Лекция №5
по курсу
«Системное программирование»

Лектор: д.т.н., Оцоков Шамиль Алиевич,

email: otsokovShA@mpei.ru

Москва, 2021

Работа со строками

Извлечение подстроки

```
string path = "C:\\Pics\\CoolPic.jpg";
string fileName = path.Substring(8, 7);
// fileName = "CoolPic"
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	:	\	P	i	c	s	\	C	o	o	l	P	i	c	.	j	p	g

Let's consider again the example of the well-known **file path**. If we have no information about the exact contents of the variable, but we know that it contains a file path, we can stick to the above scheme:

```
string path = "C:\\Pics\\CoolPic.jpg";
int index = path.LastIndexOf("\\");
// index = 7
string fullName = path.Substring(index + 1);
// fullName = "CoolPic.jpg"
```

Работа со строками

Разбиение строки на подстроки по разделителям

Splitting Strings by Multiple Separators – Example

The easiest and more flexible method for resolving this issue is the following:

```
char[] separators = new char[] {' ', ',', '.'};
string[] beersArr = listOfBeers.Split(separators);
```

Using the built-in functionality of the method **Split(…)** from the class **String**, we will split the contents of a given string by array of characters – separators, which are passed as an argument of the method. All substrings among which are space, comma or dot will be removed and stored in the **beersArr** array.

Удаление пустых слов

```
foreach (string beer in beersArr)
{
  if (beer != "")
  {
    Console.WriteLine(beer);
  }
}
```

```
string[] beersArr = listOfBeers.Split(
  separators, StringSplitOptions.RemoveEmptyEntries);
```

Работа со строками

Замена одной строки на другую строку

```
string doc = "Hello, some@gmail.com, " +
  "you have been using some@gmail.com in your registration.";
string fixedDoc =
  doc.Replace("some@gmail.com", "john@smith.com");
Console.WriteLine(fixedDoc);

// Console output:
// Hello, john@smith.com, you have been using
// john@smith.com in your registration.
```

As it can be seen from the example, the method **Replace(…) replaces all occurrences** of a given substring with another substring, not just the first.

Удаление пустых символов

```
string fileData = "   \n\n     David Allen      ";
```

If we print the contents to the console, we get two blank lines followed by some spaces, the requested name and some additional spaces at the end. We can reduce the information just to the required name, in the following way:

```
string reduced = fileData.Trim();
```

Работа со строками

Удаление ненужных символов с конца и начала строки по списку

Removing Unnecessary Characters by a Given List

The method **Trim(…)** can accept an array of characters, which we want to remove from the string. We can make it in the following way:

```
string fileData = "   111 $  %    David Allen  ### s   ";
char[] trimChars = new char[] {' ', '1', '$', '%', '#', 's'};
string reduced = fileData.Trim(trimChars);
// reduced = "David Allen"
```

Again, we get the desired result "**David Allen**".

Please note that we must list all the characters we want to eliminate, including the empty spaces (spaces, tabs, new line, etc.). Without a ' ' in the array trimChars, we would not get the desired result!

Работа со строками StringBuilder

Не делайте в цикле объединение строк с помощью класса String

```
string str1 = "Super";
string str2 = "Star";
string result = str1 + str2;
```

What will happen with the memory? Creating the variable **result** will allocate a new area in dynamic memory, which will record the outcome of the **str1 + str2**, which is "**SuperStar**". Then the variable itself will keep the address of the allocated area. As a result we will have three areas in memory and three references to them. This is convenient, but allocating a new area, recording a value, creating a new variable and referencing it in the memory is time-consuming process that would be a problem when repeated many times, typically inside a loop.

```
string collector = "Numbers: ";
for (int index = 1; index <= 20000; index++)
{
  collector += index;
}
```

Может выполняться около 1-2 сек

Работа со строками StringBuilder

Правильный вариант для объединения строк

```
class ElegantNumbersConcatenator
{
  static void Main()
  {
    Console.WriteLine(DateTime.Now);

    StringBuilder sb = new StringBuilder();
    sb.Append("Numbers: ");

    for (int index = 1; index <= 200000; index++)
     {
       sb.Append(index);
     }

     Console.WriteLine(sb.ToString().Substring(0, 1024));
     Console.WriteLine(DateTime.Now);
  }
}
```

Работа со строками StringBuilder

Вывод в обратном порядке строки

Reversing a String – Example

Consider another example: we want to reverse an existing string (backwards). For example, if we have the string "**abcd**", the returned result should be "**dcba**". We get the original string, iterate it backwards character by character and add each character to a variable of type **StringBuilder**:

```
public class WordReverser
{
  static void Main()
  {
    string text = "EM edit";
    string reversed = ReverseText(text);
```

```
    Console.WriteLine(reversed);

    // Console output:
    // tide ME
  }

  static string ReverseText(string text)
  {
    StringBuilder sb = new StringBuilder();
    for (int i = text.Length - 1; i >= 0; i--)
    {
      sb.Append(text[i]);
    }
    return sb.ToString();
  }
```

Пространства имён

Пространство имен (пакет) в ООП мы называем контейнером для группы классов, которые объединены общей функцией или используются в общем контексте. Пространства имен способствуют лучшей логической организации исходного кода, создавая семантическое разделение классов по категориям и облегчая их использование в исходном коде. Теперь мы рассмотрим пространства имен в C # и увидим, как мы можем их использовать.

Все определяемые классы и структуры, как правило, не существуют сами по себе, а заключаются в специальные контейнеры - пространств имен. Создаваемый по умолчанию класс Program уже находится в пространстве имен, которое обычно совпадает с названием проекта:

```
1  namespace HelloApp
2  {
3      class Program
4      {
5          static void Main(string[] args)
6          {
7          }
8      }
9  }
```

Пространство имен определяется с помощью ключевого слова **namespace**, после которого идет название. Так в данном случае полное название класса Program будет HelloApp.Program.

Пространства имён

Класс Program видит все классы, которые объявлены в том же пространстве имен:

```
namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
             Account account = new Account(4);
        }
    }
    class Account
    {
        public int Id { get; private set;} // номер счета
        public Account(int _id)
        {
            Id = _id;
        }
    }
}
```

Пространства имён

Но чтобы задействовать классы из других пространств имен, эти пространства надо подключить с помощью директивы using:

```
using System;
namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("hello");
        }
    }
}
```

Здесь подключается пространство имен System, в котором определен класс Console. Иначе нам бы пришлось писать полный путь к классу:

```
static void Main(string[] args)
{
    System.Console.WriteLine("hello");
}
```

Псевдонимы

Для различных классов мы можем использовать псевдонимы. Затем в программе вместо названия класса используется его псевдоним. Например, для вывода строки на экран применяется метод `Console.WriteLine()`. Но теперь зададим для класса Console псевдоним:

```
using printer = System.Console;
class Program
{
    static void Main(string[] args)
    {
        printer.WriteLine("Hello from C#");
        printer.Read();
    }
}
```

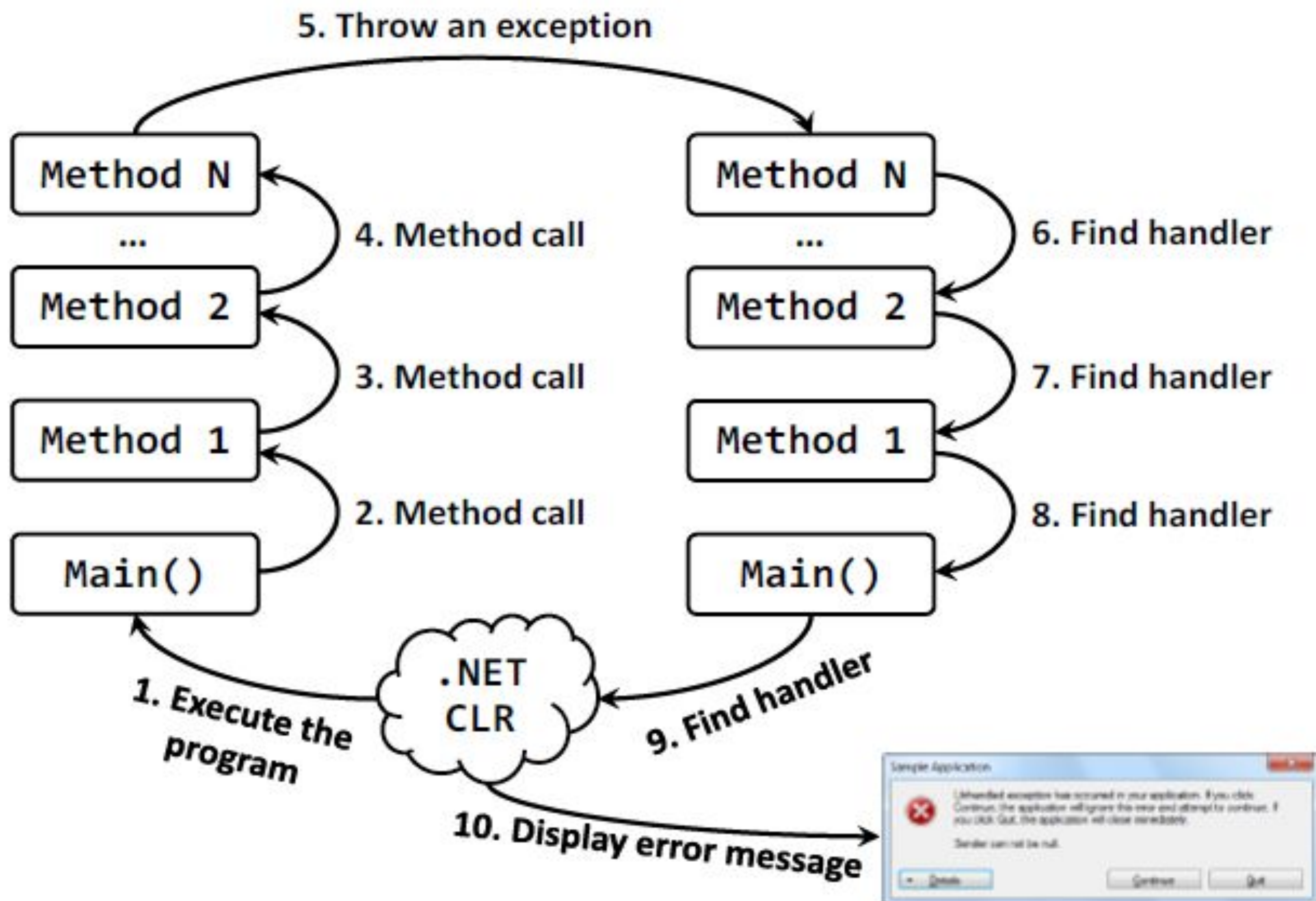
Исключения

Исключение - это уведомление о том, что что-то прерывает нормальное выполнение программы. Исключения обеспечивают парадигму программирования для обнаружения неожиданных событий и реагирования на них. Когда возникает исключение, состояние программы сохраняется, нормальный поток прерывается, и управление передается обработчику исключения (если таковой существует в текущем контексте). Исключения возникают или генерируются программным кодом, который должен послать сигнал исполняющейся программе об ошибке или необычной ситуации. Перехват и обработка исключений Обработка исключений - это механизм, который позволяет генерировать и перехватывать исключения. Этот механизм предоставляется внутри CLR (Common Language Runtime). Части инфраструктуры обработки исключений - это языковые конструкции C # для генерации и перехвата исключений.
В объектно-ориентированном программировании (ООП) исключения являются мощным механизмом для централизованной обработки ошибок и исключительных ситуаций. Этот механизм заменяет процедурно-ориентированный метод обработки ошибок, в котором каждая функция возвращает код, указывающий на ошибку или успешное выполнение. Обычно в ООП код, выполняющий какую-либо операцию, вызывает исключение, если есть проблема и операция не может быть успешно завершена.

Исключения

ООП исключения - это классы, и их можно наследовать для построения иерархий. Когда исключение обрабатывается (перехватывается), механизм обработки может перехватить целый класс исключений, а не только конкретную ошибку (как в традиционном процедурном программировании)
Иногда исключения используются не столько для того, чтобы сигнализировать о проблеме, сколько для обработки некоторого ожидаемого события. Это не считается хорошей практикой, поскольку исключения не должны контролировать нормальный ход программы.
В ООП рекомендуется использовать исключения для управления ошибочными ситуациями или неожиданными событиями, которые могут возникнуть во время выполнения программы.
Выявление исключений в C # После того, как метод выдает исключение, CLR ищет обработчик исключения, который может обработать ошибку.

Исключения



Исключения

```
try
{
// Some code that may throw an exception
}
catch (ExceptionType objectName)
{
// Code handling an Exception
}
catch (ExceptionType objectName)
{
// Code handling an Exception
}
finally
{
// This code will always execute
}
```

Выбрасывание исключений (конструкция throw) Исключения в C # генерируются с помощью ключевого слова throw. Нам нужно предоставить экземпляр исключения, содержащий всю необходимую информацию об ошибке.

```
static void Main()
{
Exception e = new Exception("There was a problem");
throw e;
}
```

Исключения

Исключения в .NET бывают двух типов - системные и прикладные. Системные исключения определены в библиотеках .NET и используются фреймворком, в то время как исключения приложений определяются разработчиками приложений и используются прикладным программным обеспечением

Потоки

Поток - это упорядоченная последовательность байтов, которая отправляется из одного приложения или устройства ввода в другое приложение или устройство вывода. Эти байты записываются и читаются один за другим и всегда прибывают в том же порядке, в котором они были отправлены. Потоки - это абстракция канала передачи данных, который соединяет два устройства или приложения. Потоки - это основное средство обмена информацией в компьютерном мире. Благодаря потокам различные приложения могут получать доступ к файлам на компьютере и могут устанавливать сетевое взаимодействие между удаленными компьютерами. В мире компьютеров многие операции можно интерпретировать как чтение и запись в поток. Например, печать - это процесс отправки последовательности байтов в поток, связанный с соответствующим портом, к которому подключен принтер.
Каждый раз, когда вы читаете или записываете из или в файл, вам нужно открыть поток в соответствующий файл, выполнить чтение или запись

Потоки

Основные операции с потоками.
С потоками можно выполнять следующие операции: создание / открытие, чтение данных, запись данных, поиск / позиционирование, закрытие /
Creation
To **create or open a stream**
Reading
Reading means extracting data from the stream.
Writing
Writing means sending data to the stream in a specific way.
Positioning
Positioning or seeking in the stream
Closing
To close or disconnect a stream

Потоки

Binary and Text Streams

FileStream, BinaryReader и BinaryWriter.

Text Streams

Text streams are very similar to binary, but only **work with text data** or rather a sequence of characters (**char**) and strings (**string**).

ReadLine() – reads one line of text and returns a string.
- **ReadToEnd()** – reads the entire stream to its end and returns a string.
- **Write()** – writes a string to the stream.
- **WriteLine()** – writes one line of text into the stream.

```
// Create a StreamReader connected to a file
StreamReader reader = new StreamReader("test.txt");
// Read the file here …
// Close the reader resource after you've finished using it
reader.Close();
```

Чтение текстового файла построчно

Reading a Text File Line by Line – Example

```
class FileReader
{
static void Main()
{
// Create an instance of StreamReader to read from a file
StreamReader reader = new StreamReader("Sample.txt");
int lineNumber = 0;
// Read first line from the text file
string line = reader.ReadLine();
// Read the other lines from the text file
while (line != null)
{
lineNumber++;
Console.WriteLine("Line {0}: {1}", lineNumber, line);
line = reader.ReadLine();
}
// Close the resource after you've finished using it
reader.Close();
}
}
```

Чтение текстового файла построчно

Автоматическое закрытие потока после работы с ним Как отмечалось в предыдущем примере, закончив работу с объектом типа StreamReader, мы вызвали Close () и закрыли поток за объектом StreamReader. Однако очень часто начинающие программисты забывают вызвать метод Close (), тем самым блокируя используемый файл.
Конструкция C # using (…) гарантирует, что после выхода из тела автоматически будет вызван метод Close (). Это произойдет, даже если при чтении файла возникнет исключение.

Чтение текстового файла построчно

```
class FileReader
{
static void Main()
{
// Create an instance of StreamReader to read from a file
StreamReader reader = new StreamReader("Sample.txt");
using (reader)
{
int lineNumber = 0;
// Read first line from the text file
string line = reader.ReadLine();
// Read the other lines from the text file
while (line != null)
{
lineNumber++;
Console.WriteLine("Line {0}: {1}", lineNumber, line);
line = reader.ReadLine();
}
}
}
}
```

Чтение текстового файла построчно

```
class FileReader
{
static void Main()
{
// Create an instance of StreamReader to read from a file
StreamReader reader = new StreamReader("Sample.txt");
using (reader)
{
int lineNumber = 0;
// Read first line from the text file
string line = reader.ReadLine();
// Read the other lines from the text file
while (line != null)
{
lineNumber++;
Console.WriteLine("Line {0}: {1}", lineNumber, line);
line = reader.ReadLine();
}
}
}
}
```

Чтение текстового файла построчно

Reading a Cyrillic Content

You probably already guessed that if we want to read from a file that contains characters from the Cyrillic alphabet, we must use the correct encoding that "understands" correctly these special characters. Typically, in a Windows environment, text files, containing Cyrillic text, are stored in Windows-1251 encoding. To use it, we should set it as the encoding of the stream, which our StreamReader will process: Encoding win1251 = Encoding.GetEncoding("Windows-1251");
StreamReader reader = new StreamReader("test.txt", win1251);

Стандарт Юникода. Чтение в Юникоде

Unicode - это отраслевой стандарт, который позволяет компьютерам и другим электронным устройствам всегда представлять и манипулировать текстом, который был написан в большинстве мировых литературных источников. Он состоит из более чем 100 000 символов, а также различных схем кодирования (кодировок). Объединение различных символов, которые предлагает Unicode, приводит к его большему распространению. Как вы знаете, символы в C # (типы char и string) также представлены в Unicode.

Чтобы читать символы, хранящиеся в Unicode, мы должны использовать одну из поддерживаемых схем кодирования для этого стандарта. Самым популярным и широко используемым является UTF-8. Мы можем установить его как схему кода уже знакомым способом

Запись текстового файла построчно

Writing to a Text File

Text files are very convenient for storing various types of information. For example, we can record the results of a program. We can use text files to make something like a journal (log) for the program – a convenient way to monitor it at runtime.

Again, as with reading a text file, we will use a similar to the **Console** class when writing, called **StreamWriter**.

```
static void Main()
{
  // Create a StreamWriter instance
  StreamWriter writer = new StreamWriter("numbers.txt");

  // Ensure the writer will be closed when no longer used
  using(writer)
  {
    // Loop through the numbers from 1 to 20 and write them
    for (int i = 1; i <= 20; i++)
    {
      writer.WriteLine(i);
    }
  }
}
```

Упражнения

1. Write a program that reads a text file and **prints its odd lines** on the console.
2. Write a program that **joins two text files** and records the results in a third file.
3. Write a program that reads the contents of a text file and **inserts the line numbers** at the beginning of each line, then rewrites the file contents.
4. Write a program that **compares two text files** with the same number of rows line by line and prints the number of equal and the number of different lines.
5. Write a program that reads a square matrix of integers from a file and **finds the sub-matrix with size 2 × 2 that has the maximal sum** and writes this sum to a separate text file. The first line of input file contains the size of the recorded matrix (N). The next N lines contain N integers

Считывание с исключением

```
class HandlingExceptions
{
static void Main()
{
string fileName = @"somedir\somefile.txt";
try
{
StreamReader reader = new
StreamReader(fileName);
Console.WriteLine(
"File {0} successfully opened.", fileName);
Console.WriteLine("File contents:");
using (reader)
{
Console.WriteLine(reader.ReadToEnd());
}
}
catch (FileNotFoundException)
{
Console.Error.WriteLine(
"Can not find file {0}.",
fileName);
}
catch
(DirectoryNotFoundException)
{
Console.Error.WriteLine(
"Invalid directory in the file
path.");
}
catch (IOException)
{
Console.Error.WriteLine(
"Can not open the file {0}",
fileName);
}
}
```

Подсчёт слов

```
static void Main()
{
   string fileName = @"..\..\sample.txt";
   string word = "C#";
   try
   {
      StreamReader reader = new StreamReader(fileName);
      using (reader)
      {
         int occurrences = 0;
         string line = reader.ReadLine();
         while (line != null)
         {
            int index = line.IndexOf(word);
            while (index != -1)
            {
               occurrences++;
               index = line.IndexOf(word, (index + 1));
            }
            line = reader.ReadLine();
         }
         Console.WriteLine(
            "The word {0} occurs {1} times.", word, occurrences);
      }
   }
   catch (FileNotFoundException)
```

Подсчёт слов

```
    {
        Console.Error.WriteLine(
            "Can not find file {0}.", fileName);
    }
    catch (IOException)
    {
        Console.Error.WriteLine(
            "Cannot read the file {0}.", fileName);
    }
}
```

Безопасное считывание файла

```
static void ReadFile(string fileName)
{
  TextReader reader = null;
  try
  {
    reader = new StreamReader(fileName);
    string line = reader.ReadLine();
    Console.WriteLine(line);
  }
  finally
  {
    // Always close "reader" (if it was opened)
    if (reader != null)
    {
      reader.Close();
    }
  }
}
```

Упражнение

Напишите программу, которая считывает список имен из текстового файла, упорядочивает их в алфавитном порядке и записывает в другой файл. Строки пишутся по одной в строке.

7. Напишите программу, которая заменяет каждое вхождение подстроки «начало» на «конец» в текстовом файле. Можете ли вы переписать программу, чтобы заменить только целые слова? Программа работает с большими файлами (например, 800 МБ)?
8. Напишите предыдущую программу так, чтобы она изменяла только слова целиком (а не части слова).