**Часть\_1 – Clean code**

**Обратить внимание: культура написания кода. Краткость и лаконичность, коротки методы, осознанное именование, повторное использование.**

а) Простота  
Здесь все уже придумано до нас — существует замечательный принцип [KISS](http://ru.wikipedia.org/wiki/KISS_%28%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF%29), а также афоризм Альберта Эйнштейна «Все должно быть изложено так просто, как только возможно, но не проще», которыми всем нам стоит руководствоваться.  
Простой код лучше сложного по всем параметрам — проще для понимания, легче поддается изменениям и тестированию, требует меньше комментариев и документации, содержит меньше ошибок (хотя бы чисто статистически, если мы принимаем среднее количество ошибок по отношению к количеству языковых и логических конструкций в коде за некоторую постоянную величину для каждого конкретного программиста).  
Также, перефразируя положение [ТРИЗ](http://ru.wikipedia.org/wiki/%D2%E5%EE%F0%E8%FF_%F0%E5%F8%E5%ED%E8%FF_%E8%E7%EE%E1%F0%E5%F2%E0%F2%E5%EB%FC%F1%EA%E8%F5_%E7%E0%E4%E0%F7) об идеальной системе можно сказать, что идеальный код — тот, которого нет, причем задача, которую он должен решать — успешно решается. Очевидно, что этот идеал (как и любой другой), недостижим на практике, однако способен направить ум разработчика в нужном направлении.  
На практике все эти красивые фразы означают одно — не используйте инструменты, *реальная* (а не гипотетическая) польза применения которых не превышает ущерб от усложнения и запутывания кода. Не стройте приложение вокруг целого фреймворка ради одной узкой задачи, которую он может помочь решить. Не используйте сложные паттерны там, где можно обойтись простым классом. Не используйте класс для пары функций, которые даже не работают со свойствами родного объекта в своем коде (в обратном случае — используйте). Не оборачивайте в функцию код из 3-х строк, который нигде более в приложении не используется повторно (и не будет использоваться в ближайшем будущем, а не через 150 лет).  
Разумеется, все это вовсе не означает, что нужно писать спагетти-код в императивном стиле на 5000 строк. Как только возникнет такое желание, нужно еще раз перечитать и осознать вторую часть цитаты приведенной выше, о том что код должен быть "*простым, насколько возможно, но не проще*".  
Не нужно бояться простых конструкций, если они позволяют решить поставленную задачу. Также не стоит прикрываться размышлениями на тему того, как ваш код может быть *теоретически* использован при развитии проекта или смене пожеланий заказчика через год-другой. Дело даже не в том, что ваш код будет «в любом случае выброшен и переписан с нуля», как заявляют многие сторонники простой и быстрой разработки. Возможно, что не будет. Но проблема в том, что с высокой долей вероятности вы просто не угадаете нужного направления. Вы спроектируете молоток так, чтобы им можно было (после небольшой доработки напильником) закручивать саморезы, а окажется, что нужно прикрутить автоматическую подачу гвоздей и лазерное наведение при замахе. И ваша архитектура, рассчитанная под развитие в определенную сторону, только проиграет по сравнению с «просто молотком». Поверьте, я знаю, о чем говорю, т.к. уже 5 лет поддерживаю проект в рунете, который вырос и стал довольно успешным вообще не на том, куда были вложены максимальные усилия и ради чего он создавался, а за счет небольшой ветки «побочного» функционала.

б) Концептуальность  
Этот критерий во многом пересекается с «простотой» кода, но все-таки я решил его вынести в отдельный раздел. Суть подхода — в использовании концепций, причем в идеале — общепринятых концепций, которые уже широко применяются в других решениях.  
Поясню на простой аналогии, что я имею ввиду:  
Представьте, что перед вами поставлена задача — заучить результаты умножения всех чисел от 2 до 9 друг на друга. Вы можете просто выписать в строку: 2х2=4, 2х3=6, ..., 3х7=21,… и т.д., после чего приступить к зазубриванию этого текста.  
Формально — этим можно пользоваться, и это даже можно выучить. Но есть значительно более практичный вариант — использовать концепцию «таблицы» — т.е. структуры данных, определяющей списки значений X и Y, с возможностью быстро определить точки их пересечения и значение этих точек по заранее заданной формуле (в данном случае Z = X \* Y). В результате мы получим известную всем с детства таблицу умножения, которая помимо банальной экономии количества символов в тексте обладает рядом других преимуществ:

* *Структурное восприятие* — помимо самих результатов умножения, мозг получает массу другой полезной информации — о том, что для каждого числа существует одинаковое количество результатов, включая умножение на самого себя, что шаг увеличения произведения при движении вниз по столбцу равняется числу в заголовке этого столбца (и аналогично для строк), и т.д.
* *Легкость масштабирования* — имея на руках таблицу и понимая ее *концепцию*, нет ничего проще чем масштабировать ее для, например, чисел от 1 до 20. При использовании простого текста время на его доработку увеличивается в разы (я бы даже сказал в квадрат раз).
* *Легкость модифицирования* — также очень легко превратить таблицу умножения в таблицу сложения, либо деления, либо в какую-то другую. Используемая базовая концепция позволяет легко варьировать свойства продукта в определенных пределах без изменения архитектуры.

Обратите внимание, что нам не приходится прикладывать каких-либо дополнительных усилий чтобы таблица «работала» таким образом, а также не нужно прилагать «инструкцию по пользованию таблицами» к нашему документу — в этой «бесплатности» и состоит основная сила использования концепций в программировании. Решения, в которых грамотно используются концепции, мы часто называем *изящными*.  
Примером блестящей концепции является [«перенаправление» в UNIX-системах](http://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BD%D0%B0%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%28UNIX%29), которое позволяет создавать различные программы, не тратя ни строчки кода на то, как они будут работать с чтением/записью в файлы или другие потоки, либо на взаимодействие с другими программами, например текстовым фильтром grep. Все что нужно — обеспечить стандартный интерфейс ввода/вывода текста.  
Таким образом даже несколько десятков наиболее популярных команд порождают тысячи вариантов их использования. Напоминает что-нибудь? Правильно — это решение задачи с помощью «кода, которого нет».  
Более простой пример из практики — однажды мне потребовалось ротировать на одном рекламном месте сайта объявления сторонней рекламной сети с двух различных аккаунтов, с условием примерно равного распределения прибыли между ними (с допустимым отклонением 1-2%). Громоздкое решение, которое можно было бы применить «в лоб», состояло в том, чтобы фиксировать в базе данных показы объявлений по каждому аккаунту с учетом стоимости этих показов (которые тоже могли отличаться), и на основании этих данных при каждом следующем открытии страницы принимать решение о том, чьи объявление показывать, чтобы обеспечить минимальный дисбаланс.  
Чуть позже выяснилось, что согласно [закону больших чисел](http://ru.wikipedia.org/wiki/%C7%E0%EA%EE%ED_%E1%EE%EB%FC%F8%E8%F5_%F7%E8%F1%E5%EB) банальная проверка вероятностей по принципу «орел-решка» при выборе аккаунта полностью решает задачу в рамках заданных условий. Более того, не составляет никаких проблем таким образом распределять показы между 3 и более аккаунтами. Само собой, это не придуманная человеком концепция, а базовый математический принцип, но с точки зрения кода важно лишь одно — это работает.

в) Уникальность функционала (Don't repeat yourself)  
Принцип [«Don't repeat yourself»](http://ru.wikipedia.org/wiki/Don%E2%80%99t_repeat_yourself) (сокр. DRY) говорит нам о том, что каждая функциональная единица системы, будь то логический блок кода, функция или целый класс, должна быть представлена в коде только один раз.  
Повторяющиеся блоки кода обычно выносятся в функции. Для функций используются подключаемые библиотеки и пространства имен. Когда же дело доходит до классов, в бой вступает целый зоопарк методик, направленных на снижение дублирования кода, начиная от банального наследования и заканчивая сложными системами из «фабрик по производству фабрик» вперемешку с «фасадными адаптерами», которые зачастую требуют больше кода и умственных усилий программиста, чем позволяют сэкономить.  
Стремление с самого начала спроектировать код так, чтобы избежать любых повторений — заведомо губительно, т.к. для 80% ситуаций вы не будете знать о дублировании кода, пока не напишите его.   
Реальный совет для практического использования в любом проекте — решайте проблему дублирования кода сразу после ее обнаружения (но не раньше!), и делайте это наиболее простым и очевидным способом, доступным вам в текущей ситуации.  
Есть несколько очень похожих блоков кода из 5-10 строк, отличающихся лишь парой условий и переменных, но вы пока не уверены, как лучше завернуть их в функцию? Используйте циклы.  
Не знаете, в какой класс/объект положить код нового метода? Просто создайте функцию и положите ее в подключаемую библиотеку до тех времен, когда у вас появится более полная информация о разрабатываемой системе.  
Используете наследование классов, но приходится полностью переопределять 50-строчный метод в наследнике ради изменения одной логической конструкции? В первую очередь подумайте о разделении большого метода на несколько более изолированных, либо об использовании инстансов других классов со всеми их методами в качестве значений свойств исходного объекта.  
Простые и проверенные решения позволят решить 95% всех проблем с дублированием кода, а для оставшихся 5% надо 10 раз подумать о целесообразности механического применения сложных конструкций «из учебников» по сравнению с осмысленным перепроектированием «плохого» участка кода (с упором на снижение сложности и повышение читабельности).  
Вы всегда сможете заменить простое решение на более высокоуровневое, и сделать это будет намного проще, чем применять сложный рефакторинг к сырому коду на более позднем этапе, когда дубликаты уже разбросаны по всей системе.

г) Удобочитаемость

* Используйте названия переменных, функций, классов и методов, наиболее точно описывающие суть того, что они именуют. Избегайте односимвольных переменных (кроме, разве что, счетчиков цикла типа i, n, k, и то не всегда), и названий типа var1, function2 и т.д. Не бросайтесь и в другую крайность — названия типа CreateTextDocumentWithAttachedFilesViaHTTP — тоже не очень повышают удобство чтения. Оптимальная длина названия: 1-3 слова.
* Старайтесь быть последовательными в именовании функций и методов, а также структуры их аргументов и возвращаемых значений. Не повторяйте ошибки разработчиков языка PHP (как пример — имена функций для работы с массивами — array\_filter, preg\_grep, array\_unique, in\_array, sort, current т.д.)
* Не используйте [«магические числа»](http://ru.wikipedia.org/wiki/%CC%E0%E3%E8%F7%E5%F1%EA%EE%E5_%F7%E8%F1%EB%EE_%28%EF%F0%EE%E3%F0%E0%EC%EC%E8%F0%EE%E2%E0%ED%E8%E5%29#.D0.9F.D0.BB.D0.BE.D1.85.D0.B0.D1.8F_.D0.BF.D1.80.D0.B0.D0.BA.D1.82.D0.B8.D0.BA.D0.B0_.D0.BF.D1.80.D0.BE.D0.B3.D1.80.D0.B0.D0.BC.D0.BC.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.B8.D1.8F). Исключение только одно — вы на 150% уверены, что никто и никогда не увидит этот код кроме вас, а сами вы никогда не приметесь за его поддержку спустя пару-тройку лет, мучительно вспоминая, что же такое «574».
* Не экономьте на пробелах и переносах строк при отделении языковых конструкций друг от друга. Помните, что в литературном английском языке (как и в русском), принято отделять текст после запятых пробелом, а абзацы разделять пустой строкой. Те же самые правила подойдут и программному коду.
* Используйте наиболее лаконичные из доступных стандартных конструкций языка. Например, тернарный оператор вместо if-else для блока кода, помещающегося в одну строчку. Не используйте операторы нестандартным, неизвестным основной массе разработчиков образом, если только ваш код не готовится для [конкурса](http://webreflection.blogspot.ru/2009/01/32-bytes-to-know-if-your-browser-is-ie.html).
* Комментируйте неочевидные участки кода. Не комментируйте очевидные. Для тренировки внутреннего «определителя очевидности» здорово помогает раз в месяц показывать код коллегам или хотя бы просто знакомым программистам.

Разумеется, само по себе хорошее оформление кода еще не гарантирует качество программы и легкость ее восприятия. Но именно такое внимание к мелочам и отличает настоящего профессионала от простого ремесленника, работающего в стиле «авось сгодится».

**Литература: The Robert C. Martin Clean Code Collection**

**Code Complete: A Practical Handbook of Software Construction, Second Edition**

–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

**Часть\_2 – Обзор .NET/C#, типы данных, классы vs структуры**

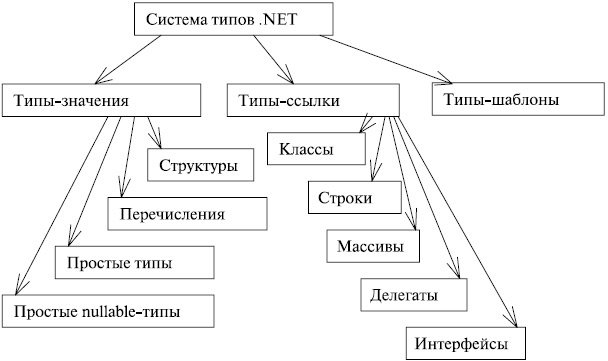
**Обратить внимание: Модель выполнения .NET платформы, управление памятью, базовая иерархия наследования типов, value types/reference types (их суть и разница).**

Платформа .NET содержит общеязыковую среду выполнения (Common Language Runtime — CLR). Общеязыковая среда выполнения CLR поддерживает управляемое выполнение, которое характеризуется рядом преимуществ. Совместно с общей системой типов (Common Type System — CTS) общеязыковая среда выполнения CLR поддерживает возможность взаимодействия языков платформы .NET. Кроме того, платформа .NET предоставляет большую полнофункциональную библиотеку классов .NET Framework.

Название среды — общеязыковая среда выполнения (Common Language Runtime, CLR) — говорит само за себя: это среда выполнения, которая подходит для разных языков программирования. Основные возможности CLR (управление памятью, загрузка сборок, безопасность, обработка исключений, синхронизация) доступны в любых языках программирования, использующих эту среду. Например, при об- работке ошибок среда выполнения опирается на исключения, а значит, во всех языках программирования, использующих эту среду выполнения, сообщения об ошибках передаются при помощи механизма исключений. Или, например, среда выполнения позволяет создавать программные потоки, а значит, во всех языках программирования, использующих эту среду, тоже могут создаваться потоки.Компиляция исходного кода в управляемые модули 29 Фактически во время выполнения программы в среде CLR неизвестно, на каком языке программирования разработчик написал исходный код. А это значит, что можно выбрать любой язык программирования, который позволяет проще всего решить конкретную задачу. Разрабатывать программное обеспечение можно на любом языке программирования, если только используемый компилятор этого языка поддерживает CLR. Так в чем же тогда преимущество одного языка программирования перед дру- гим? Я рассматриваю компиляторы как средства контроля синтаксиса и анализа «правильности кода». Компиляторы проверяют исходный код, убеждаются, что все написанное имеет некий смысл, и затем генерируют код, описывающий решение данной задачи. Разные языки программирования позволяют разрабатывать про- граммное обеспечение, используя различный синтаксис. Не стоит недооценивать значение выбора синтаксиса языка программирования. Например, для математи- ческих или финансовых приложений выражение мысли программиста на языке APL может сохранить много дней работы по сравнению с применением в данной ситуации языка Perl. Компания Microsoft разработала компиляторы для следующих языков про- граммирования, используемых на этой платформе: C++/CLI, C# (произносится «си шарп»), Visual Basic, F# (произносится «эф шарп»), Iron Python, Iron Ruby и ассемблер Intermediate Language (IL). Кроме Microsoft, еще несколько компаний и университетов создали компиляторы, предназначенные для среды выполнения CLR. Мне известны компиляторы для Ada, APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML, Mondrian, Oberon, Pascal, Perl, Php, Prolog, RPG, Scheme, Smalltalk и Tcl/Tk. Рисунок 1.1 иллюстрирует процесс компиляции файлов с исходным кодом. Как видно из рисунка, исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный код программы. Вне зависимости от типа используемого компилятора результатом компиляции будет являться управляе- мый модуль (managed module) — стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения CLR. Кстати, управляемые сборки всегда используют преимущества функции безопасности «предотвращения выполнения данных» (DEP, Data Execution Prevention) и технологию ASLR (Address Space Layout Optimization), применение этих технологий повышает информационную безопасность всей системы. Компиляторы машинного кода производят код, ориентированный на конкретную процессорную архитектуру, например x86, х64 или ARM. В отличие от этого, все CLR-совместимые компиляторы генерируют IL-код. (Подробнее об IL-коде рас- сказано далее в этой главе.) IL-код иногда называют управляемым (managed code), потому что CLR управляет его выполнением.

Перед выполнением на конкретной машине, код на промежуточном языке Microsoft— MS1L (Microsoft Intermediate Language) транслируется оперативным компилятором, или ЛТ-компилятором (JIT — "just-in-time" или "как раз вовремя") в собственный (внутренний) код. Во время работы программы некоторые участки кода выполняться никогда не будут. Следовательно, более эффективной может оказаться трансляция кода из промежуточного языка MSIL в собственный (внутренний) код, осуществляемая по мере необходимости в процессе выполнения приложения. Собственный (внутренний) код при этом сохраняется с целью повторного его использования.   
После загрузки типа, к каждому его методу загрузчик присоединяет заглушку. При первом вызове заглушка передает управление ЛТ-компилятору, который генерирует собственный (внутренний) код и сохраняет адрес оттранслированного собственного (внутреннего) кода в заглушке. При последующих вызовах метода управление передается непосредственно собственному (внутреннему) коду.

Наиболее общим концептом иерархии является пространство имен System, характеризующее конфигурацию среды Microsoft .NET Framework и содержащее, в частности, параметры среды времени выполнения приложений, удаленной обработки данных, процессов, безопасности, ввода-вывода, системной конфигурации и др. Среди подпространств пространство имен System можно выделить такие пространства имен, как System.Web, System.Windows.Forms, System.Data, System.Drawing и System.Xml, которые описывают такие характеристики среды Microsoft .NET Framework, как конфигурации веб-сервисов, формы ввода-вывода данных, форматы представления данных, графических подсистем и др.



**Рисунок 1.1 – Система типов .Net**

**Ссылочные и значимые типы**

CLR поддерживает две разновидности типов: ссылочные (reference types) и значимые  
(value types). Помимо этих есть еще и примитивые типы(но я их опишу в следующих статьях).  
Большинство типов в FCL(Framework Class Library)- ссылочные, но программисты чаще всего используют значимые. В чем между ними разница спросите вы?   
  
Переменные ссылочного типа содержат в себе ссылки на фактические данные и при этом ссылка указывает на определенную область в памяти, которая была выделена при создании такой переменной. Память же выделяется при этом в управляемой куче — это область памяти, в которой размещаются управляемые объекты и работает сборщик мусора. Так как С# это полностью управляемый язык, то одной из его главных особенностей то, что в процессе работы программы он все время контролирует расход ресурсов и все объекты, которые он создает(если они уже не нужны и не используются) сборщик мусора уничтожает. То есть контроль над расходом ресурсов автоматизирован, при этом в неуправляемой куче за освобождением памяти от ненужных объектов следит сам программист, что усложняет написание задачи.  
  
У переменной значимого типа поля экземпляра размещаются в самой переменной. Поле представляет собой изменяемое или неизменяемое значение. Поле может быть статическим и является частью ТИПА, или же быть экземплярным(нестатическим) и являться частью самого ОБЪЕКТА. Чтобы наиболее полностью понять разницу между объектом и типом, нужно хорошо разбираться в концепциях ООП. Также в отличие от ссылочного типа, память для значимого типа выделяется при этом в стеке потока. Стек потока — это область памяти, которая используется для передачи параметров в методы и хранения определенных в пределах методов локальных переменных. Те, кто может быть знаком с многопоточным выполнением программ, то такой поток еще по-другому называют «Стек пользовательского режима». Его размер всегда равен 1Мб. Чтобы вы правильно поняли, то метод static void Main() это и есть главный поток программы, в котором мы создаем значимые переменные, а также методы с параметрами и без параметров. А ведь эти переменные и параметры надо где-то хранить, вот для этого и используется стек потока.  
  
Так в чем же состоит существенная разница между этими двумя типами. Все кроется в том, как выделяется память для этих двух типов. Как выше было сказано, в одном случае память выделяется в куче, в другом случае в стеке. Так вот, если все типы были бы ссылочными, то скорость работы программы очень резко бы упала. Ведь представьте, что при каждом обращении к типу Int32, то сколько бы раз нужно было бы выделять память в куче при создании переменной. Поэтому для ускорения работы, CLR предлагает нам «облегченные» — значимые типы. В такой переменной нет указателя на экземпляр, поля экземпляра размещаются в самой переменной. При этом переменные не обрабатываются сборщиком мусора. А также момент размещения в стеке достаточно быстр. Предположим мы создаем переменную Int32 — это 32-разрядное значение и среда знает уже изначально, сколько нужно выделить памяти для этой переменной, поэтому выделение происходит моментально.   
  
**К ссылочным типам относятся:**

* class
* interface
* delegate
* object
* string

**Значимые типы:**  
Типы значений состоят из двух основных категорий:  
  
1. Структуры (struct)  
2.Перечисления (enum)   
  
Структуры делятся на следующие категории:

* Числовые типы:  
  1.Целочисленные типы (sbyte, byte,char,short,ushort,int,uint,long,ulong)  
  2.Типы с плавающей точкой (float,double)  
  3.decimal (обозначает 128-разрядный тип данных. По сравнению с типом данных с плавающей запятой, тип decimal имеет более точный и узкий диапазон, благодаря чему он походит для финансовых расчетов.)
* bool (для хранения логических значений, true и false)
* Пользовательские структуры

Как известно корнем иерархии всех типов, классов и т.д. является класс Object. По умолчанию он всегда является базовым классом. В документации .NET Framework сказано, что структуры являются прямыми потомками типа System.ValueType, который является производным в свою очередь от System.Object, при этом метод Equels возвращает true, если значения полей у обоих объектов совпадают. Так же алгоритм метода GetHashCode реализован с учетом значений полей. Но при создании своего значимого типа рекомендуется переопределить оба этих метода. При этом значимые типы являются изолированными в целях безопасности. Поэтому при создании своего собственного значимого типа нельзя в качестве базового указывать другие типы, например Boolean, Char, Int32 и так далее.  
  
**Выделение ресурсов** из управляемой кучи В CLR память для всех ресурсов выделяется из так называемой управляемой кучи (managed heap). При инициализации процесса CLR резервирует область адресного пространства под управляемую кучу, а также указатель, который я называю NextObjPtr. Он определяет, где в куче будет выделена память для следующего объекта, и изначально указывает на базовый адрес этой зарезервированной области адресного пространства. По мере заполнения области объектами CLR выделяет новые области, вплоть до заполнения всего адресного пространства. Таким образом, память приложения ограничивается виртуальным адресным пространством процесса. Для 32-разрядных процессов можно выделить до 1,5 гигабайта памяти, а для 64-разрядных процессов — около 8 терабайт памяти. При выполнении оператора C# new среда CLR:

1) подсчитывает количество байтов, необходимых для размещения полей типа (и всех полей, унаследованных от базового типа);

2) прибавляет к полученному значению количество байтов, необходимых для размещения системных полей объекта.

У каждого объекта есть пара таких полей: указатель на объект-тип и индекс блока синхронизации. В 32-разрядных приложениях для каждого из этих полей требуется 32 бита, что увеличивает размер каждого объекта на 8 байт, а в 64-разрядных приложениях каждое поле занимает 64 бита, добавляя к каждому объекту 16 байт;

3) проверяет, хватает ли в зарезервированной области байтов на выделение памяти для объекта (при необходимости передает память). Если в управляемой куче достаточно места для объекта, ему выделяется память, начиная с адреса, на который ссылается указатель NextObjPtr, а занимаемые им байты обнуляются. Затем вызывается конструктор типа (передающий NextObjPtr в качестве параметра this), и оператор new возвращает ссылку на объект. Перед возвратом этого адреса NextObjPtr переходит на первый адрес после объекта, указывая на адрес, по которому в куче будет помещен следующий объект.

Сразу после инициализации в управляемой куче нет объектов. Говорят, что создаваемые в куче объекты составляют поколение 0. Проще говоря, к нулевому поколению относятся только что созданные объекты, которых не касался уборщик мусора.

На высоком уровне, значения **констант**, заменяються своими значениями во время компиляции, в то время как значение "static readonly" поля определяеться в момент выполнения кода.

Это означает, что любая библиотека / сборка, у которых есть ссылки на "const" необходимо перекомпилировать если изменять константу. Изменение же статических переменных не потребует в дальнейшем перекомпиляции.Безусловный выигрыш статик полей.

"Static readonly" поля могут содержать ссылочные типы, в то время как константы только поддерживают чисельные типы, строки .NET string и null.

Вы не можете использовать конструктор по умолчанию (без параметров) для **структуры**, потому что компилятор *всегда* генерирует его сам. Что же касается класса, то компилятор создает конструктор по умолчанию, только в том случае, если Вы его не создали. Сгенерированный конструктор для структуры*всегда* устанавливает поля в *0*, *false* или *null* – как и для классов. Поэтому Вы можете быть уверенными в том, что созданная структура всегда будет вести себя “адекватно” в соответствии со значениями по умолчанию в используемых типах. Если Вы не хотите использовать значения по умолчанию, Вы можете инициализировать поля своими значениями в конструкторе с параметрами для инициализации. Однако, если в этом конструкторе не будет инициализировано какое-нибудь значение, компилятор не будет его инициализировать за Вас и покажет ошибку.

**Первое правило Структуры**: *Всегда* все переменные должны быть инициализированы!

В классах Вы можете инициализировать  значение полей непосредственно в месте их объявления. В структурах такого сделать не получится.

**Второе правило Структуры**: Нельзя инициализировать переменные в месте их объявления!

**Абстрактный(Abstract) класс** - класс, который имеет хотя б 1 абстрактный (не определенный) метод; обозначается как abstract.  
**Интерфейс** - такой же абстрактный класс,только в нем не может быть свойств и не определены тела у методов.  
  
Так же стоит заметить, что абстрактный класс наследуется(etxends), а интерфейс реализуется (implements). Вот и возникает разница между ними, что наследовать мы можем только 1 класс, а реализовать сколько угодно.

**Передача параметров в метод по ссылке. Операторы ref и out**  
  
В C# значения переменных по-умолчанию передаются по значению (в метод передается локальная копия параметра, который используется при вызове). Это означает, что мы не можем внутри метода изменить параметр из вне:

public static void ChangeValue(object a)  
{  
   a = 2;  
}  
static void Main(string[] args)  
{  
   int a = 1;  
   ChangeValue(a);  
   Console.WriteLine(a); // 1  
   Console.ReadLine();  
}

Чтобы передавать параметры по ссылке, и иметь возможность влиять на внешнюю переменную, используются ключевые слова ref и out.  
  
**Ключевое слово ref**  
  
Чтобы использовать *ref*, это ключевое слово стоит указать перед типом параметра в методе, и перед параметром при вызове метода:

public static void ChangeValue(ref int a)  
{  
   a = 2;  
}  
static void Main(string[] args)  
{  
   int a = 1;  
   ChangeValue(ref a);  
   Console.WriteLine(a); // 2  
   Console.ReadLine();  
}

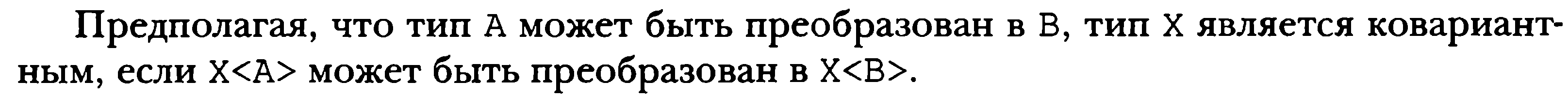
В этом примере мы изменили значение внешней переменной внутри метода.  
  
Особенностью ref является то, что переменная, которую мы передаем в метод, обязательно должна быть проинициализирована значением, иначе компилятор выдаст ошибку «Use of unassigned local variable 'a'». Это является главным отличием *ref* от *out*.   
  
**Ключевое слово out**  
  
*Out* используется точно таким же образом как и *ref*, за исключением того, что параметр не обязан быть проинициализирован перед передачей, но при этом в методе переданному параметру обязательно должно быть присвоено новое значение:

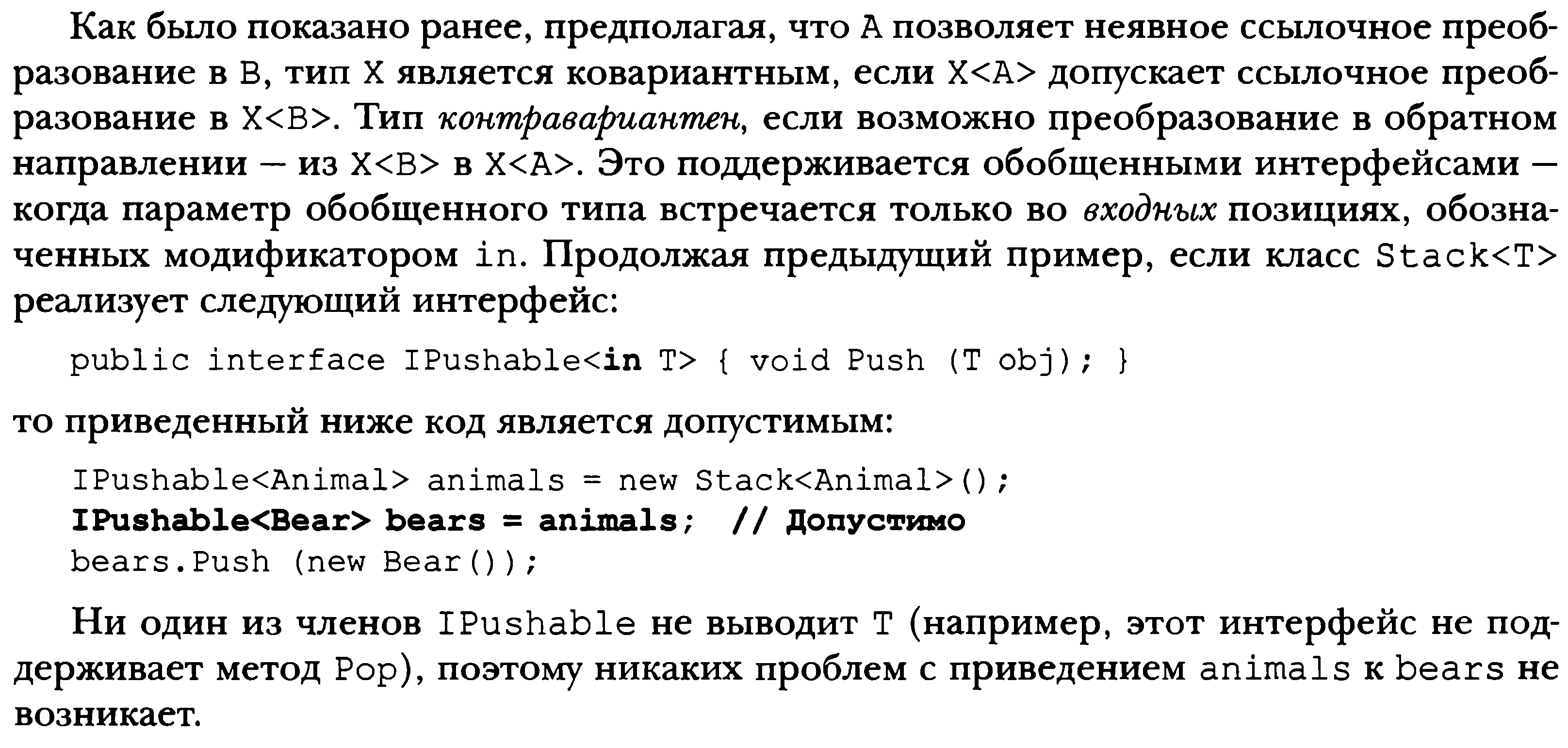
public static void ChangeValue(out int a)  
{  
   a = 2;  
}  
static void Main(string[] args)  
{  
   int a;  
   ChangeValue(out a);  
   Console.WriteLine(a); // 2  
   Console.ReadLine();  
}

Если не присвоить новое значение параметру *out*, мы получим ошибку «The out parameter 'a' must be assigned to before control leaves the current method»  
  
**Производительность**  
  
Учитывая тот факт, что по умолчанию в метод передаются параметры по значению и создаются их копии в стеке, при использовании сложных типов данных (пользовательские структуры), или если метод вызывается много раз, это плохо скажется на производительности. В таком случае также стоит использовать ключевые слова *ref* и *out*.  
Если говорить в целом о ссылочных типах и типах значений, то производительность приложения упадет, если использовать только ссылочные типы. На создание переменной ссылочного типа в куче выделяется память под данные, а в стеке под ссылку на эти данные. Для типов значений память выделяется только в стеке. Время на размещение данных в стеке меньше, чем в куче, это также идет в плюс типам значений в плане производительности.

**Вариантность**(нет связи) **–** это сохранение совместимости присваивания исходных типов у производных типов.  
**Ковариантность** (List<>) – это сохранение совместимости присваивания исходных типов у производных в прямом порядке.  
**Контравариантность**(делегаты Action)– это сохранение совместимости присваивания исходных типов у производных в обратном порядке.

**Ковариантность**позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата. **Контравариантность**позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.





**Литература: CLR via C# (4th Edition, Jeffrey Richter), С# 5.0 (Joseph & Ben Albahary)**

–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

**Часть\_3 – Циклы, коллекции, Generic типы, Generic коллекции**

**Основные моменты:**

**- Циклы: Виды циклов. Ключевые слова break, continue, yield;**

**- Коллекции: В каких ситуациях следует использовать основные типы коллекций: ArrayList, SortedList, Queue, Stack, Hashtable, BitArray, StringCollection, StringDictionary, ListDictionary, HybridDictionary, NameValueCollection;**

**- Интерфейс IEnumerable;**

**- Generic типы: принцип работы, преимущества, ограничения ("constraints");**

**- Generic коллекции: List<>, Queue<>, Stack<>, Dictionary<>, SortedList<>, SortedDictionary<>, LinkedList<>.**

**Циклы** также являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

* for
* foreach
* while
* do...while

**Цикл for**

Цикл for имеет следующее формальное определение:

|  |  |
| --- | --- |
| 1  2  3  4 | for ([инициализация счетчика]; [условие]; [изменение счетчика])  {      // действия  } |

Рассмотрим стандартный цикл for:

|  |  |
| --- | --- |
| 1  2  3  4 | for (int i = 0; i < 9; i++)  {      Console.WriteLine("Квадрат числа {0} равен {1}", i, i \* i);  } |

Первая часть объявления цикла - int i = 0 - создает и инициализирует счетчик i. Счетчик необязательно должен представлять типint. Это может быть и другой числовой тип, например, float. И перед выполнением цикла его значение будет равно 0. В данном случае это то же самое, что и объявление переменной.

Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9.

И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.

В итоге блок цикла сработает 9 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i = 0;  for (; ;)  {      Console.WriteLine("Квадрат числа {0} равен {1}", ++i, i \* i);      System.Threading.Thread.Sleep(500);  } |

Формально определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). У нас нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно - бесконечный цикл.

Мы также можем опустить ряд блоков:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int i = 0;  for (; i<9;)  {      Console.WriteLine("Квадрат числа {0} равен {1}", ++i, i \* i);  } |

Этот пример по сути эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

**Цикл foreach**

Цикл foreach предназначен для перебора элементов в контейнерах. Формальное объявление цикла foreach:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach (тип\_данных название\_переменной in контейнер)  {      // действия  } |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] array = new int[] { 1, 2, 3, 4, 5 };  foreach (int i in array)  {      Console.WriteLine(i);  } |

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

Подобные действия мы можем сделать и с помощью цикл for:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] array = new int[] { 1, 2, 3, 4, 5 };  for (int i = 0; i < array.Length; i++)  {      Console.WriteLine(array[i]);  } |

В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[] array = new int[] { 1, 2, 3, 4, 5 };  for (int i = 0; i < array.Length; i++)  {      array[i] = array[i] \* 2;      Console.WriteLine(array[i]);  } |

**Цикл do**

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int i = 6;  do  {      Console.WriteLine(i);      i--;  }  while (i > 0); |

Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int i = -1;  do  {      Console.WriteLine(i);      i--;  }  while (i > 0); |

Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.

**Цикл while**

В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int i = 6;  while (i > 0)  {      Console.WriteLine(i);      i--;  } |

**Операторы continue и break**

Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int[] array = new int[] { 1, 2, 3, 4, 12, 9 };  for (int i = 0; i < array.Length; i++)  {      if (array[i] > 10)          break;      Console.WriteLine(array[i]);  } |

Поскольку в цикле идет проверка, больше ли элемент массива 10. То мы никогда не увидим на консоли последние два элемента, так как, увидев, что элемент массива больше 10, сработает оператор break, и цикл завершится.

Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто переходил к следующему элементу. Для этого мы можем воспользоваться оператором continue:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int[] array = new int[] { 1, 2, 3, 4, 12, 9 };  for (int i = 0; i < array.Length; i++)  {      if (array[i] > 10)          continue;      Console.WriteLine(array[i]);  } |

В этом случае цикл, когда дойдет до числа 12, которое не удовлетворяет условию проверки, просто пропустит это число и перейдет к следующему элементу массива.

**Необобщенные или простые коллекции** определены в пространстве имен System.Collections. Их особенность состоит в том, что их функциональность, функциональные возможности описываются в интерфейсах, которые также находятся в этом пространстве имен.

Рассмотрим основные интерфейсы:

* **IEnumerable**: определяет метод GetEnumerator. Данный метод возвращает перечислитель - то есть некоторый объект, реализующий интерфейс IEnumerator.
* **IEnumerator**: реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach

Эти интерфейсы реализуются следующими классами коллекций в пространстве имен **System.Collections**:

* **ArrayList**: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable
* **BitArray:** класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable
* **Hashtable:** класс коллекции, представляющей хэш-таблицу и хранящий набор пар "ключ-значение"
* **Queue:** класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable
* **SortedList:** класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable
* **Stack:** класс стека.

Как мы увидели, основной для большинства коллекций является реализация интерфейсов **IEnumerable и IEnumerator**. Благодаря такой реализации мы можем перебирать объекты в цикле foreach.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IEnumerable  {      IEnumerator GetEnumerator();  } |

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface IEnumerator  {      bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов      object Current {get;}  // текущий элемент в контейнере      void Reset(); // перемещение в начало контейнера  } |

Мы можем не полагаться на реализацию перечислителя в массиве, а создать итератор с помощью ключевого слова **yield**. **Итератор** представляет метод, в котором используется ключевое слово yield для перебора по коллекции или массиву. Например, перепишем вышеопределенный метод GetEnumerator в классе Library с использованием yield:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | IEnumerator IEnumerable.GetEnumerator()  {      for (int i = 0; i < books.Length; i++)      {          yield return books[i];      }  } |

Метод GetEnumerator() теперь и будет являться итератором. И когда мы будем осуществлять перебор в объекте Library в цикле foreach, то будет идти к обращение к вызову yield return books[i];. При обращении к оператору yield return будет сохраняться текущее местоположение. И когда метод foreach перейдет к следующей итерации для получения нового объекта, итератор начнет выполнения с этого местоположения.

Ну и в основной программе в цикле foreach выполняется собственно перебор, благодаря реализации итератора:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach (Book b in library)  {      Console.WriteLine(b.Name);  } |

Хотя при реализации итератора в методе GetEnumerator() применялся перебор массива в цикле for, но это необязательно делать. Мы можем просто определить несколько вызовов оператора yield return:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | IEnumerator IEnumerable.GetEnumerator()  {      yield return books[0];      yield return books[1];      yield return books[2];  } |

В этом случае при каждом вызове оператора yield return итератор также будет запоминать текущее местоположение и при последующих вызовах начинать с него.

**Именованный итератор**

Выше для создания итератора мы использовали метод GetEnumerator. Но оператор yield можно использовать внутри любого метода, только такой метод должен возвращать объект интерфейса IEnumerable. Подобные методы еще называют **именованными итераторами**.

Создадим такой именованный итератор в классе Library и используем его:

class Book

{

    public Book(string name)

    {

        this.Name=name;

    }

    public string Name { get; set; }

}

class Library

{

    private Book[] books;

    public Library()

    {

        books = new Book[] { new Book("Отцы и дети"), new Book("Война и мир"),

            new Book("Евгений Онегин") };

    }

    public int Length

    {

        get { return books.Length; }

    }

    public Book this[int index]

    {

        get

        {

            return books[index];

        }

        set

        {

            books[index] = value;

        }

    }

    public IEnumerable GetBooks(int max)

    {

        for (int i = 0; i < max; i++)

        {

            if (i == books.Length)

            {

                yield break;

            }

            else

            {

                yield return books[i];

            }

        }

    }

}

Определенный здесь итератор - метод IEnumerable GetBooks(int max) в качестве параметра принимает количество выводимых объектов. В процессе работы программы может сложиться, что его значение будет больше, чем длина массива books. И чтобы не произошло ошибки, используется оператор **yield break**. Этот оператор прерывает выполнение итератора.

Применение итератора:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Library library = new Library();  foreach (Book b in library.GetBooks(5))  {      Console.WriteLine(b.Name);  } |

Вызов library.GetBooks(5) будет возвращать набор из не более чем 5 объектов Book. Но так как у нас всего три таких объекта, то в методе GetBooks после трех операций сработает оператор yield break.

**Литература: Andrew Troelsen Pro C#, Chapter 3 and 9**

–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––