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.

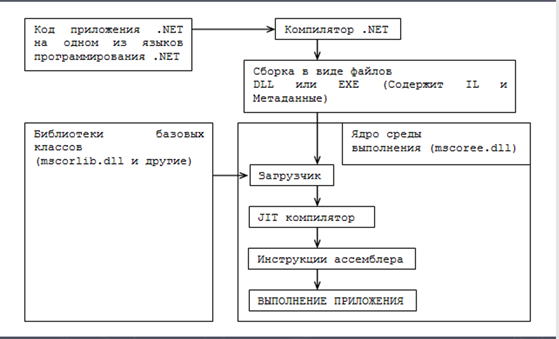
**.Net Framework** – программная платформа. Обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows. Ее компоненты:

* **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».

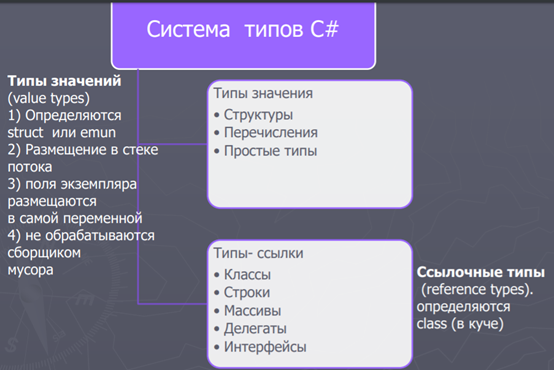
# ===4. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.

**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:**

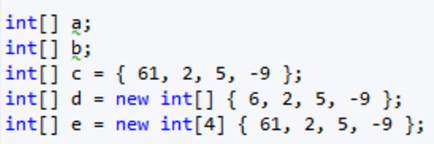
тип[] имя;

тип[] имя = new тип [ размерность ];

тип[] имя = { список инициализаторов };

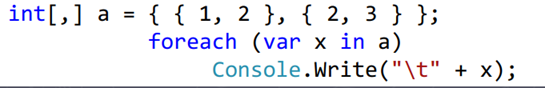
тип[] имя = new тип [] { список инициализаторов };

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

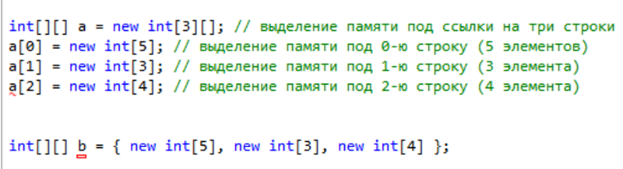


**Цикл foreach** предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

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.