Лекция 12.
Работа со строками.
Работа с файлами

Строки в C#

- Строки являются объектами
- Строки являются **неизменяемыми объектами**, т.е. объект строки нельзя изменить после его создания
- Поэтому все функции, которые работают со строками, возвращают новые строки, а старые – остаются без изменения
- Строки можно сравнивать через == и !=, либо через **Equals**

Документация по строкам

- https://msdn.microsoft.com/ru-ru/library/system.string(v=vs.110).aspx

- Там перечислены все функции строк и их описание

Функции для работы со строками

- int CompareTo(string s)
- Сравнивает строки лексикографически:
  - возвращает 0, если строки равны,
  - положительное число, если данная строка больше переданной
  - отрицательное число – в противном случае
- ```
  string s = "123";
  if (s.CompareTo("344") > 0)
  {
     // код
  }
  ```
  ```

Лексикографическое сравнение строк

- Сравниваем коды первых символов строк. Если один из кодов больше, то это строка больше
- Если коды равны, переходим к проверке следующих символов и т.д.
- Если при этом одна из строк кончилась, то она считается меньше

- Пример верных высказываний:
- "abc" меньше "b", "ab" > "a"

Contains

- bool Contains(string s)
- Проверяет, входит ли переданная строка в данную строку

- Пример:
- string s = "1000 dollars";
- if (s.Contains("1000"))
  {
  Console.WriteLine("Есть 1000");
  }

EndsWith, StartsWith

- bool EndsWith(string s)
- Проверяет, заканчивается ли текущая строка на переданную строку

- bool StartsWith(string s)
- Проверяет, начинается ли текущая строка на переданную строку

IndexOf

- int IndexOf(string s)
- Выдает первый индекс, начиная с которого в текущей строке находится переданная строка. Если переданной строки нет в строке, то выдается -1

- Пример:
- string s = "Of 1004 1004 1004";
- int index = s.IndexOf("1004"); // 3

IndexOf

- int IndexOf(string s, int startIndex)
- Аналогично, только ищет, начиная с переданного индекса startIndex

- Пример:

```
string s = "Of 1004 1004 1004";
int index = s.IndexOf("1004", 4);   // 8
```

lastIndexOf

- int LastIndexOf(string s)
- Выдает последний индекс, начиная с которого в текущей строке находится переданная строка. Если переданной строки нет в строке, то выдается -1

- Пример:

```
string s = "Of 1004 1004 1004";
int index = s.LastIndexOf("1004");   // 13
```

Replace

- string Replace(string toSearch, string replacement)

- Заменяет все вхождения первой переданной строки на вторую переданную строку

- "1004 1004 1004".Replace("1004", "1005")
  // 1005 1005 1005

Split

- string[] Split(string s, StringSplitOptions options)
- Разбивает строку на массив подстрок по указанной строке

- Пример:

```
string[] numberStrings = "1, 2, 3".Split(new string[] { ", " },
    StringSplitOptions.RemoveEmptyEntries);
// массив из элементов "1", "2" и "3"
```

- Параметр StringSplitOptions имеет 2 возможных варианта – оставлять, либо убирать пустые строки из результата
- Обычно нужно их убирать

ToLower, ToUpper

- string ToLower()
- Переводит новый объект строки, который содержит текущую строку, но в нижнем регистре

- string ToUpper()
- Аналогично, только в верхнем регистре

Trim

- string Trim()
- Возвращает новый объект строки, который содержит текущую строку, но в которой обрезаны пробельные символы в начале и конце строки

- Пример:
- " 123\t ".Trim()
  // "123"

Substring

- string Substring(int startIndex, int length)
- Возвращает подстроку, начиная с начального индекса startIndex, длина итоговой строки будет length символов
- Можно указать только начальный индекс, тогда подстрока возьмется до конца строки
- Пример: "123 456".Substring(4, 3); // "456"
- То же самое: "123 456".Substring(4); // "456"

Преобразование строк в числа

- Convert.ToInt32(string s)
- Convert.ToDouble(string s)

- Пример:
- int a = Convert.ToInt32("345"); // 345
- double b = Convert.ToDouble("3.2"); // 3.2

StringBuilder

- Класс StringBuilder используется для формирования больших строк
- StringBuilder sb = new StringBuilder();
  sb.Append("Номер квартиры = ")
  .Append(flatNumber)
  .Append(", номер подъезда = ")
  .Append(entranceNumber);

  string result = sb.ToString();
  // получение результирующей строки

Вызовы Append можно составлять в цепочки

Это потому что Append делает в конце return this;

StringBuilder

- Важные методы:
  - Append() – вставка в конец
  - Remove(int startIndex, int length) – удаление заданного количества символов от начального индекса
  - ToString() – преобразование в строку
  - Length – получение длины строки

Задача «StringBuilder»

- Создать строку из чисел от 1 до 100 через запятую при помощи StringBuilder
- Распечатать строку в консоль

- Не называйте свой класс StringBuilder, чтобы не было конфликтов имен со стандартным классом

Задача «URL»

- Написать программу, которая вычленяет из URL адреса имя сервера. Имеется в виду следующее. Для строки вида http://SomeServerName/abcd/dfdf.htm?dfdf=dfdf вычленить SomeServerName
- Строка может начинаться не обязательно с http, но и с https или чего-то другого. Но :// есть всегда
- Учесть случай, когда после :// больше нет слэша:
- http://SomeServerName
- Использовать IndexOf и Substring

Задача «Разбиение строки»

- Разбить строку "1, 2, 3, 4, 5" и получить массив из этих чисел и найти их сумму

- Использовать Split и Convert.ToInt32

Работа с файлами

Пример файла

- Первое число n – целое, означает количество чисел
- Далее идёт n вещественных чисел

- Пример:

  3 1,3 4,4 5,5

- Хотим прочитать файл и положить вещественные числа в массив

Чтение файлов – отличие в .NET Core

- Код примера будет немного разный в зависимости от типа проекта

- Будет отличаться эта строка кода:
- `StreamReader reader = new StreamReader("..\\..\\input.txt");`

- В .NET Core проекте она должна быть такой:
- `StreamReader reader = new StreamReader("..\\..\\..\\input.txt");`

Чтение файлов

```
using System.IO;
using System;

public class Main {
    public static void Main() {
        // создаем StreamReader, указываем путь к файлу
        // см. следующие слайды про относительные пути и ..
        StreamReader reader = new StreamReader("..\\..\\input.txt");

        // читаем строки при помощи метода ReadLine – в цикле по очереди
        string currentLine;
        while ((currentLine = reader.ReadLine()) != null) {
            // дальше нужно при помощи строковых функций вытащить данные
        }

        // когда мы все прочитали, reader нужно закрыть методом Dispose
        reader.Dispose();
    }
}
```

В .NET Core путь:
“..\\..\\..\\input.txt”

Частые ошибки при запуске примера

- Файл находится не в той папке:
  - Нужно создавать файл в папке project’а (рядом с кодом)
- Неверное имя файла:
  - В Windows по умолчанию скрываются расширения файлов, лучше настроить Проводник так, чтобы расширения показывались
  - Убедиться, что имя файла вместе с расширением совпадает с тем, что написано в коде
- Неверные данные в файле:
  - Должна быть запятая для дробной части вещ-х чисел
  - Убедиться, что содержимое файла соответствует логике задания

Правильная работа с файлами

Закрытие потока

- ```
  public static void Main()
  {
      string filePath = "..\\..\\input.txt";
      using (StreamReader reader = new StreamReader(filePath))
      {
          // работаем с reader'ом
          string x = reader.ReadLine();
      }
  }
  ```
- После того, как работа со **reader'ом** завершена, его обязательно нужно закрывать, вызвав метод **Dispose**()
- Но лучше всего использовать конструкцию using, которая закрывает ресурсы при завершении блока
  ```

Правильное чтение файлов

```
using System.IO;
using System;

public class Main {
    public static void Main() {
        // создаем StreamReader, указываем путь к файлу
        // файл должен лежать в папке bin/Debug
        using (StreamReader reader = new StreamReader("..\\..\\input.txt")) {
            // читаем строки при помощи метода ReadLine – в цикле по очереди
            string currentLine;
            while ((currentLine = reader.ReadLine()) != null) {
                // дальше нужно при помощи строковых функций вытащить данные
            }
        }
    }
}
```

Теперь не нужно вызывать Dispose, using вызовет его сам в конце блока

Зачем нужен using?

- Допустим, у нас такой код:
- ```
  StreamReader reader = new StreamReader("..\\..\\input.txt");
  // произвольный код
  reader.Dispose();
  ```

- Сейчас если код в середине упадет с ошибкой или в нем, например, будет return или другая конструкция перехода, то **Dispose** может не вызваться
- Тогда, если наша программа не завершится, то файл останется незакрытым
- И тогда другие программы не смогут с ним нормально работать, пока программа не завершится
  ```

Зачем нужен using?

- Конструкция using гарантирует, что **Dispose** будет вызван в любом случае:
  - если блок using просто успешно завершится
  - если внутри кода в using будет ошибка
  - если внутри кода в using будет вызван return или другая конструкция для перехода
- Т.е. файл в любом случае закроется, и это правильно

Using при объявлении переменной

- Кроме конструкции using, также можно использовать using при объявлении переменной:
- using StreamReader reader = new StreamReader("input.txt");
- Этот способ эквивалентен конструкции using, которая начинается от объявления переменной и заканчивается в конце области видимости переменной
- Этот способ более краткий и удобный
- Но ресурсы нужно стараться освобождать как можно раньше
- Поэтому этот вариант стоит использовать только в тех местах, где конструкция using кончалась бы именно в конце области видимости переменной

Виды путей

Виды путей

- **Абсолютный путь** – это полный путь (в Windows - с полным указанием буквы диска и т.д.):
  - F:\Users\Pavel\IdeaProjects\Test\folder\input.txt
- **Относительный путь** - это путь относительно некоторой папки:
  - folder\input.txt
- Этот путь указывает туда же, что и абсолютный путь в предыдущем примере относительно папки, которая находится здесь:
  - F:\Users\Pavel\IdeaProjects\Test

Путь к файлу

- **Пример относительного пути:** input.txt
- Такой путь означает, что файл **input.txt** лежит рядом с **exe** файлом:
  - В **.NET Framework** проекте это папка **bin/Debug** внутри проекта
  - В .**NET Core** проекте это папка **bin/Debug/netcoreapp3.1**
    - Вместо 3.1 может быть другая версия
  - Вместо **Debug** может быть **Release** и др.

Специальные символы . и ..

- В относительных путях могут использоваться специальные символы . и ..
- Одна точка означает текущую папку
  Т.е. эквивалентно:
  - input.txt
  - .\input.txt
- Две точки означают родительскую папку
  - ..\input.txt // находится в родительской папке

Путь к файлу

- Именно поэтому и отличаются пути в **.NET Framework** и **.NET Core** проектах при обращении к файлам из project'а
- В **.NET Framework** нам нужно выйти на 2 уровня вверх, а в **.NET Core** – на 3 уровня

- .NET Framework:
- `StreamReader reader = new StreamReader("..\\..\\input.txt");`

- .NET Core:
- `StreamReader reader = new StreamReader("..\\..\\..\\input.txt");`

Еще про чтение файлов

Разбор данных строки

- ```
  string line = reader.ReadLine();
  string[] splits = line.Split(new string[] { " " },
      StringSplitOptions.RemoveEmptyEntries);

  int count = Convert.ToInt32(splits[0]);
  double[] numbers = new double[count];

  for (int i = 1; i <= count; ++i)
  {
      numbers[i - 1] = Convert.ToDouble(splits[i]);
  }
  ```
  ```

Чтение неизвестного количества данных

- Мы пока что изучили только массивы, а у массива фиксированная длина
- Если массив переполнился, то нужно создать новый массив, например, вдвое большей длины, и перекопировать данные туда. А дальше работать с этим новом массивом
- Если он тоже переполнится, то опять создать новый массив и т.д., пока все не будет прочитано

- Существуют классы, которые уже имеют внутри себя такую логику – это списки, они будут рассмотрены на курсе ООП

Работа с большими файлами

- Гипотетически файлы могут быть очень большими, и не влезать в оперативную память
- Поэтому лучше не считывать все содержимое файла в массив или в единую строку
- Лучше обрабатывать файлы построчно

- Чтение файла целиком допускается только если файл небольшой
- Во всех задачах с курса предполагается, что файлы могут быть большими

Что если формат файла более сложный?

- По сути файл придется разбирать вручную, есть 2 подхода, которые при желании можно комбинировать:
  - Читать из файла строки. Разбирать эти строки вручную через функции работы со строками (например, **split**, **substring** и др.)
  - Использовать **регулярные выражения**

- В любом случае вы должны знать формат файла, иначе прочитать его не удастся

Регулярные выражения

- **Регулярные выражения** – это специальный язык для поиска подстрок в тексте
- Регулярные выражения позволяют найти подстроки, соответствующие некоторому **шаблону**, а потом с этими подстроками можно работать – получать, заменять, удалять их и т.д.

Регулярные выражения – пример задачи

- Допустим, мы хотим найти все даты в этом тексте:
- Матч Египет-Уругвай прошел **15.06.2018**, а матч Португалия-Испания прошел **16.06.2018**.

- Чтобы найти даты, нам понадобится такое регулярное выражение: **\d{2}\.\d{2}\.\d{4}**
- **\d** - означает цифра (т.е. символ 0-9)
- **\.** – означает символ «точка». Здесь нужно экранирование, т.к. у точки в регулярных выражениях особый смысл
- **{n}** – это означает, что последовательность должна встречаться **n** раз

Регулярные выражения – пример задачи

- **\d{2}\.\d{2}\.\d{4}**
- В итоге получаем, что сначала идет 2 цифра, потом точка, потом еще 2 цифры, потом точка, потом 4 цифры
- Все подстроки такого вида могут быть найдены

- Во всех языках программирования есть некоторые функции или классы для работы с регулярными выражениями
- Кстати, в Java функция **split** принимает именно регулярное выражение, а не просто строку

Запись в файл

Запись в файл

- ```
  StreamWriter writer = new StreamWriter("output.txt");
  writer.WriteLine("OK!");
  writer.Dispose();
  ```

- Класс StreamWriter имеет те же методы, что Console, т.е. можно использовать **Write**, **WriteLine**
- Все это будет записываться в файл
- Файл с указанным именем будет создан если его нет, либо перезаписан, если файл уже существует
- Как и при чтении, после окончания работы, **writer** нужно закрыть при помощи метода **Dispose**
  ```

Правильное закрытие потока

```
public static void Main()
{
    using (StreamWriter writer = new StreamWriter("output.txt"))
    {
        // что-то пишем во writer
        writer.WriteLine("OK");
    }
}
```

- После того, как работа с **writer'**ом завершена, его обязательно нужно закрывать, вызвав метод **Dispose**()
- Но лучше всего использовать конструкцию using, которая закрывает ресурсы при завершении блока, автоматически вызывая метод **Dispose**

Несколько блоков using

- Блоки using можно вкладывать друг в друга
-
```
public static void Main()
{
    using (StreamReader reader = new StreamReader("input.txt"))
    {
        using (StreamWriter writer = new StreamWriter("out.txt"))
        {
            string currentLine;
            while ((currentLine = reader.ReadLine()) != null)
            {
                writer.WriteLine(currentLine);
            }
        }
    }
}
```

Копирование текстового файла вручную

ReadToEnd

- У StreamReader кроме метода **ReadLine**(), который читает одну строку, еще есть метод **ReadToEnd**()
- **ReadToEnd()** выдает все содержимое файла от текущей позиции до конца файла в виде строки
- Этот метод удобно использовать для небольших файлов, чтобы сразу прочитать все содержимое
-
```
using (StreamReader reader = new StreamReader("..\\..\\in.txt"))
{
    string x = reader.ReadToEnd();  // прочитали все строки
}
```
- Этот метод не отбрасывает переводы строки, которые были в файле

Работа с файлами в целом

Классы и методы для работы с файлами

- Мы рассматривали классы для работы с содержимым текстовых файлов
- Но часто бывает нужно работать с самими файлами (например, копировать, перемещать, удалять файлы или папки)
- Здесь документация, в которой есть примеры самых частых задач:
- https://docs.microsoft.com/ru-ru/dotnet/standard/io/common-i-o-tasks?view=netframework-4.7.2

Домашняя работа

Задача «Перевод файла в верх.регистр»

- Написать программу, которая читает строки файла, переводит их в верхний регистр и записывает результат во второй файл

Задача на дом «Число вхождений»

- Прочитать текст из файла, и написать функцию, которая считает количество вхождений некоторой строки в этот текст без учета регистра символов

- Использовать цикл и **IndexOf**, который принимает начальный индекс, с которого искать