Плывите сосиски (сосите плывиски))))

[**===1. Системы контроля версий. Классификация. Git, основные возможности**](#_7amqgnhnumnw) **5**

[**===2. Состав. NET Framework. Структура среды выполнения CLR.**](#_274wg0q08mu4) **7**

[**===3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.**](#_7tvslk3ztz37) **8**

[**===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.**](#_xgzgdd6jsabh) **9**

[**===5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение**](#_3882iqo8upx) **11**

[**===6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.**](#_h6bohmrzr536) **13**

[**===7. Неявная типизация – назначение и использование.**](#_8sgya1fc66n) **16**

[**===8. Массивы C# одномерные, прямоугольные и ступенчатые.**](#_bnh17vquy2nd) **17**

[**===9. Понятие кортежей. Свойства, создание**](#_rweq1rux0lyi) **18**

[**===10. Принципы объектно-ориентированного программирования.**](#_7rzj8rakmwfi) **19**

[**===11. Класс. Элементы класса. Свойства и индексаторы.**](#_otg05e818ij8) **20**

[**===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.**](#_uni0yd1f0j2) **22**

[**===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.**](#_5kxncni0hefs) **22**

[**===14. Класс. Конструкторы и их свойства. Деструкторы**](#_ywnpeel3ri9s) **24**

[**===15. Класс и методы System.Object.**](#_57bzabi2jrq) **25**

[**===16. Статические методы и статические конструкторы класса.**](#_71yb8lmg42hk) **26**

[**===17. Статические классы. Методы расширения и правила их определения.**](#_qv9nofwk0t69) **27**

[**===18. Анонимные типы.**](#_60qkucfr2yl5) **28**

[**===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.**](#_1u69n72sibuu) **29**

[**===20. Перегрузка методов и операторов. Правила перегрузки операторов.**](#_8miqlfn98c30) **31**

[**===21. Операции преобразования типа. Явная и неявная форма. Ограничения.**](#_9aobmmwrtqvk) **32**

[**===22. Вложенные типы. Вложенные объекты**](#_m0pq2d1s4obw) **34**

[**===23. Правила наследования C#.**](#_frerxjds3t6b) **35**

[**===24. Сокрытие имен при наследовании. Обращение к скрытым членам**](#_laeumnhr3wop) **36**

[**===25. Использование операций is и as**](#_h88s66qicypi) **37**

[**===26. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.**](#_2sa4szet7kui) **38**

[**===27. Понятие раннего и позднего связывания.**](#_kdxxum623v4e) **39**

[**===28. Абстрактные классы и методы. Бесплодные классы.**](#_atk643nh56jv) **40**

[**===29. Структуры в C#.**](#_4b1h4s712ypy) **41**

[**===30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.**](#_d5ybnbgh036x) **43**

[**===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.**](#_thhomwlifctz) **44**

[**===32. Ковариантность интерфейсов. Контравариантность интерфейсов**](#_j96ls140y8wt) **46**

[**===33. Стандартные интерфейсы .NET. Назначение и применение.**](#_t48awy2ddmxv) **48**

[**===34. Исключительные ситуации. Генерация и повторная генерация исключений.**](#_ewec33w6eul4) **50**

[**===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений**](#_ndn4ndy97317) **51**

[**===36. Обобщения (generics). Свойства обобщений.**](#_gfngvgxe1tr1) **53**

[**===37. Концепция ограничений обобщений. Статические члены обобщений.**](#_w7qbmyvasi5y) **55**

[**===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.**](#_7zkw8kq8i699) **58**

[**===39. Анонимные функции. Лямбда-выражения.**](#_jr1ez2j2k1ns) **60**

[**===40. Обобщённые делегаты .NET. Action, Func, Predicate**](#_ckt8hlvf6d4) **62**

[**===41. События и делегаты.**](#_f1wf07wd8vjp) **63**

[**===42. Стандартные коллекции .NET. Типы коллекций.**](#_1u6hl884n2tp) **65**

[**===43. Стандартные интерфейсы коллекций.**](#_3tucywn9cyni) **67**

[**===44. IEnumerable и IEnumerator**](#_fi8us788c9bv) **69**

[**===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.**](#_r63o2b7f2pm8) **69**

[**===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy**](#_ubdve0sx3jgf) **73**

[**===47. Рефлексия. System Type.**](#_41ecczxgjq35) **77**

[**===48. Классы для работы с файловой системой.**](#_lhpsvmqqyxh8) **78**

[**===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.**](#_3zh4cicqfn0l) **80**

[**===50. Классы адаптеры потоков.**](#_4qy9nlw3s8wu) **82**

[**===51. Сериализация. Форматы сериализации.**](#_bl7a1epyecag) **85**

[**===52. Сериализация контрактов данных. интерфейс ISerializable.**](#_kmh5rs4e93o8) **88**

[**===53. Атрибуты. Создание собственного атрибута.**](#_8zkew264v0mm) **91**

[**===54. Процесс. Домен приложений. Поток выполнения.**](#_949jjdrjbsql) **93**

[**===55. Создание потоков , классы приоритетов. Состояния потоков**](#_hyvcaqg54ntl) **96**

[**===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore**](#_ampxa5g702ix) **99**

[**===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.**](#_oluv7xv50tyg) **104**

[**===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.**](#_xttpk6gc4db0) **106**

[**===59. Параллелизм при императивной обработке данных. Класс Parallel**](#_cis53sjzzzlj) **108**

[**===60. Асинхронные методы. async и await**](#_h9j54ilahosy) **111**

[**===61. Проектирование отношений. Агрегация, композиция и ассоциация**](#_ayhzpsv76cru) **112**

Вопросы

# ===1. Системы контроля версий. Классификация. Git, основные возможности

Система контроля версий (СКВ) — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. 3 типа:

* Локальные (Revision control System , RCS)
* Централизованные (Centralized version control system, CVCS)
* Распределенные (Distributed version control system, DVCS)

**Локальные системы контроля версий**

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию. Однако можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели. Программисты разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий. Система RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

**Репозиторий** - хранилище (место) , где хранятся и поддерживаются какие-либо данные.

**Централизованные системы контроля версий**

Следующая серьёзная проблема — это необходимость взаимодействовать с другими разработчиками. Были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

**Распределённые системы контроля версий**

В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

**Git. Три состояния**

**Git** – это распределённая система управления версиями файлов.

Особенностью Git является то, что работа над версиями проекта может происходить не в хронологическом порядке. Разработка может вестись в нескольких параллельных ветвях, которые могут сливаться и разделяться в любой момент проектирования.

Файлы в Git могут находится в 3х состояниях:

Git имеет три основных состояния, в которых могут находиться ваши файлы:

* зафиксированном (**committed**) - файл уже сохранён в вашей локальной базе
* изменённом (**modified**) - файлы, которые поменялись, но ещё не были зафиксированы
* подготовленном (**staged**) - изменённые файлы, отмеченные для включения в следующий коммит

**Основные возможности**

**Основные: git init, git add, git commit, git status.**

**Ветвление: Ветвления –** отклонение от основной линии разработки, после которого работа перестает затрагивать основную линию. Команда **git branch** — это своего рода "менеджер веток". Она умеет перечислять ваши ветки, создавать новые, удалять и переименовывать их. Скажи про Git checkout.

Команда **git merge** используется для **слияния** одной или нескольких веток в текущую. Затем она устанавливает указатель текущей ветки на результирующий коммит.

**С удалённым репозиторием:** Команда **git fetch** связывается с удалённым репозиторием и забирает из него все изменения, которых у вас пока нет и сохраняет их локально.

Команда **git pull** работает как комбинация команд **git merge + git fetch**, т.е. Git вначале забирает изменения из указанного удалённого репозитория, а затем пытается слить их с текущей веткой.

Команда **git push** используется для установления связи с удалённым репозиторием, вычисления локальных изменений отсутствующих в нём, и собственно их передачи в вышеупомянутый репозиторий.

# ===2. Состав. NET Framework. Структура среды выполнения CLR.

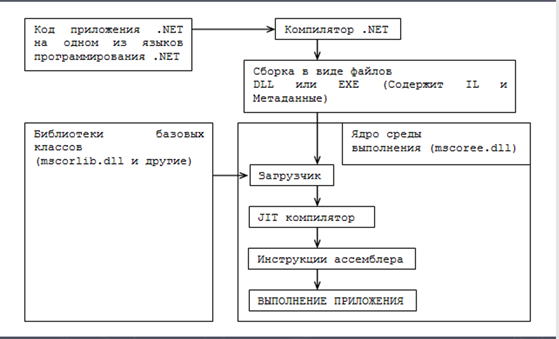
**.Net Framework** – программная платформа. Обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows. Ее компоненты:

* **CLR** (Common Language Runtime) – общеязыковая среда исполнения, виртуальная машина на которой исполняются все приложения, работающие в среде .Net. Обеспечивает выполнение сборки (управление памятью, загрузка сборок, безопасность, обработка исключений, синхронизация). CLR - реализация спецификации CLI (англ. Common Language Infrastructure), спецификации общеязыковой инфраструктуры компании Microsoft.
  + **JIT** – компилятор (Just in time)
* **FCL** (Framework Class Library) – объектно-ориентированная библиотека классов, интерфейсов и системы типов(тип-значения)
  + **CLS** (common language specification) – общеязыковая спецификация, предназначенная для разработчиков компиляторов
  + **CTS** (common type systems) – спецификация типов, которые должны поддерживаться всеми языками, ориентированными на CLR. Microsoft выпустил несколько компиляторов соответствующих этой спецификации: C++, C#, VB, .Net, JScript

Проблемы, решаемые .Net:

* Интеграция языков программирования - CLS и CTS содержат спецификации языков
* Работа на многих платформах - т.к. код генерируется на CIL, приложение должно работать везде, где установлены и работают CLR и FCL
* Упрощенное повторное использование кода - код на одном языке можно использовать в программе на другом языке
* Автоматическое управление памятью - автоматически отслеживается использование ресурсов, есть сборщик мусора
* Проверка безопасности типов - практически исключена возможность записать (стереть) данные в область памяти, которая для этого не предназначена. Нет возможности передать управление в произвольную точку.
* Единый принцип обработки сбоев - для обработки ошибок и сбоев в CLR используется только механизм исключений

Структура CLR:



# ===3. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.

**Portable Executable** — формат исполняемых файлов, объектного кода и динамических библиотек, используемый в 32- и 64-разрядных версиях ОС Windows. Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта API функций, данные для управления ресурсами и данные локальной памяти потока (TLS).

**Сборка** является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является **файл .exe или .dll** (в зависимости от выбранных настроек), который **называется сборкой** приложения.

Исполнение сборки:

1. CLR ищет типы данных и загружает во внутренние структуры
2. Для каждого метода CLR заносит адрес внутренней ЦЛР функции JitCompiler
3. JitCompiler ищет в метаданных соответствующей сборки IL- код вызываемого метода, проверяет и компилирует IL-код в машинные команды
4. Они хранятся в динамически выделенном блоке памяти.
5. JitCompiler Заменяет адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды
6. JitComplier передает управление коду в этом блоке памяти

**Common Intermediate Language (CIL)** — «высокоуровневый ассемблер» виртуальной машины .NET. Промежуточный язык для платформы .NET Framework. JIT-компилятор CIL является частью CLR.

Все компиляторы, поддерживающие платформу .NET, должны транслировать код с языков высокого уровня платформы .NET на язык CIL. В частности, код на языке CIL генерируют все компиляторы .NET фирмы «Microsoft», входящие в среду разработки Microsoft Visual Studio. Язык CIL также нередко называют просто IL — буквально «промежуточный язык».

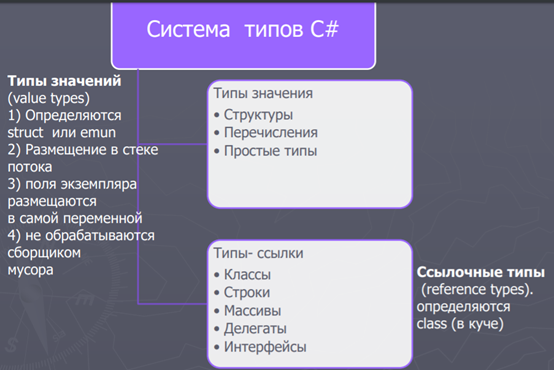
# ===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.

**CTS** (общая система типов) представляет собой формальную спецификацию, в которой описано то, как должны быть определены типы для того, чтобы они могли обслуживаться в CLR-среде. CTS описывает не просто примитивные типы данных, а целую развитую иерархию типов.

**Типы данных С#:**

* **Типы – значения** – структуры, перечисления и простые типы. Размещаются в стеке потока. Не обрабатываются сборщиком мусора.
* **Ссылочные типы** – классы, строки, делегаты, массивы, интерфейсы. Определяются в куче.

Типы данных, которые поддерживаются компилятором напрямую, называются **примитивными** или встроенными.





# ===5. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение

**Упаковкой (boxing)** называется процесс преобразования типа значения в тип System.Object или в тип интерфейса, который реализуется данным типом значением:

1. в управляемой куче выделяется память 2. поля копируются 3. возвращается адрес объекта

Int32 x = 5;

Object o = x; // Упаковка x; o ссылается на упакованный объект

**Распаковка (unboxing) -** получение указателя на исходный значимый тип (поля данных), содержащийся в объекте.

Объекты значимого типа существуют в двух формах: неупакованной (unboxed) и упакованной (boxed). Ссылочные типы бывают только в упакованной форме.

int x = 5;

Object o = x; // Упаковка x

byte m =(byte)o; // Генерируется InvalidCastException

1. Если переменная, содержащая ссылку на упакованный значимый тип, равна null, генерируется исключение NullReferenceException.

2. Если ссылка указывает на объект, не являющийся упакованным значением требуемого значимого типа, генерируется исключение InvalidCastException

int x = 5;

Object o = x; // Упаковка x

byte m = (byte)(int)o; // Распаковка, а затем приведение типа

**Назначение:** ►позволяет использовать типы-значения в коллекциях ( где элементы являются элементами типа object) ►внутренний механизм, который обеспечивает возможность вызывать для типов-значений, подобных int и struct, методы Object.

Одно из отличий ссылочных типов от типов значений состоит в том,

что переменные ссылочных типов могут принимать значение null.

object o = null;

string s = null;

Переменные значимых типов мы так задать не сможем, а если попробуем написать что-то вроде int x = null;, то получим ошибку. Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null.

Для этого надо использовать знак вопроса ? после типа значений. Например:

int? z = null;

bool? enabled = null;

Запись ? является упрощенной формой использования структуры System.Nullable<T>. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в конкретной задача уже подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

int? z1 = 5;

bool? enabled1 = null;

Double? d1 = 3.3;

Nullable<int> z2 = 5;

Nullable<bool> enabled2 = null;

Nullable<System.Double> d2 = 3.3;

Для всех типов Nullable определено **два свойства**: Value, которое представляет значение объекта, и HasValue, которое возвращает true, если объект Nullable хранит некоторое значение.

Причем свойство Value хранит объект того типа, которым типизируется Nullable:

int? x = 7;

Console.WriteLine(x.Value); // 7

Nullable<State> state = new State() { Name = "Narnia" };

Console.WriteLine(state.Value.Name); // Narnia

Однако если мы попробуем получить значение переменной, которая равна null, то мы столкнемся с ошибкой.

Преобразование типов Nullable

явное преобразование от T? к T

int? x1 = null;

if(x1.HasValue)

{

I nt x2 = (int)x1;

}

неявное преобразование от T к T?

int x1 = 4;

int? x2 = x1;

Оператор ?? называется оператором **null-объединения.** Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null. Пример:

int? x = null;

int y = x ?? 100; // равно 100, так как x равен null

int? z = 200;

int t = z ?? 44; // равно 200, так как z не равен null

# ===6. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.

**Тип string** предназначен для работы со строками символов в кодировке Unicode. Ему соответствует базовый класс System.String библиотеки .NET.

Создавать строки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса String:

string s; // инициализация отложена

string s1 = "hello";

string s2 = null;

string s3 = new String('a', 6); // результатом будет строка "aaaaaa"

string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});

char[] a = { '0', '0', '0' }; // создание массива символов

**Операции для строк** ► присваивание (=); ► проверка на равенство содержимого (==); ► проверка на неравенство (!=); ► обращение по индексу ([]); ► сцепление (конкатенация) строк (+) ► <,>, >=,<= - сравнивают ссылки!!!!!!!!

Для приведения строки к верхнему и нижнему регистру используются соответственно функции **ToUpper() и ToLower().**

**Строки равны**, если имеют одинаковое количество символов и совпадают посимвольно.

Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения.

Строки типа string относятся к **неизменяемым типам данных.**

Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

**Строковые литералы**

String path;

path = "C:\\Windows\\regedit.exe"; //верно

path = @"C:\Windows\regedit.exe"; //верно, наиболее предпочтительно

path = "C:/Windows/regedit.exe"; //верно

path = "C:\Windows\regedit.exe"; //неверно, специального символа C# строки являются объектами

**Пустые строки и строки null**

**Пустая строка** — экземпляр объекта System.String, содержащий 0 символов:

string s = "";

Для пустых строк можно вызывать методы.

**Строки со значениями null** не ссылаются на экземпляр объекта System.String, попытка вызвать метод для строки null вызовет исключение NullReferenceException. строки null можно использовать в операциях объединения и сравнения с другими строками.

**Форматированные ввод\вывод**(Преобразует значения объектов в строки на основе указанных форматов и вставляет их в другую строку):

Decimal pricePerOunce = 17.36m;

String s = String.Format("The current price is {0} per ounce.", pricePerOunce);

// Result: The current price is 17.36 per ounce.

Decimal pricePerOunce = 17.36m;

String s = String.Format("The current price is {0:C2} per ounce.", pricePerOunce);

// Result if current culture is en-US:

// The current price is $17.36 per ounce.

**Основные методы строк**

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* **Compare:** сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal:** сравнивает две строки без учета локали
* **Contains:** определяет, содержится ли подстрока в строке
* **Concat:** соединяет строки
* **CopyTo:** копирует часть строки или всю строку в другую строку
* **EndsWith:** определяет, совпадает ли конец строки с подстрокой
* **Format:** форматирует строку
* **IndexOf:** находит индекс первого вхождения символа или подстроки в строке
* **Insert:** вставляет в строку подстроку
* **Join:** соединяет элементы массива строк
* **LastIndexOf:** находит индекс последнего вхождения символа или подстроки в строке
* **Replace:** замещает в строке символ или подстроку другим символом или подстрокой
* **Split:** разделяет одну строку на массив строк
* **Substring:** извлекает из строки подстроку, начиная с указанной позиции
* **ToLower:** переводит все символы строки в нижний регистр
* **ToUpper:** переводит все символы строки в верхний регистр
* **Trim:** удаляет начальные и конечные пробелы из строки

**Класс StringBuilder**

► Любые модификации строки происходят внутри блока памяти Length - длина строки Capacity - максимальная длина строки.

# ===7. Неявная типизация – назначение и использование.

Локальные переменные можно объявлять без указания конкретного типа. **Ключевое слово var** указывает, что компилятор должен вывести тип переменной из выражения справа от оператора инициализации. Выведенный тип может быть встроенным, анонимным. Оператор может применяться только в объявлении локальной переменной. Неявно типизированные локальные переменные не допускают множественного объявления. При объявлении обязательна инициализация.

var i = 5;

// s is compiled as a string

var s = "Hello";

// a is compiled as int[]

var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>

// or perhaps IQueryable<Customer>

var expr =

from c in customers

where c.City == "London"

select c;

Эти переменные подобны обычным, однако они имеют некоторые ограничения:

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать.

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null.

**Тип данных dynamic.**

**Назначение:** Отражения или коммуникации с другими компонентами

**Использование:** для членов класса - поля, свойства/индексаторы, структур , для метода, делегата, или унарных/бинарных операторов

**Что происходит:** может получить какое угодно начальное значение, и на протяжении времени его существования это значение может быть заменено новым. компилятор конвертирует System.Object; применяет атрибут System.Runtime.CompilerServices.DynamicAttribute

**Ограничение динамических типов.**

- могут использоваться только для обращения к членам экземпляров ( должна ссылаться на объект)

- не могут использовать лямбда-выражения или анонимные методы C# при вызове метода

- не могут воспринимать расширяющие методы

# ===8. Массивы C# одномерные, прямоугольные и ступенчатые.

**Массив** представляет набор однотипных переменных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки.

**Одномерные массивы:**

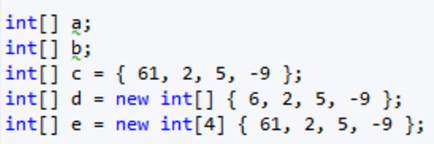
тип[] имя;

тип[] имя = new тип [ размерность ];

тип[] имя = { список инициализаторов };

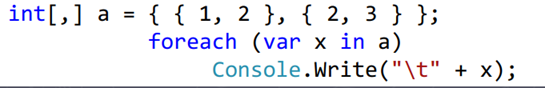
тип[] имя = new тип [] { список инициализаторов };

тип[] имя = new тип [ размерность ] { список инициализаторов };

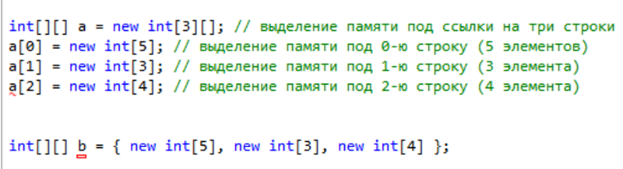


**Цикл foreach** предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

ПРЯМОУГОЛЬНЫЕ



СТУПЕНЧАТЫЕ



# ===9. Понятие кортежей. Свойства, создание

**Кортеж** представляет набор значений, заключенных в круглые скобки:

var tuple = (5, 10);

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый\_номер\_поля\_в\_кортеже].

**Свойства:**

►создается один раз и остается не именным (все свойства доступны только для чтения)

►позволяют использовать методы CompareTo, Equals, GetHashCode и ToString, свойство Size

►реализуют интерфейсы IStructuralEquatable, IStructuralComparable и IComparable (можно сравнивать)

**Создание:**

static void Main(string[] args)

{

var tuple = (5, 10);

Console.WriteLine(tuple.Item1); // 5

Console.WriteLine(tuple.Item2); // 10

tuple.Item1 += 26;

Console.WriteLine(tuple.Item1); // 31

Console.Read();

}

**Использование кортежей:** кортежи могут передаваться в качестве параметров в метод, могут быть возвращаемым результатом функции, либо использоваться иным образом.

# ===10. Принципы объектно-ориентированного программирования.

**Объектно-ориентированное программирование** — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования

Принципы ООП:

* **Наследование** (Inheritance) - позволяет расширять поведение базового (родительского) класса, наследуя основную функциональность в производном подклассе (дочернем классе):
  + Наследование обеспечивает принцип иерархической классификации: Person -> Student
* **Инкапсуляция** (Encapsulation) - это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных:
  + Никто не знает что внутри
  + Никто не может менять данные снаружи
  + Свойства инкапсуляции:
    - Совместное хранение данных и функций
    - Сокрытие внутренней информации от пользователя
    - Изоляция пользователя от особенностей реализации.
* **Полиморфизм** (Polymorphism) - реализация задач одной и той же идеи разными способами; способность функции обрабатывать данные разных типов
  + Пример - руль в транспортных средствах
  + Поддержка полиморфизма осуществляется через виртуальные функции, механизм перегрузки функций и операторов, а также обобщения
* **Абстракция** - выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.
  + Основная идея состоит в том, чтобы представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи.

# ===11. Класс. Элементы класса. Свойства и индексаторы.

**Класс** – это некоторое абстрактное понятие-шаблон, по которому определяется форма объекта.

Описанием объекта является класс, а объект представляет экземпляр этого класса. **Объект** – это физическая реализация класса(шаблона).

**Класс:**

**Данные члены:**

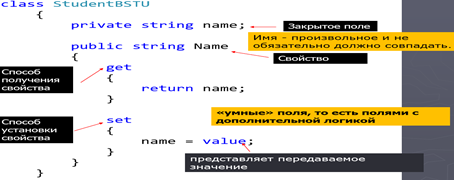
* Поля (любые переменные, ассоциированные с классом
* Константы (могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова **const.** Если она объявлена как public, то в этом случае становится доступной извне класса.)
* События (определяют уведомления, которые может генерировать класс)

**Функции-члены:**

* Методы (реализуют вычисления или другие действия, выполняемые классом или экземпляром)
* Свойства (определяют методы записи и чтения)
* Конструкторы (вызываются при создании нового объекта данного класса, выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров)
* Финализаторы
* Операции (задают действия с объектами с помощью знаков операций)
* Индексаторы (обеспечивают воз-сть доступа к элементам класса по их порядковому номеру)

**Свойства класса**

**Свойства** – специальные методы класса, служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки (предоставляет инкапсуляцию).



**Назначение** - свойства позволяют вложить дополнительную логику

**Ограничения свойств**:

1) не может быть передано методу в качестве параметра ref или out.

2) не подлежит перегрузке

3) не должно изменять состояние базовой переменной при вызове аксессора get

4) могут быть статическими, экземплярными, абстрактными и виртуальными

5) могут иметь модификатор доступа 6) могут определяться в интерфейсах

**Автоматические свойства:** тип имя { get; set; }. Компилятор автоматически реализует методы для правильного возвращения значения из поля и назначения значения полю

**Проблемы:**

• неявная инициализация

• проблемы при сериализации и десериализации

• во время отладки нельзя установить точку останова

**Индексаторы** (свойства с параметрами)

►Позволяют индексировать объекты таким же способом, как массив или коллекцию

►«умный» индекс для объектов

►средство, позволяющее разработчику перегружать оператор []

Атрибуты спецификаторы тип this [ список\_параметров ] get код доступа set код доступа

**Ограничения на индексаторы:**

1) значение, выдаваемое индексатором, нельзя передавать методу в качестве параметра ref или out

2) индексатор не может быть объявлен как static.

Можно перегружать

# ===12. Класс. Константы. Поля только для чтения. Инициализаторы класса.

**Класс** – это некоторое абстрактное понятие-шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса(шаблона).

**Константы.**

const int CC =100; // значение не может изменено

1) компилятор сохраняет значение константы в метаданных модуля константы можно определять только для таких типов, которые компилятор считает примитивными (или не примитивный но тогда = null)

2) константы считаются не явно статическими, всегда связаны с типом, а не с экземпляром тип

3) нельзя получать адрес константы и передавать ее по ссылке

4) определять можем один раз

5) к моменту компиляции они должны быть определены.

**Поля для чтения readonly**

1) Запись в поле разрешается при объявлении или в коде конструктора

2) Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

public readonly double K = 23;

**Инициализаторы.**

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

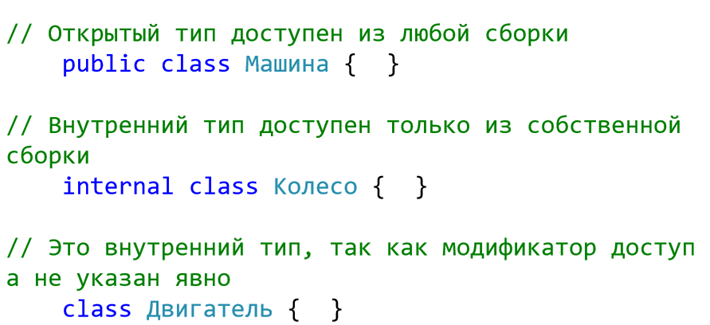
Person tom = new Person { name = "Tom", age=31 };

tom.GetInfo(); // Имя: Tom Возраст: 31

# ===13. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть контекст, в котором можно употреблять данную переменную или метод.

**Видимость типа** может быть открытой (public) или внутренней (internal).



**Доступ к членам типов**

►**public** - доступ не ограничен – все члены во всех сборках

►**private** - по умолчанию для членов класса (используется для вложенных классов). Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.

►**protected** - (используется для вложенных классов) такой член класса доступен из любого места в текущем классе или в производных классах.

► **internal** - класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок.

► **protected internal -** совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal.

Существуют различные **контексты:**

* **Контекст класса.** Переменные, определенные на уровне класса, доступны в любом методе этого класса
* **Контекст метода.** Переменные, определенные на уровне метода, являются локальными и доступны только в рамках данного метода. В других методах они недоступны
* **Контекст блока кода.** Переменные, определенные на уровне блока кода, также являются локальными и доступны только в рамках данного блока. Вне своего блока кода они не доступны.

# ===14. Класс. Конструкторы и их свойства. Деструкторы

**Класс** – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса (шаблона).

**Конструкторы** — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа.

Есть конструкторы по умолчанию и с аргументами. Если нет конструктора, то компилятор создает конструктор по умолчанию, который инициализирует все поля класса значениями по умолчанию (0, false, null).

Свойства конструкторов:

* не может возвращать значение
* имя равно имени класса
* не наследуются
* нельзя применять ключевые слова virtual, new, override, sealed, abstract
* для класса без явно заданных конструкторов компилятор создает конструктор по умолчанию (без параметров)
* для статических классов (запечатанных и абстрактных) компилятор не создает конструктор по умолчанию
* может быть несколько конструкторов, сигнатуры и уровни доступа к конструкторам обязательно должны отличаться
* можно явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова this: обеспечивает доступ к текущему экземпляру класса

**Деструкторы** вызываются непосредственно перед окончательным уничтожением объекта системой "сборки мусора", чтобы гарантировать четкое окончание срока действия объекта. Если программа завершиться до того, как произойдет "сборка мусора", деструктор может быть вообще не вызван.

**СИНТАКСИС: ~имя\_класса() { // код деструктора }**

Свойства деструкторов:

* Класс может иметь только один деструктор
* Деструкторы не могут быть унаследованы или перегружены
* Деструкторы невозможно вызвать, они запускаются автоматически
* Деструктор не принимает модификаторы и не имеет параметров

# ===15. Класс и методы System.Object.

**Класс** – это некоторое абстрактное понятие - шаблон, по которому определяется форма объекта.

**Объект** – это физическая реализация класса (шаблона).

**System.Object** - класс .Net, от которого наследуются все классы, объекты, типы значений. Переменная ссылочного типа object может ссылаться на объект любого другого типа.

Методы System.Object:

* Открытые:
  + **ToString()** – получение строкового представления данного объекта.
    - для базовых типов выводится их строковое значение
    - для классов - выводит полное название класса с указанием пространства имен, в котором определен этот класс: Console.WriteLine(Olga.ToString());
  + **GetHashCode()** – возвращает хэш-код для значения данного объекта. Позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код.
    - Максимально быстрый
    - Объекты с одинаковым значением должны возвращать одинаковые коды
  + **Equals()** – сравнивает 2 объекта на равенство.
  + **GetType()** – возвращает тип данного объекта
* Закрытые:
  + **Finalize()** – деструктор - вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом.
  + **Clone()** – создает копию объекта и возвращает ссылку на эту копию (в случае типа значение – ссылку на упаковку).
    - Копируются все типы значений в классе.
    - Если класс включает в себя члены ссылочных типов, то копируются только ссылки, а не объекты, на которые они указывают.
    - Не виртуальный, переопределять его реализацию нельзя

# ===16. Статические методы и статические конструкторы класса.

Переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические.

Методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические.

public class StudentBSTU {

private static string uo;

public static string UO { get; set; } = "BSTU";

public static void getUo (){ Console.WriteLine(UO); } }

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса. При использовании статических членов необязательно создавать экземпляр класса:

StudentBSTU.getUo();

Console.WriteLine(StudentBSTU.UO);

Для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.

**Свойства статических методов:**

► отсутствует ссылка this, поскольку такой метод не выполняется относительно какого-либо объекта

► в методе static допускается непосредственный вызов только других методов типа static

► для метода static непосредственно доступными оказываются только другие данные типа static, определенные в его классе

**Статические конструкторы или конструкторы типа.**

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются).

Конструктор экземпляра инициализирует данные экземпляра.

Конструктор класса (типа)— данные класса.

**Свойства:**

► закрытые автоматически

► не имеет параметров

► как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса

► нельзя вызвать явным образом (вызываются до создания первого экземпляра объекта или до вызова любого статического метода).

class D

{

private D() { } // закрытый конструктор

static D(){ \_a = 200; }// статический конструктор

static int \_a;

}

# ===17. Статические классы. Методы расширения и правила их определения.

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы.

static class имя класса { // ...

► прямой потомок System.Object

► экземпляры такого класса создавать запрещено

►не должен реализовывать никаких интерфейсов (не вызвать)

►нельзя использовать в качестве поля, параметра метода или локальной переменной

►от него запрещено наследовать

►все элементы такого класса должны явным образом объявляться с модификатором static

►может иметь статический конструктор

►Компилятор не создает автоматически конструктор по умолчанию

Статические классы применяются главным образом в двух случаях. Во-первых, статический класс требуется при создании метода расширения. Методы расширения связаны в основном с языком LINQ. И во-вторых, статический класс служит для хранения совокупности связанных друг с другом статических методов:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

**Методы расширения** (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем.

public static class NewFromAlex {

public static bool isLetter(this String st, char a) {

for (Int32 index = 0; index < st.Length; index++)

if (st[index] == a) return true;

return false; } }

Для того, чтобы создать метод расширения, вначале надо создать статический класс, который и будет содержать этот метод. Затем объявляем статический метод. Собственно метод расширения - это обычный статический метод, который в качестве первого параметра всегда принимает такую конструкцию: this имя\_типа название\_параметра. Затем у всех строк мы можем вызвать данный метод.

1) Проверяется класс и его базовые

2) Ищется любой статический класс с методом ####, у которого первый параметр соответствует типу выражения ( this)

**Правила для методов расширений**

►1) Методы расширения должны быть объявлены в статическом необобщенном классе (первого уровня)

►2) this перед первым аргументом и только один

►3) использовать аккуратно

# ===18. Анонимные типы.

Позволяют создать объект с некоторым набором свойств без определения класса (тип в одном контексте или один раз). Анонимный тип определяется с помощью ключевого слова **var и инициализатора объектов**:

1. var user = new { Name = "Tom", Age = 34 };
2. Console.WriteLine(user.Name);

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - **свойства анонимных типов доступны только для чтения.**

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту.

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа.

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если нам захочется добавить свойство, то мы сразу же на месте анонимного объекта это можем сделать. В случае с классом придется изменять еще и класс, который может больше нигде не использоваться. Типичная ситуация - получение результата выборки из базы данных: объекты используются только для получения выборки, часто больше нигде не используются, и классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

# ===19. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.

**Параметры** представляют собой переменные, которые определяются в сигнатуре метода и создаются при его вызове.

Существует **два способа передачи параметров** в метод в языке C#: по значению и по ссылке.

Для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

►По умолчанию- параметры-значения;

► параметры-ссылки — ref;

► выходные параметры-ссылки — out:

► переменное количество — params (один и последний).

**Назначение:**

► позволить методу менять содержимое его аргументов

► возвращать более одного значения

**ref** заставляет С# организовать вместо вызова по значению вызов по ссылке.

При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

Аргументу, передаваемому методу "в сопровождении" модификатора ref, должно быть присвоено значение до вызова метода.

Модификатор **out** подобен модификатору ref за одним исключением: его можно использовать для передачи значения из метода.

out-параметр "поступает" в метод без начального значения, но метод (до своего завершения) обязательно должен присвоить этому параметру значение.

По сути, как и в случае с ключевым словом ref, ключевое слово out применяется для для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

**params** позволяет передавать методу переменное количество аргументов одного типа.

static void Addition(params int[] integers)

{

int result = 0;

for (int i = 0; i < integers.Length; i++)

{

result += integers[i];

}

Console.WriteLine(result);

}

**Необязательные аргументы**

►позволяет определить используемое по умолчанию значение для параметра метода

►можно применять в конструкторах, индексаторах

static int OptionalParam(int x, int y, int z=5, int s=4)

**Именованные аргументы**

Позволяет указать имя того параметра, которому присваивается его значение ( в конструкторах, индексаторах или делегатах.) И в этом случае порядок следования аргументов уже не имеет никакого значения.

// Использование именованных аргументов

// при вызове метода

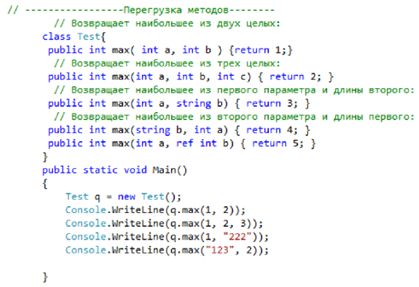
int sum1 = mySum(a: 3, b: 10);

# ===20. Перегрузка методов и операторов. Правила перегрузки операторов.

**Перегрузка методов**

* один и тот же метод, но с разным набором параметров
* позволяет обращаться к связанным методам посредством одного, общего для всех имени.
* никакие два метода внутри одного и того же класса не должны иметь одинаковую сигнатуру сигнатура (signature) = имя метода + список его параметров (не включает тип значения, возвращаемого методом, не включает params параметр)

Разные версии метода могут иметь разные возвращаемые значения.



**Перегрузка операций**

Перегрузка операторов заключается в определении в классе, для объектов которого мы хотим определить оператор, специального метода:

public static возвращаемый\_тип operator оператор(параметры)

{ }

►способ объявления новых операций для типа

Спецификация CLR требует, чтобы перегруженные операторные методы были:

1) открытыми и статическими

2) тип одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод

**Операции подлежащие перегрузке**

* +, -, !, ++, --
* true, false (попарно)
* +, -, \*, /, %, &, |, ^, <<, >>
* ==, !=, <, >, <=, >= (перегрузка парами)

**Операции не подлежащие перегрузке:**

* [] (но есть индексатор)
* () (можно определить новые операторы преобразования)
* +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= (но получаем автоматически в случае перегрузки бинарной операции)
* &&, ||
* =, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof

**Правила:**

* префиксные операции ++ и – – перегружаются парами;
* операции сравнения перегружаются парами: == и != ; < и >;<= и >=.
* Перегруженные операции обязаны возвращать значения
* Должны объявляться как public и static
* префиксная и постфиксная формы операций ++ и --, в отличие от оригинальных операций, семантически НЕ различаются.

Если перегружаются операторы == и !=, то для этого требуется переопределить методы Object.Equals() и Object.GetHashCode().

# ===21. Операции преобразования типа. Явная и неявная форма. Ограничения.

Когда данные одного типа присваиваются переменной другого типа, **неявное** преобразование типов происходит **автоматически** при следующих условиях:

* оба типа совместимы
* диапазон представления чисел целевого типа шире, чем у исходного типа

Если оба эти условия удовлетворяются, то происходит расширяющее преобразование.

**Операции преобразования типа**

* преобразует объект исходного класса в другой тип
* явная и неявна форма - будет ли этот алгоритм выполняться неявно или необходимо будет явным образом указывать необходимость соответствующего преобразования.

**implicit** operator *тип* ( *параметр* ) // неявное преобразование - преобразование вызывается автоматически.

**explicit** operator *тип* ( *параметр* ) // явное преобразование - преобразование вызывается в том случае, когда выполняется привидение типов

*тип* - тип, в который выполняется преобразование.

*параметр* - тип, который преобразуется.

Преобразуемые типы не должны быть связаны отношениями наследования.

Ключевые слова implicit и explicit в сигнатуру не включаются.

**Ограничения на операторах преобразования:**

* Исходный или целевой тип преобразования должен относиться к классу, для которого объявлено данное преобразование
* Нельзя указывать преобразование в класс object или же из этого класса
* Для одних типов данных нельзя указывать одновременно явное и неявное преобразование
* Нельзя указывать преобразование базового класса в производный класс
* Нельзя указывать преобразование в/из интерфейс

class Counter

{

public static implicit operator Counter(int x)

{

return new Counter { Seconds = x };

}

public static explicit operator int(Counter counter)

{

return counter.Seconds;

}

}

Так как операция преобразования из Counter в int определена с ключевым словом explicit, то есть как явное преобразование, то в этом случае необходимо применить операцию приведения типов:

int x = (int)counter1;

В случае с операцией преобразования от int к Counter ничего подобного делать не надо, поскольку данная операция определена с ключевым словом implicit, то есть как неявная.

# ===22. Вложенные типы. Вложенные объекты

Тип, определенный внутри типа называется **вложенным типом:**

class Person {

public class Date {

public Date() { }

}

}

Person.Date today = new Person.Date();

**По умолчанию являются private.** Однако можно указать модификатор доступа, определяющий доступность вложенного типа:

* Вложенные типы класса могут иметь модификаторы public, protected, internal, protected internal, private или private protected
* Вложенные типы структуры могут иметь модификаторы public, internal или private.

Вложенный тип может получить доступ к внешнему типу, а внутренний тип — к внешнему. Чтобы получить доступ к вмещающему типу, передайте его в качестве аргумента в конструктор вложенного типа. Пример:

public class Container {

public class Nested {

private Container parent;

public Nested() { }

public Nested(Container parent) {

this.parent = parent;

}

}

}

Вложенный тип имеет доступ ко всем членам, которые доступны вмещающему типу.

**Вложенные объекты**

public class Date {

public Date() { }

}

class Person {

Date birthday;

}

Person anna = new Person();

anna.birthday = null;

Вложение или включение классов **модель включения-делегирования.**

**#region -** сворачивание и разворачивания блоков кода.

class Person {

#region Peson filed

public Date birthday;

#endregion

}

# ===23. Правила наследования C#.

**Наследование** - это механизм получения нового класса на основе уже существующего

Роль наследования:

* формирует иерархию
* поощряет повторное использование кода

Правила наследования:

* В C# наследование всегда подразумевается открытым
  + class Student : Person
* Запрещено множественное наследование классов (но не интерфейсов)
* Если класс объявлен с модификатором **sealed**, то от этого класса **нельзя наследовать** и создавать производные классы
* наследуются все свойства, методы, поля и т.д., которые есть в базовом классе
* Производному классу доступны public, internal, protected и protected internal члены базового класса (private – недоступны)
* не наследуются конструкторы базового класса
* тип доступа к производному классу должен быть таким же, как и у базового класса или более строгим
* Ссылке на объект базового класса можно присвоить объект производного класса (но вызываются для него только методы и свойства, определенные в базовом классе.)
* С помощью ключевого слова **base** мы можем обратиться к базовому классу:

class Person {

public string Name { get; set; }

public Person(string name) { Name = name; }

public void Display() { Console.WriteLine(Name); }

}

class Employee : Person {

public string Company { get; set; }

public Employee(string name, string company) : base(name)

{ Company = company; }

}

# ===24. Сокрытие имен при наследовании. Обращение к скрытым членам

Сокрытие имен происходит, когда в базовом классе и в классе-наследнике объявлены методы с одинаковым именем. В такой ситуации метод базового класса скрывается и программа может работать не так как предусматривал программист. В таких случаях необходимо воспользоваться модификатором **new**, который скажет компилятору о вашем явном намерении скрыть метод базового класса и использовать метод, объявленный в классе наследнике.

class MyClass {

public int x = 10, y = 5, z = 6;

}

class ClassA : MyClass {

//скрываем члены класса MyClass

public new int x = 12, y = -2, z = -5;

}

**Применение ключевого слова base для доступа к скрытому имени**

Имеется ещё одна форма ключевого слова **base**, которая действует подобно ключевому слову this, за исключением того, что она всегда ссылается на базовый класс в том производном классе, в котором она используется: **base.член**, где член может обозначать метод или поле экземпляра. Эта форма ключевого слова base чаще всего применяется в тех случаях, когда под именами членов производного класса скрываются члены базового класса с теми же именами. С помощью base могут вызываться скрытые методы.

class MyClass {

public int x;

}

class ClassA : MyClass {

new int x = 10;

public void method(int i1, int i2) {

//Координата х из базового класса MyClass

base.x=i1;

Console.Writelene(base.x);

//Координата х из класса ClassA

x = 12;

Console.Writelene(x);

}

}

# ===25. Использование операций is и as

Операция **is** дает в результате булевское значение, говорящее о том, можно ли преобразовать данное выражение в указанный тип, как с помощью преобразования ссылки, так и посредством операций упаковки и распаковки.

Оператор is никогда не генерирует исключение.

String derivedObj = "Dummy"; // derivedObj is Object - true

Object baseObjl = new Object (); // baseObjl is String - false

Object baseObj2 = derivedObj; // baseObj2 is String - true

int j = 123;

object boxed = j; // boxed is int - true; boxed is ValueType - true

object obj = new Object () ; // obj is int - false

Как уже упоминалось, операция is принимает во внимание только преобразования ссылок. Это значит, что она не может проверять определенные пользователем преобразования, имеющиеся в типах.

Для null-ссылок оператор is всегда возвращает false, так как объекта, тип которого нужно проверить, не существует.

Операция **as** подобна is за исключением того, что она возвращает ссылку на целевой тип. Поскольку гарантируется, что она никогда не сгенерирует исключения, то если данное преобразование невозможно, просто возвращается null-ссылка. Подобно is, операция as принимает во внимание только преобразования ссылок или преобразования с упаковкой/распаковкой. Операнд as <тип>

Операция as выполняется:

* Если <операнд> имеет тип, заданный в <тип>.
* Если <операнд>, может быть неявно преобразован в <тип>.
* Если операнд <операнд>, может быть упакован в <тип>.

class A {}

class B : A {}

public static void Main() {

A obj1 = new A();

B obj2 = obj1 as B; //null

B obj3 = new B();

A obj4 = obj3 as A;

# ===26. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.

**Полиморфизм** - ключевой аспект объектно-ориентированного программирования, способность к изменению функций, унаследованных от базового класса.

Простыми словами, полиморфизм – это различная реализация однотипных действий.

**Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе. Такие методы помечаются модификатором **virtual.**

Виртуальными могут быть: методы, свойства, индексаторы.

**Абстрактный метод** – это метод, который ДОЛЖЕН быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое), в отличии от виртуального.

**Переопределение метода** – это изменение реализации метода, установленного как виртуальный (в классе наследнике метод будет работать отлично от базового класса).

Чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

**Полиморфный интерфейс в базовом классе** - набор членов класса, которые могут быть переопределены в классе-наследнике.

virtual public void A\_method() { }

Переопределение виртуального метода в производном классе:

override public void A\_method() { }

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором **sealed.** При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

**Правила переопределения**

* Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.
* не может быть static или abstract
* вызывается ближайший вариант, обнаруживаемый вверх по иерархии (многоуровневая)
* Если не virtual переопределять нельзя

# ===27. Понятие раннего и позднего связывания.

Перегрузка методов обеспечивает статический полиморфизм, а виртуальный метод – динамический.

**Связывание** – это ассоциация синтаксического элемента, содержащего имя метода, с логической частью программы.

**Раннее связывание** – адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции.

**Достоинства:** высокое быстродействие получаемых выполнимых программ.

**Недостатки:** снижение гибкости программ.

**Позднее связывание** (только для методов классов) – во время выполнения приложения определяется действительный класс объекта, адрес которого находится в указателе, и вызывается метод нужного класса.

**Достоинства:** высокая гибкость выполняемой программы, возможность реакции на события.

**Недостатки:** относительно низкое быстродействие программы.

Для реализации позднего связывания компилятор формирует таблицу виртуальных методов (Virtual Method Table или VMT). В нее записывается адреса виртуальных методов. Для каждого класса создается своя таблица.

Когда создается объект класса, то компилятор передает в конструктор объекта специальный код, который связывает объект и таблицу VMT.

А при вызове виртуального метода из объекта берется адрес его таблицы VMT. Затем из VMT извлекается адрес метода и ему передается управление. Собственно так и выполняется виртуальный метод.

Виртуальные функции предоставляют механизм позднего или динамического связывания.

# ===28. Абстрактные классы и методы. Бесплодные классы.

**Бесплодные (запечатанные) классы** - класс, от которого наследовать запрещается.

sealed class A: AAA

**Абстрактный класс** – это класс объявленный с ключевым словом abstract.

abstract class MyClass {}

**Абстрактные классы:**

* Служат только для порождения потомков, предоставляют базовый функционал для классов-наследников.
* Задают интерфейс для всей иерархии
* Могут содержать и полностью определенные методы, переменные, конструкторы, свойства
* Создавать объект абстрактного класса нельзя!!!!!!! (ссылку можно)

public abstract class Person {

public abstract void work();

}

public class Employee : Person {

public override void work() { //... }

}

В абстрактном классе обычно реализуется некоторая общая часть нескольких сущностей или другими словами - абстрактная сущность, которая, как объект, не может существовать, и эта часть необходима в классах наследниках.

**Абстрактный метод** определяет полиморфный интерфейс, не имеют никакой реализации.

**Абстрактный метод** – это метод, который не имеет своей реализации в базовом классе, и он ДОЛЖЕН быть реализован в классе-наследнике. Абстрактный метод может быть объявлен только в абстрактном классе.

Производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. Если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

**Свойства abstract методов**

* абстрактные методы автоматически виртуальные (virtual не ставится)
* абстрактные методы не используются со static

public abstract class Person

{

public abstract void work();

}

private static void Main(string[] args)

{

Person anna = new Person(); //ошибка

}

* А.К. может быть параметром метода - полиморфные методы

Реализация абстрактного метода в классе наследнике происходит так же, как и переопределение метода – при помощи ключевого слова override:

[модификатор доступа] override [тип] [имя метода] ([аргументы])

{

// реализация метода

}

# ===29. Структуры в C#.

Кроме базовых элементарных типов данных и перечислений в C# имеется и составной тип данных, который называется структурой.

**Структура** – это составной объект (пользовательский тип данных), совокупность логически связанных данных различного типа, объединенных под одним идентификатором.

Структура относится **к типу значения**, класс – к ссылочному типу данных, т.е. структуры размещаются в стеке, а классы – в куче.

**Также структуры не поддерживают наследование.**

**Может содержать:** структура этого же типа, указатель на такую же структуру, указатель на функцию, прототип функции, объединение, перечисление, реализовывать интерфейсы, значение NULL, объявление конструкторов с параметрами.

**Не может:** инициализированные при объявлении поля (если не константы и не статические), не может объявлять конструктор (деструктор) по умолчанию.

struct Book {

public string name;

public string author;

public int year;

public void Info() {

Console.WriteLine($"Книга '{name}' (автор {author}) была издана в {year} году");

}

}

Чтобы можно было использовать переменные и методы структуры из любого места программы мы ставим перед переменными и методом модификатор доступа public

Структуру можно задать как внутри пространства имен (как в данном случае), так и внутри класса, но не внутри метода.

Также можно использовать массив структур:

Book[] books=new Book[3];

books[0].name = "Война и мир";

books[0].author = "Л. Н. Толстой";

books[0].year = 1869;

books[1].name = "Преступление и наказание";

books[1].author = "Ф. М. Достоевский";

books[1].year = 1866;

Кроме обычных методов структура может содержать специальный метод - **конструктор**, который выполняет некую начальную инициализацию объекта, например, присваивает всем полям некоторые значения по умолчанию. Нельзя определить конструктор, используемый по умолчанию (конструктор без параметров). Он определяется для всех структур автоматически и не подлежит изменению В принципе для структуры не обязательно использовать конструктор:

Book book1;

Однако в этом случае, чтобы использовать структуру, нам надо будет первоначально проинициализировать все ее поля:

book1.name = "Война и мир";

book1.author = "Л. Н. Толстой";

book1.year = 1869;

**Перечисление** – тип данных, чьё множество значений представляет собой ограниченный список идентификаторов.

**Тип перечисления** - целочисленный тип (byte, int, short, long). По умолчанию используется тип int:

enum MathOperation { Add , Subtract, Multiply, Divide }

Каждому элементу перечисления присваивается целочисленное значение, 0, 1 т.д. Можно определять явным образом:

enum MathOperation :short

{

Add = 4,

Subtract, //5

Multiply =9 , //9

Divide //10

}

# ===30. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.

**Интерфейсы** позволяют определить некоторый функционал, не имеющий конкретной реализации. Задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах.

interface **I**имя\_интерфейса [ : предки]

Тело интерфейса [; ]

Интерфейс **не может содержать**: константы, поля, операции, конструкции, деструкторы, типы, любые статические элементы.

Интерфейс **может содержать:** абстрактные методы, обобщения свойства и индексаторы, события.

**Public по умолчанию.**

interface IDo {

void Go();

int Jump(int a);

int Energy {get; } //шаблон свойства

}

Назначение: задание общих характеристик объектов различных иерархий - навязывание контракта.

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы.

Интерфейс или класс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую.

interface IDo {

void Go();

int Jump(int a);

int Energy {get; } //шаблон свойства

}

interface IKnow {

Void Count();

int Math();

}

interface IPosebel : IDo, IKnow { }

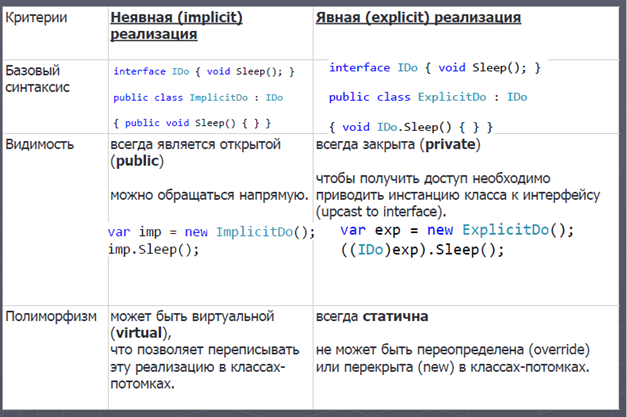
# ===31. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.

**Интерфейсы** - задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах

[атрибуты] [спецификаторы] interface Iимя\_интерфейса [ : предки] Тело интерфейса[; ]

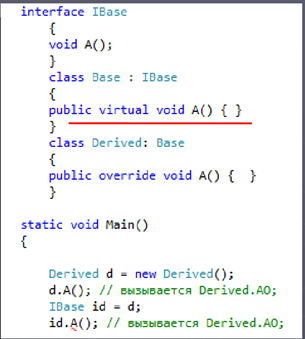
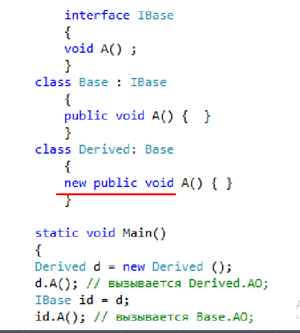
При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается явная реализация члена интерфейса, или просто **явная реализация.**

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он не стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.



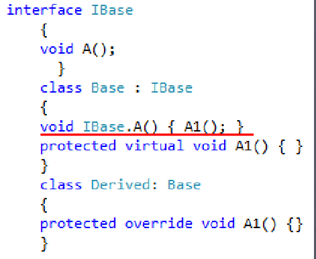
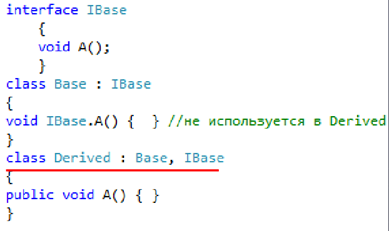
**Работа с объектами через интерфейсы (преобразования)**

Класс наследует все методы своего предка (интерфейсы). Он может переопределить ( new), но обращаться к ним - через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия



Если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату

Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается. При необходимости переопределить в потомках его поведение



Существует возможность повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом предком, уже реализовавшим этот интерфейс.

# ===32. Ковариантность интерфейсов. Контравариантность интерфейсов

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Имеется три возможных варианта поведения:

* **Ковариантность:** позволяет использовать более конкретный тип, чем заданный изначально. Это средство, разрешающее методу возвращать тип, производный от класса, указанного в параметре типа.
* **Контравариантность:** позволяет использовать более универсальный тип, чем заданный изначально. Это средство, разрешающее методу использовать аргумент, тип которого относится к базовому классу, указанному в соответствующем параметре типа.
* **Инвариантность:** позволяет использовать только заданный тип

По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются **инвариантными.**

Параметр **ковариантного типа** объявляется с помощью ключевого слова **out**, которое предваряет имя этого параметра.

// В этом обобщенном интерфейсе поддерживается ковариантность,

public interface IMyInfo<out Т> {

T GetObject();

}

В данном контексте ключевое слово out обозначает, что обобщенный тип T является ковариантным. А раз он ковариантный, то метод GetObject() может возвращать ссылку на обобщенный тип T или же ссылку на любой класс, производный от типа Т.

На применение ковариантности накладываются некоторые ограничения. Ковариантность параметра типа может распространяться только на тип, возвращаемый методом. Следовательно, ключевое слово out нельзя применять в параметре типа, служащем для объявления параметра метода. Ковариантность оказывается пригодной только для ссылочных типов. Ковариантный тип нельзя использовать в качестве ограничения в интерфейсном методе. Так, следующий интерфейс считается недопустимым:

public interface IMyCoVarGenIF2<out Т> {

void M<V>() where V:T; // Ошибка, ковариантный тип T нельзя использовать как ограничение }

Параметр **контравариантного типа** объявляется с помощью ключевого слова **in**, которое предваряет имя этого параметра.

// Это обобщенный интерфейс, поддерживающий контравариантность.

public interface IMyContraVarGenlFc<in Т> {

void Show(T obj);

}

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к аргументам, используемым в качестве возвращаемых типов.

Контравариантность оказывается пригодной только для ссылочных типов, а параметр контравариантного типа можно применять только к аргументам методов. Следовательно, ключевое слово in нельзя указывать в параметре типа, используемом в качестве возвращаемого типа.

# ===33. Стандартные интерфейсы .NET. Назначение и применение.

Стандартные интерфейсы поддерживаются многими стандартными классами библиотеки.

**Сравнение объектов (интерфейс IComparable)**

Интерфейс IComparable содержит всего один метод CompareTo(),возвращающий результат сравнения двух объектов – текущего и переданного ему в качеству параметра:

interface IComparable

{

int CompareTo(object obj)

}

Метод должен возвращать:

* 0, если текущий объект и параметр равны
* отрицательное число, если текущий объект меньше параметра;
* положительное число, если текущий объект больше параметра.

**Сортировка объектов по различным критериям (интерфейс IComparer)**

Данный интерфейс определен в пространстве имен System.Collections. Он также содержит один метод Compare(),возвращающий результат сравнения двух объектов, переданных ему в качестве параметров.

interface IComparer

{

int Compare(object obj1,object obj2)

}

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий данный интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента.

**Клонирование объектов (интерфейс ICloneable)**

**Клонирование** - создание копии объекта. Копия объекта принято называть клоном. При присваивании одного экземпляра другому копируется ссылка, а не сам объект. В случае если крайне важно скопировать в другую область памяти поля объекта͵ можно воспользоваться методом MemberwiseClone(), который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта͵ в свою очередь являющиеся ссылками, не копируются. Это принято называть *поверхностным клонированием*.

Важно заметить, что для создания полностью независимых объектов крайне важно глубокое копирование, когда в памяти создается дубликат всего дерева объектов, то есть объектов, на которые ссылаются поля объекта͵ поля полей, и т.д. Алгоритм глубокого копирования сложен, требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Объект, имеющий собственные алгоритмы клонирования, должен объявляться как производный интерфейса ICloneable и переопределять его единственный метод Clone().

**Перебор объектов (интерфейс IEnumerable) и итераторы**

Оператор foreach является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки .NET позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы **IEnumerable и IEnumerator.** Для применения оператора foreach к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

**Интерфейс IEnumerable** ( перечислимый ) определяет всего один метод — GetEnumerator, возвращающий объект типа IEnumerator ( перечислитель ), который можно использовать для просмотра элементов объекта.

**Интерфейс IEnumerator** задает три элемента:

* свойство Current, возвращающее текущий элемент объекта;
* метод MoveNext, продвигающий перечислитель на следующий элемент объекта;
* метод Reset, устанавливающий перечислитель в начало просмотра.

**Итератор** представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова yield.

# ===34. Исключительные ситуации. Генерация и повторная генерация исключений.

**Structured Exception Handling** (SHE) - унифицированный подход для языков ориентированных на платформу .NET

System.Exception

Исключительная ситуация **exception** - это состояние ошибки, обнаруженное в программе в ходе ее выполнения (деление на ноль, невозможность выделения памяти при создании нового объекта и т.д. ). Исключение генерируется оператором throw( <выражение>), где аргумент - объект типа исключение.

if (b==0) throw new Exception("Zero devision");

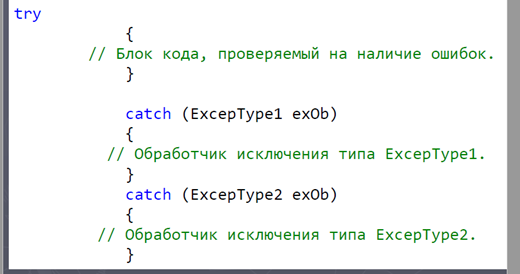
else c = a / b;

ГЕНЕРИРОВАНИЕ И РАСПОЗНАВАНИЕ ИСКЛЮЧЕНИЙ

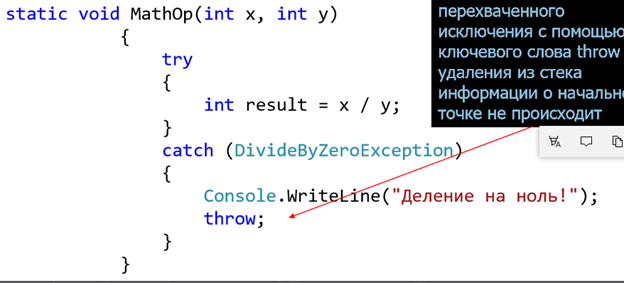
* **try** – контролируемый блок
* **throw** - генерация исключительной ситуации внутри try
* **catch** – обработчики исключений, идут за try (может быть несколько)
* **finally** - код, очищающий ресурсы и др. действия (выполняется всегда) (только 1)

Если код в блоке try не порождает исключение, CLR никогда не переходит к выполнению кода в соответствующем блоке catch.

Так как тип Exception является базовым классом для всех исключений, то выражение catch (Exception ex) будет обрабатывать практически все исключения.



**Повторная генерация исключения** - создание нового объекта посредством повторного использования старого с помощью оператора throw без параметров. При повторном вызове перехваченного исключения с помощью ключевого слова throw удаления из стека информации о начальной точке не происходит.



# ===35. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

См. Вопрос 34. (начало по исключениям)

try {…throw выражение;..}

catch(объявление исключения)

{ операторы обработчика }

catch(объявление исключения)

{ операторы обработчика }

catch(объявление исключения)

{ операторы обработчика }

**try-catch, try-finally, try-catch-finally**

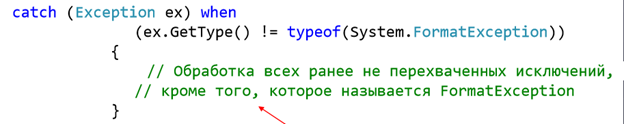
Поиск подходящего блока catch в CLR осуществляется сверху вниз, поэтому наиболее конкретные обработчики должны находиться в начале списка.

Сначала следуют потомки с наибольшей глубиной наследования, потом — их базовые классы (если таковые имеются) и, наконец, — класс System.Exception

Механизм:

1. Исключение не произошло
   1. try выполняем до конца
   2. catch пропускаем
   3. finally выполняем
2. Исключение произошло
   1. выполнение try
   2. ищем блок catch на соответствие по типу исключения
      1. если нет catch
         1. разматывает стек, локальные объекты, выходят из области видимости
         2. снова генерируется исключение в точке вызова метода
         3. если блока не найдено, то смс необработанное исключении и вып-ние программы stop
      2. catch найден
         1. Передается управление ближайшему catch-обработчику, совместимому с типом выброшенного исключения
         2. объект-исключения передается, если это предусмотрено, обработчику в качестве параметра.
         3. 2.3. переходим в finally
   3. если нет finally
      1. выполнение программы продолжается начиная с позиции, след. за последним обработчиком данного блока try

**Фильтр исключения** позволяет указать дополнительные условия, при которых используется обработчик исключения. Эти условия принимают форму булева выражения, перед которым ставится ключевое слово **when**.



int x = 1;

int y = 0;

try

{ int result = x / y; }

catch(Exception ex) when (y==0)

{ Console.WriteLine("y не должен быть равен 0"); }

catch(Exception ex)

{ Console.WriteLine(ex.Message); }

В данном случае будет выброшено исключение, так как y=0. Здесь два блока catch, но поскольку для первого блока указано условие с помощью ключевого слова when, то сработает первый блок catch. Если бы y не было бы равно 0, то сработал бы второй блок catch.

# ===36. Обобщения (generics). Свойства обобщений.

**Обобщение** - параметризированный тип. Определены для CLR – поддержка разных языков.



Угловые скобки в описании class SuperArray<T> указывают, что класс является обобщенным, а тип T, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву T, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр T в угловых скобках еще называется универсальным параметром, так как вместо него можно подставить любой тип.

Например, вместо параметра T можно использовать объект int, то есть число, представляющее номер счета. Это также может быть объект string, либо или любой другой класс или структура.

**Недостатки использования object:**

* InvalidCastException - два типа не совместимы друг с другом.
* вероятность дополнительного потребления памяти и процессорного времени, если в ходе выполнения потребуется преобразовывать

**Свойства:**

1. Обобщенный тип может содержать другой обобщенный тип

public class B<T>

{

private A<T> one;

private A<int> two;

}

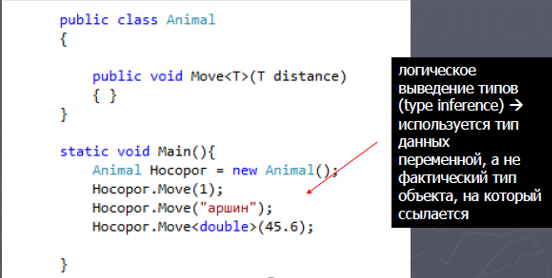
1. Обобщенные типы перегружаются на основе количества параметров (арность)

public class A {}

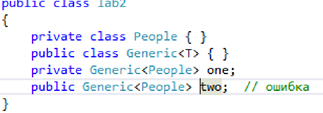
public class A<T> {}

public class A<T, U> {}

1. Обобщенными могут быть классы, структуры , интерфейсы, делегаты, методы public void Method <R> (A<R> iA, B<R,T> iB)



1. Могут содержать статические типы
2. Доступность конструируемых типов определяется на основе пересечения доступности обобщённого типа и типа в списке аргументов



1. могут использовать несколько универсальных параметров одновременно
2. поддерживает механизм ограничений

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и null. Но напрямую мы его присвоить не можем:

T id = null;

В этом случае нам надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения null, а типам значений - значение 0:

class Account<T>

{

T id = default(T);

}

# ===37. Концепция ограничений обобщений. Статические члены обобщений.

В CLR существует механизм ограничений (constraints) инструмент определения обобщенного типа с указанием допустимых для него аргументов типа.

private static T Min<T>(T o1, T o2) {

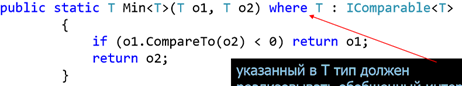
if (o1.CompareTo(o2) < 0) return o1;

return o2; //ошибка

}

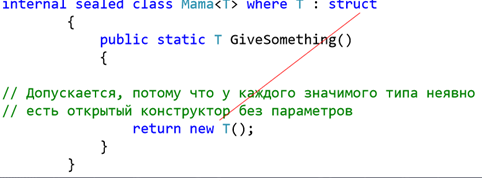
Ограничение сужает перечень типов, которые можно передать в обобщенном аргументе, и расширяет возможности по работе с этими типами. Указанный в T тип должен реализовывать обобщенный интерфейс IComparable того же типа (T).

**Ограничение на интерфейс**

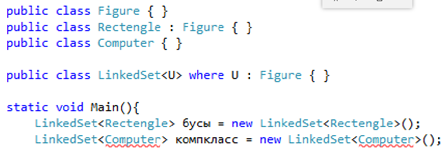


**Ограничение типа значения**

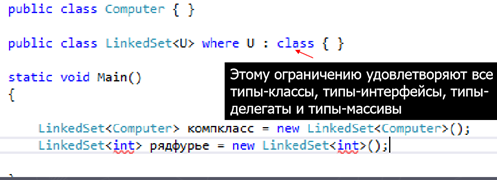
Гарантирует компилятору, что указанный аргумент типа будет иметь значимый тип, но значимые типы с поддержкой null (System.Nullable<T>) не подходят под это ограничение.

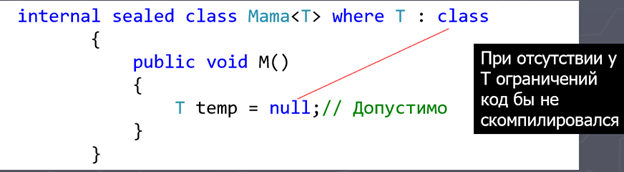


**Ограничение на базовый класс**

****

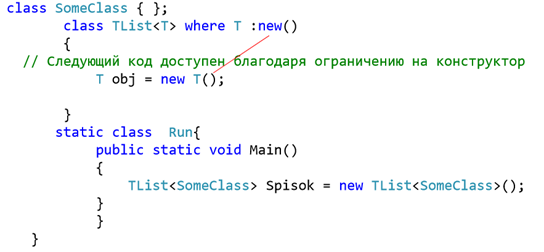
**Ограничение ссылочного типа**

****

****

**Ограничение на конструктор**

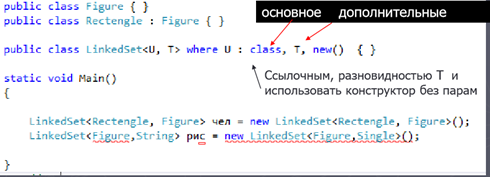
Гарантирует компилятору, что указанный аргумент-тип будет иметь не абстрактный тип, имеющий открытый конструктор без параметров.



**Особенности последним** по порядку

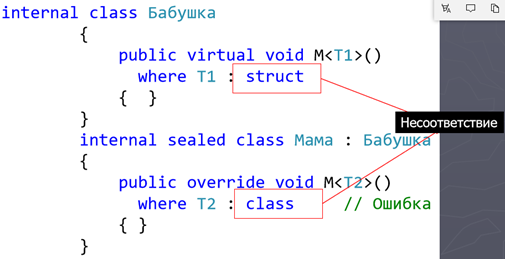
Ограничение new () позволяет конструировать объект, используя только конструктор без параметров. Ограничение new() нельзя использовать одновременно с ограничением типа значения.

**Ограничение на связь параметров**

****

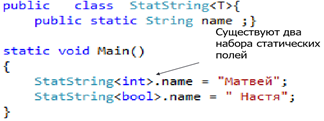
**Наследование**

При переопределении виртуального обобщенного метода в переопределяющем методе должно быть задано то же число параметров-типов, а они, в свою очередь, наследуют ограничения, заданные для них методом базового класса.



**Статические члены**

В CLR размещает статические поля типа в самом объекте-типе , каждый закрытый тип имеет свои статические поля



# ===38. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.

**Делегат** - это объект, предназначенный для хранения ссылок на методы. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается.

Синтаксис:

**delegate возвращаемый\_тип имя (список\_параметров);**

Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы, возвращаемый тип и параметры которых соответствуют указанным в объявлении делегата.

Пример:

delegate int **IntOperation** (int i, int j);

class Program {

static int Sum(int x, int y) { return x + y; }

static int Mul(int x, int y) { return x \* y; }

static void Main() {

**IntOperation** op1 = new **IntOperation**(Sum);

int result = op1(5, 10);

Console.WriteLine("Сумма: " + result);

op1 = new **IntOperation**(Mul); // меняем ссылку

result = op1(5, 10);

Console.WriteLine("Произведение: " + result);

}

}

В частности, объект делегата поддерживает три важных фрагмента информации:

* адрес метода, на котором он вызывается;
* аргументы (если есть) этого метода;
* возвращаемое значение (если есть) этого метода.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

Свойства делегатов:

* Тип данных
* Наследовать от делегата нельзя
* Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса (в любом месте, где может быть определен класс)
* может вызывать только такие методы, у которых тип возвращаемого значения и список параметров совпадают
* Может быть статический метод класса
* Имеет тот же синтаксис, что и вызов метод
* Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат (цепочки (chaining))
* делегаты могут выполняться в асинхронном режиме, при этом не надо создавать второй поток, надо вместо метода Invoke использовать пару методов BeginInvoke/EndInvoke
* делегат можно вызывать как обычный метод
* делегаты могут быть параметрами методов

**Обобщенные делегаты**:

Синтаксис:

**delegate** возвр\_тип имя<список\_параметров\_типа> (список\_аргументов);

Список параметров типа следует непосредственно после имени делегата. Преимущество обобщенных делегатов заключается в том, что их допускается определять в типизированной обобщенной форме, которую можно затем согласовать с любым совместимым методом.

Обобщенные делегаты предоставляют более гибкий способ спецификации метода, подлежащего вызову в безопасной к типам манере.

Пример:

delegate T MyDel<T> (T obj1, T obj2); // обобщ. дел.

class MySum {

public static int SumInt(int a, int b) {return a + b; }

public static string SumStr(string s1, string s2) { return s1 + " " + s2;}

public static char SumCh(char a, char b){ return (char)(a + b); }

}

class Program {

static void Main() {

MyDel<int> del1 = MySum.SumInt;

Console.WriteLine("6 + 7 = " + del1(6,7)); //вывод: 6 + 7 = 13

MyDel<char> del3 = MySum.SumCh;

Console.WriteLine("'a' + 'c' = " + del3('a','c')); //вывод: ‘а’ + ‘с’ = A

}

}

# ===39. Анонимные функции. Лямбда-выражения.

С делегатами тесно связаны Анонимные методы. **Анонимные методы** используются для создания экземпляров делегатов.

**Анонимные функции** представляет собой безымянный кодовый блок, передаваемый конструктору делегата

Определение анонимных методов начинается с ключевого слова **delegate**, после которого идет в скобках список параметров и тело метода в фигурных скобках:

class Program

{

delegate void MessageHandler(string message);

static void Main(string[] args)

{

MessageHandler handler = delegate(string mes)

{

Console.WriteLine(mes);

};

handler("hello world!");

Console.Read();

} }

Свойства анонимных функций:

* Параметры должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то в анонимном методе можно вовсе опустить параметры.
* может не содержать никаких параметров. Тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.
* метод имеет доступ ко всем переменным, определенным во внешнем коде

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата. И через эту переменную делегата можно вызвать данный анонимный метод.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Анонимные методы используются, когда надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. Иногда такие методы нужны для обработки одного события и больше ценности не представляют и нигде не применяются.

**Лямбда-выражения -** упрощенная запись анонимных методов.

Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

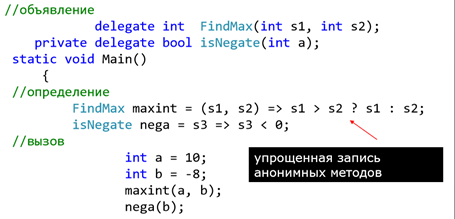
**Синтаксис:** параметр => выражение

i => i \* i;

(список\_параметров) => выражение

(x, y) => x + y;

Не надо указывать тип параметров, не надо использовать оператор return



При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому типы параметров должны быть одинаковыми. Кроме того, количество параметров должно быть таким же, как и у делегата. И возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата.

Список параметров может быть пустым

public delegate void D();

class Class1 { static void Main() {

D del = () => { Console.Write("Hello"); };

del();

} }

Можно передавать в качестве аргументов методу. Может принимать ссылку на метод. Лямбда-выражения можно передавать в качестве параметров методу.

# ===40. Обобщённые делегаты .NET. Action, Func, Predicate

**Делегаты** - объекты предназначенные для хранения ссылок на методы. Используются для поддержки событий.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action, Predicate и Func.

Определенные в пространстве имен **System**

void Action<in T1, in Т2, in ТЗ ….in T16 >

TResult Func<out TResult>

TResult Func<in T1,……T16, out TResult>

Вместо определения собственных типов делегатов рекомендуется по мере возможности использовать обобщенные делегаты.

**Action.**

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

public delegate void Action<T>(T obj)

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия.

**Func.**

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

TResult Func<out TResult>()

TResult Func<in T, out TResult>(T arg)

TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)

TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)

**Predicate.**

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

Predicate<int> isPositive = delegate (int x) { return x > 0; };

Console.WriteLine(isPositive(20));

Console.WriteLine(isPositive(-20));

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

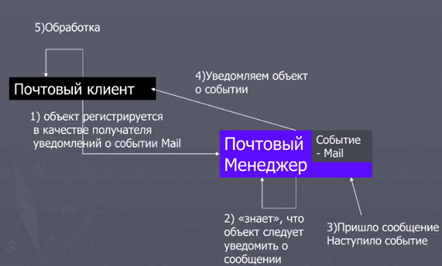
# ===41. События и делегаты.

**Делегаты** - объекты предназначенные для хранения ссылок на методы. Используются для поддержки событий.

Делегат может хранить ссылки на несколько методов и вызывать их поочередно - сигнатуры всех методов должны совпадать.

**События** - элемент класса , позволяющий ему посылать другим объектам уведомления об изменении своего состояния.

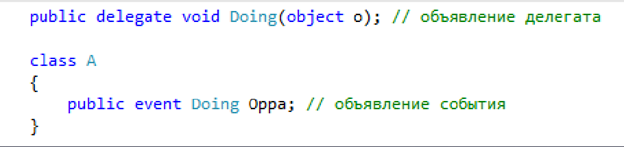
Модель “публикация – подписка” или паттерн “наблюдатель” , класс , являющийся отправителем , сообщения , публикует события, которые он может инициировать , а другие классы , являющиеся получателями сообщения , подписываются на получение этих событий.



События объявляются в классе с помощью ключевого слова **event**, после которого идет название делегата.

События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

* Описание делегата, задающего сигнатуру обработчиков событий
* Описание события
* Описание метода (методов), инициирующих событие



1. Обработка событий выполняется в классах-получателях
2. сигнатура методов-обработчиков событий, == типу делегата
3. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод :+= и -=
4. Поддерживается групповая адресация

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат.

**Стандартные .NET делегаты**

Правила:

1. имя делегата заканчивается суффиксом EventHandler;
2. делегат получает два параметра: первый параметр задает источник события и имеет тип object; второй параметр задает аргументы события и имеет тип EventArgs или производный от него.
3. Имя обработчика события принято составлять из префикса On и имени события;
4. Классы содержащие информацию о событиях должны наследовать от типа System.EventArgs, а имя типа должно заканчиваться словом EventArgs

+делегаты

Делегат можно вызвать асинхронно (в отдельном потоке), при этом в исходном потоке можно продолжать действия. Анонимный делегат (без создания классанаблюдателя): s.Go += new SomeDelegat(()=> { Console.WriteLine("Я слышу что идете!"); });

Делегаты и события обеспечивают взаимодействие взаимосвязанных объектов. События включены во многие стандартные классы .NET, используемые для разработки Windows-приложений.

# ===42. Стандартные коллекции .NET. Типы коллекций.

В C# **коллекция** представляет собой совокупность объектов. Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

**Типы коллекций:**

* необобщенные (наличие разнотипных данных, ссылки на данные типа object (не обеспечивают типовую безопасность), System.Collections)
* Обобщенные (обеспечивают типовую безопасность, System.Collections.Generic)
* специальные (System.Collections.Specialized)
* с поразрядной организацией (BitArray)
* Параллельные (многопоточный доступ к коллекции, System.Collections.Concurrent)

**Необобщенные коллекции** (ArrayList (IList, ICollection, IEnumerable, ICloneabl), BitArray (ICollection, IEnumerable, ICloneable), Hashtable, Queue, SortedList (класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу), Stack).

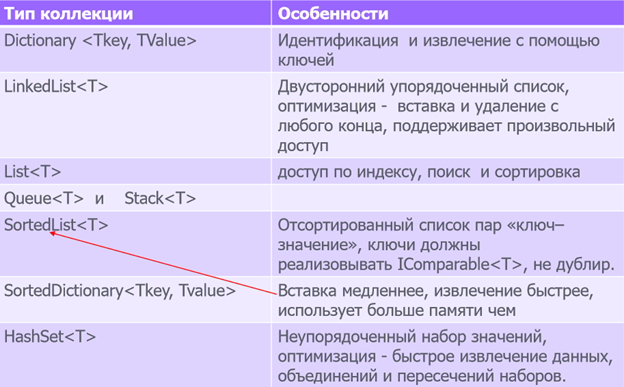
Реализуют ряд основных структур данных, включая динамический массив, стек, очередь, а также словари, в которых можно хранить пары "ключ-значение". В отношении необобщенных коллекций важно иметь в виду следующее: они оперируют данными типа object. Таким образом, необобщенные коллекции могут служить для хранения данных любого типа, причем в одной коллекции допускается наличие разнотипных данных.

Хранят ссылки на объекты, при сохранении или извлечении элементов требуется приведение типов (исключение BitArray). Включены в библиотеку с целью обратной совместимости с существующими приложениями применять не рекомендуется. B UWP эти классы недоступны.

**Специальные коллекции.**

Оперируют данными конкретного типа или же делают это каким-то особым образом. Например, имеются специальные коллекции для символьных строк, а также специальные коллекции, в которых используется однонаправленный список. Специальные коллекции объявляются в пространстве имен System.Collections.Specialized.

**Обобщенные коллекции** (Dictionary <Tkey, TValue>, LinkedList<T>, List<T>, Queue<T>, SortedDictionary<Tkey, Tvalue>, SortedList<T> (использовании памяти и в скорости вставки и удаления), HashSet<T> и SortedSet<T>, Stack<T>).



Обеспечивают обобщенную реализацию нескольких стандартных структур данных, включая связные списки, стеки, очереди и словари. Такие коллекции являются типизированными в силу их обобщенного характера. Это означает, что в обобщенной коллекции могут храниться только такие элементы данных, которые совместимы по типу с данной коллекцией. Благодаря этому исключается случайное несовпадение типов. Обобщенные коллекции объявляются в пространстве имен System.Collections.Generic.

**Параллельные коллекции.**

Поддерживают многопоточный доступ к коллекции. Это обобщенные коллекции, определенные в пространстве имен System.Collections.Concurrent.

В пространстве имен System.Collections.ObjectModel находится также ряд классов, поддерживающих создание пользователями собственных обобщенных коллекций. Коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений

**Поразрядная коллекция.**

В прикладном интерфейсе Collections API определена одна коллекция с поразрядной организацией — это BitArray. Коллекция типа BitArray поддерживает поразрядные операции, т.е. операции над отдельными двоичными разрядами, например И, ИЛИ, исключающее ИЛИ, а следовательно, она существенно отличается своими возможностями от остальных типов коллекций. Коллекция типа BitArray объявляется в пространстве имен System.Collections.

# ===43. Стандартные интерфейсы коллекций.

В C# коллекция представляет собой совокупность объектов.

Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.

Главное преимущество коллекций заключается в том, что они стандартизируют обработку групп объектов в программе. Все коллекции разработаны на основе набора четко определенных интерфейсов. Некоторые встроенные реализации таких интерфейсов, в том числе ArrayList, Hashtable, Stack и Queue, могут применяться в исходном виде и без каких-либо изменений. Имеется также возможность реализовать собственную коллекцию, хотя потребность в этом возникает крайне редко.

В среде .NET Framework поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

**IEnumerable<T>** — интерфейс IEnumerable необходим, когда с коллекцией используется оператор foreach. Этот интерфейс определяет метод GetEnumerator(), возвращающий перечислитель, который реализует IEnumerator.

**ICollection<T>** — это интерфейс, реализованный классами обобщенных коллекций. С его помощью можно получить количество элементов в коллекции (свойство Count) и скопировать коллекцию в массив (метод СоруТо()). Можно также добавлять и удалять элементы из коллекции (Add(), Remove(), Clear()).

**IList<T>** интерфейс IList<T> предназначен для создания списков, элементы которых доступны по своим позициям. Этот интерфейс определяет индексатор, а также способы вставки и удаления элементов в определенные позиции (методы Insert() и Remove()). IList<T> унаследован от ICollection<T>.

**ISet<T>** интерфейс ISet<T> появился в версии .NET 4. Этот интерфейс реализуется множествами. Он позволяет комбинировать различные множества в объединения, а также проверять, не пересекаются ли два множества. ISet<T> унаследован от ICollection<T>.

**IDictionary<TKey, TValue>** интерфейс IDictionary<TKey, TValue> реализуется обобщенными классами коллекций, элементы которых состоят из ключа и значения. С помощью этого интерфейса можно получать доступ ко всем ключам и значениям, извлекать элементы по индексатору типа ключа, а также добавлять и удалять элементы.

**ILookup<TKey, TValue>** подобно IDictionary<TKey, TValue> поддерживает ключи и значения. Однако в этом случае коллекция может содержать множественные значения для одного ключа.

**IComparer<T> интерфейс IComparer<T>** реализован компаратором и используется для сортировки элементов внутри коллекции с помощью метода Compare().

**IEqualityComparer<T>** интерфейс IEqualityComparer<T> реализован компаратором, который может быть применен к ключам словаря. Через этот интерфейс объекты могут быть проверены на предмет эквивалентности друг другу. В .NET 4 этот интерфейс также реализован массивами и кортежами.

**IProducerConsumerCollection<T>** интерфейс IProducerConsumerCollection<T> был добавлен в версию .NET 4 для поддержки новых, безопасных в отношении потоков классов коллекций.

# ===44. IEnumerable и IEnumerator

К элементам коллекции нередко приходится обращаться циклически, например, для отображения каждого элемента коллекции. С этой целью можно, с одной стороны, организовать цикл foreach, а с другой — воспользоваться перечислителем.

**Перечислитель** — это объект, который реализует необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T>.

Интерфейс **IEnumerator** представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле foreach. А интерфейс **IEnumerable** через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable(IEnumerable<T>) является базовым для всех коллекций.

**Интерфейс IEnumerator** содержит следующие методы и свойства:

**Метод** **MoveNext()** перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

**Свойство Current** возвращает объект в последовательности, на который указывает указатель.

**Метод Reset()** сбрасывает указатель позиции в начальное положение.

# ===45. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и не отложенные операции.

**LINQ** (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.LINQ.

Простейшее определение запроса LINQ выглядит следующим образом:

*from переменная in набор\_объектов*

*select переменная;*

**Грамматика выражений запросов:**

1. Начало - from
2. 0..\* from, let или where.
3. orderby, ascending или descending
4. select или group.
5. конструкции into, join, или повторение с п.2.

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

class LINQQueryExpressions

{

static void Main()

{

// Specify the data source.

int[] scores = new int[] { 97, 92, 81, 60 };

// Define the query expression.

IEnumerable<int> scoreQuery =

from score in scores

where score > 80

select score;

// Execute the query.

foreach (int i in scoreQuery)

{

Console.Write(i + " ");

}

}

}

// Output: 97 92 81

Есть два способа выполнения запроса LINQ: отложенное и немедленное выполнение.

**Отложенное выполнение**

При отложенном выполнении LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке. Рассмотрим отложенное выполнение:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// выполнение LINQ-запроса

foreach (string s in selectedTeams)

Console.WriteLine(s);

То есть фактическое выполнение запроса происходит не в строке определения: *var selectedTeams = from t...*, а при переборе в цикле *foreach*.

Фактически LINQ-запрос разбивается на три этапа:

Получение источника данных

Создание запроса

Выполнение запроса и получение его результатов

После определения запроса он может выполняться множество раз. И до выполнения запроса источник данных может изменяться.

Теперь выборка будет содержать два элемента, а не три, так как второй элемент после изменения не будет соответствовать условию.

Важно понимать, что переменная запроса сама по себе не выполняет никаких действий и не возвращает никаких данных. Она только хранит набор команд, которые необходимы для получения результатов. То есть выполнение запроса после его создания откладывается. Само получение результатов производится при переборе в цикле foreach.

**Немедленное выполнение запроса**

С помощью ряда методов мы можем применить немедленное выполнение запроса. Это методы, которые возвращают одно атомарное значение или один элемент. Например, Count(), Average(), First() / FirstOrDefault(), Min(), Max() и т.д. Например, метод Count() возвращает числовое значение, которое представляет количество элементов в полученной последовательности. А метод First() возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройтись по ней циклом foreach, который вызывается неявно внутри структуры запроса.

Рассмотрим пример с методом Count(), который возвращает число элементов последовательности:

string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};

// определение и выполнение LINQ-запроса

int i = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t select t).Count();

Console.WriteLine(i); //3

teams[1] = "Ювентус";

Console.WriteLine(i); //3

Результатом метода Count будет объект int, поэтому сработает немедленное выполнение.

Сначала создается запрос: from t in teams where t.ToUpper().StartsWith("Б") orderby t select t. Далее к нему применяется метод Count(), который выполняет запрос, неявно выполняет перебор по последовательности элементов, генерируемой этим запросом, и возвращает число элементов в этой последовательности.

Также для немедленного выполнения LINQ-запроса и кэширования его результатов мы можем применять методы преобразования ToArray<T>(), ToList<T>(), ToDictionary() и т.д.. Эти методы получают результат запроса в виде объектов Array, List и Dictionary соответственно.

# ===46. LINQ to Objects. Операции Where, Select, Take, OrderBy, Join, GroupBy

**Операция Where.**

Операция Where используется для фильтрации элементов в последовательность. Операция Where имеет два прототипа.

**Первый прототип Where.**

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, bool> predicate);

Этот прототип Where принимает входную последовательность и делегат метода-предиката, а возвращает объект, который при перечислении проходит по входной последовательности, выдавая элементы, для которых делегат метода-предиката возвращает true.

Поскольку это расширяющий метод, входная последовательность в действительности не передается, поскольку операция Where вызывается с использованием синтаксиса метода экземпляра.

При вызове Where передается делегат метода-предиката. Этот метод-предикат должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать bool.

Операция Where вызовет метод-предикат для каждого элемента входной последовательности и передаст ему этот элемент. Если метод-предикат вернет true, то Where выдаст этот элемент в выходную последовательность Where. Если метод-предикат вернет false, то Where этого не сделает.

**Второй прототип Where.**

public static IEnumerable<T> Where<T>(

this IEnumerable<T> source,

Func<T, int, bool> predicate);

Второй прототип Where идентичен первому, но с тем отличием, что он указывает на то, что делегат метода-предиката принимает дополнительный целочисленный аргумент. Этот аргумент будет индексом элемента во входной последовательности.

Нумерация индекса начинается с нуля, поэтому индексом первого элемента будет 0. Последний элемент имеет номер, соответствующий количеству элементов в последовательности минус 1.

**Select.**

Операция Select используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа. Эти типы не обязательно должны совпадать. Существуют два прототипа этой операции.

**Первый прототип Select.**

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, S> selector);

Этот прототип Select принимает входную последовательность и делегат метода-селектора в качестве входных параметров, а возвращает объект, который при перечислении проходит по входной последовательности и выдает последовательность элементов типа S. T и S могут быть как одного, так и разных типов.

При вызове Select делегат метода-селектора передается в аргументе selector. Метод-селектор должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа S. Операция Select вызовет метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Метод-селектор выберет интересующую часть входного элемента, создаст новый элемент — возможно, другого типа (даже анонимного) — и вернет его.

**Второй прототип Select.**

public static IEnumerable<S> Select<T, S>(

this IEnumerable<T> source,

Func<T, int, S> selector);

В этом прототипе операции Select методу-селектору передается дополнительный целочисленный параметр. Это индекс, начинающийся с нуля, входного элемента во входной последовательности.

**Take.**

Операция Take возвращает указанное количество элементов из входной последовательности, начиная с ее начала. Операция Take имеет один прототип, описанный ниже:

public static IEnumerable<T> Take<T>(

this IEnumerable<T> source,

int count);

Этот прототип указывает, что Take принимает входную последовательность и целое число count, задающее количество элементов, которые нужно вернуть, и возвращает объект, который при перечислении выдает первые count элементов из входной последовательности.

Если значение count больше количества элементов во входной последовательности, тогда каждый элемент из нее попадает в выходную последовательность.

**OrderBy.**

Операция OrderBy позволяет упорядочить входную последовательность на основе метода keySelector, который возвращает значение ключа для каждого входного элемента. Упорядоченная выходная последовательность IOrderedEnumerable<T> выдается в порядке возрастания на основе значений возвращенных ключей. Сортировка, выполненная операцией OrderBy, определена как неустойчивая. Это значит, что она не сохраняет входной порядок элементов.

Операция OrderBy имеет два прототипа.

**Первый прототип OrderBy.**

public static IOrderedEnumerable<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector)

where

К : IComparable<K>;

В этом прототипе операции OrderBy передается входная последовательность source и делегат метода keySelector, а возвращается объект, который при перечислении проходит входную коллекцию source, собирая все элементы и передавая каждый из них методу keySelector, таким образом, извлекая каждый ключ и упорядочивая последовательность на основе этих ключей.

Методу keySelector получает входной элемент типа T и возвращает поле внутри элемента, которое используется в качестве значения ключа типа К для этого входного элемента. Типы К и T могут быть одинаковыми или разными. Тип значения, возвращенного методом keySelector, должен реализовывать интерфейс IComparable.

**Второй прототип OrderBy.**

public static IOrderedEnumerable<T> OrderBy<T, K>(

this IEnumerable<T> source,

Func<T, K> keySelector,

IComparer<K> comparer);

Этот прототип такой же, как первый, за исключением того, что он позволяет передавать объект-компаратор. Если используется эта версия операции OrderBy, то нет необходимости в том, чтобы тип K реализовывал интерфейс IComparable.

**Join.**

Операция Join выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей, извлеченных из каждого элемента этих последовательностей.

public static IEnumerable<V> Join<T, U, K, V>(

this IEnumerable<T> outer,

IEnumerable<U> inner,

Func<T, K> outerKeySelector,

Func<U, K> innerKeySelector,

Func<T, U, V> resultSelector);

Первый аргумент метода имеет имя outer. Поскольку это расширяющий метод, последовательность, на которой вызывается операция Join, будет называться внешней последовательностью.

Операция Join возвращает объект, который при перечислении сначала проходит последовательность inner элементов типа U, вызывая метод innerKeySelector по одному разу для каждого элемента и сохраняя элемент, на который ссылается его ключ, в хеш-таблице. Затем он проходит последовательность outer элементов типа T. По мере того, как возвращаемый объект перечисляет каждый объект последовательности outer, он вызывает метод outerKeySelector для получения ключа и извлекает соответствующий элемент последовательности inner из хеш-таблицы, используя этот ключ.

**Операция GroupBy.**

Операции группирования помогают объединять вместе элементы последовательности по общему ключу. Операция GroupBy используется для группирования элементов входной последовательности.

Все прототипы операции GroupBy возвращают последовательность элементов IGrouping<K, Т>.

Здесь IGrouping<K, Т> - интерфейс, который определен, как показано ниже:

public interface IGrouping<K, Т> : IEnumerable<T>

{

K Key { get; }

}

Таким образом, IGrouping — это последовательность элементов типа T с ключами типа К.

# ===47. Рефлексия. System Type.

**Рефлексия** - это процесс обнаружения типов во время выполнения приложения. Например, рефлексия позволяет извлечь список всех типов, которые содержатся внутри определенной сборки \*.dll или \*.ехе (или даже внутри файла \*.netmodule если речь идет о многофайловой сборке), в том числе методов, полей, свойств и событий, определенных в каждом из них

Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой
* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

Эти классы представляют составные блоки типа и приложения: методы, свойства и т.д. Но чтобы получить информацию о членах типа, нам надо воспользоваться классом **System.Type**.

Класс **System.Type** представляет изучаемый тип, инкапсулируя всю информацию о нем. С помощью его свойств и методов можно получить эту информацию. Некоторые из его свойств и методов:

* Метод FindMembers() возвращает массив объектов MemberInfo данного типа
* Метод GetConstructors() возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
* Метод GetEvents() возвращает все события данного типа в виде массива объектов EventInfo
* Метод GetFields() возвращает все поля данного типа в виде массива объектов FieldInfo
* Метод GetInterfaces() получает все реализуемые данным типом интерфейсы в виде массива объектов Type
* Метод GetMembers() возвращает все члены типа в виде массива объектов MemberInfo
* Метод GetMethods() получает все методы типа в виде массива объектов MethodInfo
* Метод GetProperties() получает все свойства в виде массива объектов PropertyInfo
* Свойство IsAbstract возвращает true, если тип является абстрактным
* Свойство IsArray возвращает true, если тип является массивом
* Свойство IsClass возвращает true, если тип представляет класс
* Свойство IsEnum возвращает true, если тип является перечислением
* Свойство IsInterface возвращает true, если тип представляет интерфейс

Получение типа

Чтобы управлять типом и получать всю информацию о нем, нам надо сперва получить данный тип. Это можно сделать тремя способами:

* с помощью ключевого слова typeof,
  + Type myType = typeof(User);
* с помощью метода GetType() класса Object
  + Type myType = user.GetType();
* применяя статический метод Type.GetType().
  + Type myType = Type.GetType("TestConsole.User", false, true);

# ===48. Классы для работы с файловой системой.

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: **Directory и DirectoryInfo**.

**Класс Directory (СТАТИЧЕСКИЙ)**

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

* CreateDirectory(path) - создает каталог по указанному пути path
* Delete(path) - удаляет каталог по указанному пути path
* Exists(path) - определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
* GetDirectories(path): получает список каталогов в каталоге path
* GetFiles(path): получает список файлов в каталоге path
* Move(sourceDirName, destDirName): перемещает каталог
* GetParent(path): получение родительского каталога

**Класс DirectoryInfo**

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

* Create(): создает каталог
* CreateSubdirectory(path): создает подкаталог по указанному пути path
* Delete(): удаляет каталог
* Свойство Exists: определяет, существует ли каталог
* GetDirectories(): получает список каталогов
* GetFiles(): получает список файлов
* MoveTo(destDirName): перемещает каталог
* Свойство Parent: получение родительского каталога
* Свойство Root: получение корневого каталога

Создание каталога:

string path = @"C:\SomeDir";

string subpath = @"program\avalon";

DirectoryInfo dirInfo = new DirectoryInfo(path);

if (!dirInfo.Exists)

{ dirInfo.Create(); }

dirInfo.CreateSubdirectory(subpath);

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов **File и FileInfo**. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Методы и свойства класса **FileInfo**:

* CopyTo(path): копирует файл в новое место по указанному пути path
* Create(): создает файл
* Delete(): удаляет файл --- объект\_файла.Delete()
* MoveTo(destFileName): перемещает файл в новое место
* Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
* Свойство DirectoryName: получает полный путь к родительскому каталогу
* Свойство Exists: указывает, существует ли файл
* Свойство Length: получает размер файла
* Свойство Extension: получает расширение файла
* Свойство Name: получает имя файла
* Свойство FullName: получает полное имя файла

Класс **File**: (СТАТИЧЕСКИЙ) - т.е. не привязан к конкретному файлу:

* Copy(): копирует файл в новое место - File.Copy(path)
* Create(): создает файл
* Delete(): удаляет файл --- File.Delete(path)
* Move: перемещает файл в новое место
* Exists(file): определяет, существует ли файл

Получение информации о файле

string path = @"C:\apache\hta.txt";

FileInfo fileInf = new FileInfo(path);

if (fileInf.Exists)

{

Console.WriteLine("Имя файла: {0}", fileInf.Name);

Console.WriteLine("Время создания: {0}", fileInf.CreationTime);

Console.WriteLine("Размер: {0}", fileInf.Length);

}

# ===49. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.

Интерфейс *IDisposable* объявляет один единственный метод *Dispose*, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов.

Синтаксис C# также предлагает синонимичную конструкцию для автоматического вызова метод *Dispose* - конструкцию **using**:

*using (Font font1 = new Font("Arial", 10.0f))*

*{*

*byte charset = font1.GdiCharSet;*

*}*

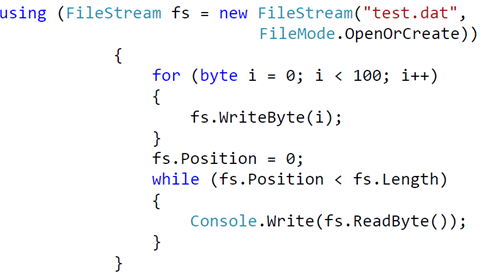
Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

* Свойство Length: возвращает длину потока в байтах
* Свойство Position: возвращает текущую позицию в потоке
* Метод Read: считывает данные из файла в массив байтов. Принимает три параметра: int Read(byte[] array, int offset, int count) и возвращает количество успешно считанных байтов. Здесь используются следующие параметры:
  + array - массив байтов, куда будут помещены считываемые из файла данные
  + offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  + count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
* Метод long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
* Метод Write: записывает в файл данные из массива байтов. Принимает три параметра: Write(byte[] array, int offset, int count)
  + array - массив байтов, откуда данные будут записываться в файла
  + offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  + count - максимальное число байтов, предназначенных для записи

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.



*Поток данных* – это абстрактное представление данных в виде последовательности байт.

Классы для работы с потоками, связанными с хранилищами:

* **FileStream** – класс для работы с файлами, как с потоками (System.IO).
* **MemoryStream** – класс для представления потока в памяти (System.IO).
* **NetworkStream** – работа с сокетами, как с потоками (System.Net.Sockets).
* **PipeStream** – абстрактный класс из пространства имён System.IO.Pipes, базовый для классов-потоков, которые позволяют передавать данные между процессами операционной системы.

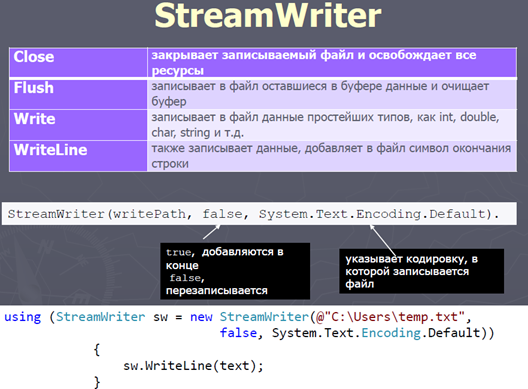
# ===50. Классы адаптеры потоков.

Адаптеры потоков служат оболочкой потока, преобразуя информация определенного формата в набор байтов (сами адаптеры потоками не являются).

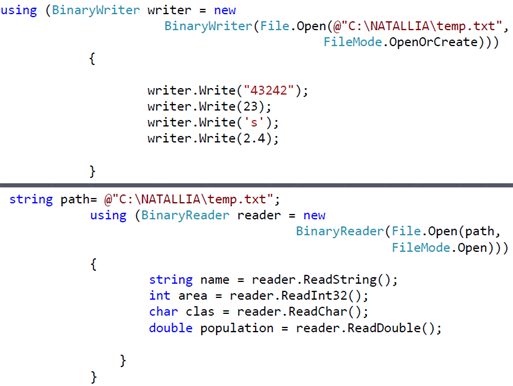
* Текстовые адаптеры (для типов string и char):
  + TextReader
  + TextWriter
  + StreamReader
    - Close - закрывает считываемый файл и освобождает все ресурсы
    - Peek - возвращает следующий доступный символ или -1
    - Read - считывает и возвращает следующий символ в численном представлении.
    - ReadLine - считывает одну строку в файле
    - ReadToEnd - считывает весь текст из файла



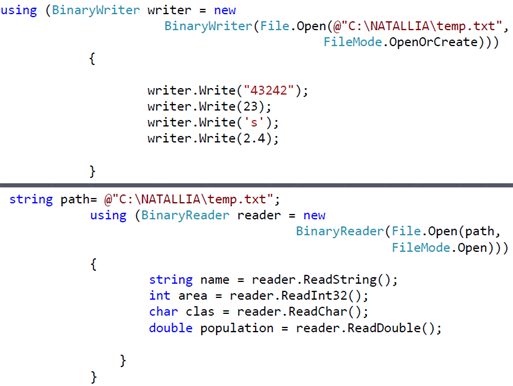
* + StreamWriter
    - Close - закрывает записываемый файл и освобождает все ресурсы
    - Flush - записывает в файл оставшиеся в буфере данные и очищает буфер
    - Write - записывает в файл данные простейших типов, как int, double, char, string и т.д.
    - WriteLine - также записывает данные, добавляет в файл символ окончания строки



* + StringReader
  + StringWriter
* Двоичные адаптеры (для типов int, bool, string и float):
  + BinaryReader
    - Close - закрывает поток и освобождает ресурсы
    - ReadBoolean - считывает значение bool и перемещает указатель на один байт считывает один байт и перемещает указатель на один байт
    - ReadChar - считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
    - ReadDecimal - считывает значение decimal и перемещает указатель на 16 байт
    - ReadDouble - считывает значение double и перемещает указатель на 8 байт
    - ReadInt16 - считывает значение short и перемещает указатель на 2 байта
    - ReadInt32 - считывает значение int и перемещает указатель на 4 байта
    - ReadInt64 - считывает значение long и перемещает указатель на 8 байт
    - ReadSingle - считывает значение float и перемещает указатель на 4 байта
    - ReadString - считывает значение string. Каждая строка предваряется значением длины строки



* + BinaryWriter
    - Close - закрывает поток и освобождает ресурсы
    - Flush - очищает буфер, дописывая из него оставшиеся данные в файл
    - Seek - устанавливает позицию в потоке
    - Write - записывает данные в поток



* Адаптеры XML:
  + XmlReader
  + XmlWriter

# ===51. Сериализация. Форматы сериализации.

**Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом Serializable:

[Serializable]

class Person

{

}

При отсутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение *SerializationException*.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом *NonSerialized*:

[NonSerialized]

public string accNumber;

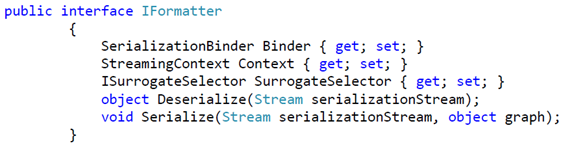
При наследовании подобного класса атрибут Serializable автоматически не наследуется.

Форматы сериализации:

**BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

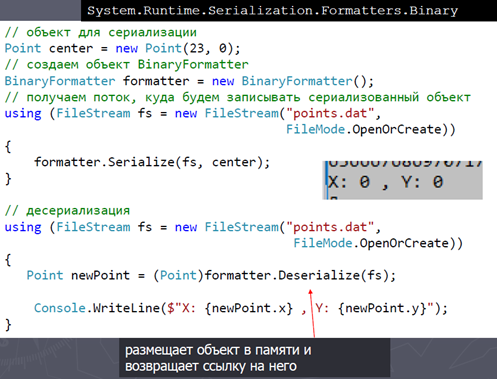
* Бинарный (класс *BinaryFormatter*) (это же и есть двоичная сериализация *мистер кэп*)
* SOAP (класс *SoapFormatter*)
* xml (класс *XmlSerializer*)
* JSON (класс *DataContractJsonSerializer*)

Для классов *BinaryFormatter* и *SoapFormatter* сам функционал сериализации определен в интерфейсе **IFormatter**:



BinaryFormatter сериализация:

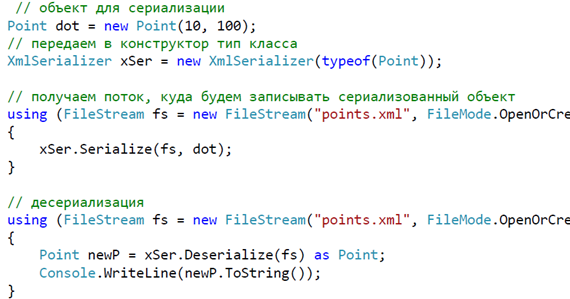
сохраняются данные полей объектов, полное квалифицированное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер общедоступного ключа и культура)

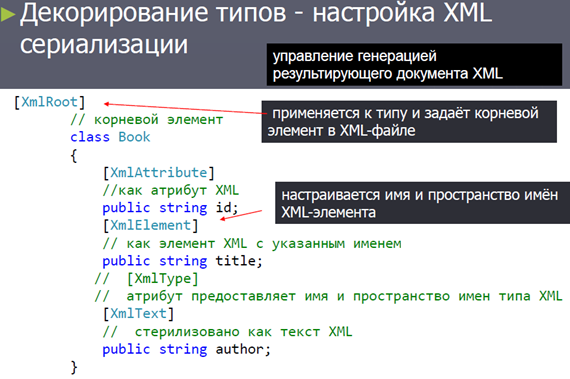


XmlSerializer сериализация

Ограничения:

* Класс должен иметь конструктор без параметров
* Сериализации подлежать только открытые члены
* Приватные данные, не представленные свойствами, игнорируются.
* Требует указания типа





# ===52. Сериализация контрактов данных. интерфейс ISerializable.

*Контракт данных* - формальное соглашение между службой и клиентом, абстрактно описывающее данные, обмен которыми происходит. Это значит, что для взаимодействия клиент и служба не обязаны совместно использовать одни и те же типы, достаточно совместно использовать одни и те же контракты данных.

*Контракт данных* – это тип (класс или структура), объект которого описывает информационный фрагмент (открытые поля и свойства).

Использование сериализатора контрактов данных:

* выбрать класс для использования: DataContractSerializer или NetDataContractSerializer
* добавить сериализуемым типам атрибут [DataContract], а их членам, которые необходимо включить в сериализацию - [DataMember]
* создать экземпляр сериализатора и вызвать его метод WriteObject или ReadObject

Существует два сериализатора контрактов данных:

* DataContractSerializer — обеспечивает слабую привязку типов .NET к типам контрактов данных. Может генерировать совместимый со стандартами XML код. Требует предварительной явной регистрации сериализуемых производных типов, чтобы иметь возможность сопоставлять имена контрактов данных с именами типов .NET
* NetDataContractSerializer — характеризуется тесной привязкой типов .NET к типам контрактов данных, не требует явной регистрации сериализуемых производных типов, т.к. самостоятельно записывает полные имена типов и сборок сериализуемых типов

*namespace SerialTest*

*{*

*[DataContract] public class Person*

*{*

*[DataMember] public string Name;*

*[DataMember] public int Age;*

*}*

*}*

// …..

*Person p = new Person { Name = "Stacey", Age = 30 };*

*var ds = new DataContractSerializer (typeof (Person));*

*using (Stream s = File.Create ("person.xml"))*

*ds.WriteObject (s, p); // Сериализация*

*Person p2;*

*using (Stream s = File.OpenRead ("person.xml"))*

*p2 = (Person) ds.ReadObject (s); // Десериализация*

*Console.WriteLine (p2.Name + " " + p2.Age); // Stacey 30*

Конструктору DataContractSerializer необходимо передать тип корневого объекта — сериализуемый объект, который в XML дереве будет корневым элементом. Конструктор NetDataContractSerializer этого не требует. В остальном их применение аналогично.

Оба типа сериализаторов по умолчанию принимаю форматер XML, поэтому с ними можно использовать XmlWriter:

*Person p = new Person { Name = "Stacey", Age = 30 };*

*var ds = new DataContractSerializer (typeof (Person));*

*// Добавить в вывод отступы:*

*XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };*

*using (XmlWriter w = XmlWriter.Create ("person.xml", settings))*

*ds.WriteObject (w, p);*

*System.Diagnostics.Process.Start ("person.xml");*

Имена XML элементов в выводе соответствуют именам контрактов данных, которые по умолчанию совпадают с именами типов в .NET, однако это можно переопределить с помощью атрибутов, задав альтернативное имя для элемента:

*[DataContract (Name="Candidate")]*

*public class Person { ... }*

*Также можно изменить пространство имен по умолчанию для корневого элемента:*

*[DataContract (Namespace="http://somedomain.com/contacts")]*

*public class Person { ... }*

*Можно также переопределить имена членов данных:*

*[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]*

*public class Person*

*{*

*[DataMember (Name="FirstName")] public string Name;*

*[DataMember (Name="ClaimedAge")] public int Age;*

*}*

**Интерфейс ISerializable**

Реализация интерфейса **ISerializable** предоставляет типу полный контроль над тем, как производится его двоичная сериализация и десериализация. Определение интерфейса выглядит следующим образом:

*public interface ISerializable*

*{*

*void GetObjectData (SerializationInfo info, StreamingContext context);*

*}*

Метод **GetObjectData** запускается при сериализации и наполняет объект SerializationInfo (словарь пар имя-значение) данными из всех полей, подлежащих сериализации.

Рекомендуется объявлять метод GetObjectData как *virtual*, чтобы производные классы могли расширять сериализацию не реализуя заново интерфейс ISerializable.

Параметры GetObjectData:

* SerializationInfo - представляет из себя нечто похожее на коллекцию "Dictionary". Сюда с помощью метода "AddValue()" добавляется ключ и значение. А при десериализации при помощи “GetValue()” это значение будет вытаскиваться по ключу.
* StreamingContext - это
  + структура, помимо прочего содержащая значения enum, указывающие откуда поступает сериализованный экземпляр
  + объект, который содержит дополнительную информацию о потоке, в который записываются эти данные

*(выбирайте сами определение, первое из ответов на билеты, но второе мне больше нравится)*

Помимо реализации интерфейса ISerializable, тип, управляющий собственной сериализацией *должен содержать конструктор десериализации*, принимающий те же два параметра, что и метод GetObjectData. Он может быть объявлен с любым модификатором доступа.

*[Serializable]*

*public class Team : ISerializable*

*{*

*public string Name;*

*public List<Person> Players;*

*public virtual void GetObjectData (SerializationInfo si, StreamingContext sc)*

*{*

*si.AddValue ("Name", Name);*

*si.AddValue ("PlayerData", Players.ToArray());*

*}*

*public Team() {}*

*// Конструктор десериализации*

*protected Team (SerializationInfo si, StreamingContext sc)*

*{*

*Name = si.GetString ("Name");*

*Person[] a = (Person[]) si.GetValue ("PlayerData", typeof (Person[]));*

*Players = new List<Person> (a);*

*}*

*}*

# ===53. Атрибуты. Создание собственного атрибута.

**Атрибуты** позволяют добавлять *пользовательскую информацию* к метаданным элементов кода: сборкам, типам, членам, возвращаемым значениям и параметрам. Например, атрибуты могут быть использованы для *сериализации* (а именно [Serializable] и [NonSerialized]) — процесс преобразования произвольных объектов в определенный формат. В этом случае атрибуты полей могут точно устанавливать как преобразовывать поля.

В .NET атрибуты представляют собой типы классов, которые расширяют абстрактный базовый класс System.Attribute.

Для создания собственного атрибута нужно:

* создать класс, производный от System.Attribute или его потомка; по соглашению имя класса должно заканчиваться словом Attribute (но не обязательно)
* применить к этому классу описанный выше атрибут AttributeUsage. *Если атрибут не требует свойств или аргументов конструктора, на этом его создание завершено*
  + Тут же указываются ограничения для атрибута (т.е. где он может применяться). Их задает перечисление AttributeTargets, значения которого могут быть:
    - All: используется всеми типами
    - Assembly: атрибут применяется к сборке
    - Constructor: атрибут применяется к конструктору
    - Delegate: атрибут применяется к делегату
    - Enum: применяется к перечислению
    - Event: атрибут применяется к событию
    - Field: применяется к полю типа
    - Interface: атрибут применяется к интерфейсу
    - Method: применяется к методу
    - Property: применяется к свойству
    - Struct: применяется к структуре

Пример задавания ограничения: [AttributeUsage (AttributeTargets.Method)]

* добавить в класс один или несколько публичных конструкторов. Параметры конструктора определяют позиционные параметры конструктора и становятся обязательными при использовании атрибута.

Параметры атрибута могут быть:

* + примитивные типы: bool, byte, char, double, float, int, long, short или string
  + Type
  + enum
  + одномерный массив любого из указанных выше типов
* объявить публичное свойство или поле для каждого именованного параметра атрибута

Пример:

*[AttributeUsage (AttributeTargets.Method)] // <= атрибут с ограничением*

*public sealed class TestAttribute : Attribute*

*{*

*public int Repetitions; // публичное свойство для int repetitions*

*public string FailureMessage;*

*public TestAttribute () : this (1) { } // конструктор*

*public TestAttribute (int repetitions) // еще один конструктор с позиционными параметрами*

*{*

*Repetitions = repetitions;*

*}*

*}*Использование атрибута:

*class Foo*

*{*

*[Test]*

*public void Method1() { ... }*

*[Test(20)] // атрибут с параметром int repetitions*

*public void Method2() { ... }*

*[Test(20, FailureMessage="Debugging Time!")]*

*public void Method3() { ... }*

*}*

# ===54. Процесс. Домен приложений. Поток выполнения.

*вопрос на самом деле небольшой, просто свойств и методов в классах много*

При запуске приложения операционная система создает для него отдельный **процесс**, которому выделяется определённое адресное пространство в памяти и который изолирован от других процессов. Процесс может иметь несколько потоков. Как минимум, процесс содержит один - главный поток, создаваемый при вызове метода Main. А из главного потока могут запускаться вторичные потоки.

В .NET процесс представлен классом **Process** из пространства имен **System.Diagnostics**. Этот класс позволяет управлять уже запущенными процессами, а также запускать новые. В данном классе определено ряд свойств и методов, позволяющих получать информацию о процессах и управлять ими:

* Свойство *Handle*: возвращает дескриптор процесса
* Свойство *Id*: получает уникальный идентификатор процесса в рамках текущего сеанса ОС
* Свойство *MachineName*: возвращает имя компьютера, на котором запущен процесс
* Свойство *Modules*: получает доступ к коллекции ProcessModuleCollection, которая хранит набор модулей (файлов dll и exe), загруженных в рамках данного процесса
* Свойство *ProcessName*: возвращает имя процесса, которое нередко совпадает с именем приложения
* Свойство *StartTime*: возвращает время, когда процесс был запущен
* Свойство *VirtualMemorySize64*: возвращает объем памяти, который выделен для данного процесса
* Метод *CloseMainWindow()*: закрывает окно процесса, который имеет графический интерфейс
* Метод *GetProcesses()*: возвращающий массив всех запущенных процессов
* Метод *GetProcessesByName()*: возвращает процессы по его имени. Так как можно запустить несколько копий одного приложения, то возвращает массив
* Метод *Kill()*: останавливает процесс
* Метод *Start()*: запускает новый процесс

**Домен приложения** — это своеобразный контейнер, внутри которого запускается программа, и который изолирует ее во время выполнения. Домен задает границы управляемой памяти для приложения и является контейнером для загруженных сборок и параметров конфигурации приложения. Каждый процесс *обычно размещает только один домен приложения* — стандартный домен, автоматически созданный CLR при запуске процесса. Однако в рамках одного процесса возможно создавать дополнительные домены приложений.

Домены:

1) существуют внутри процессов

2) содержат загруженные сборки

3) процесс запускает при старте домен по умолчанию (AppDomain.CurrentDomain)

4) домены могут создаваться и уничтожаться в ходе работы в рамках процесса (менее затраты по сравн. с процессами)

5) обеспечивают уровень изоляции кода

Для управления домена платформа .NET предоставляет класс **AppDomain**. Рассмотрим некоторые основные методы и свойства данного класса:

* Свойство *BaseDirectory*: базовый каталог, который используется для получения сборок (как правило, каталог самого приложения)
* Свойство *CurrentDomain*: домен текущего приложения
* Свойство *FriendlyName*: имя домена приложения
* Свойство *SetupInformation*: представляет объект AppDomainSetup и хранит конфигурацию домена приложения
* Метод *CreateDomain()*: статический метод, позволяющий создать новый домен приложения в рамках текущего процесса
* Метод *CreateInstance()*: создает экземпляр типа из внешней сборки после ее загрузки в домен приложения
* Метод *ExecuteAssembly()*: запускает сборку exe в рамках текущего домена приложения
* Метод *GetAssemblies()*: получает набор сборок .NET, загруженных в домен приложения
* Метод *Load()*: загружает сборку домена приложения
* Метод *Unload()*: выгружает домен приложения из определенного процесса

**Потоки процесса**

Свойство Threads представляет коллекцию потоков процесса - объект ProcessThreadCollection, каждый поток в которой является объектом **ProcessThread**. В данном классе можно выделить следующие свойства:

* *CurrentPriority*: возвращает текущий приоритет потока
* *Id*: идентификатор потока
* *IdealProcessor*: позволяет установить процессор для обработки потока
* *PriorityLevel*: уровень приоритета потока
* *StartAddress*: адрес в памяти функции, запустившей поток
* *StartTime*: время запуска потока

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Класс **Thread** определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства и методы класса:

* Статическое свойство *CurrentContext* позволяет получить контекст, в котором выполняется поток
* Статическое свойство *CurrentThread* возвращает ссылку на выполняемый поток
* Свойство *IsAlive* указывает, работает ли поток в текущий момент
* Свойство *IsBackground* указывает, является ли поток фоновым
* Свойство *Name* содержит имя потока
* Свойство *Priority* хранит приоритет потока - значение перечисления ThreadPriority
* Свойство *ThreadState* возвращает состояние потока - одно из значений перечисления ThreadState
* Статический метод *GetDomain()* возвращает ссылку домен приложения
* Статический метод *GetDomainId()* возвращает id домена приложения, в котором выполняется текущий поток
* Статический метод *Sleep()* останавливает поток на определенное количество миллисекунд
* Метод *Abort()* уведомляет среду CLR о том, что надо прекратить поток, однако прекращение работы потока происходит не сразу, а только тогда, когда это становится возможно. Для проверки завершенности потока следует опрашивать его свойство ThreadState
* Метод *Interrupt()* прерывает поток на некоторое время
* Метод *Join()* блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
* Метод *Resume()* возобновляет работу ранее приостановленного потока
* Метод *Start()* запускает поток
* Метод *Suspend()* приостанавливает поток

# ===55. Создание потоков , классы приоритетов. Состояния потоков

Используя класс Thread, мы можем выделить в приложении несколько потоков, которые будут выполняться одновременно. Во-первых, для запуска нового потока нам надо определить задачу в приложении, которую будет выполнять данный поток. Для этого мы можем добавить новый метод, производящий какие-либо действия.

Для создания нового потока используется делегат ThreadStart, который получает в качестве параметра метод, который мы определил выше.

И чтобы запустить поток, вызывается метод Start. Рассмотрим на примере:

using System.Threading;

class Program

{

static void Main(string[] args)

{

// создаем новый поток

Thread myThread = new Thread(new ThreadStart(Count));

myThread.Start(); // запускаем поток

for (int i = 1; i < 9; i++)

{

Console.WriteLine("Главный поток:");

Console.WriteLine(i \* i);

Thread.Sleep(300);

}

Console.ReadLine();

}

public static void Count()

{

for (int i = 1; i < 9; i++)

{

Console.WriteLine("Второй поток:");

Console.WriteLine(i \* i);

Thread.Sleep(400);

}

}

}

Здесь новый поток будет производить действия, определенные в методе Count. В данном случае это возведение в квадрат числа и вывод его на экран. И после каждого умножения с помощью метода Thread.Sleep мы усыпляем поток на 400 миллисекунд.

Чтобы запустить этот метод в качестве второго потока, мы сначала создаем объект потока: Thread myThread = new Thread(new ThreadStart(Count));. В конструктор передается делегат ThreadStart, который в качестве параметра принимает метод Count. И следующей строкой myThread.Start() мы запускаем поток. После этого управление передается главному потоку, и выполняются все остальные действия, определенные в методе Main.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток. Кроме действий по созданию второго потока, в главном потоке также производятся некоторые вычисления. Как только все потоки отработают, программа завершит свое выполнение.

Существует еще одна форма создания потока: Thread myThread = new Thread(Count); Хотя в данном случае явным образом мы не используем делегат ThreadStart, но неявно он создается. Компилятор C# выводит делегат из сигнатуры метода Count и вызывает соответствующий конструктор.

Приоритеты потоков располагаются в перечислении ThreadPriority:

· Lowest

· BelowNormal

· Normal

· AboveNormal

· Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

Состояния потоков

Свойство ThreadState возвращает состояние потока - одно из значений перечисления ThreadState

# ===56. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

class Program

{

static int x=0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

}

Здесь у нас запускаются пять потоков, которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово lock. Оператор lock определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

class Program

{

static int x=0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

lock (locker)

{

x = 1;

for(int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

}

}

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Когда выполнение доходит до оператора lock, объект locker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект locker освобождается и становится доступным для других потоков.

Monitor

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor. Фактически конструкция оператора lock из прошлой темы инкапсулирует в себе синтаксис использования мониторов. А рассмотренный в прошлой теме пример будет эквивалентен следующему коду:

class Program

{

static int x=0;

static object locker = new object();

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

myThread.Name = "Поток " + i.ToString();

myThread.Start();

}

Console.ReadLine();

}

public static void Count()

{

try

{

Monitor.Enter(locker);

x = 1;

for(int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

X++;

Thread.Sleep(100);

}

}

Finally {

Monitor.Exit(locker);

}

}

}

Метод Monitor.Enter блокирует объект locker так же, как это делает оператор lock. А в блоке try...finally с помощью метода Monitor.Exit Происходит освобождение объекта locker, и он становится доступным для других потоков.

Кроме блокировки и разблокировки объекта класс Monitor имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод Monitor.Wait освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки. Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод Monitor.PulseAll отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

Мьютекс

System.Threading.Mutex - позволяет организовать критическую секцию для нескольких процессов

WaitOne() - входа в критическую секцию

ReleaseMutex() – для выхода из неё (выход может быть произведен только в том же потоке выполнения, что и вход).

Семафор

объект синхронизации, позволяющий войти в заданный участок кода не более чем N потокам (N – ёмкость семафора) получение и снятие блокировки в случае семафора может выполняться из разных потоков

**классы**

System.Threading.Semaphore (между процессами) и SemaphoreSlim(в рамках одного процесса)

Wait()-получение блокировки

Release()–снятие блокировки

# ===57. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.

Библиотека параллельных задач TPL (Task Parallel Library) позволяет распараллелить задачи и выполнять их сразу на нескольких процессорах (для создания многопоточных приложений)

using System.Threading.Tasks

Планировщик библиотеки выполняет диспетчеризацию задач, а также предоставляет единообразный механизм отмены задач и обработки исключительных ситуаций.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

Для определения и запуска задачи можно использовать различные способы. Первый способ создание объекта Task и вызов у него метода Start:

1 Task task = new Task(() => Console.WriteLine("Hello Task!"));

2 task.Start();

В качестве параметра объект Task принимает делегат Action, то есть мы можем передать любое действие, которое соответствует данному делегату, например, лямбда-выражение, как в данном случае, или ссылку на какой-либо метод. То есть в данном случае при выполнении задачи на консоль будет выводиться строка "Hello Task!", а метод Start() собственно запускает задачу.

Второй способ заключается в использовании статического метода Task.Factory.StartNew(). Этот метод также в качестве параметра принимает делегат Action, который указывает, какое действие будет выполняться. При этом этот метод сразу же запускает задачу:

1 Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));

В качестве результата метод возвращает запущенную задачу.

Третий способ определения и запуска задач представляет использование статического метода Task.Run():

1 Task task = Task.Run(() => Console.WriteLine("Hello Task!"));

Метод Task.Run() также в качестве параметра может принимать делегат Action - выполняемое действие и возвращает объект Task.

Свойства класса Task

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

· AsyncState: возвращает объект состояния задачи

· CurrentId: возвращает идентификатор текущей задачи

· Exception: возвращает объект исключения, возникшего при выполнении задачи

· Status: возвращает статус задачи

# ===58. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.

*Способы создания расписаны в 57 вопросе*

**Возврат результата:**

Задача может возвращать значение. Это очень удобно по двум причинам:

* с помощью задачи можно вычислить некоторый результат. Подобным образом *поддерживаются параллельные вычисления.*
* вызывающий процесс окажется заблокированным до тех пор, пока не будет получен результат. Это означает, что для *организации ожидания результата не требуется никакой особой синхронизации*.

Для того, чтобы возвратить результат из задачи, достаточно создать эту задачу, используя обобщенную форму Task<TResult> класса Task. Ниже приведены два конструктора этой формы класса Task:

*public Task(Func<TResult> функция)*

*public Task(Func<Object, TResult> функция, Object состояние)*

Пример возврата значения из задачи:

*static bool MyTask() { /\* какой-то код функции \*/ }* // определение метода MyTask

// … далее пишется в Main()

*Task<bool> tsk = Task<bool>.Factory.StartNew(MyTask); // в tsk результат задачи*

// … это в самом конце Main()

*tsk.Dispose();* // уничтожение объекта

**Отмена выполнения задач:**

*лучше сразу смотреть пример имо. Так непонятно. Еще лучше проверить в вижле.*

Признак *отмены* является экземпляром объекта типа **CancellationToken**, т.е. структуры, определенной в пространстве имен **System.Threading**. В структуре CancellationToken определено несколько свойств и методов, но мы рассмотрим два из них:

* доступное только для чтения свойство **IsCancellationRequested** - возвращает true, если отмена задачи была запрошена для вызывающего признака, иначе false
* Метод **ThrowIfCancellationRequested()** - выкидывает ошибку, если задача была отменена

*CancellationTokenSource cancelTokenSource = new CancellationTokenSource(); // создаем объект CancellationTokenSource*

*CancellationToken token = cancelTokenSource.Token; // берез токен объекта*

*Task task1 = new Task(() =>*

*{*

*while (true)*

*{*

*if (token.IsCancellationRequested) // проверка, если потребовалась отмена*

*{*

*Console.WriteLine("Операция прервана");*

*return; // задача прерывается*

*}*

*Console.WriteLine("...");*

*Thread.Sleep(500); // многоточие пишется каждые полсекунды*

*}*

*});*

*task1.Start(); // задача начинается*

*while (true) {*

*ConsoleKeyInfo s = Console.ReadKey(); // если ввести “q”, то задача прекратится*

*if (s.KeyChar == 'q')*

*{*

*cancelTokenSource.Cancel(); // отправляет запрос на отмену задачи*

*break;*

*}*

*}*

*Console.WriteLine("Введите клавишу, чтобы завершить программу...");*

*Console.ReadKey();*

**Продолжение задачи** (continuation task): сообщает задаче, что после её завершения она должна продолжить делать что-то другое

static void Main(string[] args)

{

Task task1 = new Task(() => {

Console.WriteLine("1 задача");

});

// задача продолжения

Task task2 = task1.ContinueWith(Display);

task1.Start();

// ждем окончания второй задачи

task2.Wait();

Console.WriteLine("Выполняется работа метода Main");

Console.ReadLine();

}

static void Display(Task t)

{

Console.WriteLine("2 задача");

Thread.Sleep(3000);

}

}

# ===59. Параллелизм при императивной обработке данных. Класс Parallel

Класс Parallel также является частью TPL и предназначен для упрощения параллельного выполнения кода. Parallel имеет ряд методов, которые позволяют распараллелить задачу.

Одним из методов, позволяющих параллельное выполнение задач, является метод Invoke:

static void Main(string[] args)

{

Parallel.Invoke(Display,

() => {

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

},

() => Factorial(5));

Console.ReadLine();

}

static void Display()

{

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

}

static void Factorial(int x)

{

int result = 1;

for (int i = 1; i <= x; i++)

{

result \*= i;

}

Console.WriteLine("Выполняется задача {0}", Task.CurrentId);

Thread.Sleep(3000);

Console.WriteLine("Результат {0}", result);

}

Метод Parallel.Invoke в качестве параметра принимает массив объектов Action, то есть мы можем передать в данный метод набор методов, которые будут вызываться при его выполнении. Количество методов может быть различным, но в данном случае мы определяем выполнение трех методов. Опять же как и в случае с классом Task мы можем передать либо название метода, либо лямбда-выражение. И таким образом, при наличии нескольких ядер на целевой машине данные методы будут выполняться параллельно на различных ядрах.

**Parallel.For**

Метод Parallel.For позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: For(int, int, Action<int>), где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр - делегат Action - указывает на метод, который будет выполняться один раз за итерацию.

**Parallel.ForEach**

Метод Parallel.ForEach осуществляет итерацию по коллекции, реализующей интерфейс IEnumerable, подобно циклу foreach, только осуществляет параллельное выполнение перебора. Он имеет следующее определение: ParallelLoopResult ForEach<TSource>(IEnumerable<TSource> source, Action<TSource> body), где первый параметр представляет перебираемую коллекцию, а второй параметр - делегат, выполняющийся один раз за итерацию для каждого перебираемого элемента коллекции.

Здесь метод Factorial, обрабатывающий каждую итерацию, принимает дополнительный параметр - объект ParallelLoopState. И если счетчик в цикле достигнет значения 5, вызывается метод Break. Благодаря чему система осуществит выход и прекратит выполнение метода Parallel.For при первом удобном случае.

Методы Parallel.ForEach и Parallel.For возвращают объект ParallelLoopResult, наиболее значимыми свойствами которого являются два следующих:

· IsCompleted: определяет, завершилось ли полное выполнение параллельного цикла

· LowestBreakIteration: возвращает индекс, на котором произошло прерывание работы цикла

# ===60. Асинхронные методы. async и await

Операторы async и await используются вместе для создания асинхронного метода. Такой метод, определенный с помощью модификатора async и содержащий одно или несколько выражений await, называется **асинхронным методом.**

При асинхронном вызове поток выполнения разделяется на две части:

* в одной выполняется метод,
* а в другой –процесс программы.

Асинхронный вызов методов реализуется средой исполнения при помощи пула

Потоков

**Назначение:**

* Доступ к web HttpClient
* Работа с файлами
* Работа с изображениями
* WCF программирование

Ключевое слово **async** указывает, что метод или лямбда-выражение являются асинхронными. А оператор **await** применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

class Program

{

static void Main(string[] args)

{

DisplayResultAsync();

Console.ReadLine();

}

static async void DisplayResultAsync()

{

int num = 5;

int result = await FactorialAsync(num);

Thread.Sleep(3000);

Console.WriteLine("Факториал числа {0} равен {1}", num, result);

}

static Task<int> FactorialAsync(int x)

{

int result = 1;

return Task.Run(() =>

{

for (int i = 1; i <= x; i++)

{

result \*= i;

}

return result;

});

}

}

Итак, здесь определен метод, возвращающий объект Task - задачу, которая выполняется асинхронно. И так так мы получаем факториал целого числа, то объект задачи типизирован типом int.

Сам факториал мы получаем в асинхронном методе DisplayResultAsync. Асинхронным он является потому, что он объявлен с модификатором async и содержит использование ключевого слова **await.**

По негласным правилам в названии асинхроннных методов принято использовать суффикс Async.

# ===61. Проектирование отношений. Агрегация, композиция и ассоциация

UML (Unified Modeling Language) - унифицированный язык моделирования. Он определил следующие типы связей:

* **Ассоциации** - отношение, где два класса как-то связаны между собой, т.е. один включен в другой. Типы ассоциаций:
  + **Агрегация** - это ассоциация между классом A (часть) и классом B (целое), которая означает, что экземпляры (один или несколько) класса A ходят в состав экземпляра класса B.

Объекты могут быть равноправными; один *включает* в себя другой; целое хоть и содержит свою составную часть, *время их жизни не связано*

*Агрегация* может быть частью нескольких объектов.

* + **Композиция** - это ассоциация между классом A (часть) и классом B (целое), которая дополнительно накладывает более сильные ограничения в сравнении с агрегацией: композиционно часть A может входить только в одно целое B, часть существует, только пока существует целое и прекращает свое существование вместе с целым.

Объекты *не равноправны*; один *состоит* из другого; целое *контролирует время жизни* своей составной части (часть не может существовать без целого)

*Композиция* может быть частью одного и только одного целого.

* **зависимости**
* **обобщения** (наследования)

Отношение *обобщения* часто применяется на диаграмме классов. Действительно, трудно представить себе ситуацию, когда между классами в одной системе нет ничего общего. Как правило, общее есть, и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами.

Хороший пример для понимания различия агрегации и композиции.

“Приведем простой пример: У нас есть наш *Car* — машина. У каждой машины есть двигатель. Кроме того, у каждой машины есть пассажиры внутри.

В чем же принципиальная разница между полями *Engine engine* и *Passenger [] passengers*? Если у машины внутри сидит пассажир А, это не значит, что в ней не могут находиться пассажиры B и C. Одна машина может соответствовать нескольким пассажирам. Кроме того, если всех пассажиров высадить из машины, она продолжит спокойно функционировать.

Связь между классом *Car* и массивом пассажиров *Passenger [] passengers* менее строгая. Она называется **агрегацией**.

**Композиция** — более строгий тип связи. При использовании композиции объект не только является частью какого-то объекта, но и не может принадлежать другому объекту того же типа.

Самый простой пример — двигатель автомобиля. Двигатель является частью автомобиля, но не может быть частью другого автомобиля. Как видишь, их связь гораздо более строгая, чем у *Car* и *Passengers*.”

Чаще всего в моделях используются бинарные ассоциации, отражающие связи между объектами двух классов. В UML определены также **многополюсные ассоциации**, отражающие связи между бóльшим числом объектов. С формальной точки зрения многополюсные ассоциации излишни, поскольку их можно выразить через комбинацию бинарных ассоциаций введением дополнительных сущностей. Действительно, упорядоченную тройку объектов (a, b, c) ‒ элемент трехполюсной ассоциации ‒ можно представить как упорядоченную пару (a, d), где d ‒ новый объект, представляющий упорядоченную пару (b, c).

Диаграммы классов (принципы):

* Описывать структуру удобнее параллельно с описанием поведения.
* Каждая итерация должна быть небольшим уточнением
* Не обязательно включать в модель все классы сразу
* Не обязательно определять все составляющие класса сразу.
* Не обязательно определять все отношения между классами сразу