

Лабораторная работа №7. Создание приложений в WPF

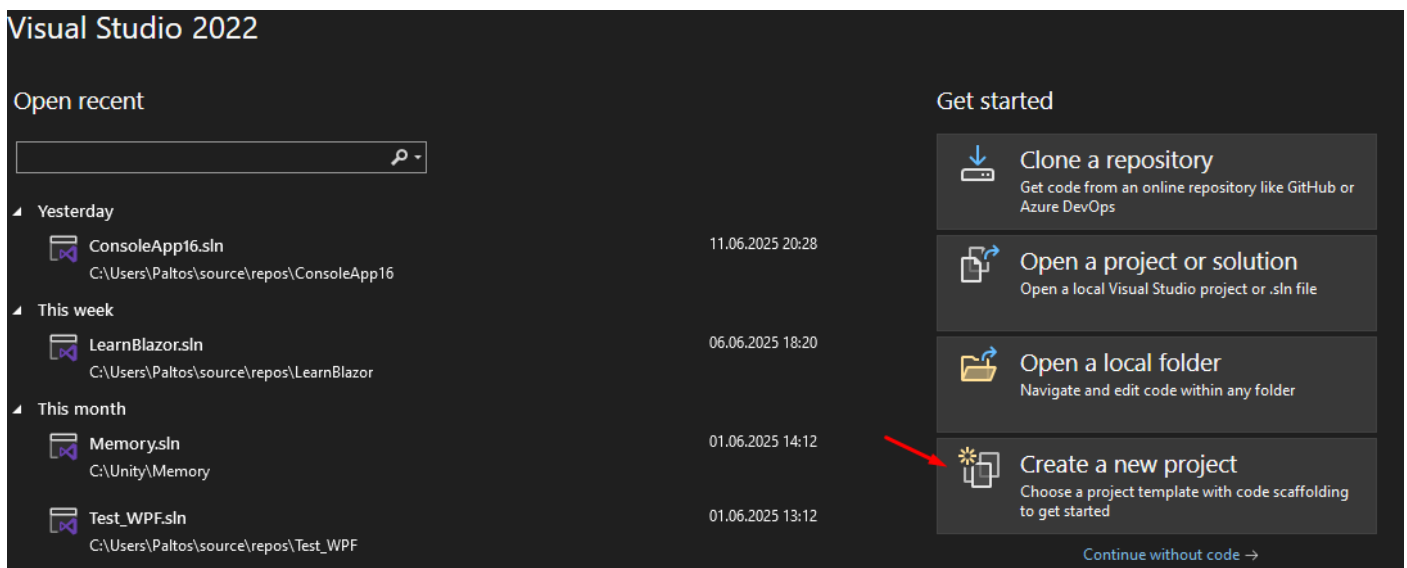
Технология **WPF (Windows Presentation Foundation)** является частью экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

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.

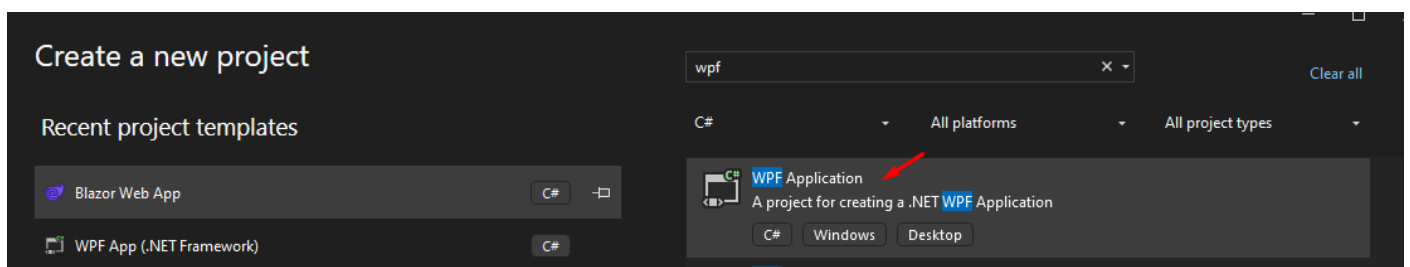
Одной из важных особенностей является использование языка декларативной разметки интерфейса XAML, основанного на XML: вы можете создавать насыщенный графический интерфейс, используя или декларативное объявление интерфейса, или код на управляемых языках C#, VB.NET и F#, либо совмещать и то, и другое.

Шаг 1. Создание проекта.

1. Откройте Visual Studio 2022. Выберите **Create a new project**:



2. Для быстрого поиска шаблона можете ввести в поиске **wpf** и выбрать язык C#. Нас интересует **WPF Application**, выбрав нажимаете **Next**:



3. Далее вам нужно ввести название проекта – **WPF_Demo**, остальные параметры можно оставить по умолчанию и нажмите **Next**:

Configure your new project

WPF Application C# Windows Desktop

Project name
WPF_Demo

Location
C:\Users\Paltos\VisualStudioProject

Solution name
WPF_Demo

☒ Place solution and project in the same directory

Project will be created in "C:\Users\Paltos\VisualStudioProject\WPF_Demo\"

Back Next

4. Можете выбрать **.NET 8** или **9** и нажать **Create**:

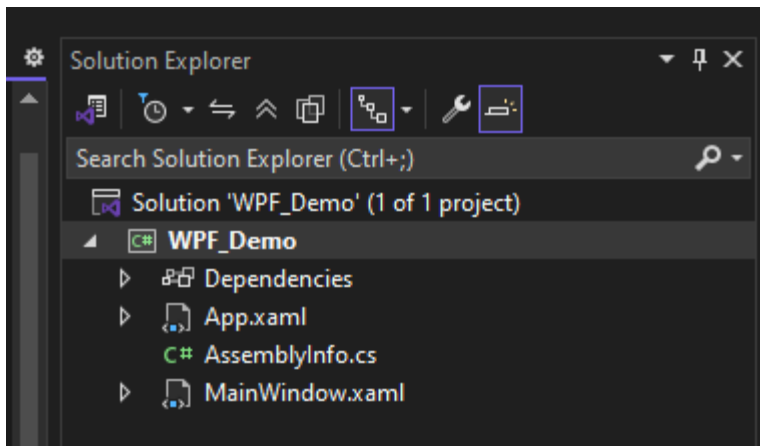
Additional information

WPF Application C# Windows Desktop

Framework
.NET 8.0 (Long Term Support)

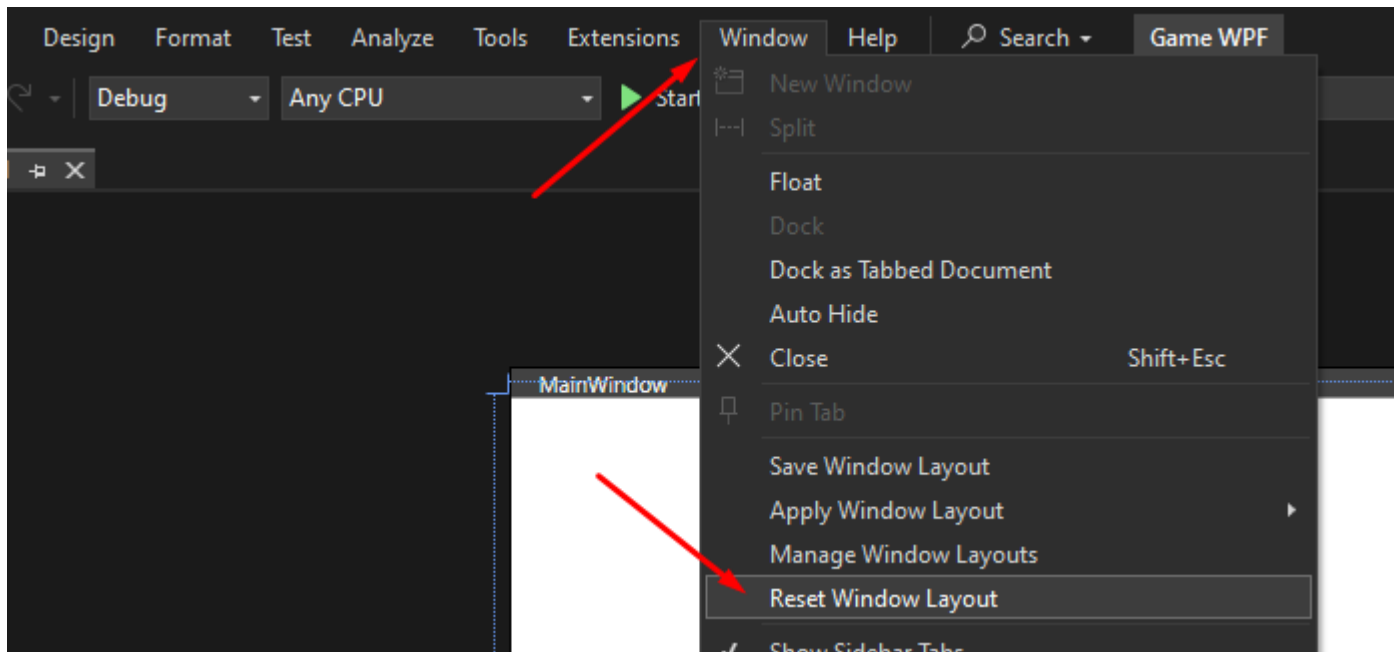
Back Create

5. Нажмите на **Solution Explorer**. Вы увидите ваш проект с вложенными файлами:

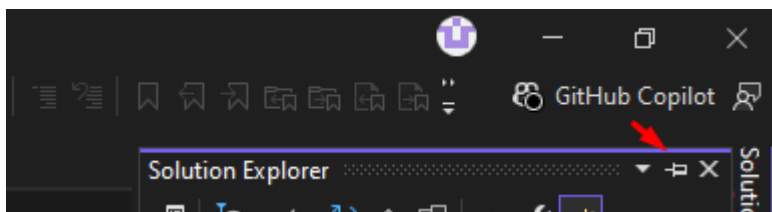


Нас интересует два файла - **MainWindow.xaml** (где мы видим пользовательский интерфейс главного окна) и **MainWindow.xaml.cs** (где размещается код C#, обеспечивающий работоспособность вашей игры).

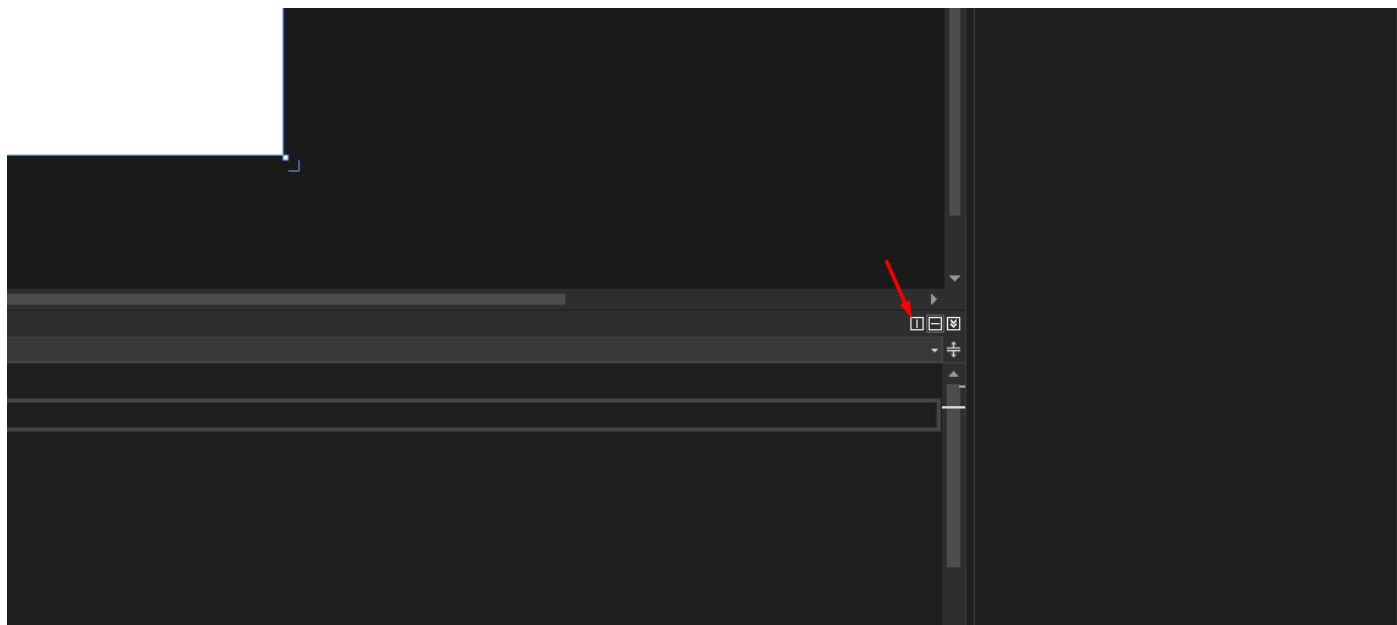
6. Если у вас вдруг сбилось расположение окон, нажмите **Windows – Reset Windows Layout**:



Для удобства работы можно нажать на скрепку и скрыть ненужные панели:



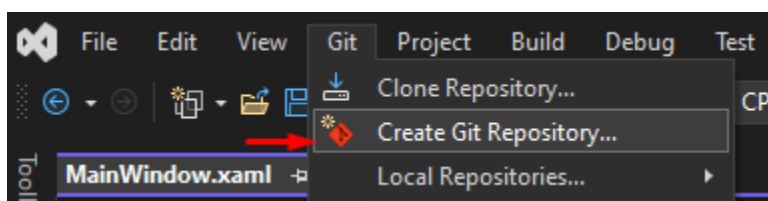
Также для удобства можно нажать на Vertical Split, чтобы слева был виден предварительный просмотр и справа **xml**:



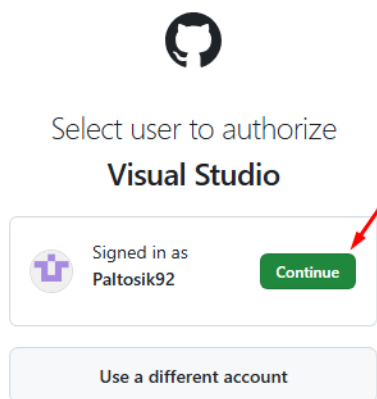
Шаг 2. Подключение к GitHub и первый коммит

В начале в **браузере** по **умолчанию** войдите в свой аккаунт на **GitHub**.

1. В **Visual Studio 2022** откройте вкладку **Git** → **Create Git Repository**:



Вас перебросит на сайт где нужно будет выбрать **Continue** для своей учётной записи:



Возвращаемся в Visual Studio 2022.

2. Можете поменять имя проекта, сделать его видимым для всех или оставить по умолчанию приватным и нажимаете **Create and Push**:

Create a Git repository

Push to a new remote

- GitHub
- Azure DevOps

Other

- Existing remote
- Local only

Initialize a local Git repository

Local path

.gitignore template

License template

☐ Add a README.md

Create a new GitHub repository

Account

Owner

Repository name

Description

Visibility

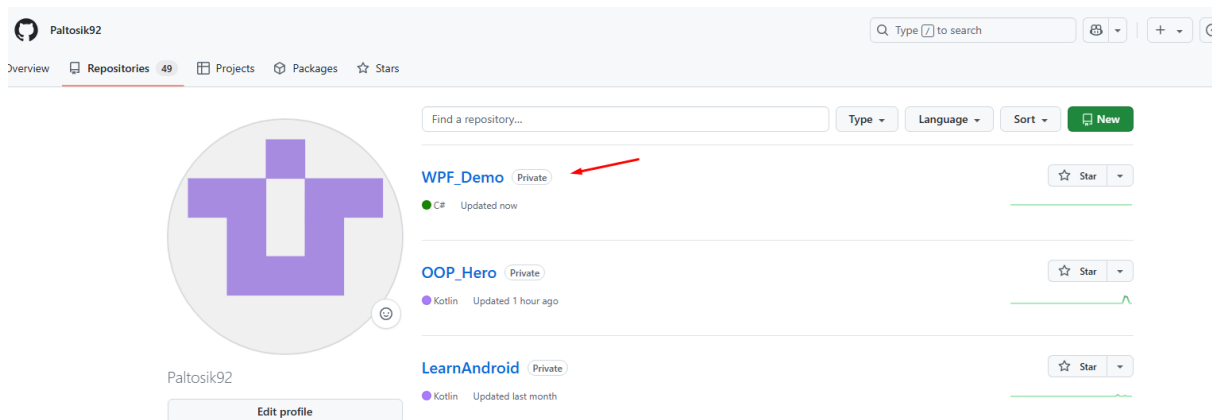
You choose who can see and commit to this repository.

Push your code to GitHub

<https://github.com/Paltosik92/WPFDemo>

Create and Push Cancel

Перейдите на свой аккаунт в **GitHub** в меню **Repositories** и убедитесь в наличие проекта:

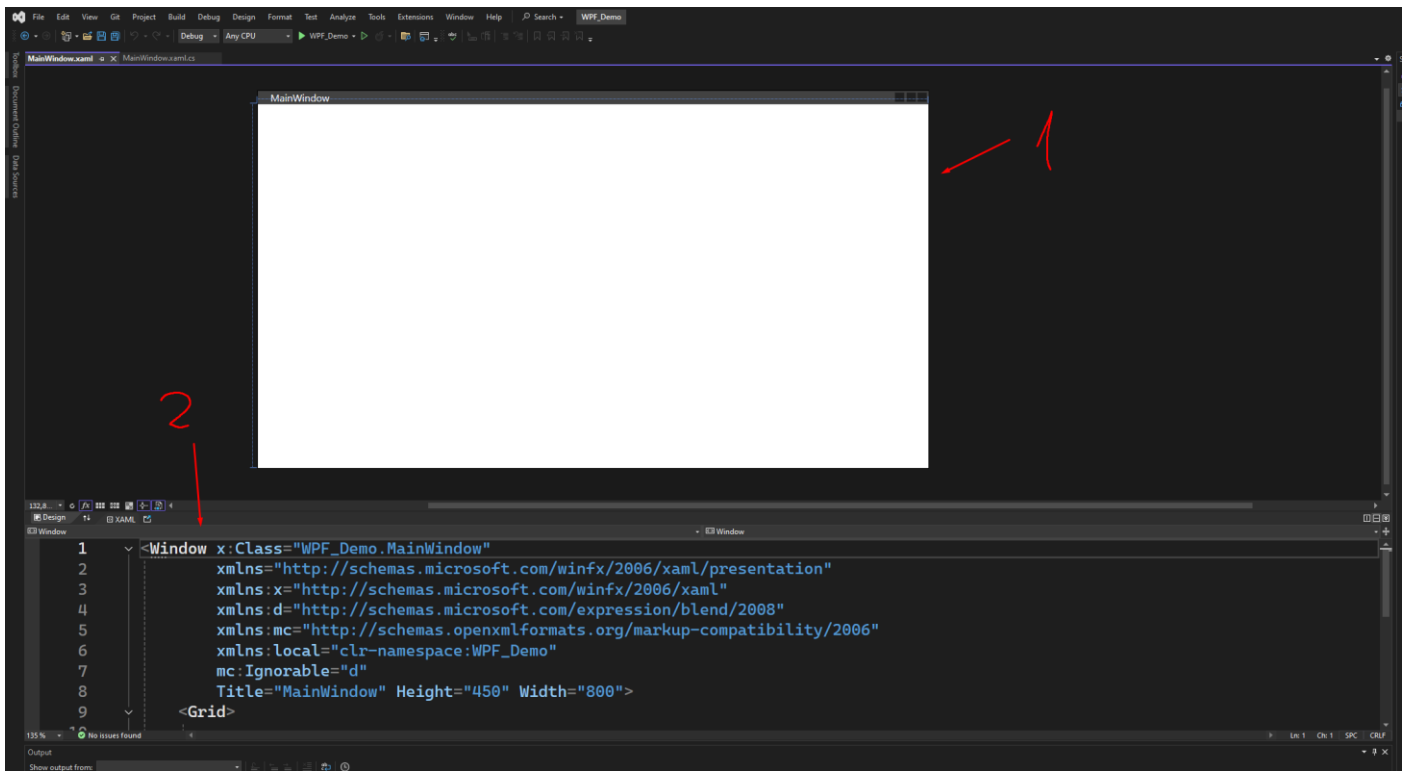


Шаг 3. Знакомство с интерфейсом и структурой

После создания проекта в WPF вы увидите два основных окна:

1. **Design (Дизайнер)** — графическое представление пользовательского интерфейса. Здесь вы можете визуальнo размещать элементы управления. Это своеобразный визуальный конструктор формы, но в WPF элементы не размещаются "на форме", как в Windows Forms, а внутри разметки XAML.

2. **XAML** — это язык разметки, с помощью которого описывается внешний вид и структура интерфейса. Также здесь можно указывать некоторые аспекты поведения элементов управления.



Структура файла MainWindow.xaml

При создании проекта в файле MainWindow.xaml автоматически формируется базовый шаблон окна. Рассмотрим его основные части:

- **x:Class** — связывает XAML-разметку с кодом C# (например, классом MainWindow).
- **xmlns** — пространства имён, аналогичные using в C#:
 - `xmlns="..."` — подключает стандартные элементы WPF;
 - `xmlns:x` — подключает XAML-специфичные директивы (например, `x>Name`);
 - `xmlns:d` и `xmlns:mc` — используются дизайнером (например, Blend) и для обеспечения совместимости;
 - `xmlns:local` — предоставляет доступ к локальным классам проекта.
- **mc:Ignorable="d"** — указывает, что пространство d: предназначено только для дизайна и может быть проигнорировано во время выполнения.
- **Title, Height, Width** — задают заголовок окна, а также его размеры.
- **<Grid>** — основной контейнер, внутри которого размещаются элементы управления.

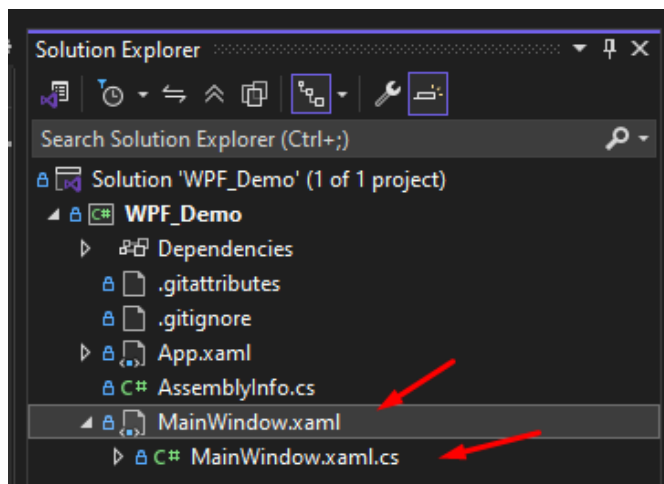
Контейнер **<Grid>** работает по принципу таблицы и позволяет удобно размещать элементы по строкам и столбцам. Именно в нём обычно строится структура интерфейса.

Если сейчас нажать "Пуск" (F5), откроется пустое окно, что означает — приложение успешно скомпилировалось и работает.

Структура проекта в Solution Explorer

В проводнике решений (Solution Explorer) обратите внимание на следующие файлы:

- **MainWindow.xaml** — отвечает за визуальную часть и XAML-разметку окна;
- **MainWindow.xaml.cs** — содержит программную логику, связанную с этим окном.



Откройте файл **MainWindow.xaml.cs**.

Здесь мы видим, что определён класс **MainWindow**, который наследуется от класса **Window**:

```
public partial class MainWindow : Window
```

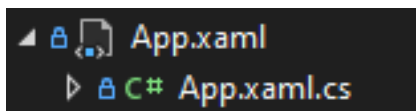
Также реализован конструктор:

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

Метод **InitializeComponent()** отвечает за инициализацию всех компонентов пользовательского интерфейса, определённых в XAML-разметке. Без его вызова элементы из XAML не будут отображаться и не будут доступны в коде.

App.xaml и точка входа приложения

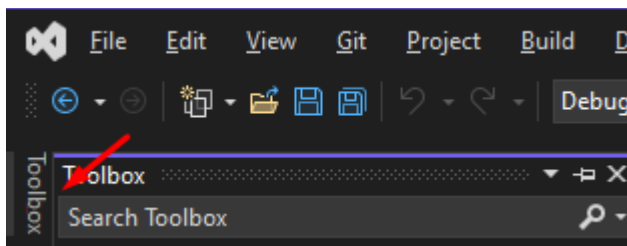
В проводнике решений также присутствуют файлы **App.xaml** и **App.xaml.cs**. Они играют роль **точки входа** в WPF-приложение, аналогичной **static void Main()** в консольных приложениях.



Перейдите в файл App.xaml.cs. Если вы нажмёте F12 по классу Application, откроется его реализация в .NET, содержащая логику начальной загрузки приложения. Обычно этот код не изменяется, но полезно знать, где он находится.

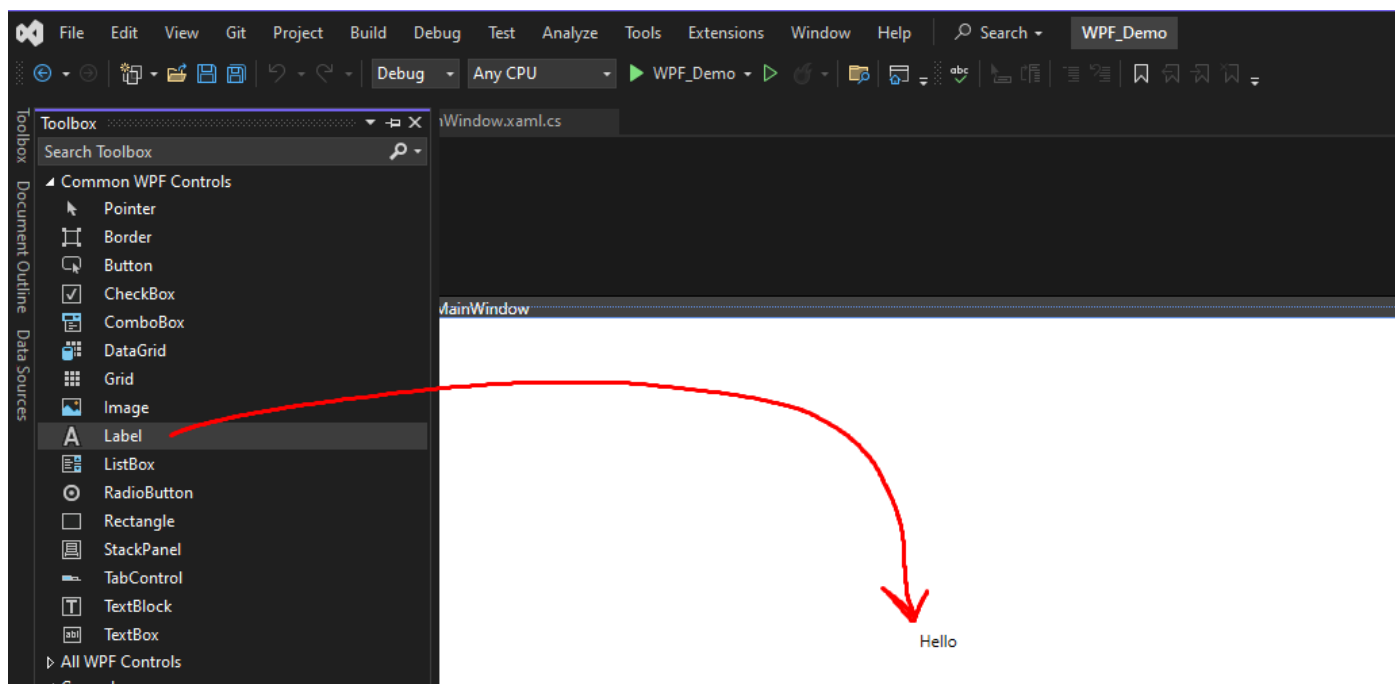
Добавление элемента управления

Слева находится панель инструментов **Toolbox**.



Если вы её не видите, откройте меню **View** → **Toolbox** или нажмите **Ctrl+Alt+X**.

Теперь давайте добавим элемент управления — например, **Label**. Перетащите его из панели **Toolbox** в окно конструктора.



В результате в разметке XAML появится следующий код:

```
<Grid>
    <Label Content="Hello" HorizontalAlignment="Left"
        Margin="369,225,0,0" VerticalAlignment="Top"/>
</Grid>
```

Мы видим, что внутри контейнера `<Grid>` появился элемент `<Label>`. У него есть следующие свойства:

- **Content** — отображаемый текст;
- **HorizontalAlignment** / **VerticalAlignment** — выравнивание по горизонтали и вертикали;
- **Margin** — внешние отступы.

Если вы хотите изменить отображаемый текст, просто замените значение свойства **Content**:

```
<Label Content="Hello World!"
```

При добавлении любого элемента управления он сразу же отображается в XAML-разметке. Каждый элемент имеет набор свойств, которые вы можете настраивать вручную через XAML или с помощью визуального редактора. Понимание структуры XAML и взаимодействия с кодом C# — основа эффективной работы с WPF.

Шаг 4. Использование сетки Grid с разметкой строк и столбцов

Контейнер **<Grid>** в WPF представляет собой мощный инструмент для размещения элементов управления по строкам и столбцам, аналогично таблице. Чтобы получить контроль над расположением элементов, можно явно задать количество и размеры строк и столбцов с помощью элементов **<Grid.ColumnDefinitions>** и **<Grid.RowDefinitions>**.

Добавим следующий XAML-код:

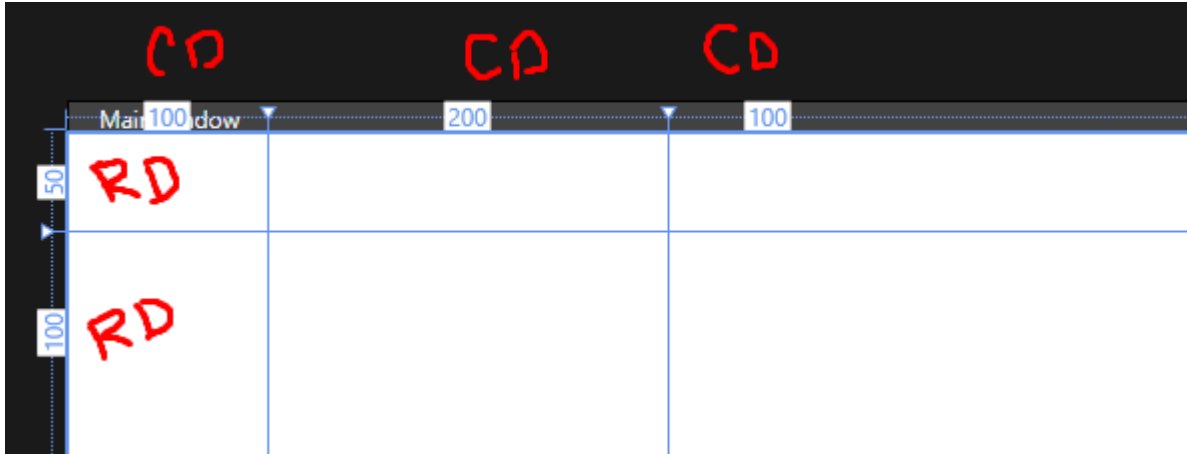
```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="200"/>
    <ColumnDefinition Width="100"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="50"/>
    <RowDefinition Height="100"/>
  </Grid.RowDefinitions>
  <Label Content="Hello World!" HorizontalAlignment="Left"
    Margin="369,225,0,0" VerticalAlignment="Top"/>
</Grid>
```

- Внутри контейнера **<Grid>** мы определяем:
 - **Три столбца** (**<ColumnDefinition>**) с фиксированной шириной: 100, 200 и 100 пикселей;

- **Две строки** (<RowDefinition>) с фиксированной высотой: 50 и 100 пикселей.

Это создаёт сетку **3 × 2**, в которую можно помещать элементы управления, указывая в какие строки и столбцы они попадают с помощью атрибутов Grid.Row и Grid.Column.

Обратите внимание как это выглядит в дизайнерае:



Шаг 5. Типы размеров в сетке: Fixed, Auto и Star (*)

Теперь, когда мы познакомились с основами размещения элементов в Grid, давайте углубимся в работу с **разметкой строк и столбцов**, а также рассмотрим три типа размеров, которые можно задавать:

- **Fixed (фиксированный размер)** — задаётся числом (например, 100), означает строго заданную ширину или высоту в пикселях.
- **Auto** — размер рассчитывается автоматически, в зависимости от содержимого ячейки.
- **Star (*)** — относительный размер, при котором свободное пространство делится пропорционально между ячейками.

Удаляем позиционирование через Margin

В предыдущем примере мы использовали Margin для позиционирования Label, но это не очень удобно и плохо масштабируется. Вместо этого используем строки и столбцы Grid.

Удалим у метки все лишние свойства, оставим только Content и зададим ей позицию с помощью Grid.Row и Grid.Column:

```
<Label Grid.Row="1" Grid.Column="1" Content="Hello World!"/>
```


- Grid.Row="1" — вторая строка (нумерация начинается с нуля);

- Grid.Column="1" — второй столбец.

Пример с Auto

Аналогично можно задать столбцы:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="100"/>
  <ColumnDefinition Width="Auto"/>
  <ColumnDefinition Width="100"/>
```




- Средний столбец получит ширину, необходимую для отображения содержимого.
- Если, например, текст в метке станет длиннее, столбец автоматически расширится.

Пример со звёздочкой (*)

Теперь зададим среднему столбцу относительный размер:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="100"/>
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="100"/>
</Grid.ColumnDefinitions>
```



В этом случае:

- Левый и правый столбцы по 100 пикселей;
- Средний столбец получит всё **оставшееся пространство** — это и есть поведение *.

Комбинируем

Теперь можно комбинировать все типы размеров:

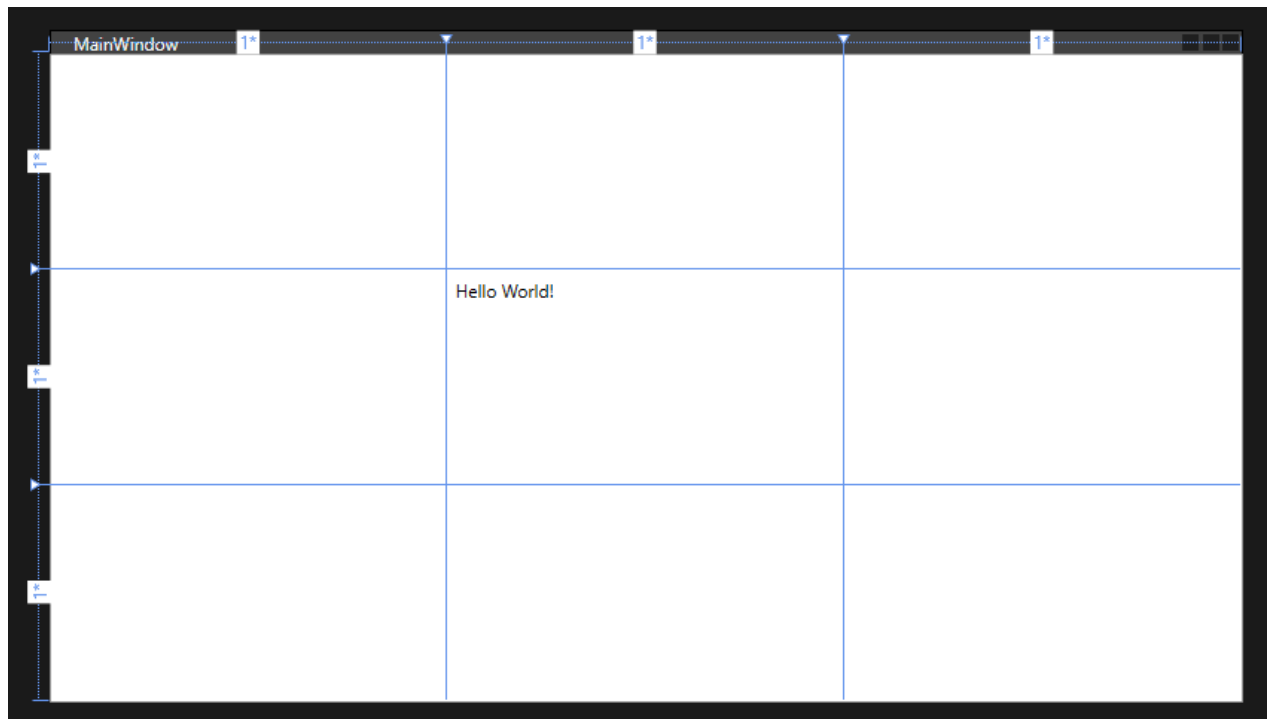
- использовать Auto для адаптации под содержимое;
- использовать * для равномерного или пропорционального распределения пространства;
- использовать фиксированные значения, если нужно строгое выравнивание.

Это позволяет создавать **адаптивные, масштабируемые интерфейсы**, не зависящие от абсолютных координат и Margin.

Если мы везде поставим *:

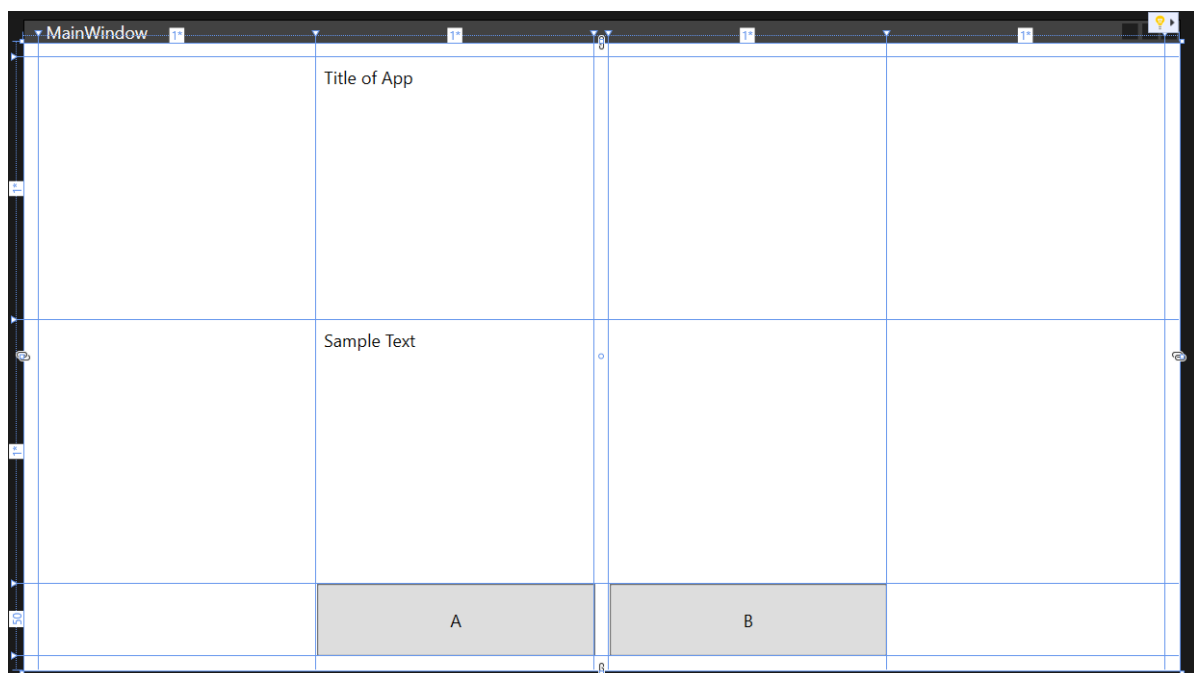
```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
  <RowDefinition Height="*" />
  <RowDefinition Height="*" />
  <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

То наша сетка автоматически заполнит все ряды и столбцы :



Шаг 6. Практика создания Grid

Попробуйте самостоятельно создать следующую сетку :



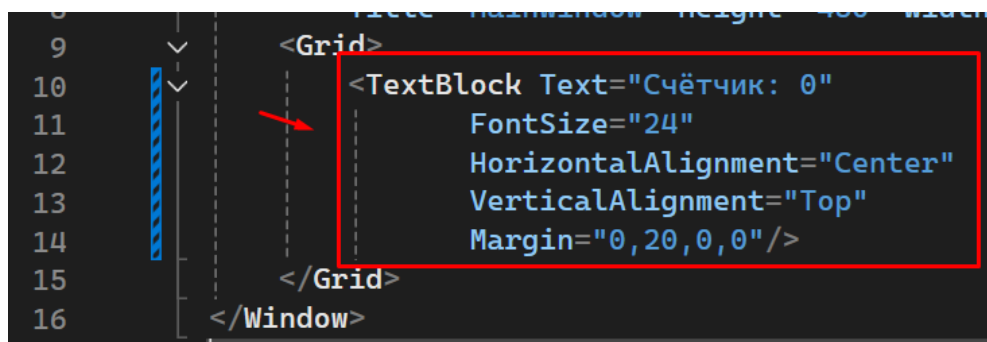
Решение:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="10"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="10"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="10"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="10"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="50"/>
    <RowDefinition Height="10"/>
  </Grid.RowDefinitions>
  <Label Grid.Row="1" Grid.Column="2" Content="Title of App"/>
  <Label Grid.Row="2" Grid.Column="2" Content="Sample Text"/>
  <Button Grid.Row="3" Grid.Column="2" Content="A"/>
  <Button Grid.Row="3" Grid.Column="4" Content="B"/>
</Grid>
```

Приложение Кликер

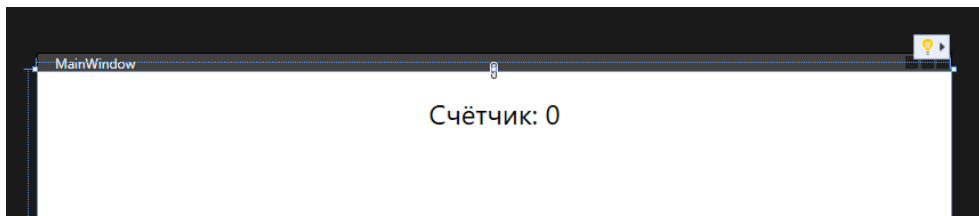
1. Создайте проект с именем **Clicker_WPF**.

Добавим текст – **TextBlock**



- **<TextBlock>** — элемент для вывода текста (аналог Label в WF).
- **Text="Счётчик: 0"** — отображаемое значение.
- **FontSize="24"** — увеличиваем размер шрифта.
- **HorizontalAlignment="Center"** — выравнивание по центру.
- **Margin** — отступ сверху (в пикселях).

Теперь вы уже **видите текст**, и это первый визуальный элемент:



Добавляем кнопку — Button

Теперь под текстом добавим кнопку. Для этого добавим **Button** внутрь **Grid** и обернём всё в **StackPanel**:

```
<Grid>
  <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
    <TextBlock Text="Счётчик: 0"
      FontSize="24"
      HorizontalAlignment="Center"
      VerticalAlignment="Top"
      Margin="0,20,0,0"/>
    <Button Content="Клики меня"
      Width="150"
      Height="40"
      FontSize="16"
      Click="Button_Click"/>
  </StackPanel>
</Grid>
</Window>
```

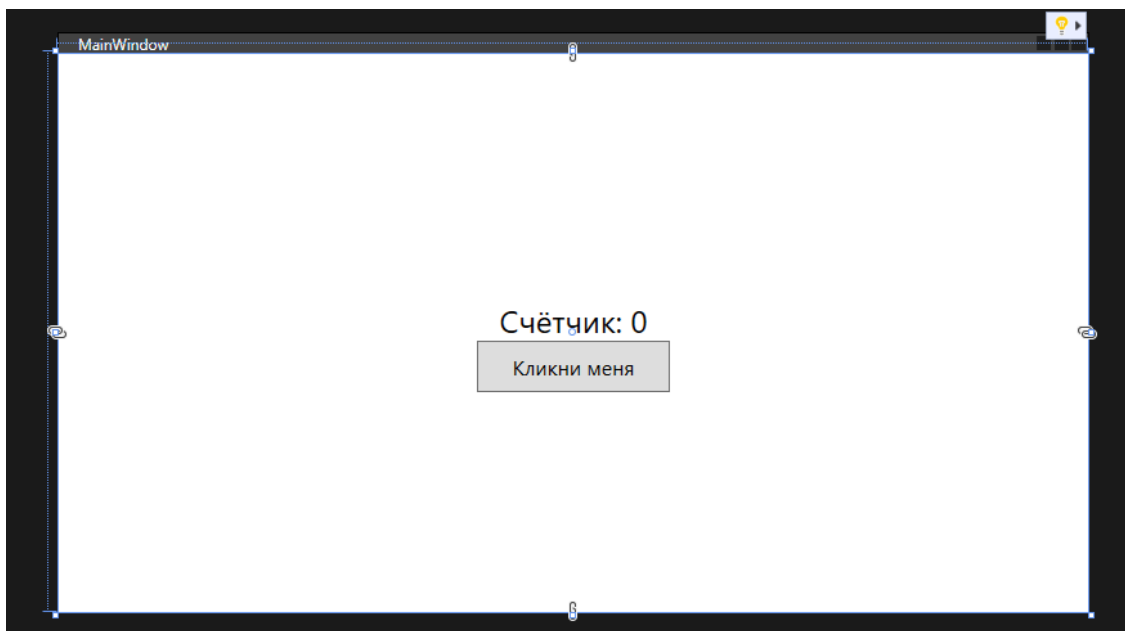
- **<StackPanel>** — размещает элементы **в столбик** (по вертикали).
- **x:Name="CounterText"** — мы дали имя TextBlock, чтобы позже изменить его текст в коде.
- **<Button>** — создаёт кнопку.
- **Content="Клики меня"** — надпись на кнопке.
- **Click="Button_Click"** — указывает, что при нажатии будет вызываться метод Button_Click.

Обратите внимание:

- Все визуальные элементы (TextBlock, Button) — **самозакрывающиеся теги**, если внутри них нет других элементов.
- Свойства элемента (Text, Width, Height и т.д.) — задаются как **атрибуты тега**.
- Мы работаем с **иерархией элементов**: окно → Grid → StackPanel → элементы внутри.

После добавления этого кода и запуска программы у вас отобразится:

- Текст: **"Счётчик: 0"**
- Кнопка: **"Клики меня"**



Пишем логику нажатия кнопки

Теперь, когда мы создали визуальный интерфейс в XAML (текст и кнопку), пора добавить логику: чтобы при нажатии на кнопку число увеличивалось. Перейдите во вкладку **MainWindow.xaml.cs**. Это файл, где мы пишем C#-код, управляющий интерфейсом. В начале удалим неиспользуемые пространства имён:

```
1  using System.Text;
2  using System.Windows;
3  using System.Windows.Controls;
4  using System.Windows.Data;
5  using System.Windows.Documents;
6  using System.Windows.Input;
7  using System.Windows.Media;
8  using System.Windows.Media.Imaging;
9  using System.Windows.Navigation;
10 using System.Windows.Shapes;
```

Оставим:

```
1  using System.Windows;
2
3  namespace Clicker_WPF
```

По умолчанию часто появляется такой блок автосгенерированного XML-комментария, удалим его, т.к. он не несёт полезной информации:

```
namespace Clicker_WPF
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 references
    public partial class MainWindow : Window
```

Теперь создадим переменную-счётчик, которая будет хранить количество нажатий:

```
2 references
public partial class MainWindow : Window
{
    private int counter = 0;
    0 references
    public MainWindow()
```

Перейдите обратно в XAML и нажмите F12 на надписи Click="Button_Click" в теге <Button>:

```
15      Margin="0,20,0,0"/>
16      <Button Content="Клики меня"
17      Width="150"
18      Height="40"
19      FontSize="16"
20      Click="Button_Click"/>
21  </StackPanel>
22  </Grid>
23  </Window>
```

Visual Studio автоматически создаст метод Button_Click в коде MainWindow.xaml.cs:

```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
}
```

Теперь запишем внутри него основную логику:

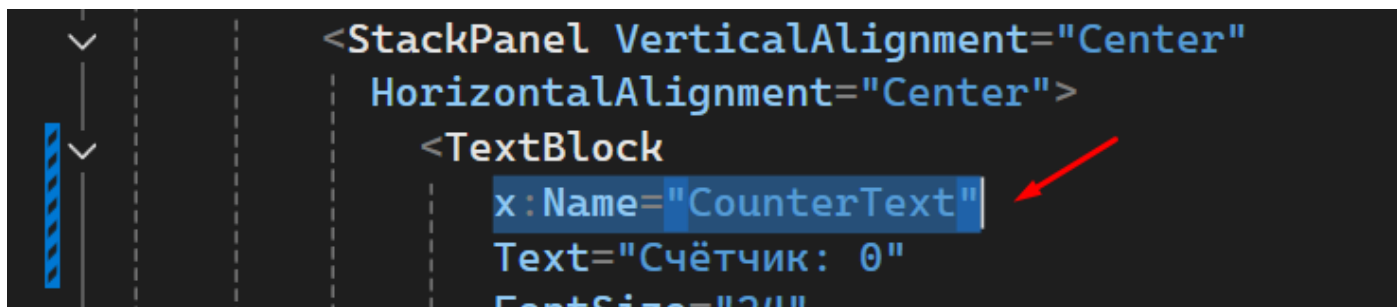
```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
    counter++;
}
```

Это увеличивает значение переменной counter каждый раз при нажатии.

Если вы сейчас запустите приложение (F5), оно откроется, **но внешне ничего не изменилось**. Почему?

Потому что мы **не обновляем текст на экране**. Переменная **counter** меняется, но текст в интерфейсе — нет.

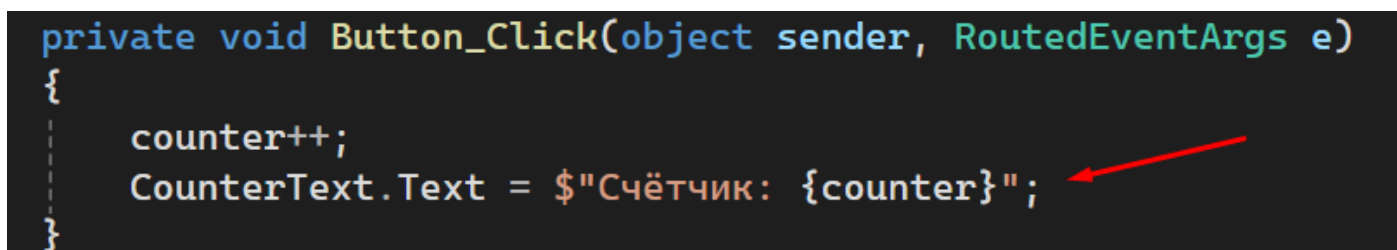
Чтобы программа могла менять текст в `TextBlock`, нужно дать ему имя в XAML:
x:Name="CounterText"



```
<StackPanel VerticalAlignment="Center"
HorizontalAlignment="Center">
    <TextBlock
        x:Name="CounterText"
        Text="Счётчик: 0"
        FontSize="24"/>
```

Теперь в **MainWindow.xaml.cs** мы можем обращаться к нему по имени.

Допишем в обработчик нажатия кнопку:



```
private void Button_Click(object sender, RoutedEventArgs e)
{
    counter++;
    CounterText.Text = $"Счётчик: {counter}";
}
```

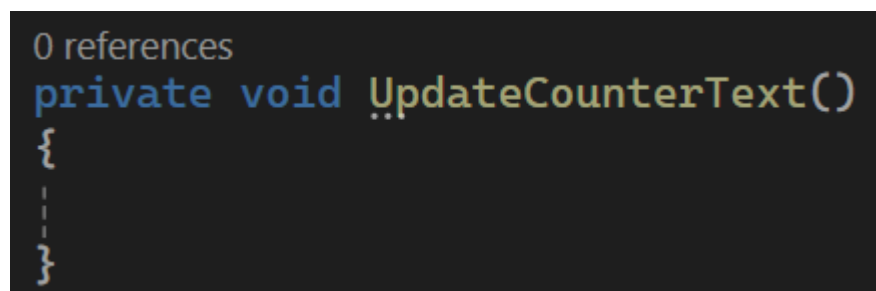
Теперь нажмите **Пуск (F5)**. При каждом клике по кнопке текст будет меняться.

Рефакторинг кода (улучшенная версия)

Если мы посмотрим на текущий код, то увидим проблемы:

1. Дублирование строки `CounterText.Text = $"Счётчик: {counter}";`;
2. Счётчик изменяется напрямую, без централизованного обновления интерфейса.

Поэтому давайте улучшим код. Для начала напишем новый метод `UpdateCounterText()`:



```
0 references
private void UpdateCounterText()
{
    ...
}
```

В него мы вынесем обновление текста:

```
private void UpdateCounterText()
{
    CounterText.Text = $"Счётчик: {counter}";
}
```

Вызовем его в событиях кнопки:

```
1 reference
private void Button_Click(object sender, RoutedEventArgs e)
{
    counter++;
    UpdateCounterText();
}

0 references
private void Button_Reset(object sender, RoutedEventArgs e)
{
    counter = 0;
    UpdateCounterText();
}

0 references
private void UpdateCounterText()
{
    CounterText.Text = $"Счётчик: {counter}";
}
```

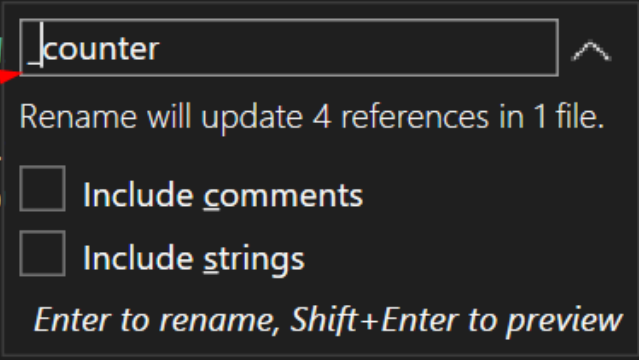
- Теперь логика отображения счётчика находится в одном месте.
- Если формат текста нужно изменить (например, добавить "Кликов: N"), правка делается **единожды**.

Также вызовем его в **MainWindow()**:

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

Далее поменяем название переменной counter на `_counter`, потому что так они обозначаются для лучшей читаемости. Для этого выберите переменную counter, и нажмите **Ctrl+R+R**, введите **новое имя** и нажмите **Enter**:

```
{
    private int _counter = 0;
    0 references
    public MainWindow
    {
        Initiali
        UpdateCo
    }
    1 reference
    private void Button_Click(object sender, RoutedEventArgs)
    {
        _counter++;
        UpdateCounterText();
    }
}
```



counter

Rename will update 4 references in 1 file.

☐ Include comments

☐ Include strings

Enter to rename, Shift+Enter to preview