

Лекция 4

WPF. Начало

Разработка программных модулей

Тимашева Эльза Ринадовна

1. Введение в WPF
2. XAML
3. компоновка
 - 3.1. Grid
 - 3.2. GridSplitter
 - 3.3. StackPanel
 - 3.4. DockPanel
 - 3.5. WrapPanel
 - 3.6. Canvas
4. Свойства компоновки элементов

WPF

- Технология WPF (**Windows Presentation Foundation**) является частью экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.
- Если при создании традиционных приложений на основе WinForms за отрисовку элементов управления и графики отвечали такие части ОС Windows, как User32 и GDI+, то приложения WPF основаны на **DirectX**. В этом состоит ключевая особенность рендеринга графики в WPF: используя WPF, значительная часть работы по отрисовке графики, как простейших кнопочек, так и сложных 3D-моделей, ложится на графический процессор на видеокарте, что также позволяет воспользоваться аппаратным ускорением графики.
- Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: можно создавать насыщенный графический интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C# и VB.NET, либо совмещать и то, и другое.

Начальная страница - Microsoft Visual Studio

Файл Правка Вид Проект Отладка Команда Сервис Тест Анализ Окно Справка

Начальная страница

Начало работы

Новичок в Visual Studio? Ознакомьтесь с примерами проектов

Пройдите обучение новым платформам



Создайте закрытый репозиторий кода и проекта

Узнайте, как просто начать работать с облаком

Узнайте способы расширения и настройки

Последние

Вчера

-  **qwertyqwert.sln**
C:\Users\F\Downloads\Ахмадеев\Ах...
-  **ExamBat.sln**
C:\Users\F\Downloads\Баталев\Бат...












Вывод

Показать выходные данные из:

Создание проекта

Последние файлы .NET Framework 4.5.2 Сортировка: По умолчанию

- Установленные
- Шаблоны
 - Visual C#
 - Универсальные приложения
 - Классический рабочий стол
 - Веб
 - .NET Core
 - .NET Standard
 - Android
 - Cloud
 - Cross-Platform
 - iOS
 - tvOS
 - WCF
 - Тест
 - Visual Basic
 - Visual C++
 - Visual F#
 - SQL Server
 - Azure Data Lake
 - В сети

Иконка	Название шаблона	Язык
	Приложение WPF (.NET Framework)	Visual C#
	Приложение Windows	C#
	Консольное приложение (.NET Framework)	Visual C#
	Библиотека классов (.NET Framework)	Visual C#
	Общий проект	Visual C#
	Служба Windows (.NET Framework)	Visual C#
	Пустой проект (.NET Framework)	Visual C#
	Приложение браузера WPF (.NET Framework)	Visual C#
	Библиотека настраиваемых элементов управления...	Visual C#
	Библиотека пользовательских элементов управления...	Visual C#
	Библиотека элементов управления Windows Forms (...)	Visual C#

Тип: Visual C#
Клиентское приложение Windows Presentation Foundation

Имя: WpfApp1

Расположение: c:\users\F\documents\visual studio 2017\Projects

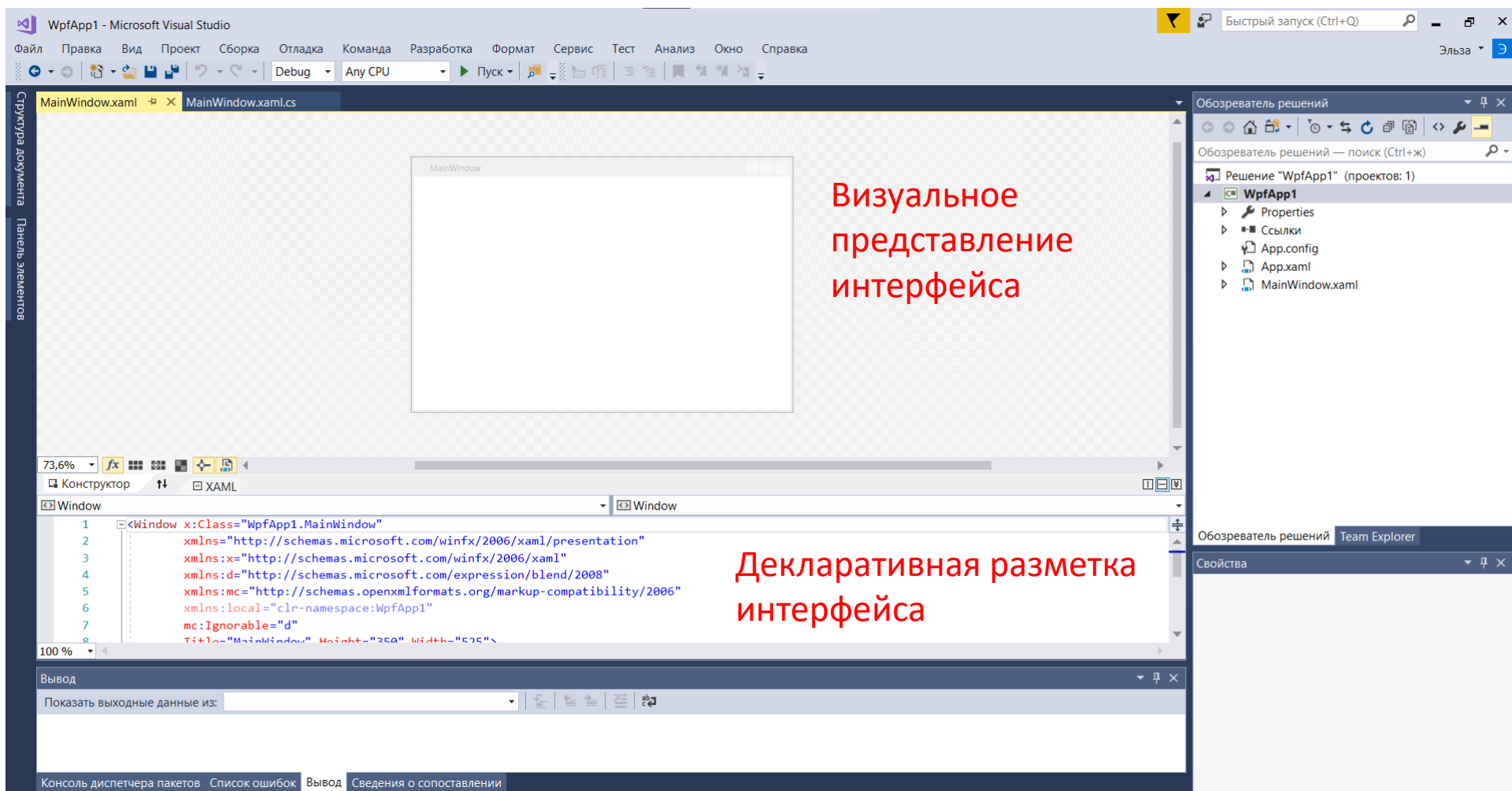
Имя решения: WpfApp1

Обзор...

☒ Создать каталог для решения

☐ Создать новый репозиторий Git

OK Отмена



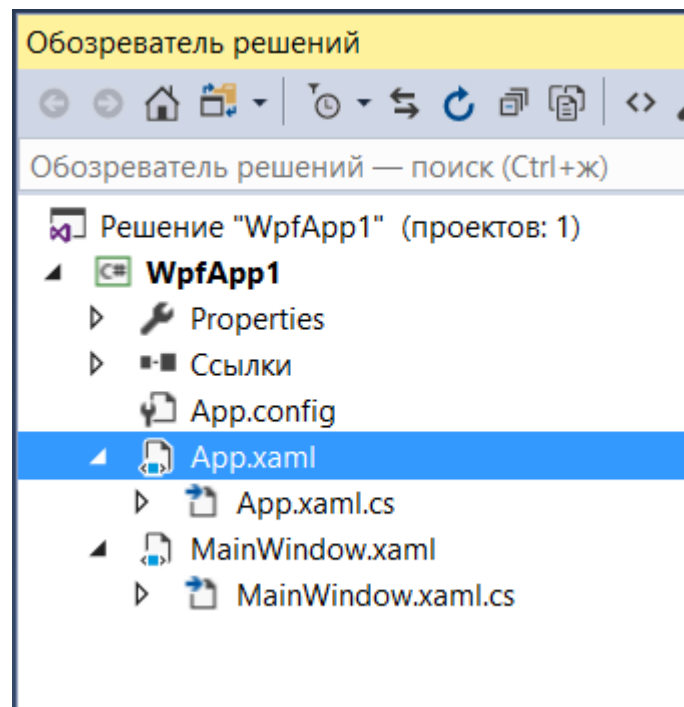
MainWindow.xaml MainWindow.xaml.cs X

WpfApp1 WpfApp1.MainWindow

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6 using System.Windows;
7 using System.Windows.Controls;
8 using System.Windows.Data;
9 using System.Windows.Documents;
10 using System.Windows.Input;
11 using System.Windows.Media;
12 using System.Windows.Media.Imaging;
13 using System.Windows.Navigation;
14 using System.Windows.Shapes;
15
16 namespace WpfApp1
17 {
18     /// <summary>
19     /// Логика взаимодействия для MainWindow.xaml
20     /// </summary>
21     public partial class MainWindow : Window
22     {
23         public MainWindow()
24         {
25             InitializeComponent();
26         }
27     }
28 }
29
```

100 %

Файл кода



Для большинства созданных элементов используется похожий набор свойств: ширина, высота, размер шрифта, отступы и др. Чтобы применять определенные наборы свойств для элементов, WPF предлагает использование глобальных стилей в проекте. Чтобы их создавать в проекте, есть файл *App.xaml*. Например, используется тег *Style* и свойство *TargetType*, чтобы указать, для каких элементов предназначен данный стиль.

XAML

XAML в целом напоминает язык разметки HTML: здесь сначала определен элемент верхнего уровня Window - окно приложения, в нем определен элемент Grid - контейнер верхнего уровня, в который можно добавлять другие элементы. Каждый элемент может иметь определенные атрибуты.

```
<Window x:Class="WpfApp1.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:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```


XAML (eXtensible Application Markup Language) - язык разметки, используемый для инициализации объектов в технологиях на платформе .NET. Применительно к WPF данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем.

Применительно к WPF будем говорить о нем чаще всего именно как о языке разметки, который позволяет создавать декларативным путем интерфейс, наподобие HTML в веб-программировании. Однако, сводить XAML к одному интерфейсу было бы неправильно, далее на примерах увидим.

XAML - не является обязательной частью приложения, вообще можно обходиться без него, создавая все элементы в файле связанного с ним кода на языке C#. Однако использование XAML все-таки несет некоторые преимущества:

- Возможность отделить графический интерфейс от логики приложения, благодаря чему над разными частями приложения могут относительно автономно работать разные специалисты: над интерфейсом - дизайнеры, над кодом логики - программисты.
- Компактность, понятность, код на XAML относительно легко поддерживать.
- При компиляции приложения в Visual Studio код в xaml-файлах также компилируется в бинарное представление кода baml, которое называется BAML (Binary Application Markup Language). И затем код baml встраивается в финальную сборку приложения - exe или dll-файл.

Структура и пространства имен XAML

Подобно структуре веб-страницы на html, здесь есть некоторая иерархия элементов. Элементом верхнего уровня является Window, который представляет собой окно приложения. При создании других окон в приложении потребуется начинать объявление интерфейса с элемента Window, поскольку это элемент самого верхнего уровня.

Кроме Window существует еще два элемента верхнего уровня:

- Page
- Application

Элемент Window имеет вложенный пустой элемент Grid, а также подобно html-элементам ряд атрибутов (Title, Width, Height) - они задают заголовок, ширину и высоту окна соответственно.

При создании кода на языке C#, чтобы были доступны определенные классы, подключаем пространства имен с помощью директивы using, например, using System.Windows;

Чтобы задействовать элементы в XAML, также подключаем пространства имен. Вторая и третья строчки как раз и представляют собой пространства имен, подключаемые в проект по умолчанию. А атрибут **xmlns** представляет специальный атрибут для определения пространства имен в XML.

- Так, пространство имен **<http://schemas.microsoft.com/winfx/2006/xaml/presentation>** содержит описание и определение большинства элементов управления. Так как является пространством имен по умолчанию, то объявляется без всяких префиксов.
- **<http://schemas.microsoft.com/winfx/2006/xaml>** - это пространство имен, которое определяет некоторые свойства XAML, например свойство Name или Key. Используемый префикс x в определении xmlns:x означает, что те свойства элементов, которые заключены в этом пространстве имен, будут использоваться с префиксом x - x:Name или x:Key. Это же пространство имен используется уже в первой строчке **x:Class="XamlApp.MainWindow"** - здесь создается новый класс MainWindow и соответствующий ему файл кода, куда будет прописываться логика для данного окна приложения.

Это два основных пространства имен. Рассмотрим остальные:

- **xmlns:d="http://schemas.microsoft.com/expression/blend/2008"**: предоставляет поддержку атрибутов в режиме дизайнера. Это пространство имен преимущественно предназначено для другого инструмента по созданию дизайна на XAML - Microsoft Expression Blend
- **xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"**: обеспечивает режим совместимости разметки XAML. В определении объекта Window двумя строчками ниже можно найти его применение:
mc:Ignorable="d"

Это выражение позволяет игнорировать парсерам XAML во время выполнения приложения дизайнерские атрибуты из пространства имен с префиксом d, то есть из "http://schemas.microsoft.com/expression/blend/2008"

- **xmlns:local="clr-namespace:XamlApp"**: пространство имен текущего проекта. Так как в моем случае проект называется XamlApp, то пространство имен называется аналогично. И через префикс *local* можно получить в XAML различные объекты, которые определены в проекте.

Элементы и их атрибуты

XAML предлагает очень простую и ясную схему определения различных элементов и их свойств. Каждый элемент, как и любой элемент XML, должен иметь открытый и закрытый тег, как в случае с элементом Window:

```
<Window атрибуты> </Window>
```

Либо элемент может иметь сокращенную форму с закрывающим слешем в конце, наподобие:

```
<Window />
```

Но в отличие от элементов xml каждый элемент в XAML соответствует определенному классу C#. Например, элемент Button соответствует классу System.Windows.Controls.Button. А свойства этого класса соответствуют атрибутам элемента Button.

Например, добавим кнопку в создаваемую по умолчанию разметку окна:

```
<Button x:Name="button" Width="100" Height="30" Content="Кнопка">  
</Button>
```

```
<Window x:Class="WpfApp1.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:WpfApp2"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="button" Width="100" Height="30" Content="Кнопка"></Button>
    </Grid>
</Window>
```

Сначала идет элемент самого высшего уровня - Window, затем идет вложенный элемент Grid - контейнер для других элементов, и в нем уже определен элемент Button, представляющий кнопку.

Для кнопки можно определить свойства в виде атрибутов. Здесь определены атрибуты x:Name (имя кнопки), Width, Height и Content. Причем, атрибут x:Name берется в данном случае из пространства имен "http://schemas.microsoft.com/winfx/2006/xaml", которое сопоставляется с префиксом x. А остальные атрибуты не используют префиксы, поэтому берутся из основного пространства имен "http://schemas.microsoft.com/winfx/2006/xaml/presentation".

Подобным образом можно определить и другие атрибуты, которые нужны. Либо можно не определять атрибуты, и тогда они будут использовать значения по умолчанию.

При создании нового проекта WPF в дополнение к создаваемому файлу *MainWindow.xaml* создается также файл отделенного кода *MainWindow.xaml.cs*, где, как предполагается, должна находиться логика приложения связанная с разметкой из *MainWindow.xaml*. Файлы XAML позволяют определить интерфейс окна, но для создания логики приложения, например, для определения обработчиков событий элементов управления, придется воспользоваться кодом C#.

По умолчанию в разметке окна используется атрибут **x:Class**:

Атрибут **x:Class** указывает на класс, который будет представлять данное окно и в который будет компилироваться код в XAML при компиляции. То есть во время компиляции будет генерироваться класс **WpfApp1.MainWindow**, унаследованный от класса *System.Windows.Window*.

Кроме того в файле отделенного кода *MainWindow.xaml.cs*, который Visual Studio создает автоматически, также можно найти класс с тем же именем - в данном случае класс *WpfApp1.MainWindow*.

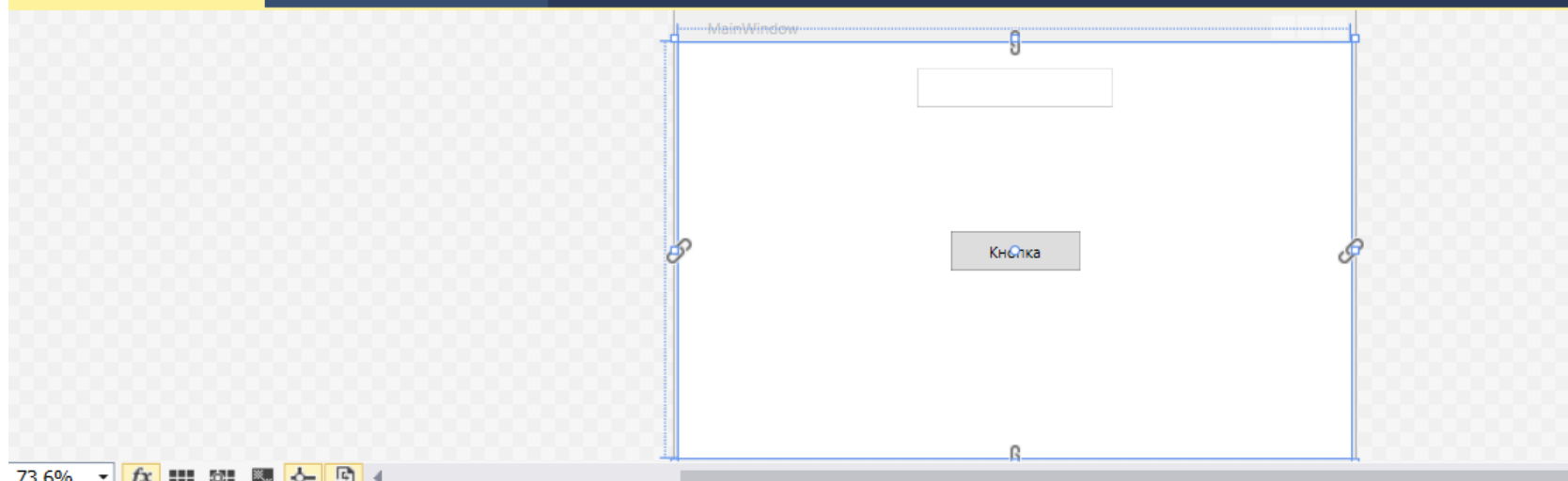
Во время компиляции этот класс объединяется с классом, сгенерированном из кода XAML. Чтобы такое слияние классов во время компиляции произошло, класс *WpfApp1.MainWindow* определяется как частичный с модификатором **partial**. А через метод *InitializeComponent()* класс *MainWindow* вызывает скомпилированный ранее код XAML, разбирает его и по нему строит графический интерфейс окна.

Взаимодействие кода C# и XAML

- В приложении часто требуется обратиться к какому-нибудь элементу управления. Для этого надо установить у элемента в XAML свойство Name.
- Еще одной точкой взаимодействия между xaml и C# являются события. С помощью атрибутов в XAML можно задать события, которые будут связаны с обработчиками в коде C#.

В разметке главного окна определим два элемента: кнопку и текстовое поле.

```
<Button x:Name="button" Width="100" Height="30"  
Content="Кнопка" Click="button_Click"></Button>  
<TextBox Name="textBox" Width="150" Height="30"  
VerticalAlignment="Top" Margin="20"></TextBox>
```



73,6% fx

Конструктор XAML

TextBox

TextBox

```

3      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6      xmlns:local="clr-namespace:WpfApp2"
7      mc:Ignorable="d"
8      Title="MainWindow" Height="350" Width="525">
9      <Grid>
10         <Button x:Name="button" Width="100" Height="30" Content="Кнопка" Click="button_Click"></Button>
11         <TextBox Name="textBox" Width="150" Height="30" VerticalAlignment="Top" Margin="20"></TextBox>
12     </Grid>
13 </Window>
14

```



```
- namespace WpfApp2
{
-     /// <summary>
-     /// Логика взаимодействия для MainWindow.xaml
-     /// </summary>
-     public partial class MainWindow : Window
-     {
-         public MainWindow()
-         {
-             InitializeComponent();
-         }
-
-         private void button_Click(object sender, RoutedEventArgs e)
-         {
-             string text = textBox.Text;
-             if (text != "")
-             {
-                 MessageBox.Show(text);
-             }
-         }
-     }
}
```

- Определив имена элементов в XAML, можно к ним обращаться в коде с#:
`string text = textBox.Text`
- При определении имен в XAML надо учитывать, что оба пространства имен `"http://schemas.microsoft.com/winfx/2006/xaml/presentation"` и `"http://schemas.microsoft.com/winfx/2006/xaml"` определяют атрибут **Name**, который устанавливает имя элемента. Во втором случае атрибут используется с префиксом **x**: `x>Name`. Какое именно пространство имен использовать в данном случае, не столь важно, а следующие определения имени `x>Name="button"` и `Name="button"` фактически будут равноценны.
- В обработчике нажатия кнопки просто выводится сообщение , введенное в текстовое поле. После определения обработчика его можно связать с событием нажатия кнопки в xaml через атрибут Click: `Click="Button_Click"`. В результате после нажатия на кнопку увидим в окне введенное в текстовое поле сообщение.

С помощью атрибутов можно задать различные свойства кнопки. Height и Width являются простыми свойствами. Они хранят числовое значение. А например, свойства HorizontalAlignment или Background являются более сложными по своей структуре.

```
<Button x:Name="button" Width="100" Height="30" Content="Кнопка" Click="button_Click">
    <Button.HorizontalAlignment>
        <HorizontalAlignment>Center</HorizontalAlignment>
    </Button.HorizontalAlignment>

    <Button.Background>
        <SolidColorBrush Opacity="0.5" Color="Red" />
    </Button.Background>
</Button>
```

Компоновка (layout)

Компоновка (layout) представляет собой процесс размещения элементов внутри контейнера.

Благодаря компоновке можно удобным образом настроить элементы интерфейса, позиционировать их определенным образом. Например, элементы компоновки в WPF позволяют при ресайзе - сжатии или растяжении масштабировать элементы, что очень удобно, а визуально не создает всяких шероховатостей типа незаполненных пустот на форме.

В WPF компоновка осуществляется при помощи специальных контейнеров. Фреймворк предоставляет нам следующие контейнеры:

Grid,
UniformGrid,
StackPanel,
WrapPanel,
DockPanel и
Canvas.

Различные контейнеры могут содержать внутри себя другие контейнеры.

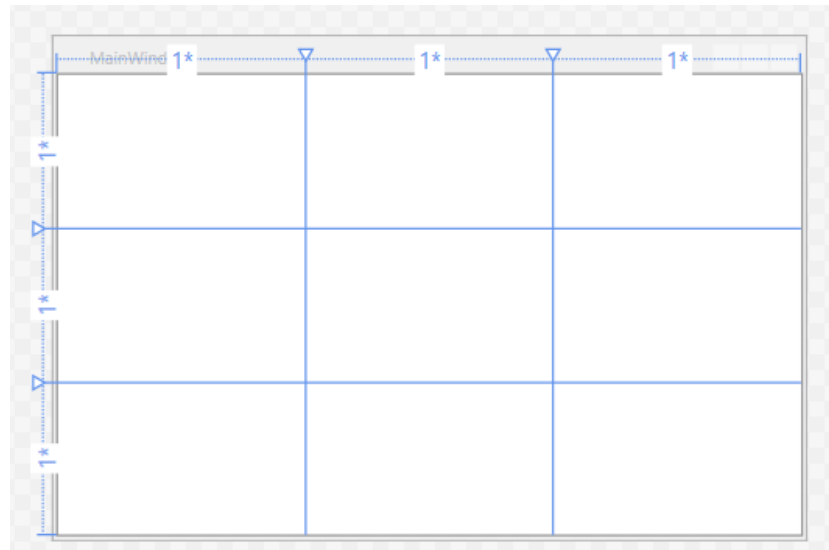
В WPF при компоновке и расположении элементов внутри окна надо придерживаться следующих принципов:

- Нежелательно указывать явные размеры элементов (за исключением минимальных и максимальных размеров). Размеры должны определяться контейнерами.
- Нежелательно указывать явные позицию и координаты элементов внутри окна. Позиционирование элементов всецело должно быть прерогативой контейнеров. И контейнер сам должен определять, как элемент будет располагаться. Если надо создать сложную систему компоновки, то можно вкладывать один контейнер в другой, чтобы добиться максимально удобного расположения элементов управления.

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

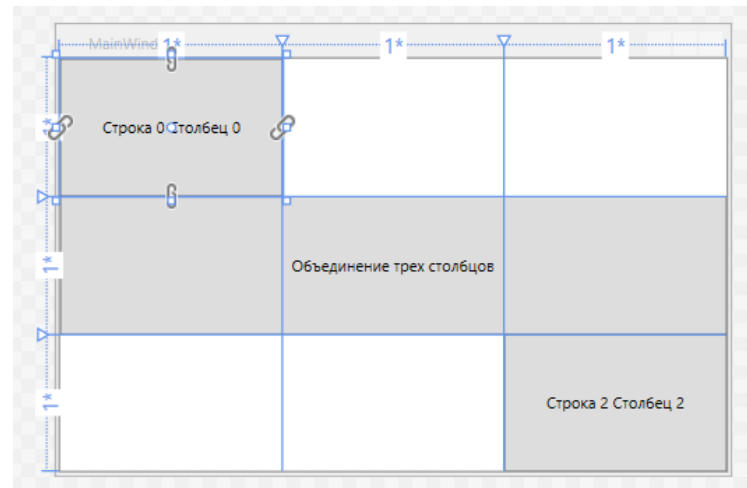


Чтобы задать позицию элемента управления с привязкой к определенной ячейке Grid, в разметке элемента нужно прописать значения свойств **Grid.Column** и **Grid.Row**, тем самым указывая, в каком столбце и строке будет находиться элемент. Кроме того, если нужно растянуть элемент управления на несколько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

```
<Button Grid.Column="0" Grid.Row="0"  
Content="Строка 0 Столбец 0"></Button>
```

```
<Button Grid.Column="0" Grid.Row="1"  
Content="Объединение трех столбцов"  
Grid.ColumnSpan="3"></Button>
```

```
<Button Grid.Column="2" Grid.Row="2"  
Content="Строка 2 Столбец 2"></Button>
```



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:

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

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

- **Абсолютные размеры**

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

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

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

Также абсолютные размеры можно задать в пикселях (px), дюймах (in), сантиметрах (cm) или точках (pt):

```
<RowDefinition Height="1 in"></RowDefinition>
```

```
<ColumnDefinition Width="100 px"></ColumnDefinition>
```


Установка размеров

- **Пропорциональные размеры.**

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

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

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

Если строка или столбец имеет высоту, равную *, то данная строка или столбец будет занимать все оставшееся место. Если есть несколько строк или столбцов, высота которых равна *, то все доступное место делится поровну между всеми такими строками 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>
```

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

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

В этом случае сумма коэффициентов равна $1* + 0.5* + 1.5* = 3*$. Если у нас грид имеет ширину 300 единиц, то для коэффициент $1*$ 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

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

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

Чтобы использовать элемент GridSplitter, надо поместить его в ячейку в Grid. По сути это обычный элемент, такой же, как кнопка. В примере три ячейки (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.

- В случае, если задается горизонтальный разделитель, то тогда соответственно надо использовать свойство `Grid.ColumnSpan`
- Затем надо настроить свойства. Во-первых, надо настроить ширину (`Width`) для вертикальных сплиттеров и высоту (`Height`) для горизонтальных. Если не задать соответствующее свойство, то сплиттер не увидим, так как он изначально очень мал.
- Затем надо задать выравнивание. Если хотим, чтобы сплиттер заполнял всю высоту доступной области (то есть если у нас вертикальный сплиттер), то надо установить для свойства **`VerticalAlignment`** значение `Stretch`.
- Если же у нас горизонтальный сплиттер, 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 сплиттера. Если же оно равно `True`, тогда изменение границ будет происходить только после того, как перемещение сплиттера завершится, и при перемещении сплиттера увидим его проекцию.
- В отличие от элемента `SplitContainer` в `WinForms`, в `WPF` можно установить различное количество динамически регулируемых частей окна.

<Grid>

<Grid.ColumnDefinitions>

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

<ColumnDefinition Width="Auto"></ColumnDefinition>

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

</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>

<GridSplitter Grid.Row="1" Grid.ColumnSpan="3" Height="3"

HorizontalAlignment="Stretch" VerticalAlignment="Center"></GridSplitter>

<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>

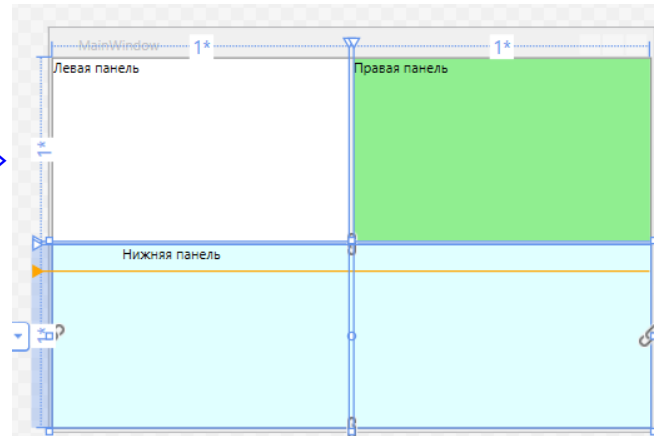
</Canvas>

<Canvas Grid.ColumnSpan="3" Grid.Row="2" Background="LightCyan">

<TextBlock Canvas.Left="60">Нижняя панель</TextBlock>

</Canvas>

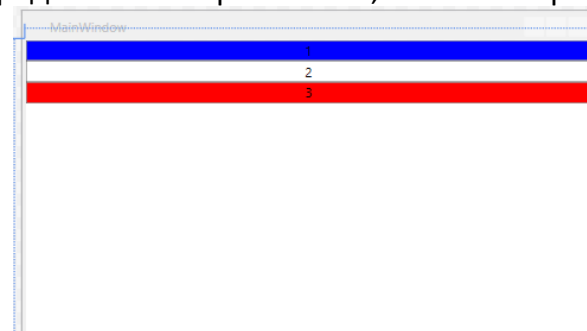
</Grid>



StackPanel

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

```
<StackPanel>
    <Button Background="Blue" Content="1"></Button>
    <Button Background="White" Content="2"></Button>
    <Button Background="Red" Content="3"></Button>
</StackPanel>
```

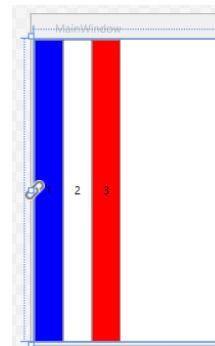


В данном случае для свойства Orientation по умолчанию используется значение Vertical, то есть StackPanel создает вертикальный ряд, в который помещает все вложенные элементы сверху вниз.

Также можно задать горизонтальный стек.

Для этого нам надо указать свойство Orientation="Horizontal":

```
<StackPanel Orientation="Horizontal">
    <Button Background="Blue" MinWidth="30" Content="1"></Button>
    <Button Background="White" MinWidth="30" Content="2"></Button>
    <Button Background="Red" MinWidth="30" Content="3"></Button>
</StackPanel>
```



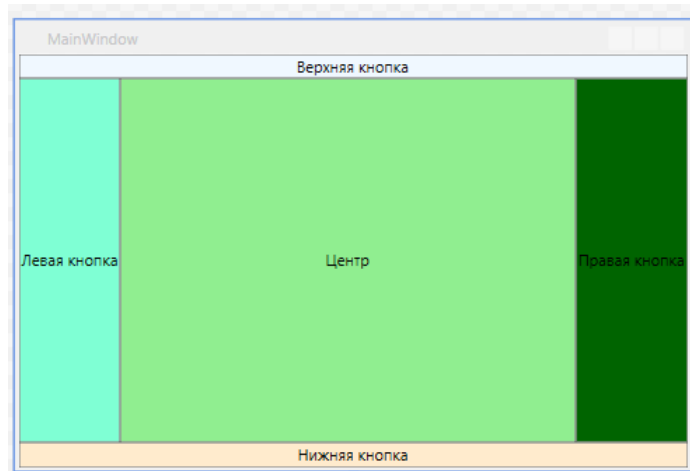
При горизонтальной ориентации все вложенные элементы располагаются слева направо. Если мы хотим, чтобы наполнение стека начиналось справа налево, то нам надо задать свойство FlowDirection: <StackPanel Orientation="Horizontal" FlowDirection="RightToLeft">. По умолчанию это свойство имеет значение LeftToRight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

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

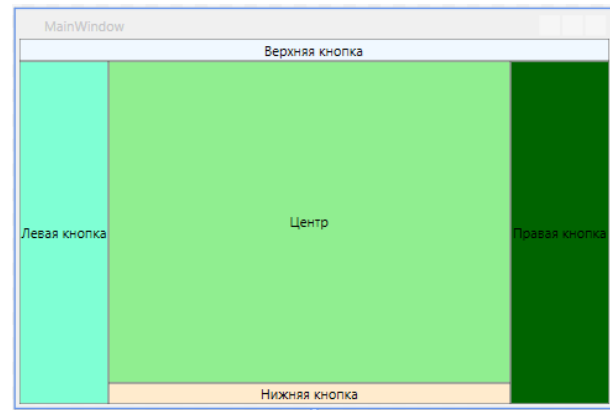
Причем у последней кнопки можно не устанавливать свойство DockPanel.Dock. Она уже заполняет все оставшееся пространство. Такой эффект получается благодаря установке у DockPanel свойства LastChildFill="True", которое означает, что последний элемент заполняет все оставшееся место. Если у этого свойства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



DockPanel

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

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



В этом случае нижняя кнопка уже будет заполнять меньшее место.

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

```
<DockPanel LastChildFill="True">  
  <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка 1"></Button>  
  <Button DockPanel.Dock="Top" Background="AliceBlue" Content="Верхняя кнопка 2"></Button>  
  <Button DockPanel.Dock="Bottom" Background="BlanchedAlmond" Content="Нижняя кнопка"></Button>  
  <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка1"></Button>  
  <Button DockPanel.Dock="Left" Background="Aquamarine" Content="Левая кнопка2"></Button>  
  <Button DockPanel.Dock="Right" Background="DarkGreen" Content="Правая кнопка"></Button>  
  <Button Background="LightGreen" Content="Центр"></Button>  
</DockPanel>
```

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.

`<WrapPanel>`

```
<Button Background="AliceBlue" Content="Кнопка 1"></Button>
<Button Background="Blue" Content="Кнопка 2"></Button>
<Button Background="Aquamarine" Content="Кнопка 3" Height="30"></Button>
<Button Background="DarkGreen" Content="Кнопка 4" Height="20"></Button>
<Button Background="LightGreen" Content="Кнопка 5"></Button>
<Button Background="RosyBrown" Content="Кнопка 6" Width="250" ></Button>
<Button Background="GhostWhite" Content="Кнопка 7" ></Button>
```

`</WrapPanel>`

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.

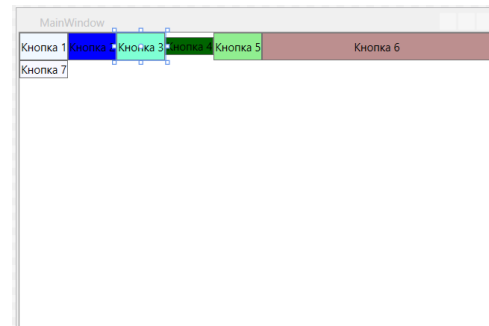
В вертикальном стеке элементы, у которых явным образом не указана ширина, автоматически принимают ширину самого широкого элемента.

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

```
<WrapPanel ItemHeight="30" ItemWidth="80" Orientation="Horizontal">
```

...

`</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 размерах, 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:

```
<Canvas Background="Lavender">
    <Button Background="AliceBlue" Content="Top 20 Left 40" Canvas.Top="20" Canvas.Left="40"></Button>
    <Button Background="LightSkyBlue" Content="Top 20 Right 20" Canvas.Top="20" Canvas.Right="20">
</Button>
    <Button Background="Aquamarine" Content="Bottom 30 Left 20" Canvas.Bottom="30" Canvas.Left="20">
</Button>
    <Button Background="LightCyan" Content="Bottom 20 Right 40" Canvas.Bottom="20" Canvas.Right="40">
</Button>
</Canvas>
```

Свойства компоновки элементов

Элементы WPF обладают набором свойств, которые помогают позиционировать данные элементы. Рассмотрим некоторые из этих свойств.

- **Ширина и высота**

У элемента можно установить ширину с помощью свойства `Width` и высоту с помощью свойства `Height`. Эти свойства принимают значение типа `double`. Хотя общая рекомендация состоит в том, что желательно избегать жестко закодированных в коде ширины и высоты.

Также можно задать возможный диапазон ширины и высоты с помощью свойств `MinWidth/MaxWidth` и `MinHeight/MaxHeight`. И при растяжении или сжатии контейнеров элементы с данными заданными свойствами не будут выходить за пределы установленных значений.

Возможно, возникает вопрос, а в каких единицах измерения устанавливаются ширина и высота? Да и в общем какие единицы измерения используются? В WPF можно использовать несколько единиц измерения: сантиметры (cm), точки (pt), дюймы (in) и пиксели (px). Например, зададим размеры в других единицах: `<Button Content="Кнопка" Width="5cm" Height="0.4in" />`

Если единица измерения не задана явно, а просто стоит число, то используются по умолчанию пиксели. Но эти пиксели не равны обычным пикселям, а являются своего рода "логическими пикселями", независимыми от конкретного устройства. Каждый такой пиксель представляет 1/96 дюйма вне зависимости от разрешения экрана.

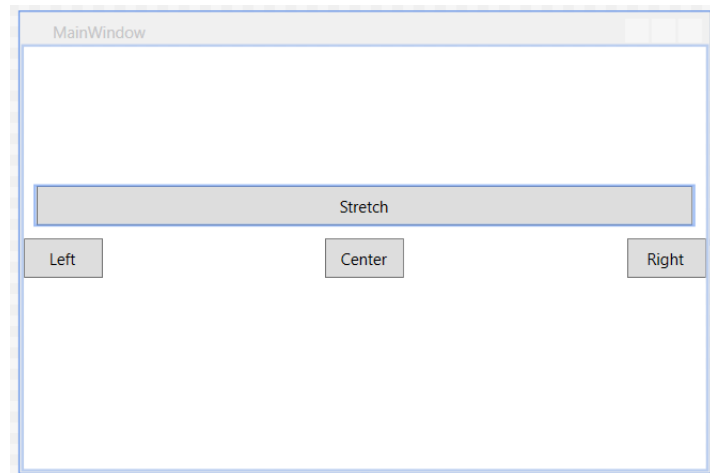
Свойства компоновки элементов

Выравнивание

HorizontalAlignment

С помощью специальных свойств можно выравнивать элемент относительно определенной стороны контейнера по горизонтали или вертикали.

Свойство **HorizontalAlignment** выравнивает элемент по горизонтали относительно правой или левой стороны контейнера и соответственно может принимать значения Left, Right, Center (положение по центру), Stretch (растяжение по всей ширине).



<Grid>

```
<Button Content="Left" Width="60" Height="30" HorizontalAlignment="Left"></Button>
<Button Content="Center" Width="60" Height="30" HorizontalAlignment="Center"></Button>
<Button Content="Right" Width="60" Height="30" HorizontalAlignment="Right"></Button>
<Button Content="Stretch" Height="30" HorizontalAlignment="Stretch" Margin="10 -80 10
0"></Button>
</Grid>
```

Свойства компоновки элементов

VerticalAlignment

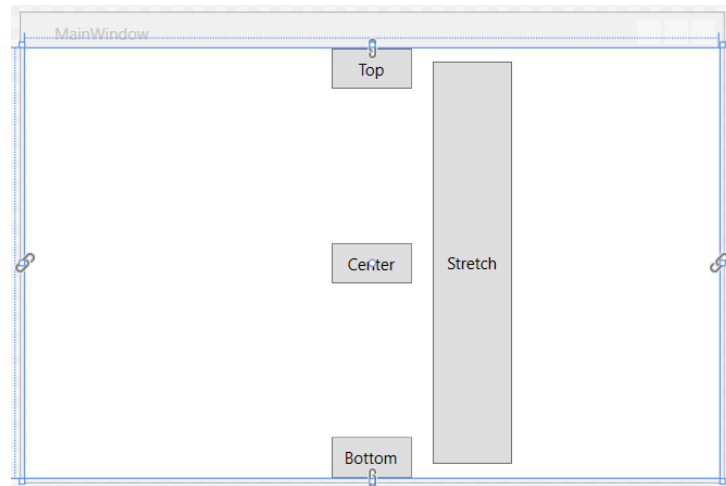
Также можно задать для элемента выравнивание по вертикали с помощью свойства **VerticalAlignment**, которое принимает следующие значения:

Top (положение в верху контейнера),

Bottom (положение внизу),

Center (положение по центру),

Stretch (растяжение по всей высоте).



`<Grid>`

```
<Button Content="Bottom" Width="60" Height="30" VerticalAlignment="Bottom"></Button>
```

```
<Button Content="Center" Width="60" Height="30" VerticalAlignment="Center" ></Button>
```

```
<Button Content="Top" Width="60" Height="30" VerticalAlignment="Top" ></Button>
```

```
<Button Content="Stretch" Width="60" VerticalAlignment="Stretch" Margin="150 10 0 10"
```

```
></Button>
```

```
</Grid>
```

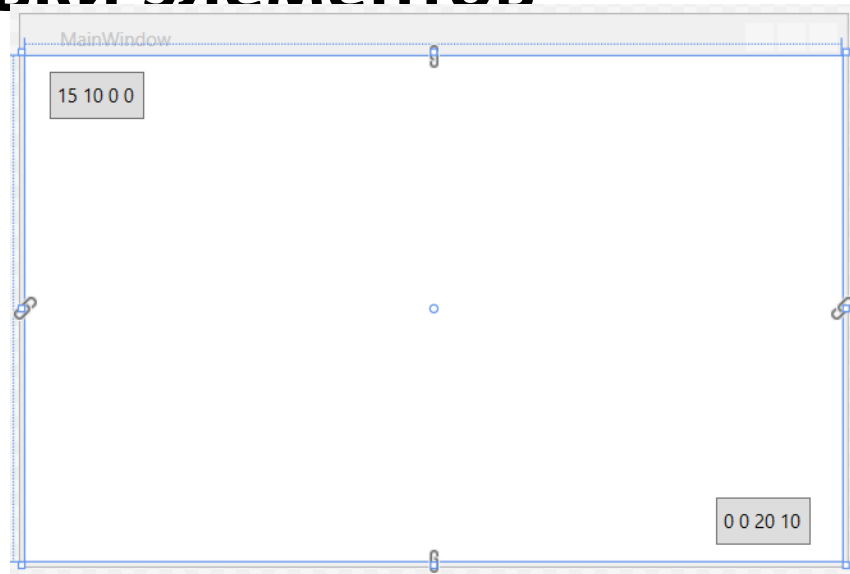
Свойства компоновки элементов

Отступы margin

Свойство **Margin** устанавливает отступы вокруг элемента. Синтаксис:

Margin="левый_отступ верхний_отступ правый_отступ нижний_отступ".

Например, установим отступы у одной кнопки слева и сверху, а у другой кнопки справа и снизу:



```
<Grid>
```

```
<Button Content="15 10 0 0" Width="60" Height="30" Margin ="15 10 0 0"  
HorizontalAlignment="Left" VerticalAlignment="Top"></Button>
```

```
<Button Content="0 0 20 10" Width="60" Height="30" Margin ="0 0 20 10"  
HorizontalAlignment="Right" VerticalAlignment="Bottom"></Button>
```

```
</Grid>
```

Если задать свойство таким образом: Margin="20", то сразу установится отступ для всех четырех сторон.