1 Бренд .Net. Visual Studio .Net – открытая среда разработки. Каркас Framework .Net. Библиотека классов FCL - статический компонент каркаса.

В программных продуктах .Net за этим именем стоит вполне конкретное содержание, которое предполагает, в частности, наличие открытых стандартов коммуникации, переход от создания монолитных приложений к созданию компонентов, допускающих повторное использование в разных средах и приложениях. Возможность повторного использования уже созданных компонентов и легкость расширения их функциональности - все это непременные атрибуты новых технологий. Важную роль в этих технологиях играет язык XML, ставший стандартом обмена сообщениями в сети.

Visual Studio .Net - открытая среда разработки

Среда разработки Visual Studio .Net . Выделю две важнейшие, идеи:

- открытость для языков программирования;
- принципиально новый подход к построению каркаса среды - Framework .Net.

Среда разработки теперь является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой Microsoft - Visual C++ .Net (с управляемыми расширениями), Visual C# .Net, J# .Net, Visual Basic .Net, - в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами-производителями.

Открытость среды не означает полной свободы. Все разработчики компиляторов при включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение, которое можно считать и главным достоинством, состоит в том, что все языки, включаемые в среду разработки Visual Studio .Net, должны использовать единый каркас - Framework .Net. Благодаря этому достигаются многие желательные свойства: легкость использования компонентов, разработанных на различных языках; возможность разработки нескольких частей одного приложения на разных языках; возможность бесшовной отладки такого приложения; возможность написать класс на одном языке, а его потомков - на других языках. Единый каркас приводит к сближению языков программирования, позволяя вместе с тем сохранять их индивидуальность и имеющиеся у них достоинства. Преодоление языкового барьера - одна из важнейших задач современного мира.

Framework .Net - единый каркас среды разработки

В каркасе Framework .Net можно выделить два основных компонента:

- статический - FCL (Framework Class Library) - библиотеку классов каркаса;

•динамический - CLR (Common Language Runtime) - общеязыковую исполнительную среду.

Библиотека классов FCL - статический компонент каркаса

Понятие каркаса приложений - Framework Applications - появилось достаточно давно. Несмотря на то, что каркас был представлен только статическим компонентом, уже тогда была очевидна его роль в построении приложений. Уже в то время важнейшее значение в библиотеке классов MFC имели классы, задающие архитектуру строящихся приложений. Когда разработчик выбирал один из возможных типов приложения, например, архитектуру Document-View, то в его приложение автоматически встраивались класс Document, задающий структуру документа, и класс View, задающий его визуальное представление. Класс Form и классы, задающие элементы управления, обеспечивали единый интерфейс приложений. Выбирая тип приложения, разработчик изначально получал нужную ему функциональность, поддерживаемую классами каркаса. Библиотека классов поддерживала и более традиционные для программистов классы, задающие расширенную систему типов данных, в частности, динамические типы данных - списки, деревья, коллекции, шаблоны.

Единство каркаса

Каркас стал единым для всех языков среды. Поэтому, на каком бы языке программирования ни велась разработка, она использует классы одной и той же библиотеки. Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными хранилищами данных и прочая универсальность.

Встроенные примитивные типы

Важной частью библиотеки FCL стали классы, задающие примитивные типы - те типы, которые считаются встроенными в язык программирования. Типы каркаса покрывают все множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке Visual Basic - Integer, а в языке C# - int, проецируется на один и тот же тип каркаса System.Int32. В каждом языке программирования, наряду с "родными" для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе. Поэтому, по сути, все языки среды разработки могут пользоваться единой системой встроенных типов, что, конечно, способствует облегчению взаимодействия компонентов, написанных на разных языках.

Структурные типы

Частью библиотеки стали не только простые встроенные типы, но и структурные типы, задающие организацию данных - строки, массивы, перечисления, структуры (записи). Это также способствует унификации и реальному сближению языков программирования.

Архитектура приложений

Существенно расширился набор возможных архитектурных типов построения приложений. Помимо традиционных Windows- и консольных приложений, появилась возможность построения Web-приложений. Большое внимание уделяется возможности создания повторно используемых компонентов - разрешается строить библиотеки классов, библиотеки элементов управления и библиотеки Web-элементов управления. Популярным архитектурным типом являются Web-службы, ставшие сегодня благодаря открытому стандарту одним из основных видов повторно используемых компонентов.

Модульность

Число классов библиотеки FCL велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединяются в группы, называемые пространством имен (Namespace). Для динамического компонента CLR физической единицей, объединяющей классы и другие ресурсы, является сборка (assembly).

Основным пространством имен библиотеки FCL является пространство System, содержащее как классы, так и другие вложенные пространства имен.

2 Общеязыковая исполнительная среда CLR - динамический компонент каркаса. Управляемый код. Общеязыковые спецификации CLR и совместимые модули.

Общеязыковая исполнительная среда CLR - динамический компонент каркаса

Наиболее революционным изобретением Framework .Net явилось создание исполнительной среды CLR. С ее появлением процесс написания и выполнения приложений становится принципиально другим. Но обо всем по порядку.

Управляемый модуль и управляемый код

Компиляторы языков программирования, включенные в Visual Studio .Net, создают модули на промежуточном языке MSIL (Microsoft Intermediate Language), называемом далее просто - IL. Фактически компиляторы создают так называемый управляемый модуль - переносимый исполняемый файл (Portable Executable или PE-файл). Этот файл содержит код на IL и метаданные - всю необходимую информацию как для CLR, так и конечных пользователей, работающих с приложением.

Виртуальная машина

Отделение каркаса от студии явилось естественным шагом. Каркас Framework .Net перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание PE-модулей на IL отделены от выполнения, и эти процессы могут быть реализованы на разных платформах. В состав CLR входят трансляторы JIT (Just In Time Compiler), которые и выполняют трансляцию IL в командный код той машины, где установлена и функционирует исполнительная среда CLR.

Дизассемблер и ассемблер

Если у вас есть готовый PE-файл, то иногда полезно анализировать его IL-код и связанные с ним метаданные. В состав Framework SDK входит дизассемблер - ildasm, выполняющий дизассемблирование PE-файла и показывающий метаданные, а также IL-код с комментариями в наглядной форме.

Метаданные

Переносимый исполняемый PE-файл является самодокументируемым файлом и, содержит и код, и метаданные, описывающие код. Файл начинается с манифеста и включает в себя описание всех классов, хранимых в PE-файле, их свойств, методов, всех аргументов этих методов - всю необходимую CLR информацию. Поэтому помимо PE-файла не требуется никаких дополнительных файлов и записей в реестр - вся нужная информация извлекается из самого файла. Среди классов библиотеки FCL имеется класс Reflection, методы которого позволяют извлекать необходимую информацию.

Сборщик мусора - Garbage Collector - и управление памятью

Еще одной важной особенностью построения CLR является то, что исполнительная среда берет на себя часть функций, традиционно входящих в ведение разработчиков трансляторов, и облегчает тем самым их работу. Один из таких наиболее значимых компонентов CLR - сборщик мусора (Garbage Collector). Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения.

Исключительные ситуации

Что происходит, когда при вызове некоторой функции (процедуры) обнаруживается, что она не может нормальным образом выполнить свою работу? Возможны разные варианты обработки такой ситуации. Функция может возвращать код ошибки или специальное значение типа HResult, может выбрасывать исключение, тип которого характеризует возникшую ошибку.

3 Создание языка. Его особенности. Решения, проекты, пространства имен.

Язык C# является наиболее известной новинкой в области создания языков программирования.

Создателем языка является сотрудник Microsoft Андреас Хейлсберг. Как отмечал сам Андреас Хейлсберг, C# создавался как язык компонентного программирования, и в этом одно из главных достоинств языка, направленное на возможность повторного использования созданных компонентов.

Виды проектов

Как уже отмечалось, Visual Studio .Net для языков C#, Visual Basic и J# предлагает 12 возможных видов проектов. Среди них есть пустой проект, в котором изначально не содержится никакой функциональности; есть также проект, ориентированный на создание Web-служб, обычные Windows-приложения, консольные приложения и т.д.

С точки зрения программиста, компилятор создает решение, с точки зрения CLR - сборку, содержащую PE-файл. Программист работает с решением, CLR - со сборкой.

Решение содержит один или несколько проектов, ресурсы, необходимые этим проектам, возможно, дополнительные файлы, не входящие в проекты. Один из проектов решения должен быть выделен и назначен стартовым проектом. Выполнение решения начинается со стартового проекта. Проекты одного решения могут быть зависимыми или независимыми. *Например, все проекты одной лекции данной книги могут быть для удобства собраны в одном решении и иметь общие свойства. Изменяя стартовый проект, получаем возможность перехода к нужному примеру. Заметьте, стартовый проект должен иметь точку входа - класс, содержащий статическую процедуру с именем Main, которой автоматически передается управление в момент запуска решения на выполнение. В уже имеющееся решение можно добавлять, как новые, так и существующие проекты. Один и тот же проект может входить в несколько решений.*

Проект состоит из классов, собранных в одном или нескольких пространствах имен. Пространства имен позволяют структурировать проекты, содержащие большое число классов, объединяя в одну группу близкие классы. Если над проектом работает несколько исполнителей, то, как правило, каждый из них создает свое пространство имен. Помимо структуризации, это дает возможность присваивать классам имена, не задумываясь об их уникальности. В разных пространствах имен могут существовать одноименные классы. Проект - это основная единица, с которой работает программист. Он выбирает тип проекта, а Visual Studio создает скелет проекта в соответствии с выбранным типом.

Интегрированная среда разработки IDE (Integrated Development Environment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей. В окне Solution Explorer представлена структура построенного решения. В окне Properties можно увидеть свойства выбранного элемента решения. Интегрированная среда разработки IDE включает в себя конструктор форм: предназначенный для создания вин-приложений или веб-приложений, причем компоненты для создания вин-форм отличны от компонентов для создания веб-

форм. Также и Интегрированная среда разработки IDE присутствует обозреватель классов: позволяющий упростить с ним работу к контексте больших проектов.

Верхнее выпадающее меню содержи функционал создания проектов, решений, файлов, отладки(debug) проектов, компилирование проектов, справочной информацией.

Пространство имен

В общем виде формат обращения к статическим членам класса:

Название_пространства_имен.имя_класса.имя_члена

Эта конструкция в языке C# называется уточненным, квалифицированным или составным именем. Первым элементом квалифицированного имени является наименование пространства имен.

"Пространство имен "пространство имен" – механизм, посредством которого поддерживается независимость используемых в каждой программе имен и исключается возможность их случайного взаимного влияния"

"Пространство имен определяет декларативную область, которая позволяет отдельно хранить множества имен. По существу, имена, объявленные в одном пространстве имен, не будут конфликтовать с такими же именами, объявленными в другом."

Каждая программа на C# может использовать либо свое собственное уникальное пространство имен, либо размещает свои имена в пространстве, предоставляемом программе по умолчанию. Но мы вынуждены использовать пространства имен тех библиотек, средства которых применяются в наших программах.

Понятие пространства имен появилось в программировании в связи с необходимостью различать одноименные понятия из разных библиотек, используемых в одной программе. Пространство имен System объединяет те классы из .NET Framework, которые наиболее часто используются в консольных программах на C#.

Если в программе необходимо многократно обращаться к классам из одного и того же пространства имен, то можно упростить составные имена, используя в начале программы (до определения класса) специальный оператор:

using имя_пространства_имен;

После такого оператора для обращения к статическому члену класса из данного пространства имен можно использовать сокращенное квалифицированное имя

имя_класса.имя_члена

В нашей программе используются: пространство имен System, из этого пространства - класс Console и два статических метода этого класса WriteLine( ) и ReadLine( ).

Поместив в программу оператор

using System;

можно обращаться к названным методам с помощью сокращенных составных имен Console.WriteLine( ) и Console.ReadLine( ). Именно так мы будем поступать в следующих примерах программ.

4 Консольные и Windows-приложения C#, построенные по умолчанию.

Консольный проект

У себя на компьютере я открыл установленную лицензионную версию Visual Studio .Net 2003, выбрал из предложенного меню - создание нового проекта на C#, установил вид проекта - консольное приложение, дал имя проекту - ConsoleHello, указал, где будет храниться проект.

Если принять эти установки, то компилятор создаст решение, имя которого совпадает с именем проекта.

Интегрированная среда разработки IDE (Integrated Development Environment) Visual Studio является многооконной, настраиваемой, обладает большим набором возможностей. Обратим внимание на три окна.

В окне Solution Explorer представлена структура построенного решения.

В окне Properties можно увидеть свойства выбранного элемента решения.

В окне документов отображается выбранный документ, *в данном случае, программный код класса проекта - ConsoleHello.Class1*. Заметьте, в этом окне можно отображать и другие документы, список которых показан в верхней части окна.

Построенное решение содержит, естественно, единственный заданный нами проект - ConsoleHello. Наш проект, включает в себя папку со ссылками на системные пространства имен из библиотеки FCL, файл со значком приложения и два файла с расширением cs. Файл AssemblyInfo содержит информацию, используемую в сборке, а файл со стандартным именем Class1 является построенным по умолчанию классом, который задает точку входа - процедуру Main, содержащую для данного типа проекта только комментарий.

Заметьте, класс проекта погружен в пространство имен, имеющее по умолчанию то же имя, что и решение, и проект. Итак, при создании нового проекта автоматически создается достаточно сложная вложенная структура - решение, содержащее проект, содержащий пространство имен, содержащее класс, содержащий точку входа. Для простых решений такая структурированность представляется избыточной, но для сложных - она осмысленна и полезна.

Помимо понимания структуры решения, стоит также разобраться в трех важных элементах, включенных в начальный проект - предложение using, тэги документации в комментариях и атрибуты.

Пространству имен может предшествовать одно или несколько предложений using, где после ключевого слова следует название пространства имен - из библиотеки FCL или из проектов, связанных с текущим проектом. В данном случае задается пространство имен System - основное пространство имен библиотеки FCL. *Предложение using NameA облегчает запись при использовании классов, входящих в пространство NameA, поскольку в этом случае не требуется каждый раз задавать полное имя класса с указанием имени пространства, содержащего этот класс.* Заметьте, полное имя может

потребоваться, если в нескольких используемых пространствах имен имеются классы с одинаковыми именами.

Все языки допускают комментарии. В C#, как и в C++, допускаются однострочные и многострочные комментарии. Первые начинаются с двух символов косой черты. Весь текст до конца строки, следующий за этой парой символов, (например, "//любой текст") воспринимается как комментарий, не влияющий на выполнение программного кода. Началом многострочного комментария является пара символов /*, а концом - */. Заметьте, тело процедуры Main содержит три однострочных комментария.

Здесь же, в проекте, построенном по умолчанию, мы встречаемся с еще одной весьма важной новинкой C# - XML-тегами, формально являющимися частью комментария. Отметим, что описанию класса Class1 и описанию метода Main предшествует заданный в строчном комментарии XML-тег <summary>. Этот тэг распознается специальным инструментарием, строящим XML-отчет проекта. Идея самодокументируемых программных проектов, у которых документация является неотъемлемой частью, является важной составляющей стиля компонентного надежного программирования на C#. Мы рассмотрим реализацию этой идеи в свое время более подробно, но уже с первых шагов будем использовать теги документирования и строить XML-отчеты. Заметьте, кроме тега <summary> возможны и другие тэги, включаемые в отчеты. Некоторые теги добавляются почти автоматически. Если в нужном месте (перед объявлением класса, метода) набрать подряд три символа косой черты, то автоматически вставится тэг документирования, так что останется только дополнить его соответствующей информацией.

5 Общий взгляд. Система типов. Типы-значения и ссылочные типы. Встроенные типы. Сравнение с типами C++.

В первых языках программирования понятие класса отсутствовало - рассматривались только типы данных. При определении типа явно задавалось только множество возможных значений, которые могут принимать переменные этого типа. Например, тип integer задает целые числа в некотором диапазоне.

Классы и объекты впервые появились в программировании в языке Симула 67. Определение класса наряду с описанием данных содержало четкое определение операций или методов, применимых к данным. Объекты - экземпляры класса являются обобщением понятия переменной.

-данные, задающие свойства объектов класса;

-методы, определяющие поведение объектов класса;

-события, которые могут происходить с объектами класса.

Типы данных принято разделять на простые и сложные в зависимости от того, как устроены их данные. У простых (скалярных) типов возможные значения данных едины и неделимы. Сложные типы характеризуются способом структуризации данных - одно значение сложного типа состоит из множества значений данных, организующих сложный тип.

Есть и другие критерии классификации типов. Так, типы разделяются на встроенные типы и типы, определенные программистом (пользователем). Встроенные типы изначально принадлежат языку программирования и составляют его базис. В основе системы типов любого языка программирования всегда лежит базисная система типов, встроенных в язык. На их основе программист может строить собственные, им самим определенные типы данных. Но способы (правила) создания таких типов являются базисными, встроенными в язык.

Типы данных разделяются также на статические и динамические. Для данных статического типа память отводится в момент объявления, требуемый размер данных (памяти) известен при их объявлении. Для данных динамического типа размер данных в момент объявления обычно неизвестен и память им выделяется динамически по запросу в процессе выполнения программы.

Еще одна важная классификация типов - это их деление на значимые и ссылочные. Для значимых типов значение переменной (объекта) является неотъемлемой собственностью переменной (точнее, собственностью является память, отводимая значению, а само значение может изменяться). Для ссылочных типов значением служит ссылка на некоторый объект в памяти, расположенный обычно в динамической памяти - "куче". Объект, на который указывает ссылка, может быть разделяемым. Это означает, что несколько ссылочных переменных могут указывать на один и тот же объект и разделять его значения. Значимый тип принято называть развернутым, подчеркивая тем самым, что значение объекта развернуто непосредственно в памяти, отводимой объекту.

Для большинства процедурных языков, реально используемых программистами - Паскаль, C++, Java, Visual Basic, C#, - система встроенных типов более или менее одинакова. Всегда в языке присутствуют арифметический, логический (булев), символьный типы. Арифметический тип всегда разбивается на подтипы. Всегда допускается организация данных в виде массивов и записей (структур). Внутри арифметического типа всегда допускаются преобразования, всегда есть функции, преобразующие строку в число и обратно

Система типов

для C# принципиальный характер. Согласно этой классификации все типы можно разделить на четыре категории:

Типы-значения (value), или значимые типы.

Ссылочные (reference).

Указатели (pointer).

Тип void.

Эта классификация основана на том, где и как хранятся значения типов. Для ссылочного типа значение задает ссылку на область памяти в "куче", где расположен соответствующий объект. Для значимого типа используется прямая адресация, значение хранит собственно данные, и память для них отводится, как правило, в стеке.

В отдельную категорию выделены указатели, что подчеркивает их особую роль в языке. Указатели имеют ограниченную область действия и могут использоваться только в небезопасных блоках, помеченных как unsafe.

Особый статус имеет и тип void, указывающий на отсутствие какого-либо значения.

В языке C# жестко определено, какие типы относятся к ссылочным, а какие - к значимым. К значимым типам относятся: логический, арифметический, структуры, перечисление. Массивы, строки и классы относятся к ссылочным типам.

В C# массивы рассматриваются как динамические, их размер может определяться на этапе вычислений, а не в момент трансляции. Строки в C# также рассматриваются как динамические переменные, длина которых может изменяться. Поэтому строки и массивы относятся к ссылочным типам, требующим распределения памяти в "куче".

Структуры C# представляют частный случай класса. Определив свой класс как структуру, программист получает возможность отнести класс к значимым типам, что иногда бывает крайне полезно.

Логический тип			
Имя типа	**Системный тип**	**Значения**	**Размер**
bool	System.Boolean	true, false	8 бит
Арифметические целочисленные типы			
Имя типа	**Системный тип**	**Диапазон**	**Размер**
sbyte	System.SByte	-128 — 127	Знаковое, 8 Бит
byte	System.Byte	0 — 255	Беззнаковое, 8 Бит
short	System.Short	-32768 — 32767	Знаковое, 16 Бит
ushort	System.UShort	0 — 65535	Беззнаковое, 16 Бит
int	System.Int32	≈ (-2*10^9 — 2*10^9)	Знаковое, 32 Бит
uint	System.UInt32	≈ (0 — 4*10^9)	Беззнаковое, 32 Бит
long	System.Int64	≈ (-9*10^18 — 9*10^18)	Знаковое, 64 Бит
ulong	System.UInt64	≈ (0— 18*10^18)	Беззнаковое, 64 Бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**

float	System.Single	+1.5*10^-45 -/+3.4*10^38	7 цифр
double	System.Double	+5.0*10^-324 -/+1.7*10^308	15-16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
decimal	System.Decimal	+1.0*10^-28 -+7.9*10^28	28-29 значащих цифр
Символьные типы			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
char	System.Char	U+0000 - U+ffff	16 бит Unicode символ
string	System.String	Строка из символов Unicode	
Объектный тип			
Имя типа	**Системный тип**	**Примечание**	
object	System.Object	Прародитель всех *встроенных* и пользовательских типов	

Система встроенных типов языка C# не только содержит практически все встроенные типы (за исключением long double) стандарта языка C++, но и перекрывает его разумным образом. В частности, тип string является встроенным в язык, что вполне естественно. В области совпадения сохранены имена типов, принятые в C++, что облегчает жизнь тем, кто привык работать на C++, но собирается по тем или иным причинам перейти на язык C#.

6. Типы и классы. Преобразование переменных в объекты. Операции «упаковать» и «распаковать»

Язык C# в большей степени, чем язык C++, является языком объектного программирования. В языке C# сглажено различие между типом и классом. Все типы - встроенные и пользовательские - одновременно являются классами, связанными отношением наследования. Родительским, базовым классом является класс Object. Все остальные типы или, точнее, классы являются его потомками, наследуя методы этого класса. У класса Object есть четыре наследуемых метода:

bool Equals (object obj) - проверяет эквивалентность текущего объекта и объекта, переданного в качестве аргумента;

System.Type GetType () - возвращает системный тип текущего объекта;

string ToString () - возвращает строку, связанную с объектом. Для арифметических типов возвращается значение, преобразованное в строку;

int GetHashCode() - служит как хэш-функция в соответствующих алгоритмах поиска по ключу при хранении данных в хэш-таблицах.

Естественно, что все встроенные типы нужным образом переопределяют методы родителя и добавляют собственные методы и свойства. Учитывая, что и типы, создаваемые пользователем, также являются потомками класса Object, то для них необходимо переопределить методы родителя, если предполагается использование этих методов; реализация родителя, предоставляемая по умолчанию, не обеспечивает нужного эффекта.

Начнем с вполне корректного в языке C# примера объявления переменных и присваивания им значений:

```
int x=11;
int v = new Int32();
v = 007;
string s1 = "Agent";
s1 = s1 + v.ToString() +x. ToString();
```

В этом примере переменная x объявляется как обычная переменная типа int. В то же время для объявления переменной v того же типа int используется стиль, принятый для объектов. В объявлении применяется конструкция new и вызов конструктора класса. В операторе присваивания, записанном в последней строке фрагмента, для обеих переменных вызывается метод ToString, как это делается при работе с объектами. Этот метод, наследуемый от родительского класса Object, переопределенный в классе int, возвращает строку с записью целого. Сообщу еще, что класс int не только наследует методы родителя - класса Object, - но и дополнительно определяет метод CompareTo, выполняющий сравнение целых, и метод GetTypeCode, возвращающий системный код типа.

int - это и тип, и класс. В зависимости от контекста x может восприниматься как переменная типа int или как объект класса int. Это же верно и для всех остальных value-типов. Замечу еще, что все значимые типы фактически реализованы как структуры, представляющие частный случай класса.

Остается понять, для чего в языке C# введена такая двойственность. Дело в том, что значимые типы эффективнее в реализации, им проще отводить память, так что именно соображения эффективности реализации заставили авторов языка сохранить значимые типы. Более важно, что зачастую необходимо оперировать значениями, а не ссылками на

них, хотя бы из-за различий в семантике присваивания для переменных ссылочных и значимых типов.

С другой стороны, в определенном контексте крайне полезно рассматривать переменные типа int как настоящие объекты и обращаться с ними как с объектами. В частности, полезно иметь возможность создавать и работать со списками, чьи элементы являются разнородными объектами, в том числе и принадлежащими к значимым типам.

Семантика присваивания

Рассмотрим присваивание:

x = e

Чтобы присваивание было допустимым, типы переменной x и выражения e должны быть согласованными. Пусть сущность x согласно объявлению принадлежит классу T. Будем говорить, что тип T основан на классе T и является базовым типом x, так что базовый тип определяется классом объявления. Пусть теперь в рассматриваемом нами присваивании выражение e связано с объектом типа T1.

Определение: тип T1 согласован по присваиванию с базовым типом T переменной x, если класс T1 является потомком класса T.

Присваивание допустимо, если и только если имеет место согласование типов. Так как все классы в языке C# - встроенные и определенные пользователем - по определению являются потомками класса Object, то отсюда и следует наш частный случай - переменным класса Object можно присваивать выражения любого типа.

Несмотря на то, что обстоятельный разговор о наследовании, родителях и потомках нам еще предстоит, лучше с самого начала понимать отношения между родительским классом и классом-потомком, отношения между объектами этих классов.

Класс-потомок при создании наследует все свойства и методы родителя. Родительский класс не имеет возможности наследовать свойства и методы, создаваемые его потомками.

Наследование - это односторонняя операция от родителя к потомку. Ситуация с присваиванием симметричная. Объекту родительского класса присваивается объект класса-потомка. Объекту класса-потомка не может быть присвоен объект родительского класса.

Присваивание - это односторонняя операция от потомка к родителю. Одностороннее присваивание реально означает, что ссылочная переменная родительского класса может быть связана с любыми объектами, имеющими тип потомков родительского класса.

Например, пусть задан некоторый класс Parent, а класс Child - его потомок, объявленный следующим образом:

class Child:Parent {...}

Пусть теперь в некотором классе, являющемся клиентом классов Parent и Child, объявлены переменные этих классов и созданы связанные с ними объекты:

Parent p1 = new Parent(), p2 = new Parent();

Child ch1 = new Child(), ch2 = new Child();

Тогда допустимы присваивания:

p1 = p2; p2= p1; ch1=ch2; ch2 = ch1; p1 = ch1; p2 = ch2;

Но недопустимы присваивания:

ch1 = p1; ch2 = p1; ch2 = p2; ch1 = p2;

Заметьте, ситуация не столь удручающая - сын может вернуть себе переданный родителю объект, задав явное преобразование. Так что следующие присваивания допустимы:

p1 = ch1; ... ch1 = (Child)p1;

Семантика присваивания справедлива и для другого важного случая - при рассмотрении соответствия между формальными и фактическими аргументами процедур и функций. Если формальный аргумент согласно объявлению имеет тип T, а выражение, задающее фактический аргумент, имеет тип T1, то имеет место согласование типов формального и фактического аргумента, если и только если класс T1 является потомком класса T. Отсюда незамедлительно следует, что если формальный параметр процедуры принадлежит классу Object, то фактический аргумент может быть выражением любого типа.

Преобразование к типу object

Рассмотрим частный случай присваивания x = e; когда x имеет тип object. В этом случае гарантируется полная согласованность по присваиванию - выражение e может иметь любой тип. В результате присваивания значением переменной x становится ссылка на объект, заданный выражением e. Заметьте, текущим типом x становится тип объекта, заданного выражением e. Уже здесь проявляется одно из важных различий между классом и типом. Переменная, лучше сказать сущность x, согласно объявлению принадлежит классу Object, но ее тип - тип того объекта, с которым она связана в текущий момент, - может динамически изменяться.

Семантика присваивания. Преобразования между ссылочными и значимыми типами

Рассматривая семантику присваивания и передачи аргументов, мы обошли молчанием один важный вопрос. Будем называть целью левую часть оператора присваивания, а также формальный аргумент при передаче аргументов в процедуру или функцию. Будем называть источником правую часть оператора присваивания, а также фактический аргумент при передаче аргументов в процедуру или функцию. Поскольку источник и цель могут быть как значимого, так и ссылочного типа, то возможны четыре различные комбинации. Рассмотрим их подробнее.

1)Цель и источник значимого типа. Здесь наличествует семантика значимого присваивания. В этом случае источник и цель имеют собственную память для хранения значений. Значения источника заменяют значения соответствующих полей цели. Источник и цель после этого продолжают жить независимо. У них своя память, хранящая после присваивания одинаковые значения.

2)Цель и источник ссылочного типа. Здесь имеет место семантика ссылочного присваивания. В этом случае значениями источника и цели являются ссылки на объекты, хранящиеся в памяти ("куче"). При ссылочном присваивании цель разрывает связь с тем объектом, на который она ссылалась до присваивания, и становится ссылкой на объект, связанный с источником. Результат ссылочного присваивания двоякий. Объект, на который ссылалась цель, теряет одну из своих ссылок и может стать висячим, так что его дальнейшую судьбу определит сборщик мусора. С объектом в памяти, на который ссылался источник, теперь связываются, по меньшей мере, две ссылки, рассматриваемые как различные имена одного объекта. Ссылочное присваивание приводит к созданию псевдонимов - к появлению разных имен у одного объекта. Особо следует учитывать ситуацию, когда цель и/или источник имеет значение void. Если такое значение имеет источник, то в результате присваивания цель получает это значение и более не ссылается ни на какой объект. Если же цель имела значение void, а источник - нет, то в результате присваивания ранее "висячая" цель становится ссылкой на объект, связанный с источником.

3)Цель ссылочного типа, источник значимого типа. В этом случае "на лету" значимый тип преобразуется в ссылочный. Как обеспечивается двойственность существования значимого и ссылочного типа - переменной и объекта? Ответ прост: за счет специальных, эффективно реализованных операций, преобразующих переменную значимого типа в объект и обратно. Операция " **упаковать** " (boxing) выполняется автоматически и неявно в тот момент, когда по контексту требуется объект, а не переменная. *Например, при вызове процедуры WhoIsWho требуется, чтобы аргумент any был объектом. Если фактический аргумент является переменной значимого типа, то автоматически выполняется операция " упаковать ". При ее выполнении создается настоящий объект, хранящий значение переменной. Можно считать, что происходит упаковка переменной в объект. Необходимость в упаковке возникает достаточно часто. Примером может служить и процедура консольного вывода WriteLine класса Console, которой требуются объекты, а передаются зачастую переменные значимого типа.*

4)Цель значимого типа, источник ссылочного типа. В этом случае "на лету" ссылочный тип преобразуется в значимый. Операция " **распаковать** " (unboxing) выполняет обратную операцию, - она "сдирает" объектную упаковку и извлекает хранимое значение. Заметьте, операция " распаковать " не является обратной к операции " упаковать " в строгом смысле этого слова. Оператор obj = x корректен, но выполняемый следом оператор x = obj приведет к ошибке. Недостаточно, чтобы хранимое значение в упакованном объекте точно совпадало по типу с переменной, которой присваивается объект. Необходимо явно заданное преобразование к нужному типу.

7. Преобразования типов. Преобразования внутри арифметического типа. Преобразование строкового типа. Класс Convert и его методы.

Преобразования типов.

Необходимость в преобразовании типов возникает в выражениях, присваиваниях, замене формальных аргументов метода фактическими.

Если при вычислении выражения операнды операции имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа byte должны быть приведены к типу int, поскольку сложение не определено над байтами.

Преобразования ссылочных типов

Поскольку операции над ссылочными типами не определены (исключением являются строки, но операции над ними, в том числе и присваивание, выполняются как над значимыми типами), то необходимость в них возникает только при присваиваниях и вызовах методов. При присваиваниях (замене аргументов) тип источника должен быть согласован с типом цели, то есть объект, связанный с источником, должен принадлежать классу, являющемуся потомком класса цели. В случае согласования типов, ссылочная переменная цели связывается с объектом источника и ее тип динамически изменяется, становясь типом источника. Это преобразование выполняется автоматически и неявно, не требуя от программиста никаких дополнительных указаний. Если же тип цели является потомком типа источника, то неявное преобразование отсутствует, даже если объект, связанный с источником, принадлежит типу цели. Явное преобразование, заданное программистом, позволяет справиться с этим случаем. Ответственность за корректность явных преобразований лежит на программисте, так что может возникнуть ошибка на этапе выполнения, если связываемый объект реально не является объектом класса цели.

Преобразования типов в выражениях

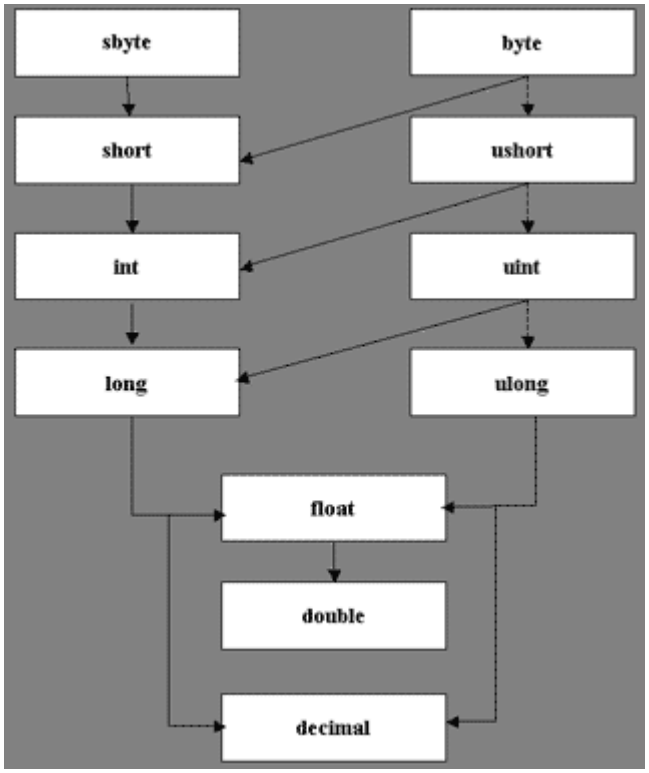
В C# такие преобразования делятся на неявные и явные.

К неявным относятся те преобразования, результат выполнения которых всегда успешен и не приводит к потере точности данных. Неявные преобразования выполняются автоматически. Для арифметических данных это означает, что в неявных преобразованиях диапазон типа назначения содержит в себе диапазон исходного типа. Например, преобразование из типа byte в тип int относится к неявным, поскольку диапазон типа byte является подмножеством диапазона int. Это преобразование всегда успешно и не может приводить к потере точности. Заметьте, преобразования из целочисленных типов к типам с плавающей точкой относятся к неявным. Хотя здесь и может происходить некоторое искажение значения, но точность представления значения сохраняется.

К явным относятся разрешенные преобразования, успех выполнения которых не гарантируется или может приводить к потере точности. Такие потенциально опасные преобразования должны быть явно заданы программистом. Преобразование из типа int в тип byte относится к явным, поскольку оно небезопасно и может приводить к потере значащих цифр. Заметьте, не для всех типов существуют явные преобразования. В этом случае требуются другие механизмы преобразования типов.

Преобразования внутри арифметического типа

Арифметический тип, как показано в таблице, распадается на 11 подтипов.



Диаграмма, приведенная на рисунке, позволяет ответить на ряд важных вопросов, связанных с существованием преобразований между типами. Если на диаграмме задан путь (стрелками) от типа A к типу B, то это означает существование неявного преобразования из типа A в тип B. Все остальные преобразования между подтипами арифметического типа существуют, но являются явными. Заметьте, что циклов на диаграмме нет, все стрелки односторонние, так что преобразование, обратное к неявному, всегда должно быть задано явным образом.

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

Явные преобразования

Явные преобразования могут быть опасными из-за потери точности. Поэтому они выполняются по указанию программиста, - на нем лежит вся ответственность за результаты.

Преобразования строкового типа

Важным классом преобразований являются преобразования в строковый тип и наоборот. Преобразования в строковый тип всегда определены, поскольку, все типы являются потомками базового класса Object, а, следовательно, обладают методом ToString(). Для встроенных типов определена подходящая реализация этого метода. В частности, для всех подтипов арифметического типа метод ToString() возвращает в подходящей форме строку, задающую соответствующее значение арифметического типа. Заметьте, метод ToString можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда по контексту требуется преобразование к строковому типу. Вот соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования в строку данных различного типа.
/// </summary>
public void ToStringTest()
```

```
{
  s ="Владимир Петров ";
  s1 =" Возраст: "; ux = 27;
  s = s + s1 + ux.ToString();
  s1 =" Зарплата: "; dy = 2700.50;
  s = s + s1 + dy;
  WhoIsWho("s",s);
}
```

Здесь для переменной ux метод был вызван явно, а для переменной dy он вызывается автоматически.

Преобразования из строкового типа в другие типы, например, в арифметический, должны выполняться явно. Но явных преобразований между арифметикой и строками не существует. Необходимы другие механизмы, и они в C# имеются. Для этой цели можно использовать соответствующие методы класса Convert библиотеки FCL, встроенного в пространство имен System. Приведу соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования строки в данные различного типа.
/// </summary>
public void FromStringTest()
{
  s ="Введите возраст ";
  Console.WriteLine(s);
  s1 = Console.ReadLine();
  ux = Convert.ToUInt32(s1);
  WhoIsWho("Возраст: ",ux);
  s ="Введите зарплату ";
  Console.WriteLine(s);
  s1 = Console.ReadLine();
  dy = Convert.ToDouble(s1);
  WhoIsWho("Зарплата: ",dy);
}
```

Этот пример демонстрирует ввод с консоли данных разных типов. Данные, читаемые с консоли методом ReadLine или Read, всегда представляют собой строку, которую затем необходимо преобразовать в нужный тип. Тут-то и вызываются соответствующие методы класса Convert. Естественно, для успеха преобразования строка должна содержать значение в формате, допускающем подобное преобразование.

Преобразования и класс Convert

Класс Convert, определенный в пространстве имен System, играет важную роль, обеспечивая необходимые преобразования между различными типами.

Методы класса Convert поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит 15 статических методов вида To <Type> (ToBoolean(),...ToUInt64()), где Type может принимать значения от Boolean до UInt64 для всех встроенных типов. Единственным исключением является тип object, - метода ToObject нет по понятным причинам, поскольку для всех типов существует неявное преобразование к типу object.

Существует возможность преобразования к системному типу DateTime, который хотя и не является встроенным типом языка C#, но допустим в программах, как и любой другой системный тип.

```
// System type: DateTime
System.DateTime dat = Convert.ToDateTime("15.03.2003");
Console.WriteLine("Date = {0}", dat);
```

Результатом вывода будет строка:

```
Date = 15.03.2003  0:00:00
```

Все методы To <Type> класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#.

Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования значений объектов и другие методы.

8. Проверяемые преобразования. Управление проверкой арифметических преобразований.

Проверяемые преобразования

При выполнении явных преобразований могут возникать нежелательные явления, например, потеря точности. Вся ответственность за это ложится на программиста, и легче ему от этого не становится.

Устранить эту проблему позволит язык C#. Язык C# позволяет создать проверяемый блок, в котором будет осуществляться проверка результата вычисления арифметических выражений. Если результат вычисления значения источника выходит за диапазон возможных значений целевой переменной, то возникнет исключение (говорят также: "будет выброшено исключение ") соответствующего типа. Если предусмотрена обработка исключения, то дальнейшее зависит от обработчика исключения. В лучшем случае, программа сможет продолжить корректное выполнение. В худшем, - она остановится и выдаст информацию об ошибке. Заметьте, не произойдет самого опасного - продолжения работы программы с неверными данными.

Синтаксически проверяемый блок предваряется ключевым словом checked. В теле такого блока арифметические преобразования проверяются на допустимость. Естественно, подобная проверка требует дополнительных временных затрат. Если группа операторов в теле такого блока нам кажется безопасной, то их можно выделить в непроверяемый блок, используя ключевое слово unchecked. Замечу еще, что и в непроверяемом блоке при работе методов Convert все опасные преобразования проверяются и приводят к выбрасыванию исключений.

```
/// <summary>
/// Демонстрация проверяемых и непроверяемых преобразований.
/// Опасные проверяемые преобразования приводят к исключениям.
/// Опасные непроверяемые преобразования приводят к неверным
/// результатам, что совсем плохо.
/// </summary>
public void CheckUncheckTest()
{
  x = -25^2;
  WhoIsWho ("x", x);
  b= 255;
  WhoIsWho("b",b);
  // Проверяемые опасные преобразования.
  // Возникают исключения, перехватываемые catch-блоком.
  checked
  {
        try
        {
              b += 1;
        }
        catch (Exception e)
        {
              Console.WriteLine("Переполнение при вычислении b");
              Console.WriteLine(e);
        }
        try
```

```
{
        b = (byte)x;
}
catch (Exception e)
{
        Console.WriteLine("Переполнение при преобразовании к byte");
        Console.WriteLine(e);
}
// непроверяемые опасные преобразования
unchecked
{
        try
        {
                b +=1;
                WhoIsWho ("b", b);
                b = (byte)x;
                WhoIsWho ("b", b);
                ux= (uint)x;
                WhoIsWho ("ux", ux);
                Console.WriteLine("Исключений нет, но результаты не верны!");
        }
        catch (Exception e)
        {
                Console.WriteLine("Этот текст не должен появляться");
                Console.WriteLine(e);
        }
        // автоматическая проверка преобразований в Convert
        // исключения возникают, несмотря на unchecked
        try
        {
                b = Convert.ToByte(x);
        }
        catch (Exception e)
        {
                Console.WriteLine("Переполнение при
                        преобразовании к byte!");
                Console.WriteLine(e);
        }
        try
        {
                ux= Convert.ToUInt32(x);
        }
        catch (Exception e)
        {
                Console.WriteLine("Потеря знака при
                        преобразовании к uint!");
                Console.WriteLine(e);
```

9. Преобразование типов. Преобразования внутри арифметического типа. Преобразование строкового типа.

Преобразования типов.

Необходимость в преобразовании типов возникает в выражениях, присваиваниях, замене формальных аргументов метода фактическими.

Если при вычислении выражения операнды операции имеют разные типы, то возникает необходимость приведения их к одному типу. Такая необходимость возникает и тогда, когда операнды имеют один тип, но он несогласован с типом операции. Например, при выполнении сложения операнды типа byte должны быть приведены к типу int, поскольку сложение не определено над байтами.

Преобразования ссылочных типов

Поскольку операции над ссылочными типами не определены (исключением являются строки, но операции над ними, в том числе и присваивание, выполняются как над значимыми типами), то необходимость в них возникает только при присваиваниях и вызовах методов. При присваиваниях (замене аргументов) тип источника должен быть согласован с типом цели, то есть объект, связанный с источником, должен принадлежать классу, являющемуся потомком класса цели. В случае согласования типов, ссылочная переменная цели связывается с объектом источника и ее тип динамически изменяется, становясь типом источника. Это преобразование выполняется автоматически и неявно, не требуя от программиста никаких дополнительных указаний. Если же тип цели является потомком типа источника, то неявное преобразование отсутствует, даже если объект, связанный с источником, принадлежит типу цели. Явное преобразование, заданное программистом, позволяет справиться с этим случаем. Ответственность за корректность явных преобразований лежит на программисте, так что может возникнуть ошибка на этапе выполнения, если связываемый объект реально не является объектом класса цели.

Преобразования типов в выражениях

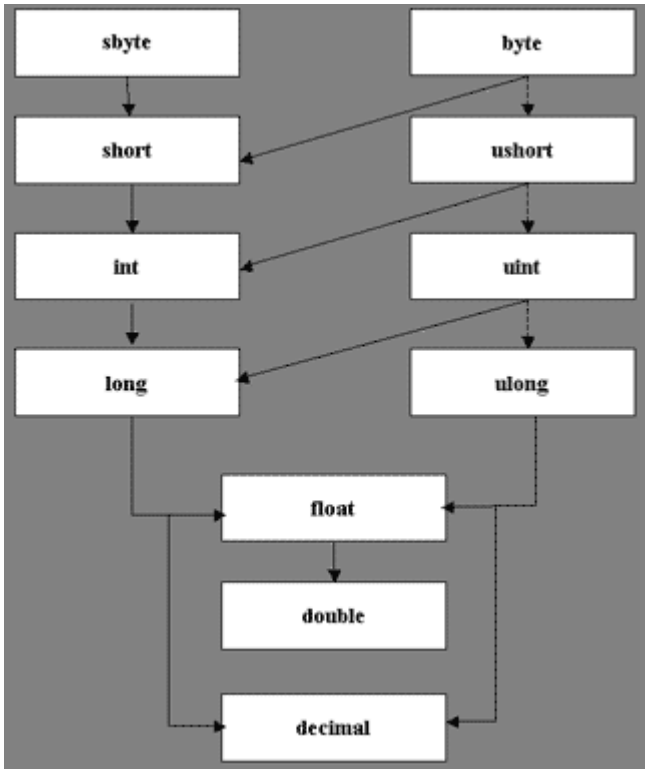
В C# такие преобразования делятся на неявные и явные.

К неявным относятся те преобразования, результат выполнения которых всегда успешен и не приводит к потере точности данных. Неявные преобразования выполняются автоматически. Для арифметических данных это означает, что в неявных преобразованиях диапазон типа назначения содержит в себе диапазон исходного типа. Например, преобразование из типа byte в тип int относится к неявным, поскольку диапазон типа byte является подмножеством диапазона int. Это преобразование всегда успешно и не может приводить к потере точности. Заметьте, преобразования из целочисленных типов к типам с плавающей точкой относятся к неявным. Хотя здесь и может происходить некоторое искажение значения, но точность представления значения сохраняется.

К явным относятся разрешенные преобразования, успех выполнения которых не гарантируется или может приводить к потере точности. Такие потенциально опасные преобразования должны быть явно заданы программистом. Преобразование из типа int в тип byte относится к явным, поскольку оно небезопасно и может приводить к потере значащих цифр. Заметьте, не для всех типов существуют явные преобразования. В этом случае требуются другие механизмы преобразования типов.

Преобразования внутри арифметического типа

Арифметический тип, как показано в таблице, распадается на 11 подтипов.



Диаграмма, приведенная на рисунке, позволяет ответить на ряд важных вопросов, связанных с существованием преобразований между типами. Если на диаграмме задан путь (стрелками) от типа A к типу B, то это означает существование неявного преобразования из типа A в тип B. Все остальные преобразования между подтипами арифметического типа существуют, но являются явными. Заметьте, что циклов на диаграмме нет, все стрелки односторонние, так что преобразование, обратное к неявному, всегда должно быть задано явным образом.

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

Явные преобразования

Явные преобразования могут быть опасными из-за потери точности. Поэтому они выполняются по указанию программиста, - на нем лежит вся ответственность за результаты.

Преобразования строкового типа

Важным классом преобразований являются преобразования в строковый тип и наоборот. Преобразования в строковый тип всегда определены, поскольку, все типы являются потомками базового класса Object, а, следовательно, обладают методом ToString(). Для встроенных типов определена подходящая реализация этого метода. В частности, для всех подтипов арифметического типа метод ToString() возвращает в подходящей форме строку, задающую соответствующее значение арифметического типа. Заметьте, метод ToString можно вызывать явно, но, если явный вызов не указан, то он будет вызываться неявно, всякий раз, когда по контексту требуется преобразование к строковому типу. Вот соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования в строку данных различного типа.
/// </summary>
public void ToStringTest()
```

```
{
  s ="Владимир Петров ";
  s1 =" Возраст: "; ux = 27;
  s = s + s1 + ux.ToString();
  s1 =" Зарплата: "; dy = 2700.50;
  s = s + s1 + dy;
  WhoIsWho("s",s);
}
```

Здесь для переменной ux метод был вызван явно, а для переменной dy он вызывается автоматически.

Преобразования из строкового типа в другие типы, например, в арифметический, должны выполняться явно. Но явных преобразований между арифметикой и строками не существует. Необходимы другие механизмы, и они в C# имеются. Для этой цели можно использовать соответствующие методы класса Convert библиотеки FCL, встроенного в пространство имен System. Приведу соответствующий пример:

```
/// <summary>
/// Демонстрация преобразования строки в данные различного типа.
/// </summary>
public void FromStringTest()
{
  s ="Введите возраст ";
  Console.WriteLine(s);
  s1 = Console.ReadLine();
  ux = Convert.ToUInt32(s1);
  WhoIsWho("Возраст: ",ux);
  s ="Введите зарплату ";
  Console.WriteLine(s);
  s1 = Console.ReadLine();
  dy = Convert.ToDouble(s1);
  WhoIsWho("Зарплата: ",dy);
}
```

Этот пример демонстрирует ввод с консоли данных разных типов. Данные, читаемые с консоли методом ReadLine или Read, всегда представляют собой строку, которую затем необходимо преобразовать в нужный тип. Тут-то и вызываются соответствующие методы класса Convert. Естественно, для успеха преобразования строка должна содержать значение в формате, допускающем подобное преобразование.

10. Класс Convert и его методы. Проверяемые преобразования. Управление проверкой арифметических преобразований.

Преобразования и класс Convert

Класс Convert, определенный в пространстве имен System, играет важную роль, обеспечивая необходимые преобразования между различными типами.

Методы класса Convert поддерживают общий способ выполнения преобразований между типами. Класс Convert содержит 15 статических методов вида To <Type> (ToBoolean(),...ToUInt64()), где Type может принимать значения от Boolean до UInt64 для всех встроенных типов. Единственным исключением является тип object, - метода ToObject нет по понятным причинам, поскольку для всех типов существует неявное преобразование к типу object.

Существует возможность преобразования к системному типу DateTime, который хотя и не является встроенным типом языка C#, но допустим в программах, как и любой другой системный тип.

```
// System type: DateTime
System.DateTime dat = Convert.ToDateTime("15.03.2003");
Console.WriteLine("Date = {0}", dat);
```

Результатом вывода будет строка:

```
Date = 15.03.2003  0:00:00
```

Все методы To <Type> класса Convert перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа. Так что фактически эти методы задают все возможные преобразования между всеми встроенными типами языка C#.

Кроме методов, задающих преобразования типов, в классе Convert имеются и другие методы, например, задающие преобразования символов Unicode в однобайтную кодировку ASCII, преобразования значений объектов и другие методы.

Проверяемые преобразования

При выполнении явных преобразований могут возникать нежелательные явления, например, потеря точности. Вся ответственность за это ложится на программиста, и легче ему от этого не становится.

Устранить эту проблему позволит язык C#. Язык C# позволяет создать проверяемый блок, в котором будет осуществляться проверка результата вычисления арифметических выражений. Если результат вычисления значения источника выходит за диапазон возможных значений целевой переменной, то возникнет исключение (говорят также: "будет выброшено исключение ") соответствующего типа. Если предусмотрена обработка исключения, то дальнейшее зависит от обработчика исключения. В лучшем случае, программа сможет продолжить корректное выполнение. В худшем, - она остановится и выдаст информацию об ошибке. Заметьте, не произойдет самого опасного - продолжения работы программы с неверными данными.

Синтаксически проверяемый блок предваряется ключевым словом checked. В теле такого блока арифметические преобразования проверяются на допустимость. Естественно, подобная проверка требует дополнительных временных затрат. Если группа операторов в теле такого блока нам кажется безопасной, то их можно выделить в непроверяемый блок, используя ключевое слово unchecked. Замечу еще, что и в непроверяемом блоке при работе методов Convert все опасные преобразования проверяются и приводят к выбрасыванию исключений.

11. ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ. СИНТАКСИСИ ОБЪЯВЛЕНИЯ. ИНИЦИАЛИЗАЦИЯ. ВРЕМЯ ЖИЗНИ И ОБЛАСТЬ ВИДИМОСТИ. ГДЕ ОБЪЯВЛЯЮТСЯ ПЕРЕМЕННЫЕ?

Объявление переменных

С объектной точки зрения *переменная* — это экземпляр типа. Скалярную *переменную* можно рассматривать как сущность, обладающую именем, значением и типом. Имя и тип задаются при объявлении *переменной* и остаются неизменными на все *время ее жизни*. *Значение переменной* может меняться в ходе вычислений, эта возможность вариации значений и дала имя понятию *переменная* **(Variable)** в математике и программировании. Получение начального значения *переменной* называется ее *инициализацией*. Важной новинкой языка C# является требование обязательной *инициализации переменной* до начала ее использования. Попытка использовать неинициализированную *переменную* приводит к ошибкам, обнаруживаемым еще на этапе компиляции. *Инициализация переменных*, как правило, выполняется в момент объявления, хотя и может быть отложена.

Синтаксис объявления

Общий *синтаксис объявления* сущностей в C# похож на *синтаксис объявления* в C++, хотя и имеет ряд отличий. Общая структура объявления:

[<атрибуты>] [<модификаторы>] <тип> <объявители>;

При объявлении *переменных* чаще всего задаются модификаторы доступа - public, private и другие. Если атрибуты и модификаторы могут и не указываться в объявлении, то задание типа необходимо всегда.

При объявлении простых *переменных* указывается их тип и список *объявителей*, где *объявитель* - это имя или имя с *инициализацией*. Список *объявителей* позволяет в одном объявлении задать несколько *переменных* одного типа. Если *объявитель* задается именем *переменной*, то имеет место *объявление с отложенной инициализацией*. Хороший стиль программирования предполагает задание *инициализации переменной* в момент ее объявления. *Инициализацию* можно осуществлять двояко - обычным присваиванием или в объектной манере. Во втором случае для *переменной* используется конструкция new и вызывается конструктор по умолчанию. Процедура SimpleVars класса Testing иллюстрирует различные способы объявления *переменных* и простейшие вычисления над ними:

```
public void SimpleVars()
{
 //Объявления локальных переменных
 int x, s; //без инициализации
 int y =0, u = 77; //обычный способ инициализации
 //допустимая инициализация
 float w1=0f, w2 = 5.5f, w3 =w1+ w2 + 125.25f;
```

```
        //допустимая инициализация в объектном стиле
        int z= new int();
        //Недопустимая инициализация.
        //Конструктор с параметрами не определен
        //int v = new int(77);
              x=u+y; //теперь x инициализирована
        if(x> 5) s = 4;
        for (x=1; x<5; x++)s=5;
        //Инициализация в if и for не рассматривается,
        //поэтому s считается неинициализированной переменной
        //Ошибка компиляции:использование неинициализированной переменной
        //Console.WriteLine("s= {0}",s);
    }  //SimpleVars
```

В первой строке объявляются *переменные* x и s с отложенной *инициализацией*. Заметьте (и это важно!), что всякая попытка использовать еще не инициализированную *переменную* в правых частях операторов присваивания, в вызовах функций, вообще в вычислениях приводит к ошибке уже на этапе компиляции.

Последующие объявления *переменных* эквивалентны по сути, но демонстрируют два стиля *инициализации* - обычный и объектный. Обычная форма *инициализации* предпочтительнее не только в силу своей естественности, но она и более эффективна, поскольку в этом случае инициализирующее выражение может быть достаточно сложным, с *переменными* и функциями. На практике объектный стиль для скалярных *переменных* используется редко. Вместе с тем полезно понимать, что объявление с *инициализацией* int y =0 можно рассматривать как создание нового объекта (new) и вызова для него конструктора по умолчанию. При *инициализации* в объектной форме может быть вызван только конструктор по умолчанию, другие конструкторы с параметрами для встроенных типов не определены. В примере закомментировано объявление *переменной* v с *инициализацией* в объектном стиле, приводящее к ошибке, где делается попытка дать *переменной* значение, передавая его конструктору в качестве параметра.

Время жизни и область видимости переменных

Существует два основных вида области видимости: локальная область видимости и глобальная область видимости.

Переменная, объявленная вне всех функций, помещается в глобальную область видимости. Доступ к таким переменным может осуществляться из любого места программы. Такие переменные располагаются в глобальном пуле памяти, поэтому время их жизни совпадает со временем жизни программы.

Переменная, объявленная внутри блока (часть кода, заключенная в фигурные скобки), принадлежит локальной области видимости. Такая переменная не видна (поэтому и недоступна) за пределами блока, в котором она объявлена. Переменная, объявленная локально, располагается на стеке, и время жизни такой переменной совпадает со временем жизни функции.

Так как областью видимости локальной переменной является блок, в котором она объявлена, то существует возможность объявлять переменные с именем, совпадающим с именами переменных, объявленных в других блоках; а также объявленных на более верхних уровнях, вплоть до глобального.

Область видимости зависит от того, где и как, в каком контексте объявлены переменные. В языке C# не так уж много возможностей для объявления переменных, пожалуй, меньше, чем в любом другом языке. В C# вообще нет настоящих глобальных переменных. Их отсутствие не следует считать некоторым недостатком C#, это достоинство языка.

Первая важнейшая роль переменных, - они задают свойства структур, интерфейсов, классов. В языке C# такие переменные называются полями (fields). Поля объявляются при описании класса (и его частных случаев - интерфейса, структуры). Когда конструктор класса создает очередной объект - экземпляр класса, то он в динамической памяти создает набор полей, определяемых классом, и записывает в них значения, характеризующие свойства данного конкретного экземпляра. Так что каждый объект в памяти можно рассматривать как набор соответствующих полей класса со своими значениями. Время существования и область видимости полей определяются объектом, которому они принадлежат. Объекты в динамической памяти, с которыми не связана ни одна ссылочная переменная, становятся недоступными. Реально они оканчивают свое существование, когда сборщик мусора (garbage collector) выполнит чистку "кучи". Для значимых типов, к которым принадлежат экземпляры структур, жизнь оканчивается при завершении блока, в котором они объявлены.

Есть одно важное исключение. Некоторые поля могут жить дольше. Если при объявлении класса поле объявлено с модификатором static, то такое поле является частью класса и единственным на все его экземпляры. Поэтому static - поля живут так же долго, как и сам класс. Более подробно эти вопросы будут обсуждаться при рассмотрении классов, структур, интерфейсов.

12. ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ. ЕСТЬ ЛИ ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ В C#? КОНСТАНТЫ

Локальные переменные

Во всех языках программирования, в том числе и в C#, основной контекст, в котором появляются *переменные*, - это процедуры. *Переменные*, объявленные на уровне процедуры, называются *локальными*, - они локализованы в процедуре.

Переменную можно объявлять в любой точке процедурного блока. *Область ее видимости* распространяется от точки объявления до конца процедурного блока.

Процедурный блок в C# имеет сложную структуру; в него могут быть вложены другие блоки, связанные с операторами выбора, цикла и так далее. В каждом таком блоке, в свою очередь, допустимы вложения блоков. В каждом внутреннем блоке допустимы объявления *переменных*. *Переменные*, объявленные во внутренних блоках, локализованы именно в этих блоках, их *область видимости* и *время жизни* определяются этими блоками. *Локальные переменные* начинают существовать при достижении вычислений в блоке точки объявления и перестают существовать, когда процесс вычисления завершает выполнение операторов блока. Можно полагать, что для каждого такого блока выполняется так называемый пролог и эпилог. В прологе *локальным переменным* отводится память, в эпилоге память освобождается. Фактически ситуация сложнее, поскольку выделение памяти, а следовательно, и начало жизни *переменной*, объявленной в блоке, происходит не в момент входа в блок, а лишь тогда, когда достигается точка объявления *локальной переменной*.

Пример. В класс Testing добавлен метод с именем ScopeVar, вызываемый в процедуре Main. Вот код этого метода:

```
public void ScopeVar(int x)
{int y =77; string s = name;
  if (s=="Точка1")
  {int u = 5; int v = u+y; x +=1;
                    Console.WriteLine("y= {0}; u={1};
                          v={2}; x={3}", y,u,v,x);}
  else
  {int u= 7; int v= u+y;
                    Console.WriteLine("y= {0}; u={1}; v={2}", y,u,v);}
  //Console.WriteLine("y= {0}; u={1}; v={2}",y,u,v);
  //Локальные переменные не могут быть статическими.
  //static int Count = 1;
  //Ошибка: использование sum до объявления
  //Console.WriteLine("x= {0}; sum ={1}", x,sum);
  int i;long sum =0;
  for(i=1; i<x; i++)
  {//ошибка: коллизия имен: y
          //float y = 7.7f;
          sum +=i; }
  Console.WriteLine("x= {0}; sum ={1}", x,sum);
}//ScopeVar
```

В теле метода встречаются имена *полей*, аргументов и *локальных переменных*. Эти имена могут совпадать. Например, имя x имеет *поле* класса и формальный аргумент метода. Это допустимая ситуация. В языке C# разрешено иметь *локальные*

*переменные*с именами, совпадающими с именами *полей* класса, - в нашем примере таким является имя y ; однако, запрещено иметь *локальные переменные*, имена которых совпадают с именами формальных аргументов. Этот запрет распространяется не только на внешний уровень процедурного блока, что вполне естественно, но и на все внутренние блоки.

В процедурный блок вложены два блока, порожденные оператором if. В каждом из них объявлены *переменные* с одинаковыми именами u и v. Это корректные объявления, поскольку время существования и *области видимости* этих *переменных* не пересекаются. Итак, для невложенных блоков разрешено объявление *локальных переменных* с одинаковыми именами. *переменные* u и v перестают существовать после выхода из блока, так что операторы печати, расположенные внутри блока, работают корректно, а оператор печати вне блока приводит к ошибке, - u и v здесь не видимы, кончилось *время их жизни*. По этой причине оператор закомментирован.

Выражение, проверяемое в операторе if, зависит от значения *поля* name. Значение *поля* глобально для метода и доступно всюду, если только не перекрывается именем аргумента (как в случае с *полем* x) или *локальной переменной* (как в случае с *полем* y).

В языке C# статическими могут быть только *поля*, но не *локальные переменные*.

Глобальные переменные уровня модуля. Существуют ли они в C#?

Во многих языках программирования *переменные* могут объявляться на уровне модуля. Такие *переменные* называются ***глобальными***. Их область действия распространяется, по крайней мере, на весь модуль. *Глобальные переменные* играют важную роль, поскольку они обеспечивают весьма эффективный способ обмена информацией между различными частями модуля. Обратная сторона эффективности аппарата *глобальных переменных*, - их опасность. Если какая-либо процедура, в которой доступна *глобальная переменная*, некорректно изменит ее значение, то ошибка может проявиться в другой процедуре, использующей эту *переменную*. Найти причину ошибки бывает чрезвычайно трудно.

В языке C# роль модуля играют классы, пространства имен, проекты, решения. *Поля* классов могут рассматриваться как *глобальные переменные* класса. *Поля* экземпляра (открытые и закрытые) - это глобальная информация, которая доступна всем методам класса, играющим второстепенную роль - они обрабатывают данные.

Статические *поля* класса хранят информацию, общую для всех экземпляров класса. Они представляют определенную опасность, поскольку каждый экземпляр способен менять их значения.

В других видах модуля - пространствах имен, проектах, решениях - нельзя объявлять *переменные*. В пространствах имен в языке C# разрешено только объявление классов и их частных случаев: структур, интерфейсов, делегатов, перечислений. Поэтому *глобальных переменных* уровня модуля, в привычном для других языков программирования смысле, в языке C# нет. Классы не могут обмениваться информацией, используя *глобальные переменные*. Все взаимодействие между ними обеспечивается способами, стандартными для объектного подхода. Между классами могут существовать два типа отношений -

клиентские и наследования, а основной способ инициации вычислений - это вызов метода для объекта-цели или вызов обработчика события. *Поля* класса и аргументы метода позволяют передавать и получать нужную информацию. Устранение *глобальных переменных* как источника опасных, трудно находимых ошибок существенно повышает надежность создаваемых на языке C# программных продуктов.

Глобальные переменные уровня процедуры. Существуют ли?

Отвечая на вопрос, вынесенный в заголовок, следует сказать, что *глобальные переменные* на уровне процедуры в языке C#, конечно же, есть, но нет *конфликта имен* между *глобальными* и *локальными переменными* на этом уровне. *Область видимости глобальных переменных* процедурного блока распространяется на весь блок, в котором они объявлены, начиная от точки объявления, и не зависит от существования внутренних блоков. Когда говорят, что в C# нет *глобальных переменных*, то, прежде всего, имеют в виду их отсутствие на уровне модуля. Уже во вторую очередь речь идет об отсутствии *конфликтов имен* на процедурном уровне.

Константы

Константы C# могут появляться, как обычно, в виде литералов и именованных *констант*. Вот пример *константы*, заданной литералом и стоящей в правой части оператора присваивания:

```
const float y = 7.7f;
```

Значение константы " 7.7f " является одновременно ее именем, оно же позволяет однозначно определить тип *константы*.

Всюду, где можно объявить *переменную*, можно объявить и именованную *константу*. *Синтаксис объявления* схож. В объявление добавляется модификатор const, *инициализация констант* обязательна и не может быть отложена. Инициализирующее *выражение* может быть сложным, важно, чтобы оно было вычислимым в момент его определения. Вот пример объявления *констант*:

```
public void Constants()
{
  const int SmallSize = 38, LargeSize =58;
  const int MidSize = (SmallSize + LargeSize)/2;
  const double pi = 3.141593;
  //LargeSize = 60; //Значение константы нельзя изменить.
  Console.WriteLine("MidSize= {0}; pi={1}",
        MidSize, pi);
}//Constants
```

13. ПОСТРОЕНИЕ ВЫРАЖЕНИЙ. ОПЕРАЦИИ И ИХ ПРИОРИТЕТЫ. ОПИСАНИЕ ОПЕРАЦИЙ

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

приоритет операций ;

для операций одного приоритета порядок применения - слева направо или справа налево;

преобразование типов операндов и выбор реализации для перегруженных операций;

тип и значение результата выполнения операции над заданными значениями операндов определенного типа.

Большинство операций в языке C#, их приоритет и порядок наследованы из языка C++. Однако имеются и различия: например, нет операции " , ", позволяющей вычислять список выражений ; добавлены уже упоминавшиеся операции checked и unchecked, применимые к выражениям.

таблица приоритетов операций, в каждой строке которой собраны операции одного приоритета, а строки следуют в порядке приоритетов, от высшего к низшему.

Приоритет	Категория	Операции	Порядок
0	Первичные	(exp) x.y f(x) a[x] x++ x—new sizeof(t) typeof(t) checked(expr) unchecked(expr)	Слева направо
1	Унарные	+ - ! ~ ++x --x (T)x.	Слева направо
2	Мультипликативные(умножение)	* / %	Слева направо
3	Аддитивные(сложение)	+ -	Слева направо
4	Сдвиг	<< >>	Слева направо
5	Отношения, проверка типов	< > <= >= is as	Слева направо
6	Эквивалентность	== !=	Слева направо
7	Логическое И	&	Слева направо
8	Логическое исключающие ИЛИ(xor)	^	Слева направо
9	Логическое ИЛИ(or)	|	Слева направо
10	Условное И	&&	Слева направо
11	Условное ИЛИ	||	Слева направо
12	Условное выражение	? :	Слева направо
13	Присваивание	= *= /= %= += -= <<= >>= &= ^= |=	Слева направо

Под перегрузкой операции понимается существование нескольких реализаций одной и той же операции. Большинство операций языка C# перегружены - одна и та же операция может применяться к операндам различных типов. Поэтому перед выполнением операции идет поиск реализации, подходящей для данных типов операндов. Если же операнды разных типов, то предварительно происходит неявное преобразование типа операнда. Оба операнда могут быть одного типа, но преобразование типов может все равно происходить - по той причине, что для заданных типов нет соответствующей перегруженной операции.

Вычисление выражения начинается с *выполнения операций высшего приоритета.* Первым делом вычисляются выражения в круглых скобках - (expr), определяются значения полей объекта - x.y, вычисляются функции - f(x), переменные с индексами - a[i]. Выполнение этих операций достаточно понятно и не нуждается в комментировании. Операции checked и unchecked включают и выключают проверку преобразований арифметического типа в выражениях, которым они предшествуют.

Операции "увеличить на единицу" и " уменьшить на единицу " могут быть префиксными и постфиксными. К высшему приоритету относятся постфиксные операции x++ и x--. Префиксные операции имеют на единицу меньший приоритет. Главной особенностью как префиксных, так и постфиксных операций является побочный эффект, в результате которого значение x увеличивается (++) или уменьшается (--) на единицу. Для префиксных (++x, --x) операций результатом их выполнения является измененное значение x, постфиксные операции возвращают в качестве результата значение x до изменения.

Разный приоритет префиксных и постфиксных операций носит условный характер. Эти операции применимы только к переменным, свойствам и индексаторам класса, то есть к выражениям, которым отведена область памяти. В языках C++ и C# такие выражения называются l-value, поскольку они могут встречаться в левых частях оператора присваивания.

Операция sizeof возвращает размер значимых типов, заданный в байтах.

Операция typeof, примененная к своему аргументу, возвращает его тип. И здесь в роли аргумента может выступать имя класса, как встроенного, так и созданного пользователем. Возвращаемый операцией результат имеет тип Type. К экземпляру класса применять операцию нельзя, но зато для экземпляра можно вызвать метод GetType, наследуемый всеми классами, и получить тот же результат, что дает typeof с именем данного класса.

операция new. Ключевое слово new используется в двух контекстах - как модификатор и как операция в инициализирующих выражениях объявителя. Во втором случае результатом выполнения операции new является создание нового объекта и вызов соответствующего конструктора.

В языке C# имеются обычные для всех языков ***арифметические операции*** - " +, -, *, /, % ". Все они перегружены. Операции " + " и " - " могут быть унарными и бинарными. Операция деления " / " над целыми типами осуществляет деление нацело, для типов с плавающей и фиксированной точкой - обычное деление. Операция " % " определена над всеми арифметическими типами и возвращает остаток от деления нацело.

Операции отношения можно просто перечислить - в объяснениях они не нуждаются. Всего операций 6 (==, !=, <, >, <=, >=).

Операции сдвига вправо " >> " и сдвига влево " << " в обычных вычислениях применяются редко. Они особенно полезны, если данные рассматриваются как строка битов. Результатом операции является сдвиг строки битов влево или вправо на K разрядов. В применении к обычным целым положительным числам сдвиг вправо равносилен делению нацело на 2^K, а сдвиг влево - умножению на 2^K. Для отрицательных чисел сдвиг влево и деление дают разные результаты, отличающиеся на единицу. В языке C# операции сдвига определены только для некоторых целочисленных типов - int, uint, long, ulong. Величина сдвига должна иметь тип int.

Правила работы с логическими выражениями в языках C# и C++ имеют принципиальные различия. В языке C# неявных преобразований к логическому типу нет даже для целых арифметических типов. логические операции делятся на две категории:

одни выполняются над логическими значениями операндов, другие осуществляют выполнение логической операции над битами операндов. По этой причине в C# существуют две унарные операции отрицания - логическое отрицание, заданное операцией " !", и побитовое отрицание, заданное операцией " ~ ". Первая из них определена над операндом типа bool, вторая - над операндом целочисленного типа, начиная с типа int и выше (int, uint, long, ulong). Результатом операции во втором случае является операнд, в котором каждый бит заменен его дополнением.

Бинарные логические операции " && - условное И " и " || - условное ИЛИ " определены только над данными типа bool. Операции называются условными или краткими, поскольку, будет ли вычисляться второй операнд, зависит от уже вычисленного значения первого операнда. В операции " && ", если первый операнд равен значению false, то второй операнд не вычисляется и результат операции равен false. Аналогично, в операции " || ", если первый операнд равен значению true, то второй операнд не вычисляется и результат операции равен true.

Три бинарные побитовые операции - " & - AND " , " | - OR ", " ^ - XOR " используются двояко. Они определены как над целыми типами выше int, так и над булевыми типами. В первом случае они используются как побитовые операции, во втором - как обычные логические операции. Иногда необходимо, чтобы оба операнда вычислялись в любом случае, тогда без этих операций не обойтись.

Условное выражение начинается с условия, заключенного в круглые скобки, после которого следует знак вопроса и пара выражений, разделенных двоеточием " : ". Условием является выражение типа bool. Если оно истинно, то из пары выражений выбирается первое, в противном случае результатом является значение второго выражения.

Операция приведения к типу

имеет следующий синтаксис:

(type) <унарное выражение>

Она задает явное преобразование типа, определенного выражением, к типу, указанному в скобках. Чтобы операция была успешной, необходимо, чтобы такое явное преобразование существовало. существуют явные преобразования внутри арифметического типа, но не существует, например, явного преобразования арифметического типа в тип bool. При определении пользовательских типов для них могут быть заданы явные преобразования в другие, в том числе встроенные, типы.

```
int p;
double x=3.14159;
p = (int)x;
//b = (bool)x;
```

В данном примере явное преобразование из типа double в тип int выполняется, а преобразование double в тип bool приводит к ошибке, потому и закомментировано.

14. ПРИСВАИВАНИЕ. НОВИНКА C# - ОПРЕДЕЛЕННОЕ ПРИСВАИВАНИЕ

В большинстве языков программирования присваивание - это оператор, а не операция. В языке C# присваивание унаследовало многие особенности присваивания языка C++. В C# оно толкуется как операция, используемая в выражениях. Однако в большинстве случаев присваивание следует рассматривать и использовать как обычный оператор.

Рассмотрим случай реального использования присваивания как операции. В ситуации, называемой множественным присваиванием, списку переменных присваивается одно и тоже значение. Вот пример:

```
public void Assign()
{double x,y,z,w =1, u =7, v= 5;
  x = y = z = w =(u+v+w)/(u-v-w);}
```

Правильно построенное выражение присваивания состоит из левой и правой части. Левая часть - это список переменных, в котором знак равенства выступает в качестве разделителя. Правая часть - это выражение. Выражение правой части вычисляется, при необходимости приводится к типу переменных левой части, после чего все переменные левой части получают значение вычисленного выражения. Последние действия можно рассматривать как побочный эффект операции присваивания. все переменные в списке левой части должны иметь один тип или неявно приводиться к одному типу. Операция присваивания выполняется справа налево, поэтому вначале значение выражения получит самая правая переменная списка левой части, при этом значение самого выражения не меняется. Затем значение получает следующая справа по списку переменная - и так до тех пор, пока не будет достигнут конец списка. Так что реально можно говорить об одновременном присваивании, в котором все переменные списка получают одно и то же значение.

Специальные случаи присваивания

В языке C++ для двух частных случаев присваивания предложен отдельный синтаксис. Язык C# наследовал эти полезные свойства. Для присваиваний вида " x=x+1 ", в которых переменная увеличивается или уменьшается на единицу, используются специальные префиксные и постфиксные операции " ++ " и " -- ". Другой важный частный случай - это присваивания вида:

X = X <operator> (expression) краткая форма записи: X <operator>= expression

В качестве операции разрешается использовать арифметические, логические (побитовые) операции и операции сдвига языка C#. Семантика такого присваивания

```
x += u+v;
```

Определенное присваивание

Присваивание в языке C# называется определенным присваиванием (definite assignment). Это означает, что все используемые в выражениях переменные должны быть ранее инициализированы и иметь определенные значения.

```
//определенное присваивание
int an =0 ; //переменные должны быть инициализированы
for (int i= 0;i<5;i++)
{an =i+1;}
x+=an; z+=an; y = an;
string[] ars = new string[3];
double[] ard = new double[3];
for (int i= 0;i<3;i++)
{//массивы могут быть без инициализации
  ard[i] += i+1;
  ars[i] += i.ToString()+1;
  Console.WriteLine("ard[" +i + "]=" +ard[i] +
        ";  ars[" +i + "]=" +ars[i]); }
```

Будет ли семантика *значимой или ссылочной* - определяется типом левой части присваивания. *Переменные значимых типов* являются единоличными владельцами памяти, в которой хранятся их значения. При значимом присваивании память для хранения значений остается той же - меняются лишь сами значения, хранимые в ней. *Переменные ссылочных типов* (объекты) являются ссылками на реальные объекты динамической памяти. Ссылки могут разделять одну и ту же область памяти - ссылаться на один и тот же объект. Ссылочное присваивание - это операция над ссылками. В результате ссылочного присваивания ссылка начинает указывать на другой объект.

```
int x=3, y=5;
object obj1, obj2;
```

Здесь объявлены четыре сущности: две переменные значимого типа и две - объектного. Значимые переменные x и y проинициализированы и имеют значения, объектные переменные obj1 и obj2 являются пустыми ссылками со значением void.

```
obj1 = x; obj2 = y;
```

Эти присваивания ссылочные, поэтому правая часть приводится к ссылочному типу. В результате неявного преобразования - операции boxing - в динамической памяти создаются два объекта, обертывающие соответственно значения переменных x и y. Сущности obj1 и obj2 получают значения ссылок на эти объекты.

15. КЛАССЫ MATH, RANDOM И ВСТРОЕННЫЕ ФУНКЦИИ

Кроме переменных и констант, первичным материалом для построения выражений являются функции. Большинство их в проекте будут созданы самим программистом, но не обойтись и без встроенных функций. рассмотрим класс Math, содержащий стандартные математические функции, без которых трудно обойтись при построении многих выражений. Этот класс содержит два статических поля, задающих константы E и PI, а также 23 статических метода. Методы задают:

тригонометрические функции - Sin, Cos, Tan ;

обратные тригонометрические функции - ASin, ACos, ATan, ATan2 (sinx, cosx) ;

гиперболические функции - Tanh, Sinh, Cosh ;

экспоненту и логарифмические функции - Exp, Log, Log10 ;

модуль, корень, знак - Abs, Sqrt, Sign ;

функции округления - Ceiling, Floor, Round ;

минимум, максимум, степень, остаток - Min, Max, Pow, IEEERemainder.

Пример: …. x=a*Math.sin(x);

Y=a*Math.Log(b*t);….

Класс Random и его функции

Класс Random содержит все необходимые для этого средства. Класс Random имеет конструктор класса: для того, чтобы вызывать методы класса, нужно вначале создавать экземпляр класса. класс Random является наследником класса Object, а, следовательно, имеет в своем составе и методы родителя.

Рассмотрим конструктор класса. Он перегружен и имеет две реализации. Одна из них позволяет генерировать неповторяющиеся при каждом запуске серии случайных чисел. Начальный элемент такой серии строится на основе текущей даты и времени, что гарантирует уникальность. Конструктор вызывается из параметров public Random(). Другой конструктор с параметром - public Random (int) обеспечивает важную возможность генерирования повторяющейся серии случайных чисел. Параметр конструктора используется для построения начального элемента серии, поэтому при задании одного и того же значения параметра серия будет повторяться.

Перегруженный метод public int Next() при каждом вызове возвращает положительное целое, равномерно распределенное в некотором диапазоне. Диапазон задается параметрами метода. Три реализации метода отличаются набором параметров:

public int Next () - метод без параметров выдает целые положительные числа во всем положительном диапазоне типа int ;

public int Next (int max) - выдает целые положительные числа в диапазоне [0,max] ;

public int Next (int min, int max) - выдает целые числа в диапазоне [min,max].

Метод public double NextDouble () имеет одну реализацию. При каждом вызове этого метода выдается новое случайное число, равномерно распределенное в интервале [0,1).

16. ОПЕРАТОРЫ ЯЗЫКА C#. ОПЕРАТОР ПРИСАИВАНИЯ. СОСТАВНОЙ ОПЕРАТОР. ПУСТОЙ ОПЕРАТОР

Состав операторов языка C#, их синтаксис и семантика унаследованы от языка C++.

Оператор присваивания

Как в языке C++, так и в C# присваивание формально считается операцией. Вместе с тем запись: X = expr; следует считать настоящим оператором присваивания, так же, как и одновременное присваивание со списком переменных в левой части: X1 = X2 = ... = Xk = expr;

Блок или составной оператор

С помощью фигурных скобок несколько операторов языка (возможно, перемежаемых объявлениями) можно объединить в единую синтаксическую конструкцию, называемую блоком или составным оператором.

Синтаксически блок воспринимается как единичный оператор и может использоваться всюду в конструкциях, где синтаксис требует одного оператора. Тело цикла, ветви оператора if, как правило, представляются блоком.

Пустой оператор - это "пусто", завершаемое точкой с запятой. Иногда полезно рассматривать отсутствие операторов как существующий пустой оператор. Синтаксически допустимо ставить лишние точки с запятой, полагая, что вставляются пустые операторы. Например, синтаксически допустима следующая конструкция:

```
for (int j=1; j<5; j++)
                {;;;};
```

Она может рассматриваться как задержка по времени, работа на холостом ходе.

17. ОПЕРАТОРЫ ВЫБОРА. IF-ОПЕРАТОР. SWITCH-ОПЕРАТОР. ОПЕРАТОРЫ ПЕРЕХОДА. ОПЕРАТОР GOTO

Операторы выбора

Как в C++ и других языках программирования, в языке C# для выбора одной из нескольких возможностей используются две конструкции - if и switch. Первую из них обычно называют альтернативным выбором, вторую - разбором случаев.

Оператор if

Синтаксис оператора if:

if(выражение_1) оператор_1
else if(выражение_2) оператор_2
...
else if(выражение_K) оператор_K
else оператор_N

Выражения if должны заключаться в круглые скобки и быть булевого типа. Точнее, выражения должны давать значения true или false.

Ветви else и if, позволяющие организовать выбор из многих возможностей, могут отсутствовать. Может быть опущена и заключительная else -ветвь. В этом случае краткая форма оператора if задает альтернативный выбор - делать или не делать - выполнять или не выполнять then -оператор.

Семантика оператора if проста и понятна. Выражения if проверяются в порядке их написания. Как только получено значение true, проверка прекращается и выполняется оператор (это может быть блок), который следует за выражением, получившим значение true. С завершением этого оператора завершается и оператор if. Ветвь else, если она есть, относится к ближайшему открытому if.

Оператор switch

Конструкция:

switch(выражение)
{case константное_выражение_1: [операторы_1 оператор_перехода_1]
...
case константное_выражение_K: [операторы_K оператор_перехода_K]
[default: операторы_N оператор_перехода_N]}

Ветвь default может отсутствовать. Заметьте, по синтаксису допустимо, чтобы после двоеточия следовала пустая последовательность операторов, а не последовательность, заканчивающаяся оператором перехода. Константные выражения в case должны иметь тот же тип, что и switch -выражение.

Вначале вычисляется значение switch -выражения. Затем оно поочередно в порядке следования case сравнивается на совпадение с константными выражениями. Как только достигнуто совпадение, выполняется соответствующая последовательность операторов case - ветви. Поскольку последний оператор этой последовательности является оператором перехода, то обычно он завершает выполнение оператора switch.

Операторы перехода

Операторов перехода, позволяющих прервать естественный порядок выполнения операторов блока, в языке C# имеется несколько. К операторам перехода относятся операторы: goto, break, continue, return.

Оператор goto

Оператор goto имеет простой синтаксис и семантику:

goto [метка|case константное_выражение|default];

Все операторы языка C# могут иметь метку - уникальный идентификатор, предшествующий оператору и отделенный от него символом двоеточия. Передача управления помеченному оператору - это классическое использование оператора goto.

18. ОПЕРАТОРЫ BREAK, CONTINUE. ОПЕРАТОРЫ ЦИКЛА. FOR-ОПЕРАТОР. ЦИКЛЫ WHILE. ЦИКЛ FOREACH

Операторы break и continue

В структурном программировании признаются полезными "переходы вперед" (но не назад), позволяющие при выполнении некоторого условия выйти из цикла, из оператора выбора, из блока. Для этой цели можно использовать оператор goto, но лучше применять специально предназначенные для этих целей операторы break и continue.

Оператор break может стоять в теле цикла или завершать case - ветвь в операторе switch. В теле цикла, чаще всего, оператор break помещается в одну из ветвей оператора if, проверяющего условие преждевременного завершения цикла.

Оператор continue используется только в теле цикла. В отличие от оператора break, завершающего внутренний цикл, continue осуществляет переход к следующей итерации этого цикла.

Еще одним оператором, относящимся к группе операторов перехода, является оператор return, позволяющий завершить выполнение процедуры или функции. Его синтаксис:

return [выражение];

Для функций его присутствие и аргумент обязательны, поскольку выражение в операторе return задает значение, возвращаемое функцией.

Операторы цикла

Оператор for

Наследованный от C++ весьма удобный оператор цикла for обобщает известную конструкцию цикла типа арифметической прогрессии. Его синтаксис:

for(инициализаторы; условие; список_выражений) оператор

Оператор задает тело цикла. Инициализаторы задают начальное значение одной или нескольких переменных, часто называемых счетчиками или просто переменными цикла. В большинстве случаев цикл for имеет один счетчик. Список выражений, записанный через запятую, показывает, как меняются счетчики цикла на каждом шаге выполнения. Если условие цикла истинно, то выполняется тело цикла, затем изменяются значения счетчиков и снова проверяется условие. Как только условие становится ложным, цикл завершает свою работу. В нормальной ситуации тело цикла выполняется конечное число раз. Счетчики цикла зачастую объявляются непосредственно в инициализаторе и соответственно являются переменными, локализованными в цикле, так что после завершения цикла они перестают существовать.

Циклы While

Цикл while (выражение) является универсальным видом цикла, включаемым во все языки программирования. Тело цикла выполняется до тех пор, пока остается истинным выражение while. В языке C# у этого вида цикла

две модификации - с проверкой условия в начале и в конце цикла. Первая модификация имеет следующий синтаксис:

while(выражение) оператор

Цикл, проверяющий условие завершения в конце, соответствует стратегии: "сначала делай, а потом проверь". Тело такого цикла выполняется, по меньшей мере, один раз. Вот синтаксис этой модификации:

```
do
  оператор
while(выражение);
```

Цикл foreach

Не унаследованным от C++, является цикл foreach, удобный при работе с массивами, коллекциями и другими подобными контейнерами данных.

foreach(тип идентификатор in контейнер) оператор

Цикл работает в полном соответствии со своим названием - тело цикла выполняется для каждого элемента в контейнере. Тип идентификатора должен быть согласован с типом элементов, хранящихся в контейнере данных. Предполагается также, что элементы контейнера (массива, коллекции) упорядочены. На каждом шаге цикла идентификатор, задающий текущий элемент контейнера, получает значение очередного элемента в соответствии с порядком, установленным на элементах контейнера. С этим текущим элементом и выполняется тело цикла - выполняется столько раз, сколько элементов находится в контейнере. Цикл заканчивается, когда полностью перебраны все элементы контейнера.

Недостаток циклов foreach: цикл работает только на чтение, но не на запись элементов. Так что наполнять контейнер элементами приходится с помощью других операторов цикла.

19. ПРОЦЕДУРЫ И ФУНКЦИИ – ДВЕ ФОРМЫ ФУНКЦИОНАЛЬНОГО МОДУЛЯ. ЧЕМ ОТЛИЧАЮТСЯ ЭТИ ФОРМЫ? ПРОЦЕДУРЫ И ФУНКЦИИ – ЭТО МЕТОДЫ КЛАССА.

Процедуры и функции - функциональные модули

Первыми формами модульности, появившимися в языках программирования, были процедуры и функции. Они позволяли задавать определенную функциональность и многократно выполнять один и тот же параметризованный программный код при различных значениях параметров. Уже с первых шагов процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, - задачу повторного использования программного кода. Встроенные в язык функции давали возможность существенно расширить возможности языка программирования. (Процедура void)

Процедуры и функции - методы класса

Долгое время процедуры и функции играли не только функциональную, но и архитектурную роль. Весьма популярным при построении программных систем был метод функциональной декомпозиции "сверху вниз", и сегодня еще играющий важную роль. Но с появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ООП-языков, к которым относится и язык C#, в роли архитектурного модуля выступает класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеет содержательную начинку, задавая некоторую абстракцию данных.

Процедуры и функции связываются теперь с классом, они обеспечивают функциональность данных класса и называются методами класса. Главную роль в программной системе играют данные, а функции лишь служат данным.

В языке C# нет специальных ключевых слов - procedure и function, но присутствуют сами эти понятия. Синтаксис объявления метода позволяет однозначно определить, чем является метод - процедурой или функцией.

Прежнюю роль библиотек процедур и функций теперь играют библиотеки классов. Библиотека классов FCL, доступная в языке C#, существенно расширяет возможности языка. Знание классов этой библиотеки и методов этих классов совершенно необходимо для практического программирования на C#.

Отличия:

Функция отличается от процедуры двумя особенностями:

- всегда вычисляет некоторое значение, возвращаемое в качестве результата функции ;

- вызывается в выражениях.

Процедура C# имеет свои особенности:

- возвращает формальный результат void, указывающий на отсутствие результата ;

- вызов процедуры является оператором языка;

- имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

20. ОПИСАНИЕ МЕТОДОВ (ПРОЦЕДУР И ФУНКЦИЙ). СИНТАКСИС. АТРИБУТЫ ДОСТУПА. СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ МЕТОДЫ. ФОРМАЛЬНЫЕ АРГУМЕНТЫ. СТАТУС АРГУМЕНТОВ. ТЕЛО МЕТОДОВ.

Описание методов (процедур и функций). Синтаксис

Синтаксически в описании метода различают две части - описание заголовка и описание тела метода.

Рассмотрим синтаксис заголовка метода:

[атрибуты][модификаторы]{void| тип_результата_функции}

имя_метода([список_формальных_аргументов])

Имя метода и список формальных аргументов составляют сигнатуру метода. Заметьте, в сигнатуру не входят имена формальных аргументов - здесь важны типы аргументов. В сигнатуру не входит и тип возвращаемого результата.

Квадратные скобки (метасимволы синтаксической формулы) показывают, что атрибуты и модификаторы могут быть опущены при описании метода.

У модификатора доступа 4 возможных значения в том числе public и private. *Модификатор public* показывает, что метод открыт и доступен для вызова клиентами и потомками класса. *Модификатор private* говорит, что метод предназначен для внутреннего использования в классе и доступен для вызова только в теле методов самого класса. Заметьте, если модификатор доступа опущен, то по умолчанию предполагается, что он имеет значение private и метод является закрытым для клиентов и потомков класса.

Обязательным при описании заголовка является указание типа результата, имени метода и круглых скобок, наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует. Формально тип результата метода указывается всегда, но значение void однозначно определяет, что метод реализуется процедурой. Тип результата, отличный от void, указывает на функцию. Вот несколько простейших примеров описания методов:

```
void A() {...};
int B(){...};
public void C(){...};
```

Методы A и B являются закрытыми, а метод C - открыт. Методы A и C реализованы процедурами, а метод B - функцией, возвращающей целое значение.

Статические и динамические методы

Статические методы, объявленные с модификатором static. Такие методы не используют информацию о свойствах конкретных объектов класса - они обрабатывают общую для класса информацию, хранящуюся в его статических полях. Например, в классе Person может быть статический метод, обрабатывающий данные из статического поля message. Другим частым случаем применения статических методов является ситуация, когда класс предоставляет свои сервисы объектам других классов. Таковым является класс Math из библиотеки FCL, который не имеет собственных полей - все его статические методы работают с объектами арифметических классов.

Список формальных аргументов

Как уже отмечалось, список формальных аргументов метода может быть пустым, и это довольно типичная ситуация для методов класса. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Обязательным является указание типа и имени аргумента. Заметьте, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, классом, структурой, интерфейсом, перечислением.

Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему произвольное число фактических аргументов. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово ***params***. Оно задается один раз и указывается только для последнего аргумента списка, объявляемого как массив произвольного типа. При вызове метода этому формальному аргументу соответствует произвольное число фактических аргументов.

Содержательно, все аргументы метода разделяются на три группы: ***входные, выходные и обновляемые***. Аргументы первой группы передают информацию методу, их значения в теле метода только читаются. Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода. Аргументы третьей группы выполняют обе функции.

Их значения используются в ходе вычислений и обновляются в результате работы метода. Выходные аргументы всегда должны сопровождаться ключевым словом out, обновляемые - ref. Что же касается входных аргументов, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром ref. если аргумент объявлен как выходной с ключевым словом out, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

Тело метода

Синтаксически тело метода является блоком, который представляет собой последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор перехода, возвращающий значение функции в форме return (выражение).

Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.

21. Вызов процедур и функций. Фактические аргументы. Семантика вызова. Поля класса и аргументы метода? Поля класса или функции без аргументов? Проектирование класса Account.

Вызов метода. Синтаксис

Метод может вызываться в выражениях или быть вызван как оператор. В качестве оператора может использоваться любой метод - как *процедура*, так и *функция*.

Сам *вызов метода*, независимо от того, *процедура* это или *функция*, имеет один и тот же синтаксис:

имя_метода([список_фактических_аргументов])

Если это оператор, то вызов завершается точкой с запятой. Формальный аргумент, задаваемый при описании метода - это всегда имя аргумента (идентификатор). Фактический аргумент - это выражение, значительно более сложная синтаксическая конструкция. Вот точный синтаксис фактического аргумента:

[ref|out]выражение

О соответствии списков формальных и фактических аргументов

Между списком формальных и списком фактических аргументов должно выполняться определенное *соответствие* по числу, порядку следования, типу и статусу аргументов. Если в первом списке n формальных аргументов, то фактических аргументов должно быть не меньше n (*соответствие* по числу). Каждому i -му формальному аргументу (для всех i от 1 до n-1) ставится в *соответствие* i -й фактический аргумент. Последнему формальному аргументу, при условии, что он объявлен с ключевым словом *params*, ставятся в *соответствие* все оставшиеся фактические аргументы (*соответствие* по порядку). Если формальный аргумент объявлен с ключевым словом *ref* или *out,* то фактический аргумент должен сопровождаться таким же ключевым словом в точке вызова (*соответствие* по статусу).

Если формальный аргумент объявлен с типом T, то выражение, задающее фактический аргумент, должно быть согласовано по типу с типом T: допускает преобразование к типу T, совпадает с типом T или является его потомком (*соответствие* по типу).

Если формальный аргумент является *выходным* - объявлен с ключевым словом *ref* или *out,* - то соответствующий фактический аргумент не может быть выражением, поскольку используется в левой части оператора присваивания; следовательно, он должен быть именем, которому можно присвоить значение.

Вызов метода. Семантика

В момент *вызова метода* выполнение начинается с вычисления фактических аргументов, которые, как мы знаем, являются выражениями. Вычисление этих выражений может приводить, в свою очередь, к вызову других методов, таким образом можно полагать, что в точке вызова создается *блок*, соответствующий телу метода. В этом *блоке* происходит

замена имен формальных аргументов фактическими аргументами. Для *выходных аргументов*, для которых фактические аргументы также являются именами, эта замена или передача аргументов осуществляется по ссылке, то есть заменяет формальный аргумент ссылкой на реально существующий объект, заданный фактическим аргументом.

Чуть более сложную семантику имеет *вызов по значению*, применяемый к формальным аргументам, которые объявлены без ключевых слов *ref* и *out*.

При вычислении выражений, заданных такими фактическими аргументами, их значения присваиваются специально создаваемым переменным, локализованным в теле исполняемого *блока*. Имена этих локализованных переменных и подставляются вместо имен формальных аргументов. Таким образом, семантика вызова по значению заключается в том, что, если вы забыли указать ключевое слово *ref* или *out* для аргумента, фактически являющегося *выходным*, то к нему будет применяться *вызов по значению*. Даже если в теле метода происходит изменение значения этого аргумента, то оно действует только на время выполнения тела метода. Как только метод заканчивает свою работу (завершается *блок*), все локальные переменные (в том числе, созданные для замены формальных аргументов) оканчивают свое существование, так что изменения не затронут фактических аргументов и они сохранят свои значения, бывшие у них до вызова. Отсюда вывод: все *выходные аргументы*, значения которых предполагается изменить в процессе работы, должны иметь ключевое слово *ref* или *out*. Еще один важный вывод: ключевым словом *ref* полезно иногда снабжать и *входные аргументы*.

Говоря о семантике *вызова по ссылке* и по значению, следует сделать одно важное уточнение. В объектном программировании основную роль играют ссылочные типы - мы работаем с *классами* и объектами. Когда методу передается объект ссылочного типа, то все поля этого объекта могут меняться в. Сама ссылка на объект, как и положено, остается неизменной, но состояние объекта, его поля могут полностью обновиться.

Почему у методов мало аргументов?

Методы *класса* имеют значительно меньше аргументов, чем *процедуры* и *функции* ООП. Это связано с тем, что методы *класса* - это не просто *процедуры*; это *процедуры*, обслуживающие данные. Все поля доступны любому методу по определению. И работа идет не с *классом*, а с объектами - экземплярами *класса*. Из полей соответствующего объекта - цели вызова - извлекается информация, нужная методу в момент вызова, а работа метода чаще всего сводится к обновлению значений полей этого объекта. Очевидно, что методу не нужно через *входные аргументы* передавать информацию, содержащуюся в полях.

Поля класса или функции без аргументов?

Поля хранят информацию о состоянии объектов *класса*. Состояние объекта динамически изменяется в ходе вычислений - обновляются значения полей. Часто возникающая дилемма при проектировании *класса*: что лучше - создать ли поле, хранящее информацию, или создать *функцию* без аргументов, вычисляющую значение этого поля всякий раз, когда это значение понадобится. Если предпочесть поле, то это приводит к дополнительным расходам памяти. Они могут быть значительными, когда создается большое число объектов. Если предпочесть *функцию*, то это потребует временных затрат на вычисление значения, и затраты могут быть значительными в сравнении с выбором текущего значения поля.

Пример: две версии класса Account

Проиллюстрируем рассмотренные выше вопросы на примере проектирования *классов* Account и Account1, описывающих такую абстракцию данных, как банковский счет. Определим на этих данных две основные операции - занесение денег на счет и снятие денег. В первом варианте - *классе* Account - будем активно использовать поля *класса*. Помимо двух основных полей credit и debit, хранящих приход и расход счета, введем поле balance, которое задает текущее состояние счета, и два поля, связанных с последней выполняемой операцией. Поле sum будет хранить сумму денег текущей операции, а поле result - результат выполнения операции.

```
public class Account
{
    //закрытые поля класса
    int debit=0, credit=0, balance =0;
    int sum =0, result=0;
    /// Зачисление на счет с проверкой
    /// зачисляемая сумма
    public void putMoney(int sum)
    {
            this.sum = sum;
            if (sum >0)
            {
                    credit += sum; balance = credit - debit; result =1;
            }
            else result = -1;
            Mes();
    }//завершается putMoney
    /// Снятие со счета с проверкой
    /// снимаемая сумма
    public void getMoney(int sum)
    {
            this.sum = sum;
            if(sum <= balance)
            {
```

```
                debit += sum; balance = credit - debit; result =2;
            }
            else result = -2;
            Mes();
    }// завершается getMoney
    /// <summary>
    /// Уведомление о выполнении операции
    void Mes()
    {
            switch (result)
            {
                    case 1:
         Console.WriteLine("Операция зачисления денег    прошла успешно!");
        Console.WriteLine("Сумма={0}, Ваш текущий баланс={1}",sum, balance);
            break;
                    case 2:
            Console.WriteLine("Операция снятия денег прошла успешно!");
        Console.WriteLine("Сумма={0}, Ваш текущий баланс={1}", sum,balance);
                        break;
                    case -1:
                  Console.WriteLine("Операция зачисления денег     не выполнена!");
    Console.WriteLine("Сумма должна быть больше нуля!");
        Console.WriteLine("Сумма={0}, Ваш текущий баланс={1}", sum,balance);
                        break;
                    case -2:
    Console.WriteLine("Операция снятия денег не выполнена!");
    Console.WriteLine("Сумма должна быть не больше баланса!");
        Console.WriteLine("Сумма={0}, Ваш текущий баланс={1}", sum,balance);
                        break;
                    default:
                    Console.WriteLine("Неизвестная операция!");
                        break;
            }       }       }
```

Как можно видеть, только у методов getMoney и putMoney имеется один *входной аргумент*. Это тот аргумент, который нужен по сути дела, поскольку только клиент может решить, какую сумму он хочет снять или положить на счет. Других аргументов у методов *класса* нет - вся информация передается через поля *класса*. Уменьшение числа аргументов приводит к повышению эффективности работы с методами, так как исчезают затраты на передачу фактических аргументов. Однако, это влечет за собой накладные расходы при выполнении кода. В данном случае, усложняются сами операции работы со вкладом, поскольку нужно в момент выполнения операции обновлять значения многих полей *класса*.

Закрытый метод Mes вызывается после выполнения каждой операции, сообщая о том, как прошла операция, и информируя клиента о текущем состоянии его баланса.

22. Корректность метода. Спецификация

Корректность методов

Корректность метода - это не внутреннее понятие, подлежащее определению в терминах самого метода. *Корректность* определяется по отношению к внешним спецификациям метода. Если нет спецификаций, то говорить о *корректности* "некорректно".

Спецификации можно задавать по-разному. Мы определим их здесь через понятия предусловий и постусловий метода (используя символику триад Хоара, введенных Чарльзом Энтони Хоаром - выдающимся программистом и ученым).

Пусть P(x,z) - *программа* P с входными аргументами x и выходными z. Пусть Q(y) - некоторое логическое условие (*предикат*) над переменными программы y. Язык для записи предикатов Q(y) формализовывать не будем. Отметим только, что он может быть шире языка, на котором записываются условия в программах, и включать, например, кванторы. Предусловием программы P(x,z)будем называть *предикат* Pre(x), заданный на входах программы. Постусловием программы P(x,z) будем называть *предикат* Post(x,z), связывающий входы и выходы программы. Для простоты будем полагать, что *программа* P не изменяет своих входов x в процессе своей работы.

Определение (*частичной корректности*): *Программа* P(x,z) корректна (частично, или условно) по отношению к предусловию Pre(x) и постусловию Post(x,z), если из истинности предиката Pre(x) следует, что для программы P(x,z), запущенной на входе x, гарантируется выполнение предиката Post(x,z) при условии завершения программы.

Условие *частичной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

[Pre(x)]P(x,z)[Post(x,z)]

Определение (*полной корректности*): *Программа* P(x,z) корректна (полностью, или тотально) по отношению к предусловию Pre(x) и постусловию Post(x,z), если из истинности предиката Pre(x) следует, что для программы P(x,z), запущенной на входе x, гарантируется ее завершение и выполнение предиката Post(x,z).

Условие *полной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

{Pre(x)}P(x,z){Post(x,z)}

23. Предусловие метода. Постусловие метода. Корректность метода по отношению к предусловию и постусловию.

Пусть P(x,z) - *программа* P с входными аргументами x и выходными z. Пусть Q(y) - некоторое логическое условие (*предикат*) над переменными программы y. Язык для записи предикатов Q(y) формализовывать не будем. Отметим только, что он может быть шире языка, на котором записываются условия в программах, и включать, например, кванторы. Предусловием программы P(x,z)будем называть *предикат* Pre(x), заданный на входах программы. Постусловием программы P(x,z) будем называть *предикат* Post(x,z), связывающий входы и выходы программы. Для простоты будем полагать, что *программа* P не изменяет своих входов x в процессе своей работы.

Любая *программа* корректна по отношению к предусловию, заданному тождественно ложным предикатом False. Любая завершающаяся *программа* корректна по отношению к постусловию, заданному тождественно истинным предикатом True.

Корректная *программа* говорит своим клиентам: если вы хотите вызвать меня и ждете гарантии выполнения постусловия после моего завершения, то будьте добры гарантировать выполнение предусловия на входе. Задание предусловий и постусловий методов - это такая же важная часть работы программиста, как и написание самого метода. На языке C# пред- и постусловия обычно задаются в теге <summary>, предшествующем методу, и являются частью *XML-отчета*. К сожалению, технология работы в *Visual* Studio не предусматривает возможности автоматической проверки предусловия перед вызовом метода и проверки постусловия после его завершения с *выбрасыванием исключений* в случае их невыполнения. Проверку предусловий важно оставлять и в готовой системе, поскольку истинность предусловий должен гарантировать не разработчик метода, а клиент, вызывающий метод. Клиентам же свойственно ошибаться и вызывать метод в неподходящих условиях.

24. Частичная корректность. Завершаемость. Полная коррекность.

Пусть P(x,z) - *программа* P с входными аргументами x и выходными z. Пусть Q(y) - некоторое логическое условие (*предикат*) над переменными программы y. Язык для записи предикатов Q(y) формализовывать не будем. Отметим только, что он может быть шире языка, на котором записываются условия в программах, и включать, например, кванторы. Предусловием программы P(x,z)будем называть *предикат* Pre(x), заданный на входах программы. Постусловием программы P(x,z) будем называть *предикат* Post(x,z), связывающий входы и выходы программы. Для простоты будем полагать, что *программа* P не изменяет своих входов x в процессе своей работы. Теперь несколько определений:

Определение 1 (*частичной корректности*): *Программа* P(x,z) корректна (частично, или условно) по отношению к предусловию Pre(x) и постусловию Post(x,z), если из истинности предиката Pre(x) следует, что для программы P(x,z), запущенной на входе x, гарантируется выполнение предиката Post(x,z) при условии завершения программы.

Условие *частичной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

[Pre(x)]P(x,z)[Post(x,z)]

Определение 2 (*полной корректности*): *Программа* P(x,z) корректна (полностью, или тотально) по отношению к предусловию Pre(x) и постусловию Post(x,z), если из истинности предиката Pre(x) следует, что для программы P(x,z), запущенной на входе x, гарантируется ее завершение и выполнение предиката Post(x,z).

Условие *полной корректности* записывается в виде триады Хоара, связывающей программу с ее предусловием и постусловием:

{Pre(x)}P(x,z){Post(x,z)}

Доказательство полной корректности обычно состоит из двух независимых этапов - *доказательства частичной корректности* и доказательства завершаемости программы. Заметьте, *полностью корректная программа*, которая запущена на входе, не удовлетворяющем ее предусловию, вправе зацикливаться, а также возвращать любой результат. Любая *программа* корректна по отношению к предусловию, заданному тождественно ложным предикатом False. Любая завершающаяся *программа* корректна по отношению к постусловию, заданному тождественно истинным предикатом True.

Корректная *программа* говорит своим клиентам: если вы хотите вызвать меня и ждете гарантии выполнения постусловия после моего завершения, то будьте добры гарантировать выполнение предусловия на входе. Задание предусловий и постусловий методов - это такая же важная часть работы программиста, как и написание самого метода. На языке C# пред- и постусловия обычно задаются в теге <summary>, предшествующем методу, и

являются частью *XML-отчета*. К сожалению, технология работы в *Visual* Studio не предусматривает возможности автоматической проверки предусловия перед вызовом метода и проверки постусловия после его завершения с *выбрасыванием исключений* в случае их невыполнения. Проверку предусловий важно оставлять и в готовой системе, поскольку истинность предусловий должен гарантировать не разработчик метода, а клиент, вызывающий метод. Клиентам же свойственно ошибаться и вызывать метод в неподходящих условиях.

25. Инвариант цикла. Вариант цикла. Подходящий вариант. Корректность циклов.

Инварианты и варианты цикла

Циклы, как правило, являются наиболее сложной частью метода - большинство ошибок связано именно с ними. При написании корректно работающих циклов крайне важно понимать и использовать понятия *инварианта* и *варианта цикла*. Без этих понятий не обходится и формальное *доказательство корректности* циклов. Ограничимся рассмотрением цикла в следующей форме:

Init(x,z); while(B)S(x,z);

Здесь B - условие цикла while, S - его тело, а Init - группа предшествующих операторов, задающая инициализацию цикла. Реально ни один цикл не обходится без инициализирующей части. Синтаксически было бы правильно, чтобы Init являлся бы формальной частью оператора цикла. В операторе for это частично сделано - инициализация счетчиков является частью цикла.

Определение (*инварианта цикла*): предикат Inv(x, z) называется *инвариантом* цикла while, если истинна следующая триада Хоара:

{Inv(x, z)& B}S(x,z){Inv(x,z)}

Содержательно это означает, что из истинности *инварианта* цикла до начала выполнения тела цикла и из истинности условия цикла, гарантирующего выполнение тела, следует истинность *инварианта* после выполнения тела цикла. Сколько бы раз ни выполнялось тело цикла, его *инвариант* остается истинным.

Для любого цикла можно написать сколь угодно много *инвариантов*. Любое тождественное условие (2*2 =4) является *инвариантом* любого цикла. Поэтому среди *инвариантов* выделяются так называемые подходящие *инварианты* цикла. Они называются подходящими, поскольку позволяют доказать *корректность цикла* по отношению к его пред- и постусловиям. Как доказать *корректность цикла*? Рассмотрим соответствующую триаду:

{Pre(x)} Init(x,z); while(B)S(x,z);{Post(x,z)}

Доказательство разбивается на три этапа. Вначале доказываем истинность триады:

(*) {Pre(x)} Init(x,z){RealInv(x,z)}

Содержательно это означает, что предикат RealInv становится истинным после выполнения инициализирующей части. Далее доказывается, что RealInv является *инвариантом* цикла:

(**) {RealInv(x, z)& B} S(x,z){RealInv(x,z)}

На последнем шаге доказывается, что наш *инвариант* обеспечивает решение задачи после завершения цикла:

(***) ~B & RealInv(x, z) -> Post(x,z)

Это означает, что из истинности *инварианта* и условия завершения цикла следует требуемое постусловие.

Определение (подходящего *инварианта)*: предикат RealInv, удовлетворяющий условиям (*), (**), (***), называется подходящим *инвариантом* цикла.

С циклом связано еще одно важное понятие - *варианта* цикла, используемое для доказательства завершаемости цикла.

Определение (*варианта цикла*): целочисленное неотрицательное выражение Var(x, z) называется *вариантом цикла*, если выполняется следующая триада:

{(Var(x,z)= n) & B} S(x,z){(Var(x,z)= m) & (m < n)}

Содержательно это означает, что каждое выполнение тела цикла приводит к уменьшению значения его *варианта*. После конечного числа шагов *вариант* достигает своей нижней границы, и цикл завершается. Простейшим примером *варианта* цикла является выражение n-i для цикла:

for(i=1; i<=n; i++) S(x, z);

Пользоваться *инвариантами* и *вариантами* цикла нужно не только и не столько для того, чтобы проводить формальное *доказательство корректности* циклов. Они способствуют написанию *корректных циклов*. Правило корректного программирования гласит: "При написании каждого цикла программист должен определить его подходящий *инвариант* и *вариант* ". Задание предусловий, постусловий, *вариантов* и *инвариантов* циклов является такой же частью процесса разработки корректного метода, как и написание самого кода.

26. Рекурсия. Прямая и косвенная рекурсия.

Рекурсия

Рекурсия является одним из наиболее мощных средств в арсенале программиста. *Рекурсивные структуры данных* и *рекурсивные методы* широко используются при построении программных систем. *Рекурсивные методы*, как правило, наиболее всего удобны при работе с рекурсивными структурами данных - списками, деревьями. *Рекурсивные методы* обхода деревьев служат классическим примером.

Определение (*рекурсивного метода*): метод P (процедура или *функция*) называется *рекурсивным,* если при выполнении тела метода происходит *вызов метода* P.

Рекурсия может быть *прямой*, если вызов P происходит непосредственно в теле метода P. *Рекурсия* может быть косвенной, если в теле P вызывается метод Q (эта цепочка может быть продолжена), в теле которого вызывается метод P. Определения методов P и Q взаимно рекурсивны, если в теле метода Q вызывается метод P, вызывающий, в свою *очередь*, метод Q.

Для того чтобы *рекурсия* не приводила к зацикливанию, в тело нормального *рекурсивного метода* всегда встраивается *оператор выбора*, одна из ветвей которого не содержит рекурсивных вызовов. Если в теле *рекурсивного метода рекурсивный вызов* встречается только один раз, значит, что *рекурсию* можно заменить обычным циклом, что приводит к более эффективной программе, поскольку реализация *рекурсии* требует временных затрат и работы со стековой памятью. Приведу вначале простейший пример *рекурсивного определения функции*, вычисляющей *факториал* целого числа:

```
public long factorial(int n)
{
   if (n<=1) return(1);
   else return(n*factorial(n-1));
}//factorial
```

Функция factorial является примером прямого рекурсивного определения - в ее теле она сама себя вызывает. Здесь, как и положено, есть нерекурсивная *ветвь*, завершающая вычисления, когда n становится равным единице. Это пример так называемой "хвостовой" *рекурсии*, когда в теле встречается ровно один *рекурсивный вызов*, стоящий в конце соответствующего выражения. Хвостовую *рекурсию* намного проще записать в виде обычного *цикла*. Вот циклическое *определение* той же функции:

```
public long fact(int n)
{
   long res =1;
   for(int i = 2; i <=n; i++) res*=i;
```

```
    return(res);
}//fact
```

Конечно, циклическое *определение* проще, понятнее и эффективнее, и применять *рекурсию* в подобных ситуациях не следует. Интересно сравнить время вычислений, дающее некоторое *представление* о том, насколько эффективно реализуется *рекурсия*. Вот соответствующий тест, решающий эту задачу:

```
public void TestTailRec()
{
    Hanoi han = new Hanoi(5);
    long time1, time2;
    long f=0;
    time1 = getTimeInMilliseconds();
    for(int i = 1; i <1000000; i++)f =han.fact(15);
    time2 =getTimeInMilliseconds();
    Console.WriteLine(" f= {0}, " + "Время работы
            циклической процедуры: {1}",f,time2 -time1);
    time1 = getTimeInMilliseconds();
    for(int i = 1; i <1000000; i++)f =han.factorial(15);
    time2 =getTimeInMilliseconds();
    Console.WriteLine(" f= {0}, " + "Время работы
            рекурсивной процедуры: {1}",f,time2 -time1);
}
```

Каждая из функций вызывается в цикле, работающем 1000000 раз. До начала *цикла* и после его окончания вычисляется *текущее время*. *Разность* этих времен и дает оценку времени работы функций. Обе функции вычисляют *факториал* числа 15.

27. Сложность рекурсивных алгоритмов. Задача «Ханойские башни»

Сложность рекурсивных вычислений. При относительной простоте написания, у рекурсивных подпрограмм часто встречается существенный недостаток – неэффективность. Так, сравнивая скорость вычисления чисел Фибоначчи с помощью итеративной и рекурсивной функции можно заметить, что итеративная функция выполняется почти «мгновенно», не зависимо от значения n. При использовании же рекурсивной функции уже при n=40 заметна задержка при вычислении, а при больших n результат появляется весьма нескоро.

Неэффективность рекурсии проявляется в том, что одни и те же вычисления производятся помногу раз.

Рекурсивное решение задачи "Ханойские башни"

Рассмотрим известную задачу о конце света - "Ханойские башни". Ее содержательная постановка такова. В одном из буддийских монастырей монахи уже тысячу лет занимаются перекладыванием колец. Они располагают тремя пирамидами, на которых надеты кольца разных размеров.

В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру. Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие - кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды. Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света.

Беспокоиться о близком конце света не стоит. Задача эта не под силу и современным компьютерам. Число ходов в ней равно 2^{64}, а это, как известно, большое число, и *компьютер*, работающий в сотню миллионов раз быстрее монахов, не справится с этой задачей в ближайшие тысячелетия.

Рассмотрим эту задачу в компьютерной постановке. Спроектирован *класс* Hanoi, в котором роль пирамид играют три массива, а числа играют роль колец. Вот описание данных этого класса и некоторых его методов:

```
public class Hanoi
{
    int size,moves;
    int[] tower1, tower2,tower3;
    int top1,top2,top3;
    Random rnd = new Random();
    public Hanoi(int size)
    {
            this.size = size;
            tower1 = new int[size];
            tower2 = new int[size];
            tower3 = new int[size];
            top1 = size; top2=top3=moves =0;
```

```
    }
    public void Fill()
    {
            for(int i =0; i< size; i++)
                    tower1[i]=size-i;
    }
}//Hanoi
```

Массивы *tower* играют роль ханойских башен, связанные с ними переменные top задают вершину - первую свободную ячейку при перекладывании колец (чисел). *Переменная* size задает размер массивов (число колец), а *переменная* moves используется для подсчета числа ходов. Для дальнейших экспериментов нам понадобится генерирование случайных чисел, поэтому в классе определен *объект* уже известного нам класса Random. *Конструктор класса* инициализирует *поля класса*, а метод Fill формирует начальное состояние, задавая для первой пирамиды числа, идущие в порядке убывания к ее вершине (top).

Рекурсивный *вариант* решения задачи прозрачен, хотя и напоминает некоторый род фокуса, что характерно для рекурсивного стиля мышления. *Базис рекурсии* прост. Для перекладывания одного кольца задумываться о решении не нужно - оно делается в один ход. Если есть *базисное решение*, то оставшаяся часть также очевидна. Нужно применить рекурсивно *алгоритм*, переложив n-1 кольцо с первой пирамиды на третью пирамиду. Затем сделать очевидный ход, переложив последнее самое большое кольцо с первой пирамиды на вторую. Затем снова применить *рекурсию*, переложив n-1 кольцо с третьей пирамиды на вторую пирамиду. Задача решена. Столь же проста ее *запись* на языке программирования:

```
public void HanoiTowers()
{
    HT(ref tower1,ref tower2, ref tower3,
            ref top1, ref top2, ref top3,size);
    Console.WriteLine("\nВсего ходов 2^n -1 = {0}",moves);
}
```

Как обычно в таких случаях, вначале пишется нерекурсивная процедура, вызывающая рекурсивный *вариант* с аргументами. В качестве фактических аргументов процедуре HT передаются *поля класса*, обновляемые в процессе многочисленных рекурсивных вызовов и потому снабженные ключевым словом ref. Рекурсивный *вариант* реализует описанную выше идею алгоритма:

```
/// <summary>
/// Перенос count колец с tower1 на tower2, соблюдая
/// правила и используя tower3. Свободные вершины
/// башен - top1, top2, top3
/// </summary>
void HT(ref int[] t1, ref int[] t2, ref int[] t3,
    ref int top1, ref int top2, ref int top3, int count)
```

```
{
    if (count == 1)Move(ref t1,ref t2, ref top1,ref top2);
    else
    {
            HT(ref t1,ref t3,ref t2,ref top1,ref top3, ref top2,count-1);
            Move(ref t1,ref t2,ref top1, ref top2);
            HT(ref t3,ref t2,ref t1,ref top3,ref top2, ref top1,count-1);
    }
}//HT
```

Процедура Move описывает очередной ход. Ее аргументы однозначно задают, с какой и на какую пирамиду нужно перенести кольцо. Никаких сложностей в ее реализации нет:

```
void Move(ref int[]t1, ref int[] t2, ref int top1, ref int top2)
{
    t2[top2] = t1[top1-1];
    top1--; top2++; moves++;
    //PrintTowers();
}//Move
```

Метод PrintTowers позволяет проследить за ходом переноса. Приведу еще *метод класса* Testing, тестирующий работу по переносу колец:

```
public void TestHanoiTowers()
{
    Hanoi han = new Hanoi(10);
    Console.WriteLine("Ханойские башни");
    han.Fill();
    han.PrintTowers();
    han.HanoiTowers();
    han.PrintTowers();
}
```

На рис. 10.2 показаны результаты работы с включенной печатью каждого хода для случая переноса трех колец.

Рис. 10.2. "Ханойские башни"

28. Общий взгляд на массивы. Сравнение с массивами C++

Общий взгляд

Массив задает способ организации данных. *Массивом* называют упорядоченную совокупность элементов одного типа. Каждый элемент *массива* имеет индексы, определяющие порядок элементов. Число индексов характеризует *размерность массива*. Каждый *индекс* изменяется в некотором диапазоне [a,b]. В языке C#, как и во многих других языках, индексы задаются целочисленным типом. При объявлении *массива* границы задаются выражениями. Если все границы заданы константными выражениями, то число элементов *массива* известно в момент его объявления и ему может быть выделена *память* еще на этапе трансляции. Такие *массивы* называются *статическими*. Если же выражения, задающие границы, зависят от переменных, то такие *массивы* называются *динамическими*, поскольку *память* им может быть отведена только динамически в процессе выполнения программы, когда становятся известными значения соответствующих переменных.

В языке C++ все *массивы* являются *статическими*; более того, все *массивы* являются *0-базируемыми*. Это означает, что *нижняя граница* всех индексов *массива* фиксирована и равна нулю. Введение такого ограничения имеет свою логику, поскольку здесь широко используется адресная арифметика.

В языке C# снято существенное ограничение языка C++ на статичность *массивов*. *Массивы* в языке C# являются настоящими *динамическими массивами*. Как следствие этого, напомню, *массивы* относятся к ссылочным типам, *память* им отводится динамически в "куче". К сожалению, не снято ограничение *0-базируемости*, хотя, в таком ограничении уже нет логики из-за отсутствия в C# адресной арифметики. Было бы гораздо удобнее во многих задачах иметь возможность работать с *массивами*, у которых *нижняя граница* не равна нулю.

В языке C++ "классических" *многомерных массивов* нет. Здесь введены одномерные *массивы* и *массивы массивов*. Последние являются более общей структурой данных и позволяют задать не только многомерный куб, но и изрезанную, ступенчатую структуру. Однако использование *массива массивов* менее удобно.

В языке C#, соблюдая преемственность, сохранены одномерные *массивы* и *массивы массивов*. В *дополнение* к ним в язык добавлены *многомерные массивы*. Динамические *многомерные массивы* языка C# являются весьма мощной, надежной, понятной и удобной структурой данных, которую смело можно рекомендовать к применению не только профессионалам, но и новичкам, программирующим на C#.

29. Почему массивы C# лучше, чем массивы C++. Виды массивов – одномерные, многомерные и изрезанные

В языке C++ "классических" многомерных массивов нет. Здесь введены одномерные массивы и массивы массивов. Последние являются более общей структурой данных и позволяют задать не только многомерный куб, но и изрезанную, ступенчатую структуру. Однако использование массива массивов менее удобно.

В языке C#, соблюдая преемственность, сохранены одномерные массивы и массивы массивов. В дополнение к ним в язык добавлены многомерные массивы. Динамические многомерные массивы языка C# являются весьма мощной, надежной, понятной и удобной структурой данных, которую смело можно рекомендовать к применению не только профессионалам, но и новичкам, программирующим на C#.

Объявление массивов

Объявление одномерных массивов

Напомню общую структуру объявления:

[<атрибуты>] [<модификаторы>] <тип> []<объявители>;

Объявление одномерного массива выглядит следующим образом:

<тип>[] <объявители>;

Заметьте, в отличие от языка C++ квадратные скобки приписаны не к имени переменной, а к типу. Они являются неотъемлемой частью определения класса, так что запись T[] следует понимать как класс одномерный массив с элементами типа T.

Что же касается границ изменения индексов, то эта характеристика к классу не относится, она является характеристикой переменных - экземпляров, каждый из которых является одномерным массивом со своим числом элементов, задаваемых в объявителе переменной.

Как и в случае объявления простых переменных, каждый объявитель может быть именем или именем с инициализацией. В первом случае речь идет об отложенной инициализации. Нужно понимать, что при объявлении с отложенной инициализацией сам массив не формируется, а создается только ссылка на массив, имеющая неопределенное значение Null. Поэтому пока массив не будет реально создан и его элементы инициализированы, использовать его в вычислениях нельзя. Вот пример объявления трех массивов с отложенной инициализацией:

```
int[] a, b, c;
```

Чаще всего при объявлении массива используется имя с инициализацией. И опять-таки, как и в случае простых переменных, могут быть два варианта инициализации. В первом случае инициализация является явной и задается константным массивом. Вот пример:

```
double[] x= {5.5, 6.6, 7.7};
```

Следуя синтаксису, элементы константного массива следует заключать в фигурные скобки.

Во втором случае создание и инициализация массива выполняется в объектном стиле с вызовом конструктора массива. И это наиболее распространенная практика объявления массивов. Приведу пример:

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

Итак, если массив объявляется без инициализации, то создается только висячая ссылка со значением Null. Если инициализация выполняется конструктором, то в динамической памяти создается сам массив, элементы которого инициализируются константами соответствующего типа, и ссылка связывается с этим массивом. Если массив инициализируется константным массивом, то в памяти создается константный массив, с которым и связывается ссылка.

Как обычно задаются элементы массива, если они не заданы при инициализации? Они либо вычисляются, либо вводятся пользователем.

```
public void TestDeclaration()
{
 //объявляются три одномерных массива A,B,C
 int[] A = new int[5], B= new int[5], C= new int[5];
 Arrs.CreateOneDimAr(A);
 Arrs.CreateOneDimAr(B);
 for(int i = 0; i<5; i++)
        C[i] = A[i] + B[i];
 //объявление массива с явной инициализацией
 int[] x ={5,5,6,6,7,7};
 //объявление массивов с отложенной инициализацией
 int[] u,v;
 u = new int[3];
 for(int i=0; i<3; i++) u[i] =i+1;
 //v= {1,2,3}; //присваивание константного массива
 //недопустимо
 v = new int[4];
 v=u; //допустимое присваивание
 int [,] w = new int[3,5];
 //v=w; //недопустимое присваивание: объекты разных классов
 Arrs.PrintAr1("A", A); Arrs.PrintAr1("B", B);
 Arrs.PrintAr1("C", C); Arrs.PrintAr1("X", x);
 Arrs.PrintAr1("U", u); Arrs.PrintAr1("V", v);
}
```

В процедуре показаны разные способы объявления массивов. Вначале объявляются одномерные массивы A, B и C, создаваемые конструктором. Значения элементов этих трех массивов имеют один и тот же тип int. То, что они имеют одинаковое число элементов, произошло по воле программиста, а не диктовалось требованиями языка.

Массив x объявлен с явной инициализацией. Число и значения его элементов определяется константным массивом.

Массивы u и v объявлены с отложенной инициализацией. В последующих операторах массив u инициализируется в объектном стиле - элементы получают его в цикле значения.

Для поддержки работы с массивами создан специальный класс Arrs, статические методы которого выполняют различные операции над массивами.

Многомерные массивы

Одномерные массивы -это частный случай многомерных. Одномерные массивы позволяют задавать такие математические структуры как векторы, двумерные - матрицы, трехмерные - кубы данных, массивы большей размерности - многомерные кубы данных. Заметим, что при работе с базами данных многомерные кубы, так называемые кубы OLAP, встречаются сплошь и рядом.

Для объявления многомерного массива указывается его размерность с использованием запятых. Вот как выглядит объявление многомерного массива в общем случае:

<тип>[, ... ,] <объявители>;

Число запятых, увеличенное на единицу, и задает размерность массива. Что касается объявителей, то все, что сказано для одномерных массивов, справедливо и для многомерных. Можно лишь отметить, что, хотя явная инициализация с использованием многомерных константных массивов возможна, но применяется редко из-за громоздкости такой структуры. Проще инициализацию реализовать программно, но иногда она все же применяется. Вот пример:

```
public void TestMultiArr()
{
int[,]matrix = {{1,2},{3,4}};
Arrs.PrintAr2("matrix", matrix);}
```

Массивы массивов

Еще одним видом массивов C# являются массивы массивов, называемые также изрезанными массивами (jagged arrays). Такой массив массивов можно рассматривать как одномерный массив, элементы которого являются массивами, элементы которых, в свою очередь, снова могут быть массивами, и так может продолжаться до некоторого уровня вложенности.

Подобные структуры данных применяются для представления деревьев, у которых узлы могут иметь произвольное число потомков. Таковым может быть, например, генеалогическое дерево. Вершины первого уровня - Fathers, представляющие отцов, могут задаваться одномерным массивом, так что Fathers[i] - это i -й отец. Вершины второго уровня представляются массивом массивов - Children, так что Children[i] - это массив детей i -го отца, а Children[i][j] - это j -й ребенок i -го отца. Для представления внуков понадобится третий уровень, так что GrandChildren [i][j][k] будет представлять к -го внука j -го ребенка i -го отца.

Есть некоторые особенности в объявлении и инициализации таких массивов. Если при объявлении типа многомерных массивов для указания размерности использовались запятые, то для изрезанных массивов применяется более ясная символика - совокупности пар квадратных скобок; например, int[][] задает массив, элементы которого - одномерные массивы элементов типа int.

Сложнее с созданием самих массивов и их инициализацией. Здесь нельзя вызвать конструктор new int[3][5], поскольку он не задает изрезанный массив. Фактически нужно вызывать конструктор для каждого массива на самом нижнем уровне. В этом и состоит сложность объявления таких массивов.

```
//объявление и инициализация
int[][] jagger = new int[3][]
{
  new int[] {5,7,9,11},
  new int[] {2,8},
  new int[] {6,12,4}
};
```

Массив jagger имеет всего два уровня. Можно считать, что у него три элемента, каждый из которых является массивом. Для каждого такого массива необходимо вызвать конструктор new, чтобы создать внутренний массив. В данном примере элементы внутренних массивов получают значение, будучи явно инициализированы константными массивами. Конечно, допустимо и такое объявление:

```
int[][] jagger1 = new int[3][]
{
  new int[4],
  new int[2],
  new int[3]
};
```

В этом случае элементы массива получат при инициализации нулевые значения. Реальную инициализацию нужно будет выполнять программным путем. Стоит заметить, что в конструкторе верхнего уровня константу 3 можно

опустить и писать просто new int[][]. Вызов этого конструктора можно вообще опустить - он будет подразумеваться:

```
int[][] jagger2 =
{
 new int[4],
 new int[2],
 new int[3]
};
```

А вот конструкторы нижнего уровня необходимы. Еще одно важное замечание - динамические массивы возможны и здесь.

```
int Fcount =3;
string[] Fathers = new string[Fcount];
Fathers[0] ="Николай"; Fathers[1] = "Сергей";
 Fathers[2] = "Петр";
string[][] Children = new string[Fcount][];
Children[0] = new string[] {"Ольга", "Федор"};
Children[1] = new string[]
 {"Сергей","Валентина","Ира","Дмитрий"};
Children[2] = new string[]{"Мария","Ирина","Надежда"};
myar.PrintAr3(Fathers,Children);
```

Здесь отцов описывает обычный динамический одномерный массив Fathers. Для описания детей этих отцов необходим уже массив массивов, который также является динамическим на верхнем уровне, поскольку число его элементов совпадает с числом элементов массива Fathers.

В классе Arrs для печати массива создан специальный метод PrintAr3, которому в качестве аргументов передаются массивы Fathers и Children.

```
public void PrintAr3(string [] Fathers, string[][] Children)
{
 for(int i = 0; i < Fathers.Length; i++)
 {
        Console.WriteLine("Отец : {0}; Его дети:", Fathers[i]);
        for(int j = 0; j < Children[i].Length; j++)
              Console.Write( Children[i][j] + " ");
        Console.WriteLine();
 }
}//PrintAr3
```

Комментарии:

1.Внешний цикл по i организован по числу элементов массива Fathers. Заметьте, здесь используется свойство Length, в отличие от ранее применяемого метода GetLength.

2.В этом цикле с тем же успехом можно было бы использовать и имя массива Children. Свойство Length для него возвращает число элементов верхнего уровня, совпадающее, как уже говорилось, с числом элементов массива Fathers.

3.Во внутреннем цикле свойство Length вызывается для каждого элемента Children[i], который является массивом.

30. Динамические массивы

Динамические массивы

Во всех вышеприведенных примерах объявлялись *статические массивы*, поскольку *нижняя граница* равна нулю по определению, а *верхняя* всегда задавалась в этих примерах константой. Чисто синтаксически нет существенной разницы в объявлении *статических* и *динамических массивов*. Выражение, задающее границу изменения индексов, в динамическом случае содержит переменные. Единственное требование - значения переменных должны быть определены в момент объявления. Это ограничение в C# выполняется автоматически.

Работа с *динамическим массивом*:

```
public void TestDynAr()
{
    //объявление динамического массива A1
    Console.WriteLine("Введите число элементов массива A1");
    int size = int.Parse(Console.ReadLine());
    int[] A1 = new int[size];
    Arrs.CreateOneDimAr(A1);
    Arrs.PrintAr1("A1", A1);
}//TestDynAr
```

Многомерные массивы

Одномерные *массивы* -это частный случай *многомерных*. Одномерные *массивы* позволяют задавать такие *математические структуры* как векторы, двумерные - матрицы, трехмерные - *кубы данных, массивы* большей *размерности* - многомерные *кубы данных*. Заметим, что при работе с базами данных многомерные кубы, так называемые кубы *OLAP*, встречаются сплошь и рядом.

Для объявления многомерного массива указывается его размерность с использованием запятых. Вот как выглядит объявление *многомерного массива* в общем случае:

<тип>[, ... ,] <объявители>;

Число запятых, увеличенное на единицу, и задает *размерность массива*. Что касается объявителей, то все, что сказано для одномерных *массивов*, справедливо и для *многомерных*. Можно лишь отметить, что, хотя явная *инициализация* с использованием *многомерных* константных *массивов* возможна, но применяется редко из-за громоздкости такой структуры. Проще инициализацию реализовать программно, но иногда она все же применяется. Вот пример:

```
public void TestMultiArr()
{
    int[,]matrix = {{1,2},{3,4}};
    Arrs.PrintAr2("matrix", matrix);}
```

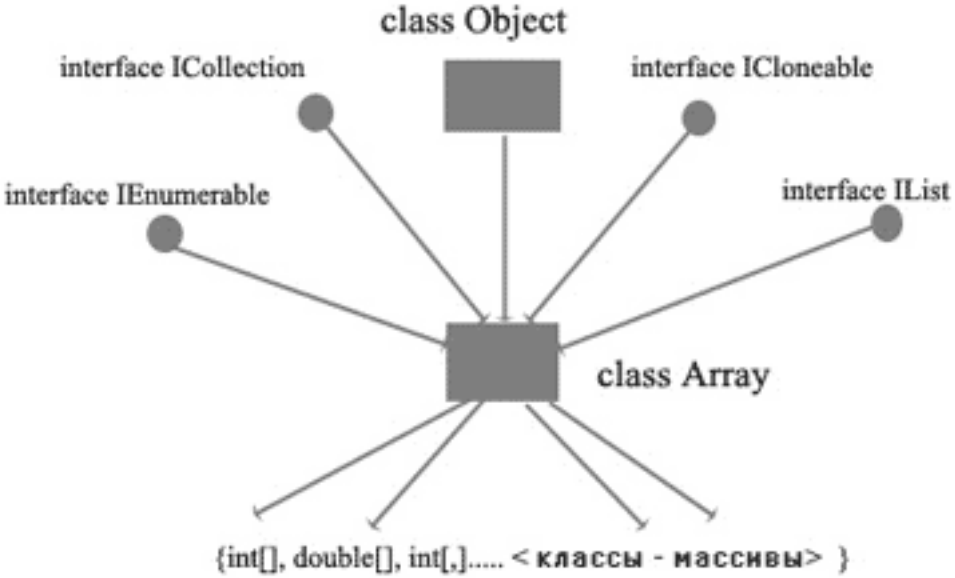
31. Семейство классов-массивов.

Нельзя понять многие детали работы с массивами в C#, если не знать устройство класса Array из библиотеки *FCL*, потомками которого являются все классы-массивы. Рассмотрим следующие объявления:

```
int[] ar1 = new int[5];
double[] ar2 ={5.5, 6.6, 7.7};
int[,] ar3 = new Int32[3,4];
```

Все объекты (**ar1, ar2, ar3**) принадлежат к разным классам. *Переменная* **ar1** принадлежит к классу **int[]** - одномерному массиву значений типа **int**, **ar2** - **double**[] - одномерному массиву значений типа **double**, **ar3** - двумерному массиву значений типа **int**. Прежде всего, все три класса этих объектов, как и другие классы, являются потомками класса Object, а потому имеют общие методы, наследованные от класса Object и доступные объектам этих классов.

У всех классов, являющихся массивами, много общего, поскольку все они являются потомками *класса System.Array*. *Класс System.Array* наследует ряд интерфейсов: ***ICloneable, IList, ICollection, IEnumerable***, а, следовательно, обязан реализовать все их методы и свойства. Помимо наследования свойств и методов класса Object и вышеперечисленных интерфейсов, *класс* Array имеет довольно большое число собственных методов и свойств. Взгляните, как выглядит *отношение наследования* на семействе классов, определяющих массивы.



Благодаря такому мощному родителю, над массивами определены самые разнообразные *операции* - *копирование*, *поиск*, обращение, *сортировка*, получение различных характеристик.

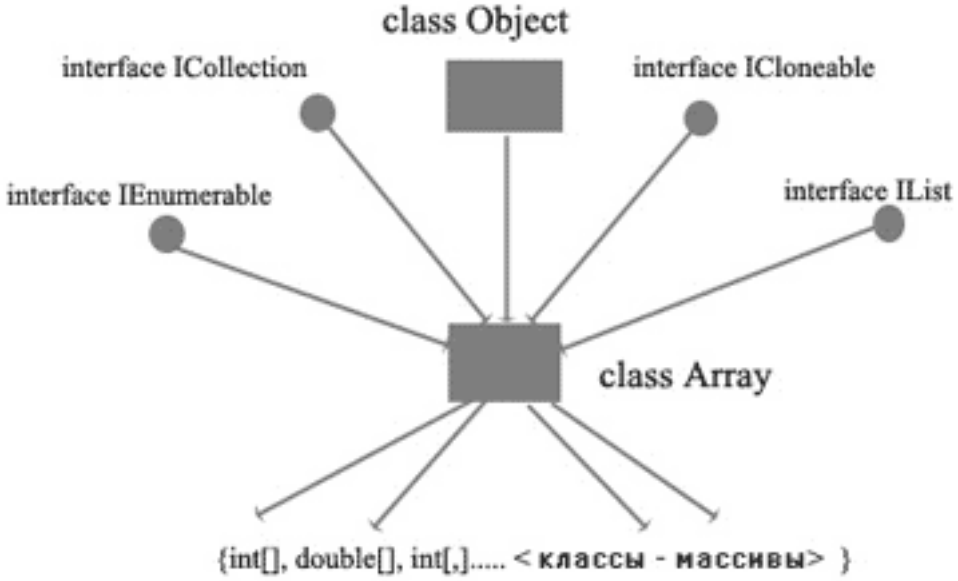
32. Родительский класс Array и наследуемые им интерфейсы.

Нельзя понять многие детали работы с массивами в C#, если не знать устройство класса Array из библиотеки *FCL*, потомками которого являются все классы-массивы. Рассмотрим следующие объявления:

```
int[] ar1 = new int[5];
double[] ar2 ={5.5, 6.6, 7.7};
int[,] ar3 = new Int32[3,4];
```

Все объекты (**ar1, ar2, ar3**) принадлежат к разным классам. *Переменная* **ar1** принадлежит к классу **int[]** - одномерному массиву значений типа **int**, **ar2 - double**[] - одномерному массиву значений типа **double**, **ar3** - двумерному массиву значений типа **int**. Прежде всего, все три класса этих объектов, как и другие классы, являются потомками класса Object, а потому имеют общие методы, наследованные от класса Object и доступные объектам этих классов.

У всех классов, являющихся массивами, много общего, поскольку все они являются потомками *класса System.Array*. *Класс System.Array* наследует
ряд интерфейсов: ***ICloneable, IList, ICollection, IEnumerable***, а, следовательно, обязан реализовать все их методы и свойства. Помимо наследования свойств и методов класса Object и вышеперечисленных интерфейсов, *класс* Array имеет довольно большое число собственных методов и свойств. Взгляните, как выглядит *отношение наследования* на семействе классов, определяющих массивы.



Благодаря такому мощному родителю, над массивами определены самые разнообразные *операции - копирование, поиск*, обращение, *сортировка*, получение различных характеристик.

Массивы можно рассматривать как *коллекции* и устраивать циклы **for each** для перебора всех элементов. Важно и то, что когда у семейства классов есть общий родитель, то можно иметь общие процедуры обработки различных потомков этого родителя. Эта возможность обеспечивается тем, что класс **Array** наследует интерфейс *IEnumerable*. Этот интерфейс обеспечивает только возможность чтения элементов *коллекции* (массива), не допуская их изменения.

Статические методы класса **Array** позволяют решать самые разнообразные задачи:

Copy - позволяет копировать весь массив или его часть в другой массив.

IndexOf, ***LastIndexOf*** - определяют индексы первого и последнего вхождения образца в массив, возвращая -1, если такового вхождения не обнаружено.

Reverse - выполняет обращение массива, переставляя элементы в обратном порядке.

Sort - осуществляет *сортировку* массива.

BinarySearch - определяет индекс первого вхождения образца в отсортированный массив, используя алгоритм двоичного *поиска*.

Динамические методы класса Array

Метод	Родитель	Описание
Equals	Класс Object	Описание и примеры даны в предыдущих главах.
GetHashCode	Класс Object	Описание и примеры даны в предыдущих главах.
GetType	Класс Object	Описание и примеры даны в предыдущих главах.
ToString	Класс Object	Описание и примеры даны в предыдущих главах.
Clone	Интерфейс *ICloneable*	Позволяет создать плоскую или глубокую копию массива. В первом случае создаются только элементы первого уровня, а ссылки указывают на те же самые объекты. Во втором случае копируются объекты на всех уровнях. Для массивов создается только плоская копия.
CopyTo	Интерфейс *ICollection*	Копируются все элементы одномерного массива в другой одномерный массив, начиная с заданного индекса: col1.CopyTo(col2,0);
GetEnumerator	Интерфейс *IEnumerable*	Стоит за спиной цикла foreach
GetLength		Возвращает число элементов массива по указанному измерению. Описание и примеры даны в тексте главы.
GetLowerBound, GetUpperBound		Возвращает нижнюю и верхнюю границу по указанному измерению. Для массивов нижняя граница всегда равна нулю.
GetValue, SetValue		Возвращает или устанавливает значение элемента массива с указанными индексами.

33.Возможности массивов в C#. Корректная работа с массивами объектов.

Массивы как коллекции

В ряде задач массивы C# целесообразно рассматривать как *коллекции*, не используя систему индексов для *поиска* элементов. Это, например, задачи, требующие однократного или многократного прохода по всему массиву - нахождение суммы элементов, нахождение максимального элемента, печать элементов. В таких задачах вместо циклов типа For по каждому измерению достаточно рассмотреть единый цикл For Each по всей *коллекции*. Эта возможность обеспечивается тем, что класс Array наследует интерфейс *IEnumerable*. Обратите внимание, этот интерфейс обеспечивает только возможность чтения элементов *коллекции* (массива), не допуская их изменения. Применим эту стратегию и построим еще одну версию процедуры печати. Эта версия будет самой короткой и самой универсальной, поскольку подходит для печати массива, независимо от его размерности и типа элементов. Вот ее код:

```
public static void PrintCollection(string name,Array A)
{
   Console.WriteLine(name);
   foreach (object item in A )
         Console.Write("\t {0}", item);
   Console.WriteLine();
}//PrintCollection
```

Конечно, за все нужно платить. Платой за универсальность процедуры печати является то, что *многомерный массив* печатается как одномерный без разделения элементов на строки.

К сожалению, ситуация с чтением и записью элементов массива не симметрична. Приведу вариант процедуры CreateCollection:

```
public static void CreateCollection(Array A)
{ int i=0;
   foreach (object item in A )
         //item = rnd.Next(1,10);       //item read only
         A.SetValue(rnd.Next(1,10), i++);  }//CreateCollection
```

Заметьте, эту процедуру сделать универсальной не удается, поскольку невозможно модифицировать элементы *коллекции*. Поэтому цикл For Each здесь ничего не дает, и разумнее использовать обычный цикл. Данная процедура не универсальна и позволяет создавать элементы только для одномерных массивов.

Массивы объектов

В реальных программах *массивы объектов* и других ссылочных типов встречаются не менее часто. Каков бы ни был тип элементов, большой разницы при работе с массивами нет. Но один важный нюанс все же есть, и его стоит

отметить. Он связан с инициализацией элементов *по* умолчанию. Уже говорилось о том, что *компилятор* не следит за инициализацией элементов массива и доверяет инициализации, выполненной конструктором массива *по* умолчанию. Но для массивов ссылочного типа *инициализация по* умолчанию присваивает ссылкам *значение* Null. Это означает, что создаются только ссылки, но не сами объекты. *По* этой причине, пока не будет проведена настоящая *инициализация* с созданием объектов и заданием ссылок на конкретные объекты, работать с массивом ссылочного типа будет невозможно.

Рассмотрим детали этой проблемы на примере. Определим достаточно простой и интуитивно понятный *класс*, названный Winners, свойства которого задают имя победителя и его премию, а методы позволяют установить размер премии для каждого победителя и распечатать его свойства. Приведу код, описывающий этот *класс*:

```
/// <summary> Класс победителей с именем и премией
/// </summary>
public class Winners
{  //поля класса
   string name;
   int price;
   //статическое или динамическое поле rnd?
   //static Random rnd = new Random();
   Random rnd = new Random();
   // динамические методы
   public void SetVals(string name)
   {
         this.name = name;
         this.price = rnd.Next(5,10)* 1000;
   }//SetVals
   public void PrintWinner(Winners win)
   {
         Console.WriteLine("Имя победителя: {0}," +
               " его премия - {1}", win.name, win.price);
   }//PrintWinner
}//class Winners
```

Коротко прокомментирую этот текст.

1. Свойство name описывает имя победителя, а свойство price - величину его премии.
2. Свойство rnd необходимо при работе со случайными числами.
3. Метод SetVals выполняет инициализацию. Он присваивает полю name значение, переданное в качестве аргумента, и полю price - случайное значение.

4. Метод PrintWinner - метод печати свойств класса. Без подобного метода не обходится ни один класс.

5. В классе появится еще один статический метод InitAr, но о нем скажу чуть позже.

Пусть теперь в одном из методов нашего тестирующего класса Testing предполагается работа с классом Winners, начинающаяся с описания победителей. Естественно, задается *массив*, элементы которого имеют тип Winners. Приведу начало тестирующего метода, в котором дано соответствующее объявление:

```
public void TestWinners()
{
   //массивы объектов
   int nwin = 3;
   Winners[] wins = new Winners[nwin];
   string[] winames = {"Т. Хоар", "Н. Вирт", "Э. Дейкстра"};
```

В результате создан *массив* wins, состоящий из объектов класса Winners. Что произойдет, если попытаться задать значения полей объектов, вызвав специально созданный для этих целей метод SetVals? Рассмотрим фрагмент кода, осуществляющий этот вызов:

```
//создание значений элементов массива
                    for(int i=0; i < wins.Length; i++)
                              wins[i].SetVals(winames[i]);
```

На этапе выполнения будет сгенерировано *исключение* - нулевая *ссылка*. Причина понятна: хотя *массив* wins и создан, но это *массив*ссылок, имеющих *значение* null. Сами объекты, на которые должны указывать ссылки, не создаются в момент объявления массива ссылочного типа. Их нужно создавать явно. Ситуация аналогична объявлению массива массивов. И там необходим явный вызов конструктора для создания каждого массива на внутреннем уровне.

Как же создавать эти объекты? Конечно, можно возложить эту обязанность на пользователя, объявившего *массив* wins, - пусть он и создаст экземпляры для каждого элемента массива. Правильнее все-таки иметь в классе соответствующий метод. Метод должен быть статическим, чтобы его можно было вызывать еще до того, как созданы экземпляры класса, поскольку метод предназначен для создания этих самых экземпляров. Так в нашем классе появился *статический метод* InitAr:

```
//статический метод
public static Winners[] InitAr(Winners[] Winar)
{
   for(int i=0; i < Winar.Length; i++)
            Winar[i] = new Winners();
   return(Winar);
```

```
}//InitAr
```

Методу передается *массив* объектов, возможно, с нулевыми ссылками. Он возвращает тот же *массив*, но уже с явно определенными ссылками на реально созданные объекты. Теперь достаточно вызвать этот метод, после чего можно спокойно вызывать и метод SetVals. Вот как выглядит правильная последовательность вызовов методов класса Winners:

```
Winners.InitAr(wins);
        //создание значений элементов массива
        for(int i=0; i < wins.Length; i++)
            wins[i].SetVals(winames[i]);
        //печать значений элементов массива
        for(int i=0; i < wins.Length; i++)
            wins[i].PrintWinner(wins[i]);
    }//TestWinners
```

34.Общий взгляд. Строки в C#.

В первых языках программирования строковому типу уделялось гораздо меньше внимания, чем арифметическому типу или массивам. Поэтому в разных языках строки представлены *по*-разному и стандарт на *строковый тип*сложился относительно недавно. Когда говорят о строковом типе, то обычно различают тип, представляющий:

1)отдельные символы, чаще всего, его называют типом char ;

2)*строки постоянной длины*, часто они представляются массивом символов;

3)строки переменной длины - это, как правило, тип string, соответствующий современному представлению о строковом типе.

Символьный тип char, представляющий частный случай строк длиной 1, полезен во многих задачах. Основные *операции* над строками - это разбор и *сборка*. При их выполнении приходится, чаще всего, доходить до каждого символа строки. В языке *Паскаль*, где был введен тип char, сам *строковый тип* рассматривался, как char[] -*массив* символов. При таком подходе получение i -го символа строки становится такой же простой операцией, как и получение i -го элемента массива. Следовательно, эффективно реализуются обычные *операции* над строками - *определение* вхождения одной строки в другую, *выделение подстроки*, замена символов строки. Однако заметьте, *представление* строки массивом символов хорошо только для *строк постоянной длины*. *Массив* не приспособлен к изменению его размеров, вставки или удалению символов (подстрок).

Наиболее часто используемым строковым типом является тип, обычно называемый string, который задает строки переменной длины. Над этим типом допускаются *операции* поиска вхождения одной строки в другую, *операции* вставки, замены и удаления подстрок.

Строки C#

В C# есть *символьный класс Char*, основанный на классе System.Char и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы - символьные литералы. Константу можно задавать:

1) символом, заключенным в одинарные кавычки;

2) escape-последовательностью, задающей код символа;

3) Unicode-последовательностью, задающей Unicode-код символа.

примеры объявления символьных переменных и работы с ними:

```
public void TestChar()
{
   char ch1='A', ch2 ='\x5A', ch3='\u0058';
   char ch = new Char();
   int code; string s;
   ch = ch1;
```

```
    //преобразование символьного типа в тип int
    code = ch; ch1=(char) (code +1);
    //преобразование символьного типа в строку
    //s = ch;
    s = ch1.ToString()+ch2.ToString()+ch3.ToString();
    Console.WriteLine("s= {0}, ch= {1}, code = {2}",
            s, ch, code);
}//TestChar
```

Три символьные переменные инициализированы константами, значения которых заданы тремя разными способами. Переменная chобъявляется в объектном стиле, используя new и вызов конструктора класса. Тип char, как и все типы C#, является классом. Этот класс наследует свойства и методы класса Object и имеет большое число собственных методов.

Явные или неявные преобразования между *классами char* и string отсутствуют, но, благодаря методу ToString, переменные типа char стандартным образом преобразуются в тип string. Как отмечалось в лекции 3, существуют *неявные преобразования типа* char в целочисленные типы, начиная с типа ushort. Обратные преобразования целочисленных типов в тип char также существуют, но они уже явные.

Не раз отмечалось, что семантика присваивания справедлива при вызове методов и замене формальных аргументов на фактические. Приведу две процедуры, выполняющие взаимно-обратные операции - получение по коду символа и получение символа по его коду:

```
public int SayCode(char sym)
{
   return (sym);
}//SayCode
public char SaySym(object code)
{    return ((char)((int)code));
}// SaySym
```

Как видите, в первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное.

Статические методы и свойства класса Char	
Метод	**Описание**
GetNumericValue	Возвращает численное значение символа, если он является цифрой, и (-1) в противном случае
GetUnicodeCategory	Все символы разделены на категории. Метод возвращает Unicode категорию символа. Ниже приведен пример
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой

IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичной или шестнадцатиричной цифрой)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsSurrogate	Некоторые символы Unicode с кодом в интервале [0x10000, 0x10FFF] представляются двумя 16-битными "суррогатными" символами. Метод возвращает true, если символ является суррогатным
IsUpper	Возвращает true, если символ задан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
Parse	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру
MaxValue, *MinValue*	Свойства, возвращающие символы с максимальным и минимальным кодом. Возвращаемые символы не имеют видимого образа

35.Строки постоянной и переменной длинны. Классы char, char[].

В C# есть *символьный класс* ***Char***, основанный на классе System.Char и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы - символьные литералы. Константу можно задавать:

1) символом, заключенным в одинарные кавычки;
2) escape-последовательностью, задающей код символа;
3) Unicode-последовательностью, задающей Unicode-код символа.

Примеры объявления символьных переменных и работы с ними:

```
public void TestChar()
{char ch1='A', ch2 ='\x5A', ch3='\u0058';
   char ch = new Char();
   int code; string s;
   ch = ch1;
   //преобразование символьного типа в тип int
   code = ch; ch1=(char) (code +1);
   //преобразование символьного типа в строку
   //s = ch;
   s = ch1.ToString()+ch2.ToString()+ch3.ToString();
   Console.WriteLine("s= {0}, ch= {1}, code = {2}",
         s, ch, code);}
```

Три символьные переменные инициализированы константами, значения которых заданы тремя разными способами. Переменная ch объявляется в объектном стиле, используя new и вызов конструктора класса. Тип char, как и все типы C#, является классом. Этот класс наследует свойства и методы класса Object и имеет большое число собственных методов.

Явные или неявные преобразования между *классами char* и string отсутствуют, но, благодаря методу ToString, переменные типа char стандартным образом преобразуются в тип string. Существуют *неявные преобразования типа* char в целочисленные типы, начиная с типа ushort. Обратные преобразования целочисленных типов в тип char также существуют, но они уже явные.

В результате работы процедуры TestChar строка s, полученная сцеплением трех символов, преобразованных в строки, имеет значение BSX, переменная ch равна A, а ее код - переменная code - 65.

Не раз отмечалось, что семантика присваивания справедлива при вызове методов и замене формальных аргументов на фактические. Приведу две процедуры, выполняющие взаимно-обратные операции - получение по коду символа и получение символа по его коду:

```
public int SayCode(char sym)
{
   return (sym);
```

```
}//SayCode
public char SaySym(object code)
{
   return ((char)((int)code));
}// SaySym
```

Как видите, в первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное. Ради универсальности она слегка усложнена. Формальный параметр имеет тип Object, что позволяет передавать ей в качестве аргумента код, заданный любым целочисленным типом. Платой за это является необходимость выполнять два явных преобразования.

Класс Char, как и все классы в C#, наследует свойства и методы родительского класса Object. Но у него есть и собственные методы и свойства, и их немало. Сводка этих методов приведена в таблице 13.1.

Большинство статических методов перегружены. Они могут применяться как к отдельному символу, так и к строке, для которой указывается номер символа для применения метода. Основную группу составляют методы Is, крайне полезные при разборе строки.

Кроме статических методов, у *класса Char* есть и динамические. Большинство из них - это методы родительского класса Object, унаследованные и переопределенные в *классе Char*. Из собственных динамических методов стоит отметить метод CompareTo, позволяющий проводить сравнение символов. Он отличается от метода Equal тем, что для несовпадающих символов выдает "расстояние" между символами в соответствии с их упорядоченностью в кодировке Unicode.

Статические методы и свойства класса Char	
Метод	**Описание**
GetNumericValue	Возвращает численное значение символа, если он является цифрой, и (-1) в противном случае
GetUnicodeCategory	Все символы разделены на категории. Метод возвращает Unicode категорию символа. Ниже приведен пример
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичной или шестнадцатиричной цифрой)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем

IsSurrogate	Некоторые символы Unicode с кодом в интервале [0x10000, 0x10FFF] представляются двумя 16-битными "суррогатными" символами. Метод возвращает true, если символ является суррогатным
IsUpper	Возвращает true, если символ задан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
Parse	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру
MaxValue, *MinValue*	Свойства, возвращающие символы с максимальным и минимальным кодом. Возвращаемые символы не имеют видимого образа

Класс char[] - массив символов

В языке C# определен *класс Char[]*, и его можно использовать для представления *строк постоянной длины*, как это делается в C++. Более того, поскольку массивы в C# динамические, то расширяется класс задач, в которых можно использовать массивы символов для представления строк. Так что имеет смысл разобраться, насколько хорошо C# поддерживает работу с таким представлением строк.

Массив char[] - это обычный массив. Более того, его нельзя инициализировать строкой символов, как это разрешается в C++. Константа, задающая строку символов, принадлежит классу String, а в C# не определены взаимные преобразования между классами String и *Char[]*, даже явные. У класса String есть, правда, динамический метод ToCharArray, задающий подобное преобразование. Возможно также посимвольно передать содержимое переменной string в массив символов.

Приведу пример:

```
public void TestCharArAndString()
{
    //массивы символов
    //char[] strM1 = "Hello, World!";
    //ошибка: нет преобразования класса string в класс char[]
    string hello = "Здравствуй, Мир!";
    char[] strM1 = hello.ToCharArray();
    PrintCharAr("strM1",strM1);
    //копирование подстроки
    char[] World = new char[3];
```

```
    Array.Copy(strM1,12,World,0,3);
    PrintCharAr("World",World);
    Console.WriteLine(CharArrayToString(World));
}//TestCharArAndString
```

В нашем примере часть строки strM1 копируется в массив World. По ходу дела в методе вызывается процедура PrintCharArкласса Testing, печатающая массив символов как строку.

```
void PrintCharAr(string name,char[] ar)
{Console.WriteLine(name);
    for(int i=0; i < ar.Length; i++)
            Console.Write(ar[i]);
    Console.WriteLine();}
```

Метод ToCharArray позволяет преобразовать строку в массив символов. К сожалению, обратная операция не определена, поскольку метод ToString, которым, конечно же, обладают все объекты *класса Char[]*, печатает информацию о классе, а не содержимое массива. Ситуацию легко исправить, написав подходящую процедуру.

Класс Char[], как и всякий класс-массив в C#, является наследником не только класса Object, но и класса Array, и, следовательно, обладает всеми методами родительских классов.

36.Строки C#. Класс String. Изменяемые и неизменяемые строковые классы.

Класс String

Основным типом при работе со строками является тип *string*, задающий строки переменной длины. *Класс String* в языке C# относится к ссылочным типам. Над строками - объектами этого класса - определен широкий набор операций.

Объявление строк. Конструкторы класса string

Объекты *класса String* объявляются как все прочие объекты простых типов - с явной или отложенной инициализацией, с явным или неявным вызовом конструктора класса. Чаще всего, при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковой константой. Но у класса String достаточно много конструкторов. Они позволяют сконструировать строку из:

- символа, повторенного заданное число раз;
- массива символов char[] ;
- части массива символов.

Некоторым конструкторам в качестве параметра инициализации можно передать строку, заданную типом char*. Но все это небезопасно, и подобные примеры приводиться и обсуждаться не будут. Приведу примеры объявления строк с вызовом разных конструкторов:

```
public void TestDeclStrings()
{//конструкторы
   string world = "Мир";
   //string s1 = new string("s1");
   //string s2 = new string();
   string sssss = new string('s',5);
   char[] yes = "Yes".ToCharArray();
   string stryes = new string(yes);
   string strye = new string(yes,0,2);
   Console.WriteLine("world = {0}; sssss={1}; stryes={2};"+
            " strye= {3}", world, sssss, stryes, strye); }
```

Объект world создан без явного вызова конструктора, а объекты sssss, stryes, strye созданы разными конструкторами *класса String*.

Операции над строками

Над строками определены следующие операции:

- присваивание (=);
- две операции проверки эквивалентности (==) и (!=);
- конкатенация или сцепление строк (+);
- взятие индекса ([]).

Строковые константы

В C# существуют два вида строковых констант:

- обычные константы, которые представляют строку символов, заключенную в кавычки;
- *Эт-константы*, заданные обычной константой с предшествующим знаком @.

Неизменяемый класс string

В языке C# существует понятие *неизменяемый (immutable) класс*. Для такого класса невозможно изменить значение объекта при вызове его методов. Динамические методы могут создавать новый объект, но не могут изменить значение существующего объекта.

К таким *неизменяемым классам* относится и *класс String*. Ни один из методов этого класса не меняет значения существующих объектов. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Невозможность изменять значения строк касается не только методов. Аналогично, при работе со строкой как с массивом разрешено только чтение отдельных символов, но не их замена.

Статические свойства и методы класса String

Таблица 14.1. Статические методы и свойства класса String

Метод	Описание
Empty	Возвращается пустая строка. Свойство со статусом read only
Compare	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать или не учитывать регистр, особенности национального форматирования дат, чисел и т.д.
CompareOrdinal	Сравнение двух строк. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
Concat	Конкатенация строк. Метод перегружен, допускает сцепление произвольного числа строк
Copy	Создается копия строки
Format	Выполняет форматирование в соответствии с заданными спецификациями *формата*. Ниже приведено более полное описание метода
Intern, IsIntern	Отыскивается и возвращается ссылка на строку, если таковая уже хранится во внутреннем пуле данных. Если же строки нет, то первый из методов добавляет строку во внутренний пул, второй - возвращает null. Методы применяются обычно тогда, когда строка

	создается с использованием построителя строк - класса *StringBuilder*
Join	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом *Join*, является обратной к операции, заданной методом *Split*. Последний является динамическим методом и, используя разделители, осуществляет разделение строки на элементы

Динамические методы класса String

Операции, разрешенные над строками в C#, разнообразны. Методы этого класса позволяют выполнять вставку, удаление, замену, поиск вхождения подстроки в строку. *Класс String* наследует методы класса Object, частично их переопределяя. *Класс String*наследует и, следовательно, реализует методы четырех интерфейсов: IComparable, *ICloneable*, IConvertible, IEnumerable. Три из них уже рассматривались при описании классов-массивов.

Динамические методы и свойства класса String

Метод	Описание
Insert	Вставляет подстроку в заданную позицию
Remove	Удаляет подстроку в заданной позиции
Replace	Заменяет подстроку в заданной позиции на новую подстроку
Substring	Выделяет подстроку в заданной позиции
IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
StartsWith, EndsWith	Возвращается true или false, в зависимости от того, начинается или заканчивается строка заданной подстрокой
PadLeft, PadRight	Выполняет набивку нужным числом пробелов в начале и в конце строки
Trim, TrimStart, TrimEnd	Обратные операции к методам Pad. Удаляются пробелы в начале и в конце строки, или только с одного ее конца
ToCharArray	Преобразование строки в массив символов

37.Классы Net Framework, расширяющие строковый тип.

Методы Join и Split выполняют над строкой текста взаимно обратные преобразования. **Динамический метод Split** позволяет осуществить разбор текста на элементы. **Статический метод Join** выполняет обратную операцию, собирая строку из элементов.

Заданный строкой текст, зачастую, представляет собой совокупность структурированных элементов – абзацев, предложений, слов, скобочных выражений и т.д. При работе с таким текстом необходимо разделить его на элементы, пользуясь тем, что есть специальные разделители элементов, – это могут быть пробелы, скобки, знаки препинания. Практически подобные задачи возникают постоянно при работе со структурированными текстами. Методы Split и Join облегчают решение подобных задач.

Динамический метод Split, как обычно перегружен. Наиболее часто используемая реализация имеет следующий синтаксис:

public string[] Split(params char[])

На вход методу Split передается один или несколько символов, интерпретируемых, как разделители. Объект string, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода. Другая реализация метода позволяет ограничить число элементов возвращаемого массива.

Синтаксис статического метода Join таков:

public static string Join(string delimiters, string[] items)

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива items, между которыми вставляется строка разделителей delimiters. Как правило, строка delimiters состоит из одного символа, который и разделяет в результирующей строке элементы массива items, но в отдельных случаях ограничителем может быть строка из нескольких символов.

Методы Split и Join хорошо работают, когда при разборе используется только один разделитель. В этом случае сборка действительно является обратной операцией и позволяет восстановить исходную строку. Если же при разборе задается некоторое множество разделителей, то возникают две проблемы:

1)невозможно при сборке восстановить строку в прежнем виде, поскольку не сохраняется информация о том, какой из разделителей был использован при разборе строки. Поэтому при сборке между элементами вставляется один разделитель, возможно, состоящий из нескольких символов.

2)При разборе двух подряд идущих разделителей предполагается, что между ними находится пустое слово. Обратите внимание, в тексте нашего примера, как и положено, после запятой следует пробел. При разборе текста на

слова в качестве разделителей указаны символы пробела и запятой. По этой причине в массиве слов, полученном в результате разбора, имеются пустые слова.

Если при разборе предложения на слова, использовать в качестве разделителя только пробел, то пустые слова не появятся, но запятая будет являться частью некоторых слов.

Как всегда, есть несколько способов справиться с проблемой. Один из них состоит в том, чтобы написать собственную реализацию этих функций, другой – в корректировке полученных результатов, третий – в использовании более мощного аппарата регулярных выражений.

38.Класс StringBuilder.

Класс StringBuilder - построитель строк

Объекты этого класса объявляются с явным вызовом конструктора класса. Поскольку специальных констант этого типа не существует, то вызов конструктора для инициализации объекта просто необходим. Конструктор класса перегружен, и наряду с конструктором без параметров, создающим пустую строку, имеется набор конструкторов, которым можно передать две группы параметров. Первая группа позволяет задать строку или подстроку, значением которой будет инициализироваться создаваемый объект класса *StringBuilder*. Вторая группа параметров позволяет задать *емкость объекта* - объем памяти, отводимой данному экземпляру класса *StringBuilder*. Каждая из этих групп не является обязательной и может быть опущена. Примером может служить конструктор без параметров, который создает объект, инициализированный пустой строкой, и с некоторой *емкостью* , заданной по умолчанию, значение которой зависит от реализации. Приведу в качестве примера синтаксис трех конструкторов:

public StringBuilder (string str, int cap). Параметр str задает строку инициализации, cap - *емкость объекта* ;

public StringBuilder (int curcap, int maxcap). Параметры curcap и maxcap задают начальную и максимальную *емкость объекта* ;

public StringBuilder (string str, int start, int len, int cap). Параметры str, start, len задают строку инициализации, cap - *емкость объекта*.

Операции над строками

Над строками этого класса определены практически те же операции с той же семантикой, что и над строками *класса String*:

присваивание (=);

две операции проверки эквивалентности (==) и (!=);

взятие индекса ([]).

Операция конкатенации (+) не определена над строками класса *StringBuilder*, ее роль играет метод Append, дописывающий новую строку в хвост уже существующей.

Со строкой этого класса можно работать как с массивом, но, в отличие от *класса String*, здесь уже все делается как надо: допускается не только чтение отдельного символа, но и его изменение.

Основные методы

У класса *StringBuilder* методов значительно меньше, чем у *класса String*. Технология работы класса: конструируется строка класса *StringBuilder* ; выполняются операции, требующие изменение значения; полученная строка преобразуется в строку *класса String* ; над этой строкой выполняются операции, не требующие изменения значения строки.

1)public StringBuilder Append (<объект>). К строке, вызвавшей метод, присоединяется строка, полученная из объекта, который передан методу в качестве параметра. Метод перегружен и может принимать на входе объекты всех простых типов, начиная от char и bool до string и long. Поскольку объекты всех этих типов имеют метод ToString, всегда есть возможность преобразовать объект в строку, которая и присоединяется к исходной строке. В качестве результата возвращается ссылка на объект, вызвавший метод. Поскольку возвращаемую ссылку ничему присваивать не нужно, то правильнее считать, что метод изменяет значение строки;

2)public StringBuilder Insert (int location,<объект>). Метод вставляет строку, полученную из объекта, в позицию, указанную параметром location. Метод Append является частным случаем метода Insert ;

3)public StringBuilder Remove (int start, int len). Метод удаляет подстроку длины len, начинающуюся с позиции start ;

4)public StringBuilder Replace (string str1,string str2). Все вхождения подстроки str1 заменяются на строку str2 ;

5)public StringBuilder AppendFormat (<строка форматов>, <объекты>). Метод является комбинацией *метода Format класса String* и метода Append. Строка *форматов*, переданная методу, содержит только спецификации *форматов*. В соответствии с этими спецификациями находятся и форматируются объекты. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода **Remove**, все рассмотренные методы являются перегруженными. В их описании дана схема вызова метода, а не точный синтаксис перегруженных реализаций.

Каждый экземпляр строки класса *StringBuilder* имеет буфер, в котором хранится строка. Объем буфера - его емкость - может меняться в процессе работы со строкой. Объекты класса имеют две характеристики емкости - текущую и максимальную. В процессе работы текущая емкость изменяется, естественно, в пределах максимальной емкости, которая реально достаточно высока.

У класса *StringBuilder* имеется 2 свойства и один метод, позволяющие анализировать и управлять емкостными свойствами буфера.

1)свойство Capacity - возвращает или устанавливает текущую емкость буфера;

2)свойство MaxCapacity - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса;

3)метод int EnsureCapacity (int capacity) - позволяет уменьшить емкость буфера. Метод пытается вначале установить емкость, заданную параметром capacity ; если это значение меньше размера хранимой строки, то емкость устанавливается такой, чтобы гарантировать размещение строки. Это число и возвращается в качестве результата работы метода.

39.Регулярные выражения. Пространство RegularExpression и его классы.

Регулярные выражения - это более простой и удобный способ записи *регулярных множеств* в виде обычной строки. Этот класс языков - достаточно мощный, с его помощью можно описать интересные языки, но устроены они довольно просто - их можно определить также с помощью простых грамматик, например, правосторонних грамматик. Более важно, что для любого *регулярного выражения* можно построить конечный автомат, который распознает, принадлежит ли заданное слово языку, порожденному *регулярным выражением*. На этом основана практическая ценность *регулярных выражений*.

С точки зрения практики *регулярное выражение* задает образец поиска. После чего можно проверить, удовлетворяет ли заданная строка или ее подстрока данному образцу. В языках программирования синтаксис *регулярного выражения* существенно обогащается, что дает возможность более просто задавать сложные образцы поиска. Такие синтаксические надстройки, хотя и не меняют сути *регулярных выражений*, крайне полезны для практиков, избавляя программиста от ненужных сложностей. (В Net Framework эти усложнения, на мой взгляд, чрезмерны. Выигрывая в мощности языка, проигрываем в простоте записи его выражений.)

Синтаксис регулярных выражений

Регулярное выражение на C# задается строковой константой. Это может быть обычная или @ -константа. Чаще всего, следует использовать именно @ -константу. Дело в том, что символ " \ " широко применяется в *регулярных выражениях* как для записи escape-последовательностей, так и в других ситуациях. Обычные константы в таких случаях будут выдавать синтаксическую ошибку, а @ -константы не выдают ошибок и корректно интерпретируют запись *регулярного выражения*.

Синтаксис *регулярного выражения* простой формулой не описать, здесь используются набор разнообразных средств:

1)символы и escape-последовательности;

2)символы операций и символы, обозначающие специальные классы множеств;

3)имена *групп* и *обратные ссылки*;

4)символы утверждений и другие средства.

Конечно, *регулярное выражение* может быть совсем простым, например, строка " abc " задает образец поиска, так что при вызове соответствующего метода будут разыскиваться одно или все вхождения подстроки " abc " в искомую строку. Но могут существовать и очень сложно устроенные *регулярные выражения*. Приведу таблицу, (15.1) в которой дается интерпретация символов в соответствии с их делением на *группы*. Таблица не полна, в ней отражаются не все *группы*, а описание *группы* не содержит всех символов. Она позволяет дать общее представление о синтаксисе, которое будет дополнено большим числом примеров. За деталями придется обращаться к справочной системе, которая, к сожалению, далеко не идеальна для данного раздела.

Повторяю, данная таблица не полна. В ней не отражены, например, такие категории, как подстановки, *обратные ссылки*, утверждения.

Знакомство с классами пространства RegularExpressions

В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.

Класс Regex

Это основной класс, всегда создаваемый при работе с *регулярными выражениями*. Объекты этого класса определяют *регулярные выражения*. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая *регулярное выражение*. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению RegexOptions и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать *регулярное выражение*. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Классы Match и MatchCollection

Как уже говорилось, объекты этих классов создаются автоматически при вызове *методов Match* и Matches. *Коллекция MatchCollection*, как и все коллекции, позволяет получить доступ к каждому ее элементу - объекту *Match*. Можно, конечно, организовать цикл foreach для последовательного доступа ко всем элементам коллекции.

Классы Group и GroupCollection

Коллекция GroupCollection возвращается при вызове свойства Group объекта Match. Имея эту коллекцию, можно добраться до каждого объекта Group, в нее входящего. *Класс Group* является наследником класса Capture и, одновременно, родителем *класса Match*. От своего родителя он наследует свойства Index, Length и Value, которые и передает своему потомку.

Классы Capture и CaptureCollection

Коллекция CaptureCollection возвращается при вызове свойства Captures объектов *класса Group* и *Match*. *Класс Match* наследует это свойство у своего родителя - *класса Group*.

Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска *регулярные выражения*. В этом случае необходимо будет создать объект класса RegexCompilationInfo и передать ему информацию о *регулярных выражениях*, подлежащих компиляции, и о том, куда поместить оттранслированную программу.

40.Теория регулярных выражений. Практика применения регулярных выражений.

Пусть T = {a1, a2, ...an} - алфавит символов. Словом в алфавите T называется последовательность записанных подряд символов, а длиной слова - число его символов. Пустое слово, не содержащее символов, обычно обозначается как e. Алфавит T можно рассматривать как множество всех слов длины 1. Рассмотрим операцию конкатенации над множествами, так, что конкатенация алфавита T с самим собой дает множество всех слов длины 2. Обозначается конкатенация TT как T2. Множество всех слов длины k обозначается - Tk, его можно рассматривать как k -кратную конкатенацию алфавита T. Множество всех непустых слов произвольной длины, полученное объединением всех множеств Tk, обозначается T+, а объединение этого множества с пустым словом называется итерацией языка и обозначается T*. Итерация описывает все возможные слова, которые можно построить в данном алфавите. Любое подмножество слов L(T), содержащееся в T*, называется языком в алфавите T.

Определим класс языков, задаваемых регулярными множествами. Регулярное множество определяется рекурсивно следующими правилами:

пустое множество, а также множество, содержащее пустое слово, и одноэлементные множества, содержащие символы алфавита, являются регулярными базисными множествами;

если множества P и Q являются регулярными, то множества, построенные применением операций объединения, конкатенации и итерации - P>>Q, PQ, P*, Q* - тоже являются регулярными.

Регулярные выражения представляют удобный способ задания регулярных множеств. Аналогично множествам, они определяются рекурсивно:

регулярные базисные выражения задаются символами и определяют соответствующие регулярные базисные множества, например, выражение f задает одноэлементное множество {f} при условии, что f - символ алфавита T;

если p и q - регулярные выражения, то операции объединения, конкатенации и итерации - p+q, pq, p*, q* - являются регулярными выражениями, определяющими соответствующие регулярные множества.

По сути, регулярные выражения - это более простой и удобный способ записи регулярных множеств в виде обычной строки. Каждое регулярное множество, а, следовательно, и каждое регулярное выражение задает некоторый язык L(T) в алфавите T. Этот класс языков - достаточно мощный, с его помощью можно описать интересные языки, но устроены они довольно просто - их можно определить также с помощью простых грамматик, например, правосторонних грамматик. Более важно, что для любого регулярного выражения можно построить конечный автомат, который распознает, принадлежит ли заданное слово языку, порожденному регулярным

выражением. На этом основана практическая ценность регулярных выражений.

С точки зрения практики регулярное выражение задает образец поиска. После чего можно проверить, удовлетворяет ли заданная строка или ее подстрока данному образцу. В языках программирования синтаксис регулярного выражения существенно обогащается, что дает возможность более просто задавать сложные образцы поиска. Такие синтаксические надстройки, хотя и не меняют сути регулярных выражений, крайне полезны для практиков, избавляя программиста от ненужных сложностей. (В Net Framework эти усложнения, на мой взгляд, чрезмерны. Выигрывая в мощности языка, проигрываем в простоте записи его выражений.)

41.Свойства и методы класса Regex и других классов, связанных с регулярными выражениями. Примеры выполнения регулярных выражений.

Класс Regex

Это основной класс, всегда создаваемый при работе с *регулярными выражениями*. Объекты этого класса определяют *регулярные выражения*. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая *регулярное выражение*. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению RegexOptions и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать *регулярное выражение*. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Рассмотрим четыре основных метода *класса Regex*.

Метод Match запускает поиск соответствия. В качестве параметра методу передается строка поиска, где разыскивается первая подстрока, которая удовлетворяет образцу, заданному *регулярным выражением*.В качестве результата метод возвращает объект *класса Match*, описывающий результат поиска. При успешном поиске свойства объекта будут содержать информацию о найденной подстроке.

Метод Matches позволяет разыскать все вхождения, то есть все подстроки, удовлетворяющие образцу. У алгоритма поиска есть важная особенность - разыскиваются непересекающиеся вхождения подстрок. Можно считать, что метод Matches многократно запускает *метод Match*, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск. В качестве результата возвращается объект *MatchCollection*, представляющий коллекцию объектов *Match*.

Метод NextMatch запускает новый поиск, начиная с того места, на котором остановился предыдущий поиск.

Метод Split является обобщением метода Split класса String. Он позволяет, используя образец, разделить искомую строку на элементы. Поскольку образец может быть устроен сложнее, чем простое множество разделителей, то метод Split *класса Regex* эффективнее, чем его аналог класса String.

Класс Regex

Это основной класс, всегда создаваемый при работе с *регулярными выражениями*. Объекты этого класса определяют *регулярные выражения*. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая *регулярное выражение*. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению RegexOptions и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать *регулярное выражение*. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Классы Match и MatchCollection

Как уже говорилось, объекты этих классов создаются автоматически при вызове *методов Match* и Matches. *Коллекция MatchCollection*, как и все коллекции, позволяет получить доступ к каждому ее элементу - объекту *Match*. Можно,

конечно, организовать цикл foreach для последовательного доступа ко всем элементам коллекции.

Классы Group и GroupCollection

Коллекция GroupCollection возвращается при вызове свойства Group объекта Match. Имея эту коллекцию, можно добраться до каждого объекта Group, в нее входящего. *Класс Group* является наследником класса Capture и, одновременно, родителем *класса Match*. От своего родителя он наследует свойства Index, Length и Value, которые и передает своему потомку.

Классы Capture и CaptureCollection

Коллекция CaptureCollection возвращается при вызове свойства Captures объектов *класса Group* и *Match*. *Класс Match* наследует это свойство у своего родителя - *класса Group*.

Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска *регулярные выражения*. В этом случае необходимо будет создать объект класса RegexCompilationInfo и передать ему информацию о *регулярных выражениях*, подлежащих компиляции, и о том, куда поместить оттранслированную программу.

Примеры работы с регулярными выражениями

Полагаю, что примеры дополнят краткое описание возможностей *регулярных выражений* и позволят лучше понять, как с ними работать. Начну с функции FindMatch, которая производит поиск первого вхождения подстроки, соответствующей образцу:

```
static string FindMatch(string str, string strpat)
{
  Regex pat = new Regex(strpat);
  Match match =pat.Match(str);
  string found = "";
  if (match.Success)
  {
          found =match.Value;
          Console.WriteLine("Строка ={0}\tОбразец={1}\
                tНайдено={2}", str,strpat,found);
  }
  return(found);
}//FindMatch
```

В качестве входных аргументов функции передается строка str, в которой ищется вхождение, и строка strpat, задающая образец - *регулярное выражение*. Функция возвращает найденную в результате поиска подстроку. Если соответствия нет, то возвращается пустая строка. Функция начинает свою работу с создания объекта pat *класса Regex*, конструктору которого передается образец поиска. Затем вызывается *метод Match* этого объекта, создающий объект match *класса Match*. Далее анализируются свойства этого объекта. Если соответствие обнаружено, то найденная подстрока возвращается в качестве результата, а соответствующая информация выводится на печать. (Чтобы спокойно работать с классами *регулярных выражений*, я не забыл добавить в начало проекта предложение: using System.Text.RegularExpressions.)

Поскольку запись *регулярных выражений* - вещь, привычная не для всех программистов, я приведу достаточно много примеров:

```
public void TestSinglePat()
{
  //поиск по образцу первого вхождения
  string str,strpat,found;
  Console.WriteLine("Поиск по образцу");
  //образец задает подстроку, начинающуюся с символа a,
  //далее идут буквы или цифры.
  str ="start"; strpat =@"a\w+";
  found = FindMatch(str,strpat);
  str ="fab77cd efg";
  found = FindMatch(str,strpat);
  //образец задает подстроку,начинающуюся с символа a,
  //заканчивающуюся f с возможными символами b и d в середине
  strpat = "a(b|d)*f";  str = "fabadddbdf";
  found = FindMatch(str,strpat);
  //диапазоны и escape-символы
  strpat = "[X-Z]+"; str = "aXYb";
  found = FindMatch(str,strpat);
  strpat = @"\u0058Y\x5A"; str = "aXYZb";
  found = FindMatch(str,strpat);
}//TestSinglePat
```

Некоторые комментарии к этой процедуре.

Регулярные выражения задаются @ -константами, описанными в лекции 14. Здесь они как нельзя кстати.

В первом образце используется последовательность символов \w+, обозначающая, как следует из таблицы 15.1, непустую последовательность латиницы и цифр. В совокупности образец задает подстроку, начинающуюся символом a, за которым следуют буквы или цифры (хотя бы одна). Этот образец применяется к двум различным строкам.

В следующем образце используется символ * для обозначения *итерации*. В целом *регулярное выражение* задает строки, начинающиеся с символа a и заканчивающиеся символом f, между которыми находится возможно пустая последовательность символов из b и d.

Последующие два образца демонстрируют использование диапазонов и escape-последовательностей для представления символов, заданных кодами (в Unicode и шестнадцатеричной кодировке).

Пример "кок и кук"

Этот пример на поиск множественных соответствий навеян словами песни Высоцкого, где говорится, что дикари не смогли распознать, где кок, а где Кук. Наше *регулярное выражение* также не распознает эти слова. Обратите внимание на точку в *регулярном выражении*, которая соответствует любому символу, за исключением символа конца строки. Все слова в строке поиска - кок, кук, кот и другие - будут удовлетворять шаблону, так что в результате поиска найдется множество соответствий.

```
Console.WriteLine("кок и кук");
strpat="(т|к).(т|к)";
str="кок тот кук тут как кот";
FindMatches(str, strpat);
```

42.Две роли класса в ООП. Синтаксис описания класса. Поля и методы класса.

Две роли классов

У класса две различные роли: модуля и типа данных. *Класс - это модуль*, архитектурная единица построения программной системы. Модульность построения - основное свойство программных систем. В ООП программная система, строящаяся по модульному принципу, состоит из классов, являющихся основным видом модуля. Модуль может не представлять собой содержательную единицу - его размер и содержание определяется архитектурными соображениями, а не семантическими. Ничто не мешает построить монолитную систему, состоящую из одного модуля - она может решать ту же задачу, что и система, состоящая из многих модулей.

Вторая роль класса не менее важна. *Класс - это тип данных*, задающий реализацию некоторой абстракции данных, характерной для задачи, в интересах которой создается программная система. С этих позиций классы - не просто кирпичики, из которых строится система. Каждый кирпичик теперь имеет важную содержательную начинку. Представьте себе современный дом, построенный из кирпичей, и дом будущего, где каждый кирпич выполняет определенную функцию: один следит за температурой, другой - за составом воздуха в доме. ОО-программная система напоминает дом будущего.

Состав класса, его размер определяется не архитектурными соображениями, а той абстракцией данных, которую должен реализовать класс. Если вы создаете класс Account, реализующий такую абстракцию как банковский счет, то в этот класс нельзя добавить *поля из класса* Car, задающего автомобиль.

Объектно-ориентированная разработка программной системы основана на стиле, называемом *проектированием от данных*. Проектирование системы сводится к поиску абстракций данных, подходящих для конкретной задачи. Каждая из таких абстракций реализуется в виде класса, которые и становятся модулями - архитектурными единицами построения нашей системы. В основе класса лежит абстрактный тип данных.

В хорошо спроектированной ОО-системе каждый класс играет обе роли, так что каждый модуль нашей системы имеет вполне определенную смысловую нагрузку. Типичная ошибка - рассматривать класс только как архитектурную единицу, объединяя под обложкой класса разнородные *поля* и функции, после чего становится неясным, какой же тип данных задает этот класс.

Синтаксис класса

Ни одна из предыдущих лекций не обходилась без появления классов и обсуждения многих деталей, связанных с ними. Сейчас попробуем сделать некоторые уточнения, подвести итоги и с новых позиций взглянуть на уже знакомые вещи. Начнем с *синтаксиса описания класса*:

[атрибуты][модификаторы]class имя_класса[:список_родителей]
{тело_класса}

Атрибутам будет посвящена отдельная лекция. Возможными модификаторами в объявлении класса могут быть модификаторы new, abstract, sealed, о которых подробно будет говориться при рассмотрении наследования, и четыре *модификатора доступа*, два из которых - private и protected - могут быть заданы только для вложенных классов. Обычно *класс* имеет *атрибут* доступа public, являющийся значением *по* умолчанию. Так что в простых случаях объявление класса выглядит так:

public class Rational {тело_класса}

В теле класса могут быть объявлены: *константы* ; *поля* ; *конструкторы* и *деструкторы* ; *методы* ; события; делегаты; классы (структуры, интерфейсы, перечисления).

Из синтаксиса следует, что классы могут быть вложенными. Такая ситуация - довольно редкая. Ее стоит использовать, когда некоторый *класс* носит вспомогательный характер, разрабатывается в интересах другого класса, и есть полная уверенность, что внутренний *класс* никому не понадобится, кроме класса, в который он вложен. Как уже упоминалось, внутренние классы обычно имеют *модификатор доступа*, отличный от public. Основу любого класса составляют его *конструкторы*, *поля* и *методы*.

Поля класса

Поля класса синтаксически являются обычными переменными (объектами) языка. Содержательно *поля* задают представление той самой абстракции данных, которую реализует класс. *Поля* характеризуют *свойства объектов класса*. Когда создается новый объект класса (в динамической памяти или в стеке), то этот объект представляет собой набор *полей класса*. Два объекта одного класса имеют один и тот же набор *полей*, но разнятся значениями, хранимыми в этих *полях*. Все объекты класса Person могут иметь *поле*, характеризующее рост персоны, но один объект может быть высокого роста, другой - низкого, а третий - среднего роста.

Методы класса

Методы класса синтаксически являются обычными процедурами и функциями языка. Их описание удовлетворяет обычным правилам объявления процедур и функций. Содержательно *методы* определяют ту самую абстракцию данных, которую реализует класс. *Методы* содержат описания *операций*, доступных над объектами класса. Два объекта одного класса имеют один и тот же набор *методов*.

43.Конструкторы и деструкторы.

Конструкторы класса

Конструктор - неотъемлемый компонент класса. Нет классов без *конструкторов*. *Конструктор* представляет собой специальный *метод класса*, позволяющий создавать объекты класса. Одна из синтаксических особенностей этого *метода* в том, что его имя должно совпадать с именем класса. Если программист не определяет *конструктор класса*, то к классу автоматически добавляется *конструктор* по умолчанию - *конструктор*без аргументов.

Разберем в деталях процесс создания *конструктора* с одним аргументом типа string:

- первым делом для сущности создается ссылка со значением null ;
- затем в динамической памяти создается объект - структура данных с *полями*, определяемыми неким классом. *Поля* объекта инициализируются значениями по умолчанию: ссылочные *поля* - значением null, арифметические - нулями, строковые - пустой строкой. Эту работу выполняет *конструктор* по умолчанию, который, можно считать, всегда вызывается в начале процесса создания.
- если *поля класса* проинициализированы, то выполняется инициализация *полей* заданными значениями;
- если вызван *конструктор* с аргументами, то начинает выполняться тело этого *конструктора*.
- На заключительном этапе ссылка связывается с созданным объектом.

Процесс создания объектов становится сложнее, когда речь идет об объектах, являющихся потомками некоторого класса. В этом случае, прежде чем создать сам объект, нужно вызвать *конструктор*, создающий родительский объект.

Зачем классу нужно несколько *конструкторов*? Дело в том, что, в зависимости от контекста и создаваемого объекта, может требоваться различная инициализация его *полей*. Перегрузка *конструкторов* и обеспечивает решение этой задачи.

Конструктор может быть объявлен с атрибутом private. Понятно, что в этом случае внешний пользователь не может воспользоваться им для создания объектов. Но это могут делать *методы класса*, создавая объекты для собственных нужд со специальной инициализацией. Пример такого *конструктора* будет дан позже.

В классе можно объявить статический *конструктор* с атрибутом static. Он вызывается автоматически - его не нужно вызывать стандартным образом. Точный момент вызова не определен, но гарантируется, что вызов произойдет до создания первого объекта класса. Такой *конструктор* может выполнять некоторую предварительную работу, которую нужно выполнить один раз,

например, связаться с базой данных, заполнить значения *статических полей* класса, создать *константы* класса, выполнить другие подобные действия. Статический *конструктор*, вызываемый автоматически, не должен иметь *модификаторов доступа*.

Подводя итоги, можно отметить, что объекты создаются динамически в процессе выполнения программы - для создания объекта всегда вызывается тот или иной *конструктор класса*.

Деструкторы класса

В языке C# у класса может быть *деструктор*, но он не занимается удалением объектов и не вызывается нормальным образом в ходе выполнения программы. Так же, как и статический *конструктор*, *деструктор класса*, если он есть, вызывается автоматически в процессе сборки мусора. Его роль - в освобождении ресурсов, например, файлов, открытых объектом.

Формальное описание *деструктора класса* Person:

```
~Person()
{
  //Код деструктора
}
```

Имя *деструктора* строится из имени класса с предшествующим ему символом ~ (тильда). Как и у статического *конструктора*, у *деструктора* не указывается *модификатор доступа*.

44.Статические поля и методы. Статические конструкторы.

Статические поля и методы класса

Не все *поля* отражаются в структуре объекта. У класса могут быть *поля*, связанные не с объектами, а с самим классом. Эти *поля* объявляются как *статические* с модификатором static. *Статические поля* доступны всем *методам класса*. Независимо от того, какой объект вызвал *метод*, используются одни и те же *статические поля*, позволяя *методу* использовать информацию, созданную другими объектами класса. *Статические поля* представляют общий информационный пул для всех объектов классов, позволяя извлекать и создавать общую информацию.

Аналогично *полям*, у класса могут быть и *статические методы*, объявленные с модификатором static. Такие *методы* не используют информацию о *свойствах конкретных объектов класса* - они обрабатывают общую для класса информацию, хранящуюся в его *статических полях*. Другим частым случаем применения *статических методов* является ситуация, когда класс предоставляет свои сервисы объектам других классов. Таковым является класс Math из библиотеки FCL, который не имеет собственных *полей* - все его *статические методы* работают с объектами арифметических классов.

Статические конструкторы

Статические классы содержат только статические члены, в том числе и конструктор, которые хранятся в памяти в единственном экземпляре. Поэтому создавать экземпляры класса нет смысла.

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором static. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором static (константы и *вложенные типы* классифицируются как *статические элементы* автоматически). Конструктор экземпляра для статического класса задавать запрещается.

45.Поля только для чтения. Закрытые поля . Стратегии доступа к полям класса.

Каждое *поле* имеет *модификатор доступа*, принимающий одно из четырех значений: public, private, protected, internal. Атрибутом доступа по умолчанию является атрибут private. Независимо от значения атрибута доступа, все *поля* доступны для всех *методов класса*. Они являются для *методов класса* глобальной информацией, с которой работают все *методы*, извлекая из *полей* нужные им данные и изменяя их значения в ходе работы. Если *поля* доступны только для *методов класса*, то они имеют атрибут доступа private, который можно опускать. Такие *поля* считаются закрытыми, но часто желательно, чтобы некоторые из них были доступны в более широком контексте. Если некоторые *поля класса* A должны быть доступны для *методов класса* B, являющегося потомком класса A, то эти *поля* следует снабдить атрибутом protected. Такие *поля* называются защищенными. Если некоторые *поля* должны быть доступны для *методов классов* B1, B2 и так далее, дружественных по отношению к классу A, то эти *поля* следует снабдить атрибутом internal, а все дружественные классы B поместить в один проект (assembly). Такие *поля* называются дружественными. Наконец, если некоторые *поля* должны быть доступны для *методов* любого класса B, которому доступен сам класс A, то эти *поля* следует снабдить атрибутом public. Такие *поля* называются общедоступными или открытыми.

Закрытые поля

При объявлении полей так, как показано выше, **мы можем решить, должны ли другие классы иметь к ним доступ**. Иногда очень важно делать поля закрытыми (private) — для того, например, чтобы другой *класс* не помещал неверные значения в поля и не нарушал работу программы. Использование полей private, protected и public определяет степень защиты объектов класса.

- **Private** – "объекты только этого класса могут обращаться к данному полю".
- **Public** – "объекты любого класса могут обращаться к этому полю".
- **Protected** – "только объекты классов-наследников могут обращаться к полю". Если построен класс *Animal*, то другой класс, например, класс *Mammal* (Млекопитающее), может объявить себя наследником класса *Animal*.

Поля только для чтения

Особым случаем реализации концепции *инкапсуляции* в языке объектно-ориентированного программирования C# является механизм *полей*, предназначенных *только для чтения*. Для описания такого рода *полей* в языке C# используется *зарезервированное слово readonly* .

Основные *свойства полей*, предназначенных *только для чтения*, сводятся к следующему. Прежде всего, такие *поля* необходимо инициализировать непосредственно при описании или в конструкторе класса. Кроме того, *значение поля*, предназначенного *только для чтения*, не обязательно должно быть вычислимым в ходе компиляции, а может быть вычислено во *время выполнения* программы. Наконец, *поля*, предназначенные *только для чтения*,

занимают предназначенные для них области памяти (что до определенной степени аналогично *статическим объектам*).

Проиллюстрируем основные *свойства полей*, предназначенных *только для чтения*, следующими примерами фрагментов программ на языке C#. Описание *поля* выглядит следующим образом:

readonly DateTime date;

Доступ к *полю* изнутри класса организуется *по краткому имени* объекта:

... value ... size ... date ...

Доступ к *полю* из других классов (на примере объекта с как конкретизации класса C) реализуется с указанием *полного имени* объекта:

c = new C();

... c.value ... c.size ... c.date ...

Доступ к *статическим полям* изнутри класса *по краткому имени* и из других классов *по полному имени* соответственно реализуется *по* аналогии с *полями только для чтения*:

... defaultColor ... scale ...

... Rectangle.defaultColor ...

Rectangle.scale ...

Манипулирование *статическими полями* и *методами* в языке программирования C# осуществляется аналогично случаям *статических полей* и *констант*.

Стратегии доступак полям

Методы, называемые свойствами (Properties), представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами. При работе со свойствами объекта (полями) часто нужно решить, какой модификатор доступа использовать, чтобы реализовать нужную ***стратегию доступа к полю класса***. Пять наиболее употребительных стратегий:

чтение, запись (Read, Write);

чтение, запись при первом обращении (Read, Write-once);

только чтение (Read-only);

только запись (Write-only);

ни чтения, ни записи (Not Read, Not Write).

Открытость свойств (атрибут public) позволяет реализовать только первую стратегию. В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через методы. Для эффективности этого процесса и введены специальные методы-свойства.

46.Процедуры(методы) свойства. Индексаторы. Примеры

Методы-свойства

Методы, называемые свойствами (Properties), представляют специальную синтаксическую конструкцию, предназначенную для обеспечения эффективной работы со свойствами. При работе со свойствами объекта (полями) часто нужно решить, какой модификатор доступа использовать, чтобы реализовать нужную стратегию доступа к полю класса. Перечислю пять наиболее употребительных стратегий:

чтение, запись (Read, Write);

чтение, запись при первом обращении (Read, Write-once);

только чтение (Read-only);

только запись (Write-only);

ни чтения, ни записи (Not Read, Not Write).

Открытость свойств (атрибут public) позволяет реализовать только первую стратегию. В языке C# принято, как и в других объектных языках, свойства объявлять закрытыми, а нужную стратегию доступа организовывать через методы. Для эффективности этого процесса и введены специальные методы-свойства.

Синтаксис методов-свойств

Имя метода обычно близко к имени поля (например, Name). Тело свойства содержит два метода - get и set, один из которых может быть опущен. Метод get возвращает значение закрытого поля, метод set - устанавливает значение, используя передаваемое ему значение в момент вызова, хранящееся в служебной переменной со стандартным именем value. Поскольку get и set - это обычные процедуры языка, то программно можно реализовать сколь угодно сложные стратегии доступа.

Индексаторы

Метод-индексатор является обобщением метода-свойства. Он обеспечивает доступ к закрытому полю, представляющему массив. Объекты класса индексируются по этому полю.

Синтаксически объявление индексатора - такое же, как и в случае свойств, но методы get и set приобретают аргументы по числу размерности массива, задающего индексы элемента, значение которого читается или обновляется. Важным ограничением является то, что у класса может быть только индексатор со стандартным именем this. Как и любые другие методы индексатор может быть перегруженным. Так что если среди полей класса есть несколько массивов одной размерности, то индексация объектов может быть выполнена только по одному из них.

Добавим в класс Person свойство children, задающее детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит индексатор:

```
const int Child_Max = 20; //максимальное число детей
```

```
Person[] children = new Person[Child_Max];
int count_children=0; //число детей
public Person this[int i] //индексатор
{get {if (i>=0 && i< count_children)return(children[i]);
                else return(children[0]);}
 set
 {if (i==count_children && i< Child_Max)
        {children[i] = value; count_children++;}}}
```

Имя у индексатора - this, в квадратных скобках в заголовке перечисляются индексы. В методах get и set, обеспечивающих доступ к массиву children, по которому ведется индексирование, анализируется корректность задания индекса. Закрытое поле count_children, хранящее текущее число детей, доступно только для чтения благодаря добавлению соответствующего метода-свойства.

47.Понятие развёрнутого и ссылочного типа. Структуры – реализация развёрнутых классов.

Развернутые и ссылочные типы

Рассмотрим объявление объекта класса T с инициализацией:

T x = new T();

Напомню, как выполняется этот оператор. В памяти создается *объект* типа T, основанного на классе T, и сущность x связывается с этим объектом. Сущность, не прошедшая инициализацию (явную или неявную), не связана ни с одним объектом, а потому не может использоваться в вычислениях - у нее нет полей, хранящих значения, она не может вызывать *методы класса*. Объектам нужна *память*, чтобы с ними можно было работать. Есть две классические стратегии выделения памяти и связывания объекта, создаваемого в памяти, и сущности, объявленной в тексте.

Определение 1. *Класс* T относится к *развернутому типу*, если *память* отводится сущности x ; *объект* разворачивается на памяти, жестко связанной с сущностью.

Определение 2. *Класс* T относится к *ссылочному типу*, если *память* отводится объекту; сущность x является ссылкой на *объект*.

Для *развернутого типа* характерно то, что каждая сущность ни с кем не разделяет свою *память*; сущность жестко связывается со своим объектом. В этом случае сущность и *объект* можно и не различать, они становятся неделимым понятием. Для *ссылочных типов* ситуация иная - несколько сущностей могут ссылаться на один и тот же *объект*. Такие сущности разделяют *память* и являются разными именами одного объекта. Полезно понимать разницу между сущностью, заданной ссылкой, и объектом, на который в текущий момент указывает *ссылка*.

Развернутые и *ссылочные типы* порождают две различные семантики присваивания - развернутое *присваивание* и ссылочное *присваивание*. Рассмотрим *присваивание*:

y = x;

Когда сущность y и *выражение* x принадлежат *развернутому типу*, то при присваивании изменяется *объект*. Значения полей объекта, связанного с сущностью y, изменяются, получая значения полей объекта, связанного с x. Когда сущность y и *выражение* x принадлежат *ссылочному типу*, то изменяется *ссылка*, но не *объект*. *Ссылка* y получает *значение* ссылки x, и обе они после присваивания указывают на один и тот же *объект*.

Язык программирования должен позволять программисту в момент определения класса указать, к *развернутому* или *ссылочному типу* относится *класс*. К сожалению, язык C# не позволяет этого сделать напрямую - в нем у класса нет модификатора, позволяющего задать *развернутый* или *ссылочный тип*.

Структуры – реализация развёрнутых классов.

Структура - это частный случай класса. Исторически структуры используются в языках программирования раньше классов. В языке C++ возможности структур были существенно расширены и они стали настоящими классами, хотя и с некоторыми ограничениями. В языке C# - наследнике C++ - сохранен именно такой подход к структурам.

Чем следует руководствоваться, делая выбор между структурой и классом?

если необходимо отнести **класс к развернутому типу, делайте его структурой ;**

если у класса число полей относительно невелико, а число возможных объектов относительно велико, делайте его структурой. В этом случае память объектам будет отводиться в стеке, не будут создаваться лишние ссылки, что позволит повысить эффективность работы;

в остальных случаях проектируйте настоящие классы.

Поскольку на структуры накладываются дополнительные ограничения, то может возникнуть необходимость в компромиссе - согласиться с ограничениями и использовать структуру либо пожертвовать развернутостью и эффективностью и работать с настоящим классом. Стоит отметить: когда говорится, что все встроенные типы - int и другие - представляют собой классы, то, на самом деле, речь идет о классах, реализованных в виде структур.

Синтаксис структур

Синтаксис объявления структуры аналогичен синтаксису объявления класса:

[атрибуты][модификаторы]struct имя_структуры[:список_интерфейсов]
{тело_структуры}

Изменения в синтаксисе в сравнении с синтаксисом класса

- ключевое слово class изменено на слово struct ;
- список родителей, который для классов, наряду с именами интерфейсов, мог включать имя родительского класса, заменен списком интерфейсов. Для структур не может быть задан родитель (класс или структура). Заметьте, структура может наследовать интерфейсы;
- для структур неприменимы модификаторы abstract и sealed. Причиной является отсутствие механизма наследования.

Все, что может быть вложено в тело класса, может быть вложено и в тело структуры: поля, методы, конструкторы и прочее, включая классы и интерфейсы.

48.Синтаксис структур. Сравнение структур и классов. Встроенные структуры

Структуры

Рассмотрим теперь более подробно вопросы описания *структур,* их *синтаксиса*, семантики и тех особенностей, что отличают их от классов.

Синтаксис структур

Синтаксис объявления структуры аналогичен синтаксису объявления класса:

[атрибуты][модификаторы]struct имя_структуры[:список_интерфейсов]
{тело_структуры}

Изменения в *синтаксисе* в сравнении с синтаксисом класса

- ключевое слово class изменено на слово struct ;
- список родителей, который для классов, наряду с именами интерфейсов, мог включать имя родительского класса, заменен списком интерфейсов. Для *структур* не может быть задан родитель (класс или *структура*). Заметьте, *структура* может наследовать интерфейсы;
- для *структур* неприменимы модификаторы abstract и sealed. Причиной является отсутствие механизма наследования.

Все, что может быть вложено в тело класса, может быть вложено и в тело *структуры*: поля, методы, конструкторы и прочее, включая классы и интерфейсы.

Аналогично классу, *структура* может иметь статические и не статические поля и методы, может иметь несколько конструкторов, в том числе статические и закрытые конструкторы. Для *структур* можно создавать собственные константы, используя поля с атрибутом readonly и статический конструктор. *Структуры* похожи на классы по своему описанию и ведут себя сходным образом, хотя и имеют существенные различия в семантике присваивания.

Перечислим *ограничения, накладываемые на структуры*.

- Самое серьезное ограничение связано с *ограничением наследования*. У *структуры* не может быть наследников. У *структуры* не может быть задан родительский класс или родительская *структура*. Конечно, всякая *структура*, как и любой класс в C#, является наследником класса Object, наследуя все свойства и методы этого класса. *Структура* может быть наследником одного или нескольких интерфейсов, реализуя методы этих интерфейсов.
- Второе серьезное ограничение связано с процессом создания объектов. Пусть T - *структура*, и дано объявление без инициализации - T x. Это объявление корректно, в результате будет создан объект без явного вызова операции new. Сущности x будет отведена память, и на этой памяти будет развернут объект. Но поля объекта не будут инициализированы и, следовательно, не будут доступны для использования в вычислениях. Об этих особенностях подробно говорилось при рассмотрении значимых типов. В этом отношении все, что верно для типа int, верно и для всех *структур*.

- Если при объявлении класса его поля можно инициализировать, что найдет отражение при работе конструктора класса, то поля *структуры*не могут быть инициализированы.
- Конструктор по умолчанию у *структур* имеется, при его вызове поля инициализируются значениями по умолчанию. Этот конструктор нельзя заменить, создав собственный конструктор без аргументов.
- В конструкторе нельзя вызывать методы класса. Поля *структуры* должны быть проинициализированы до вызова методов.

Встроенные структуры

Все значимые типы языка реализованы *структурами*. В библиотеке FCL имеются и другие *встроенные структуры*. Рассмотрим в качестве примера *структуры* Point, PointF, Size, SizeF и Rectangle, находящиеся в пространстве имен System.Drawing и активно используемые при работе с графическими объектами. Первые четыре *структуры* имеют два открытых поля X и Y (Height и Width), задающие для точек - *структур* Point и PointF - координаты, целочисленные или в форме с плавающей точкой. Для размеров - *структур* Size и SizeF - они задают высоту и ширину, целочисленными значениями или в форме с плавающей точкой. *Структуры* Point и Size позволяют задать прямоугольную область - *структуру* Rectangle. Конструктору прямоугольника можно передать в качестве аргументов две *структуры* - точку, задающую координаты левого верхнего угла прямоугольника, и размер - высоту и ширину прямоугольника.

Между четырьмя *структурами* определены взаимные преобразования: точки можно преобразовать в размеры и наоборот, сложение и вычитание определено над точками и размерами, но не над точками, плавающий тип которых разными способами можно привести к целому. Ряд операций над этими *структурами* продемонстрирован в следующем примере:

```
public void TestPointAndSize()
{
  Point pt1 = new Point(3,5), pt2 = new Point(7,10), pt3;
  PointF pt4 = new PointF(4.55f,6.75f);
  Size sz1 = new Size(10,20), sz2;
  SizeF sz3 = new SizeF(10.3f, 20.7f);
  pt3 = Point.Round(pt4);
  sz2 = new Size(pt1);
  Console.WriteLine ("pt1: " + pt1);
  Console.WriteLine ("sz2 =new Size(pt1): " + sz2);
  Console.WriteLine ("pt4: " + pt4);
  Console.WriteLine ("pt3 =Point.Round(pt4): " + pt3);
  pt1.Offset(5,7);
  Console.WriteLine ("pt1.Offset(5,7): " + pt1);
  Console.WriteLine ("pt2: " + pt2);
  pt2 = pt2+ sz2;
  Console.WriteLine ("pt2= pt2+ sz2: " + pt2);
}//TestPointAndSize
```

49. Перечисление – частный случай класса.

Перечисление - это частный случай класса, *класс*, заданный без собственных методов. *Перечисление* задает конечное множество возможных значений, которые могут получать объекты класса *перечисление*. Поскольку у *перечислений* нет собственных методов, то *синтаксис объявления*этого класса упрощается - остается обычный заголовок и тело класса, содержащее *список* возможных значений. Вот формальное *определение*синтаксиса *перечислений*:

[атрибуты][модификаторы]enum имя_перечисления[:базовый класс]

{список_возможных_значений}

Модификаторами могут быть четыре известных модификатора доступа и модификатор *new*. *Ключевое слово* ***enum*** говорит, что определяется частный случай *класса* - *перечисление*. *Список* возможных значений задает те значения, которые могут получать объекты этого класса. Возможные значения должны быть идентификаторами; но допускаются в их написании и буквы русского алфавита. Можно указать также базовый для *перечисления класс*.

Дело в том, что значения, заданные списком, проецируются на плотное *подмножество* базового класса. Реально значения объектов *перечисления* в памяти задаются значениями базового класса, так же, как значения класса bool реально представлены в памяти нулем и единицей, а не константами true и false, удобными для их использования программистами в тексте программ. *По* умолчанию, базовым классом является *класс* int, а *подмножество* проекции начинается с нуля. Но при желании можно изменить *интервал* представления и сам *базовый класс*. Естественно, на *базовый класс* накладывается ограничение. Он должен быть одним из *встроенных классов*, задающих счетное множество (int, byte, long, другие счетные типы). Единственное *исключение* из этого правила - нельзя выбирать *класс* char в качестве базового класса. Как правило, принятый *по* умолчанию выбор базового класса и его подмножества вполне приемлем в большинстве ситуаций.

Приведу примеры объявлений *классов-перечислений*:

public enum Profession{teacher, engineer, businessman};

public enum MyColors {red, blue, yellow, black, white};

Вот несколько моментов, на которые следует обратить внимание при объявлении *перечислений*:

- как и другие классы, *перечисления* могут быть объявлены непосредственно в пространстве имен проекта или могут быть вложены в описание класса. Последний вариант часто применяется, когда *перечисление* используется в одном классе и имеет атрибут доступа private ;
- константы разных *перечислений* могут совпадать, как в *перечислениях* MyColors и TwoColors. Имя константы всегда уточняется именем *перечисления* ;
- константы могут задаваться словами русского языка, как в *перечислении* Rainbow ;
- разрешается задавать базовый класс *перечисления*. Для *перечисления* Days базовым классом задан класс long ;
- разрешается задавать не только базовый класс, но и указывать начальный элемент подмножества, на которое проецируется множество значений *перечисления*. Для *перечисления* Sex в качестве базового класса выбран класс byte, а подмножество значений начинается с 1, так что хранимым значением константы man является 1, а woman - 2.

50. Особенности перечислений. Примеры.

Рассмотрим теперь пример *работы с объектами - экземплярами различных перечислений*:

```
public void TestEnum()
{
  //MyColors color1 = new MyColors(MyColors.blue);
  MyColors color1= MyColors.white;
  TwoColors color2;
  color2 = TwoColors.white;
  //if(color1 != color2) color2 = color1;
  if(color1.ToString() != color2.ToString())
        Console.WriteLine ("Цвета разные: {0}, {1}",
              color1, color2);
  else Console.WriteLine("Цвета одинаковые: {0},
        {1}",color1, color2);
  Rainbow color3;
  color3 = (Rainbow)3;
  if (color3 != Rainbow.красный)color3 =Rainbow.красный;
  int num = (int)color3;
  Sex who = Sex.man;
  Days first_work_day = (Days)(long)1;
  Console.WriteLine ("color1={0}, color2={1},
        color3={2}",color1, color2, color3);
  Console.WriteLine ("who={0}, first_work_day={1}",
        who,first_work_day);
}
```

Данный пример иллюстрирует следующие **особенности** *работы с объектами перечислений*:

- объекты *перечислений* нельзя создавать в объектном стиле с использованием операции new, поскольку *перечисления* не имеют конструкторов;
- объекты можно объявлять с явной инициализацией, как color1, или с отложенной инициализацией, как color2. При объявлении без явной инициализации объект получает значение первой константы *перечисления*, так что color2 в момент объявления получает значение black ;
- объекту можно присвоить значение, которое задается константой *перечисления*, уточненной именем *перечисления*, как для color1 и color2. Можно также задать значение базового типа, приведенное к типу *перечисления*, как для color3 ;
- нельзя сравнивать объекты разных *перечислений*, например color1 и color2, но можно сравнивать строки, возвращаемые методом ToString, например color1.ToString() и color2.ToString() ;
- существуют явные взаимно обратные преобразования констант базового типа и констант *перечисления* ;
- Метод ToString, наследованный от класса Object, возвращает строку, задающую константу *перечисления*.