**Задание 1.**

Теория:

**Использование аргументов командной строки.**

Аргументы, предоставляемые исполняемому файлу в командной строке, доступны через [инструкции верхнего уровня](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/program-structure/top-level-statements) или через необязательный параметр для метода Main. Аргументы предоставляются в форме массива строк ( при передаче текста, его нужно заключить в двойные кавычки). Каждый элемент массива содержит один аргумент. Пробелы между аргументами удаляются.

При запуске приложения ей передаются параметры, они же "аргументы". Таким образом, программе передаются разные входные данные. Для работы с аргументами, переданными на запуск программы в методе Main есть параметр string[] args. Обычно определяют, существуют ли аргументы, проверяя Length свойство. Массив args не может быть нулевым. Таким образом, можно безопасно получить доступ к Length свойству без проверки нуля. Если длина массива аргументов равна 0, значит, их не передали.

При разработке графического приложения с использованием Windows Forms или WPF, для работы с аргументами командной строки лучше воспользоваться функцией GetCommandLineArgs() из System.Environment, она возвращает список аргументов в виде массива строк.

Использование:

- запускать нужную программу, передавая ей путь к открываемому файлу как аргумент командной строки;

- запуск приложения из другого приложения, передавая ему любую команду;

- работа с файлами;

- использование флагов (вызов справки, чтение/запись в файл и т.д.);

- преобразовать строковые аргументы в числовые типы.

**Различные способы компиляции проекта.**

Компиляция проекта может быть с использованием конфигурации Debug - поддерживает отладку приложения или Release - создает версию приложения, которое можно развернуть (отладка не поддерживается).

**Практика:** считать из строки все числа, вывести отсортированный результат в текстовый файл.

**Задание 2.**

Теория:

**Использование массивов и односвязных списков. Плюсы и минусы.**

Массив - это совокупность однотипных данных, расположенных непрерывно в памяти. Доступ к элементу осуществляется по индексу за O(1) - мы обращаемся непосредственно к нужному участку памяти. Нужно заранее определять размер массива.

Как и массивы, связный список – это группа значений. Однако, в отличие от массивов, значения в связанном списке не обязательно должны быть одного типа, и нам не нужно заранее определять размер списка.

Доступ к элементу в связном списке – определяется указателями на следующий и (или) на предыдущий элемент, в односвязном списке каждый узел хранит указатель только на следующий узел.

Связные списки обладают одним очень важным преимуществом: для них операции вставки и удаления принадлежат к классу O(1). Независимо от текущего элемента для вставки или удаления элемента всегда требуется одно и то же время.

Основным недостатком связных списков является то, что получение доступа к их элементам принадлежит к классу О(n). В этом случае важно количество элементов в списке: при поиске n-ного элемента мы начинаем с некоторой позиции в списке и переходим по ссылкам вплоть до искомого элемента. Чем больше элементов в списке, тем больше переходов придется совершить.

Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями, размер ограничен только объёмом памяти [компьютера](https://ru.wikipedia.org/wiki/%D0%AD%D0%92%D0%9C) и разрядностью указателей.

Недостатки связных списков вытекают из их главного свойства — последовательного доступа к данным:

- сложность прямого доступа к элементу, а именно определения физического адреса по его [индексу](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81_(%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2)) (порядковому номеру) в списке;

- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в [массивах](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), например, указатели не нужны);

- некоторые операции со списками медленнее, чем с массивами, так как к произвольному элементу списка можно обратиться, только пройдя все предшествующие ему элементы;

- соседние элементы списка могут быть распределены в памяти нелокально, что снизит эффективность [кэширования](https://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88) данных в процессоре;

- над связными списками, по сравнению с массивами, гораздо труднее (хоть и возможно) производить параллельные векторные операции, такие, как вычисление суммы: накладные расходы на перебор элементов снижают эффективность распараллеливания.

Массив эффективней использовать с данными одного типа и при условии, что размер массива известен изначально. Списки более динамичны, работают с разными типами и могут менять свой размер, удобней добавлять/ удалять элементы, но расходуют больше памяти. Выбор зависит от поставленной задачи.

**Практика:** из пользовательского ввода создать массив/список записей в учебном журнале вида "номер\_п/п;ФИО ученика;Оценка", результат записать в текстовый файл.

**Задание 3.**

Теория:

**Классы и структуры в C#. Разница, плюсы и минусы использования.**

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными.

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией.

Перед каждым объявлением переменной и метода указывается *доступ*. Это спецификатор доступа, например public, определяющий порядок доступа к данному члену класса. Члены класса могут быть как закрытыми (*private*) в пределах класса, так открытыми (*public*), т.е. более доступными. Спецификатор доступа определяет тип разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса.

[Структуры](https://c-sharp.pro/?p=59) синтаксически очень похожи на [классы](https://c-sharp.pro/?p=657), но существует принципиальное отличие, которое заключается в том, что класс – является ссылочным типом (reference type), а структуры – значимым типом (value type). Следовательно, классы всегда создаются в так называемой “куче” (heap), а структуры создаются в стеке (stack).

Главное отличие структур и классов: структуры, указываемые в списке параметров метода, передаются по значению (то есть копируются), объекты классов — по ссылке (структуру тоже можно передать по ссылке, используя модификаторы out и ref). Именно это является главным различием в их поведении, а не то, где они хранятся.

Чем больше использовать структуры вместо маленьких классов, тем менее затратным по ресурсам будет использование памяти.

Так же как и классы, структуры могут иметь поля, методы и конструкторы.

В отличие от классов, использование публичных полей в структурах в большинстве случаев *не рекомендуется*, потому что не существует способа контролирования значений в них. Структуры не поддерживают наследование.

Два правила структур:

*Первое правило структуры*: Всегда все переменные должны быть инициализированы.  
В классах можно инициализировать значение полей непосредственно при их объявлении. В структурах такого сделать нельзя, и поэтому данный код вызовет ошибку при компиляции. Поэтому:

*Второе правило структуры*: Нельзя инициализировать переменные в том месте, где они объявляются.

***Сравнение классов и структур в сжатом виде:***

|  |  |  |
| --- | --- | --- |
| **Вопрос** | **Структура** | **Класс** |
| Какого же типа экземпляр объекта? | Значимый (value) тип | Ссылочный (reference) тип |
| Где “живут” экземпляры этих объектов? | Экземпляры структуры называют значениями и “живут” они в стеке (stack). | Экземпляры классов называют объектами и “живут” они в куче (heap). |
| Можно ли создать конструктор по умолчанию? | Нет | Да |
| Если создается свой конструктор, будет ли компилятор генерировать конструктор по умолчанию? | Да | Нет |
| Если в своём конструкторе не будут инициализированы некоторые поля, будут ли они автоматически инициализированы компилятором? | Нет | Да |
| Разрешается ли инициализировать переменные там, где их объявляют? | Нет | Да |

Поля структуры могут быть инициализированы при использовании конструктора (если объект объявляется с помощью оператора new), причем не важно, какого «собственного» или «по умолчанию».

К особенностям структур можно отнести еще и тот факт, что вследствие того, что структуры являются значимым типом, то можно создать структуру без использования конструктора.

**Практика:** создать из пользовательского ввода сущность (класс или структуру) для хранения контактной

информации. Пользователь в произвольном порядке вводит телефон, имя и адрес электронной почты,

программа при помощи регулярных выражений добавляет эти данные в сущность и выводит на экран.

**Задание 4.**

Теория:

**Методы. Что такое сигнатура метода? Использование ref и out.**

Метод — это блок кода, содержащий ряд инструкций. Программа инициирует выполнение инструкций, вызывая метод и указывая все аргументы, необходимые для этого метода. В C# все инструкции выполняются в контексте метода. По сути метод - это именованный блок кода, который выполняет некоторые действия.

## Сигнатуры методов:

Методы объявляются в [классе](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class), [структуре](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct) или [интерфейсе](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/interfaces) путем указания уровня доступа, такого как public или private, необязательных модификаторов, таких как abstract или sealed, возвращаемого значения, имени метода и затем список аргументов в круглых скобках и далее - тело метода в фигурных скобках:

***[модификаторы] тип\_возврата ИмяМетода([параметры])***

***{***

***// Тело метода***

***}***

Все эти части вместе представляют собой сигнатуру метода.

Тип возврата метода не является частью сигнатуры метода в целях перегрузки метода. Однако он является частью сигнатуры метода при определении совместимости между делегатом и методом, который он указывает.

Каждый параметр состоит из имени типа параметра и имени, по которому к нему можно обратиться в теле метода. Вдобавок, если метод возвращает значение, то для указания точки выхода должен использоваться оператор возврата return вместе с возвращаемым значением.

Если метод не возвращает ничего, то в качестве типа возврата указывается void, поскольку вообще опустить тип возврата невозможно. Если же он не принимает аргументов, то все равно после имени метода должны присутствовать пустые круглые скобки. При этом включать в тело метода оператор возврата не обязательно — метод возвращает управление автоматически по достижении закрывающей фигурной скобки.

**Использование ref и out.**

При использовании в списке параметров метода ключевое слово ref указывает на то, что аргумент передается по ссылке, а не по значению.  Ключевое слово ref позволяет превратить этот формальный параметр в псевдоним для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом.

Для использования параметра ref и при определении метода, и при вызове метода следует явно использовать ключевое слово ref.

Аргумент, передаваемый параметру ref или in , должен быть инициализирован перед его передачей.

Ключевое out инициирует передачу аргументов по ссылке. В результате этот формальный параметр становится псевдонимом для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом. Оно схоже с ключевым словом [ref](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/ref) за исключением того, что при использовании ref перед передачей переменную необходимо инициализировать. Оно также похоже на ключевое слово [in](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/in-parameter-modifier) за исключением того, что in не позволяет вызываемому методу изменять значение аргумента.

Переменные, передаваемые в качестве аргументов out, не требуется инициализировать перед передачей в вызове метода. Но перед передачей управления из вызванного метода он должен присвоить значение.

Ключевые слова in, ref и out не считаются частью сигнатуры метода для разрешения перегрузки. Таким образом, методы не могут быть перегружены, если единственное различие состоит в том, что один метод принимает аргумент ref или in, а другой — out.

Перегрузка допустима, если один метод принимает аргумент ref, in или out, а другой не использует ни один из этих модификаторов.

**Практика:** из пользовательского ввода создать две строки - в одной все числа, в другой символы, не являющиеся числами. Результат сохранить в файл.

**Задание 5.**

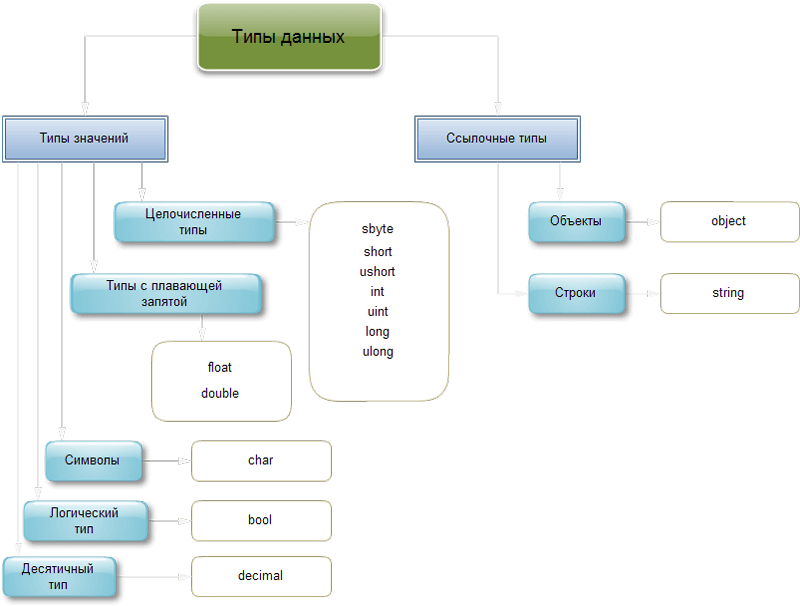
Теория:

**Типы данных и переменные. Рассказать о CTS (Common Type System).**

Типы данных имеют особенное значение в C#, поскольку это строго типизированный язык. Это означает, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, строгий контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу. Такого понятия, как "бестиповая" переменная, в данном языке программирования вообще не существует. Более того, тип значения определяет те операции, которые разрешается выполнять над ним. Операция, разрешенная для одного типа данных, может оказаться недопустимой для другого.

В C# имеются две общие категории встроенных типов данных: **типы значений** и **ссылочные типы**. Они отличаются по содержимому переменной. Концептуально разница между ними состоит в том, что тип значения (value type) хранит данные непосредственно, в то время как ссылочный тип (reference type) хранит ссылку на значение.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как [стек](https://professorweb.ru/my/csharp/charp_theory/level12/12_6.php), а ссылочные типы — в области, называемой [управляемой кучей](https://professorweb.ru/my/glossariy/glossy/files/101.php#y1).



## Целочисленные типы

## В C# определены девять целочисленных типов: *char, byte, sbyte, short, ushort, int, uint, long и ulong*. Но тип char применяется, главным образом, для представления символов и поэтому рассматривается отдельно. Остальные восемь целочисленных типов предназначены для числовых расчетов. Ниже представлены их диапазон представления чисел и разрядность в битах:

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Тип CTS** | **Разрядность в битах** | **Диапазон** |
| byte | System.Byte | 8 | 0:255 |
| sbyte | System.SByte | 8 | -128:127 |
| short | System.Int16 | 16 | -32768 : 32767 |
| ushort | System.UInt16 | 16 | 0 : 65535 |
| int | System.Int32 | 32 | -2147483648 : 2147483647 |
| uint | System.UInt32 | 32 | 0 : 4294967295 |
| long | System.Int64 | 64 | -9223372036854775808 : 9223372036854775807 |
| ulong | System.UInt64 | 64 | 0 : 18446744073709551615 |

## Типы с плавающей точкой

Типы с плавающей точкой позволяют представлять числа с дробной частью. В C# имеются две разновидности типов данных с плавающей точкой: **float** и **double**. Они представляют числовые значения с одинарной и двойной точностью соответственно. Так, разрядность типа float составляет 32 бита, что приближенно соответствует диапазону представления чисел от 5E-45 до 3,4E+38. А разрядность типа double составляет 64 бита, что приближенно соответствует диапазону представления чисел от 5E-324 до 1,7Е+308.

Тип данных float предназначен для меньших значений с плавающей точкой, для которых требуется меньшая точность. Тип данных double больше, чем float, и предлагает более высокую степень точности (15 разрядов).

Если нецелочисленное значение жестко кодируется в исходном тексте, то обычно компилятор предполагает, что подразумевается значение типа double. Если значение необходимо специфицировать как float, потребуется добавить к нему символ F (или f).

## Десятичный тип данных

Для представления чисел с плавающей точкой высокой точности предусмотрен также десятичный тип **decimal**, который предназначен для применения в финансовых расчетах. Этот тип имеет разрядность 128 бит для представления числовых значений в пределах от 1Е-28 до 7,9Е+28. Вам, вероятно, известно, что для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа decimal, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами.

## Символы

В C# символы представлены не 8-разрядным кодом, как во многих других языках программирования, например [С++](https://professorweb.ru/my/glossariy/glossy/files/101.php#ja4), а 16-разрядным кодом, который называется юникодом (Unicode). В юникоде набор символов представлен настолько широко, что он охватывает символы практически из всех естественных языков на свете. Если для многих естественных языков, в том числе английского, французского и немецкого, характерны относительно небольшие алфавиты, то в ряде других языков, например китайском, употребляются довольно обширные наборы символов, которые нельзя представить 8-разрядным кодом. Для преодоления этого ограничения в C# определен **тип char**, представляющий 16-разрядные значения без знака в пределах от 0 до 65 535. При этом стандартный набор символов в 8-разрядном коде ASCII является подмножеством юникода в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему остаются действительными в C#.

Для того чтобы присвоить значение символьной переменной, достаточно заключить это значение (т.е. символ) в одинарные кавычки.

Несмотря на то что тип char определен в C# как целочисленный, его не следует путать со всеми остальными целочисленными типами. Дело в том, что в C# отсутствует автоматическое преобразование символьных значений в целочисленные и обратно.

## Логический тип данных

**Тип bool** представляет два логических значения: "истина" и "ложь". Эти логические значения обозначаются в C# зарезервированными словами true и false соответственно. Следовательно, переменная или выражение типа bool будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение true, а 0 — в значение false.

**Ссылочные типы**

**Тип string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

**Тип object:** может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

**Использование суффиксов**

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы (дробные числа) рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul.

**Неявная типизация**

Для неявной типизации вместо названия типа данных используется ключевое слово **var**. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string.

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. Так как значение null, то компилятор не сможет вывести тип данных.

**Системные типы**

Все базовые типы имеют системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа.

| **Ключевое слово типа C#** | **Тип .NET** |
| --- | --- |
| [bool](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/bool) | [System.Boolean](https://learn.microsoft.com/ru-ru/dotnet/api/system.boolean) |
| [byte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.Byte](https://learn.microsoft.com/ru-ru/dotnet/api/system.byte) |
| [sbyte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.SByte](https://learn.microsoft.com/ru-ru/dotnet/api/system.sbyte) |
| [char](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/char) | [System.Char](https://learn.microsoft.com/ru-ru/dotnet/api/system.char) |
| [decimal](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | [System.Decimal](https://learn.microsoft.com/ru-ru/dotnet/api/system.decimal) |
| [double](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | [System.Double](https://learn.microsoft.com/ru-ru/dotnet/api/system.double) |
| [float](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | [System.Single](https://learn.microsoft.com/ru-ru/dotnet/api/system.single) |
| [int](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.Int32](https://learn.microsoft.com/ru-ru/dotnet/api/system.int32) |
| [uint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.UInt32](https://learn.microsoft.com/ru-ru/dotnet/api/system.uint32) |
| [nint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.IntPtr](https://learn.microsoft.com/ru-ru/dotnet/api/system.intptr) |
| [nuint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.UIntPtr](https://learn.microsoft.com/ru-ru/dotnet/api/system.uintptr) |
| [long](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.Int64](https://learn.microsoft.com/ru-ru/dotnet/api/system.int64) |
| [ulong](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.UInt64](https://learn.microsoft.com/ru-ru/dotnet/api/system.uint64) |
| [short](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.Int16](https://learn.microsoft.com/ru-ru/dotnet/api/system.int16) |
| [ushort](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | [System.UInt16](https://learn.microsoft.com/ru-ru/dotnet/api/system.uint16) |
| [object](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types#the-object-type) | [System.Object](https://learn.microsoft.com/ru-ru/dotnet/api/system.object) |
| [string](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types#the-string-type) | [System.String](https://learn.microsoft.com/ru-ru/dotnet/api/system.string) |
| [dynamic](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types#the-dynamic-type) | [System.Object](https://learn.microsoft.com/ru-ru/dotnet/api/system.object) |

Ключевое слово **[void](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/void)** представляет отсутствие типа. Оно используется в качестве возвращаемого типа метода, который не возвращает значение.

**Переменные**

Для хранения данных в программе применяются переменные. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом: тип имя\_переменной;

Вначале идет тип переменной, потом ее имя. В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

- имя может содержать любые цифры, буквы и символ подчеркивания, при этом первый символ в имени должен быть буквой или символом подчеркивания;

- в имени не должно быть знаков пунктуации и пробелов;

- имя не может быть ключевым словом языка C#. Таких слов не так много, и при работе в Visual Studio среда разработки подсвечивает ключевые слова синим цветом.

Хотя имя переменой может быть любым, но следует давать переменным описательные имена, которые будут говорить об их предназначении.

После определения переменной можно присвоить некоторое значение. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

Задать значение переменной можно, в частности, с помощью оператора присваивания. Кроме того, задать начальное значение переменной можно при ее объявлении. Для этого после имени переменной указывается знак равенства (=) и присваиваемое значение. Если две или более переменные одного и того же типа объявляются списком, разделяемым запятыми, то этим переменным можно задать, например, начальное значение.

В C# используются два метода для обеспечения инициализации переменных перед пользованием:

- Переменные, являющиеся полями класса или структуры, если не инициализированы явно, по умолчанию обнуляются в момент создания.

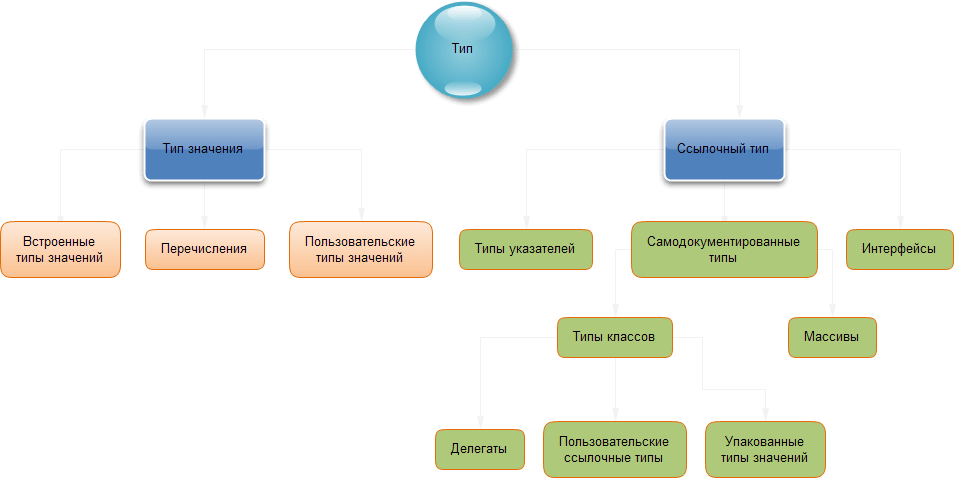
- Переменные, *локальные* по отношению к методу, должны быть явно инициализированы в коде до появления любого оператора, в котором используются их значения. В данном случае при объявлении переменной ее инициализация не происходит автоматически, но компилятор проверит все возможные пути потока управления в методе и сообщит об ошибке, если обнаружит любую возможность использования значения этой локальной переменной до ее инициализации.

В C# допускается также **динамическая инициализация** переменных с помощью любого выражения, действительного на момент объявления переменной

В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение. Отличительной чертой переменных является то, что в программе можно многократно менять их значение.

**Common Type System**

**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](https://professorweb.ru/my/csharp/charp_theory/level1/1_4.php). Внутренние детали CTS обычно интересуют только тех, кто занимается разработкой инструментов и/или компиляторов для платформы .NET. Т.е. CTS описывает не просто примитивные типы данных, а целую развитую иерархию типов, включающую хорошо определенные точки, в которых код может определять свои собственные типы. Иерархическая структура общей системы типов (CTS) отражает объектно-ориентированную методологию одиночного наследования IL и показана на следующей схеме:



Система общих типов CTS определяет способ объявления, использования и управления типами в среде CLR, а также является важной составной частью поддержки межъязыковой интеграции в среде выполнения. Система общих типов выполняет следующие функции.

* Формирует инфраструктуру, которая позволяет обеспечивать межъязыковую интеграцию, безопасность типов и высокопроизводительное выполнение кода.
* Предоставляет объектно-ориентированную модель, поддерживающую полную реализацию многих языков программирования.
* Определяет правила, которых необходимо придерживаться в языке. Эти правила помогают обеспечить взаимодействие объектов, написанных на разных языках.
* Предоставляет библиотеку, которая содержит типы-примитивы (например, [Boolean](https://learn.microsoft.com/ru-ru/dotnet/api/system.boolean), [Byte](https://learn.microsoft.com/ru-ru/dotnet/api/system.byte), [Char](https://learn.microsoft.com/ru-ru/dotnet/api/system.char), [Int32](https://learn.microsoft.com/ru-ru/dotnet/api/system.int32) и [UInt64](https://learn.microsoft.com/ru-ru/dotnet/api/system.uint64)), используемые в разработке приложений.

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

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.

**Практика:** Написать программу выводящую системное название типа long и местоположение папок "Документы" и "Cookies" в текстовый файл.

**Задание 6.**

Теория:

**Механизм обработки исключений. Цели использования, синтаксис.**

Исключениями, или исключительными ситуациями, обычно называются аномалии, которые могут возникать во время выполнения и которые трудно, а порой и вообще невозможно, предусмотреть во время программирования приложения. К числу таких возможных исключений относятся попытки подключения к базе данных, которой больше не существует, попытки открытия поврежденного файла или попытки установки связи с машиной, которая в текущий момент находится в автономном режиме. В каждом из этих случаев программист (и конечный пользователь) мало что может сделать с подобными "исключительными" обстоятельствами.

Структурированная обработка исключений в .NET представляет собой методику, предназначенную для работы с исключениями, которые могут возникать на этапе выполнения. Даже в случае программных и пользовательских ошибок, которые ускользнули от глаз программиста, однако, CLR будет часто автоматически генерировать соответствующее исключение с описанием текущей проблемы. В библиотеках базовых классов .NET определено множество различных исключений, таких как FormatException, IndexOutOfRangeException, FileNotFoundException, ArgumentOutOfRangeException и т.д.

В терминологии .NET под "исключением" подразумеваются программные ошибки, пользовательские ошибки и ошибки времени выполнения. Прежде чем погружаться в детали, давайте посмотрим, какую роль играет структурированная обработка исключений, и чем она отличается от традиционных методик обработки ошибок.

В .NET поддерживается стандартная методика для генерации и выявления ошибок в исполняющей среде, называемая **структурированной обработкой исключений (SEH - structured exception handling)**.

Прелесть этой методики состоит в том, что она позволяет разработчикам использовать в области обработки ошибок унифицированный подход, который является общим для всех языков, ориентированных на платформу .NET. Благодаря этому, программист на C# может обрабатывать ошибки почти таким же с синтаксической точки зрения образом, как и программист на VB и программист на С++, использующий C++/CLI.

Дополнительное преимущество состоит в том, что синтаксис, который требуется применять для генерации и перехвата исключений за пределами сборок и машин, тоже выглядит идентично.

Еще одно преимущество механизма исключений .NET состоит в том, что в отличие от запутанных числовых значений, просто обозначающих текущую проблему, они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент, когда изначально возникло исключение. Более того, конечному пользователю можно предоставлять справочную ссылку, которая указывает на определенный URL-адрес с описанием деталей ошибки, а также специальные данные, определенные программистом.

Программирование со структурированной обработкой исключений подразумевает использование четырех следующих связанных между собой сущностей:

- тип класса, который представляет детали исключения;

- член, способный генерировать (throw) в вызывающем коде экземпляр класса исключения при соответствующих обстоятельствах;

- блок кода на вызывающей стороне, ответственный за обращение к члену, в котором может произойти исключение;

- блок кода на вызывающей стороне, который будет обрабатывать (или перехватывать (catch)) исключение в случае его возникновения.

Для того чтобы справиться с возможными ошибочными ситуациями в коде C#, программа обычно делится на блоки трех разных типов:

Блоки **try** инкапсулируют код, формирующий часть нормальных действий программы, которые потенциально могут столкнуться с серьезными ошибочными ситуациями.

Блоки **catch** инкапсулируют код, который обрабатывает ошибочные ситуации, происходящие в коде блока try. Это также удобное место для протоколирования ошибок.

Блоки **finally** инкапсулируют код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков try или catch. Важно понимать, что этот блок выполняется независимо от того, сгенерированo исключение или нет.

Основу обработки исключительных ситуаций в C# составляет пара ключевых слов try и catch. Эти ключевые слова действуют совместно и не могут быть использованы порознь. Ниже приведена общая форма определения блоков try/catch для обработки исключительных ситуаций:

try {

// Блок кода, проверяемый на наличие ошибок.

}

catch (ExcepType1 exOb) {

// Обработчик исключения типа ExcepType1.

}

catch (ExcepType2 exOb) {

// Обработчик исключения типа ExcepType2.

}

...

finally

{

}

где ExcepType — это тип возникающей исключительной ситуации. Когда исключение генерируется оператором try, оно перехватывается составляющим ему пару оператором catch, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор catch. Так, если типы генерируемого исключения и того, что указывается в операторе catch, совпадают, то выполняется именно этот оператор, а все остальные пропускаются. Когда исключение перехватывается, переменная исключения exOb получает свое значение. На самом деле указывать переменную exOb необязательно. Так, ее необязательно указывать, если обработчику исключений не требуется доступ к объекту исключения, что бывает довольно часто. Для обработки исключения достаточно и его типа.

Следует, однако, иметь в виду, что если исключение не генерируется, то блок оператора try завершается как обычно, и все его операторы catch пропускаются. Выполнение программы возобновляется с первого оператора, следующего после завершающего оператора catch. Таким образом, оператор catch выполняется лишь в том случае, если генерируется исключение.

**Практика:** первый аргумент командной строки задаёт имя, второй - порядок вывода имени (- обратный).

Если аргументов нет или недостаточно, то запросить у пользователя. Использовать механизм обработки

исключений.

**Задание 7.**

Теория:

**Регулярные выражения. Цели использования механизма, синтаксис.**

Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex.

Комплексная нотация сопоставления шаблонов регулярных выражений позволяет быстро анализировать большие объемы текста в следующих целях:

- поиск определенных шаблонов символов;

- проверка текста на соответствие предопределенному шаблону (например, адресу электронной почты);

- извлечение, изменение, замена или удаление текстовых подстрок;

- добавление извлеченных строк в коллекцию для создания отчета.

Для многих приложений, которые работают со строками или анализируют большие блоки текста, регулярные выражения — незаменимый инструмент.

Регулярное выражение – это шаблон, который обработчик регулярных выражений пытается сопоставить с введенным текстом. Шаблон состоит из односимвольных или многосимвольных литералов, операторов или конструкций.

Механизм регулярных выражений в .NET представлен классом [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex). Механизм отвечает за синтаксический анализ и компиляцию регулярного выражение, а также выполнение операций, которые сопоставляют шаблон регулярного выражения с входной строкой. Этот механизм представляет центральный компонент объектной модели регулярных выражений .NET.

Можно использовать механизм регулярных выражений одним из двух способов:

Вызывая статичные методы класса [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex). Параметры метода включают в себя входную строку и шаблон регулярного выражения. Механизм регулярных выражений кэширует регулярные выражения, которые используются в вызовах статичных методов, чтобы повторные вызовы статичных методов регулярных выражений, применяющие одно и то же регулярное выражение, выполнялись относительно быстро.

Создавая экземпляр объекта [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex), передавая регулярное выражение конструктору класса. В этом случае объект [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex) не изменяется (доступен только для чтения) и представляет механизм регулярных выражений, который тесно связана с одним регулярным выражением. Так как регулярные выражения, используемые экземплярами [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex), не кэшируются, не следует создавать экземпляр объекта [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex) несколько раз с одним и тем же регулярным выражением.

Можно вызывать методы класса [Regex](https://learn.microsoft.com/ru-ru/dotnet/api/system.text.regularexpressions.regex), чтобы выполнить следующие операции:

- определить, соответствует ли строка шаблону регулярного выражения;

- извлечь одно сопоставление или первое сопоставление;

- извлечь все сопоставления;

- заменить сопоставленную подстроку;

- разделить одну строку на массив строк.

**Синтаксис регулярных выражений**

Некоторые элементы синтаксиса регулярных выражений:

^: соответствие должно начинаться в начале строки

$: конец строки

.: знак точки определяет любой одиночный символ

\*: предыдущий символ повторяется 0 и более раз

+: предыдущий символ повторяется 1 и более раз

?: предыдущий символ повторяется 0 или 1 раз

\s: соответствует любому пробельному символу

\S: соответствует любому символу, не являющемуся пробелом

\w: соответствует любому алфавитно-цифровому символу

\W: соответствует любому не алфавитно-цифровому символу

\d: соответствует любой десятичной цифре

\D : соответствует любому символу, не являющемуся десятичной цифрой

Для составления регулярных выражений лучше использовать справочники (<https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference>) и интернет ресурсы (<http://regexstorm.net/tester>).

**Практика:** Реализовать интерфейс, позволяющий задать имя текстового файла и регулярное выражение

для поиска. Результат работы программы вывод в консоль всех совпадений.

**Задание 8.**

Теория:

**Среда Microsoft .Net. Причины появления, плюсы и минусы использования.**

Одной из целей разработки новой платформы было объединение всех наиболее удачных наработок в рамках единой платформы и их унификация. Кроме того, ставилась задача следования всем актуальным тенденциям в области программирования на тот момент. Так, например, новая платформа должна была напрямую поддерживать объектно-ориентированность, безопасность типов, сборку мусора и структурную обработку исключений.

Платформа .NET Framework — это технология, которая поддерживает создание и выполнение веб-служб и приложений Windows. При разработке платформы .NET Framework учитывались следующие цели:

- Обеспечение согласованной объектно-ориентированной среды программирования для локального сохранения и выполнения объектного кода, для локального выполнения кода, распределенного в Интернете, либо для удаленного выполнения.

- Предоставление среды выполнения кода, в которой:

а) сведена к минимуму вероятность конфликтов в процессе развертывания программного обеспечения и управления его версиями;

б) гарантируется безопасное выполнение кода, включая код, созданный неизвестным или не полностью доверенным сторонним изготовителем;

в) исключаются проблемы с производительностью сред выполнения скриптов или интерпретируемого кода;

- обеспечиваются единые принципы разработки для разных типов приложений, таких как приложения Windows и веб-приложения;

- обеспечивается взаимодействие на основе промышленных стандартов, которое гарантирует интеграцию кода платформы .NET Framework с любым другим кодом.

Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду. В настоящее время .NET Framework развивается в виде [.NET](https://ru.wikipedia.org/wiki/.NET).

Считается, что платформа .NET Framework, появившаяся в 2002 г., является ответом компании [Microsoft](https://ru.wikipedia.org/wiki/Microsoft" \o "Microsoft) на набравшую к тому времени большую популярность платформу [Java](https://ru.wikipedia.org/wiki/Java_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BB%D0%B0%D1%82%D1%84%D0%BE%D1%80%D0%BC%D0%B0)" \o "Java (программная платформа)) компании [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems" \o "Sun Microsystems) (ныне принадлежит [Oracle](https://ru.wikipedia.org/wiki/Oracle" \o "Oracle)). Основное отличие .Net заключалось в том, что платформа официально рассчитывалась на работу с операционными системами семейства Microsoft Windows.

Прошло уже целых 20 лет, но .Net до сих пор пользуется популярностью, несмотря на то, что есть платформы нового поколения, например .Net Core.

Одной из основных идей Microsoft .NET является совместимость программных частей, написанных на разных языках. Сегодня .Net позволяет использовать одни и те же пространства имен, библиотеки и API для разных языков. Например, служба, написанная на C++ для Microsoft .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на C# можно написать класс, наследованный от класса, написанного на Visual Basic .NET, а исключение, созданное методом, написанным на C#, может быть перехвачено и обработано в Delphi. Каждая библиотека (сборка) в .NET имеет сведения о своей версии, что позволяет устранить возможные конфликты между разными версиями сборок.

Языки, поставляемые вместе с [Microsoft Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio" \o "Microsoft Visual Studio):

* [C#](https://ru.wikipedia.org/wiki/C_Sharp)
* [Visual Basic .NET](https://ru.wikipedia.org/wiki/Visual_Basic_.NET)
* [JScript .NET](https://ru.wikipedia.org/wiki/JScript_.NET)
* [C++/CLI](https://ru.wikipedia.org/wiki/C%2B%2B/CLI) — новая версия [Managed C++](https://ru.wikipedia.org/wiki/Managed_C_Plus_Plus" \o "Managed C Plus Plus)
* [F#](https://ru.wikipedia.org/wiki/F_Sharp) — член семейства языков программирования [ML](https://ru.wikipedia.org/wiki/ML), включён в VS2010/VS2012/VS2015/VS2017/VS2019/VS2022
* [J#](https://ru.wikipedia.org/wiki/Visual_J_Sharp) — последний раз был включён в VS2005

Когда программист создает программу на одном из этих языков, то в первую очередь ему необходимо подключить пространство имен System. Это позволяет организовать код программы в логические блоки, объединить и отделить от остального кода некоторую функциональность. Если бы не было .Net, то приходилось бы создавать отдельный System для каждого языка программирования, а это бы уже нарушало один из главных принципов программирования: «Не повторяйся».

Среды разработки, поддерживающие .NET:

* [Microsoft Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio) (C#, Visual Basic .NET, Managed C++, F#)
* [SharpDevelop](https://ru.wikipedia.org/wiki/SharpDevelop)
* [MonoDevelop](https://ru.wikipedia.org/wiki/MonoDevelop)
* [Embarcadero RAD Studio](https://ru.wikipedia.org/wiki/Embarcadero_RAD_Studio) (Delphi for .NET); ранее [Borland Developer Studio](https://ru.wikipedia.org/wiki/Borland_Developer_Studio" \o "Borland Developer Studio) (Delphi for .NET, C#)
* [A#](https://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))
* [Zonnon](https://ru.wikipedia.org/wiki/Zonnon)
* [PascalABC.NET](https://ru.wikipedia.org/wiki/PascalABC.NET)
* [JetBrains Rider](https://ru.wikipedia.org/wiki/JetBrains_Rider)

Приложения .NET также можно разрабатывать в текстовом редакторе, просто вызывая, [компилятор](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80) из [командной строки](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0).

.Net позволяет не устанавливать для каждой программы отдельную среду. Таким образом, уменьшается не только время на установку, но и жесткий диск не забьется лишними копиями немного отличающихся библиотек.

Если говорить о программистах, то фреймворк .Net позволяет заметно сократить время и ресурсы при разработке. С его помощью можно развить одну среду и использовать ее для нескольких языков.

Разработчики, которые знают разные языки, могут писать общий программный продукт под конкретную .NET-платформу. Элементы этого продукта, написанные на разных языках, смогут коммуницировать между собой без каких-либо проблем. Это также объясняет то, почему комьюнити .NET такое большое и разнообразное.

Помимо основных языков, фреймворк поддерживает несколько десятков других языков – это Delphi, Cobra, Oxygene и другие.

Наиболее популярные сферы, где используется технология .Net:

- Веб-разработка;

- Клиентские приложения;

- Компьютерные игры;

- Интернет вещей;

- Enterprise.

**Преимущества**

Независимость от языка. Благодаря CLR ( Common Language Runtime, общеязыковая исполняющая среда) все языки которые поддерживаются .Net компилируются на языке промежуточного уровня. Это позволяет внедрять библиотеки, которые написаны на других языках.

Среда разработки. Visual Studio (VS) одна из лучших IDE (Integrated Development Environment, интегрированная среда разработки) на сегодняшний день. Удобная в использовании, большое количество настроек позволяют подстроить среду под себя.

Сила C#. Объектно-ориентированный язык. Является основным при разработке .Net проектов.

Библиотеки. Интеграция библиотек максимально упрощена благодаря Nuget менеджеру (инструмент для работы с библиотеками). Огромное количество библиотек для различных видов проектов.

**Недостатки**

Дистрибутивы для работы приложений. Для работы приложений .Net на Windows должны быть установлены специальные дистрибутивы. Каждую новую версию дистрибутивов .Net нужно устанавливать отдельно.

Каждая технология постоянно улучшается, зависимости в них меняются. Иногда возникает ситуация, где нужно использовать старую версию какой-нибудь библиотеки. Из-за зависимости версий приходиться изменять версии других библиотек и часто обнаруживается ситуация, что что-то работает некорректно.

**Практика:** Создать класс со счётчиком. Каждый экземпляр класса должен получать свой уникальный номер.

**Задание 9.**

Теория:

**Явное и неявное преобразование типов.**

**Неявные преобразования**. Специальный синтаксис не требуется, так как преобразование всегда завершается успешно и данные не будут потеряны. Примеры включают преобразования из меньших в большие целочисленные типы и преобразования из производных классов в базовые классы. Расширяющие преобразования от типа с меньшей разрядностью к типу с большей разрядностью компилятор проводит неявно.

В следующей таблице приведены предопределенные неявные преобразования между встроенными числовыми типами:

| Исходный тип | Кому |
| --- | --- |
| [sbyte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | short, int, long, float, double, decimal или nint |
| [byte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | short, ushort, int, uint, long, ulong, float, double, decimal, nint или nuint |
| [short](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | int, long, float, double или decimal либо nint |
| [ushort](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | int, uint, long, ulong, float, double или decimal, nint или nuint |
| [int](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | long, float, double или decimal, nint |
| [uint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | long, ulong, float, double или decimal либо nuint |
| [long](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | float, doubleили decimal |
| [ulong](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | float, doubleили decimal |
| [float](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | double |
| [nint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | long, float, double или decimal |
| [nuint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | ulong, float, double или decimal |

Также обратите внимание на следующее:

Любой [целочисленный тип](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) неявно преобразуется в любой [числовой тип с плавающей запятой](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types).

Не поддерживается неявное преобразование в типы byte и sbyte. Не поддерживается неявное преобразование из типов double и decimal.

Не поддерживается неявное преобразование между типом decimal и типами float или double.

Значение константного выражения типа int (например, значение, представленное целочисленным литералом) может быть неявно преобразовано в sbyte, byte, short, ushort, uint, ulong, nint или nuint, если оно находится в диапазоне целевого типа.

**Явные преобразования (приведения)**.

Если преобразование нельзя выполнить без риска потери данных, компилятор требует выполнения явного преобразования, которое называется *приведением*. Приведение — это способ явно указать компилятору, что необходимо выполнить преобразование и что вам известно, что может произойти потеря данных или приведение может завершиться сбоем во время выполнения. Типичными примерами являются числовое преобразование в тип с меньшей точностью или меньшим диапазоном и преобразование экземпляра базового класса в производный класс. Чтобы выполнить приведение, укажите тип, в который производится приведение, в круглых скобках перед преобразуемым значением или переменной.

В следующей таблице показаны предопределенные явные преобразования между встроенными числовыми типами, для которых нет [неявного преобразования](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/numeric-conversions#implicit-numeric-conversions):

| Исходный тип | Кому |
| --- | --- |
| [sbyte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | byte, ushort, uint, ulong или nuint |
| [byte](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte |
| [short](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, ushort, uint, ulong или nuint |
| [ushort](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byteили short |
| [int](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, uint, ulong или nuint |
| [uint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int или nint |
| [long](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, ulong, nint или nuint |
| [ulong](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, long, nint или nuint |
| [float](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, decimal, nint или nuint |
| [double](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, float, decimal, nint или nuint |
| [decimal](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, float, double, nint или nuint |
| [nint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, ulong или nuint |
| [nuint](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, long или nint |

Также обратите внимание на следующее:

* При преобразовании значения целочисленного типа в другой целочисленный тип результат зависит от [контекста проверки переполнения](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/checked-and-unchecked). В проверенном контексте преобразование выполняется успешно, если исходное значение находится в диапазоне конечного типа. В противном случае возникает исключение [OverflowException](https://learn.microsoft.com/ru-ru/dotnet/api/system.overflowexception). В непроверяемом контексте преобразование всегда завершается успешно и выполняется следующим образом.
  + Если исходный тип больше целевого, исходное значение усекается путем отбрасывания его "лишних" самых значимых битов. Результат затем обрабатывается как значение целевого типа.
  + Если исходный тип меньше целевого, исходное значение дополняется знаками или нулями, чтобы иметь тот же размер, что и целевой тип. Знаки добавляются, если исходный тип имеет знак. Если у исходного типа нет знака, добавляются нули. Результат затем обрабатывается как значение целевого типа.
  + Если исходный тип совпадает по размеру с целевым, исходное значение обрабатывается как значение целевого типа.
* При преобразовании значения decimal в целочисленный тип оно округляется в сторону нуля до ближайшего целого значения. Если итоговое целое значение находится вне диапазона целевого типа, возникает исключение [OverflowException](https://learn.microsoft.com/ru-ru/dotnet/api/system.overflowexception).
* При преобразовании значения double или float в целочисленный тип оно округляется в сторону нуля до ближайшего целого значения. Если итоговое целочисленное значение выходит за пределы диапазона целевого типа, результат зависит от [контекста проверки переполнения](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/statements/checked-and-unchecked). В проверенном контексте возникает исключение [OverflowException](https://learn.microsoft.com/ru-ru/dotnet/api/system.overflowexception), а в непроверенном контексте результатом будет неопределенное значение целевого типа.
* При преобразовании из double в float значение double округляется до ближайшего значения float. Если значение double слишком мало или слишком велико для типа float, результатом будет ноль или бесконечность соответственно.
* При преобразовании из float или double в decimal исходное значение преобразуется в представление decimal и при необходимости округляется до ближайшего числа после 28-го десятичного разряда. В зависимости от исходного значения возможны следующие результаты:
  + Если исходное значение слишком мало для представления в виде decimal, результатом будет ноль.
  + Если исходное значение не является числом (NaN), равно бесконечности или слишком велико для представления в виде decimal, возникает исключение [OverflowException](https://learn.microsoft.com/ru-ru/dotnet/api/system.overflowexception).
* При преобразовании из decimal в float или double исходное значение округляется до ближайшего значения float или double соответственно.

Для ссылочных типов явное приведение является обязательным, если необходимо преобразовать базовый тип в производный тип. Операция приведения между ссылочными типами не меняет тип времени выполнения базового объекта; изменяется только тип значения, который используется в качестве ссылки на этот объект.

**Практика:** из аргументов командной строки получить текстовый файл, упорядоченный как в словаре:

лексикографически.