bool IsReadOnly
    { get { return _list.IsReadOnly; } }
```



```
    public void Remove(object value)
    { _list.Remove(value); }

    public void RemoveAt(int index)
    { _list.RemoveAt(index); }

    public object this[int index]
    {
        get { return _list[index]; }
        set { _list[index] = value; }
    }

    #endregion

    #region ICollection Members

    public void CopyTo(Array array, int index)
    { _list.CopyTo(array, index); }

    public int Count
    { get { return _list.Count; } }

    public bool IsSynchronized
    { get { return _list.IsSynchronized; } }

    public object SyncRoot
    { get { return _list.SyncRoot; } }

    #endregion

    #region IEnumerable Members

    //Явная реализация Explicit
    IEnumerator IEnumerable.GetEnumerator()
    { return _list.GetEnumerator(); }

    #endregion
}
```

Посмотрим на наш класс-оболочку ListWrp. Он поддерживает интерфейс IList, который содержит набор

членов для активной работы со списком (добавление, удаление). Интерфейс ICollection позволяет получить число объектов, откопировать коллекцию в массив и обеспечивает синхронизацию доступа к коллекции между потоками.

Посредством пары интерфейсов IEnumerator и IEnumerable реализуется возможность подставлять любую коллекцию в конструкцию foreach. «IEnumerator» это объект-итератор, который умеет перебирать объекты в коллекции, и именно он используется внутри конструкции «foreach». А на место коллекции можно подставить объект, обязательно реализующий метод ***IEnumerable***, который содержит лишь один метод для получения ***IEnumerator***-а.

Еще раз вдумаемся в идеологию предназначения интерфейсов. Формулировка предназначения интерфейсов «отделение реализации от декларации» – абсолютно верная, но очень “сухая”. Часто не все программисты, и особенно студенты, могут развить мысль о предназначении интерфейсов, кроме этой сухой фразы.

Интерфейсы это средство разделения реализации и декларации в пространстве и времени. Это означает, что интерфейсы помогают разделить работу внутри группы разработчиков, между группами разработчиков, между компаниями - это пространственное разделение. Разделение во времени означает, что сама реализация может быть разбита на части, каждая из которых может быть выполнена независимо в любое время.

Одиночный интерфейс, сам по себе ценности не представляет. ***Группа логически связанных интерфейсов называется интерфейсной моделью***. Интерфейсная модель может быть полностью отделена от любой реализации. Ее можно рассматривать как некоторое описание стандарта

модели, выраженное в конструкциях языка программирования.

Рассмотрим классическую интерфейсную модель сортировки. Сортировка должна принимать массив, в котором можно осуществлять перестановку элементов. Алгоритм сортировки может быть любым, элементы в массиве могут быть любыми и должны лишь обеспечивать возможность сравнения, так же должна быть возможность различных вариантов сравнения одного и того же элемента. Например, паспорта граждан можно отсортировать по дате выдачи, или по ФИО или некоторыми комбинированными вариантами. Т.е. вариантов сортировки элементов одного типа может быть много. Напишем стандарт этой модели в виде программного кода.

```
public interface ISort
{
    void Sort(IArray list);
    void Sort(IArray list, IComparer comparer);
}

public interface IArray
{
    int Count { get; }
    object this[int index] { get; set; }
}

public interface IComparer
{
    int Compare(object obj1, object obj2);
}

public interface IComparable
{
    int Compare(object obj);
}

public class NotComparableException : Exception {}
```

Т.е. массив «IArray» обеспечивает возможность получить количество элементов и переставлять элементы, «ISort» это интерфейс сортировки, который может реализовывать любой алгоритм и содержит два метода. Первый позволяет отсортировать список, при этом каждый объект в списке должен поддерживать интерфейс ***IComparable*** для возможности сравнения двух элементов. Второй метод в качестве параметра получает объект поддерживающий интерфейс ***IComparer*** для сравнения двух объектов из списка. Программисты, которые будут использовать эту модель смогут написать нужное число вариантов сравнения элементов и подставлять в сортировщик нужный «comparer». Интерфейсная модель может быть дополнена ***служебными типами***. Например, если объекты в первом методе не поддерживают интерфейс ***IComparable***, то будем кидать исключение ***NotComparableException***. Аналогично будем поступать при реализации ***IComparer***, если пришли объекты неправильного тиипа.

Интерфейсная модель это программное описание стандарта модели без какой-либо реализации. Интерфейсная модель может дополняться набором служебных типов с реализацией.

Соотвественно построение интерфейсной модели полностью отделяется от реализации, которая может быть крайне сложной. Эта особенность используется архитектором приложения, который отвечает за организацию взаимодействия различных компонентов системы. Т.е. стандарт системы выраженный в текстово-графическом описании (UML и т.п.), может быть представлен в виде конкретного программного кода интерфейсной модели. Что позволяет побить общую реализацию на части как в пространстве так и во времени.

Базовая библиотека .NET определяет большое количество интерфейсов и включает множество небольших интерфейсных моделей и части их реализации. Напрмер, алгоритмы сортировки реализованы внутри списков и массивов и все числовые типы поддерживают интерфейс IComparable. Пользователю лишь остается для своих типов определить этот интерфейс или реализовать нужный «comparer».

Вернемся к нашему примеру «ListWrp». Заметим, что интерфейс IEnumerable реализован непривычным образом. ***В языке C# есть два вида реализации интерфейсов Implicit и Explicit. (Явная и неявная реализация).***

```
#region IEnumerable Members

//Явная реализация Explicit
IEnumerator IEnumerable.GetEnumerator()
{ return _list.GetEnumerator(); }

#endregion

#region IEnumerable Members

//Неявная реализация Implicit
public IEnumerator GetEnumerator()
{ return _list.GetEnumerator(); }

#endregion
```

Отличие этих двух реализаций в том, что для вызова метода с явной реализацией необходимо привести объект к интерфейсу. При неявной реализации метод может быть вызван как от интерфейса, так и от объекта. Эта возможность обеспечивает еще один уровень инкапсуляции данных.

Метод *GetEnumerator()* нужен для работы конструкции *foreach*. Было бы не плохо его скрыть от «посторонних глаз».

Т.е. пока объект не будет приведен к интерфейсу IEnumerable, у него нельзя будет вызвать метод *GetEnumerator()*. Явная реализация обеспечивает эту возможность. Метод *GetEnumerator()* не будет видно даже в подсказках самого объекта.

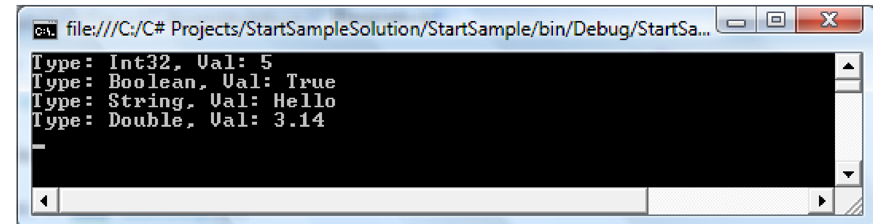
```
IEnumerator enumerator;
ListWrp list = new ListWrp();
//Ошибка: list не имеет метод GetEnumerator
enumerator = list.GetEnumerator();

IEnumerable enumerable = list;
//Корректно
enumerator = enumerable.GetEnumerator();
```

Явная реализация методов интерфейса дает возможность реализовать два одинаковых по декларации члена из разных интерфейсов. При этом не произойдет конфликта имен, а при вызове метода от интерфейса будет вызвана соотвествующая явная реализация.

Закончим наш пример реализовав метод Main:

```
static void Main(string[] args)
{
    IList list = new ListWrp();

    list.Add(5);
    list.Add(true);
    list.Add("Hello");
    list.Add(3.14);

    foreach( object obj in list )
    {
        Console.WriteLine("Type: {0}, Val: {1}",
                    obj.GetType().Name, obj);
    }

    Console.ReadLine();
}
```


Создадим объект типа *ListWrp*, приведем его к интерфейсу *IList*, добавим несколько объектов и подставим в конструкцию foreach.



5. C#. Делегаты и события.

Еще одним нововведением языка C# являются делегаты и события. На их основе строится модель сообщений между объектами. Все взаимодействие в библиотеке Windows.Forms, которая позволяет создавать графический пользовательский интерфейс, основано на делегатах и событиях.

5.1 Делегаты.

В языке C++ можно было передавать адреса функций в качестве параметров. Как правило, это использовалось для организации систем событий. (MFC).

В языке C# для этих целей предусмотрены ***делегаты***. Делегат это пользовательский класс, который внутри себя хранит указатель на метод. Кроме того, ему необходимо описать сигнатуру метода (параметры метода и возвращаемое значение).

Определение нового класса делегата выглядит следующим образом (продолжаем наш пример из прошлой главы):

```
public delegate void ListModify( ListWrp list,
                                 object elem);
```

Поскольку delegate является классом, то определять его можно прямо в пространстве имен. (Можно сделать и вложенным).

Если убрать ключевое слово ***delegate***, то мы получим определение метода, который определяется этим делегатом. На самом деле компилятор автоматически создаст следующее определение класса:


```
public class ListModify : System.MulticastDelegate
{
    //Конструктор нового экземпляра делегата
    ListModify(object target, int methodPtr);

    //Синхронная версия вызова метода
    public void Invoke(ListWrp list, object elem);

    //Асинхронная версия вызова метода
    //...//
}
```

Т.е. будет создан класс *ListModify*. Конструктор этого класса будет принимать объект и адрес метода. Если при создании делегата мы подставим метод экземпляра объекта, то targed будет содержать ссылку объекта, у которого надо вызвать метод с адресом *methodPtr*. Если в делегат подставят статический метод класса, то в конструкторе targed будет пустой ссылкой (null), а methodPtr будет содержать адрес статического метода.

Имея экземпляр делегата, можно вызвать метод, который этот делегат «оборачивает». Для синхронной версии вызова это метод Invoke, который по параметрам и возвращаемому значению идентичен тому, что было задано при объявлении нового типа делегата.

Для асинхронного вызова метода предусмотрены функции BeginInvoke, EndInvoke, но мы в данном учебном пособии не будем их рассматривать.

Продолжим наш пример и создадим новый класс *ListReactor*, который будет отслеживать добавление и удаление элементов из нашего класса ListWrp.

```
class ListReactor
{
    public void OnAddElem(ListWrp list, object elem)
    {
        Console.WriteLine("Add: List: {0} Val {1}",
                                    list, elem);
    }

    public void OnRemElem(ListWrp list, object elem)
    {
        Console.WriteLine("Rem: List: {0} Val {1}",
                                    list, elem);
    }
}
```

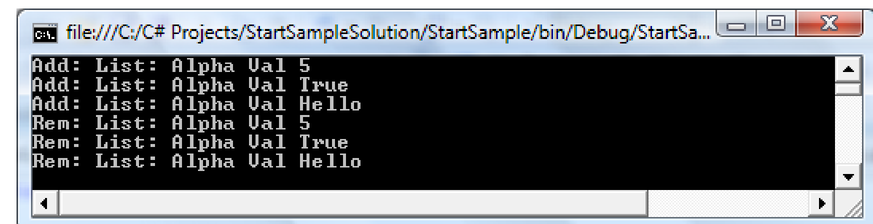
Сделаем для нашего списка *ListWrp* конструктор с именем, свойство для имени и переопределим метод *ToString*. (Для того чтобы списки можно было уникально идентифицировать в нашем примере).

```
    public class ListWrp : IList
    {
        public ListWrp(string name)
        { Name = name; }

        public string Name { get; set; }

        public override string ToString()
        { return Name; }
        …
     }
```

Реализуем теперь метод Main. В нем мы создадим новый объект *ListReactor*, создадим наш список и добавим туда несколько произвольных элементов. Создадим два делегата *onAdd* и *onRem*, которые будут содержать в себе адреса соотвествующих методов реактора. И сделаем метод *ProcList*, который принимает в качестве параметров список для обработки и делегат.

Цель этого примера наглядно показать, что делегат это всего-лишь объект, который можно создавать через опретор *new* и передавать в качестве парамера в методы. Продемонстрировать вызов метода через делегат.

```
static void Main(string[] args)
{
    ListReactor reactor = new ListReactor();

    ListWrp list = new ListWrp("Alpha");

    list.Add(5);
    list.Add(true);
    list.Add("Hello");

    ListModify onAdd =
                new ListModify(reactor.OnAddElem);
    ListModify onRem =
                new ListModify(reactor.OnRemElem);

    ProcList(list, onAdd);
    ProcList(list, onRem);

    Console.ReadLine();
}

private static void ProcList(ListWrp list,
                             ListModify listDelegat)
{
    foreach(object obj in list)
        listDelegat.Invoke(list, obj);
}
```



Вызов делегата можно выполнить любым из двух способов: (в любом случае компилятор это корректно воспримет и позовет метод Invoke).

```
listDelegat.Invoke(list, obj);
listDelegat(list, obj);
```

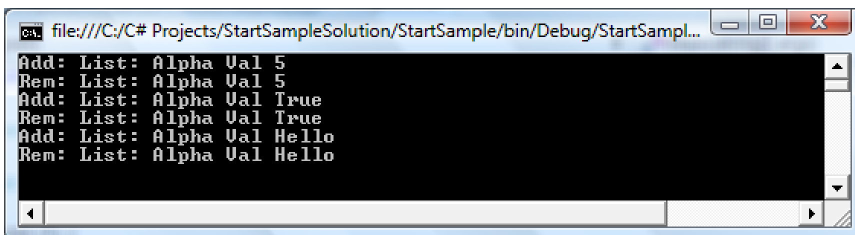
Для упрощения создания делегата так же предусмотрен упрощенный вид записи:

```
ListModify onAdd = new ListModify(reactor.OnAddElem);
ListModify onAdd = reactor.OnAddElem;
```

Рассмотрим класс ***System.MulticastDelegate***. Его название не случайно. Делегаты можно между собой складывать и вычитать. В результате суммы получается новый объект *MulticastDelegate*, который содержит несколько адресов функций и при вызове делегата, эти методы будут вызваны в соотвествующем порядке. Перепишем часть метода *Main* следующим образом:

```
ListModify onAdd = reactor.OnAddElem;
ListModify onRem = reactor.OnRemElem;

ProcList(list, onAdd + onRem);
```

Теперь, итогом работы нашего примера будет следующая последовательность вызовов методов реактора:



5.2 События.

В предыдущем разделе нам пришлось «искусственно» вызывать метода реактора, используя делегаты. Теперь нам надо добиться такого поведения, чтобы реактор срабатывал автоматически, как только в наш список добавляются или удаляются объекты.

Для этого в нашем списке необходимо реализовать события. События относятся к членам класса.

```
public class ListWrp : IList
{    …
    public int Add(object value)
    {
        if (AddElem != null)
            AddElem(this, value);

        return _list.Add(value);
    }

    public void Insert(int index, object value)
    {
        if (AddElem != null)
            AddElem(this, value);

        _list.Insert(index, value);
    }

    public void RemoveAt(int index)
    {
        if (RemElem != null)
            RemElem(this, _list[index]);

        _list.RemoveAt(index);
    }
    …
    public static event ListModify AddElem;
    public static event ListModify RemElem;
}
```

Для нашего списка мы реализовали два статических события *AddElem* и *RemElem*. Поскольку событие является членом класса, оно может принадлежать конкретному экземпляру объекта, так и быть статическим членом класса. В нашем примере нам надо обеспечить реакцию на добавления и удаления в любых списках, при этом сам список, который послал событие, приходит в виде параметра метода реактора. Этим и обусловлено решение сделать события статическими. Для событий диалогов, форм и расположенных на них элементах события будут принадлежать конкретным экземплярам объектов.

```
public static event ListModify RemElem;
```

Рассмотрим объявление события. Для этого используется ключевое слово ***event***. Для события так же необходимо указать делегат, которым это событие будет обрабатываться.

Для генерации событий в методах добавления и удаления элементов списка, необходимо позвать событие. (Проверка на пустое значение необходима для того, чтобы избежать исключения в случае, когда никто не подписался на событие).

```
public int Add(object value)
{
    if (AddElem != null)
        AddElem(this, value);

    return _list.Add(value);
}
```

Параметры, используемые в событии, должны по своей сигнатуре строго совпадать с параметрами делегата ассоциированного с этим событием. На уровне компилятора проводится проверка корректности всех подставляемых параметров, их типов и порядка следования.

Теперь осталось «подписаться» на события. Реализуем по новому метод *Main*.

```
static void Main(string[] args)
{
    ListReactor reactor = new ListReactor();

    //Подписываемся на события
    ListWrp.AddElem += reactor.OnAddElem;
    ListWrp.RemElem += reactor.OnRemElem;

    ListWrp listA = new ListWrp("Alpha");
    ListWrp listB = new ListWrp("Beta");

    //Добавляем объекты в списки
    listA.Add(5);
    listB.Add(true);
    listA.Add("Hello Alpha");
    listB.Add("Hello Beta");
    //Отключаем реактор от события добавления
    ListWrp.AddElem -= reactor.OnAddElem;
    listB.Add(3.14);

    Console.WriteLine(string.Empty);
    //Удаляем объекты
    listA.RemoveAt(0);
    listB.RemoveAt(0);
    //Отключаем реактор от события удаления
    ListWrp.RemElem -= reactor.OnRemElem;
    listB.RemoveAt(0);

    Console.ReadLine();
}
```

file:///C:/C# Projects/StartSampleSolution/StartSample/bin/Debug/StartSamp...

```
Add: List: Alpha Val 5
Add: List: Beta Val True
Add: List: Alpha Val Hello Alpha
Add: List: Beta Val Hello Beta

Rem: List: Alpha Val 5
Rem: List: Beta Val True
```

Внимательно ознакомтесь с кодом метода *Main*. После того как создали реактор, происходит подключение к событиям.

```
//Подписываемся на события
ListWrp.AddElem += reactor.OnAddElem;
ListWrp.RemElem += reactor.OnRemElem;
```

С этого момента наш реактор начинает ловить все события добавления и удаления элементов списка.

Для того, чтобы отключиться от обработки события нам необходимо отнять делегат от события.

```
//Отключаем реактор от события добавления
ListWrp.AddElem -= reactor.OnAddElem;
//Отключаем реактор от события удаления
ListWrp.RemElem -= reactor.OnRemElem;
```

После отключения реактора мы перестаем реагировать на добавления и удаления элементов списка. Метода реактора добавления и удаления можно сделать статическими, тогда код подключения будет выглядить следующим образом:

```
static void Main(string[] args)
{
    //Подписываемся на события
    ListWrp.AddElem += ListReactor.OnAddElem;
    ListWrp.RemElem += ListReactor.OnRemElem;
    …
}
```

Резюме: Композиция событий и делегатов обеспечивает новую модель генерации и обработки сообщений прямо на уровне языка C#. При этом, никто не запрещает использовать стандартную интерфейсную модель реализации обработки событий. Библиотека .NET «Windows.Forms» использует модель обработки сообщений на основе делегатов и событий.

6. RTTI. Рефлексия. Атрибуты.

В этой главе мы обсудим приведение типов, идентификацию типов во время исполнения (RTTI). Познакомимся с понятием рефлексии и использованием атрибутов. Последние два вопроса полностью отсутствуют в языке C++ и для большинства читателей будут новыми. Так же мы еще раз модифицируем пример со списком *ListWrp* и реализуем новый пример «страница свойств», использующий новые возможности языка C# - рефлексию и атрибуты.

6.1 Приведение типов. RTTI.

В данном разделе рассмотрим вопрос приведения типов. Небезопасное приведение типов позволяет привести один тип к другому. Если приведение не может быть выполнено будет брошен *InvalidCastException*.

```
//Небезопасное приведение типа
A a = (A)refToObj;
```

При приведении типа от верхнего к нижнему, компилятор сам распознает корректность приведения и никаких специальных действий при этом предпринимать не надо. Если один тип не может быть приведен к другому, то компилятор выдаст ошибку.

```
// Приведение типа вниз
IList listA = new ListWrp("Alpha");
```

При создании проекта на C++ в настройках проекта можно было включить опцию поддержки RTTI (Run Time Type Identification). Что же это такое? При выделении куска памяти происходит его маркировка, к какому типу он относится. Для каждого типа создается дескриптор, по которому во время

выполнения программы у объекта можно определить его назначение. В языке C++ для безопасного приведения типа «вверх» использовался оператор:

A* pToType = dynamic_cast<A*>(refToObj);

Смысл безопасного приведения заключается в том, что если объект поддерживает тип, к которому выполняется приведение, то приведение успешно выполняется. В противном случае ссылка или указатель устанавливается в нулевое значение (null).

Классическое RTTI должно включать несколько возможностей: Проверять является ли объект точным типом T ***isA(TypeDesc)***; Можно ли привести объект к типу T ***isKindOf(TypeDesc)***; Возможность получить у типа и у объекта ***TypeDesc***.

C# реализует полноценное RTTI на базовом уровне языка. Базовый класс *System.Object* имеет метод *GetType()*, который возвращает объект типа *Type*. Этот объект и является «дескриптором типа». Для получения дескриптора по типу, необходимо воспользоваться ключевым словом *typeof(T)*. Этот оператор вернет объект типа *Type* для указанного типа.

```
Type typeOfObj = obj.GetType();
Type typeOfListWrp = typeof (ListWrp);

object obj = new ListWrp("RTTI");
//isA()
bool isA = (obj.GetType() == typeof(ListWrp));
```

Для реализации принципа RTTI isKindOf(T) предусмотрено два оператора ***is*** и ***as***. Оператор ***is*** возвращает *bool* значение является ли объект типом T. Оператор ***as*** обеспечивает безопасное приведение типов.


```
//True  obj поддерживает T
//False obj не поддерживает T
bool isKindOfT = obj is ListWrp;

//Безопасное приведение к типу T
ListWrp list = obj as ListWrp;
if ( list == null )
{
    //obj не поддерживает тип T
}
```

Теперь закончим наш пример со списком ListWrp. Создадим новый класс Record с двумя свойствами: Имя и номер.

```
public class Record
{
    public Record(string name, int number)
    {
        Name = name;
        Number = number;
    }

    public string Name { get; set; }
    public int Number { get; set; }

    public override string ToString()
    {
       return String.Format("Name: {0}, ID: {1}",
                                       Name, Number);
    }
}
```

Создадим два класса для сравнения наших записей, поддерживающих интерфейс *IComparer<object>*. В классе ListWrp сделаем метод Sort и Print.

```
public void Sort(IComparer<object> comparer)
{
   (_list as List<object>).Sort(comparer);
}
```

```
public void Print()
{
    foreach (object obj in this)
        Console.WriteLine(obj);
}

public class NameComparer : IComparer<object>
{
    #region IComparer<object> Members

    int IComparer<object>.Compare(object x, object y)
    {
        Record rX = x as Record;
        Record rY = y as Record;
        if ( (rX == null)||(rY == null) )
            throw new InvalidCastException();

        return rX.Name.CompareTo(rY.Name);
    }

    #endregion
}

public class NumComparer : IComparer<object>
{
    #region IComparer<object> Members

    int IComparer<object>.Compare(object x, object y)
    {
        Record rX = x as Record;
        Record rY = y as Record;
        if ((rX == null) || (rY == null))
            throw new InvalidCastException();

        return rX.Number.CompareTo(rY.Number);
    }

    #endregion
}
```

В методе Sort и двух реализациях метода Compare мы применили безопасное приведение типов используя оператор

as. В случае два сравниваемых объекта не являются объектами типа Record, то генерим исключение.

В методе Sort мы не выполняем проверку на то, что IList привелся к типу List<object>, т.к. при создании объекта мы выполнили создание внутреннего списка используя этот тип. Можно было использовать и небезопасное приведение типа, но поскольку язык C# реализует RTTI в полной мере, рекомендуется полностью отказаться от небезопасного приведения типов.

```
static void Main(string[] args)
{
    ListWrp list = new ListWrp("Sort Test");
    list.Add(new Record("Beta", 3));
    list.Add(new Record("Alpha", 7));
    list.Add(new Record("Gama", 1));

    Console.WriteLine("Sort By Name");
    list.Sort(new NameComparer());
    list.Print();

    Console.WriteLine("\nSort By Number");
    list.Sort(new NumComparer());
    list.Print();

    Console.ReadLine();
}
```

```
file:///C:/C# Projects/StartSampleSolution/StartSample/bin/Debug/StartS...
Sort By Name
Name: Alpha, ID: 7
Name: Beta, ID: 3
Name: Gama, ID: 1

Sort By Number
Name: Gama, ID: 1
Name: Beta, ID: 3
Name: Alpha, ID: 7
```

6.2 Рефлексия

Рефлексия это возможность получать и использовать информацию о типах во время исполнения. Ранее мы обсудили концепцию RTTI, которая позволяет во время выполнения производить идентификацию типов и их приведение.

Дескриптор типа, который в языке C# определен типом Type, содержит исчерпывающую информацию о типе. Во время исполнения можно получить доступ к этой информации. Для этого необходимо изучить подробней тип Type. Он содержит большое число методов и свойств, которые могут выдать исчерпывающую информацию о типе. Рассмотрим некоторые свойства.

- `Assembly Assembly { get; }` позволяет получить сборку, в которой определен тип.
- `string FullName { get; }` позволяет получить полное имя типа.
- `string Name { get; }` позволяет получить короткое имя типа. (без пространств имен)
- `Type BaseType { get; }` позволяет получить базовый тип объекта.
- `bool IsAbstract { get; }` идентифицирует является ли тип абстрактным.
- `bool IsArray { get; }` идентифицирует является ли тип массивом.
- `bool IsEnum { get; }` идентифицирует является ли тип перечислением.
- `bool IsInterface { get; }` идентифицирует является ли тип интерфейсом
- `bool IsPublic { get; }` идентифицирует является ли тип публичным или нет.
- `bool IsSealed { get; }` идентифицирует является ли тип запрещенным к дальнейшему наследованию

- `bool IsValueType { get; }` идентифицирует является ли тип структурным.
- `bool IsGenericType { get; }` идентифицирует является ли тип шаблонным
- и т.д. …

Так же содержится набор методов, который позволяет получить информацию обо всех членах класса.

```
ConstructorInfo[] GetConstructors();
EventInfo[] GetEvents();
FieldInfo[] GetFields();
Type[] GetInterfaces();
MethodInfo[] GetMethods();
PropertyInfo[] GetProperties();
```

Классы *Assembly, ConstructorInfo, EventInfo, FieldInfo, MethodInfo, PropertyInfo* содержатся в пространстве имен System.Reflection. Они позволяют получить исчерпывающую информацию по членам класса, что они определяют.

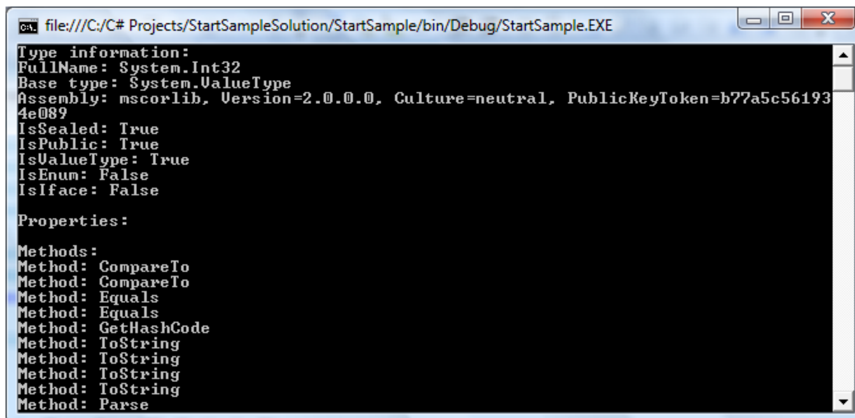
Создадим метод, который выводит информацию о типе.

```
public static void PrintTypeInfo(Type type)

Console.WriteLine("Type information:");
Console.WriteLine("FullName: {0}", type.FullName);
Console.WriteLine("Base type: {0}",
                        type.BaseType);
Console.WriteLine("Assembly: {0}",
                        type.Assembly.FullName);
Console.WriteLine("IsSealed: {0}", type.IsSealed);
Console.WriteLine("IsPublic: {0}", type.IsPublic);
Console.WriteLine("IsValueType: {0}",
                        type.IsValueType);
Console.WriteLine("IsEnum: {0}", type.IsEnum);
Console.WriteLine("IsIface: {0}", type.IsInterface);
```

```
Console.WriteLine("\nProperties:");
PropertyInfo[] props = type.GetProperties();
foreach (PropertyInfo prop in props)
    Console.WriteLine("Property: {0}", prop.Name);

Console.WriteLine("\nMethods:");
MethodInfo[] methods = type.GetMethods();
foreach(MethodInfo method in methods)
    Console.WriteLine("Method: {0}", method.Name);
```

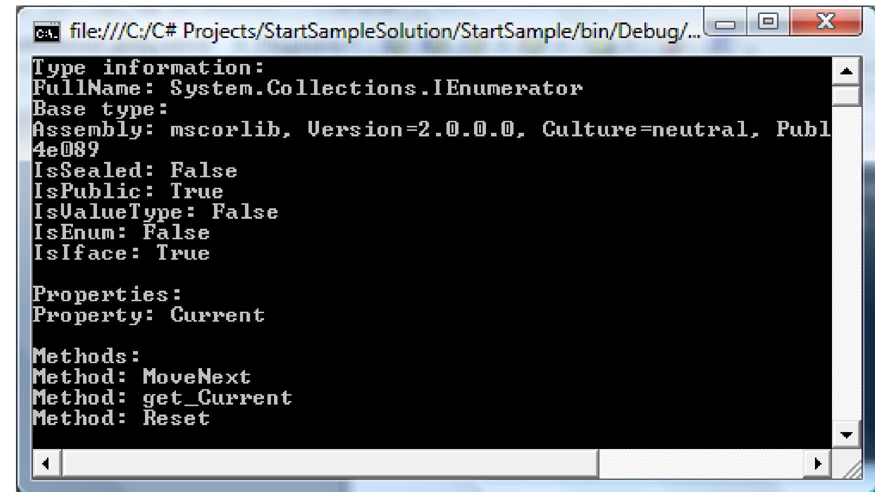
В методе Main вызовем этот метод для некоторых типов:

```
static void Main(string[] args)
{
    PrintTypeInfo(typeof(int));

    Console.ReadLine();
}
```



Выведем информацию для интерфейса IEnumerator:

```
PrintTypeInfo(typeof(IEnumerator));
```

В примере выше, мы рассмотрели возможность пассивного получения информации по типам во время исполнения. Если бы этим и ограничивались возможности рефлексии, то эта особенность не представляла бы никакой ценности. Использование рефлексии мы рассмотрим в примере «страница свойств», а сейчас ознакомимся с еще одной концепцией зыка C# - атрибуты.

6.3 Атрибуты.

В языке C# есть возможность привязывать к классам, их членам и сборкам дополнительную информацию. Для этих целей используются атрибуты. Атрибут это пользовательский тип унаследованный от класса ***System.Attribute***.

В библиотеках .NET есть немало атрибутов предназначенных для тех или иных целей. Но мы рассмотрим создание собственного атрибута и его использование. Определение атрибута полностью идентично созданию нового класса, но необходимо выполнить наследование от атрибута.

```
public class UserAttribute : System.Attribute
{
}
```

Атрибут может содержать все те же самые члены типа, что и класс. Поля, свойства, метода, события, конструкторы. Если атрибуты необходимо снабдить дополнительной информацией, то можно создать необходимое количество свойств. Добавим к нашему атрибуту пару свойств с описанием и номером.

```
public class UserAttribute : System.Attribute
{
    public string Desc { get; set; }

    public int Number { get; set; }
}
```

Теперь добавим атрибут к классу и его членам (свойствам). При этом допускается опускать в имени типа суффикс *Attribute*. Для этого необходимо разместить атрибут в квадратных скобках, а в конструкторе можно производить инициализацию свойств константными данными.

```
[User(Desc = "Человек")]
public class Human
{
    [User(Desc = "ФИО", Number = 1)]
    public string Name { get; set; }

    [User(Desc = "Возраст", Number = 2)]
    public int Age { get; set; }
}
```

Опять же должен появиться вопрос, как можно использовать атрибуты? Используя рефлексию, можно во время выполнения получать атрибуты привязанные к классам и их членам и использовать их в необходимых целях.

Например, можно написать автоматическую сериализацию объектов и помечать аттрибутами те поля, которые необходимо сохранять в файл.

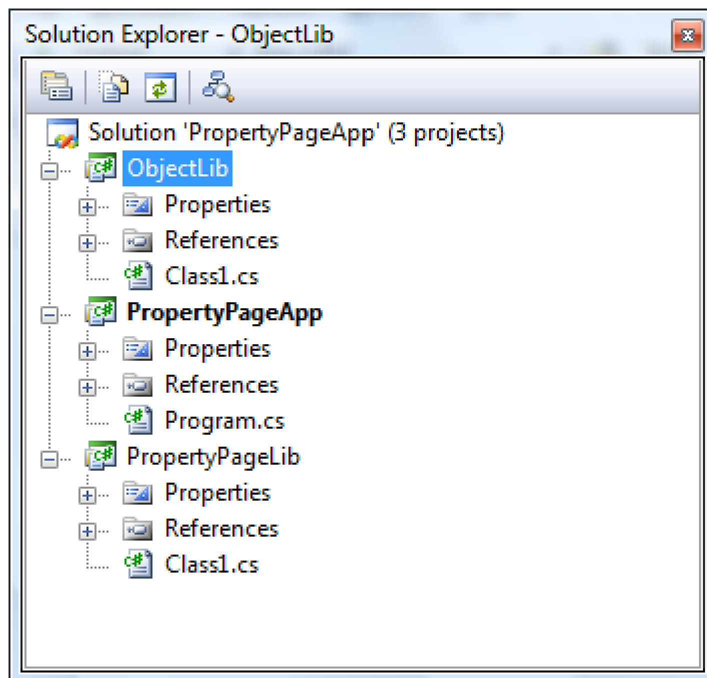
Можно пометить у объекта необходимые свойства атрибутами и использовать эту информацию при отображении страницы свойств. Что мы и продемонстрируем в следующем примере.

Предусмотрена возможность управлять тем на какие типы или члены типа можно навешать атрибут. Выполняется это опять же посредством атрибута ***AttributeUsage*** и перечислением ***AttributeTarget***.

```
public enum AttributeTargets
{
    All,
    Assembly,
    Class,
    Constructor,
    Delegate,
    Enum,
    Event,
    Field,
    Interface,
    Method,
    Module,
    Parameter,
    Property,
    ReturnValue,
    Struct
}

[AttributeUsage( AttributeTargets.Class |
                 AttributeTargets.Property)]
public class UserAttribute : System.Attribute
{
}
```

6.4 Использование рефлексии и атрибутов. Пример «Страница свойств».

Теперь рассмотрим пример использования рефлексии и атрибутов. Создадим новый проект/решение консольного приложения «*PropertyPageApp*». В текущее решение добавим два новых проекта библиотеки типов «PropertyPageLib» и «ObjectsLib». Для этого надо использовать пункт меню «File->Add->New Project». Выбрать тип нового проекта «Class Library» и задать имя проекта. В итоге в окне «Solution Explorer» мы должны получить следующую картинку.



В проекте «*PropertyPageLib*» у нас будет сам функционал для отображения страницы свойств. Опять же будем выводить свойства в консоль, для упрощения примера. Проект

«*ObjectLib*» будет содержать набор библиотечных классов, объекты которого мы будем отображать в странице свойств. В самом exe файле «PropertyPageApp» мы научимся загружать сборку, получать из нее определенные в ней типы и динамически создавать объекты, которые подставим в страницу свойств.

Для начала создадим код в проекте *PropertyPageLib*. Создадим два атрибута. Каждый тип в своем файле.

```
[AttributeUsage(AttributeTargets.Class)]
public class PropTypeAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.Property)]
public class PropAttribute : Attribute
{
    public PropAttribute()
    {
        Text = string.Empty;
        Number = 1;
    }

    public string Text { get; set; }
    public int Number { get; set; }
}
```

Теперь сделаем новый класс «PropertyPage» и в нем сделаем статический метод ShowProperties. Сделаем вложенный класс *PropPageProperty.*

```
public class PropertyPage
{
    internal class PropPageProperty
    {
        public PropertyInfo PropInfo { get; set; }
        public PropAttribute Attr { get; set; }
    }
```

```
    public static void ShowProperties(object obj)
    {
       List<PropPageProperty> props =
                            GetPropToShow(obj);

       …
    }

    private static List<PropPageProperty>
                        GetPropToShow(object obj)
    { … }
}
```

Теперь нам надо реализовать метод *GetPropToShow*, который по объекту вернет список свойств, которые нам надо отобразить. Свойства для отображения должны быть помечены нашим атрибутом PropAttribute. А свойство PropAttribute.Number должно определять порядок, в котором следует выводить атрибуты.

```
var propList = new List<PropPageProperty>();

PropertyInfo[] allProps =
                obj.GetType().GetProperties();
foreach (PropertyInfo prop in allProps)
{
    object[] attrs = prop.GetCustomAttributes(
                     typeof (PropAttribute), false);
    if ( attrs.Count() == 0 )
        continue;

    var prpToShow = new PropPageProperty();
    prpToShow.PropInfo = prop;
    prpToShow.Attr = attrs[0] as PropAttribute;

    propList.Add(prpToShow);
}

propList.Sort(new PropComparer());

return propList;
```


Рассмотрим подробно код этого метода. Мы создали список, который нам будет необходимо вернуть. Получили через тип объекта список всех свойств доступных в типе. Перебираем все свойства в цикле. Используя метод *PropertyInfo.GetCustomAttributes* получаем список атрибутов типа *PropAttribute* навешанных на свойство. Если список пустой, значит атрибуты на свойство не навешаны. Иначе мы создаем свойство (*PropPageProperty*), запоминаем в него PropertyInfo и PropAttribute и добавляем в список. После этого выполняем сортировку с помощью компаратора *PropCompaper*. Этот класс так же сделаем вложенным в *PropertyPage*. Он будет производить сортировку свойств по полю Number, заданного в атрибуте.

```
internal class PropComparer :
                   IComparer<PropPageProperty>
{

    #region IComparer<PropPageProperty> Members

    public int Compare(PropPageProperty x,
                       PropPageProperty y)
    {
        return
           x.Attr.Number.CompareTo(y.Attr.Number);
    }

    #endregion
}
```

Допишем метод *ShowProperties*. Берем свойства и выведем каждое, используя метод *ShowProperty*.

```
public static void ShowProperties(object obj)
{   List<PropPageProperty> props =
                        GetPropToShow(obj);
    foreach( PropPageProperty prop in props )
        ShowPropery(prop, obj);
}
```

```
private static void
        ShowPropery(PropPageProperty prop, object obj)
{
    PropertyInfo propInfo = prop.PropInfo;
    object val = propInfo.GetValue(obj, null);
    Console.WriteLine("{0}: {1}",
                        prop.Attr.Text, val );
}
```

В этом методе мы активно воспользовались рефлексией. По дескриптору совойства (PropertyInfo) объекта и самому объекту получено значение свойства. При этом реальный тип объекта нам впринципе не известен!!!

```
object val = propInfo.GetValue(obj, null);
```

Далее используя поле Text с именем свойства мы выводим его в консоль. На этом библиотечную часть со страницей свойств мы написали.

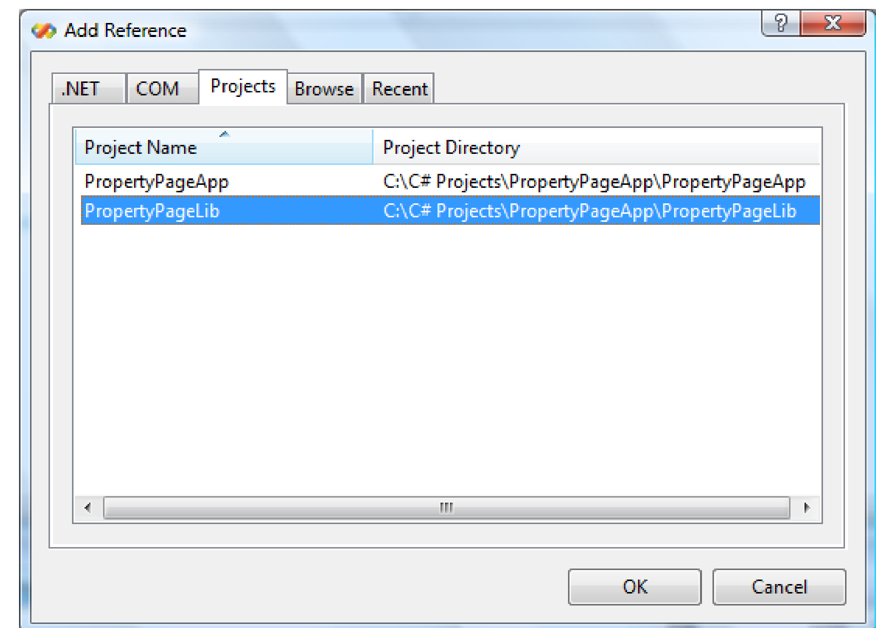
Теперь в проекте «ObjectLib» создадим три класса. Куб и Сфера.

```
    [PropType]
    public class Cube
    {
        public Cube()
        {
            Material = "Лед";
            Mass = "900";
            Length = 1;
        }
        [Prop(Text = "Материал", Number = 2)]
        public string Material { get; set; }
        [Prop(Text = "Масса", Number = 3)]
        public string Mass { get; set; }
        [Prop(Text = "Длина грани", Number = 1)]
        public double Length { get; set; }
    }
```



```
    public class Sphere
    {
        public Sphere()
        {
            Material = "Дерево";
            Mass = "650";
            Radius = 0.5;
        }
        [Prop(Text = "Материал", Number = 2)]
        public string Material { get; set; }
        [Prop(Text = "Масса", Number = 3)]
        public string Mass { get; set; }
        [Prop(Text = "Радиус", Number = 1)]
        public double Radius { get; set; }
        public string NotUsedProp { get; set; }
    }
```

Для того чтобы использовать типы определенные в проекте «PropertyPageLib» надо добавить ссылку (Add Reference), используя вкладку проекты.



Теперь перейдем к проекту «*PropertyPageApp*». Так же добавим ссылку на проект «*PropertyPageLib*». Ссылку на «*ObjectLib*» делать не надо. Добавим ссылку на *System.Windows.Forms* из вкладки «.NET» диалога «Add Reference». Нам понадобится окно открытия нового файла.

```
[STAThread]
static void Main(string[] args)
{
    OpenFileDialog openFileDialog =
                          new OpenFileDialog();
    openFileDialog.Filter =
                        "dll files (*.dll)|*.dll";
    openFileDialog.FilterIndex = 1;
    openFileDialog.RestoreDirectory = true;
    openFileDialog.Multiselect = false;

    if ( openFileDialog.ShowDialog() !=
                                DialogResult.OK )
        return;

    string fileName = openFileDialog.FileName;

    Assembly assembly = Assembly.LoadFrom(fileName);

    foreach (Type type in assembly.GetTypes())
    {
        object[] attrs = type.GetCustomAttributes(
                 typeof (PropTypeAttribute), false);

        if ( attrs.Count() == 0 )
            continue;

        object obj = Activator.CreateInstance(type);
        Console.WriteLine("\nShow properties: {0}",
                              obj.GetType().FullName);
        PropertyPage.ShowProperties(obj);
    }

    Console.ReadLine();
}
```


Разберем метод Main. Отметим что метод пришлось пометить атрибутом [STAThread], указав что у нас однопоточное приложение, что требуется для возможности отображения окна открытия файлов.

Наш проект «*PropertyPageApp*» не ссылается на *ObjectLib*. Поэтому при запуске приложения мы создаем OpenFileDialog и указываем ObjectLib.dll. После чего производим загрузку dll файла.

```
Assembly assembly = Assembly.LoadFrom(fileName);
```

В рамках клнцепции рефлексии, сборка содержит перечень всех определенных в ней типов. Получаем все типы и перебираем их.

```
foreach (Type type in assembly.GetTypes())
```

Опять используем метод извлечения атрибута, только уже от дескриптора типа *Type*.

```
object[] attrs = type.GetCustomAttributes(
                typeof (PropTypeAttribute), false);
```

Если тип помечен нашим атрибутом PropTypeAttribute, то ***!создаем экземпляр объекта имея в руках только дескриптор типа Type.!*** При этом сам тип нам не известен. Используем для этого статический метод класса *System.Activator*.

```
object obj = Activator.CreateInstance(type);
```

Далее, подставляем созданный объект в нашу страницу свойств.

```
PropertyPage.ShowProperties(obj);
```

Получается, что комбинация рефлексии и атрибутов дала нам возможность создать библиотеку для отображения свойств объектов любого типа. При этом всю необходимую информацию о типе свойства и отображаемом тексте и порядке следования свойств, мы достали путем рефйлексии и прикрепленного к свойствам атрибута. При создании объектов из загруженной сборки мы имели в руках только дескриптор типа, а сам тип мы не знали.

Посмотрим, что получилось в результате запуска нашего примера:

```
file:///C:/C# Projects/PropertyPageApp/PropertyPageApp/bin/Debug/Pr...

Show properties: ObjectLib.Cube
Длина грани: 1
Материал: Лед
Масса: 900

Show properties: ObjectLib.Sphere
Радиус: 0.5
Материал: Дерево
Масса: 650
_
```

Мы создали минимальный консольный вариант страницы свойств для отображения свойств объектов. Мы можем отобразить свойства любого объекта у которого на необходимые свойства навешаны атрибуты с нужной информацией.

Используя все эти принципы можно написать и оконную страницу свойств. В ней можно не только отображать свойства, но и производить их модификацию. В зависимоти от типа поля можно реализовывать различную логику установки свойств. В атрибут можно добавить параметр с категорией, в котором должно находитьс свойство, параметр ReadOnly –

доступно свойство только на просмотр или его можно модифицировать. Ниже приведен пример такого окна. Один раз написав компонент страницы свойств, его можно использовать повсюду и для любых типов. Необходимо лишь правильно развесить атрибуты на свойствах типов и указать нужную информацию атрибутов.