1. Aplikacja zawiera gotowy interfejs kalkulatora do liczb całkowitych. Korzystając z modelu MVVM uzupełnij aplikację o szereg poleceń (Commands), tak by kalkulator faktycznie działał.
2. Aplikacja powinna zawierać ViewModel. Powinna korzystać z Commands oraz parametrów do Commands.
3. Wciskanie kolejnych cyfr oraz poleceń powinno budować zapis całości działań, na przykład wciskając przycisk 1, następnie przycisk +, następnie przycisk 7, następnie przycisk dzielenia “/” a na końcu przycisk 5, w na ekranie aplikacji powinniśmy uzyskać ciąg:

   1+7/5
4. Wciśnięcie przycisku “=” powinno wyzwolić polecenie, które obliczy wynik. Do wykonania obliczeń możesz skorzystać na przykład z metody “Compute” obiektu DataTable:

   ```
   DataTable dt = new DataTable();
   var result = dt.Compute(moj_string_z_ciagiem_polecen, "");
   ```
5. Zabezpiecz aplikację tak, by nie można było wprowadzić na ekran dwóch operatorów działań pod rząd, np: 1+*0
6. Wciśnięcie przycisku “C” powinno czyścić ekran

Tworzenie nowego projektu

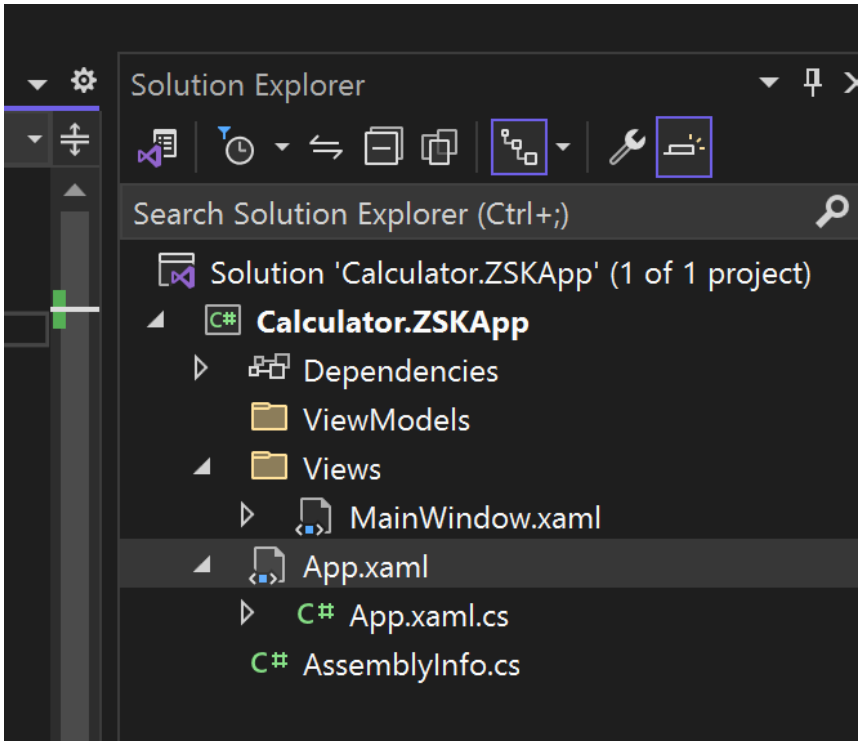
Tworzymy nowy projekt WPF Application, wybierz język C#

Kliknij przycisk Next. Wpisz odpowiednią nazwę dla naszego projektu. Może to być: **„Calculator.ZSKApp”**.
Kliknij Next. Wybierz odpowiedni framework, w naszym przypadku będzie to .NET 7.0. Przycisk Create i projekt zostanie dodany do naszej solucji.

Przygotowanie folderów pod MVVM

Będziemy stosować wzorzec MVVM, czyli Model View ViewModel. Dlatego dodamy na początek odpowiednie foldery, żeby na poziomie solucji już oddzielić te wszystkie warstwy.

Dodaj proszę najpierw folder Views i folder ViewModels. Do folderu Views przeniosę od razu cały plik MainWindow.xaml, czyli widok główny.



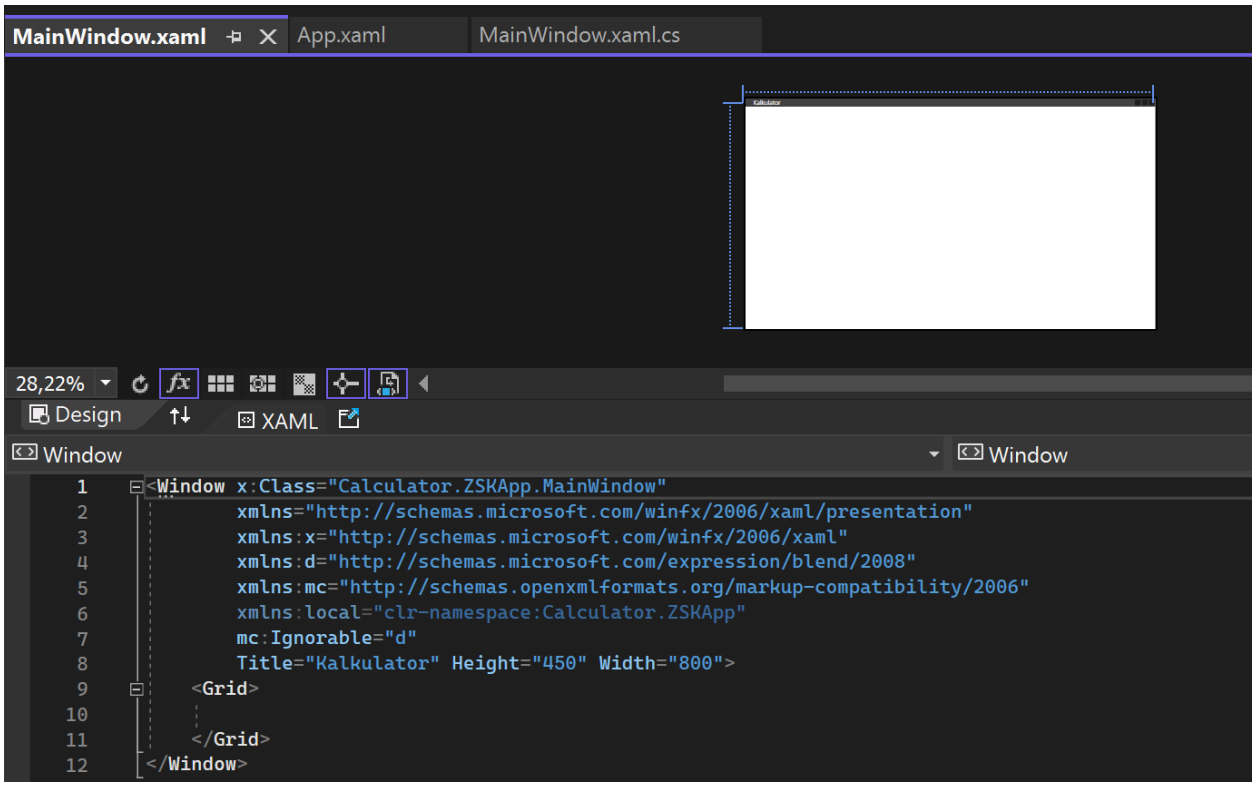
Oprócz tego jeszcze w pliku App.xaml trzeba uwzględnić te zmiany, to znaczy fakt, że MainWindows.xaml znajduje się w folderze Views.

```
<Application x:Class="Calculator.ZSKApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:Calculator.ZSKApp"
             StartupUri="Views/MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Tworzenie widoku w XAML

Widok główny naszej aplikacji będziemy tworzyć właśnie w MainWindow.xaml.



Na początek w XAML możemy sobie zmienić tytuł naszej formatki na "Kalkulator":

```
<Window x:Class="Calculator.WpfApp.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"
```

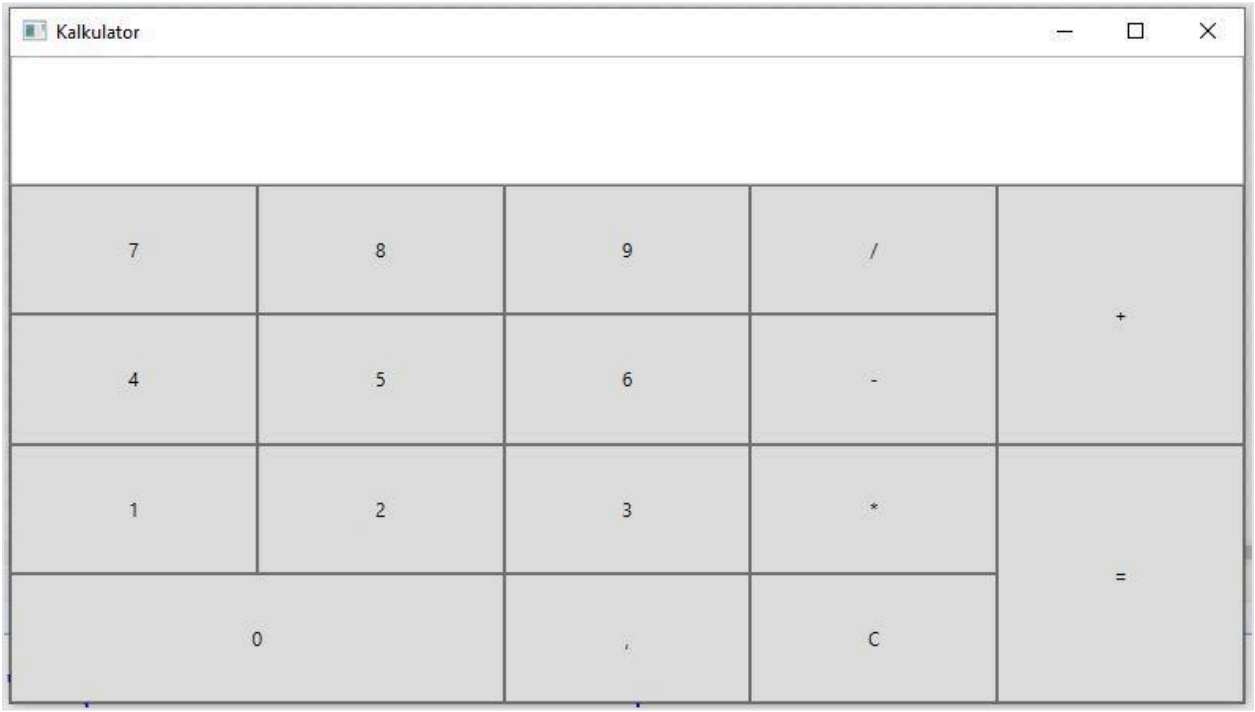
```
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Calculator.WpfApp"
        mc:Ignorable="d"
        Title="Kalkulator" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Także mamy tutaj już utworzonego Grida, skopiuj poniższe jego definicję:

```
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBox Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="5" />
        <Button Grid.Row="1" Grid.Column="0" Content="7" />
        <Button Grid.Row="1" Grid.Column="1" Content="8" />
        <Button Grid.Row="1" Grid.Column="2" Content="9" />
        <Button Grid.Row="2" Grid.Column="0" Content="4" />
        <Button Grid.Row="2" Grid.Column="1" Content="5" />
        <Button Grid.Row="2" Grid.Column="2" Content="6" />
        <Button Grid.Row="3" Grid.Column="0" Content="1" />
        <Button Grid.Row="3" Grid.Column="1" Content="2" />
        <Button Grid.Row="3" Grid.Column="2" Content="3" />
        <Button Grid.Row="4" Grid.Column="0" Content="0" Grid.ColumnSpan="2" />
        <Button Grid.Row="4" Grid.Column="2" Content="," />
        <Button Grid.Row="1" Grid.Column="3" Content="/" />
        <Button Grid.Row="2" Grid.Column="3" Content="-" />
        <Button Grid.Row="3" Grid.Column="3" Content="*" />
        <Button Grid.Row="4" Grid.Column="3" Content="C" />
        <Button Grid.Row="1" Grid.Column="4" Content="+" Grid.RowSpan="2" />
        <Button Grid.Row="3" Grid.Column="4" Content="=" Grid.RowSpan="2" />
    </Grid>
```

Także tak wygląda nasz widok główny, możemy teraz uruchomić aplikację i zobaczyć jak to faktycznie będzie wyglądało.



ViewModel

Logika naszego kalkulatora będzie w ViewModelu, dlatego dodajmy najpierw nową klasę we wcześniej utworzonym folderze ViewModels, w którym będzie ten ViewModel. Ta klasa może mieć dowolną nazwę, niech to będzie MainViewModel. Nasz każdy ViewModel powinien implementować interfejs INotifyPropertyChanged.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Calculator.ZSKApp.ViewModels
{
    internal class MainViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler? PropertyChanged;
    }
}
```

Nie zapomnij na górze dodać odpowiedniego using'a, to znaczy System.ComponentModel. Dodałem także deklaracje zdarzenia PropertyChanged, które powinno być wywołany wtedy, gdy zmieni się jakaś właściwość. Najlepiej od razu dodać sobie taką dodatkową metodę (najczęściej o nazwie OnPropertyChanged), w której będziemy wywoływać to zdarzenie, zaoszczędzi nam to trochę czasu, bo będzie ona wywoływana w wielu miejscach. Najczęściej będziemy tę metody wywoływać we właściwościach, a konkretnie w set'ach, w celu zaktualizowania widoku.

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Calculator.WpfApp.ViewModels
{
    public class MainViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Do metody OnPropertyChanged przekazujemy tylko nazwę właściwości (propertyName), która powinna zostać odświeżona, domyślnie jest to null. Dodatkowo ten parametr został oznaczony atrybutem CallerMemberName, dzięki któremu domyślnie zostanie do tej metody przekazana nazwa właściwości, w której metoda zostanie wywołana. Czyli jeżeli wywołamy metodę w set właściwości o nazwie np. FirstName, bez przekazania parametru, to do metody i tak zostanie przekazany FirstName. Także będzie trochę mniej kodu do napisania.

Wewnątrz metody za pomocą Invoke wyzwalamy event PropertyChanged, jeżeli nie jest null'em. Tak może wyglądać ta metoda i za chwilę pokażę Ci, jak będziemy jej używać. Dodamy za chwilę kilka właściwości, gdzie wykorzystamy metodę OnPropertyChanged.

Powiązanie widoku z ViewModelem

Na początek jednak, żebyśmy nie zapomnieli, musimy powiązać ten ViewModel z widokiem. Tak naprawdę, możemy to zrobić na kilka sposobów. Możemy to zrobić w widoku

w XAMLu, gdzie wystarczy ustawić DataContext, ale myślę, że łatwiejszym i szybszym sposobem jest po prostu wpisanie 1 linjki kodu w CodeBehind, czyli w pliku MainWindow.xaml.cs

```
DataContext = new MainViewModel();
```

Tak będzie wyglądała cała klasa MainWindows.xaml.cs:

```
using Calculator.ZSKApp.ViewModels;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace Calculator.ZSKApp
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = new MainViewModel();
        }
    }
}
```

Takim zapisem wskazujemy, że dla tego okna MainWindow – ViewModelem jest klasa MainViewModel.

RelayCommand

Aby móc powiązać również zdarzenia z widoków z metodami z ViewModelu potrzebujemy jeszcze jednej pomocniczej klasy. Dodajmy sobie najpierw nowy folder Commands, a w nim dodamy klasę RelayCommand. Ta nazwa jest oczywiście dowolna. Ważne, żeby implementowana ona interfejs ICommand.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace Calculator.ZSKApp.Commands
{
    public class RelayCommand : ICommand
    {
        readonly Action<object> _execute;
        readonly Predicate<object> _canExecute;

        public RelayCommand(Action<object> execute)
            : this(execute, null)
        {
        }

        public RelayCommand(Action<object> execute, Predicate<object> canExecute)
        {
            _execute = execute ?? throw new ArgumentNullException("execute");
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return _canExecute == null || _canExecute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add
            {
                CommandManager.RequerySuggested += value;
            }
            remove
            {
                CommandManager.RequerySuggested -= value;
            }
        }
```

```
        public void Execute(object parameter)
        {
            _execute(parameter);
        }
    }


}
```

Czyli mamy dwa delegaty. 1 to jest Action, który będzie przyjmował parametr object, a drugi Predicate, który również może przyjmować parametr object i zwraca boola.

```
readonly Action<object> _execute;
readonly Predicate<object> _canExecute;
```

Poniżej mamy dwa konstruktory. Jeżeli będziemy tworzyć nowy obiekt tej klasy, to jako parametr konstruktora możemy przekazać dwie metody. Jedna metoda, która ma się wykonać po wyzwoleniu jakiegoś zdarzenia, a druga mówi nam o tym, czy ta metoda może się wykonać.

```
public RelayCommand(Action<object> execute, Predicate<object> canExecute)
{
    _execute = execute ?? throw new ArgumentNullException("execute");
    _canExecute = canExecute;
}
```

Czyli jeżeli przekażemy jakąś metodę, która zwraca false, to ta akcja się nie wykona.

Możesz też utworzyć nowy obiekt, tylko przekazując metodę, która ma zostać powiązana z tą konkretną akcją i wtedy jako drugi parametr zostanie przekazany null.

```
public RelayCommand(Action<object> execute)
    : this(execute, null)
{
}
```

Poniżej mamy metodę CanExecute, która jeżeli CanExecute będzie null'em lub będzie zwracała true, to wtedy cała metoda CanExecute zwraca true. Dalej mamy jeszcze metodę Execute, która po prostu wywołuje przekazaną wcześniej metodę. Także, tak może wyglądać przykładowa implementacja klasy RelayCommand. Pokażę Ci teraz, jak wygląda właśnie to wiązanie metod i właściwości.

Binding właściwości

Przejdź proszę do widoku głównego, czyli MainWindow.xaml. Mamy tutaj na górze zwykłego TextBox'a. W TextBox'ie możemy wyświetlić, czy też wpisać dowolny tekst. Możemy to zrobić na różne sposoby, to znaczy po pierwsze możemy np. uzupełnić właściwość Text.

```
<TextBox Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="5" Text="123" />
```

To jest najprostszy sposób, ale też najrzadziej wykorzystywany, ponieważ mamy tutaj przypisany tekst na stałe. Może to być przydane tylko w niektórych przypadkach. Zazwyczaj jednak ten tekst chcemy zmieniać podczas działania programu. Trochę lepszy rozwiązaniem będzie nadanie nazwy tej kontrolce i później odnoszenie się do niej w CodeBehind. Czyli wtedy tak by to wyglądało:

```
<TextBox Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="5" Name="tbScreen" />
```

A w CodeBehind:

```
tbScreen.Text = "123";
```

To wszystko jak najbardziej zadziała. Tylko my chcemy, aby nasza aplikacja korzystała ze wzorca MVVM i nie chcemy tutaj za bardzo wpisywać żadnego kodu w CodeBehind. Całą logikę chcemy mieć w naszym ViewModelu. Także, aby to zrobić prawidłowo, musimy powiązać te dane z odpowiednimi właściwościami z MainViewModel. Wygląda to tak, w widoku wiążemy te dane za pomocą Binding:

```
<TextBox Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="5" Text="{Binding ScreenVal}"
/>
```

Czyli mówimy tutaj, że chcemy powiązać właściwość Text tej konkretnej kontrolki TextBox z właściwością o nazwie ScreenVal, którą zaraz utworzymy w naszym ViewModelu, który z kolei powiązany jest z tym widokiem.

Tworzymy w MainViewModel wraz z polem taką właściwość:

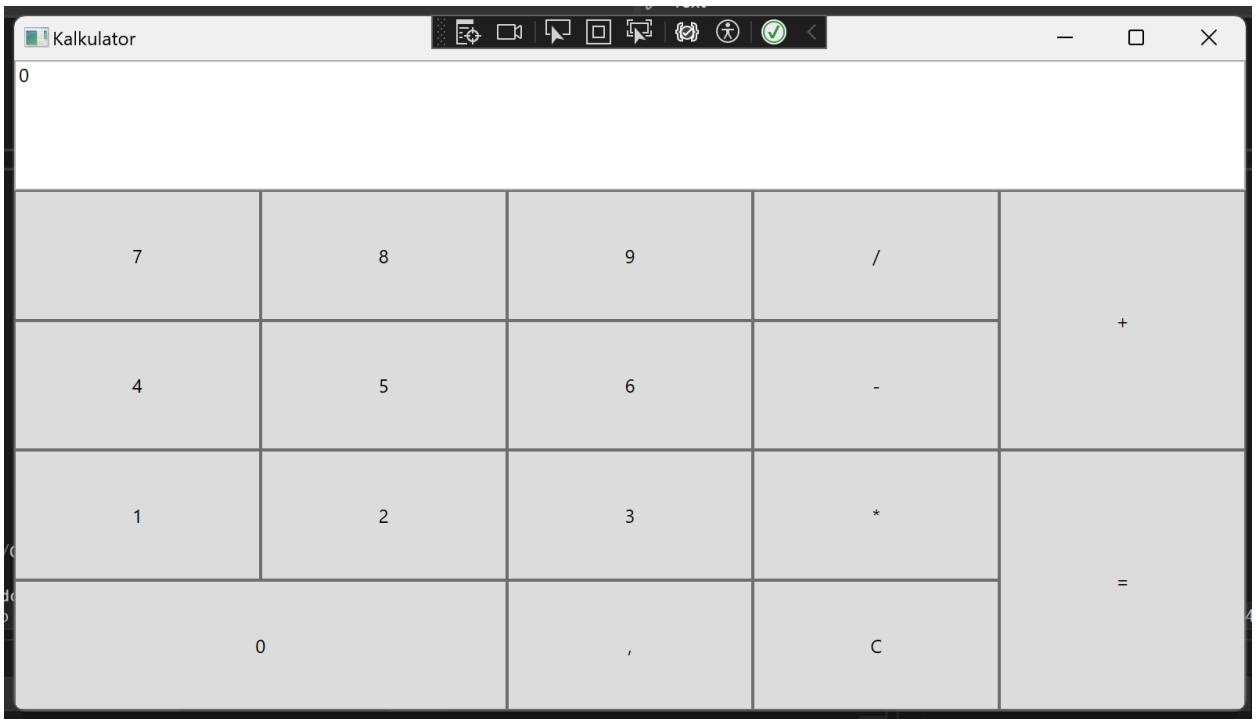
```
private string _screenVal;
public string ScreenVal
{
    get { return _screenVal; }
    set
    {
        _screenVal = value;
        OnPropertyChanged();
```

```
    }
}
```

Ta właściwość powinna być string'iem (tak samo jako właściwość Text TextBox'a). Tak jak Ci wspominałem wcześniej, w set jeszcze musimy wywołać naszą wcześniej stworzoną metodę OnPropertyChanged, tak żeby po zmianie w ViewModelu odświeżyły się dane w widoku. Bez wywołania tej metody, widok nie zostałby zaktualizowany. Teraz, jeżeli zmienimy wartość tej właściwości, możemy to np. zrobić w konstruktorze, to widok zostanie zaktualizowany.

```
public MainViewModel()
{
    ScreenVal = "0";
}
```

Tak będzie wyglądać aplikacja po uruchomieniu:



Jak widzisz, widok został zaktualizowany. Wszystko działa, tak jak powinno. Czyli mamy w widoku TextBox, którego właściwość Text jest powiązana z właściwością ScreenValue z ViewModelu. Następnie w ViewModelu mamy taką właściwość i teraz, jeżeli będziemy zmieniać wartość tej właściwości, to automatycznie będzie to wszystko wyświetlane na widoku. Dzięki temu, że w set właśnie mamy OnPropertyChanged, czyli ten event PropertyChanged tutaj jest wyzwalany, to widok tak jakby "widzi", że to się zmieniło w

ViewModelu i każda zmiana będzie od razu wyświetlana na widoku. Tak wygląda wiązanie z właściwościami.

Binding metod

Oprócz tego jeszcze chcemy oczywiście powiązać nasze zdarzenia z metodami. Do tego użyjemy komend (Command) i ponownie wiązania (Binding). Przejdźmy ponownie do widoku MainWindow i dodaj wiązanie do komendy na początek dla przycisku 7, tak to może wyglądać:

```
<Button Grid.Row="1" Content="7" Command="{Binding AddNumberCommand}" />
```

Mówimy w tym miejscu, że chcemy powiązać zdarzenie Click dla naszego przycisku z komendą AddNumberCommand, która zostanie zaraz stworzona w ViewModelu. Jeżeli przycisk zostanie kliknięty, to zostanie wywołana odpowiednia metoda.

Przejdź ponownie do MainViewModel. Dodamy tutaj wymaganą komendę, która również będzie właściwością:

```
public ICommand AddNumberCommand { get; set; }
```

Następnie musimy zainicjalizować tą komendę w konstruktorze:

```
public MainViewModel()
{
    ScreenVal = "0";
    AddNumberCommand = new RelayCommand(AddNumber);
}
```

Tutaj tworzymy instancje klasy RelayCommand, którą przed chwilą stworzyliśmy. Zauważ, że jako parametr do konstruktora RelayCommand przekazaliśmy metodę o nazwie AddNumber, także potrzebujemy stworzyć taką metodę.

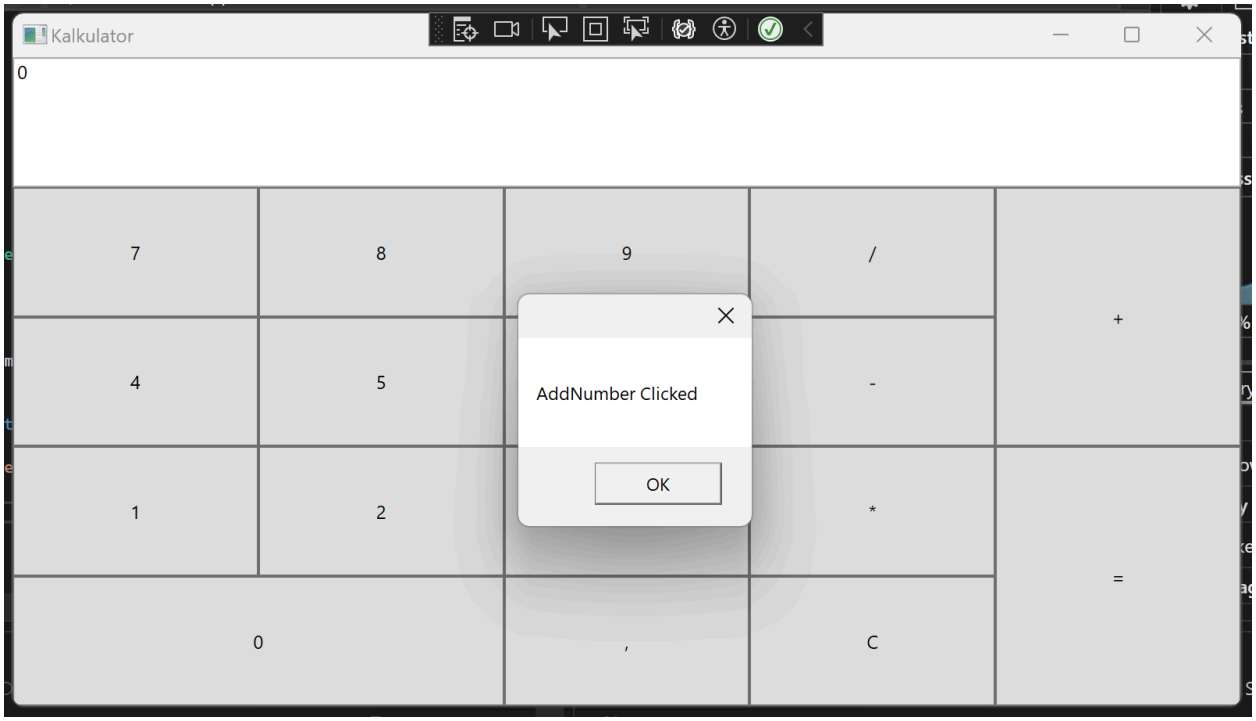
```
private void AddNumber(object obj)
{
}
```

Czyli dokładnie ta metoda zostanie wywołana, gdy zostanie kliknięty przycisk 7. Możemy to sprawdzić. W tym celu możesz wyświetlić w metodzie AddNumber odpowiedni komunikat:

```
private void AddNumber(object obj)
{
    MessageBox.Show("AddNumber Clicked");
```

}

Sprawdź teraz, czy taki komunikat u Ciebie się pojawia.



U mnie wszystko na tę chwilę działa prawidłowo. Za chwilę tutaj stworzymy już faktyczną logikę.

Czyli jeszcze raz wytłumaczę Ci jak działy cały ten mechanizm. Mamy w widoku przycisk, na tę chwilę podpięliśmy się tylko pod przycisk z 7 i wiążemy tutaj jego zdarzenie Click z komendą AddNumberCommand, która jest w ViewModelu. Komenda o takiej nazwie musi być właściwością w ViewModelu. Następnie np. w konstruktorze inicjalizujemy tę komendę i mówimy, że chcemy, żeby po kliknięciu została wywołana metoda AddNumber, która wyświetli nam w tej chwili taki testowy komunikat.

Przekazanie parametru do ViewModelu

Myślę, że możemy przejść dalej. Także, tak samo będziemy chcieli podpiąć tę komendę i metodę pod inne przyciski, ale też właśnie musimy jakoś rozróżniać, żebyśmy wiedzieli który przycisk został kliknięty. Także oprócz tego, że mamy powiązane komendę, to chcemy w widoku jeszcze przekazać parametr. W tym celu musimy do CommandParameter przypisać, to co chcemy przekazać, czyli w tym przypadku będzie to 7.

```
<Button Grid.Row="1" Content="7" Command="{Binding AddNumberCommand}"
CommandParameter="7" />
```

Dzięki temu do metody AddNumber jako parametr zostanie przekazana ta 7.

```
private void AddNumber(object obj/*7*/)
{
    MessageBox.Show("AddNumber Clicked");
}
```

Możemy też to przetestować. W metodzie jako parametr dostajemy object, także musimy go rzutować na string'a i spróbujemy to wyświetlić na ekranie.

```
private void AddNumber(object obj)
{
    MessageBox.Show(obj as string);
}
```

Pozostałe wiązanie danych

Pozostaje nam jeszcze powiązanie reszty przycisków, tak żeby wszędzie to tak działało, czyli to samo będzie dla 0, 1, 2, 3, 4, 5, 6, 7, 8 i 9, a także dla ",".

```
<Button Grid.Row="1" Content="7" Command="{Binding AddNumberCommand}"
CommandParameter="7" />

<Button Grid.Row="1" Grid.Column="1" Content="8" Command="{Binding AddNumberCommand}"
CommandParameter="8" />

<Button Grid.Row="1" Grid.Column="2" Content="9" Command="{Binding AddNumberCommand}"
CommandParameter="9" />

<Button Grid.Row="2" Content="4" Command="{Binding AddNumberCommand}"
CommandParameter="4" />

<Button Grid.Row="2" Grid.Column="1" Content="5" Command="{Binding AddNumberCommand}"
CommandParameter="5" />

<Button Grid.Row="2" Grid.Column="2" Content="6" Command="{Binding AddNumberCommand}"
CommandParameter="6" />

<Button Grid.Row="3" Content="1" Command="{Binding AddNumberCommand}"
CommandParameter="1" />
```

```
<Button Grid.Row="3" Grid.Column="1" Content="2" Command="{Binding AddNumberCommand}"
CommandParameter="2" />

<Button Grid.Row="3" Grid.Column="2" Content="3" Command="{Binding AddNumberCommand}"
CommandParameter="3" />

<Button Grid.Row="4" Grid.ColumnSpan="2" Content="0" Command="{Binding
AddNumberCommand}" CommandParameter="0" />

<Button Grid.Row="4" Grid.Column="2" Content="," Command="{Binding AddNumberCommand}"
CommandParameter="," />
```

W każdym przypadku do ViewModelu zostanie przekazany odpowiedni parametr i na tej podstawie rozróżnimy, który przycisk został kliknięty.

Trochę inna logika będzie dla naszych operacji, może sobie od razu powiązać przyciski operacji z kolejną komendą o nazwie AddOperationCommand.

```
<Button Grid.Row="1" Grid.Column="3" Content="/" Command="{Binding
AddOperationCommand}" CommandParameter="/" />

<Button Grid.Row="2" Grid.Column="3" Content="-" Command="{Binding
AddOperationCommand}" CommandParameter="-" />

<Button Grid.Row="3" Grid.Column="3" Content="*" Command="{Binding
AddOperationCommand}" CommandParameter="*" />

<Button Grid.Row="1" Grid.Column="4" Grid.RowSpan="2" Content="+" Command="{Binding
AddOperationCommand}" CommandParameter="+" />
```

Tak samo, jak wcześniej, również tutaj przekażemy odpowiedni parametr. Następnie w ViewModelu dodamy nową właściwość i zainicjalizujemy ją w konstruktorze.

```
public ICommand AddOperationCommand { get; set; }

public MainViewModel()
{
    ScreenVal = "0";
    AddNumberCommand = new RelayCommand(AddNumber);
    AddOperationCommand = new RelayCommand(AddOperation);
}
```

W konstruktorze przekażemy metodę AddOperation, którą zaraz stworzymy.

```
private void AddOperation(object obj)
```

```
{
}
```

Potrzebujemy jeszcze osobną komendę do przycisku Clear, także w widoku wpisujemy:

```
<Button Grid.Row="4" Grid.Column="3" Content="C" Command="{Binding
ClearScreenCommand}" />
```

Dodajemy taką właściwość w ViewModelu, inicjalizujemy w konstruktorze.

```
public ICommand ClearScreenCommand { get; set; }

public MainViewModel()
{
    ScreenVal = "0";
    AddNumberCommand = new RelayCommand(AddNumber);
    AddOperationCommand = new RelayCommand(AddOperation);
    ClearScreenCommand = new RelayCommand(ClearScreen);
}
```

Tworzymy metodę, która ma zostać wywołana.

```
private void ClearScreen(object obj)
{
}
```

I tak samo zrobimy dla przycisku, który będzie wyświetlał wynik, w tym przypadku powiążemy przycisk z komendą GetResultCommand.

```
<Button Grid.Row="3" Grid.Column="4" Grid.RowSpan="2" Content="=" Command="{Binding
GetResultCommand}" />
```

Nowa komenda w ViewModelu, inicjalizacja w konstruktorze i dodajemy nową metodę GetResult.

```
public ICommand GetResultCommand { get; set; }

public MainViewModel()
{
    ScreenVal = "0";
    AddNumberCommand = new RelayCommand(AddNumber);
    AddOperationCommand = new RelayCommand(AddOperation);
    ClearScreenCommand = new RelayCommand(ClearScreen);
    GetResultCommand = new RelayCommand(GetResult);
}
```

```
private void GetResult(object obj)
{
}
```

Logika w ViewModelu

Mamy już teraz powiązane wszystkie właściwości i wszystkie zdarzenia w naszym widoku, także teraz tylko musimy uzupełnić ich logikę w ViewModelu i wtedy nasza aplikacja będzie kompletna.

Zacznijmy od metody AddNumber. Najpierw interesuje nas to jaki został przekazany parametr do tej metody, także przypiszemy go do zmiennej number.

```
var number = obj as string;
```

Następnie sprawdzimy najpierw, czy obecnie na tym naszym ekranie jest wyświetlane 0 i liczba, która została przekazana, jest różna od przecinka. Jeżeli tak, to ScreenVal ustawiamy na string empty. W przeciwnym przypadku musimy sprawdzić, czy został przekazany przecinek i wcześniej została wybrana operacja, jeżeli tak, to musimy ten nasz przecinek przekazać razem z zerem.

```
if (ScreenVal == "0" && number != ",")
    ScreenVal = string.Empty;
else if (number == "," &&
_availableOperations.Contains(ScreenVal.Substring(ScreenVal.Length - 1)))
    number = "0,";
```

A tak wygląda lista operacji:

```
private List<string> _availableOperations = new List<string> { "+", "-", "/", "*" };
```

Także mamy nowe pole, jest to lista stringów z dostępnymi operacjami. Od razu zostało zainicjalizowane, czyli mamy 4 dostępne operację. Są to: dodawanie, odejmowanie, dzielenie i mnożenie. Zweryfikowaliśmy także, czy ostatnim znakiem była tutaj operacja. Jeżeli tak, to chcemy przekazać przed przecinkiem jeszcze 0, tak żeby później nie było jakichś dziwnych błędów. Na koniec, to co zostało wpisane, wyświetlamy na ekran. Doklejamy to do tych znaków, które były już tam wcześniej.

```
ScreenVal += number;
```

Tak wygląda na tę chwilę cała metoda:

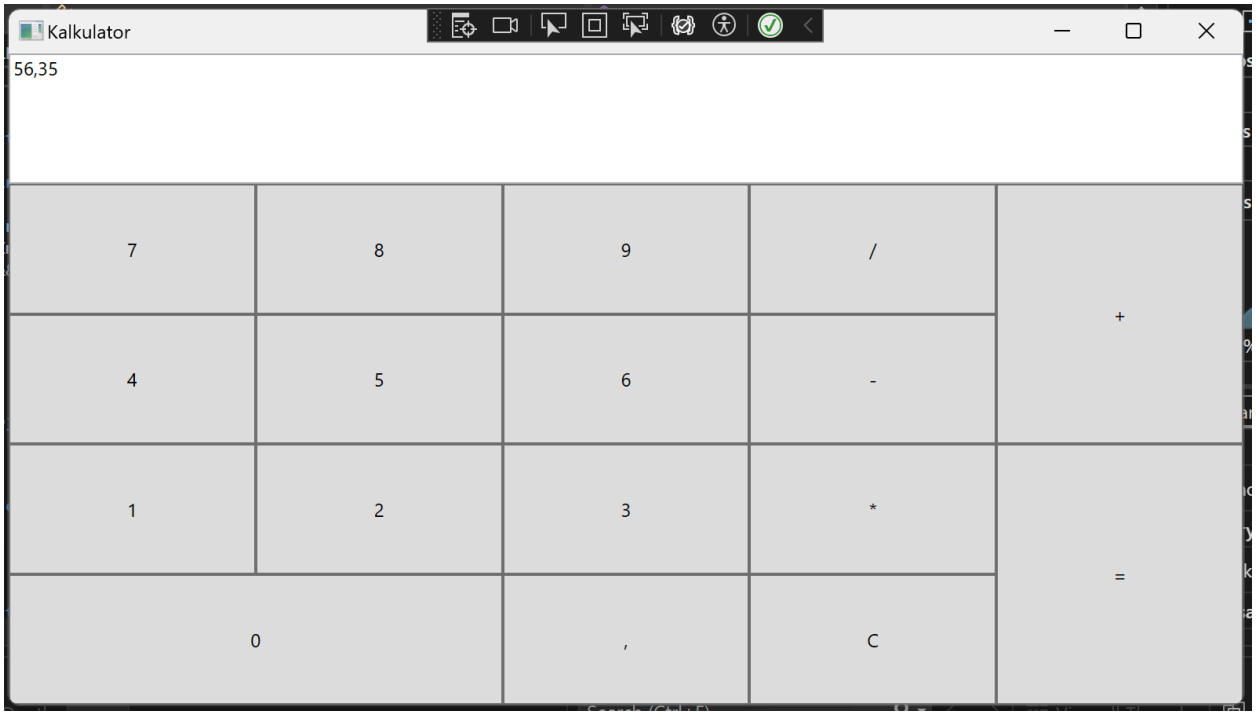
```
private void AddNumber(object obj)
```

```
{
    var number = obj as string;

    if (ScreenVal == "0" && number != ",")
        ScreenVal = string.Empty;
    else if (number == "," &&
_availableOperations.Contains(ScreenVal.Substring(ScreenVal.Length - 1)))
        number = "0,";

    ScreenVal += number;
}
```

Możemy teraz zobaczyć czy to działa.



Także tutaj już się wszystko ładnie wyświetla. Jeżeli wpiszemy cyfrę, to wyświetla się prawidłowo, jeżeli wpiszemy jako pierwszy znak przecinek, to również jest ok. Także możemy już dodawać cyfry i przecinek.

Potrzebujemy również dodać operację, czyli zaimplementować metodę AddOperation. Tak samo najpierw musimy sprawdzić, co tutaj w ogóle zostało przekazane, która dokładnie operacja została kliknięta.

```
var operation = obj as string;
```

Następnie wyświetlamy ją na ekranie.

```
ScreenVal += operation;
```

Tak będzie wyglądać metoda AddOperation:

```
private void AddOperation(object obj)
{
    var operation = obj as string;

    ScreenVal += operation;
}
```

Wszystko jest ok. Następnie możemy zaimplementować metodę ClearScreen. Tutaj wystarczy do właściwości ScreenVal przypisać 0. Tak może wyglądać ta metoda:

```
private void ClearScreen(object obj)
{
    ScreenVal = "0";
}
```

Ok, pozostaje nam jeszcze wyświetlenie wyniku. Tym razem jednak ten wynik będziemy wyświetlać trochę inaczej niż w poprzednich aplikacjach. Możemy to zrobić trochę łatwiejszym sposobem. Jest taka klasa DataTable, która ma metodę Compute, dzięki której możemy obliczyć przekazany ciąg znaków, który jest string'iem. Także stwórzmy sobie najpierw nowe pole:

```
private DataTable _dataTable = new DataTable();
```

Trzeba dodać odpowiedni using – System.Data. Teraz użyjemy już metody Compute do naszych obliczeń.

```
var result = _dataTable.Compute(ScreenVal, "");
```

Przekazujemy wartość właściwości ScreenVal i przekazujemy filtr, a może to być po prostu pusty string. Przypisujemy to, co zwróci ta metoda do nowej zmiennej result. Na koniec wyświetlamy ten wynik na ekranie:

```
ScreenVal = result.ToString();
```

Takie rozwiązanie jest ok, ale jeszcze nie dla wszystkich sytuacji. Musimy jeszcze zabezpieczyć się przed błędami związanymi z nieodpowiednim formatowaniem. To znaczy, przed przekazaniem wartości do metody Compute musimy zastąpić przecinek – kropką:

```
var result = _dataTable.Compute(ScreenVal.Replace(",", "."), "");
```

Możemy sobie jeszcze wynik zaokrąglić do 2 miejsc po przecinku. Wystarczy skorzystać z metody Round, statycznej klasy Math.

```
var result = Math.Round(Convert.ToDouble(_dataTable.Compute(ScreenVal.Replace(",",
"."), "")), 2);
```

I cała metoda może wyglądać w ten sposób:

```
private void GetResult(object obj)
{
    var result =
Math.Round(Convert.ToDouble(_dataTable.Compute(ScreenVal.Replace(",", "."), "")), 2);

    ScreenVal = result.ToString();
}
```

Teraz można przetestować aplikację, zobaczmy czy to działa. Konieczne jest sprawdzenie różnych przypadków.

Standardowa ścieżka działa prawidłowo. Jednak trzeba pamiętać, że użytkownicy klikają w najróżniejsze sposoby i trzeba zabezpieczyć się przed wszystkimi przypadkami.

Zamiast wprowadzić liczbę po wybraniu operacji, można wybrać kolejne operacje i w takiej sytuacji nasza aplikacja nie zadziałała prawidłowo. Nie możemy dopuścić do takiej sytuacji. Musimy się zabezpieczyć przed taką sytuacją. Na szczęście nie jest to trudne. Możemy sobie dodać nowe pole:

```
private bool _isLastSignAnOperation;
```

Które, jak sama nazwa wskazuje, będzie przechowywała informacje o tym, czy ostatni znak jest operacją. Jeżeli będzie operacją, to nie możemy pozwolić na kliknięcie ponownie na kolejną operację, tylko w takiej sytuacji może zostać wybrana liczba. Musimy w kilku miejscach ustawić wartość tego pola.

Po pierwsze, jeżeli klikniemy numer, to znaczy, że ostatnim znakiem nie jest operacja, także możemy ustawić wtedy to pole na false.

```
private void AddNumber(object obj)
{
    var number = obj as string;

    if (ScreenVal == "0" && number != ",")
        ScreenVal = string.Empty;
```

```
    else if (number == "," &&
_availableOperations.Contains(ScreenVal.Substring(ScreenVal.Length - 1)))
        number = "0,";

    ScreenVal += number;

    _isLastSignAnOperation = false;
}
```

Jeżeli zostanie kliknięta operacja, to oczywiście ustawiamy na true.

```
private void AddOperation(object obj)
{
    var operation = obj as string;

    ScreenVal += operation;

    _isLastSignAnOperation = true;
}
```

Jeżeli zostanie kliknięty przycisk Clear, to również ustawiamy na false.

```
private void ClearScreen(object obj)
{
    ScreenVal = "0";

    _isLastSignAnOperation = false;
}
```

Teraz na podstawie tego pola chcemy blokować również niektóre przyciski. Chcemy blokować przycisk operacji, tak żeby dwie operacje nie zostały kliknięte jedna po drugiej i też chcemy blokować przycisk result, ponieważ nie chcemy, żeby w sytuacji, gdy ostatni znak jest operacja, to był obliczany wynik. Aby to zrobić, wystarczy w konstruktorze dla komend AddOperationCommand oraz GetResultCommand uzupełnić ten drugi parametr konstruktora RelayCommand, o którym wcześniej Ci wspominałem, gdzie zdefiniujemy, kiedy chcemy, żeby ta metoda była dostępna.

```
AddOperationCommand = new RelayCommand(AddOperation, CanAddOperation);
GetResultCommand = new RelayCommand(GetResult, CanGetResult);
```

Następnie dodamy implementacje tych metod:

```
private bool CanGetResult(object obj)
{
    return !_isLastSignAnOperation;
}
```

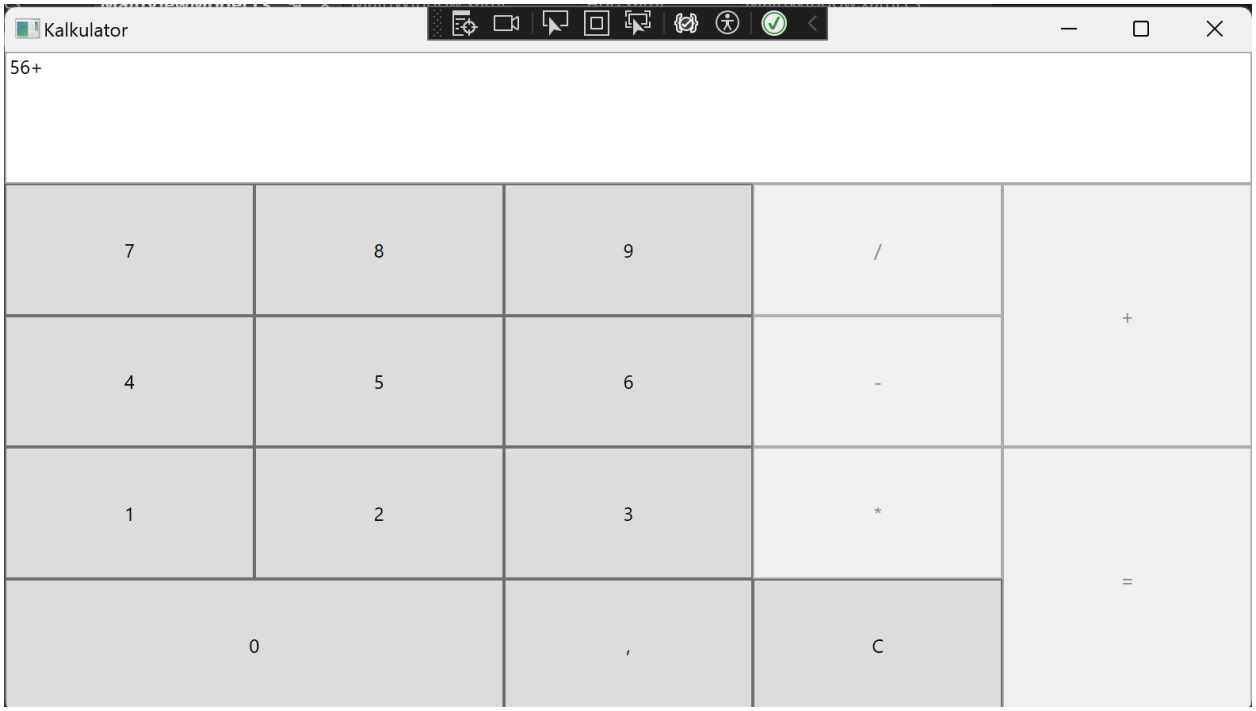
```
private bool CanAddOperation(object obj)
{
    return !_isLastSignAnOperation;
}
```

Chcemy, żeby te metody mogły zostać wywołane tylko wtedy gdy pole isLastSignAnOperation ma wartość false. Możemy sobie również skrócić wersję tych metod i zapisać ją w ten sposób:

```
private bool CanGetResult(object obj) => !_isLastSignAnOperation;

private bool CanAddOperation(object obj) => !_isLastSignAnOperation;
```

Zobaczmy czy to działa.



Zauważ, że po wybraniu operacji, nie ma już teraz możliwości wybrania kolejnej. Trzeba wybrać liczbę, także nie ma możliwości wybrania kilku operacji jedna po drugiej, ponieważ zostały zablokowane. Zawsze po wybraniu operacji będziemy oczekiwać na podanie liczby i o to chodziło.