Министерство образования и науки Пермского края
государственное бюджетное профессиональное образовательное учреждение
«Пермский химико-технологический техникум»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
ДЛЯ ОБУЧАЮЩИХСЯ
ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

для специальности 09.02.07 «Информационные системы и программирование»

ОП.04 ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
ПРАВИЛА ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ 6
ОПИСАНИЕ РАБОЧЕГО МЕСТА ОБУЧАЮЩЕГОСЯ 7
ПРАКТИЧЕСКИЕ РАБОТЫ 8
Практическая работа № 1 8
Практическая работа № 2 8
Практическая работа № 3 8
Практическая работа № 4 8
Практическая работа № 5 12
Практическая работа № 6 15
Практическая работа № 7 15
Практическая работа №8 15
Практическая работа №9 15
Практическая работа №10 15
Практическая работа №11 15
Практическая работа №12 15
Практическая работа №13 16
Практическая работа №14 16
Практическая работа №15 16
Практическая работа №16 16
Практическая работа №17 16
Практическая работа №18 16
Практическая работа №19 16
Практическая работа №20 20
Практическая работа №21 20
Практическая работа №22 26
Практическая работа №23 27
Практическая работа №24 35
Практическая работа №25 38
Практическая работа №26 41
Практическая работа №27 41
Практическая работа №28 41
Практическая работа №29 41
Практическая работа №30 42
Практическая работа №31 42
Практическая работа №32 42
Практическая работа №33 42

Практическая работа №34 43
Практическая работа №35 45
Практическая работа №36 48
Практическая работа №37 48
Практическая работа №38 48
Практическая работа №39 58
Практическая работа №40 62
Практическая работа №41 65
Практическая работа №42 67
Практическая работа №43 67
Практическая работа №44 67
Практическая работа №45 79
Практическая работа №46 85
Практическая работа №47 89
Практическая работа №48 90
Практическая работа №49 96

ВВЕДЕНИЕ

Рабочая программа учебной дисциплины ОП.04 Основы алгоритмизации и программирования является частью основной образовательной программы ГБПОУ «ПХТТ» в соответствии с ФГОС по специальности СПО: 09.02.07 Информационные системы и программирование**.**

В результате изучения дисциплины обучающийся должен уметь:
– работать в среде программирования;
– использовать языки программирования высокого уровня.

В результате освоения дисциплины обучающийся должен знать:
– типы данных;
– базовые конструкции изучаемых языков программирования;
– интегрированные среды программирования на изучаемых языках.

В результате освоения дисциплины обучающийся осваивает элементы общих компетенций (ОК):

Шифр комп.	**Наименование компетенций**	**Дескрипторы (показатели сформированности)**	**Умения**	**Знания**
ОК 1.	***Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам.***	Распознавать сложные проблемы в знакомых ситуациях. Выделять сложные составные части проблемы и описывать её причины и ресурсы, необходимые для её решения в целом. Определять потребность в информации и предпринимать усилия для её поиска. Выделять главные и альтернативные источники нужных ресурсов. Разрабатывать детальный план действий и придерживаться его. Оценивать результат своей работы, выделять в нём сильные и слабые стороны.	Распознавать задачу и/или проблему в профессиональном и/или социальном контексте. Анализировать задачу и/или проблему и выделять её составные части. Правильно определить и найти информацию, необходимую для решения задачи и/или проблемы. Составить план действия, Определить необходимые ресурсы. Владеть актуальными методами работы в профессиональной и смежных сферах. Реализовать составленный план. Оценить результат и последствия своих действий (самостоятельно или с помощью	Актуальный профессиональный и социальный контекст, в котором приходится работать и жить. Основные источники информации и ресурсов для решения задач и проблем в профессиональном и/или социальном контексте. Актуальные стандарты выполнения работ в профессиональной и смежных областях. Актуальные методы работы в профессиональной и смежных сферах.

			наставника).	
ОК 2.	***Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности.***	Планировать информационный поиск из широкого набора источников, необходимого для выполнения профессиональных задач. Проводить анализ полученной информации, выделять в ней главные аспекты. Структурировать отобранную информацию в соответствии с параметрами поиска. Интерпретировать полученную информацию в контексте профессиональной деятельности.	Определять задачи поиска информации. Определять необходимые источники информации. Планировать процесс поиска. Структурировать получаемую информацию. Выделять наиболее значимое в перечне информации. Оценивать практическую значимость результатов поиска. Оформлять результаты поиска.	Номенклатура информационных источников, применяемых в профессиональной деятельности. Приемы структурирования информации. Формат оформления результатов поиска информации.
ОК 3.	***Планировать и реализовывать собственное профессиональное и личностное развитие***		определять актуальность нормативно-правовой документации в профессиональной деятельности; выстраивать траектории профессионального и личностного развития	содержание актуальной нормативно-правовой документации; современная научная и профессиональная терминология; возможные траектории профессионального развития и самообразования

Методические указания предназначены для проведения практических занятий по дисциплине ОП.04 Основы алгоритмизации и программирования, закрепления теоретических знаний и получения навыков работы в области прикладного программирования.

Методические указания включают 16 практических работ по темам раздела «Основы алгоритмизации и программирования», 19 практических работ по темам раздела «Объектно-ориентированное программирование» и 14 практических работ по темам раздела «Разработка WPF приложений». Каждая практическая работа содержит сведения о теме, цели ее проведения и формируемых компетенциях, включает пояснения к работе, содержание отчета, контрольные задания или вопросы, список литературы.

К выполнению практических работ обучаемые приступают после подробного изучения соответствующего теоретического материала и прохождения инструктажа по технике безопасности.

Характер практических работ репродуктивный и частично-репродуктивный.

ПРАВИЛА ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ

1. Практические работы проводятся под наблюдением преподавателя.
2. К выполнению практических работ студенты допускаются только после прослушивания инструктажа по технике безопасности и противопожарным мерам. После инструктажа каждый студент расписывается в журнале.
2. Строго выполнять правила техники безопасности и санитарно-гигиенические нормы при работе в кабинете.

ОПИСАНИЕ РАБОЧЕГО МЕСТА ОБУЧАЮЩЕГОСЯ

1. Практические работы по ОП.04 Основы алгоритмизации и программирования выполняются в компьютерном классе.
2. Для выполнения практических работ необходимо:
    a. Персональный компьютер;
    b. Операционная система Windows;
    c. Приложения MS Office;
    d. Microsoft Visual Studio 2019;
    e. Методические указания.

ПРАКТИЧЕСКИЕ РАБОТЫ

Практическая работа № 1

Тема: Составление блок-схем алгоритмов линейной структуры (2)
стр. 3-11 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа № 2

Тема: Составление блок-схем алгоритмов разветвляющейся структуры (4)
стр.11-19 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа № 3

Тема: Составление блок-схем алгоритмов циклической структуры
стр.20-27 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа № 4

Тема: Объявление переменных и их инициализация. Область действия и время существования переменных. Константы: определение, виды и правила записи в программе

Теоретический материал

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.

Например, определим простейшую переменную:

```
string name;
```

В данном случае определена переменная name, которая имеет тип **string**. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:

```
string name;
string Name;
```

После определения переменной можно присвоить некоторое значение:

```
string name;
name = "Tom";
```

Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение.

Также мы можем сразу при определении присвоить переменной значение. Данный прием называется инициализацией:

```
string name = "Tom";
```

Отличительной чертой переменных является то, что в программе можно многократно менять их значение. Например, создадим небольшую программу, в которой определим переменную, поменяем ее значение и выведем его на консоль:

```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            string name = "Tom";  // определяем переменную и инициализируем ее

            Console.WriteLine(name);    // Tom

            name = "Bob";       // меняем значение переменной
            Console.WriteLine(name);    // Bob

            Console.Read();
        }
    }
}
```

Консольный вывод программы:

```
Tom
Bob
```

Литералы представляют неизменяемые значения (иногда их еще называют константами). Литералы можно передавать переменным в качестве значения. Литералы бывают логическими, целочисленными, вещественными, символьными и строчными. И отдельный литерал представляет ключевое слово null.

Логические литералы

Есть две логических константы - **true** (истина) и **false** (ложь):

```
Console.WriteLine(true);
Console.WriteLine(false);
```

Целочисленные литералы

Целочисленные литералы представляют положительные и отрицательные целые числа, например, 1, 2, 3, 4, -7, -109. Целочисленные литералы могут быть выражены в десятичной, шестнадцатеричной и двоичной форме.

С целыми числами в десятичной форме все должно быть понятно, так как они используются в повседневной жизни:

```
Console.WriteLine(-11);
Console.WriteLine(5);
Console.WriteLine(505);
```

Числа в двоичной форме предваряются символами 0b, после которых идет набор из нулей и единиц:

```
Console.WriteLine(0b11);        // 3
Console.WriteLine(0b1011);      // 11
Console.WriteLine(0b100001);    // 33
```

Для записи числа в шестнадцатеричной форме применяются символы 0x, после которых идет набор символов от 0 до 9 и от A до F, которые собственно представляют число:

```
Console.WriteLine(0x0A);    // 10
Console.WriteLine(0xFF);    // 255
Console.WriteLine(0xA1);    // 161
```

Вещественные литералы

Вещественные литералы представляют вещественные числа. Этот тип литералов имеет две формы. Первая форма - вещественные числа с фиксированной запятой, при которой дробную часть отделяется от целой части точкой. Например:

```
3.14
100.001
```

-0.38

Также вещественные литералы могут определяться в экспоненциальной форме MEp, где M — мантисса, E - экспонента, которая фактически означает "*10^" (умножить на десять в степени), а p — порядок. Например:

```
Console.WriteLine(3.2e3);   // по сути равно 3.2 * 10<sup>3</sup> = 3200
Console.WriteLine(1.2E-1);  // равно 1.2 * 10<sup>-1</sup> = 0.12
```

Символьные литералы

Символьные литералы представляют одиночные символы. Символы заключаются в одинарные кавычки.

Символьные литералы бывают нескольких видов. Прежде всего это обычные символы:

```
'2'
'A'
'T'
```

Специальную группу представляют **управляющие последовательности** Управляющая последовательность представляет символ, перед которым ставится обратный слеш. И данная последовательность интерпретируется определенным образом. Наиболее часто используемые последовательности:

- '\n' - перевод строки
- '\t' - табуляция
- '\' - обратный слеш

И если компилятор встретит в тексте последовательность \t, то он будет воспринимать эту последовательность не как слеш и букву t, а как табуляцию - то есть длинный отступ.

Также символы могут определяться в виде шестнадцатеричных кодов, также заключенный в одинарные кавычки.

Еще один способ определения символов представляет использования шестнадцатеричных кодов ASCII. Для этого в одинарных кавычках указываются символы '\x', после которых идет шестнадцатеричный код символа из таблицы ASCII. Коды символов из таблицы ASCII можно посмотреть здесь.

Например, литерал '\x78' представляет символ "x":

```
Console.WriteLine('\x78');    // x
Console.WriteLine('\x5A');    // Z
```

И последний способ определения символьных литералов представляет применение кодов из таблицы символов Unicode. Для этого в одинарных кавычках указываются символы '\u', после которых идет шестнадцатеричный код Unicode. Например, код '\u0411' представляет кириллический символ 'Б':

```
Console.WriteLine('\u0420');    // Р
Console.WriteLine('\u0421');    // С
```

Строковые литералы

Строковые литералы представляют строки. Строки заключаются в двойные кавычки:

```
Console.WriteLine("hello");
Console.WriteLine("фыва");
Console.WriteLine("hello word");
```

Если внутри строки необходимо вывести двойную кавычку, то такая внутренняя кавычка предваряется обратным слешем:

```
Console.WriteLine("Компания \"Рога и копыта\"");
```

Также в строках можно использовать управляющие последовательности. Например, последовательность '\n' осуществляет перевод на новую строку:

```
Console.WriteLine("Привет \nмир");
```

При выводе на консоль слово "мир" будет перенесено на новую строку:

```
Привет
мир
```

null

null представляет ссылку, которая не указывает ни на какой объект. То есть по сути отсутствие значения.

Практическая работа № 5

Тема: Платформа .NET Framework: назначение, составные элементы, тонкости компиляции (2)

Цель: изучить особенности платформы .NET Framework

Теоретический материал

На сегодняшний момент язык программирования C# один из самых мощных, быстро развивающихся и востребованных языков в ИТ-отрасли. В настоящий момент на нем пишутся самые различные приложения: от небольших десктопных программок до крупных веб-порталов и веб-сервисов, обслуживающих ежедневно миллионы пользователей.

C# уже не молодой язык и кFк и вся платформа .NET уже прошел большой путь. Первая версия языка вышла вместе с релизом Microsoft Visual Studio .NET в феврале 2002 года. Текущей версией языка является версия C# 9.0, которая вышла 10 ноября 2020 года вместе с релизом .NET 5.

C# является языком с Си-подобным синтаксисом и близок в этом отношении к C++ и Java. Поэтому, если вы знакомы с одним из этих языков, то овладеть C# будет легче.

C# является объектно-ориентированным и в этом плане много перенял у Java и C++. Например, C# поддерживает полиморфизм, наследование, перегрузку операторов, статическую типизацию. Объектно-ориентированный подход позволяет решить задачи по по-

строению крупных, но в тоже время гибких, масштабируемых и расширяемых приложений. И C# продолжает активно развиваться, и с каждой новой версией появляется все больше интересных функциональностей, как, например, лямбды, динамическое связывание, асинхронные методы и т.д.

Роль платформы .NET

Когда говорят C#, нередко имеют в виду технологии платформы .NET (Windows Forms, WPF, ASP.NET, Xamarin). И, наоборот, когда говорят .NET, нередко имеют в виду C#. Однако, хотя эти понятия связаны, отождествлять их неверно. Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире.

Как-то Билл Гейтс сказал, что платформа .NET - это лучшее, что создала компания Microsoft. Возможно, он был прав. Фреймворк .NET представляет мощную платформу для создания приложений. Можно выделить следующие ее основные черты:

- **Поддержка нескольких языков**. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), благодаря чему .NET поддерживает несколько языков: наряду с C# это также VB.NET, C++, F#, а также различные диалекты других языков, привязанные к .NET, например, Delphi.NET. При компиляции код на любом из этих языков компилируется в сборку на общем языке CIL (Common Intermediate Language) - своего рода ассемблер платформы .NET. Поэтому при определенных условиях мы можем сделать отдельные модули одного приложения на отдельных языках.
- **Кроссплатформенность**. .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент - .NET 5 поддерживается на большинстве современных ОС Windows, MacOS, Linux. Используя различные технологии на платформе .NET, можно разрабатывать приложения на языке C# для самых разных платформ - Windows, MacOS, Linux, Android, iOS, Tizen.
- **Мощная библиотека классов**. .NET представляет единую для всех поддерживаемых языков библиотеку классов. И какое бы приложение мы не собирались писать на C# - текстовый редактор, чат или сложный веб-сайт - так или иначе мы задействуем библиотеку классов .NET.
- **Разнообразие технологий**. Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET и Entity Framework Core. Для построения графических приложений с богатым насыщенным интерфейсом - технология WPF и UWP, для создания более простых графических приложений - Windows Forms. Для разработки мобильных приложений - Xamarin. Для создания веб-сайтов и веб-приложений - ASP.NET и т.д.

  К этому стоит добавить активной развивающийся и набирающий популяность Blazor - фреймворк, который работает поверх .NET и который позволяет создавать веб-приложения как на стороне сервера, так и на стороне клиента. А в будущем будет поддерживать создание мобильных приложений и, возможно, десктоп-приложений.

- **Производительность**. Согласно ряду тестов веб-приложения на .NET 5 в ряде категорий сильно опережают веб-приложения, построенные с помощью других тех-

нологий. Приложения на .NET 5 в принципе отличаются высокой производительностью.

Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от C++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память.

.NET Framework и .NET Core / .NET 5

Стоит отметить, что .NET долгое время развивался премущественно как платформа для Windows под названием .NET Framework. В 2019 вышла последняя версия этой платформы - .NET Framework 4.8. Она больше не развивается

С 2014 Microsoft стал развивать альтернативную платформу - .NET Core, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework и добавить новую функциональность. Затем Microsoft последовательно выпустил ряд версий этой платформы: .NET Core 1, .NET Core 2, .NET Core 3. Логическим развитием .NET Core 3.0 стала расматриваемая в этом руководстве платформа .NET 5. Поэтому следует различать .NET Framework, который предназначен преимущественно для Windows, и кросплатформенный .NET 5. В данном руководстве речь будет идти о C# в связке с .NET 5, поскольку это актуальная платформа.

Также стоит упомянуть о платформе Mono, которая была создана еще в 2004 году и представляла опенсорс-версию платформы .NET Framework для Linux и MacOS. Используя Mono, можно было создавать кросплатформенные приложения на C#. Mono по-прежнему используется. Например, Xamarin - технология для создания мобильных приложений для Android и iOS с помощью языка C# использует Mono. Тем не менее в будущем при выходе новой версии - .NET 6 в следующем году планируется, что .NET 6 вберет в себя платформу Mono.

Управляемый и неуправляемый код

Нередко приложение, созданное на C#, называют **управляемым кодом** (managed code). Что это значит? А это значит, что данное приложение создано на основе платформы .NET и поэтому управляется общеязыковой средой CLR, которая загружает приложение и при необходимости очищает память. Но есть также приложения, например, созданные на языке C++, которые компилируются не в общий язык CIL, как C# или F#, а в обычный машинный код. В этом случае .NET не управляет приложением.

В то же время платформа .NET предоставляет возможности для взаимодействия с неуправляемым кодом..

JIT-компиляция

Как выше писалось, код на C# компилируется в приложения или сборки с расширениями exe или dll на языке CIL. Далее при запуске на выполнение подобного приложения происходит JIT-компиляция (Just-In-Time) в машинный код, который затем выполняется. При этом, поскольку наше приложение может быть большим и содержать кучу инструкций, в текущий момент времени будет компилироваться лишь та часть приложения, к которой непосредственно идет обращение. Если мы обратимся к другой части кода, то она будет скомпилирована из CIL в машинный код. При том уже скомпилированная часть приложе-

ния сохраняется до завершения работы программы. В итоге это повышает производительность.

Практическая работа № 6

Тема: Обзор интегрированной среды разработки Visual Studio .NET. Создание, сохранение и закрытие проектов и решений. Структура программы. Точка входа. Базовый синтаксис C#. Пространство имен. (2)
Цель: познакомить студентов со средой программирования языка c#.

Практическая работа № 7

Тема: Консольные и линейные приложения (2)
стр. 28-57 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №8

Тема: Создание проектов с использованием условного оператора if/else (4)
стр. 57-70 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №9

Тема: Создание проектов с использованием оператора выбора switch (2)
стр. 70-83 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №10

Тема: Создание проектов циклической структуры (for) (4)
стр. 83-97 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №11

Тема: Цикл с предусловием (while), цикл с постусловием (do) и цикл с параметром (for). Правила записи, отличия в применении (2)
стр. 98-104 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №12

Тема: Создание проектов циклической структуры (while, do)

стр. 105-112 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №13

Тема: Обработка одномерных массивов: сортировка и поиск элементов. Цикл foreach (2)
стр. 112-117 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №14

Тема: Создание проектов для работы с одномерными массивами (4)
стр. 117-119 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №15

Тема: Двумерные массивы: объявление, ввод и вывод. Работа с двумерными массивами по строкам и по столбцам (2)
стр. 120-123 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №16

Тема: Создание проектов для работы с двумерными массивами (2)
стр. 124-127 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №17

Тема: Создание методов, возвращающих значения. Способы размещения методов. Конструкторы. (2)
стр. 128-133 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №18

Тема: Создание простейших методов (2)
стр. 133-135 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №19

Тема: Создание перегруженных методов (2)

Теоретический материал

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? **Сигнатура** складывается из следующих аспектов:

- Имя метода
- Количество параметров
- Типы параметров
- Порядок параметров
- Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

```
public int Sum(int x, int y)
{
   return x + y;
}
```

У данного метода сигнатура будет выглядеть так: Sum(int, int)

И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

- Количеству параметров
- Типу параметров
- Порядку параметров
- Модификаторам параметров

Например, пусть у нас есть следующий класс:

```
class Calculator
{
   public void Add(int a, int b)
   {
      int result = a + b;
      Console.WriteLine($"Result is {result}");
   }
   public void Add(int a, int b, int c)
   {
      int result = a + b + c;
      Console.WriteLine($"Result is {result}");
   }
   public int Add(int a, int b, int c, int d)
   {
      int result = a + b + c + d;
      Console.WriteLine($"Result is {result}");
      return result;
   }
```

```
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

То есть мы можем представить сигнатуры данных методов следующим образом:

```
Add(int, int)
Add(int, int, int)
Add(int, int, int, int)
Add(double, double)
```

После определения перегруженных версий мы можем использовать их в программе:

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calc = new Calculator();
        calc.Add(1, 2); // 3
        calc.Add(1, 2, 3); // 6
        calc.Add(1, 2, 3, 4); // 10
        calc.Add(1.4, 2.5); // 3.9

        Console.ReadKey();
    }
}
```

Консольный вывод:

```
Result is 3
Result is 6
Result is 10
Result is 3.9
```

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

```
1	void Increment(ref int val)
2	{
3	    val++;
4	    Console.WriteLine(val);
5	}
```

```
6
7      void Increment(int val)
8      {
9         val++;
10        Console.WriteLine(val);
11     }
```

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:

```
1      int Sum(int x, int y)
2      {
3         return x + y;
4      }
5      int Sum(int number1, int number2)
6      {
7         return x + y;
8      }
9      void Sum(int x, int y)
10     {
11        Console.WriteLine(x + y);
12     }
```

Сигнатура у всех этих методов будут совпадать:

```
1    Sum(int, int)
```

Поэтому данный набор методов не представляет корректные перегрузки метода Sum и работать не будет.

Ответить на вопросы:
Вопрос 1

Каким образом можно перегрузить метод?

- Определить версию метода с другим количеством параметров
- Определить версию метода с другим типами параметров
- Определить версию метода, где порядок параметров будет иным
- Изменить модификаторы параметров

Вопрос 2

Корректна ли следующая перегрузка методов? Если да, то почему? Если нет, то почему?

```
static void IncrementVal(ref int val)
{
   val++;
   Console.WriteLine(val);
}
```

```
static void IncrementVal(int val)
{
   val++;
   Console.WriteLine(val);
}
```

Практическая работа №20

Тема: Наследование и полиморфизм. Иерархия классов: понятие, преимущества. Синтаксис наследования. (2)

стр. 155-161 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №21

Тема: Скрытие и перекрытие методов. Ключевые слова virtual, override. (2)

Теоретический материал

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором **virtual**. Такие методы и свойства называют виртуальными.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

Например, рассмотрим следующие классы:

```
class Person
{
   public string Name { get; set; }
   public Person(string name)
   {
      Name = name;
   }
   public virtual void Display()
   {
      Console.WriteLine(Name);
   }
}
class Employee : Person
{
   public string Company { get; set; }
   public Employee(string name, string company) : base(name)
   {
      Company = company;
   }
```

```
}
```

Здесь класс Person представляет человека. Класс Employee наследуется от Person и представляет сотруднника предприятия. Этот класс кроме унаследованного свойства Name имеет еще одно свойство - Company.

Чтобы сделать метод Display доступным для переопределения, этот метод определен с модификатором **virtual**. Поэтому мы можем переопределить этот метод, но можем и не переопределять. Допустим, нас устраивает реализация метода из базового класса. В этом случае объекты Employee будут использовать реализацию метода Display из класса Person:

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill");
    p1.Display(); // вызов метода Display из класса Person

    Employee p2 = new Employee("Tom", "Microsoft");
    p2.Display(); // вызов метода Display из класса Person

    Console.ReadKey();
}
```

Консольный вывод:

```
Bill
Tom
```

Но также можем переопределить виртуальный метод. Для этого в классе-наследнике определяется метод с модификатором **override**, который имеет то же самое имя и набор параметров:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }

    public override void Display()
    {
        Console.WriteLine($"{Name} работает в {Company}");
    }
}
```

Возьмем те же самые объекты:

```
static void Main(string[] args)
{
    Person p1 = new Person("Bill");
    p1.Display(); // вызов метода Display из класса Person
```

```
    Employee p2 = new Employee("Tom", "Microsoft");
    p2.Display(); // вызов метода Display из класса Employee

    Console.ReadKey();
}
```

Консольный вывод:

```
Bill
Tom работает в Microsoft
```

Виртуальные методы базового класса определяют интерфейс всей иерархии, то есть в любом производном классе, который не является прямым наследником от базового класса, можно переопределить виртуальные методы. Например, мы можем определить класс Manager, который будет производным от Employee, и в нем также переопределить метод Display.

При переопределении виртуальных методов следует учитывать ряд ограничений:

- Виртуальный и переопределенный методы должны иметь один и тот же модификатор доступа. То есть если виртуальный метод определен с помощью модификатора public, то и переопредленный метод также должен иметь модификатор public.
- Нельзя переопределить или объявить виртуальным статический метод.

Переопределение свойств

Также как и методы, можно переопределять свойства:

```
class Credit
{
    public virtual decimal Sum { get; set; }
}
class LongCredit : Credit
{
    private decimal sum;
    public override decimal Sum
    {
        get
        {
            return sum;
        }
        set
        {
            if(value > 1000)
            {
                sum = value;
            }
        }
    }
}
class Program
{
    static void Main(string[] args)
```

```
    {
        LongCredit credit = new LongCredit { Sum = 6000 };
        credit.Sum = 490;
        Console.WriteLine(credit.Sum);
        Console.ReadKey();
    }
}
```

Ключевое слово base

Кроме конструкторов, мы можем обратиться с помощью ключевого слова **base** к другим членам базового класса. В нашем случае вызов base.Display(); будет обращением к методу Display() в классе Person:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
            :base(name)
    {
        Company = company;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine($"работает в {Company}");
    }
}
```

Запрет переопределения методов

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором **sealed**:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
                : base(name)
    {
        Company = company;
    }

    public override sealed void Display()
    {
        Console.WriteLine($"{Name} работает в {Company}");
    }
}
```

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

Ответить на вопросы
Вопрос 1

Какое ключевое слово применяется для переопределения виртуальных методов и свойств:

- overridable
- override
- overriding
- overriden

Вопрос 2

Что будет выведено на консоль в результате выполнения следующей программы и почему:

```
class Person
{
    public string Name { get; set; }
    public virtual void Display()
    {
        Console.WriteLine($"Person {Name}");
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public override void Display()
    {
        Console.WriteLine($"Employee {Name}");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person person = new Employee { Name = "Sam", Company = "Microsoft" };
        person.Display();

        Console.ReadKey();
    }
}
```

Вопрос 3

Что будет выведено на консоль в результате выполнения следующей программы и почему:

```
class Person
```

```
{
   public string Name { get; set; }
   public virtual void Display()
   {
      Console.WriteLine($"Person {Name}");
   }
}
class Employee : Person
{
   public string Company { get; set; }
   public override void Display()
   {
      Console.WriteLine($"Employee {Name}");
   }
}
class Manager : Employee
{
   public override void Display()
   {
      Console.WriteLine($"Manager {Name}");
   }
}
class Program
{
   static void Main(string[] args)
   {
      Person person = new Manager { Name = "Bob", Company = "Microsoft" };
      Employee employee = (Employee)person;
      employee.Display();

      Console.ReadKey();
   }
}
```

Вопрос 4

Что будет выведено на консоль в результате выполнения следующей программы и почему:

```
class Auto
{
   protected internal virtual void Move()
   {
      Console.WriteLine("Auto is moving");
   }
}
class Track : Auto
{
   public override void Move()
   {
      Console.WriteLine("Track is moving");
   }
}
class Program
```

```
{
   static void Main(string[] args)
   {
      Track track = new Track();
      Auto auto = track;
      auto.Move();

      Console.ReadKey();
   }
}
```

Вопрос 5

Для чего нужен модификатор **sealed**?

Вопрос 6

Что будет выведено на консоль в результате выполнения следующей программы и почему?

```
1      class Auto
2      {
3         public sealed void Display()
4         {
5            Console.WriteLine("Auto");
6         }
7      }
8      class Truck : Auto
9      {
10        public void DisplayInfo()
11        {
12           base.Display();
13        }
14     }
15     class Program
16     {
17        static void Main(string[] args)
18        {
19           Truck truck = new Truck();
20           truck.DisplayInfo();
21           Console.ReadKey();
22        }
23     }
```

Практическая работа №22

Тема: Инкапсуляция как управление доступом к данным. Свойства класса: понятие, виды, правила записи. (2)

стр. 150-155 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №23

Тема: Вызов методов базового класса («родителя»): ключевое слово base. Тонкости использования конструкторов в иерархически связанных между собой классах. (2)

Теоретический материал

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

```
class Person
{
   private string _name;

   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }
   public void Display()
   {
      Console.WriteLine(Name);
   }
}
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
class Employee : Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

```
static void Main(string[] args)
{
   Person p = new Person { Name = "Tom"};
   p.Display();
   p = new Employee { Name = "Sam" };
   p.Display();
   Console.Read();
}
```

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

  Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.

- Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

  ```
  sealed class Admin
  {
  }
  ```

- Нельзя унаследовать класс от статического класса.

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
   public void Display()
   {
      Console.WriteLine(_name);
   }
}
```

Этот код не сработает и выдаст ошибку, так как переменная _name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
class Employee : Person
{
   public void Display()
   {
```

```
        Console.WriteLine(Name);
    }
}
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

Ключевое слово base

Теперь добавим в наши классы конструкторы:

```
class Person
{
    public string Name { get;  set; }

    public Person(string name)
    {
        Name = name;
    }

    public void Display()
    {
        Console.WriteLine(Name);
    }
}

class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
        : base(name)
    {
        Company = company;
    }
}
```

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

```
static void Main(string[] args)
{
    Person p = new Person("Bill");
```

```
    p.Display();
    Employee emp = new Employee ("Tom", "Microsoft");
    emp.Display();
    Console.Read();
}
```

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base. Например, из класса Employee уберем определение конструктора:

```
class Employee : Person
{
    public string Company { get; set; }
}
```

В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

```
public Employee(string name, string company)
{
    Name = name;
    Company = company;
}
```

То есть в классе Employee через ключевое слово **base** надо явным образом вызвать конструктор класса Person:

```
public Employee(string name, string company)
        : base(name)
{
    Company = company;
}
```

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
class Person
{
    // остальной код класса
    // конструктор по умолчанию
    public Person()
    {
        FirstName = "Tom";
        Console.WriteLine("Вызов конструктора без параметров");
    }
}
```

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

```
public Employee(string company)
{
   Company = company;
}
```

Фактически был бы эквивалентен следующему конструктору:

```
public Employee(string company)
   :base()
{
   Company = company;
}
```

Порядок вызова конструкторов

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

```
class Person
{
   string name;
   int age;

   public Person(string name)
   {
      this.name = name;
      Console.WriteLine("Person(string name)");
   }
   public Person(string name, int age) : this(name)
   {
      this.age = age;
      Console.WriteLine("Person(string name, int age)");
   }
}
class Employee : Person
{
   string company;

   public Employee(string name, int age, string company) : base(name, age)
   {
      this.company = company;
      Console.WriteLine("Employee(string name, int age, string company)");
   }
}
```

При создании объекта Employee:

```
Employee tom = new Employee("Tom", 22, "Microsoft");
```

Мы получим следующий консольный вывод:

```
Person(string name)
Person(string name, int age)
Employee(string name, int age, string company)
```

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)
6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

Вопрос 1

Почему следующая программа не компилируется:

```
using System;

namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {
         Person tom = new Employee();
         Console.ReadKey();
      }
   }

   internal class Person
   {

   }
   public class Employee : Person
   {

   }
}
```
Вопрос 2

Даны следующие классы:

```
class Person
{
   string name;
   int age;

   public Person()
   {
   }
   public Person(string name) : this(name, 18)
   {
   }
   public Person(string name, int age)
   {
      this.name = name;
      this.age = age;
   }
}
class Employee : Person
{
   string company;

   public Employee()
   {
   }
   public Employee(string name, int age, string company): base(name, age)
   {
      this.company = company;
   }
   public Employee(string name, string company) : base(name)
   {
         this.company = company;
   }
}
```

Допустим, мы создаем объект класса Employee следующим образом:

```
Employee tom = new Employee("Tom", "Microsoft");
```

Какие конструкторы и в каком порядке в данном случае будет выполняться?

Ответ

Порядок выполнения конструкторов:

1. System.Object.Object()
2. Person(string name, int age)
3. Person(string name)
4. Employee(string name, string company)

Вопрос 3

Как запретить наследование от класса?

Вопрос 4

Что выведет на консоль следующая программа и почему?

```
class Auto  // легковой автомобиль
{
   public int Seats { get; set; } // количество сидений
   public Auto(int seats)
   {
      Seats = seats;
   }
}
class Truck : Auto      // грузовой автомобиль
{
   public decimal Capacity { get; set; } // грузоподъемность
   public Truck(int seats, decimal capacity)
   {
      Seats = seats;
      Capacity = capacity;
   }
}
class Program
{
   static void Main(string[] args)
   {
      Truck truck = new Truck(2, 1.1m);
      Console.WriteLine($"Грузовик с грузоподъемностью {truck.Capacity} тонн");
      Console.ReadKey();
   }
}
```

Вопрос 5

Что выведет на консоль следующая программа и почему?

```
class Auto  // легковой автомобиль
{
   public int Seats { get; set; } // количество сидений
   public Auto()
   {
      Console.WriteLine("Auto has been created");
   }
   public Auto(int seats)
   {
      Seats = seats;
   }
}
class Truck : Auto      // грузовой автомобиль
{
   public decimal Capacity { get; set; } // грузоподъемность
   public Truck(decimal capacity)
   {
      Seats = 2;
      Capacity = capacity;
```

```
        Console.WriteLine("Truck has been created");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Truck truck = new Truck(1.1m);
        Console.WriteLine($"Truck with capacity {truck.Capacity}");
        Console.ReadKey();
    }
}
```

Вопрос 6

Что выведет на консоль следующая программа и почему?

```
class Person
{
    public string Name { get; set; } = "Ben";

    public Person(string name)
    {
        Name = "Tim";
    }
}

class Employee : Person
{
    public string Company { get; set; }

    public Employee(string name, string company)
        : base("Bob")
    {
        Company = company;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Employee emp = new Employee("Tom", "Microsoft") { Name = "Sam" };

        Console.WriteLine(emp.Name); // Ben Tim Bob Tom Sam
        Console.ReadKey();
    }
}
```

Практическая работа №24

Тема: Наследование в интерфейсах. Сходства и различия интерфейсов, абстрактных классов и обычных классов. (2)

Теоретический материал

Один из принципов проектирования гласит, что при создании системы классов надо программировать на уровне интерфейсов, а не их конкретных реализаций. Под интерфейсами в данном случае понимаются не только типы C#, определенные с помощью ключевого слова interface, а определение функционала без его конкретной реализации. То есть под данное определение попадают как собственно интерфейсы, так и абстрактные классы, которые могут иметь абстрактные методы без конкретной реализации.

В этом плане у абстрактных классов и интерфейсов много общего. Нередко при проектировании программ в паттернах мы можем заменять абстрактные классы на интерфейсы и наоборот. Однако все же они имеют некоторые отличия.

Когда следует использовать абстрактные классы:

- Если надо определить общий функционал для родственных объектов
- Если мы проектируем довольно большую функциональную единицу, которая содержит много базового функционал
- Если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию. При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.

  Если же нам вдруг надо будет поменять название или параметры метода интерфейса, то придется вносить изменения и также во всех классы, которые данный интерфейс реализуют.

Когда следует использовать интерфейсы:

- Если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если мы проектируем небольшой функциональный тип

Ключевыми здесь являются первые пункты, которые можно свести к следующему принципу: если классы относятся к единой системе классификации, то выбирается абстрактный класс. Иначе выбирается интерфейс. Посмотрим на примере.

Допустим, у нас есть система транспортных средств: легковой автомобиль, автобус, трамвай, поезд и т.д. Поскольку данные объекты являются родственными, мы можем выделить у них общие признаки, то в данном случае можно использовать абстрактные классы:

```
public abstract class Vehicle
{
   public abstract void Move();
}

public class Car : Vehicle
{
   public override void Move()
   {
      Console.WriteLine("Машина едет");
   }
}
```

```
public class Bus : Vehicle
{
   public override void Move()
   {
      Console.WriteLine("Автобус едет");
   }
}

public class Tram : Vehicle
{
   public override void Move()
   {
      Console.WriteLine("Трамвай едет");
   }
}
```

Абстрактный класс Vehicle определяет абстрактный метод перемещения Move(), а классы-наследники его реализуют.

Но, предположим, что наша система транспорта не ограничивается вышеперечисленными транспортными средствами. Например, мы можем добавить самолеты, лодки. Возможно, также мы добавим лошадь - животное, которое может также выполнять роль транспортного средства. Также можно добавить дирижабль. Вобщем получается довольно широкий круг объектов, которые связаны только тем, что являются транспортным средством и должны реализовать некоторый метод Move(), выполняющий перемещение.

Так как объекты малосвязанные между собой, то для определения общего для всех них функционала лучше определить интерфейс. Тем более некоторые из этих объектов могут существовать в рамках параллельных систем классификаций. Например, лошадь может быть классом в структуре системы классов животного мира.

Возможная реализация интерфейса могла бы выглядеть следующим образом:

```
public interface IMovable
{
   void Move();
}

public abstract class Vehicle
{ }

public class Car : Vehicle, IMovable
{
   public void Move()
   {
      Console.WriteLine("Машина едет");
   }
}

public class Bus : Vehicle, IMovable
{
   public void Move()
```

```
    {
        Console.WriteLine("Автобус едет");
    }
}

public class Hourse : IMovable
{
    public void Move()
    {
        Console.WriteLine("Лошадь скачет");
    }
}

public class Aircraft : IMovable
{
    public void Move()
    {
        Console.WriteLine("Самолет летит");
    }
}
```

Теперь метод Move() определяется в интерфейсе IMovable, а конкретные классы его реализуют.

Говоря об использовании абстрактных классов и интерфейсов можно привести еще такую аналогию, как состояние и действие. Как правило, абстрактные классы фокусируются на общем состоянии классов-наследников. В то время как интерфейсы строятся вокруг какого-либо общего действия.

Например, солнце, костер, батарея отопления и электрический нагреватель выполняют функцию нагревания или излучения тепла. По большому счету выделение тепла - это единственный общий между ними признак. Можно ли для них создать общий абстрактный класс? Можно, но это не будет оптимальным решением, тем более у нас могут быть какие-то родственные сущности, которые мы, возможно, тоже захотим использовать. Поэтому для каждой вышеперечисленной сущности мы можем определить свою систему классификации. Например, в одной системе классов, которые наследуются от общего астрактного класса, были бы звезды, в том числе и солнце, планеты, астероиды и так далее - то есть все те объекты, которые могут иметь какое-то общее с солнцем состояние. В рамках другой системы классов мы могли бы определить электрические приборы, в том числе электронагреатель. И так, для каждой разноплановой сущности можно было бы составить свою систему классов, исходяющую от определенного абстрактного класса. А для общего действия определить интерфейс, например, IHeatable, в котором бы был метод Heat, и этот интерфейс реализовать во всех необходимых классах.

Таким образом, если разноплановые классы обладают каким-то общим действием, то это действие лучше выносить в интерфейс. А для одноплановых классов, которые имеют общее состояние, лучше определять абстрактный класс.

Практическая работа №25

Тема: Стандартные интерфейсы .NET: IComparable, IClonable, IEnumerable. Примеры реализации. (2)

Теоретический материал

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:

```
int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };
Array.Sort(numbers);
foreach (int n in numbers)
   Console.WriteLine(n);
```

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс **IComparable**. Он имеет всего один метод:

```
public interface IComparable
{
   int CompareTo(object o);
}
```

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

- Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
- Равен нулю. Значит, оба объекта равны
- Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

```
class Person : IComparable
{
   public string Name { get; set; }
   public int Age { get; set; }
   public int CompareTo(object o)
   {
      Person p = o as Person;
      if (p != null)
         return this.Name.CompareTo(p.Name);
      else
         throw new Exception("Невозможно сравнить два объекта");
   }
}
```

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people);

foreach(Person p in people)
{
   Console.WriteLine($"{p.Name} - {p.Age}");
}
```

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

```
class Person : IComparable<Person>
{
   public string Name { get; set; }
   public int Age { get; set; }
   public int CompareTo(Person p)
   {
      return this.Name.CompareTo(p.Name);
   }
}
```

Применение компаратора

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

```
interface IComparer
{
   int Compare(object o1, object o2);
}
```

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

```
class PeopleComparer : IComparer<Person>
{
   public int Compare(Person p1, Person p2)
   {
      if (p1.Name.Length > p2.Name.Length)
         return 1;
      else if (p1.Name.Length < p2.Name.Length)
         return -1;
      else
         return 0;
```

```
    }
}
```

В данном случае используется обобщенная версия интерфейса IComparer, чтобы не делать излишних преобразований типов. Применение компаратора:

```
Person p1 = new Person { Name = "Bill", Age = 34 };
Person p2 = new Person { Name = "Tom", Age = 23 };
Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people, new PeopleComparer());

foreach(Person p in people)
{
    Console.WriteLine($"{p.Name} - {p.Age}");
}
```

Объект компаратора указывается в качестве второго параметра метода Array.Sort(). При этом не важно, реализует ли класс Person интерфейс IComparable или нет. Правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты Person, у которых имена меньше, а в конце - у которых имена длиннее:

```
Tom - 23
Bill - 34
Alice - 21
```

Практическая работа №26

Тема: Понятие потока. Механизм буферизации. Классы библиотеки .NET для работы с потоками. Виды доступа к файлам. Объект FileStream. Классы StreamWriter и SreamReader. (2)

стр. 161-169 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №27

Тема: Общая форма определения класса. Модификаторы доступа к элементам класса: public, private, protected, internal. Примеры создания классов. (2)

стр. 135-142 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №28

Тема: Создание простейших классов (4)

стр. 143--142 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №29

Тема: Создание классов с использованием свойств (2)

стр. 144 - 147 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №30

Тема: Создание классов, иерархически связанных между собой (2)

стр. 160 - 161 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №31

Тема: Решение задач с использованием классов (2)

1. Создайте структуру с именем student, содержащую поля: фамилия и инициалы, номер группы, успеваемость (массив из пяти элементов). Создать массив из десяти элементов такого типа, упорядочить записи по возрастанию среднего балла. Добавить возможность вывода фамилий и номеров групп студентов, имеющих оценки, равные только 4 или 5.

2. Создайте структуру с именем train, содержащую поля: название пункта назначения, номер поезда, время отправления. Ввести данные в массив из пяти элементов типа train, упорядочить элементы по номерам поездов. Добавить возможность вывода информации о поезде, номер которого введен пользователем. Добавить возможность сортировки массив по пункту назначения, причем поезда с одинаковыми пунктами назначения должны быть упорядочены по времени отправления.

3. Создать класс с двумя переменными. Добавить функцию вывода на экран и функцию изменения этих переменных. Добавить функцию, которая находит сумму значений этих переменных, и функцию которая находит наибольшее значение из этих двух переменных.

4. Описать класс, реализующий десятичный счетчик, который может увеличивать или уменьшать свое значение на единицу в заданном диапазоне. Предусмотреть инициализацию счетчика значениями по умолчанию и произвольными значениями. Счетчик имеет два метода: увеличения и уменьшения, — и свойство, позволяющее получить его текущее состояние. Написать программу, демонстрирующую все возможности класса.

5. Создать класс с двумя переменными. Добавить конструктор с входными параметрами. Добавить конструктор, инициализирующий члены класса по умолчанию. Добавить деструктор, выводящий на экран сообщение об удалении объекта.

Практическая работа №32

Тема: Создание MDI-приложений и меню (2)

стр. 169 - 177 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №33

Тема: Реализация стандартных интерфейсов NET: IComparable, IClonable, IEnumerable. (2)

Практическая работа №34

Тема: Создание проектов с использованием структур и перечислений(4)

Теоретический материал

Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

Например, определим структуру, которая представляет человека:

```
struct User
{
   public string name;
   public int age;

   public void DisplayInfo()
   {
      Console.WriteLine($"Name: {name}  Age: {age}");
   }
}
```

Как и классы, структуры могут хранить состояние в виде переменных и определять поведение в виде методов. Так, в данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод DisplayInfo для вывода информации о человеке.

Используем эту структуру в программе:

```
using System;

namespace HelloApp
{
   struct User
   {
      public string name;
      public int age;

      public void DisplayInfo()
      {
         Console.WriteLine($"Name: {name}  Age: {age}");
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         User tom;
         tom.name = "Tom";
         tom.age = 34;
         tom.DisplayInfo();
```

```
            Console.ReadKey();
        }
    }
}
```

В данном случае создается объект tom. У него устанавливаются значения глобальных переменных, и затем выводится информация о нем.

Конструкторы структуры

Как и класс, структура может определять конструкторы. Но в отличие от класса нам не обязательно вызывать конструктор для создания объекта структуры:

```
User tom;
```

Однако если мы таким образом создаем объект структуры, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры. То есть, например, в следующем случае мы получим ошибку, так как обращение к полям и методам происходит до присвоения им начальных значений:

```
User tom;
int x = tom.age;    // Ошибка
tom.DisplayInfo();  // Ошибка
```

Также мы можем использовать для создания структуры конструктор без параметров, который есть в структуре по умолчанию и при вызове которого полям структуры будет присвоено значение по умолчанию (например, для числовых типов это число 0):

```
User tom = new User();
tom.DisplayInfo();  // Name:   Age: 0
```

Обратите внимание, что при использовании конструктора по умолчанию нам не надо явным образом иницилизировать поля структуры.

Также мы можем определить свои конструкторы. Например, изменим структуру User:

```
using System;

namespace HelloApp
{
    struct User
    {
        public string name;
        public int age;
        public User(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name}  Age: {age}");
```

```
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            User tom = new User("Tom", 34);
            tom.DisplayInfo();

            User bob = new User();
            bob.DisplayInfo();

            Console.ReadKey();
        }
    }
}
```

Важно учитывать, что если мы определяем конструктор в структуре, то он должен инициализировать все поля структуры, как в данном случае устанавливаются значения для переменных name и age.

Также, как и для класса, можно использовать инициализатор для создания структуры:

```
User person = new User { name = "Sam", age = 31 };
```

Но в отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении, например, следующим образом:

```
struct User
{
    public string name = "Sam";     // ! Ошибка
    public int age = 23;            // ! Ошибка
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}  Age: {age}");
    }
}
```

Практическая работа №35

Тема: Создание проектов с использованием коллекций. Работа с ArrayList (2)

Теоретический материал

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако что если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен System.Collections (простые необобщенные классы коллекций), System.Collections.Generic (обобщенные или типизированные классы коллекций) и System.Collections.Specialized (специальные классы коллекций). Также для обеспечения параллельного выполнения задач и многопоточного доступа применяются классы коллекций из пространства имен System.Collections.Concurrent

ArrayList

Итак, класс **ArrayList** представляет коллекцию объектов. И если надо сохранить вместе разнотипные объекты - строки, числа и т.д., то данный класс как раз для этого подходит.

Основные методы класса:

- int Add(object value): добавляет в список объект value
- void AddRange(ICollection col): добавляет в список объекты коллекции col, которая представляет интерфейс ICollection - интерфейс, реализуемый коллекциями.
- void Clear(): удаляет из списка все элементы
- bool Contains(object value): проверяет, содержится ли в списке объект value. Если содержится, возвращает true, иначе возвращает false
- void CopyTo(Array array): копирует текущий список в массив array.
- ArrayList GetRange(int index, int count): возвращает новый список ArrayList, который содержит count элементов текущего списка, начиная с индекса index
- int IndexOf(object value): возвращает индекс элемента value
- void Insert(int index, object value): вставляет в список по индексу index объект value
- void InsertRange(int index, ICollection col): вставляет в список начиная с индекса index коллекцию ICollection
- int LastIndexOf(object value): возвращает индекс последнего вхождения в списке объекта value
- void Remove(object value): удаляет из списка объект value
- void RemoveAt(int index): удаляет из списка элемент по индексу index
- void RemoveRange(int index, int count): удаляет из списка count элементов, начиная с индекса index
- void Reverse(): переворачивает список
- void SetRange(int index, ICollection col): копирует в список элементы коллекции col, начиная с индекса index
- void Sort(): сортирует коллекцию

Кроме того, с помощью свойства Count можно получить количество элементов в списке.

Посмотрим применение класса на примере.

```
using System;
using System.Collections;

namespace Collections
{
   class Program
   {
      static void Main(string[] args)
      {
         ArrayList list = new ArrayList();
         list.Add(2.3); // заносим в список объект типа double
```

```
            list.Add(55); // заносим в список объект типа int
            list.AddRange(new string[] { "Hello", "world" }); // заносим в список строковый мас-
сив

            // перебор значений
            foreach (object o in list)
            {
                Console.WriteLine(o);
            }

            // удаляем первый элемент
            list.RemoveAt(0);
            // переворачиваем список
            list.Reverse();
            // получение элемента по индексу
            Console.WriteLine(list[0]);
            // перебор значений
            for (int i = 0; i < list.Count; i++)
            {
                Console.WriteLine(list[i]);
            }

            Console.ReadLine();
        }
    }
}
```

Во-первых, так как класс ArrayList находится в пространстве имен System.Collections, то подключаем его (using System.Collections;).

Вначале создаем объект коллекции через конструктор как объект любого другого класса: ArrayList list = new ArrayList();. При необходимости мы могли бы так же, как и с массивами, выполнить начальную инициализацию коллекции, например, ArrayList list = new ArrayList(){1, 2, 5, "string", 7.7};

Далее последовательно добавляем разные значения. Данный класс коллекции, как и большинство других коллекций, имеет два способа добавления: одиночного объекта через метод **Add** и набора объектов, например, массива или другой коллекции через метод **AddRange**

Через цикл foreach мы можем пройтись по всем объектам списка. И поскольку данная коллекция хранит разнородные объекты, а не только числа или строки, то в качестве типа перебираемых объектов выбран тип object: foreach (object o in list)

Многие коллекции, в том числе и ArrayList, реализуют удаление с помощью методов Remove/RemoveAt. В данном случае мы удаляем первый элемент, передавая в метод RemoveAt индекс удаляемого элемента.

В завершении мы опять же выводим элементы коллекции на экран только уже через цикл for. В данном случае с перебором коллекций дело обстоит также, как и с массивами. А число элементов коллекции мы можем получить через свойство Count

С помощью индексатора мы можем получить по индексу элемент коллекции так же, как и в массивах: object firstObj = list[0];

Практическая работа №36

Тема: Создание проектов с использованием текстовых файлов (2)
стр. 168 - 169 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №37

Тема: Использование компонентов OpenFileDialog и SaveFileDialog для работы с файлами (4)
стр. 178 - 185 Серкова Е.Г. Основы алгоритмизации и программирования (ОП.04): практикум/Е.Г.Серкова. – Ростов н/Д : Феникс, 2019. – 188,[1] с. : - (Среднее профессиональное образование)

Практическая работа №38

Тема: Работа с дисками, каталогами, файлами. Классы и методы (2)

Теоретический материал

Для работы с каталогами в пространстве имен System.IO предназначены сразу два класса: **Directory** и **DirectoryInfo**.

Класс Directory

Класс Directory предоставляет ряд статических методов для управления каталогами. Некоторые из этих методов:

- **CreateDirectory(path)**: создает каталог по указанному пути path
- **Delete(path)**: удаляет каталог по указанному пути path
- **Exists(path)**: определяет, существует ли каталог по указанному пути path. Если существует, возвращается true, если не существует, то false
- **GetDirectories(path)**: получает список каталогов в каталоге path
- **GetFiles(path)**: получает список файлов в каталоге path
- **Move(sourceDirName, destDirName)**: перемещает каталог
- **GetParent(path)**: получение родительского каталога

Класс DirectoryInfo

Данный класс предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на Directory. Некоторые из его свойств и методов:

- **Create()**: создает каталог
- **CreateSubdirectory(path)**: создает подкаталог по указанному пути path
- **Delete()**: удаляет каталог
- Свойство **Exists**: определяет, существует ли каталог
- **GetDirectories()**: получает список каталогов
- **GetFiles()**: получает список файлов
- **MoveTo(destDirName)**: перемещает каталог
- Свойство **Parent**: получение родительского каталога
- Свойство **Root**: получение корневого каталога

Посмотрим на примерах применение этих классов

Получение списка файлов и подкаталогов

```
string dirName = "C:\\";

if (Directory.Exists(dirName))
{
   Console.WriteLine("Подкаталоги:");
   string[] dirs = Directory.GetDirectories(dirName);
   foreach (string s in dirs)
   {
      Console.WriteLine(s);
   }
   Console.WriteLine();
   Console.WriteLine("Файлы:");
   string[] files = Directory.GetFiles(dirName);
   foreach (string s in files)
   {
      Console.WriteLine(s);
   }
}
```

Обратите внимание на использование слешей в именах файлов. Либо мы используем двойной слеш: "C:\\", либо одинарный, но тогда перед всем путем ставим знак @: @"C:\Program Files"

Создание каталога

```
string path = @"C:\SomeDir";
string subpath = @"program\avalon";
DirectoryInfo dirInfo = new DirectoryInfo(path);
if (!dirInfo.Exists)
{
   dirInfo.Create();
}
dirInfo.CreateSubdirectory(subpath);
```

Вначале проверяем, а нету ли такой директории, так как если она существует, то ее создать будет нельзя, и приложение выбросит ошибку. В итоге у нас получится следующий путь: "C:\SomeDir\program\avalon"

Получение информации о каталоге

```
string dirName = "C:\\Program Files";

DirectoryInfo dirInfo = new DirectoryInfo(dirName);

Console.WriteLine($"Название каталога: {dirInfo.Name}");
Console.WriteLine($"Полное название каталога: {dirInfo.FullName}");
Console.WriteLine($"Время создания каталога: {dirInfo.CreationTime}");
Console.WriteLine($"Корневой каталог: {dirInfo.Root}");
```

Удаление каталога

Если мы просто применим метод Delete к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение нам выбросит ошибку. Поэтому нам надо пере-

дать в метод Delete дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым:

```
string dirName = @"C:\SomeFolder";

try
{
   DirectoryInfo dirInfo = new DirectoryInfo(dirName);
   dirInfo.Delete(true);
   Console.WriteLine("Каталог удален");
}
catch (Exception ex)
{
   Console.WriteLine(ex.Message);
}
```

Или так:

```
string dirName = @"C:\SomeFolder";

Directory.Delete(dirName, true);
```

Перемещение каталога

```
string oldPath = @"C:\SomeFolder";
string newPath = @"C:\SomeDir";
DirectoryInfo dirInfo = new DirectoryInfo(oldPath);
if (dirInfo.Exists && Directory.Exists(newPath) == false)
{
   dirInfo.MoveTo(newPath);
}
```

При перемещении надо учитывать, что новый каталог, в который мы хотим перемесить все содержимое старого каталога, не должен существовать.

Подобно паре Directory/DirectoryInfo для работы с файлами предназначена пара классов **File** и **FileInfo**. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса FileInfo:

- **CopyTo(path)**: копирует файл в новое место по указанному пути path
- **Create()**: создает файл
- **Delete()**: удаляет файл
- **MoveTo(destFileName)**: перемещает файл в новое место
- Свойство **Directory**: получает родительский каталог в виде объекта DirectoryInfo
- Свойство **DirectoryName**: получает полный путь к родительскому каталогу
- Свойство **Exists**: указывает, существует ли файл
- Свойство **Length**: получает размер файла
- Свойство **Extension**: получает расширение файла
- Свойство **Name**: получает имя файла
- Свойство **FullName**: получает полное имя файла

Класс File реализует похожую функциональность с помощью статических методов:

- **Copy()**: копирует файл в новое место
- **Create()**: создает файл
- **Delete()**: удаляет файл
- **Move**: перемещает файл в новое место
- **Exists(file)**: определяет, существует ли файл

Получение информации о файле

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    Console.WriteLine("Имя файла: {0}", fileInf.Name);
    Console.WriteLine("Время создания: {0}", fileInf.CreationTime);
    Console.WriteLine("Размер: {0}", fileInf.Length);
}
```

Удаление файла

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.Delete();
   // альтернатива с помощью класса File
   // File.Delete(path);
}
```

Перемещение файла

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.MoveTo(newPath);
   // альтернатива с помощью класса File
   // File.Move(path, newPath);
}
```

Копирование файла

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.CopyTo(newPath, true);
   // альтернатива с помощью класса File
   // File.Copy(path, newPath, true);
}
```

Метод CopyTo класса FileInfo принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если true, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра передать значение false, то если такой файл уже существует, приложение выдаст ошибку.

Метод Copy класса File принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

Класс **FileStream** представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Создание FileStream

Для создания объекта FileStream можно использовать как конструкторы этого класса, так и статические методы класса File. Конструктор FileStream имеет множество перегруженных версий, из которых отмечу лишь одну, самую простую и используемую:

FileStream(string filename, FileMode mode)

Здесь в конструктор передается два параметра: путь к файлу и перечисление **FileMode**. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

- **Append**: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
- **Create**: создается новый файл. Если такой файл уже существует, то он перезаписывается
- **CreateNew**: создается новый файл. Если такой файл уже существует, то он приложение выбрасывает ошибку
- **Open**: открывает файл. Если файл не существует, выбрасывается исключение
- **OpenOrCreate**: если файл существует, он открывается, если нет - создается новый
- **Truncate**: если файл существует, то он перезаписывается. Файл открывается только для записи.

Другой способ создания объекта FileStream представляют статические методы класса File:

FileStream File.Open(string file, FileMode mode);
FileStream File.OpenRead(string file);
FileStream File.OpenWrite(string file);

Первый метод открывает файл с учетом объекта FileMode и возвращает файловой поток FileStream. У этого метода также есть несколько перегруженных версий. Второй метод открывает поток для чтения, а третий открывает поток для записи.

Свойства и методы FileStream

Рассмотрим наиболее важные его свойства и методы класса FileStream:

- Свойство **Length**: возвращает длину потока в байтах
- Свойство **Position**: возвращает текущую позицию в потоке
- void CopyTo(Stream destination): копирует данные из текущего потока в поток destination
- Task CopyToAsync(Stream destination): асинхронная версия метода CopyToAsync
- int Read(byte[] array, int offset, int count): считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:
    - array - массив байтов, куда будут помещены считываемые из файла данные

  - offset представляет смещение в байтах в массиве array, в который считанные байты будут помещены
  - count - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.
  - Task<int> ReadAsync(byte[] array, int offset, int count): асинхронная версия метода Read
- long Seek(long offset, SeekOrigin origin): устанавливает позицию в потоке со смещением на количество байт, указанных в параметре offset.
- void Write(byte[] array, int offset, int count): записывает в файл данные из массива байтов. Принимает три параметра:
  - array - массив байтов, откуда данные будут записываться в файл
  - offset - смещение в байтах в массиве array, откуда начинается запись байтов в поток
  - count - максимальное число байтов, предназначенных для записи
- ValueTask WriteAsync(byte[] array, int offset, int count): асинхронная версия метода Write

Чтение и запись файлов

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру, FileStream может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Посмотрим на примере считывания-записи в текстовый файл:

```
using System;
using System.IO;

namespace HelloApp
{
   class Program
   {
      static void Main(string[] args)
      {
         // создаем каталог для файла
         string path = @"C:\SomeDir2";
         DirectoryInfo dirInfo = new DirectoryInfo(path);
         if (!dirInfo.Exists)
         {
            dirInfo.Create();
         }
         Console.WriteLine("Введите строку для записи в файл:");
         string text = Console.ReadLine();

         // запись в файл
         using (FileStream fstream = new FileStream($"{path}\note.txt", File-
Mode.OpenOrCreate))
         {
```

```
            // преобразуем строку в байты
            byte[] array = System.Text.Encoding.Default.GetBytes(text);
            // запись массива байтов в файл
            fstream.Write(array, 0, array.Length);
            Console.WriteLine("Текст записан в файл");
        }

        // чтение из файла
        using (FileStream fstream = File.OpenRead($"{path}\note.txt"))
        {
            // преобразуем строку в байты
            byte[] array = new byte[fstream.Length];
            // считываем данные
            fstream.Read(array, 0, array.Length);
            // декодируем байты в строку
            string textFromFile = System.Text.Encoding.Default.GetString(array);
            Console.WriteLine($"Текст из файла: {textFromFile}");
        }

        Console.ReadLine();
    }
  }
}
```

Разберем этот пример. Вначале создается папка для файла. Кроме того, на уровне операционной системы могут быть установлены ограничения на запись в опреееделенных каталогах, и при попытке создания и записи файла в подобных каталогах мы получим ошибку.

И при чтении, и при записи используется оператор using. Не надо путать данный оператор с директивой using, которая подключает пространства имен в начале файла кода. Оператор using позволяет создавать объект в блоке кода, по завершению которого вызывается метод Dispose у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная fstream.

И при записи, и при чтении применяется объект кодировки Encoding.Default из пространства имен System.Text. В данном случае мы используем два его метода: GetBytes для получения массива байтов из строки и GetString для получения строки из массива байтов.

В итоге введенная нами строка записывается в файл *note.txt*. По сути это бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

```
fstream.WriteByte(13);
fstream.WriteByte(103);
```

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - StreamReader и StreamWriter.

Хотя в данном простеньком консольном приложении, но в реальных приложениях рекомендуется использовать асинхронные версии методов FileStream, поскольку операции с файлами могут занимать продолжительное время и являются узким местом в работе про-

граммы. Например, изменим выше приведенную программу, применив асинхронные методы:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace HelloApp
{
   class Program
   {
      static async Task Main(string[] args)
      {
         // создаем каталог для файла
         string path = @"C:\SomeDir3";
         DirectoryInfo dirInfo = new DirectoryInfo(path);
         if (!dirInfo.Exists)
         {
            dirInfo.Create();
         }
         Console.WriteLine("Введите строку для записи в файл:");
         string text = Console.ReadLine();

         // запись в файл
         using (FileStream fstream = new FileStream($"{path}\note.txt", File-
Mode.OpenOrCreate))
         {
            byte[] array = System.Text.Encoding.Default.GetBytes(text);
            // асинхронная запись массива байтов в файл
            await fstream.WriteAsync(array, 0, array.Length);
            Console.WriteLine("Текст записан в файл");
         }

         // чтение из файла
         using (FileStream fstream = File.OpenRead($"{path}\note.txt"))
         {
            byte[] array = new byte[fstream.Length];
            // асинхронное чтение файла
            await fstream.ReadAsync(array, 0, array.Length);

            string textFromFile = System.Text.Encoding.Default.GetString(array);
            Console.WriteLine($"Текст из файла: {textFromFile}");
         }

         Console.ReadLine();
      }
   }
}
```

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную структуру. И, зная эту структуру, мы можем взять из файла нужную порцию информации или наоброт записать в опреде-

ленном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта идут различные метаданные - количество каналов аудио, частота дискретизации и т.д.

С помощью метода **Seek()** мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: offset (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

- **SeekOrigin.Begin**: начало файла
- **SeekOrigin.End**: конец файла
- **SeekOrigin.Current**: текущая позиция в файле

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение offset относительно позиции, указанной в качестве второго параметра. Смещение может быть отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

Рассмотрим на примере:

```
using System.IO;
using System.Text;

class Program
{
   static void Main(string[] args)
   {
      string text = "hello world";

      // запись в файл
      using (FileStream fstream = new FileStream(@"D:\note.dat", FileMode.OpenOrCreate))
      {
         // преобразуем строку в байты
         byte[] input = Encoding.Default.GetBytes(text);
         // запись массива байтов в файл
         fstream.Write(input, 0, input.Length);
         Console.WriteLine("Текст записан в файл");

         // перемещаем указатель в конец файла, до конца файла- пять байт
         fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока

         // считываем четыре символов с текущей позиции
         byte[] output = new byte[4];
         fstream.Read(output, 0, output.Length);
         // декодируем байты в строку
         string textFromFile = Encoding.Default.GetString(output);
         Console.WriteLine($"Текст из файла: {textFromFile}"); // worl

         // заменим в файле слово world на слово house
         string replaceText = "house";
         fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца потока
         input = Encoding.Default.GetBytes(replaceText);
         fstream.Write(input, 0, input.Length);
```

```
            // считываем весь файл
            // возвращаем указатель в начало файла
            fstream.Seek(0, SeekOrigin.Begin);
            output = new byte[fstream.Length];
            fstream.Read(output, 0, output.Length);
            // декодируем байты в строку
            textFromFile = Encoding.Default.GetString(output);
            Console.WriteLine($"Текст из файла: {textFromFile}"); // hello house
        }
        Console.Read();
    }
}
```

Консольный вывод:

```
Текст записан в файл
Текст из файла: worl
Текст из файла: hello house
```

Вызов fstream.Seek(-5, SeekOrigin.End) перемещает курсор потока в конец файлов назад на пять символов:

То есть после записи в новый файл строки "hello world" курсор будет стоять на позиции символа "w".

После этого считываем четыре байта начиная с символа "w". В данной кодировке 1 символ будет представлять 1 байт. Поэтому чтение 4 байтов будет эквивалентно чтению четырех сиволов: "worl".

Затем опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа "w"), и осуществляем запись строки "house". Таким образом, строка "house" заменяет строку "world".

Закрытие потока

В примерах выше для закрытия потока применяется конструкция **using**. После того как все операторы и выражения в блоке using отработают, объект FileStream уничтожается. Однако мы можем выбрать и другой способ:

```
FileStream fstream = null;
try
{
    fstream = new FileStream(@"D:\note3.dat", FileMode.OpenOrCreate);
    // операции с потоком
}
catch(Exception ex)
{

}
finally
{
    if (fstream != null)
        fstream.Close();
```

```
}
```

Если мы не используем конструкцию using, то нам надо явным образом вызвать метод Close(): fstream.Close()

Практическая работа №39

Тема: Контейнеры Grid GridSplitter (2)
Цель: изучить контейнеры Grid, GridSplitter

Теоретический материал

Grid

Это наиболее мощный и часто используемый контейнер, напоминающий обычную таблицу. Он содержит столбцы и строки, количество которых задает разработчик. Для определения строк используется свойство **RowDefinitions**, а для определения столбцов - свойство **ColumnDefinitions**:

```
<Grid.RowDefinitions>
   <RowDefinition></RowDefinition>
   <RowDefinition></RowDefinition>
   <RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
   <ColumnDefinition></ColumnDefinition>
   <ColumnDefinition></ColumnDefinition>
   <ColumnDefinition></ColumnDefinition>
</Grid.ColumnDefinitions>
```

Каждая строка задается с помощью вложенного элемента RowDefinition, который имеет открывающий и закрывающий тег. При этом задавать дополнительную информацию необязательно. То есть в данном случае у нас определено в гриде 3 строки.

Каждая столбец задается с помощью вложенного элемента ColumnDefinition. Таким образом, здесь мы определили 3 столбца. ТО есть в итоге у нас получится таблица 3х3.

Чтобы задать позицию элемента управления с привязкой к определенной ячейке Grid, в разметке элемента нужно прописать значения свойств **Grid.Column** и **Grid.Row**, тем самым указывая, в каком столбце и строке будет находиться элемент. Кроме того, если мы хотим растянуть элемент управления на несколько строк или столбцов, то можно указать свойства **Grid.ColumnSpan** и **Grid.RowSpan**, как в следующем примере:

```
<Window x:Class="LayoutApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:LayoutApp"
        mc:Ignorable="d"
        Title="Grid" Height="250" Width="350">
   <Grid ShowGridLines="True">
```

```
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Grid.Column="0" Grid.Row="0" Content="Строка 0 Столбец 0"  />
        <Button Grid.Column="0" Grid.Row="1" Content="Объединение трех столбцов"
Grid.ColumnSpan="3"  />
        <Button Grid.Column="2" Grid.Row="2" Content="Строка 2 Столбец 2"  />
    </Grid>
</Window>
```

Атрибут ShowGridLines="True" у элемента Grid задает видимость сетки, по умолчанию оно равно False.

Установка размеров

Но если в предыдущем случае у нас строки и столбцы были равны друг другу, то теперь попробуем их настроить столбцы по ширине, а строки - по высоте. Есть несколько вариантов настройки размеров.

Автоматические размеры

Здесь столбец или строка занимает то место, которое им нужно

```
<ColumnDefinition Width="Auto" />
<RowDefinition Height="Auto" />
```

Абсолютные размеры

В данном случае высота и ширина указываются в единицах, независимых от устройства:

```
<ColumnDefinition Width="150" />
<RowDefinition Height="150" />
```

Также абсолютные размеры можно задать в пикселях, дюймах, сантиметрах или точках:

пиксели	px
дюймы	in
сантиметры	cm
точки	pt

Например,

```
<ColumnDefinition Width="1 in" />
<RowDefinition Height="10 px" />
```

Пропорциональные размеры.

Например, ниже задаются два столбца, второй из которых имеет ширину в четверть от ширины первого:

```
<ColumnDefinition Width="*" />
<ColumnDefinition Width="0.25*" />
```

Если строка или столбец имеет высоту, равную *, то данная строка или столбце будет занимать все оставшееся место. Если у нас есть несколько сток или столбцов, высота которых равна *, то все доступное место делится поровну между всеми такими сроками и столбцами. Использование коэффициентов (0.25*) позволяет уменьшить или увеличить выделенное место на данный коэффициент. При этом все коэффициенты складываются (коэффициент * аналогичен 1*) и затем все пространство делится на сумму коэффициентов.

Например, если у нас три столбца:

```
<ColumnDefinition Width="*" />
<ColumnDefinition Width="0.5*" />
<ColumnDefinition Width="1.5*" />
```

В этом случае сумма коэффициентов равна 1* + 0.5* + 1.5* = 3*. Если у нас грид имеет ширину 300 единиц, то для коэфициент 1* будет соответствовать пространству 300 / 3 = 100 единиц. Поэтому первый столбец будет иметь ширину в 100 единиц, второй - 100*0.5=50 единиц, а третий - 100 * 1.5 = 150 единиц.

Можно комбинировать все типы размеров. В этом случае от ширины/высоты грида отнимается ширина/высота столбцов/строк с абсолютными или автоматическими размерами, и затем оставшееся место распределяется между столбцами/строками с пропорциональными размерами:

UniformGrid

Аналогичен контейнеру Grid контейнер **UniformGrid**, только в этом случае все столбцы и строки одинакового размера и используется упрощенный синтаксис для их определения:

```
<UniformGrid Rows="2" Columns="2">
   <Button Content="Left Top" />
   <Button Content="Right Top" />
   <Button Content="Left Bottom" />
   <Button Content="Right Bottom" />
</UniformGrid>
```

GridSplitter

Элемент **GridSplitter** помогает создавать интерфейсы наподобие элемента SplitContainer в WinForms, только более функциональные. Он представляет собой некоторый разделитель между столбцами или строками, путем сдвига которого можно регулировать ширину столбцов и высоту строк. В качестве примера можно привести стандартный интерфейс

проводника в Windows, где разделительная полоса отделяет древовидный список папок от панели со списком файлов. Например,

```
<Grid>
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
   </Grid.ColumnDefinitions>
   <Button Grid.Column="0" Content="Левая кнопка" />
   <GridSplitter Grid.Column="1" ShowsPreview="False" Width="3"
      HorizontalAlignment="Center" VerticalAlignment="Stretch" />
   <Button Grid.Column="2" Content="Правая кнопка" />
</Grid>
```

Двигая центральную линию, разделяющую правую и левую части, мы можем устанавливать их ширину.

Итак, чтобы использовать элемент GridSplitter, нам надо поместить его в ячейку в Gride. По сути это обычный элемент, такой же, как кнопка. Как выше, у нас три ячейки (так как три столбца и одна строка), и GridSplitter помещен во вторую ячейку. Обычно строка или столбец, в которые помещают элемент, имеет для свойств Height или Width значение Auto.

Если у нас несколько строк, и мы хотим, чтобы разделитель распространялся на несколько строк, то мы можем задать свойство Grid.RowSpan:

```
<Grid.ColumnDefinitions>
   <ColumnDefinition Width="*" />
   <ColumnDefinition Width="Auto" />
   <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
   <RowDefinition></RowDefinition>
   <RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<GridSplitter Grid.Column="1" Grid.RowSpan="2" ShowsPreview="False" Width="3"
   HorizontalAlignment="Center" VerticalAlignment="Stretch" />
```

В случае, если мы задаем горизонтальный разделитель, то тогда соответственно надо использовать свойство Grid.ColumnSpan

Затем нам надо настроить свойства. Во-первых, надо настроить ширину (Width) для вертикальных сплитеров и высоту (Height) для горизонтальных. Если не задать соответствующее свойство, то сплитер мы не увидим, так как он изначально очень мал.

Затем нам надо задать выравнивание. Если мы хотим, что сплитер заполнял всю высоту доступной области (то есть если у нас вертикальный сплитер), то нам надо установить для свойства **VerticalAlignment** значение Stretch.

Если же у нас горизонтальный сплитер, то надо установить свойство **HorizontalAlignment** в Stretch

Также в примере выше используется свойство **ShowsPreview**. Если оно равно False, то изменение границ кнопок будет происходить сразу же при перемещении сплитера. Если же оно равно True, тогда изменение границ будет происходить только после того, как перемещение сплитера завершится, и при перемещении сплиттера мы увидим его проекцию.

В отличие от элемента SplitContainer в WinForms, в WPF можно установить различное количество динамически регулируемых частей окна. Немного усовершенствуем предыдущий пример:

```
<Grid>
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
      <RowDefinition Height="*"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="*"></RowDefinition>
   </Grid.RowDefinitions>
   <GridSplitter Grid.Column="1"  Grid.Row="0" ShowsPreview="False" Width="3"
      HorizontalAlignment="Center" VerticalAlignment="Stretch" />
   <GridSplitter Grid.Row="1" Grid.ColumnSpan="3" Height="3"
      HorizontalAlignment="Stretch" VerticalAlignment="Center" />
   <Canvas Grid.Column="0" Grid.Row="0">
      <TextBlock>Левая панель</TextBlock>
   </Canvas>
   <Canvas Grid.Column="2" Grid.Row="0" Background="LightGreen">
      <TextBlock>Правая панель</TextBlock>
   </Canvas>
   <Canvas Grid.ColumnSpan="3" Grid.Row="2" Background="#dfffff">
      <TextBlock Canvas.Left="60">Нижняя панель</TextBlock>
   </Canvas>
</Grid>
```

Здесь у нас сразу два сплитера: один между двумя верхними и нижней панелями, а второй - между правой и левой панелями.

Практическая работа №40

Тема: Контейнеры StackPanel, DockPanel (2)

StackPanel

Это более простой элемент компоновки. Он располагает все элементы в ряд либо по горизонтали, либо по вертикали в зависимости от ориентации. Например,

```
<Window x:Class="LayoutApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
      xmlns:local="clr-namespace:LayoutApp"
      mc:Ignorable="d"
      Title="StackPanel" Height="300" Width="300">
  <Grid>
    <StackPanel>
      <Button Background="Blue" Content="1" />
      <Button Background="White" Content="2" />
      <Button Background="Red" Content="3" />
    </StackPanel>
  </Grid>
</Window>
```

В данном случае для свойства Orientation по умолчанию используется значение Vertical, то есть StackPanel создает вертикальный ряд, в который помещает все вложенные элементы сверху вниз. Мы также можем задать горизонтальный стек. Для этого нам надо указать свойство Orientation="Horizontal":

```
<Window x:Class="LayoutApp.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:local="clr-namespace:LayoutApp"
      mc:Ignorable="d"
      Title="StackPanel" Height="300" Width="300">
  <StackPanel Orientation="Horizontal">
    <Button Background="Blue" MinWidth="30" Content="1" />
    <Button Background="White" MinWidth="30" Content="2" />
    <Button Background="Red" MinWidth="30" Content="3" />
  </StackPanel>
</Window>
```

При горизонтальной ориентации все вложенные элементы располагаются слева направо. Если мы хотим, чтобы наполнение стека начиналось справа налево, то нам надо задать свойство FlowDirection: <StackPanel Orientation="Horizontal" FlowDirection="RightToLeft">. По умолчанию это свойство имеет значение LeftToRight - то есть слева направо.

DockPanel

Этот контейнер прижимает свое содержимое к определенной стороне внешнего контейнера. Для этого у вложенных элементов надо установить сторону, к которой они будут прижиматься с помощью свойства DockPanel.Dock. Например,

```
<Window x:Class="LayoutApp.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
        xmlns:local="clr-namespace:LayoutApp"
        mc:Ignorable="d"
        Title="DockPanel" Height="250" Width="300">
    <DockPanel LastChildFill="True">
        <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кноп-
ка" />
        <Button DockPanel.Dock="Bottom" Background="BlanchedAlmond"
Content="Нижняя кнопка" />
        <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кноп-
ка" />
        <Button DockPanel.Dock="Right" Background="DarkGreen" Content="Правая
кнопка" />
        <Button Background="LightGreen" Content="Центр" />
    </DockPanel>
</Window>
```

В итоге получаем массив кнопок, каждая из которых прижимается к определенной стороне элемента DockPanel:

Причем у последней кнопки мы можем не устанавливать свойство DockPanel.Dock. Она уже заполняет все оставшееся пространство. Такой эффект получается благодаря установке у DockPanel свойства LastChildFill="True", которое означает, что последний элемент заполняет все оставшееся место. Если у этого свойства поменять True на False, то кнопка прижмется к левой стороне, заполнив только о место, которое ей необходимо.

Также обратите внимание на порядок прикрепления к кнопкам свойства DockPanel.Dock. Например, если мы изменим порядок на:

```
      <DockPanel LastChildFill="True">
1        <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка" />
2        <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка" />
3        <Button DockPanel.Dock="Right" Background="DarkGreen" Content="Правая кнопка"
4     />
5        <Button DockPanel.Dock="Bottom" Background="BlanchedAlmond" Content="Нижняя
6     кнопка" />
7        <Button Background="LightGreen" Content="Центр" />
      </DockPanel>
```

В этом случае нижняя кнопка уже будет заполнять меньшее место.

Мы также можем прижать к одной стороне сразу несколько элементов. В этом случае они просто будут располагаться по порядку:

```
1     <DockPanel LastChildFill="True">
2        <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка
3     1" />
4        <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка
5     2" />
6        <Button DockPanel.Dock="Bottom" Background="BlanchedAlmond" Content="Нижняя
7     кнопка" />
8        <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка1"
9     />
         <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка2"
```

```
/>
   <Button DockPanel.Dock="Right" Background="DarkGreen" Content="Правая кнопка"
/>
   <Button Background="LightGreen" Content="Центр" />
</DockPanel>
```

Контейнер DockPanel особенно удобно использовать для создания стандартных интерфейсов, где верхнюю и левую часть могут занимать какие-либо меню, нижнюю - строка состояния, правую - какая-то дополнительная информация, а в центре будет находиться основное содержание.

Практическая работа №41

Тема: Контейнеры WrapPanel Canvas (2)

WrapPanel

Эта панель, подобно StackPanel, располагает все элементы в одной строке или колонке в зависимости от того, какое значение имеет свойство Orientation - Horizontal или Vertical. Главное отличие от StackPanel - если элементы не помещаются в строке или столбце, создаются новые столбец или строка для не поместившихся элементов.

```
<Window x:Class="LayoutApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:LayoutApp"
        mc:Ignorable="d"
        Title="WrapPanel" Height="250" Width="300">
    <WrapPanel>
        <Button Background="AliceBlue" Content="Кнопка 1" />
        <Button Background="Blue" Content="Кнопка 2" />
        <Button Background="Aquamarine" Content="Кнопка 3" Height="30"/>
        <Button Background="DarkGreen" Content="Кнопка 4" Height="20"/>
        <Button Background="LightGreen" Content="Кнопка 5"/>
        <Button Background="RosyBrown" Content="Кнопка 6" Width="80" />
        <Button Background="GhostWhite" Content="Кнопка 7" />
    </WrapPanel>
</Window>
```

В горизонтальном стеке те элементы, у которых явным образом не установлена высота, будут автоматически принимать высоту самого большого элемента из стека.

Вертикальный WrapPanel делается аналогично:

```
<WrapPanel Orientation="Vertical">
    <Button Background="AliceBlue" Content="Кнопка 1" Height="50" />
    <Button Background="Blue" Content="Кнопка 2" />
    <Button Background="Aquamarine" Content="Кнопка 3" Width="60"/>
```

```
   <Button Background="DarkGreen" Content="Кнопка 4" Width="80"/>
   <Button Background="LightGreen" Content="Кнопка 5"/>
   <Button Background="RosyBrown" Content="Кнопка 6" Height="80" />
   <Button Background="GhostWhite" Content="Кнопка 7" />
   <Button Background="Bisque" Content="Кнопка 8" />
</WrapPanel>
```

В вертикальном стеке элементы, у которых явным образом не указана ширина, автоматически принимают ширину самого широкого элемента.

Мы также можем установить для всех вложенных элементов какую-нибудь определенную ширину (с помощью свойства ItemWidth) или высоту (свойство ItemHeight):

```
<WrapPanel ItemHeight="30" ItemWidth="80" Orientation="Horizontal">
   <Button Background="AliceBlue" Content="1" />
   <Button Background="Blue" Content="2" />
   <Button Background="Aquamarine" Content="3"/>
   <Button Background="DarkGreen" Content="4"/>
   <Button Background="LightGreen" Content="5"/>
   <Button Background="AliceBlue" Content="6"  />
   <Button Background="Blue" Content="7" />
</WrapPanel>
```

Canvas

Контейнер Canvas является наиболее простым контейнером. Для размещения на нем необходимо указать для элементов точные координаты относительно сторон Canvas. Для установки координат элементов используются свойства Canvas.Left, Canvas.Right, Canvas.Bottom, Canvas.Top. Например, свойство Canvas.Left указывает, на сколько единиц от левой стороны контейнера будет находиться элемент, а свойство Canvas.Top - насколько единиц ниже верхней границы контейнера находится элемент.

При этом в качестве единиц используются не пиксели, а независимые от устройства единицы, которые помогают эффективно управлять масштабированием элементов. Каждая такая единица равна 1 /96 дюйма, и при стандартной установке в 96 dpi эта независимая от устройства единица будет равна физическому пикселю, так как 1/96 дюйма * 96 dpi (96 точек на дюйм) = 1. В тоже время при работе на других мониторах или при других установленных размеры, установленные в приложении, будут эффективно масштабироваться. Например, при разрешении в 120 dpi одна условная единица будет равна 1,25 пикселя, так как 1/96 дюйма * 120 dpi= 1,25 пикселя.

Если элемент не использует свойства Canvas.Top и другие, то по умолчанию свойства Canvas.Left и Canvas.Top будут равны нулю, то есть он будет находиться в верхнем левом углу.

Также надо учитывать, что нельзя одновременно задавать Canvas.Left и Canvas.Right или Canvas.Bottom и Canvas.Top. Если подобное произойдет, то последнее заданное свойство не будет учитываться. Например:

```
<Window x:Class="Layout.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="300" Width="300">
    <Grid>
        <Canvas Background="Lavender">
            <Button Background="AliceBlue" Content="Top 20 Left 40" Canvas.Top="20" Canvas.Left="40" />
            <Button Background="LightSkyBlue" Content="Top 20 Right 20" Canvas.Top="20" Canvas.Right="20"/>
            <Button Background="Aquamarine" Content="Bottom 30 Left 20" Canvas.Bottom="30" Canvas.Left="20"/>
            <Button Background="LightCyan" Content="Bottom 20 Right 40" Canvas.Bottom="20" Canvas.Right="40"/>
        </Canvas>
    </Grid>
</Window>
```

Практическая работа №42

Тема: Создание интерфейса приложения (2)

Практическая работа №43

Тема: Разработка приложений с элементами управление (2)

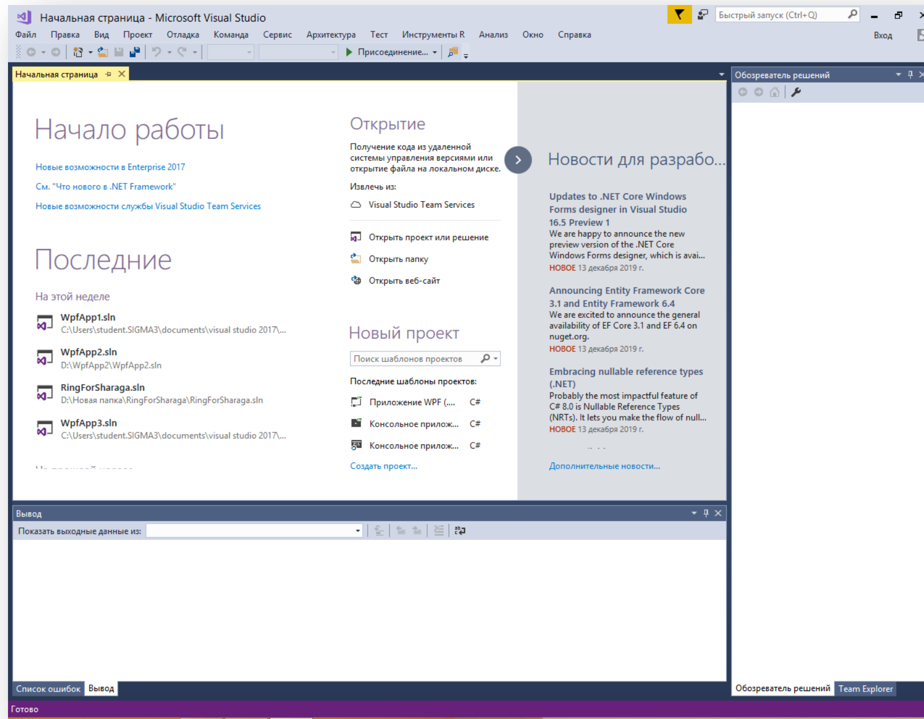
Практическая работа №44

Тема: Разработка интерфейса приложения (4)

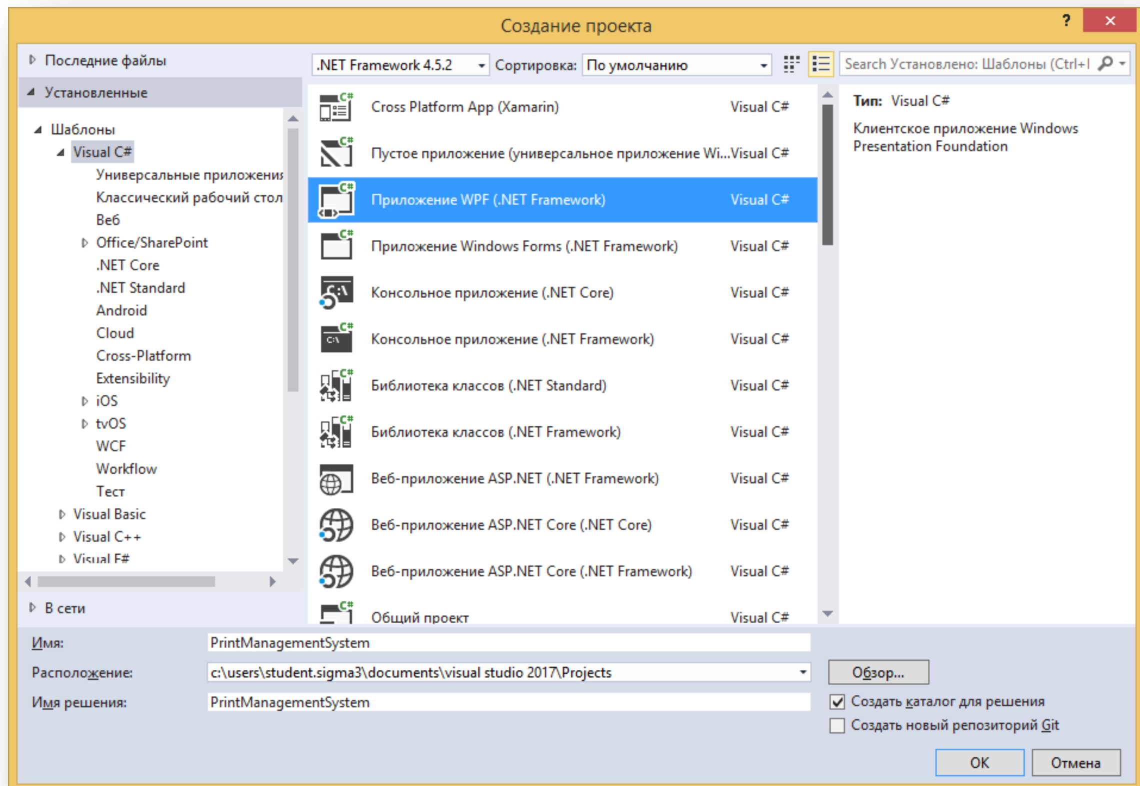
Цель: Приобрести навыки разработки графического интерфейса пользователя. Познакомиться с основными элементами интерфейса WPF.

Ход работы:

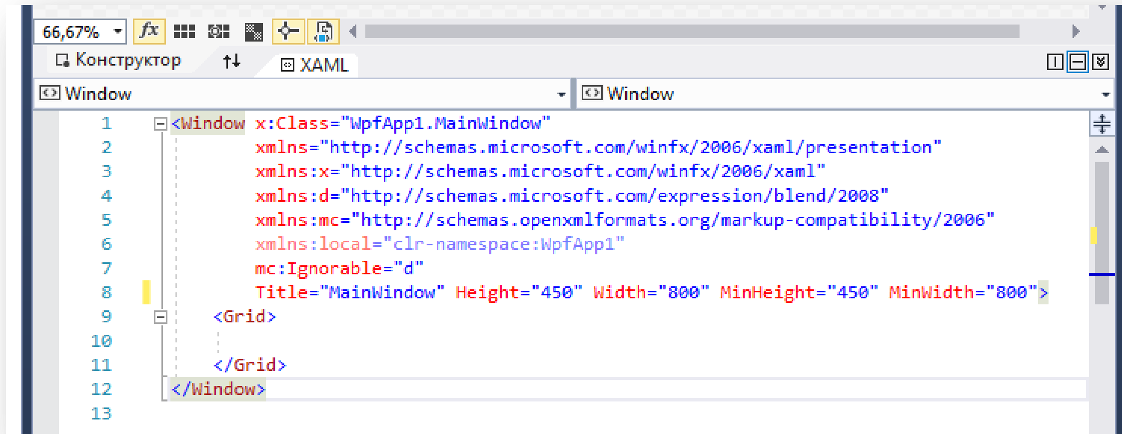
1. Запустить Microsoft Visual Studio.



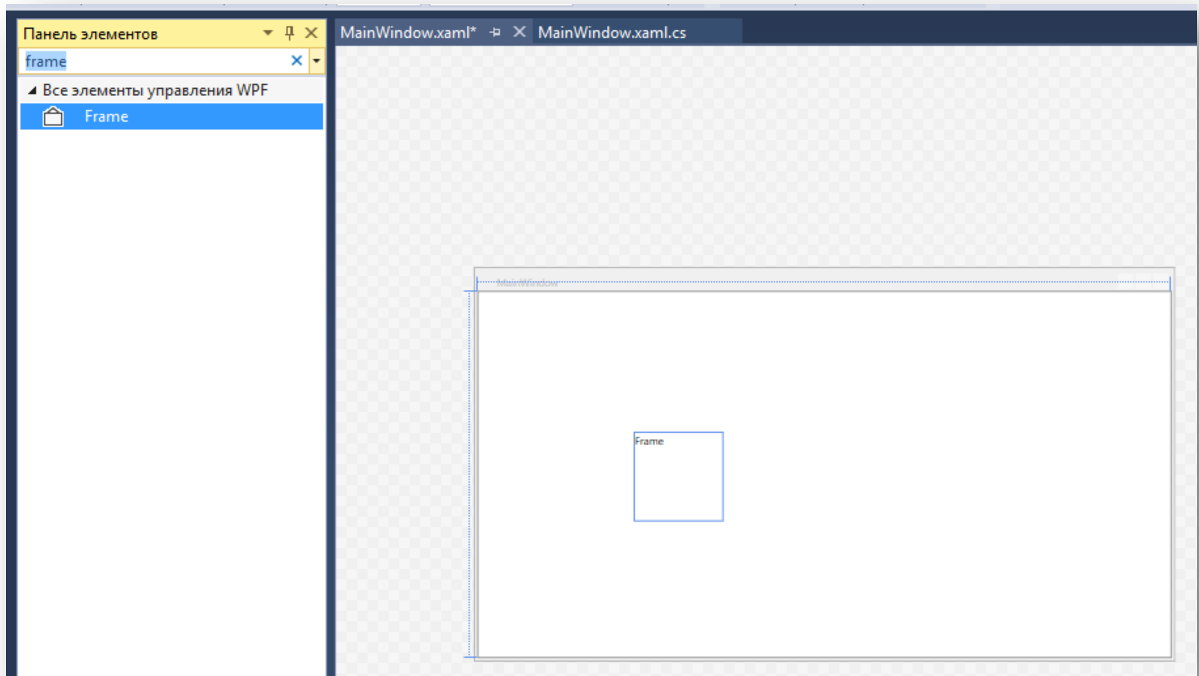
2. Создать новый проект - WPF, дать наименование своей предметной области.



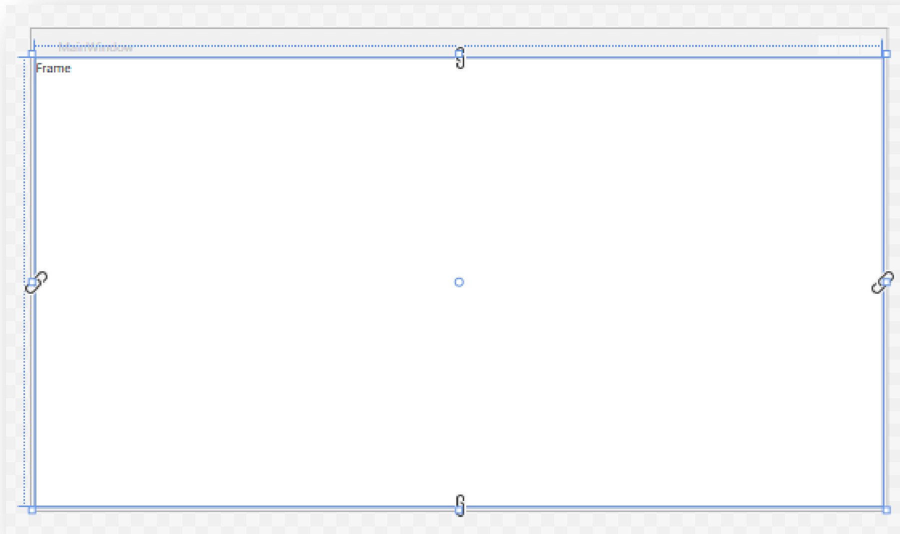
3. Укажите следующие атрибуты у стартовой страницы программы "MainWindows".



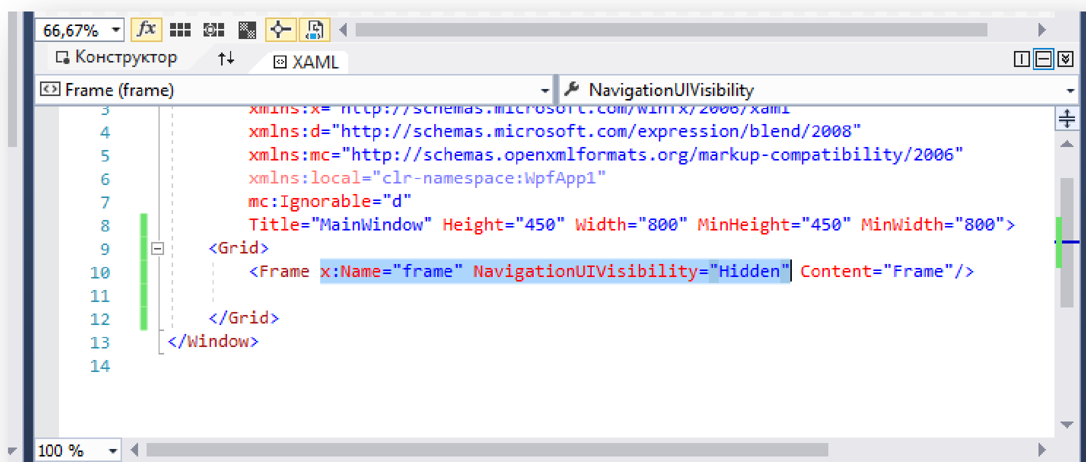
4. Добавьте на рабочее пространство программы элемент "Frame".



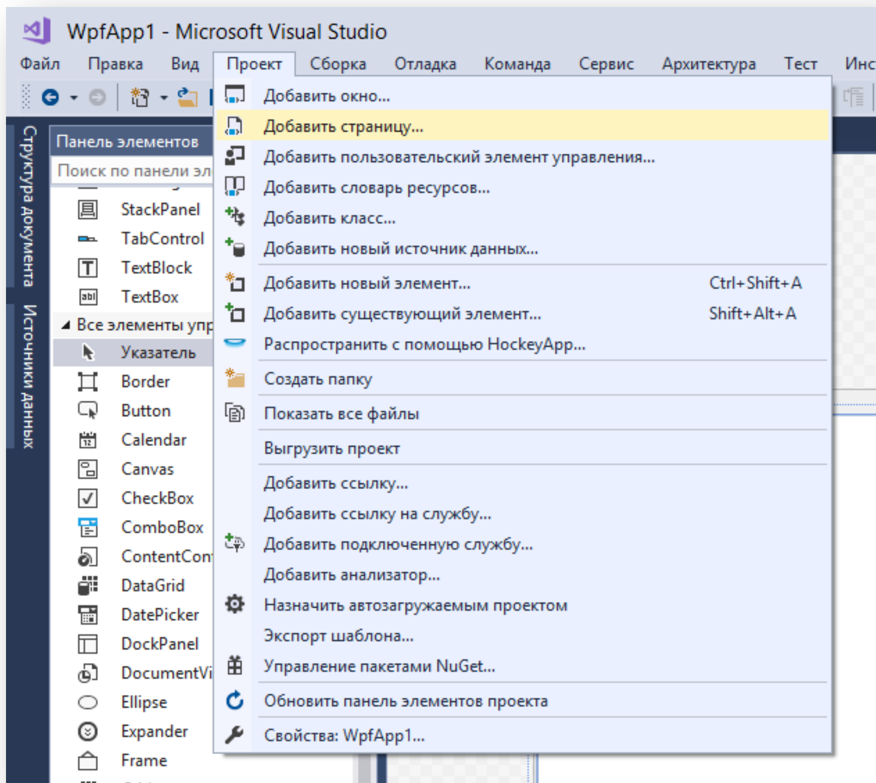
5. Растяните элемент фрейм на всю рабочую область, а также привяжите к правой и нижней грани. Это позволит элементу растягиваться при масштабировании окна.



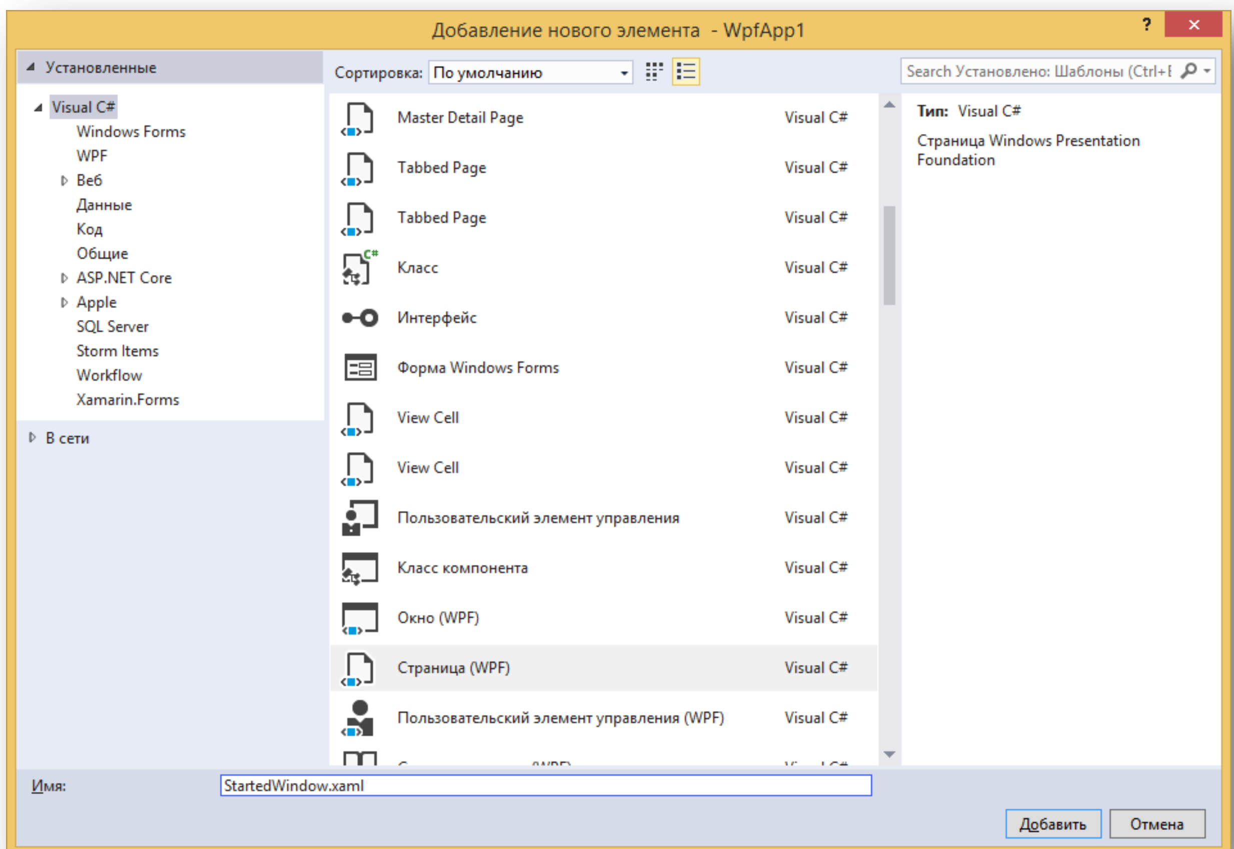
6. Дайте имя элементу, а также скройте отображение пользовательского интерфейса перехода, прописав следующие атрибуты:



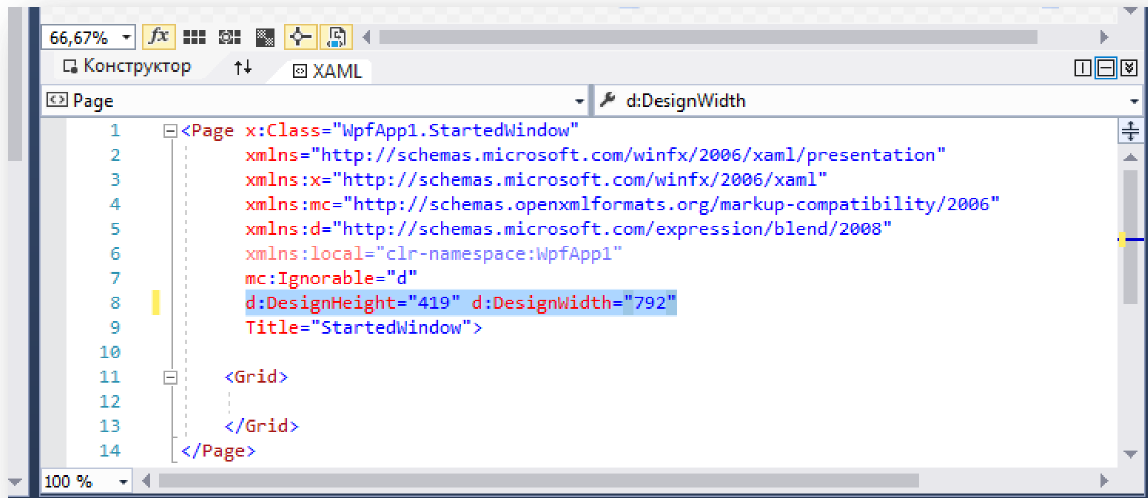
7. Создайте страницу, для этого перейдите по следующему пути: проект - добавить страницу.



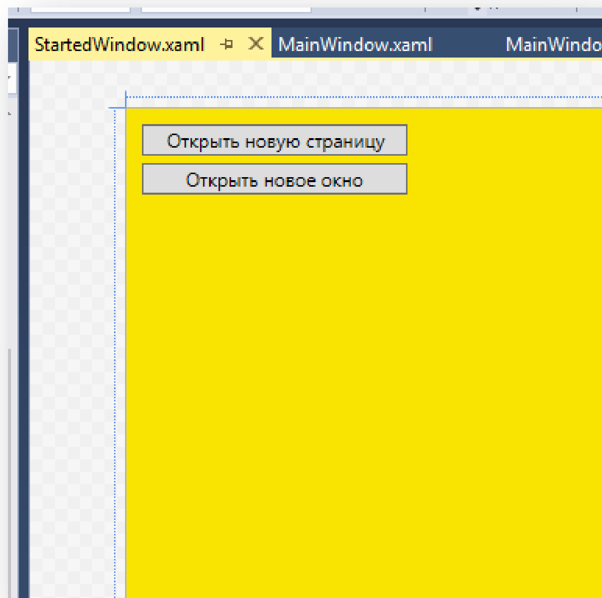
8. Дайте наименование странице и добавьте её в проект.



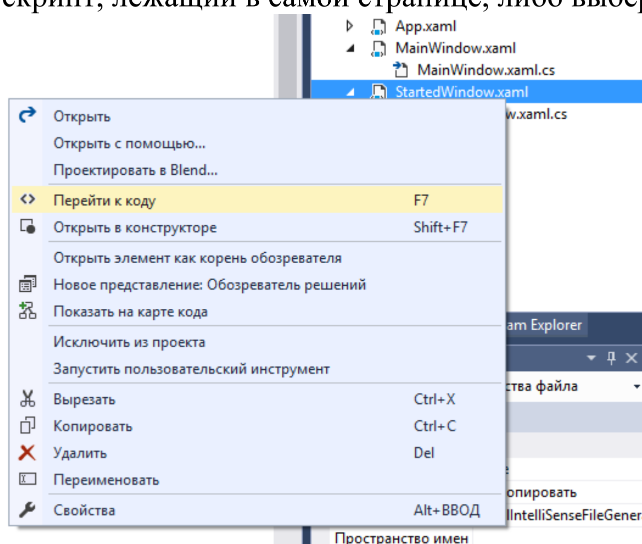
9. Измените атрибуты страницы, присвоив новую высоту и ширину.



10. Измените дизайн страницы, поменяв цвет и добавив несколько кнопок.



11. Перейдите в редактирование программного кода страницы, для этого нажмите два раза на скрипт, лежащий в самой странице, либо выберите пункт перейти к коду.



12. Добавьте ссылку на родителя в коде страницы:

```
public MainWindow mainWindow; // создадим ссылку на родителя
ссылок: 0
public StartedWindow(MainWindow _mainWindow) // при открытии принимаем родителя
{
    InitializeComponent();
    mainWindow = _mainWindow; // сохраняем родителя
}
```

13. Перейдите в программный код, главного окна "MainWindow" и напишите функцию, которая отвечает за открытие страниц.

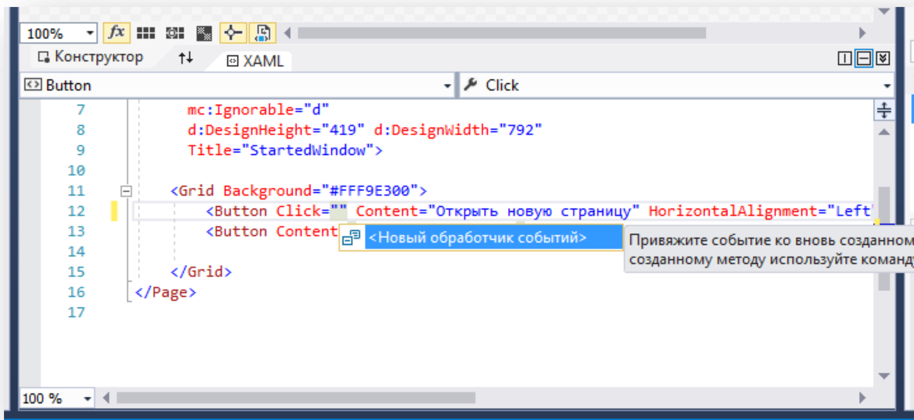
```
ссылок: 0
public void OpenPage(int indexPage) { // принимаем номер страницы
    if(indexPage == 0) // если номер страницы = 0
        frame.Navigate(new StartedWindow(this)); // открываем стартовую страницу
}
```

14. Вызовите эту функцию, после инициализации компонентов.

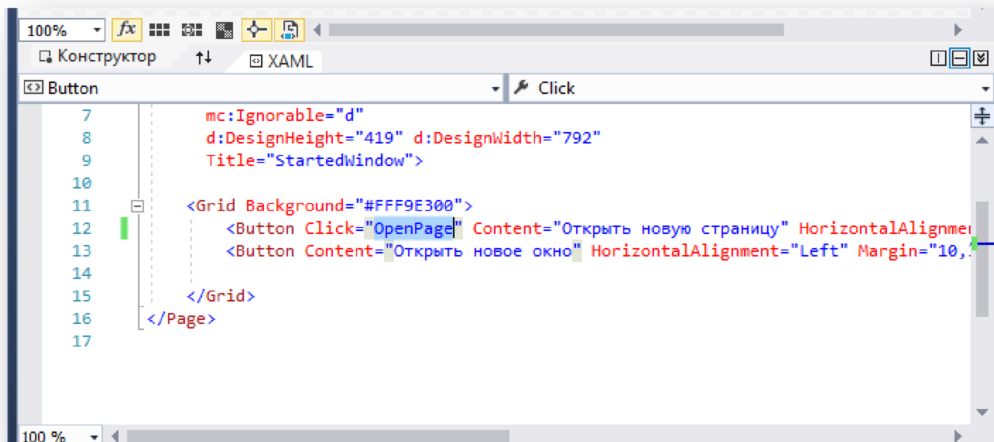
```
ссылок: 0
public MainWindow()
{
    InitializeComponent();
    OpenPage(0); // открываем страницу
}
```

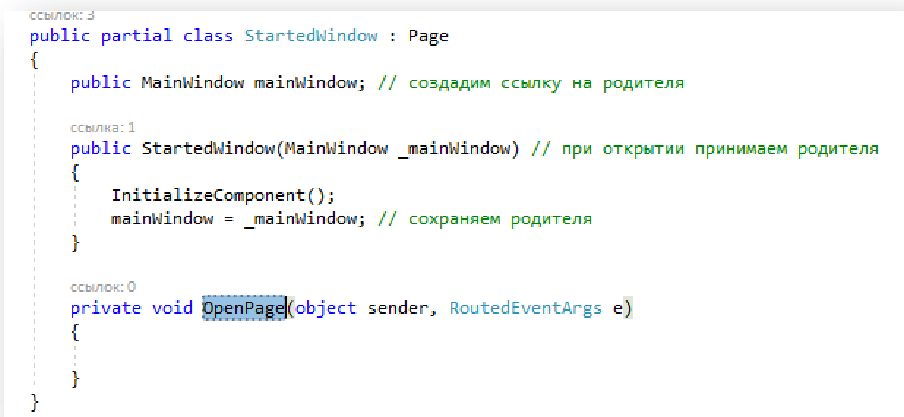
15. При запуске программы, можно увидеть, что происходит открытие страницы.

16. Добавьте кнопке, открывающей интерфейс событие Click, для этого пропишите атрибут Click и выберете создать новый обработчик событий.



17. После чего переименуйте название функции в самой странице и её коде:





18. Пропишите следующий код:

```
public partial class StartedWindow : Page
{
    public MainWindow mainWindow; // создадим ссылку на родителя

    ссылка: 1
    public StartedWindow(MainWindow _mainWindow) // при открытии принимаем родителя
    {
        InitializeComponent();
        mainWindow = _mainWindow; // сохраняем родителя
    }

    ссылок: 0
    private void OpenPage(object sender, RoutedEventArgs e)
    {
        mainWindow.OpenPage(2); // открыть вторую страницу
    }
}
```

19. Создайте вторую страницу:

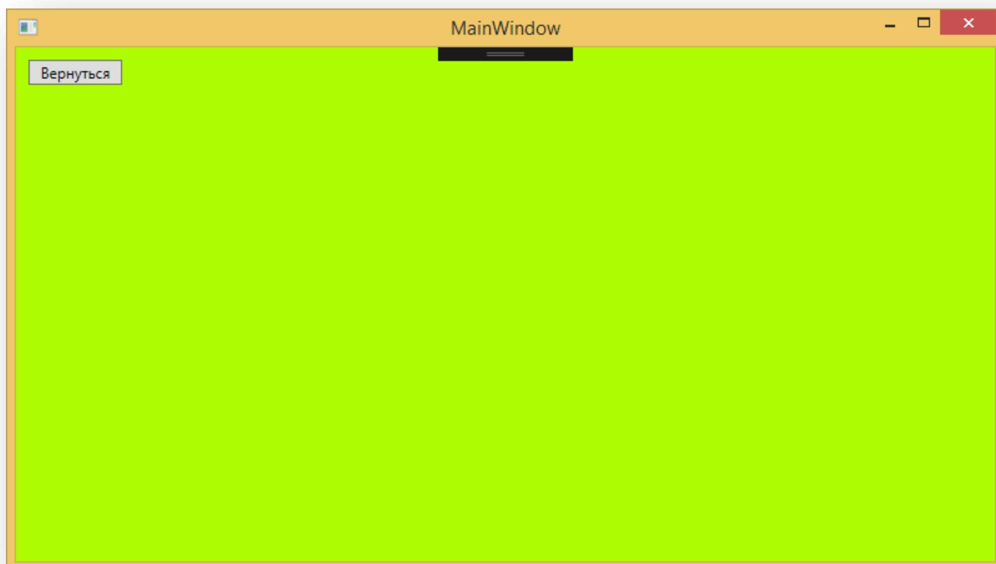
20. Не забудьте реализовать следующий код в новой странице:

```
public partial class LoadPage : Page
{
    public MainWindow mainWindow;
    ссылка: 1
    public LoadPage(MainWindow _mainWindow)
    {
        InitializeComponent();
        mainWindow = _mainWindow;
    }
}
```

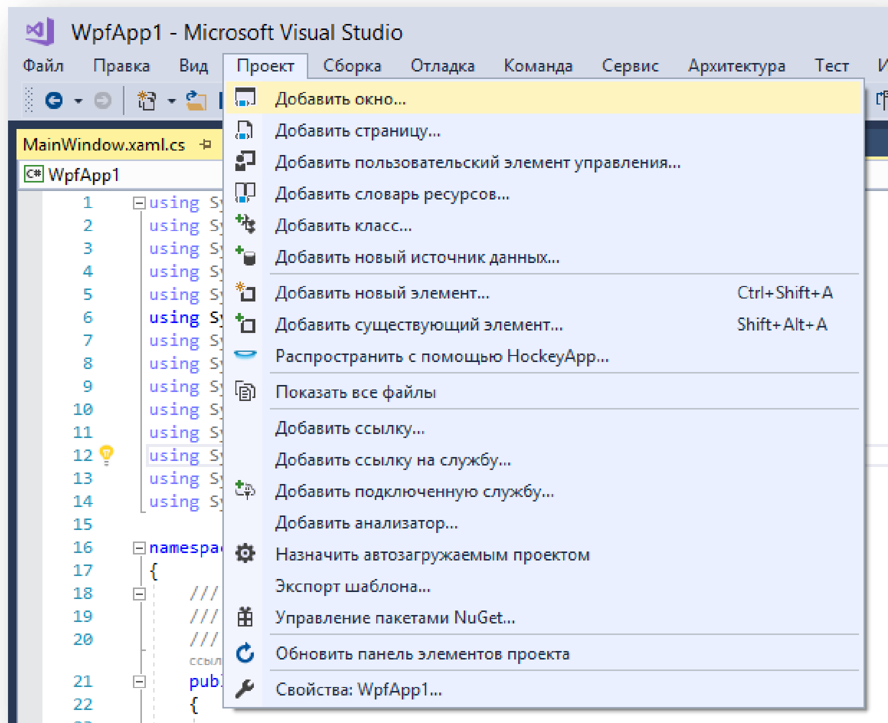
20. Вернувшись на главную форму, допишите открытие второй страницы:

```
ссылок: 2
public void OpenPage(int indexPage) { // принимаем номер страницы
    if(indexPage == 0) // если номер страницы = 0
        frame.Navigate(new StartedWindow(this)); // открываем стартоваю страницу
    else if(indexPage == 2)
        frame.Navigate(new LoadPage(this)); // открываем стартоваю страницу
}
```

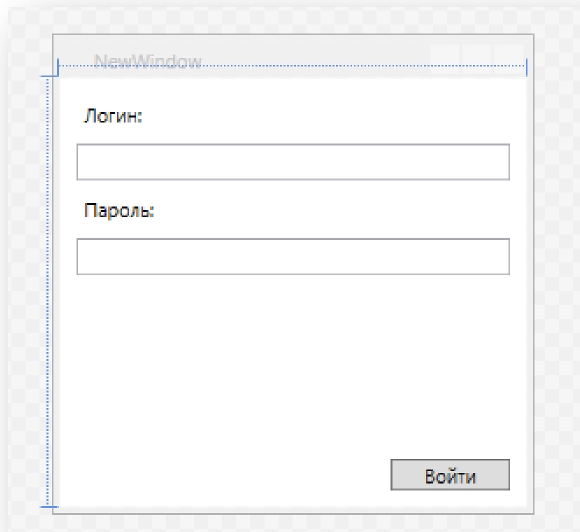
21. Запустите программу, при нажатии на кнопку "Открыть новую страницу" у вас должна открыться новая страница, непосредственно в окне программы:



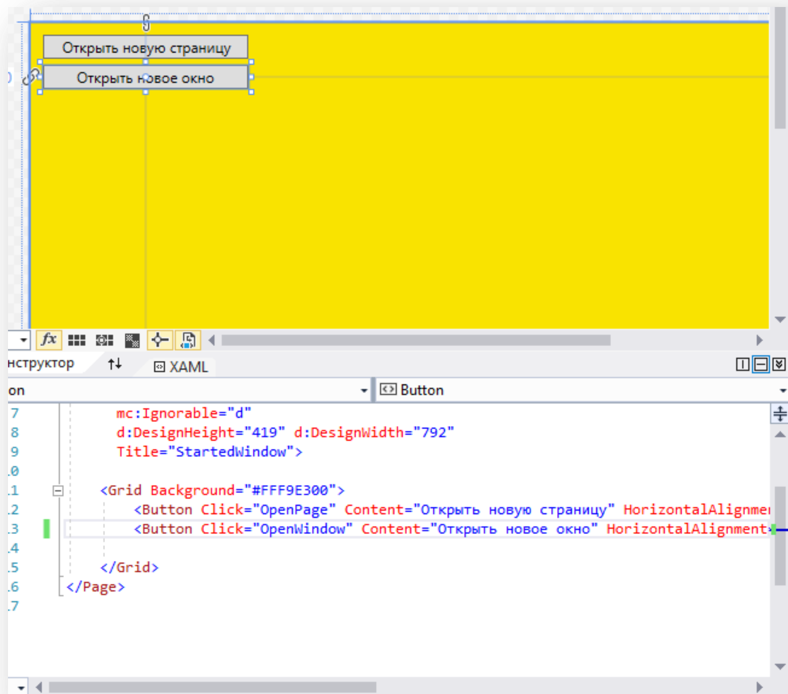
22. Реализуйте возвращение на предыдущую страницу, средством добавления события на кнопку.
23. Добавьте новое окно. Для этого перейдите по следующему пути: Проект - Добавить окно.



24. Создайте интерфейс авторизации:



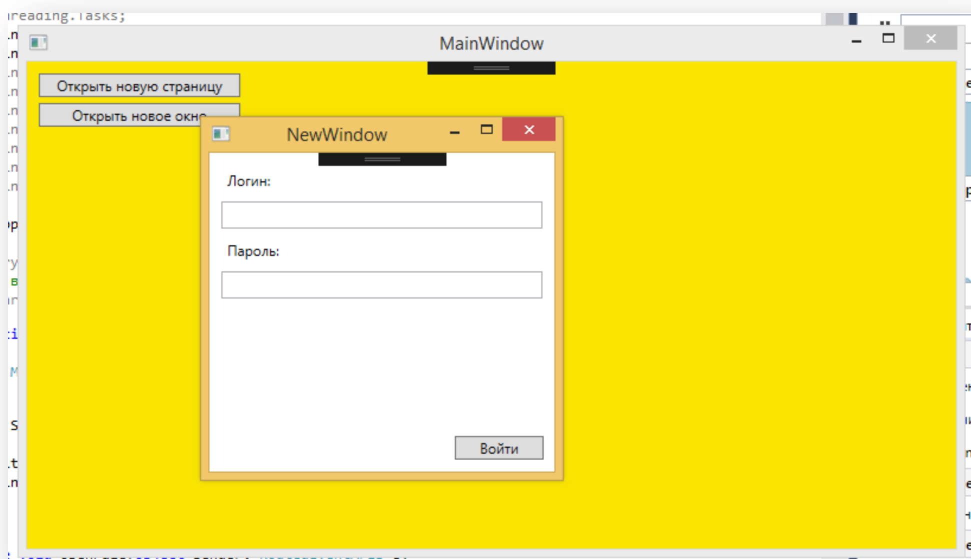
25. Вернитесь на стартовую страницу и добавьте второй кнопке "Открыть новое окно" событие, которое будет открывать новое окно.



26. Пропишите следующий код:

```
ссылок: 0
private void OpenWindow(object sender, RoutedEventArgs e)
{
    new NewWindow().ShowDialog(); // открытие нового окна
}
```

27. При запуске программы, и нажатии на "Открыть новое окно" у вас произойдет непосредственно открытие нового окна.



28. Используя данные алгоритмы, реализуйте интерфейс программы по своей предметной области. Настройте все переходы между страницами и окнами. Старайтесь не использовать новые окна, там где это не нужно.
29. Оформите отчёт, сделайте выводы.

Контрольные вопросы:

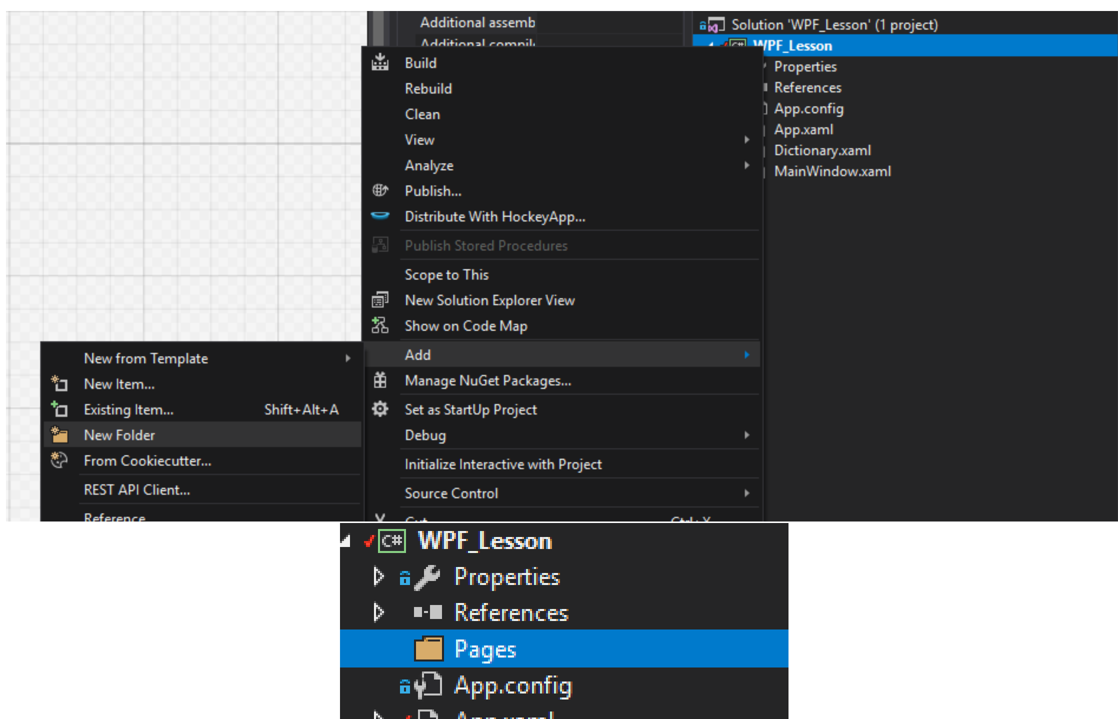
1. В чём заключается отличие Page(страницы) от Window (окна)?
2. Какие элементы вы использовали при выполнении практической работы, и за что они отвечают?
3. Какие конструкции языка программирования вы использовали?

Практическая работа №45

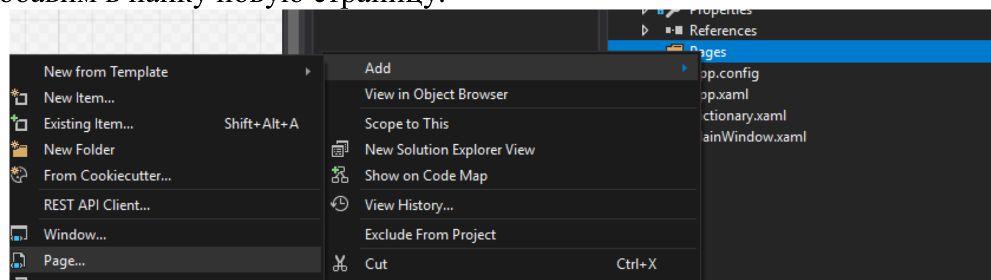
Тема: Разработка формы авторизации (4)

Ход работы:

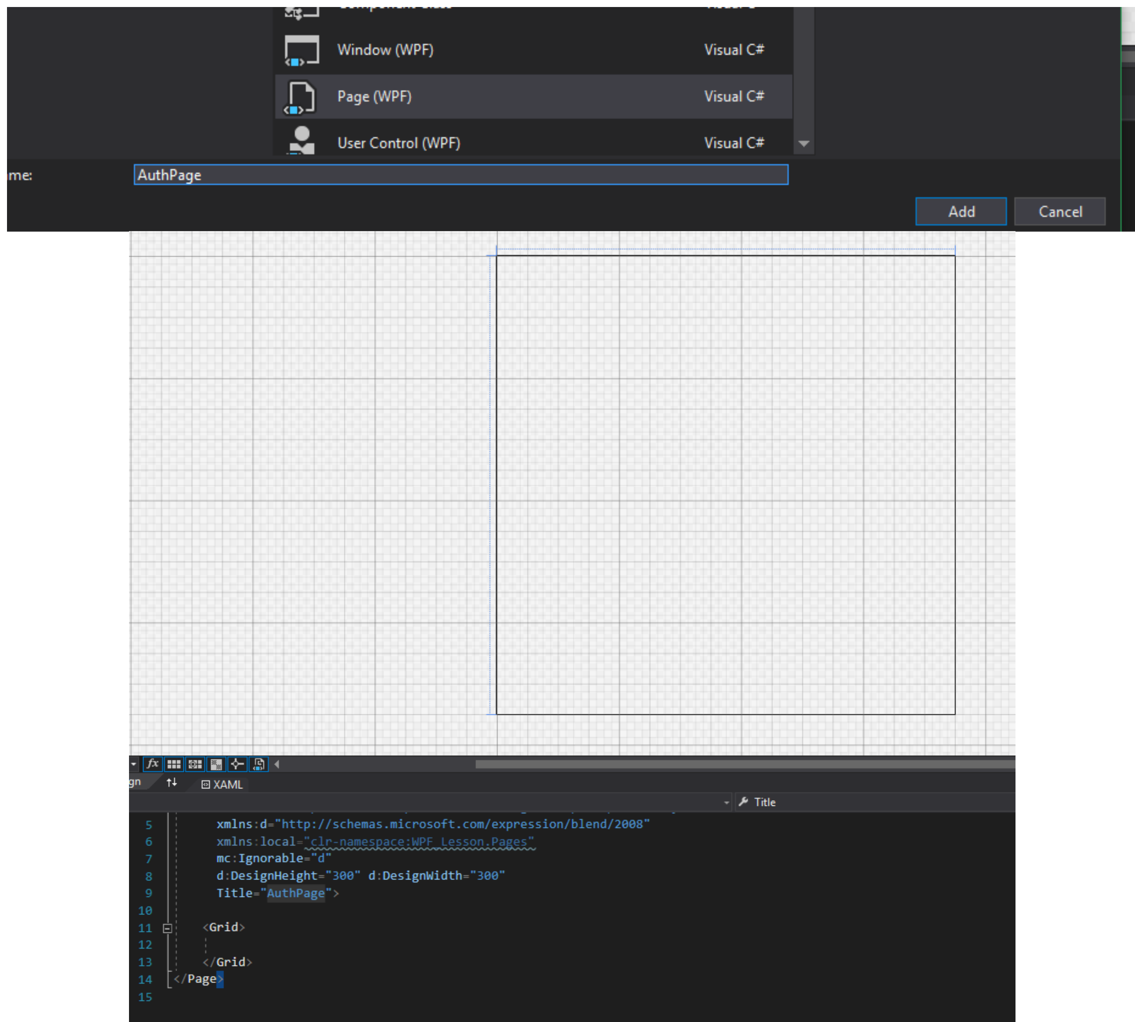
Добавим в проект новую папку и назовем ее «Pages», в этой папке будут находиться страницы (авторизация, регистрация и другие)



Добавим в папку новую страницу.



Назовем страницу AuthPage и создадим ее. После чего появится пустая страница.



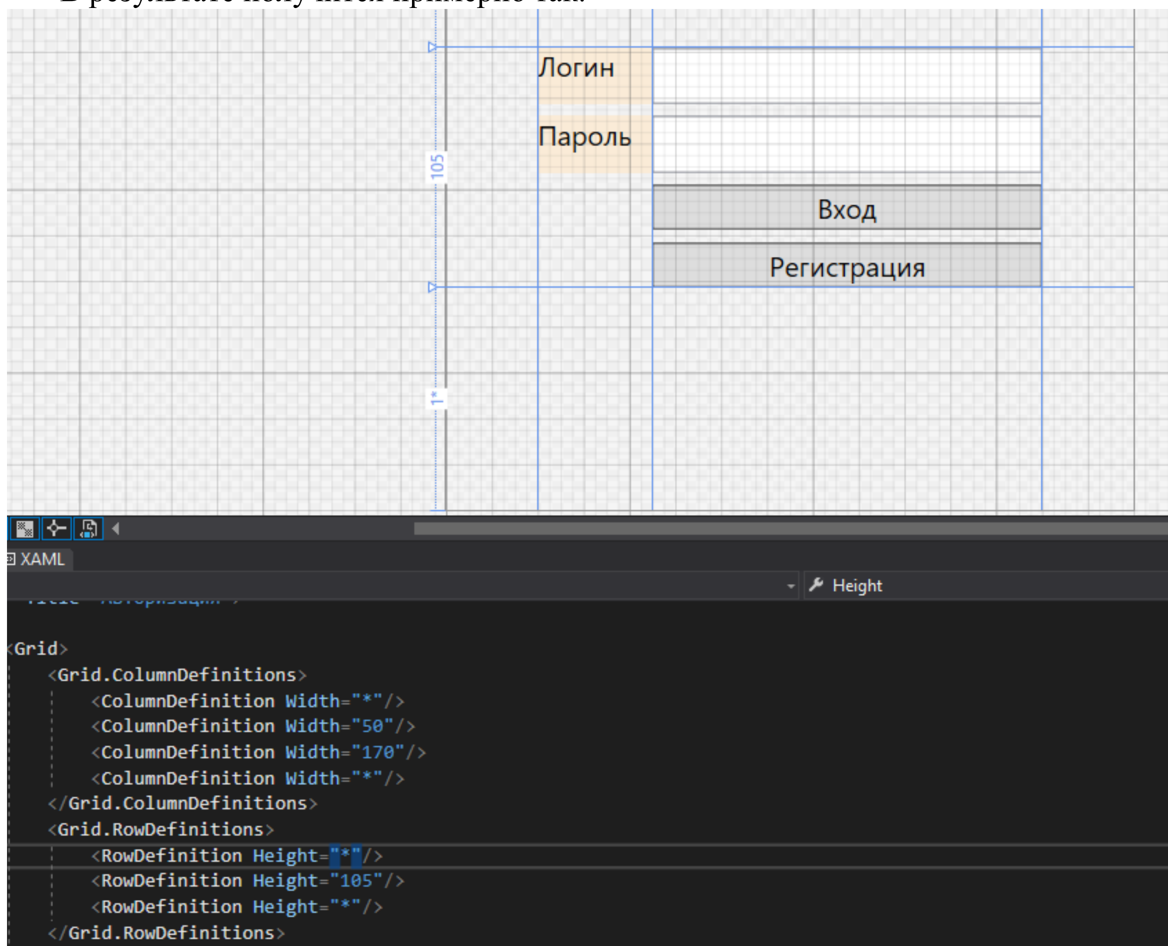
Добавим на форму компоненты «Label» и «TextBox», а затем от центруем их и добавим еще один столбец (для размещения лейблов). Также не забудем переименовать страницу, а затем закрепить лейбл и текстбокс, чтобы они растягивались по ширине.

```
d:DesignHeight="300" d:DesignWidth="300"
Title="Авторизация">

rid>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="50"/>
    <ColumnDefinition Width="170"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="225"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Label Content="Label" Grid.Column="1" Grid.Row="1" VerticalAlignment="Top" Height="25"/>
<TextBox Grid.Column="2" Height="25" Grid.Row="1" TextWrapping="Wrap" Text="TextBox" VerticalAlignment="Top"/>
```

Добавим остальные компоненты по аналогии и переименуем их.

```
    <RowDefinition Height="*"/>
    <RowDefinition Height="225"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Label Content="Логин" Grid.Column="1" Grid.Row="1" VerticalAlignment="Top" Height="25"/>
<TextBox Grid.Column="2" Height="25" Grid.Row="1" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
<Label Content="Пароль" Grid.Column="1" Grid.Row="1" VerticalAlignment="Top" Height="25" Margin="0,30,0,0"/>
<PasswordBox Grid.Column="2" Margin="0,30,0,0" Grid.Row="1" VerticalAlignment="Top" Height="25"/>
<Button Content="Вход" Grid.Column="2" Margin="0,60,0,0" Grid.Row="1" VerticalAlignment="Top"/>
<Button Content="Регистрация" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top"/>
```

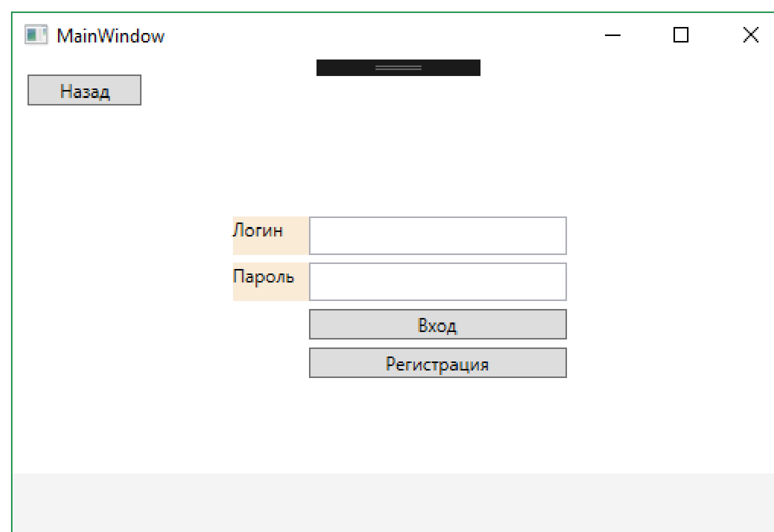
В результате получится примерно так.



Перейдем на главную форму и в компоненте «Frame» укажем в свойстве «Source» нашу страницу «AuthPage.xaml»

```xml
<Window x:Class="WPF_Lesson.MainWindow"
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="40"/>
            <RowDefinition/>
            <RowDefinition Height="42"/>
        </Grid.RowDefinitions>
        <Rectangle Fill="#FFF4F4F5" Grid.Row="2" StrokeThickness="0"/>
        <Button Content="Назад" HorizontalAlignment="Left" Margin="10,10,0,10" Width="75"/>
        <Frame x:Name="MainFrame" Source="Pages/AuthPage.xaml" Grid.Row="1"/>
    </Grid>
</Window>
```

Запустим проект и увидим, что при запуске теперь отображается страница авторизации.



Теперь добавим функционал. Добавим обработчик события на кнопку вход.

```xml
<Button Click="ButtonEnter_OnClick" Content="Вход" Grid.Column="2" Margin="0,60,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<Button Content="Регистрация" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
```

```csharp
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    
}
```

Добавим полям имена

```
    </Grid.RowDefinitions>
    <Label Content="Логин" Grid.Column="1" Grid.Row="1" Ver
    <TextBox x:Name="TextBoxLogin" Grid.Column="2" Height="
    <Label Content="Пароль" Grid.Column="1" Grid.Row="1" Ve
    <PasswordBox x:Name="PasswordBox" Grid.Column="2" Margi
    <Button Click="ButtonEnter_OnClick" Content="Вход" Gric
    <Button Content="Регистрация" Grid.Column="2" Margin="6

</Grid>
ge>
```

Добавим в код базовую проверку

```
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty(PasswordBox.Password))
    {
        MessageBox.Show("Введите логин и пароль!");
        return;
    }
}
```

Добавим запрос к базе данных

```
0 references | 0 changes | 0 authors, 0 changes
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty(PasswordBox.Password))
    {
        MessageBox.Show("Введите логин и пароль!");
        return;
    }

    using (var db = new Entities())
    {
        var user = db.User
            .AsNoTracking()
            .FirstOrDefault(u => u.Login == TextBoxLogin.Text && u.Password == PasswordBox.Password);

        if (user == null)
        {
            MessageBox.Show("Пользователь с такими данными не найден!");
            return;
        }
    }
}
```

И теперь добавим переходы в зависимости от роли на меню пользователя (для этого необходимо создать страницы меню для каждого типа пользователя, CustomerMenu или DirectorMenu и тд)

```csharp
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty(PasswordBox.Password))
    {
        MessageBox.Show("Введите логин и пароль!");
        return;
    }

    using (var db = new Entities())
    {
        var user = db.User
            .AsNoTracking()
            .FirstOrDefault(u => u.Login == TextBoxLogin.Text && u.Password == PasswordBox.Password);

        if (user == null)
        {
            MessageBox.Show("Пользователь с такими данными не найден!");
            return;
        }

        MessageBox.Show("Пользователь успешно найден!");
        // Переход на меню пользователя в зависимости от роли

        switch (user.Role)
        {
            case "Заказчик":
                NavigationService?.Navigate(new Menu());
                break;
            case "Директор":
                NavigationService?.Navigate(new Menu());
                break;
        }
    }
}
```

Практическая работа №46

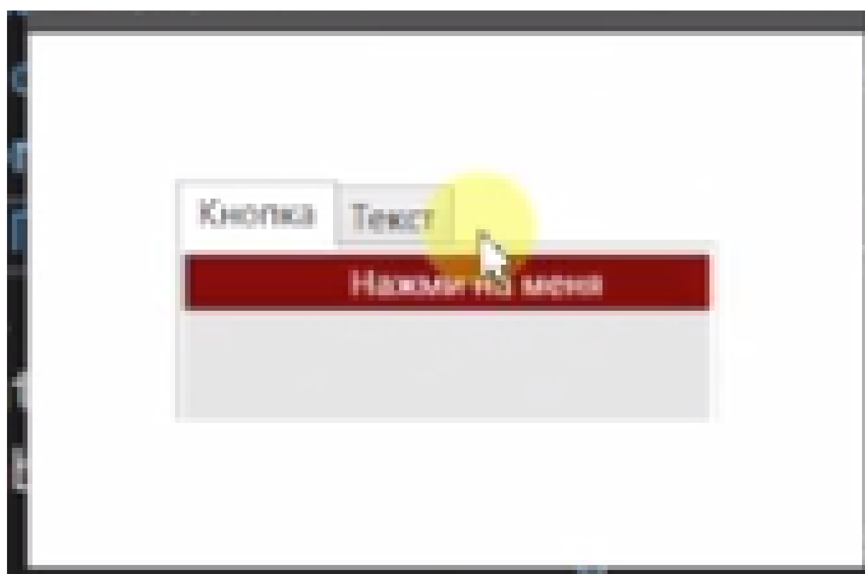
Тема: Создание страниц на форме (2)

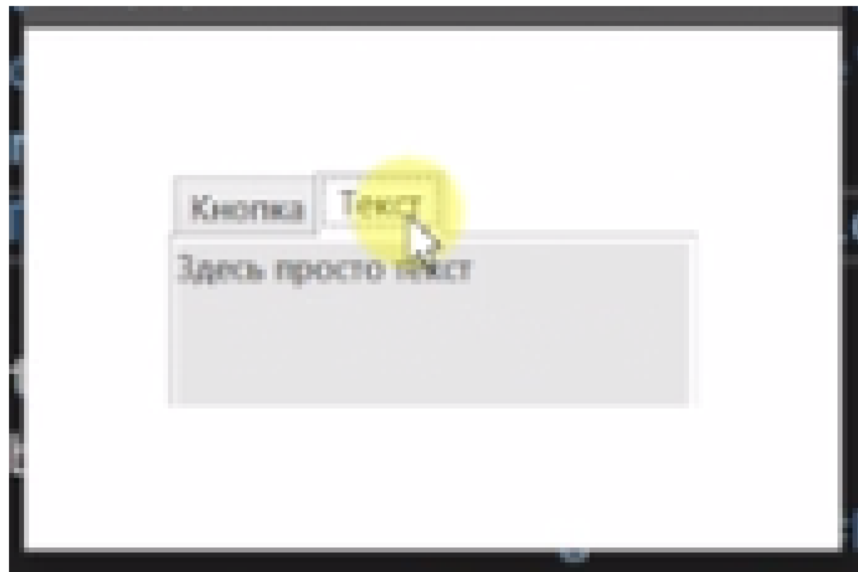
Ход работы:

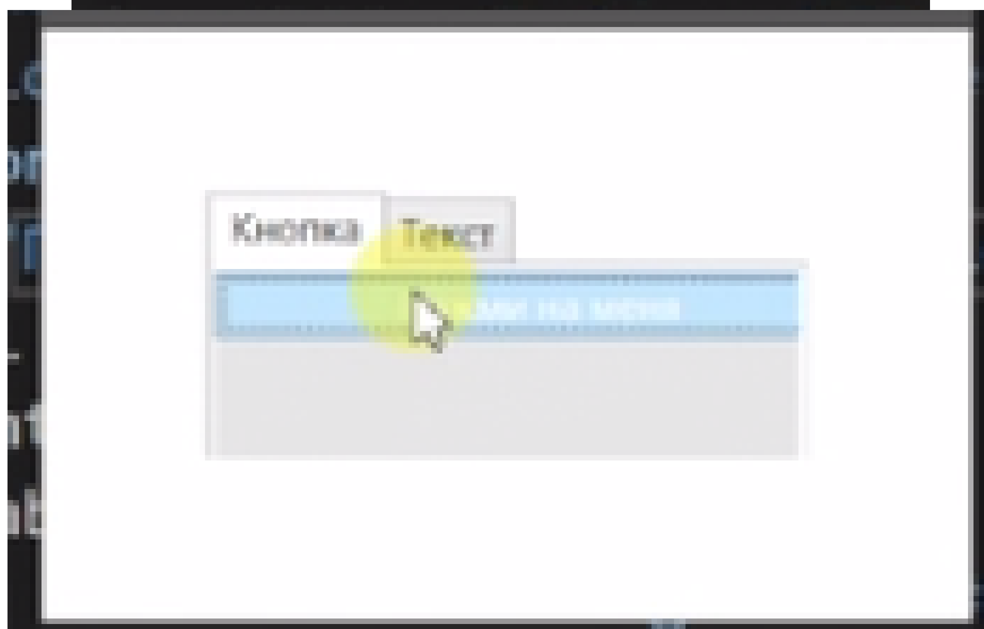
Создайте WPF проект и добавьте в него объект TabControl. Внутри объекта укажите две вкладки с названиями: «Кнопка» и «Текст».

При нажатии на кнопку в 1 вкладке вы должны скрывать/показывать текстовую надпись во второй вкладке.

Пример реализации проекта:




Кнопка
Текст
Здесь просто текст


Кнопка
Текст

Кнопка
Текст

Для реализации задания сперва пропишите следующий XAML код:

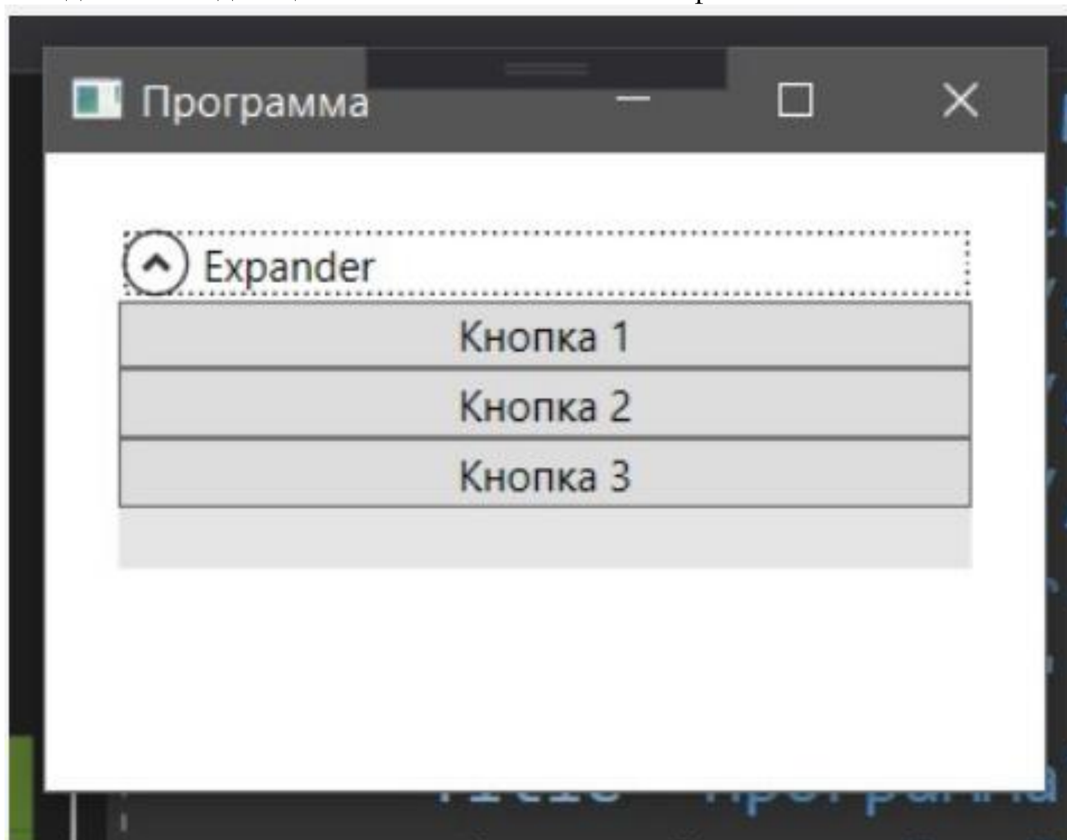
```
<Window x:Class="SImple.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:SImple"
        mc:Ignorable="d"
        Title="Программа" Height="220" Width="300">
    <StackPanel Margin="50">
        <TabControl Height="100">
            <TabItem Header="Кнопка">
                <StackPanel Background="#FFE5E5E5">
                    <Button Background="DarkRed" Foreground="White" HorizontalAlign-
ment="Left" Content="Нажми на меня" Width="200" Click="Button_Click" />
                </StackPanel>
            </TabItem>
            <TabItem Header="Текст">
                <StackPanel Background="#FFE5E5E5">
                    <TextBlock Text="Здесь просто текст" x:Name="simpleLabel" />
                </StackPanel>
            </TabItem>
        </TabControl>
    </StackPanel>
</Window>
```

Далее пропишите код для реализации функционала:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
                // Проверяем состояние текста
                // Если он сейчас виден, то будем скрывать его
        if (simpleLabel.Visibility == Visibility.Visible)
            simpleLabel.Visibility = Visibility.Hidden;
        else
            simpleLabel.Visibility = Visibility.Visible;
    }
}
```

Выпадающий список

Создайте выпадающий список в точности как на фото ниже:



Для реализации задания пропишите следующий XAML код:

```
<Window x:Class="SImple.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:SImple"
        mc:Ignorable="d"
        Title="Программа" Height="220" Width="300">
    <StackPanel Margin="20">
        <Expander Header="Expander" Height="100">
            <StackPanel Background="#FFE5E5E5">
                <Button Content="Кнопка 1" />
                <Button Content="Кнопка 2" />
                <Button Content="Кнопка 3" />
            </StackPanel>
        </Expander>

    </StackPanel>
</Window>
```

Практическая работа №47

Тема: Реализация алгоритмов обработки числовых значений (2)

Цель: отработать навыки составления алгоритмов с использованием циклов. Научиться реализовывать типичные алгоритмы обработки числовых данных в приложениях .Net WPF

Задание.

1. Напишите программу печати таблицы перевода расстояний из дюймов в сантиметры для значений длин от 1 до 20 дюймов. 1 дюйм = 2,54 см. (10 баллов)
2. Напишите программу вывода всех четных чисел от 2 до 100 включительно. (10 баллов)
3. Составьте программу, вычисляющую сумму квадратов всех чисел от 1 до N. (10 баллов)
4. Напишите программу, определяющую сумму всех нечетных чисел от 1 до 99. (10 баллов)
5. Составьте программу получения в порядке убывания всех делителей данного числа. (20 баллов)
6. Составьте программу определения наибольшего общего делителя двух натуральных чисел. (20 баллов)
7. Составьте программу определения наименьшего общего кратного двух натуральных чисел. (30 баллов)
8. Составьте программу, подсчитывающую количество цифр вводимого вами целого неотрицательного числа. Можно использовать операцию целочисленного деления. (20 баллов)
9. Составьте программу, определяющую максимальное из всех вводимых вами чисел. Концом ввода чисел является введенное число 0. (30 баллов)
10. Найти наибольшее и наименьшее значение функции y=3x2+x-4, если на заданном интервале [a,b] X изменяется с шагом 0,1. (30 баллов а)
11. Вычислите сумму квадратов N четных натуральных чисел. (10 баллов)
12. Вычислить: 1+2+4+8+…+216 (10 баллов)
13. Вычислить: (1+2)*(1+2+3)*…*(1+2+…+10) (30 баллов)
14. В бригаде, работающей на уборке сена, имеется N косилок. Первая из них работала M часов, а каждая следующая на 10 минут больше, чем предыдущая. Сколько часов проработала вся бригада? (20 баллов)
15. Билет называют «счастливым», если в его номере сумма первых трех цифр равна сумме последних трех. Подсчитать число тех «счастливых» билетов, у которых сумма трех цифр равна 13. Номер билета может быть от 000000 до 999999. (40 баллов)
16. В ЭВМ вводятся по очереди координаты N точек. Определить, сколько из них попадает в круг радиусом R с центром в точке (a,b). (20 баллов)
17. В ЭВМ вводятся по очереди данные о росте N учащихся класса. Определить средний рост учащихся в классе. (10 баллов)
18. Составьте программу, суммирующую штрафное время команд при игре в хоккей. Выводить на экран суммарное штрафное время обеих команд после любого его изменения. После окончания игры выдать итоговое сообщение. (20 баллов)
19. Составьте программу вычисления степени числа A с натуральным показателем N. Записать варианты программы со всеми видами циклов. (30 баллов)
20. Составьте программу, вычисляющую A*B, не пользуясь операцией умножения. A и B любое натуральное число. (10 баллов).
21. В 1202г. Итальянский математик Леонард Пизанский (Фибоначчи) предложил такую задачу: пара кроликов каждый месяц дает приплод – двух кроликов (самца и самку), от которых через два месяца уже получается новый приплод, Сколько кроликов будет через год, если в начале года имелась одна пара? Согласно условию задачи числа, соответствующие количеству кроликов, которые появляются через каждый месяц, составляют по-

следовательность 1, 1, 2, 3, 5, 8, 13, 21, 37, … Составьте программу, позволяющую найти все числа Фибоначчи, меньшие заданного числа N. (50 баллов).

22. Составьте программу, которая выводит полную запись десятичного числа 42*4*, в которой пропущены две цифры (обозначены *), если известно, что данное число кратно 72. (40 баллов).

23. В старояпонском календаре был принят 60-летний цикл, состоявший из пяти 12-летних подциклов. Подциклы обозначались названиями цвета: зеленый, красный, желтый, белый и черный. Внутри каждого подцикла годы носили названия животных: крысы, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки и свиньи. 1984 – год зеленой крысы – был началом очередного цикла. Напишите программу, которая вводит номер некоторого года нашей эры и печатает его название по старояпонскому календарю. (40 баллов).

24. Составьте программу, которая по введенному вами числу N (от 1 до 100) напечатает все натуральные числа X русскими буквами (двадцать, сорок пять и т.д.). Например, для числа 4 программа должная напечатать «один», так как в слове «один» четыре буквы. Для числа 9 программа должная напечатать «сорок пять», так как в записи числа «сорок пять» девять букв. (50 баллов).

25. Запишите любое число от 1 до 1000, введенное пользователем, русскими буквами. Например, 2 – два, 150 – сто пятьдесят. (50 баллов).

Критерии оценки:

- Оценка «отлично» более 100 баллов
- Оценка «хорошо» более 80 баллов
- Оценка «удовлетворительно» более 70 баллов

1. Прокомментируйте код.
2. Оформите отчёт
3. Сделайте выводы!

Контрольные вопросы:

1. Какие основные конструкции языка вы использовали и для чего они служат?
2. Что необходимо проводить при создании программы?
3. Как производить ввод и вывод данных в программе? При помощи, каких элементов?

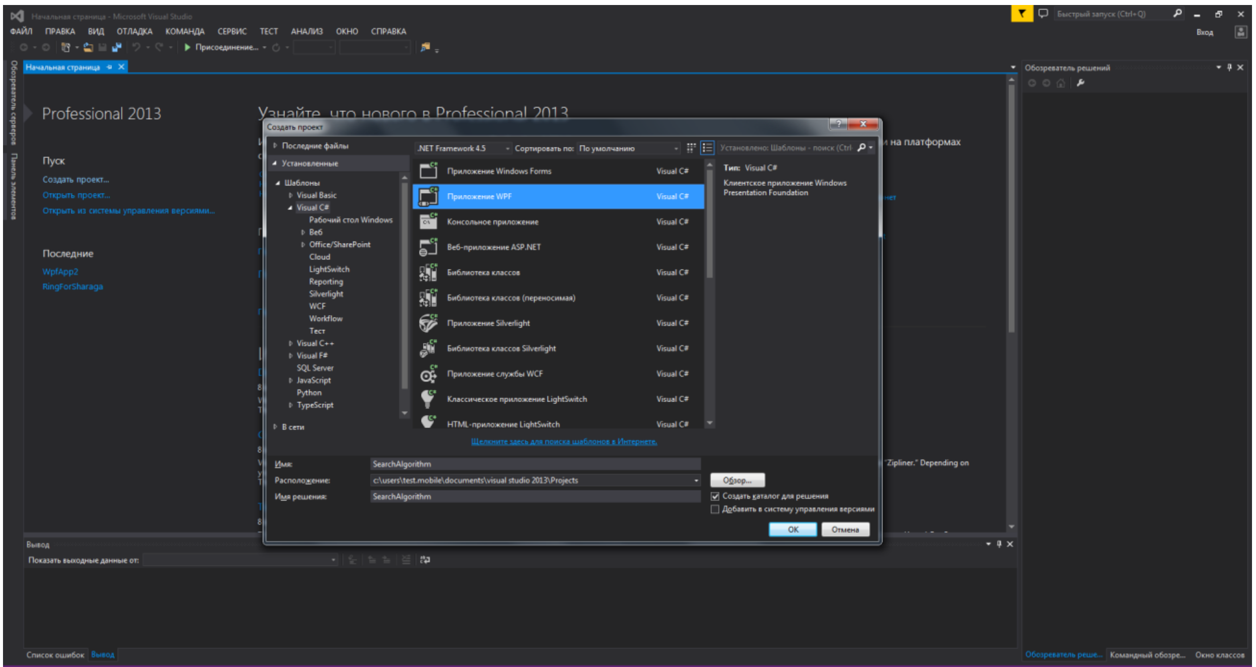
Практическая работа №48

Тема: Реализация алгоритмов поиска. Отладка приложений (2)

Цель работы: изучить и реализовать основные алгоритмы поиска. Познакомиться с элементами WPF-приложений. Приобрести практические навыки в программировании на языке C#.

Ход работы:

1. Открыть приложение Microsoft Visual Studio.
2. Создать Приложение WPF - Visual C#.



3. В окне XAML – разметки указать минимальные значения высоты и ширины, а также значения по умолчанию.

```
<Window x:Class="SearchAlgorithm.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Реализация алгоритмов поиска. " Height="450" Width="800" MinWidth="800" MinHeight="400">
    <Grid>

    </Grid>
</Window>
```

4. Реализовать следующий интерфейс, в программе при помощи таких элементов Label, ComboBox, TextBox, ListView, Button.

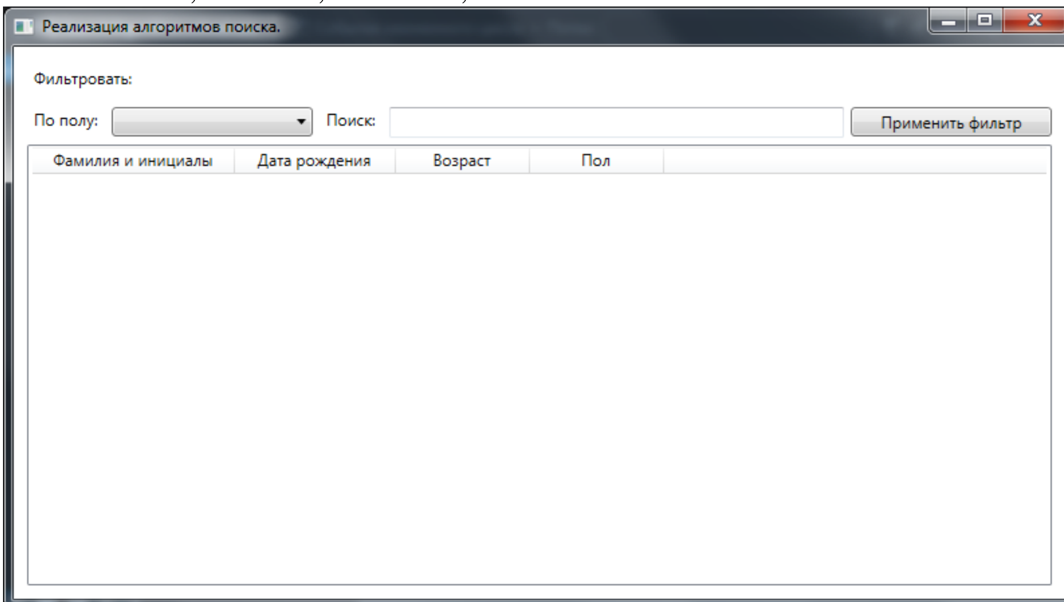


Рисунок 1: Интерфейс программы

```
<Grid>
    <Label Content="Фильтровать:" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>
    <Label Content="По полу:" HorizontalAlignment="Left" Margin="10,41,0,0" VerticalAlignment="Top"/>
    <ComboBox HorizontalAlignment="Left" Margin="73,45,0,0" VerticalAlignment="Top" Width="150"/>
    <Label Content="Поиск:" HorizontalAlignment="Left" Margin="228,41,0,0" VerticalAlignment="Top"/>
    <TextBox Height="23" Margin="280,45,165,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
    <Button Content="Применить фильтр" HorizontalAlignment="Right" Margin="0,45,10,0" VerticalAlignment="Top" Width="150"/>
    <ListView Margin="10,73,10,10">
        <ListView.View>
            <GridView>
                <GridViewColumn Header="Фамилия и инициалы" Width="150"/>
                <GridViewColumn Header="Дата рождения" Width="120"/>
                <GridViewColumn Header="Возраст" Width="100"/>
                <GridViewColumn Header="Пол" Width="100"/>
            </GridView>
        </ListView.View>
    </ListView>
```

Рисунок 2: XAML-код программы

5. Присвоить атрибут DisplayMemberBinding для тегов GridViewColumn, дав наименования в соответствии с рисунком ниже.

```
<ListView Margin="10,73,10,10">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Фамилия и инициалы" Width="150" DisplayMemberBinding="{Binding name}"/>
            <GridViewColumn Header="Дата рождения" Width="120" DisplayMemberBinding="{Binding dataOfBirth}"/>
            <GridViewColumn Header="Возраст" Width="100" DisplayMemberBinding="{Binding age}"/>
            <GridViewColumn Header="Пол" Width="100" DisplayMemberBinding="{Binding gender}"/>
        </GridView>
    </ListView.View>
</ListView>
```

6. Для элемента ComboBox задать атрибут x:Name="genderFilter", а для TextBox атрибут x:Name="nameFilter", для ListView – атрибут x:Name="userList".

7. Для элемента Button создать обработчик события на активность Click.

```
<ComboBox x:Name="genderFilter" HorizontalAlignment="Left" Margin="73,45,0,0" VerticalAlignment="Top" Width="150"/>
<Label Content="Поиск:" HorizontalAlignment="Left" Margin="228,41,0,0" VerticalAlignment="Top"/>
<TextBox x:Name="nameFilter" Height="23" Margin="280,45,165,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
<Button Click="ActiveFilter" Content="Применить фильтр" HorizontalAlignment="Right" Margin="0,45,10,0" VerticalAlignment="Top" Width="150"/>
<ListView Margin="10,73,10,10">
    <ListView.View>
        <GridView>
```

8. Создать класс объектов, хранящий в себе такие переменные как name, dataOfBirth, age, gender и реализовать функцию, отвечающую за заполнение этих данных.

```
public partial class MainWindow : Window
{
    public class User {
        public string name {get; set; }
        public string dataOfBirth { get; set; }
        public string age { get; set; }
        public string gender { get; set; }

        public User(string _name, string _dataOfBirth, string _age, string _gender) {
            this.name = _name;
            this.dataOfBirth = _dataOfBirth;
            this.age = _age;
            this.gender = _gender;
        }
    }

    public MainWindow()
    {
        InitializeComponent();
    }

    private void ActiveFilter(object sender, RoutedEventArgs e)
    {

    }
}
```

9. Создать массив этого класса: public List<User> user = new List<User>();

10. Создать функцию LoadUser, принимающую в качестве аргумента массив класса User и вызвать её после инициализации компонентов, отправив созданный массив класса user.

```
public partial class MainWindow : Window
{
    public class User {
        public string name {get; set; }
        public string dataOfBirth { get; set; }
        public string age { get; set; }
        public string gender { get; set; }

        public User(string _name, string _dataOfBirth, string _age, string _gender) {
            this.name = _name;
            this.dataOfBirth = _dataOfBirth;
            this.age = _age;
            this.gender = _gender;
        }
    }

    public List<User> user = new List<User>();

    public MainWindow()
    {
        InitializeComponent();

        LoadUser(user);
    }

    public void LoadUser(List<User> _user) {

    }

    private void ActiveFilter(object sender, RoutedEventArgs e)
    {

    }
}
```

11. Перед вызовом функции LoadUser произвести добавление элемента в массив класса user: user.Add(new User("Каримов А.О.", "27.04.1996", "23", "М"));

12. Добавить как минимум восемь пользователей системы.

```
public MainWindow()
{
    InitializeComponent();

    user.Add(new User("Каримов А.О.", "27.04.1996", "23", "М"));
    user.Add(new User("Шишкин К.А.", "25.02.1998", "21", "М"));
    user.Add(new User("Кучукбаева Л.А.", "18.02.1999", "20", "F"));
    user.Add(new User("Белов А.В.", "25.02.1997", "22", "М"));
    user.Add(new User("Хоробрых Г.Д.", "25.02.1996", "23", "М"));
    user.Add(new User("Юкович Н.Т.", "25.02.1995", "22", "М"));
    user.Add(new User("Власов А.А.", "25.02.1994", "25", "М"));
    user.Add(new User("Теплоухов Н.С.", "25.02.1993", "26", "М"));
    LoadUser(user);
}
```

13. Реализовать функцию вывода данных на экран.

```
public void LoadUser(List<User> _user) {
    userList.Items.Clear();

    for (int i = 0; i < _user.Count; i++) {
        userList.Items.Add(_user[i]);
    }
}
```

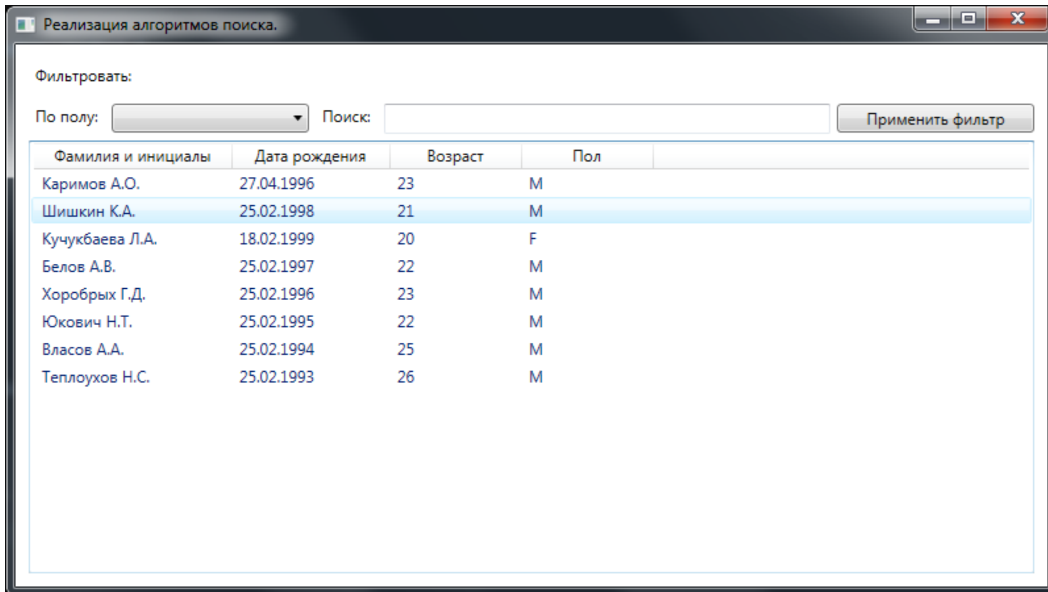


Рисунок 3: Результат выполнения программы

14. Добавить в элемент ComboBox, два TextBlock'а, задав значения Мужской и Женский.

```
<Label Content="Фильтровать:" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>
<Label Content="По полу:" HorizontalAlignment="Left" Margin="10,41,0,0" VerticalAlignment="Top"/>
<ComboBox x:Name="genderFilter" HorizontalAlignment="Left" Margin="73,45,0,0" VerticalAlignment="Top" Width="150">
    <TextBlock>Мужской</TextBlock>
    <TextBlock>Женский</TextBlock>
</ComboBox>
<Label Content="Поиск:" HorizontalAlignment="Left" Margin="228,41,0,0" VerticalAlignment="Top"/>
<TextBox x:Name="nameFilter" Height="23" Margin="280,45,165,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
<Button Click="ActiveFilter" Content="Применить фильтр" HorizontalAlignment="Right" Margin= FrameworkElement.VerticalAlignme
<ListView x:Name="userList" Margin="10,73,10,10">
    <ListView.View>
```

15. Реализовать функцию фильтрации по признаку пола: для этого необходимо создать новый массив класса User, далее в зависимости от выбранного значения произвести поиск в массиве с определённым условием:

```
private void ActiveFilter(object sender, RoutedEventArgs e)
{
    List<User> newUsers = new List<User>();

    if (genderFilter.SelectedIndex == 0)
    {
        newUsers = user.FindAll(x => x.gender == "M");
    }
    else
    {
        newUsers = user.FindAll(x => x.gender == "F");
    }

    LoadUser(newUsers);
}
```

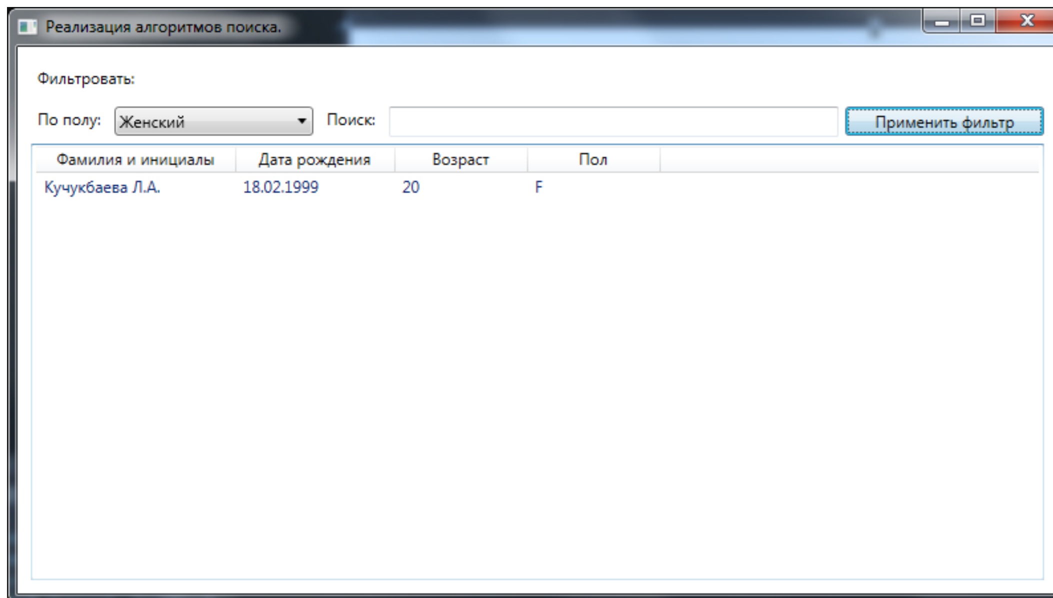


Рисунок 4: Сортировка по женскому полу

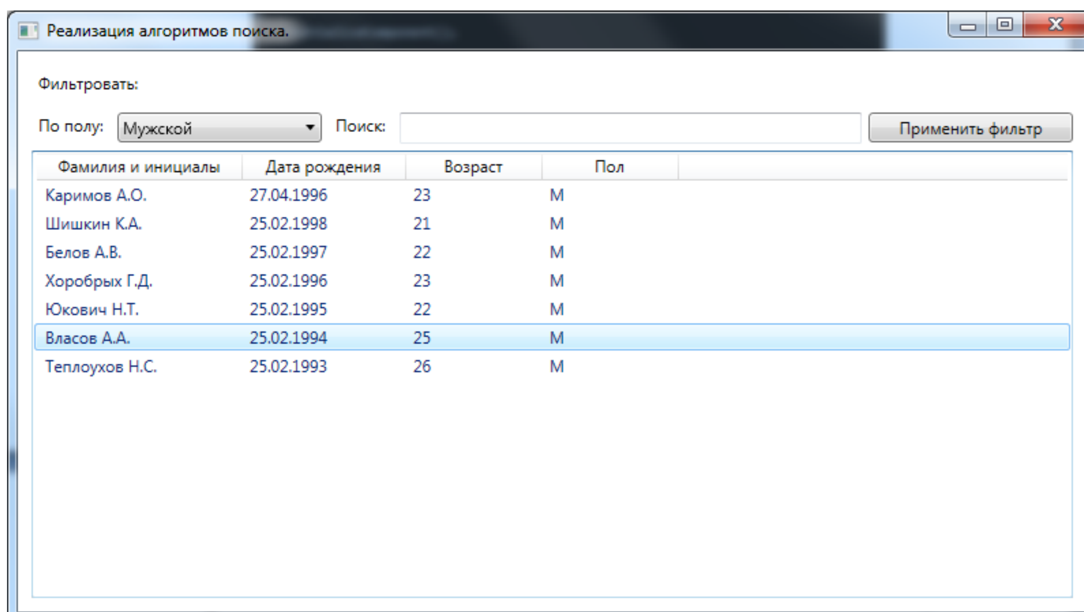


Рисунок 5: Сортировка по мужскому полу

16. После сортировки по половому фильтру необходимо добавить алгоритм кода, позволяющий производить точное совпадение, со строкой находящейся в TextBox:

```
private void ActiveFilter(object sender, RoutedEventArgs e)
{
    List<User> newUsers = new List<User>();

    if (genderFilter.SelectedIndex == 0)
    {
        newUsers = user.FindAll(x => x.gender == "M");
    }
    else
    {
        newUsers = user.FindAll(x => x.gender == "F");
    }

    newUsers = newUsers.FindAll(x => x.name.Contains(nameFilter.Text));

    LoadUser(newUsers);
}
```

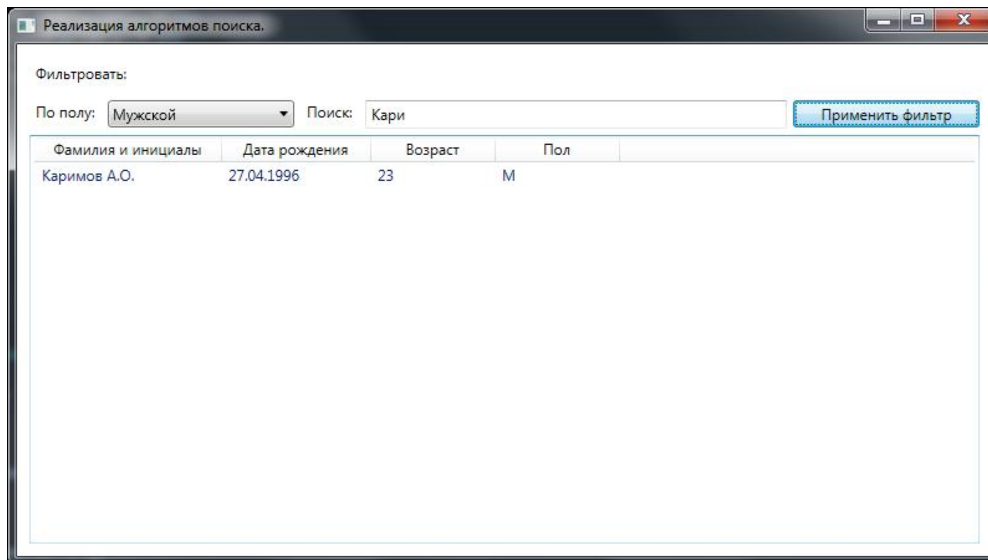


Рисунок 6: Результат выполнения программы

17. Самостоятельно прокомментируйте каждую строку кода.
18. Оформите отчёт, сделайте выводы.

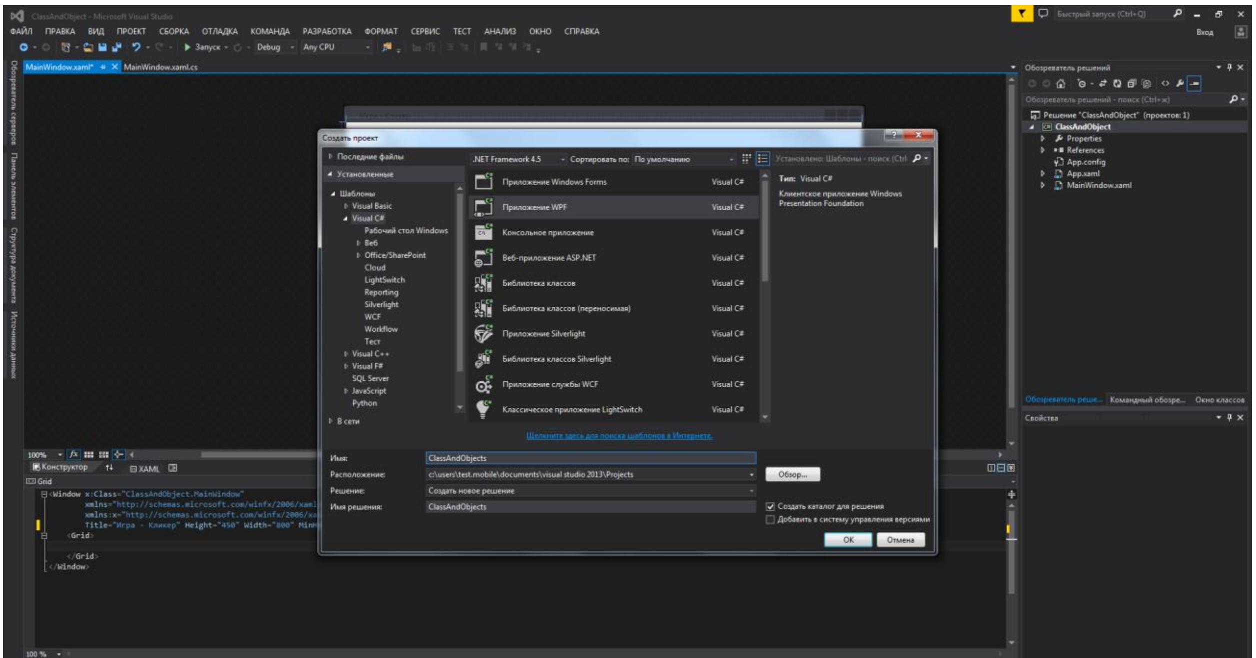
Для получения оценки «хорошо» - доработайте форму так, чтобы можно было также производить сортировку по месяцам рождения. Оформите дизайн.
Для получения оценки «отлично» - доработайте форму так, чтобы можно было производить сортировку по месяцам рождения, а так же добавлять, изменять и удалять новых пользователей. Оформите дизайн.

Контрольные вопросы:
1. Какие основные элементы вы использовали при выполнении практики, для чего они нужны?
2. Как производится фильтрация и поиск данных?
3. Какие языковые конструкции вы для этого использовали?

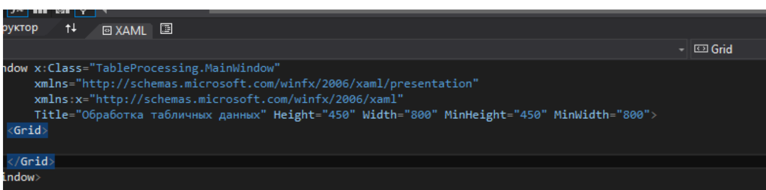
Практическая работа №49

Тема: Реализация обработки табличных данных (2)
Цель работы: Познакомиться с реализацией обработки табличных данных в программе WPF. Приобрести практические навыки в программировании на языке C#. Познакомиться с методами взаимодействия Microsoft Access.

Ход работы:
6. Открыть приложение Microsoft Visual Studio.
7. Создать Приложение WPF - Visual C#.



3. В окне XAML-разметки установить минимальные значения высоты и ширины окна, а также текущее разрешение.

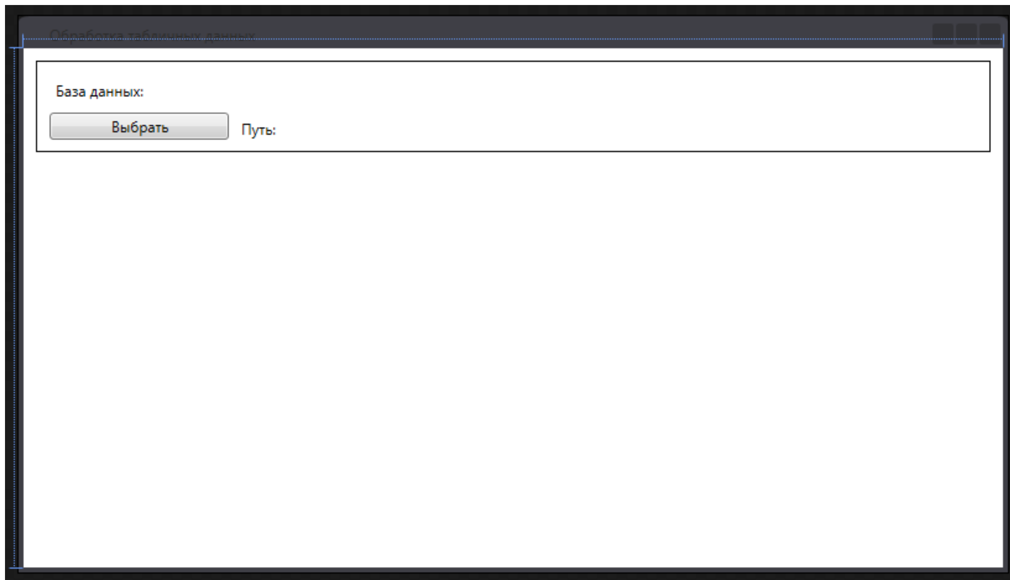


4. На рабочем столе создать базу данных Microsoft Access, и добавить в неё таблицу со следующими полями:

users

Имя поля	Тип данных
id	Счетчик
firstName	Текстовый
lastName	Текстовый
series	Текстовый
number	Текстовый
issuedBy	Текстовый
dateOfBirth	Текстовый
mail	Текстовый

5. Создать следующий интерфейс использовав такие элементы как: Label и Button.

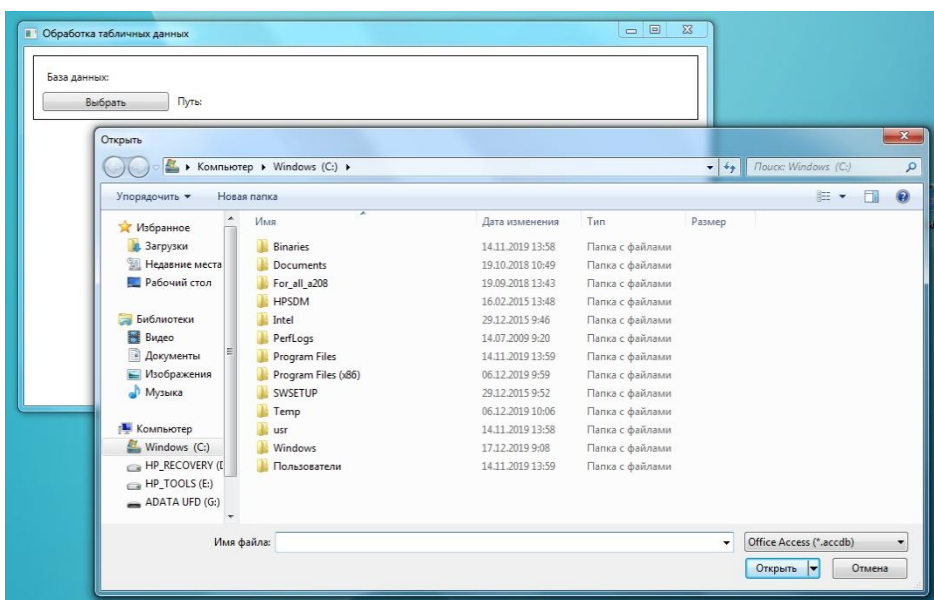


6. Для кнопки «Выбрать», следует прописать следующий код:

```
private void SelectDataBase(object sender, RoutedEventArgs e)
{
    OpenFileDialog fileDialog = new OpenFileDialog();
    fileDialog.InitialDirectory = "C:\\";
    fileDialog.Filter = "Office Access (*.accdb)|*.accdb";
    fileDialog.ShowDialog();

    if (fileDialog.FileName != "")
    {
        pathDataBase = fileDialog.FileName;

        pathDB.Content = "Путь: " + pathDataBase;
    }
    else
    {
        pathDataBase = "";
        pathDB.Content = "Путь: ";
    }
}
```

Добавить переменную pathDataBase и подключить библиотеку using Microsoft.Win32; которая даст возможность работать с проводником системы.

Теперь при нажатии на кнопку, программа позволяет вам выбрать файл базы данных, а при выборе отображает его местонахождение на диске:

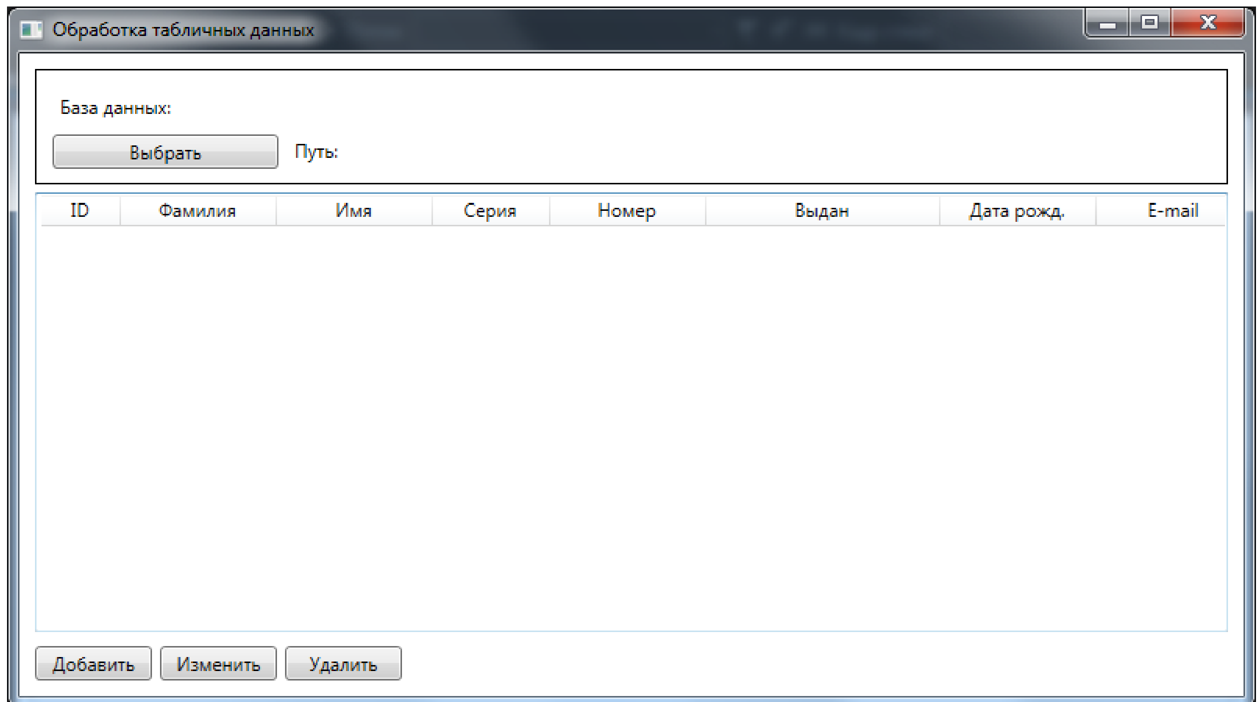


7. Прописываем функцию, которая позволит нам работать с базой данных:

```
public OleDbDataReader Query(string query)
{
    if (pathDataBase != "")
    {
        OleDbConnection connect = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source=" + pathDataBase);
        connect.Open();

        OleDbCommand cmd = new OleDbCommand(query, connect);
        OleDbDataReader reader = cmd.ExecuteReader();

        return reader;
    }
    else return null;
}
```

Данный код, обращается к базе данных и выполняет запрос query. Для работы функции необходимо добавить библиотеку using System.Data.OleDb;

8. Создать следующий интерфейс:



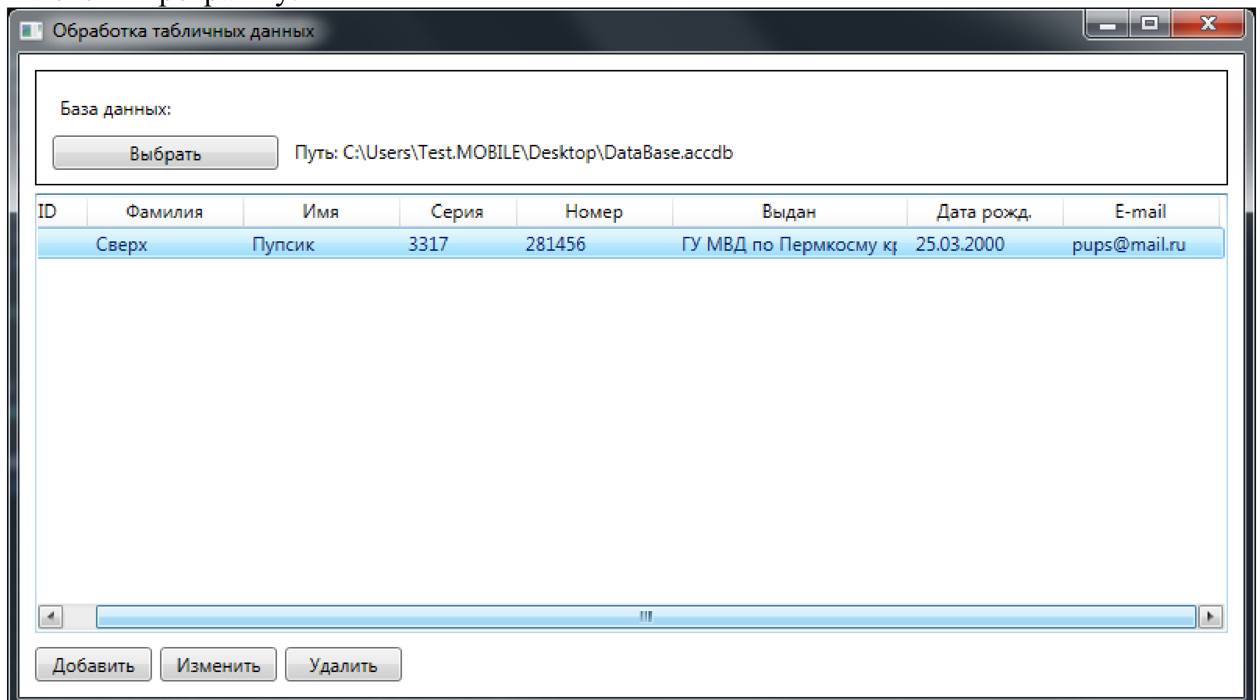
Код интерфейса:

```
            </Grid>
        </Border>
        <ListView x:Name="listView" Margin="10,89,10,41">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="ID" Width="50" DisplayMemberBinding="{Binding id}"/>
                    <GridViewColumn Header="Фамилия" Width="100" DisplayMemberBinding="{Binding firsName}"/>
                    <GridViewColumn Header="Имя" Width="100" DisplayMemberBinding="{Binding lastName}"/>
                    <GridViewColumn Header="Серия" Width="75" DisplayMemberBinding="{Binding series}"/>
                    <GridViewColumn Header="Номер" Width="100" DisplayMemberBinding="{Binding number}"/>
                    <GridViewColumn Header="Выдан" Width="150" DisplayMemberBinding="{Binding issuedBy}"/>
                    <GridViewColumn Header="Дата рожд." Width="100" DisplayMemberBinding="{Binding dateOfBirth}"/>
                    <GridViewColumn Header="E-mail" Width="100" DisplayMemberBinding="{Binding mail}"/>
                </GridView>
            </ListView.View>
        </ListView>
        <Button Content="Добавить" HorizontalAlignment="Left" Margin="10,0,0,10" VerticalAlignment="Bottom" Width="75"/>
        <Button Content="Изменить" HorizontalAlignment="Left" Margin="90,0,0,10" VerticalAlignment="Bottom" Width="75"/>
        <Button Content="Удалить" HorizontalAlignment="Left" Margin="170,0,0,10" VerticalAlignment="Bottom" Width="75"/>
    </Grid>          ContentControl.Content
</Window>
```

9. Написать функцию, которая будет получать данные из Базы данных, и выводить их в элемент ListView. Вызвать её при выборе файла базы данных.

```
public class User {
    public string ID { get; set; }
    public string firsName { get; set; }
    public string lastName { get; set; }
    public string series { get; set; }
    public string number { get; set; }
    public string issuedBy { get; set; }
    public string dateOfBirth { get; set; }
    public string mail { get; set; }
}
public List<User> user = new List<User>();


public void LoadDataBase() {
    OleDbDataReader read = Query("SELECT * FROM users");

    if (read != null) {
        while (read.Read())
        {
            User newUser = new User();
            newUser.ID = read.GetValue(0).ToString();
            newUser.firsName = read.GetValue(1).ToString();
            newUser.lastName = read.GetValue(2).ToString();
            newUser.series = read.GetValue(3).ToString();
            newUser.number = read.GetValue(4).ToString();
            newUser.issuedBy = read.GetValue(5).ToString();
            newUser.dateOfBirth = read.GetValue(6).ToString();
            newUser.mail = read.GetValue(7).ToString();

            listView.Items.Add(newUser);
            user.Add(newUser);
        }
    }
}
```

При запуске программы и выборе базы данных, можно увидеть что данные из БД подгружаются в программу:



10. Далее необходимо добавить новое окно и создать следующий интерфейс:

Фамилия:

Имя:

Серия:

Номер:

Выдан:

Дата рождения:

E-mail:

Добавить

Код страницы:

```
<Window x:Class="TableProcessing.AddUser"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Добавление пользователя" Height="480" Width="300">
    <Grid>
        <Label Content="Фамилия:" HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="firstName" Height="23" Margin="10,36,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="Имя:" HorizontalAlignment="Left" Margin="10,64,0,0" VerticalAlignment="Top"/>
        <TextBox Height="23" Margin="10,95,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="Серия:" HorizontalAlignment="Left" Margin="10,123,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="series" Height="23" Margin="10,154,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="Номер:" HorizontalAlignment="Left" Margin="10,182,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="nomber" Height="23" Margin="10,213,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="Выдан:" HorizontalAlignment="Left" Margin="10,241,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="issuedBy" Height="23" Margin="10,272,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="Дата рождения:" HorizontalAlignment="Left" Margin="10,300,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="dateOfBirth" Height="23" Margin="10,331,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Label Content="E-mail:" HorizontalAlignment="Left" Margin="10,359,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="mail" Height="23" Margin="10,390,10,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"/>
        <Button Content="Добавить" HorizontalAlignment="Left" Margin="10,418,0,0" VerticalAlignment="Top" Width="75"/>

    </Grid>
```

11. На главной форме добавить действие кнопке «Добавить»:

```
private void AddUser(object sender, RoutedEventArgs e)
{
    new AddUser(this).ShowDialog();
    listView.Items.Clear();
    user.Clear();
    LoadDataBase();
}
```

12. В форме добавления записи прописать следующий код:

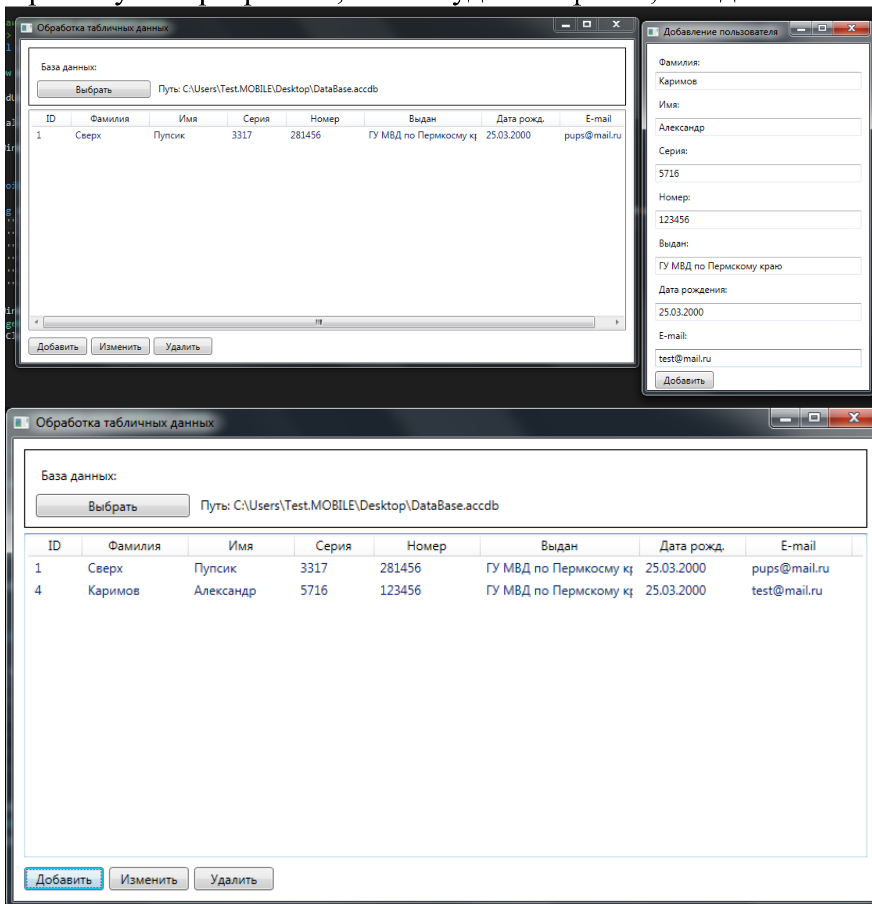
```
public partial class AddUser : Window
{
    MainWindow mainWindows;

    public AddUser(MainWindow _mainWindows)
    {
        InitializeComponent();

        mainWindows = _mainWindows;
    }


}
```

Теперь при нажатии кнопки «Добавить» будет происходить открытие дополнительного окна.

13. Дополнительной форме на кнопку «Добавить» присваиваем следующий код, который вносит данные в базу данных:

```
/// <summary>
/// Логика взаимодействия для AddUser.xaml
/// </summary>
public partial class AddUser : Window
{
    MainWindow mainWindows;

    public AddUser(MainWindow _mainWindows)
    {
        InitializeComponent();

        mainWindows = _mainWindows;
    }

    private void OnAddUser(object sender, RoutedEventArgs e)
    {
        string sql = "INSERT INTO users (firstName, lastName, series, [number], issuedBy, dateOfBirth, mail ) VALUES ('" + firstName.Text + "', " +
            "'" + lastName.Text + "'," +
            "'" + series.Text + "'," +
            "'" + number.Text + "'," +
            "'" + issuedBy.Text + "'," +
            "'" + dateOfBirth.Text + "'," +
            "'" + mail.Text + "'" + ");";

        mainWindows.Query(sql);
        MessageBox.Show("Запись добавленна.");
        this.Close();

    }

}
```

При запуске программы, можно удостоверится, что добавление работает:



14. Прокомментируйте код.
15. Сделайте выводы, оформите отчёт.

Дополнительные задания:

На оценку «хорошо» - реализуйте удаление из базы данных и программы.

На оценку «отлично» - реализуйте удаление и изменение в базе данных и программе.

Контрольные вопросы:

1. Какая библиотека позволяет работать с базой данных Microsoft Access?
2. Что позволяет выполнять язык SQL?
3. Опишите алгоритм работы с данными таблиц.