# 1. Что такое .Net Framework и из чего он состоит?

# Платформа .NET Framework — это среда выполнения, которая управляет приложениями, предназначенными для .NET Framework. Она состоит из среды CLR, которая предоставляет инструменты управления памятью и другие службы системы, обширной библиотеки классов (FCL-Framework Class Library), позволяющей программистам использовать устойчивый, надежный код во всех основных областях разработки приложений, компилятор JIT(Just-in-Time) и Microsoft IL(intermediate language)

Платформа .NET Framework — это управляемая среда выполнения для ОС Windows, предоставляющая разнообразные службы выполняемым в ней приложениям. Она состоит из двух основных компонентов: среды CLR — механизма, управляющего выполняющимися приложениями, и NET Framework Class Library (.NET FCL) — библиотеки проверенного кода, предназначенного для повторного использования, который разработчики могут вызывать из своих приложений.

В приложениях .NET Framework функции выделения и освобождения памяти выполняет среда CLR.

В .NET Framework базовые типы определяются системой типов .NET Framework, при этом для всех языков .NET Framework используются одни и те же базовые типы. Разработчиками используется удобная библиотека типов, входящая в библиотеку классов .NET Framework.

Платформа .NET Framework включает библиотеки для конкретных областей разработки приложений, например ASP.NET для веб-приложений, Windows Presentation Foundation для классических приложений Windows.

Взаимодействие языков. Языковые компиляторы, ориентированные на .NET Framework, выдают промежуточный код, называемый языком CIL (Common Intermediate Language), который, в свою очередь, компилируется во время выполнения средой CLR. С помощью этой функции подпрограммы, написанные на одном языке, доступны в других языках, поэтому разработчики могут создавать приложения на предпочитаемых языках.

Обычно приложения, разработанные на основе конкретной версии платформы .NET Framework, могут выполняться без доработок и на более поздних версиях платформы.

Параллельное выполнение. Платформа .NET Framework помогает разрешать конфликты версий, поскольку на компьютере могут быть установлены несколько версий среды CLR. Это означает, что несколько версий приложений могут существовать совместно и приложение может выполняться на версии платформы .NET Framework, для которой оно было создано.

Настройка для различных версий. При настройке в соответствии со стандартом .NET разработчики создают библиотеки классов, которые работают на различных платформах .NET Framework, поддерживаемых соответствующей версией стандарта.

# 2. Поясните, что такое CLR-среда.

CLR (Common Language Runtime) - общеязыковая среда выполнения - это среда, подходящая для .NET совместимых языков программирования (С#, F#, Visual Basic, Managed C++). Основные возможности:

Управление памятью

Загрузка сборок

Безопасность

Обработка исключений

Синхронизация

Фактически во время выполнения программы в среде CLR неизвестно, на каком языке программирования разработчик написал исходный код. А это значит, что можно выбрать любой язык программирования, который позволяет проще всего решить конкретную задачу.

CLR это исполняющая среда для выполнения IL (промежуточного кода).

CLR создает объекты в памяти, вызывает методы, удаляет объекты из памяти.

CLR является одним из основных компонентов в .NET Framework.

Выполнение С# приложения исполнительной средой CLR

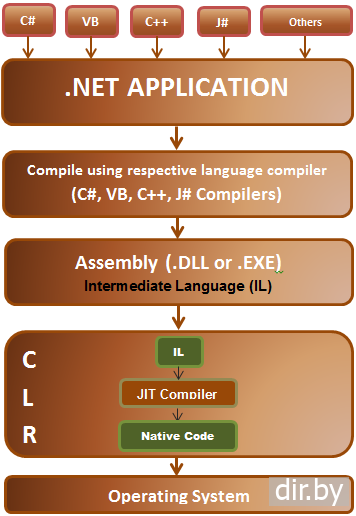
Когда мы запускам С# приложение то есть открываем exe файл, тогда начинает работать CLR. CLR исполняет каждую строчку IL кода в 2 шага. IL (промежуточный код) находится в сборке (exe файл)

Шаг 1. IL (промежуточный код) компилируется в команды процессора (0001 0011 1101… ). Этим занимается JIT компилятор в CLR среде.

Вместо компиляции всего приложения , JIT (Just-In-Time) компилятор просто компилирует каждую порцию кода при вызове. Если промежуточный код однажды скомпилирован, то результирующий машинный исполняемый код сохраняется до момента завершения работы приложения, поэтому его перекомпиляция при повторных обращениях к нему не требуется.

В Microsoft аргументируют, что такой процесс более эффективен, чем компиляция всего приложения при запуске, поскольку высока вероятность того, что крупные фрагменты кода приложения на самом деле не будут выполняться при каждом запуске.

Шаг 2. Выполняются команды процессора.



Чтобы CLR работала нужно:

Скомпилировать C# код в сборку(exe файл)-Запустить сборку (exe файл).

# 3. Что такое FCL?

На языке C# невозможно разработать даже простую программу без знания базовых классов библиотеки FCL. Для консольных приложений такими базовыми классами являются статические классы Console, Convert и Math из пространства имен System.

FCL (.NET Framework Class Library) – соответствующая CLS-спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы Microsoft .NET.

Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена служить основой при разработке .NET-приложений, компонент, элементов управления.

.NET библиотека классов является вторым компонентом CLR.

.NET FCL могут использовать все .NET-приложения, независимо от назначения архитектуры используемого при разработке языка программирования, и в частности:

встроенные (элементарные) типы, представленные в виде классов (на платформе .NET все построено на структурах или классах);

классы для разработки графического пользовательского интерфейса (Windows Forms);

классы для разработки web-приложений и web-служб на основе технологии ASP.NET (Web Forms);

классы для разработки XML и Internet-протоколов (FTP, HTTP, SMTP, SOAP);

классы для разработки приложений, работающих с базами данных (ADO .NET) и многое другое.

.NET-приложение – приложение, разработанное для выполнения на платформе Microsoft .NET. Реализуется на языках программирования, соответствующих CLS.

# 4. Какая наименьшая исполнимая единица в .NET?

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

# 5. Что такое IL?

Объектно-ориентированный машинный язык не зависящий от процессора

IL неизбежно должен воплощать некоторую определенную методологию программирования, а это означает, что исходный язык также должен быть совместим с этой методологией. Принцип, которым руководствовались в Microsoft при создании IL: классическое объектно-ориентированное программирование с реализацией одиночного наследования классов.

IL имеет способность языкового взаимодействия, под которой подразумевается возможность для классов, написанных на одном языке, напрямую обращаться к классам, написанным на другом языке, т.е.:

Как и любой язык программирования, IL предлагает множество предопределенных примитивных типов данных. Одна из особенностей IL, однако, заключается в том, что он делает четкое различие между значениями и ссылочными типами.

# 6. Пояснить работу JIT-компилятора?

Когда среда CLR выполняет IL код, то происходит следующее: при вызове блока IL-кода JIT-компилятор проверяет и преобразует его в машинные команды которые он сохраняет в динамическом блоке памяти. Затем, он возвращается к структуре данных типа и заменяет адрес вызываемого метода адресом этого блока памяти и при следующем запросе повторная компиляция не будет выполнена, а будет использован уже скомпилированный код.

# 7. Что такое CTS (Common Type System)?

Для начала напомним, что реализация .NET является независимой от языка. Это не просто означает, что программист может писать код на любом языке, который можно скомпилировать в промежуточный язык. Это также значит, что ему нужно взаимодействовать с кодом, написанным на других языках, которые поддерживаются в реализации .NET.

Чтобы сделать это без каких-либо затруднений, требуется способ описания всех поддерживаемых типов. Именно за это и отвечает система общих типов CTS. Она предназначена для выполнения следующих задач:

Создание платформы для выполнения на разных языках.

Предоставление объектно-ориентированной модели для поддержки реализации различных языков в реализации .NET.

Определение набора правил, которым должны следовать все языки при работе с типами.

Предоставление библиотеки, которая содержит базовые типы-примитивы, используемые при разработке приложений (например, Boolean, Byte, Char и т. д.).

Система CTS определяет две разновидности типов, которые должны поддерживаться: типы значений и ссылочные типы. Имена разновидностей указывают на их определения.

Объекты ссылочных типов представлены ссылкой на фактическое значение объекта. Здесь ссылка аналогична указателю в C/C++. Она просто указывает на адрес памяти, где находятся значения объектов. Это оказывает значительное влияние на способ использования этих типов. Если назначить ссылочный тип переменной, а затем передать эту переменную, например, в метод, любые изменения объекта будут отражены на основном объекте. Копирование при этом не выполняется.

В типах значений объекты, наоборот, представлены своими значениями. Назначение типа значения переменной, по сути, равнозначно копированию значения этого объекта.

Система CTS определяет несколько категорий типов, каждый из которых имеет собственную семантику и способ использования:

Классы

Структуры

Перечисления

Интерфейсы

Делегаты

Система CTS также определяет все другие свойства типов, такие как модификаторы доступа, которые являются допустимыми членами типа, порядок наследования и перегрузки и т. п.

# 8. Какие аспекты поведения определяет тип System.Object?

c аже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object.

# 9. Что находится в MSCorLib dll?

По умолчанию все проекты C# содержат ссылку на библиотеку mscorlib.dll, в которой находятся основные классы .NET Framework.

# 10. Что такое частные и общие сборки?

Частные находятся в каталоге программы, а общие в GAC.

# 11. Что такое assembly manifest?

Набор таблиц метаданных (файлы, которые входят в сборку, обедоступные экспортируемые типы, файлы ресурсов или данных).

# 12. Что такое GAC?

Global Assembly Cache (глобальный кэш сборок)

# 13. Чем managed code отличается от unmanaged code

Программный код который CLR

Кроме того, поскольку unmanaged компилируется в машинный код, а не в промежуточный язык, как managed

# 14. Как и для чего определен метод Main?

Точка входа в главную функцию.

# 15. Варианты использования директивы using( using Directive ) в C#.

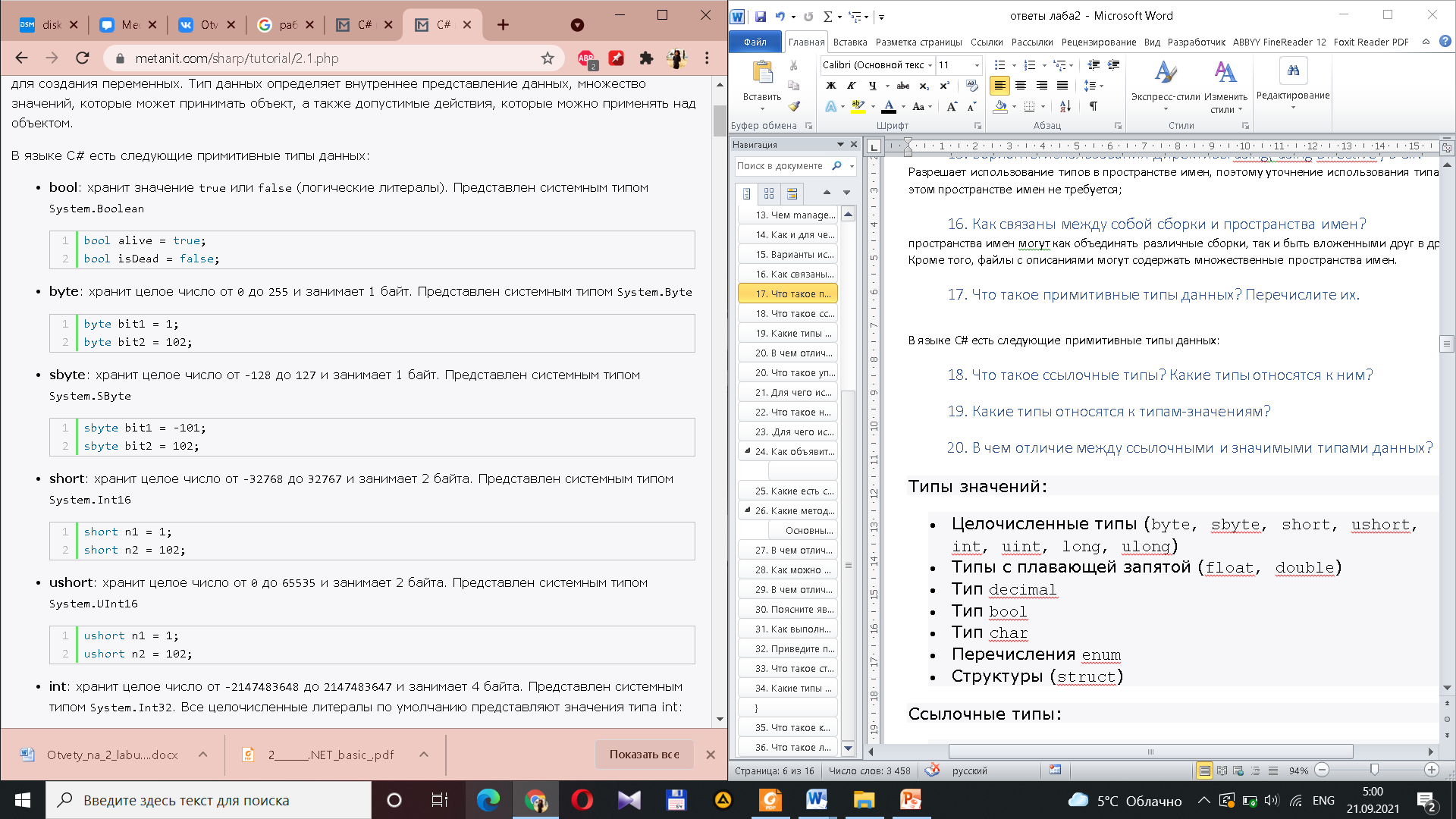
Разрешает использование типов в пространстве имен, поэтому уточнение использования типа в этом пространстве имен не требуется;

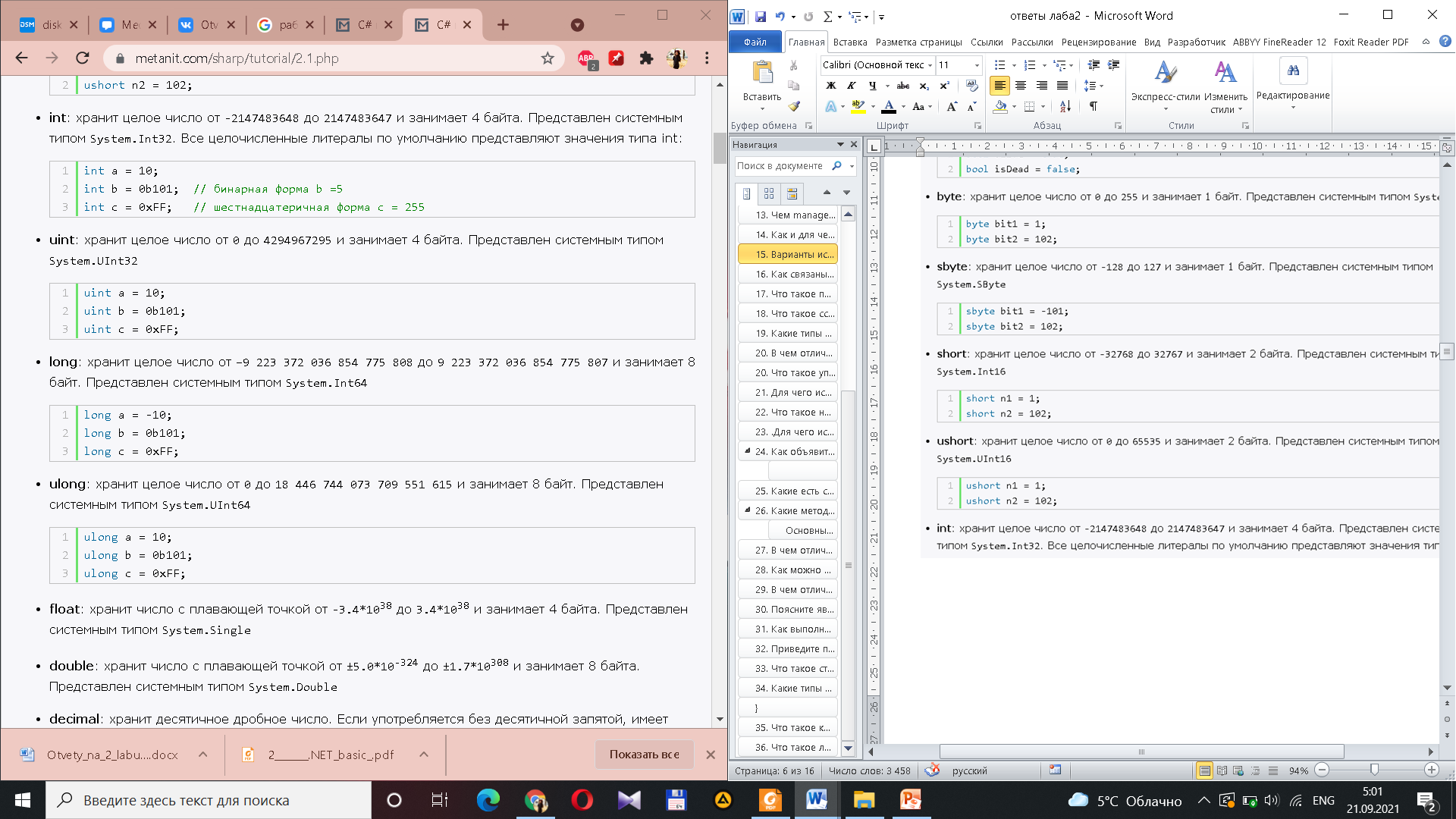
# 16. Как связаны между собой сборки и пространства имен?

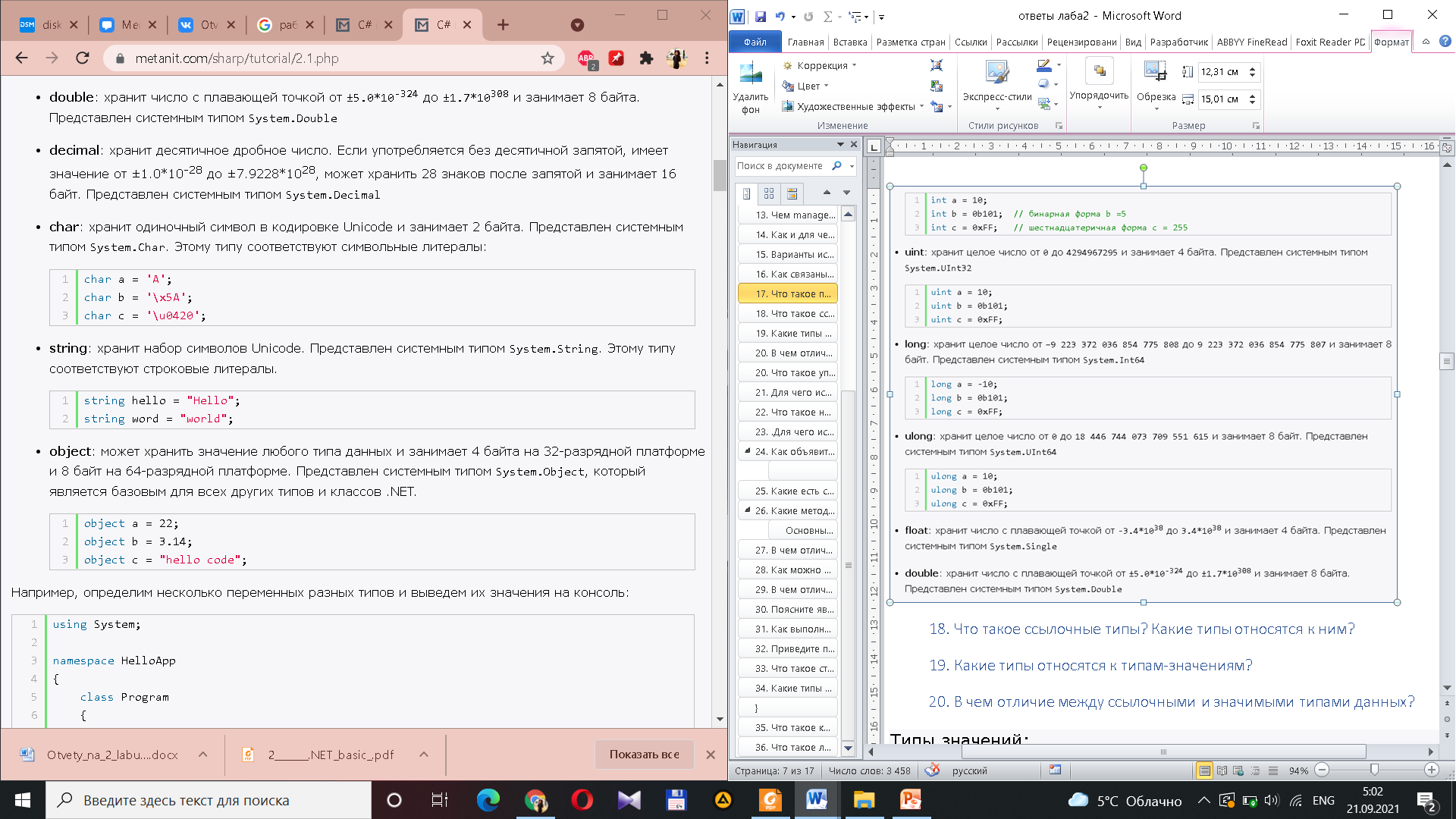
пространства имен могут как объединять различные сборки, так и быть вложенными друг в друга. Кроме того, файлы с описаниями могут содержать множественные пространства имен.

# 17. Что такое примитивные типы данных? Перечислите их.

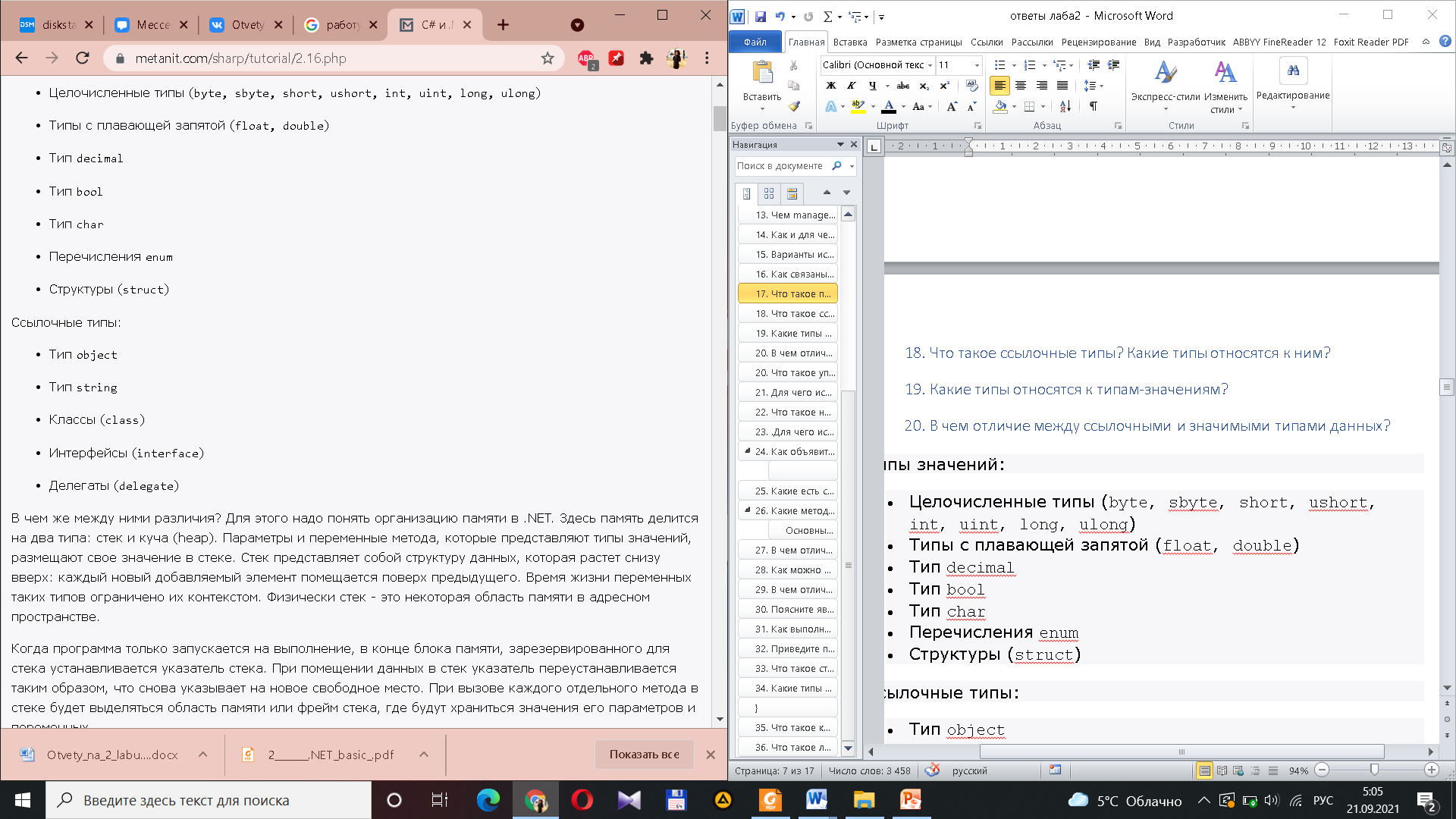
В языке C# есть следующие примитивные типы данных:







# 18. Что такое ссылочные типы? Какие типы относятся к ним?

Передается только ссылка; изменения в объекте отразятся на всех остальных ссылках на него. 

# 19. Какие типы относятся к типам-значениям?

К типам значений в .NET относятся структуры и перечисления. Простые типы, такие как int, float, decimal тоже являются типами значений, но разработчики приложений для .NET свободно могут определять собственные типы значений с помощью struct.

# 20. В чем отличие между ссылочными и значимыми типами данных?

Типы значений:

* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

Значимые типы хранят значение, а ссылочные - ссылку на значение.

Главное отличие - это где храниться значение переменной. Для ссылочных типов - это куча, для значимых - это стек. При этом, ссылочная переменная (сама ссылка) храниться на стеке. Значимыми могут быть только экземпляры структур, в противоположность им ссылочные - это экземпляры классов. Все встроенные типы (int, double и т.д.) - это структуры

Для значимых типов при копировании вы получаете новый экземпляр, содержащий копии значений исходного экземпляра. Для ссылочных типов, копирование даёт вам новую ссылку на те же данные.

# Что такое упаковка и распаковка значимых типов?

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

**распаковка (unboxing)**

получение указателя на исходный значимый тип (поля

данных), содержащийся в объекте.(из ссылочного в значений)

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

# Для чего используется тип dynamic?

Используется для коммуникации с другими компонентами.Если есть вероятность изменения значения переменной в ходе программы.

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

Ключевое слово **dynamic** позволяет **использовать** переменные, которые не зависят от **типа**, а действуют как **тип** данных, которые они содержат. Динамической переменной можно легко присвоить новое значение, и если **тип** присваиваемого значения отличается от текущего, динамическая переменная просто автоматически изменит **тип**.

# Что такое неявно типизированная переменная?

**Неявная** **типизация** — **типизация**, при которой **переменная** остается строго **типизированной**, а её тип определяется компилятором, исходя из присваиваемого значения. Объявляется ключевым словом **var**

Ограничения:

\*должно включать инициализатор (null нельзя)

int i1, i2 = 4; //допускается

var v1, v2 = 2; //вызовет ошибки CS0810 и CS081

\*ключевое слово 'var' может применяться только в объявлении локальной переменной

\*неявно типизированные локальные переменные не допускают множественного объявления

# .Для чего используют Nullable тип?

Тип Nullable используют если переменная может принимать значения null или тип указанный первоначально

Значение null по умолчанию могут принимать только объекты ссылочных типов. 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.

int? z = null;

bool? enabled = null;

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

# Как объявить строковый литерал? Какие операции можно выполнять со строкой?

Строковые литералы представляют строки. Строки заключаются в двойные кавычки:

|  |  |
| --- | --- |
| 1  2  3 | Console.WriteLine("hello");  Console.WriteLine("фыва");  Console.WriteLine("hello word"); |

* присваивание (=);
* проверка на равенство содержимого (==);
* проверка на неравенство (!=);
* обращение по индексу ([]);
* сцепление (конкатенация) строк (+)
* <,>, >=,<= - сравнивают ссылки!!!!!!!!
* Поиск в строке
* разделение строк
* обрезка
* вставка
* удаление
* замена
* смена регистра
* 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.

# Какие есть способы для задания и инициализации строк?

// Declare without initializing.

string message1;

// Initialize to null.

string message2 = null;

// Initialize as an empty string.

// Use the Empty constant instead of the literal "".

string message3 = System.String.Empty;

// Initialize with a regular string literal.

string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Use System.String if you prefer.

System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)

// you can use implicit typing.

var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from

// being used to store another string value.

const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating

// a string from a char\*, char[], or sbyte\*. See

// System.String documentation for details.

char[] letters = { 'A', 'B', 'C' };

string alphabet = new string(letters);

# Какие методы есть у типа String?

### Основные методы строк

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:(Звездочкой помечены упомянутые в его презентации)

* **\*Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **\*CompareOrdinal**: сравнивает две строки без учета локали
* **СompareTo:** Сравнение текущего экземпляра строки с другой строкой
* **Contains**: определяет, содержится ли подстрока в строке
* **\*Concat**: соединяет строки
* **\*Сopy**: создание копии строки
* **CopyTo**: копирует часть строки, начиная с определенного индекса в массив
* **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
* **\*Length**: Длина строки (количество символов)
* **\*Remove**: Удаление подстроки из заданной позиции
* **\*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

# В чем отличие пустой и null строки?

**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.

# Как можно выполнить сравнение строк?

К строкам можно применять операции сравнения. Оператор **==** возвращает true, если все символы обеих строк равны.при этом будут сравниваться значения строковых переменных, а не их ссылки, как это делается для других ссылочных типов.

Или можно использовать методы строк для сравнения включающие в себя Compare.

Остальные базовые операции сравнения **<**, **<=**, **>**, **>=** сравнивают строки в зависимости от регистра и алфавитного порядка символов. Например, строка "b" условно больше строки "a", так как символ b по алфавиту идет после символа a. А строка "a" больше строки "A". Если первые символы строки равны, то сравниваются последующие символы.

# В чем отличие типов String и StringBuilder?

Прежде всего, объект String представляет собой неизменяемую строку. Когда мы выполняем какой-нибудь метод класса String, система создает новый объект в памяти с выделением ему достаточного места. Удаление первого символа - не самая затратная операция. Однако когда подобных операций множество, а объем текста, для которого надо выполнить данные операции, также не самый маленький, то издержки при потере производительности становятся более существенными.

Чтобы выйти из этой ситуации во фреймворк .NET был добавлен новый класс **StringBuilder**, который находится в пространстве имен System.Text. Этот класс представляет динамическую строку.

При создании строки StringBuilder выделяет памяти больше, чем необходимо этой строке.

# Поясните явные преобразования переменных с помощью команд Convert.

Класс **Convert** представляет еще один способ для преобразования значений. Для этого в нем определены следующие статические методы:

* **ToBoolean(value)**
* **ToByte(value)**
* **ToChar(value)**
* **ToDateTime(value)**
* **ToDecimal(value)**
* **ToDouble(value)**
* **ToInt16(value)**
* **ToInt32(value)**
* **ToInt64(value)**
* **ToSByte(value)**
* **ToSingle(value)**
* **ToUInt16(value)**
* **ToUInt32(value)**
* **ToUInt64(value)**

В качестве параметра в эти методы может передаваться значение различных примитивных типов, необязательно строки.

Однако опять же, как и в случае с методом Parse, если методу не удастся преобразовать значение к нужному типу, то он выбрасывает исключение FormatException.

# Как выполнить консольный ввод/вывод?

Для ввода/вывода используется статический класс System.Console.

* **Clear**: очистка консоли
* **WriteLine**: вывод строки текста с переводом на новую строку
* **Write**: вывод строки текста
* **Read**: считывание введенного символа в виде числового

кода данного символа. С помощью преобразования к типу

char мы можем получить введенный символ

* **ReadKey**: считывание нажатой клавиши клавиатуры (ConsoleKeyInfo key= Console.ReadKey();)
* **ReadLine**: считывание строки текста со входного потока

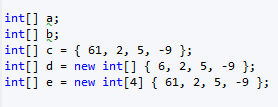
# Приведите примеры определения и инициализации одномерных и двумерных массивов.

**Одномерные массивы**

тип[] имя;

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

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

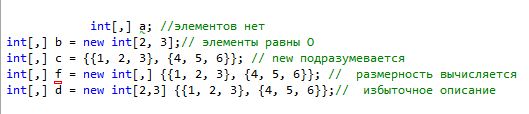
тип[] имя = new тип [ размерность ] { список инициализаторов };

**Многомерные(прямоугольные массивы)**

тип[,] имя;

тип[,] имя = new тип [ разм\_1, разм\_2 ]; тип[,] имя = { список инициализаторов };

тип[,] имя = new тип [,] { список инициализаторов }; тип[,] имя = new тип [ разм\_1, разм\_2 ] { список инициализаторов };



int[,] a = { { 1, 2 }, { 2, 3 } };

foreach (var x in a) Console.Write("\t" + x);

# Что такое ступенчатый массив? Как его задать?

От многомерных массивов надо отличать массив массивов или так называемый "зубчатый массив":

|  |  |
| --- | --- |
| 1  2  3  4 | int[][] nums = new int[3][];  nums[0] = new int[2] { 1, 2 };          // выделяем память для первого подмассива  nums[1] = new int[3] { 1, 2, 3 };       // выделяем память для второго подмассива  nums[2] = new int[5] { 1, 2, 3, 4, 5 }; // выделяем память для третьего подмассива |

Здесь две группы квадратных скобок указывают, что это массив массивов, то есть такой массив, который в свою очередь содержит в себе другие массивы. Причем длина массива указывается только в первых квадратных скобках, все последующие квадратные скобки должны быть пусты: new int[3][]. В данном случае у нас массив nums содержит три массива. Причем размерность каждого из этих массивов может не совпадать.

Ступенчатый массив представляет собой массив массивов, в котором длина каждого массива может быть разной. Следовательно, ступенчатый массив может быть использован для составления таблицы из строк разной длины.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

*тип [][] имя массива = new тип[размер] [];*

где размер обозначает число строк в массиве. Память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной.

# Какие типы можно использовать в foreach? Приведите пример.

Цикл foreach предназначен для перебора элементов в контейнерах, в том числе в массивах.Может принимать любые типы,которые принимает массив.Цикл работает только на чтение, но не на запись элементов (наполнять нельзя)

foreach (тип\_иден. название\_иден. in контейнер)

{ операторы }

int[] numbers = new int[] { 1, 2, 3, 4, 5 }; foreach (int i in numbers)

{

Console.WriteLine(i);

# }

# Что такое кортеж? Для чего и как он используется?

Кортеж представляет набор значений, заключенных в круглые скобки:

Основное предназначение **кортежей** - обобщение нескольких элементов в структуру с упрощенным синтаксисом.

***Кортежи (tuple)*** комбинируют объекты различных типов (от одного до восьми).

Кортежи могут передаваться в качестве параметров в метод, могут быть возвращаемым результатом функции, либо использоваться иным образом.

|  |  |
| --- | --- |
| 1 | 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]. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main(string[] args)  {      var tuple = (5, 10);      Console.WriteLine(tuple.Item1); // 5 |

В данном случае тип определяется неявно. Но мы ткже можем явным образом указать для переменной кортежа тип:

(string, int, double) person = ("Tom", 25, 81.23);

Мы также можем дать названия полям кортежа:

|  |  |
| --- | --- |
| 1  2  3 | var tuple = (count:5, sum:10);  Console.WriteLine(tuple.count); // 5  Console.WriteLine(tuple.sum); // 10 |

Теперь чтобы обратиться к полям кортежа используются их имена, а не названия Item1 и Item2.

Мы даже можем не использовать переменную для определения всего кортежа, а использовать отдельные переменные для его полей

|  |
| --- |
| var (name, age) = ("Tom", 23);      Console.WriteLine(name);    // Tom      Console.WriteLine(age); //23 |

Свойства:

* создается один раз и остается неименным (все свойства доступны только для чтения)
* позволяют использовать методы CompareTo, Equals, GetHashCode и ToString, свойство Size
* реализуют интерфейсы IStructuralEquatable, IStructuralComparable и IComparable (можно сравнивать)

# Что такое локальная функция?

Локальные функции

вспомогательная функция - внутри метода, в котором вызывается

public int Method(int x)

{

return LocFun(x).current;

int LocFun(int i)

{

if (i == 0) return 0; var p = LocFun(i - 1); return p+1;

}

}

Аргументы внешнего метода и его локальные переменные доступны для локальной функции