Лабораторна робота №2. Вступний приклад розробки WPF-застосунку

Ціль : Ознайомитись з основами розробки WPF-застосунку у вигляді текстового редактора

Завдання 1. Освоїти основи роботи з XAML та контейнерами компонування

Основні теоретичні відомості

XAML (Extensible Application Markup Language — мова розмітки програм, що розширюється) —мова розмітки, яка використовується для створення екземплярів об'єктів .NET. Його головне призначення - конструювання інтерфейсів WPF. Іншими словами, документи XAML визначають розташування панелей, кнопок та інших елементів керування, що становлять вікна у додатку WPF.

Найпростіший документ XAML (у Visual Studio):

```
<Window x:Class="WindowsApplicationl.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2 00 6/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="300" Width="300">
        <Grid>
        </Grid>
</Window>
```

Він містить всього два елементи - елемент верхнього рівня ***Window***, який представляє все вікно, і елемент ***Grid***, куди можна помістити свої елементи керування.

Використання контейнерів компонування дає можливість розміщеним усередині цих контейнерів компонентам позиціонуватися разом із зміною розмірів форми. Існують такі контейнери компонування: ***Grid***, ***StackPanel***, ***WrapPanel***, ***Canvas***, ***DockPanel***.

Контейнер Grid

Це контейнер дозволяє швидко розмістити компоненти всередині своїх невидимих рядків і стовпців.

Хоча ***Grid*** задуманий як невидимий елемент, можна встановити властивість ***Grid.ShowGridLines*** у **true** та отримати наочне уявлення про нього.

Створення компонування на основі ***Grid*** - двокроковий процес:

1. Спочатку вибирається необхідна кількість колонок та рядків.
2. Потім кожному елементу призначається відповідний рядок і колонка.

Колонки та рядки створюються шляхом заповнення об'єктами колекції ***Grid.ColumnDefinitions*** та ***Grid.RowDefinitions*** . Наприклад, якщо потрібно два рядки і три колонки, можна використовувати такі дескриптори:

```
<Grid ShowGridLines="True">
        <!-- Встановлюємо два рядки -->
        <Grid.RowDefinitions>
```

```
        <RowDefinition></RowDefinition>

        <RowDefinition></RowDefinition>

    </Grid.RowDefinitions>

    <!-- Встановлюємо три стовпці -->

    <Grid.ColumnDefinitions>

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

        <ColumnDefinition Width="2*"></ColumnDefinition>

    </Grid.ColumnDefinitions>

    <!-- Розміщуємо елементи у сітці -->

    < TextBox Grid.Row="0" Grid.Column="0">TextBox</ TextBox >

    <Button Grid.Row="0" Grid.Column="1">Button1</Button>

    < Label Grid.Row="1" Grid.Column="1"> Label </ Label>

    <Button Grid.Row="1" Grid.Column="2">Button2</Button>

</Grid>
```

Режим зміни розмірів встановлюється за допомогою властивості ***Width*** об'єкта ***ColumnDefinition*** або властивості ***Height*** об'єкта ***RowDefnition***. Наприклад, встановити абсолютну ширину 100 незалежних від пристрою одиниць:

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

Для використання пропорційної зміни розмірів вказується значення ***Auto*** :

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

І, нарешті, для активізації пропорційної зміни розмірів задається зірочка (*):

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

Щоб розділити простір, що залишився, нерівними частинами, можна призначити ваговий показник, який повинен вказуватися перед зірочкою. Наприклад, якщо є два рядки пропорційного розміру, і потрібно, щоб висота першого дорівнювала половині висоти другого, необхідно розділити простір, що залишився, наступним чином:

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

<RowDefinition Height="2*"></RowDefinition>
```

В цьому випадку висота другого рядка буде вдвічі більша за висоту першого рядка.

Можна також використовувати ще дві приєднані властивості, щоб розтягнути елемент на кілька комірок: ***RowSpan*** та ***ColumnSpan***. Ці властивості приймають кількість рядків або стовпчиків, які повинен зайняти елемент.

Наприклад, наступна кнопка займе все місце, доступне в першій та другій комірці першого рядка:

<Button Grid.Row=0" Grid.Column=0" Grid.RowSpan=2">Span Button</Button>

Контейнер StackPanel

Цей контейнер корисний тоді, коли потрібно помістити компоненти в колонці (вертикально) або один за одним (у рядок). Таким чином, цей компонент має властивість ***Orientation***, яка за замовчуванням дорівнює ***Vertical***, але яку можна встановити в ***Horizontal***.

Нижче наведено приклад контейнера з двома елементами управління **Label** і **TextBox**:

<StackPanel>

<Label>Label</Label>

<TextBox>TextBox</TextBox>

</StackPanel>

Панелі компонування взаємодіють зі своїми дочірніми елементами через набір властивостей компонування:

Найменування	Опис
HonzontalAlignment	Визначає позиціонування дочірнього елемента всередині контейнера компонування, коли є додатковий простір по горизонталі. Доступні значення: ***Center***, ***Left***, ***Right*** або ***Stretch.***
VerticalAlignment	Визначає позиціонування дочірнього елемента в контейнері компонування, коли є додатковий простір по вертикалі. Доступні значення: ***Center***, ***Top***, ***Bottom*** або ***Stretch*** .
Margin	Додає деякий простір навколо елемента для верхньої, нижньої, лівої та правої граней.
MinWidth та ***MinHeight***	Встановлює мінімальний розмір елемента. Якщо елемент занадто великий, щоб помістится в його контейнер компонування, він буде усічений.
MaxWidth та ***MaxHeight***	Встановлює максимальний розмір елемента. Якщо контейнер має вільний простір, елемент не буде збільшений понад зазначені межі, навіть якщо властивості ***HonzontalAlignment*** і ***VerticalAlignment*** встановлені в ***Stretc*** .
Width та ***Height***	Очевидно встановлюють розміри елемента. Ця установка перевизначає значення ***Stretch*** для властивостей ***HorizontalAlignment*** та ***VerticalAlignment***. Однак цей розмір не буде встановлений, якщо виходить за межі, задані ***MinWidth***, ***MinHeight***, ***MaxWidth*** і ***MaxHeight***

Контейнер WrapPanel

Цей контейнер корисний тим, що може переносити елементи на інший рядок в залежності від ширини простору. Він так само як і **StackPanel** має властивість ***Orientation***.

Нижче наведено приклад контейнера з елементами управління ***Button*** :

<WrapPanel HorizontalAlignment="Left" Height="319" VerticalAlignment="Top" Width="548" Margin="5">

<Button Content="Button1" Width="161" Height="28"/>

```
    <Button Content="Button2" Width="94" Height="116" Margin="12"/>

    <Button Content="Button3" Width="230" Height="102" Margin="12"/>

    <Button Content="Button4" Width="243" Height="43"/>

</WrapPanel>
```

Контейнер DockPanel

Цей контейнер дозволяє розмістити компонент притиснутим до однієї зі своїх сторін. У елементів, що розміщуються всередині контейнера, з'являється властивість ***DockPanel.Dock,*** яка має тип перерахування і може приймати значення: **left**, **right**, **top**, **bottom**.

Приклад роботи з цим контейнером:

```
<DockPanel LastChildFill="False">

    <Button DockPanel.Dock="Right" Height="auto" Width="auto">x</Button>

    <Button DockPanel.Dock="Left" Width="20"></Button>

</DockPanel>
```

Контейнер Canvas

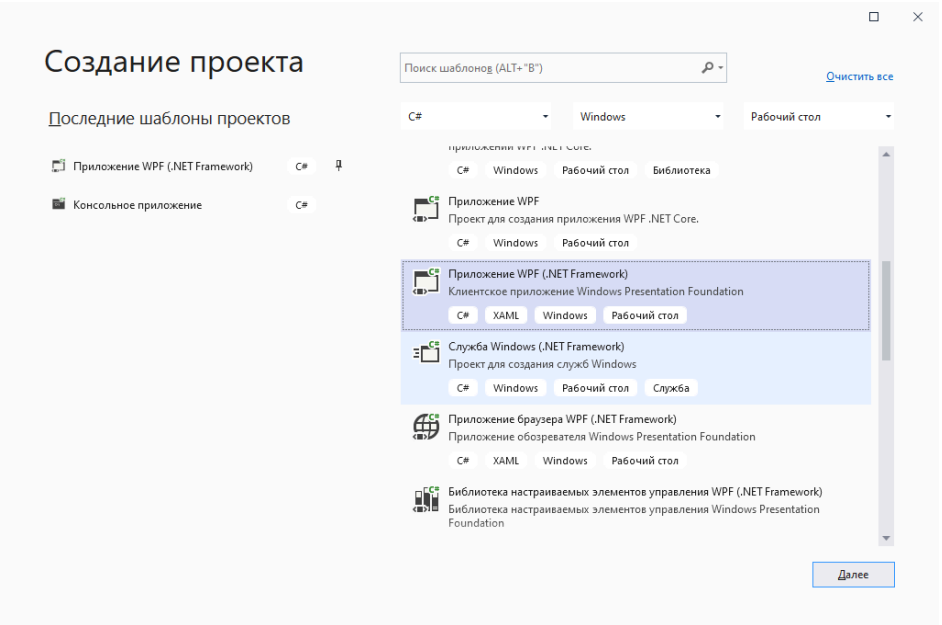
Це спеціальний контейнер, який дозволяє позиціонувати компоненти WPF за абсолютними значеннями. Кожен елемент, розміщений у **Canvas**, отримує приєднані властивості ***Canvas.Top***, ***Canvas.Bottom***, ***Canvas.Left*** та ***Canvas.Right*** . Кожна з цих властивостей дає можливість компоненту всередині контейнера прописати свої позиції на формі. Ці позиції не змінюватимуться навіть із зміною розмірів форми.

Нижче наведено xaml-розмітку, що дозволяє точно позиціонувати ліву верхню вершину кнопки в точку (20,30):

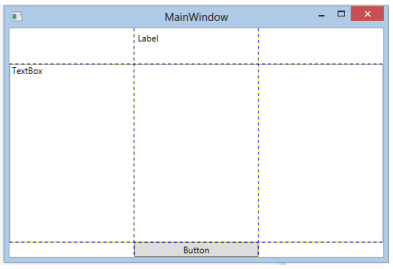
```
<TextBox Canvas.Top="10">text</TextBox>

    <Button Canvas.Top="20" Canvas.Left ="30">button</Button>

</Canvas>
```

Хід виконання

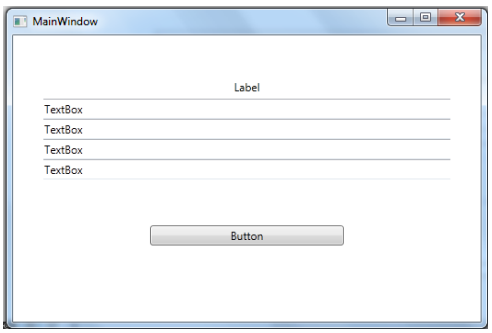
1. Запустіть Visual Studio.
2. Натисніть комбінацію клавіш **Ctrl-Shift-N** або скористайтеся командою **Файл-Новий-Проект** для відкриття вікна створення проекту.
3. У діалозі створення проекту виберіть шаблон **Застосунок WPF (.NET Framework)** і натисніть кнопку **Далі**.



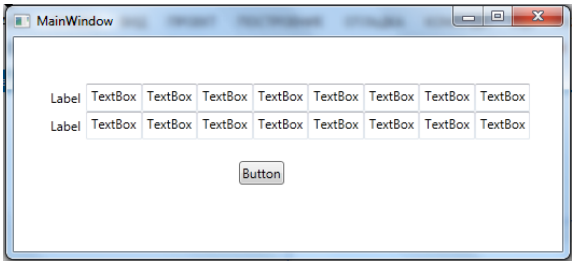
4. В діалозі настройки проекту задайте його ім'я і натисніть кнопку **Створити**.
5. Перейдіть у вікно розмітки XAML.
6. У контейнері ***Grid*** створіть видиму сітку розміром 3x3 і розмістіть у другому стовпці послідовно три елементи: **Label**, **TextBox** та **Button**.
7. Визначте висоту рядків так, щоб перший був у 4 рази меншим за другий, а висота третього визначалася автоматично.
8. Розтягніть елемент **TextBox** на весь другий рядок.
9. Переконайтеся, що виконана розмітка призводить до відображення такого вікна:



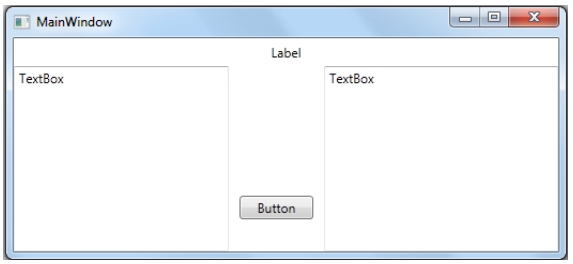
10. Фрагмент отриманої розмітки XAML скопіюйте у звіт про виконання завдання.
11. Замініть контейнер **Grid** на **StackPanel** і розмістіть у ньому елементи **Label**, **TextBox** та **Button** наступним чином:



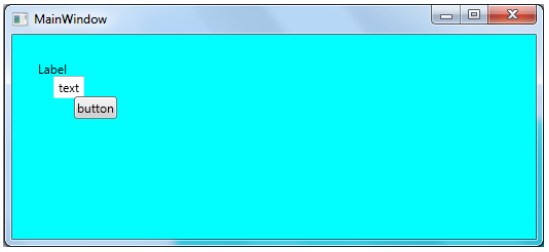
12. Фрагмент отриманої розмітки XAML скопіюйте у звіт про виконання завдання.
13. Замініть контейнер **StackPanel** на **WrapPanel** і розмістіть у ньому елементи **Label**, **TextBox** та **Button** таким чином:



14. Фрагмент отриманої розмітки XAML скопіюйте у звіт про виконання завдання.
15. Замініть контейнер **WrapPanel** на **DockPanel** і розмістіть у ньому елементи **Label**, **TextBox** та **Button** таким чином:



16. Фрагмент отриманої розмітки XAML скопіюйте у звіт про виконання завдання.
17. Замініть контейнер **DockPanel** на **Canvas** і розмістіть в ньому елементи **Label**, **TextBox** і **Button,** зміщені по горизонталі від лівого краю і по вертикалі від верхнього краю відповідно 20,40 та 60 одиниць:



18. Фрагмент отриманої розмітки XAML скопіюйте у звіт про виконання завдання 1.

Завдання 2. Розробити у WPF інтерфейс текстового редактора

Хід виконання

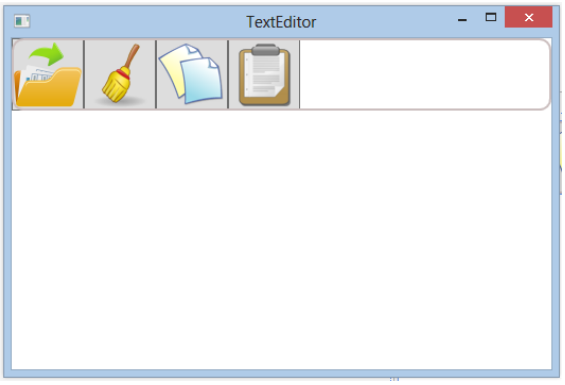
1. У вікні XAML-розмітки у тегу < ***Window*** > для властивості ***Title*** встановіть значення " **TextEditor** ".
2. У вікні XAML-розмітки вставте контейнер **Grid** , що містить два рядки – для меню та текстової області.

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

3. У верхній рядок вставте контейнер **StackPanel** і розмістіть у ньому горизонтально 4 кнопки - для виконання команд **Відкрити**, **Стерти**, **Копіювати**, **Вставити**.
4. Використовуючи *браузер рішень*, додайте в проект папку **images** для зберігання іконок, а в ній розмістіть чотири іконки.
5. Встановіть як контент кнопок відповідні іконки (тег < **Image** >) і задайте для кожної з них підказку (властивість ***ToolTip***).
6. Встановіть для першого рядка **Grid** режим автоматичного визначення висоти.
7. Вставте в перший рядок **Grid** рамку із закругленими кутами, скориставшись декоратором **Border** :

<Border Grid.Row="0" BorderBrush="#FFCBBEBE" BorderThickness="2" CornerRadius="10"> </Border>

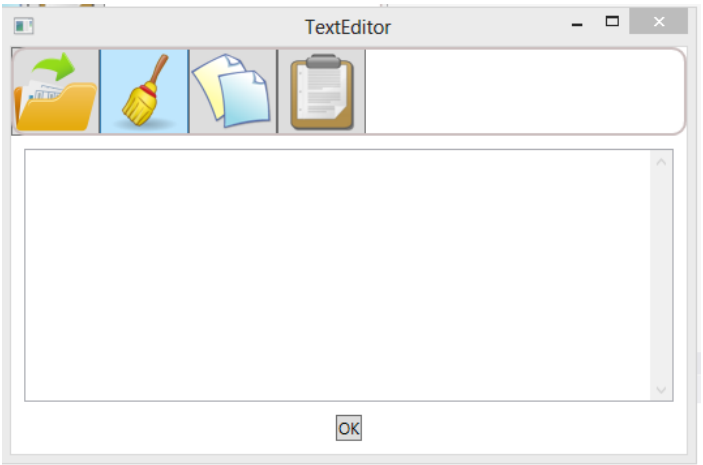
8. Переконайтеся, що отримано наступний результат:



9. Додайте в другий рядок **Grid** рамку із закругленими кутами, скориставшись декоратором **Border** :

<Border Grid.Row="1" BorderBrush="#FFCBBEBE" BorderThickness="2" CornerRadius="10"> </Border>

10. Вставте в декоратор ще один контейнер **Grid**, який містить два рядки з ненульовими полями (margin).
11. У верхньому рядку цього контейнера **Grid** розмістіть текстовий елемент **TextBox** з вертикальною смугою прокручування.
12. У нижньому рядку контейнера **Grid** розмістіть контейнер **DockPanel** і вставте в нього кнопку **OK** , "пристиковану" до нижнього краю.
13. Встановіть для нижнього рядка **Grid** режим автоматичного визначення висоти.
14. Переконайтеся, що отримано наступний результат:



15. Код XAML-розмітки вікна програми скопіюйте у звіт про виконання завдання 2.

Завдання 3. Розробити дизайн програми із застосуванням пензлів, ресурсів, стилів, тригерів

Основні теоретичні відомості

Використання пензлів

Заливку суцільним кольором можна виконати за допомогою властивості ***Background***.

Для градієнтної заливки використовується пензель **LinearGradientBrush**. Такий пензель здатний створити кілька градієнтних областей за допомогою тега < **GradientStop** >. Властивість ***Color*** визначає початковий колір, а властивість ***Offset*** дозволяє задати позицію, звідки починається той чи інший градієнт. Властивість ***Offset*** може бути встановлена в проміжку від 0.0 до 1.1.

Наприклад, задати градієнтну заливку з трьома кольорами можна за допомогою наступної XAML-розмітки:

```
< Grid >
    <Grid.Background>
        <LinearGradientBrush>
            <GradientStop Color="Aqua" Offset="0"></GradientStop>
            <GradientStop Color="Red" Offset="0.5"></GradientStop>
            <GradientStop Color="Black" Offset="0.8"></GradientStop>
        </LinearGradientBrush>
    </ Grid.Background>
</ Grid >
```

Крім пензля лінійної градієнтної **LinearGradientBrish** є ще пензель **SolidColorBrush -** для суцільної заливки кольором, пензель **RadialColorBrush -** для радіальної заливки, **ImageBrush** - для заливки за допомогою малюнка з файлу.

Створення ресурсів

Нехай створену градієнтну заливку потрібно використовувати багаторазово та застосовувати до різних елементів. Тоді визначати градієнтну заливку слід як статичний ресурс програми. Ресурси дозволяють централізовано зберігати об'єкти для їхнього подальшого застосування.

Набір ресурсів програми описується як приєднана властивість **Resources** тега < ***Window>*** . Кожен ресурс має свій унікальний ідентифікатор **x:Key** і за цим ідентифікатором до ресурсу можна звернутися і взяти звідти значення.

Наприклад, створити ресурс з ім'ям для виконання градієнтної заливки трьома кольорами можна так:

```
<Window x:Class="WpfLab1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
Title="TextEditor" Height="350" Width="525">
        <Window.Resources>
                <LinearGradientBrush x:Key="LinearPanelBrush">
                        <GradientStop Color="Aqua" Offset="0"></GradientStop>
                        <GradientStop Color="Red" Offset="0.5"></GradientStop>
                        <GradientStop Color="Black" Offset="0.8"></GradientStop>
                </LinearGradientBrush>
        </Window.Resources>
...
</Window>
```

Після цього можна як **Background** вказати наш створений статичний ресурс. Виглядатиме це так:

```
<Grid Background="{StaticResource linearPanelBrush}"> ...</Grid>
```

Стилі

Стилем називається колекція значень властивостей, які можуть застосовуватись до елемента.

Стилі використовуються для надання елементу користувача оригінального зовнішнього вигляду. До того ж, стилі можна легко переносити з додатку до програми.

Стилі створюються у секції ресурсів. У найпростішому випадку визначити стиль можна задати так:

```
<Window.Resources>
...
        <Style TargetType="Button">
                <Setter Property="Margin" Value="5"></Setter>
                <Setter Property="Background" Value="Transparent"></Setter>
                <Setter Property="BorderBrush" Value="White"></Setter>
        </Style>
</Window.Resources>
```

У коді розмітки стиль описується за допомогою тега < ***Style***>. Всередині тега розміщується колекція ***Setters*** з трьома об'єктами ***Setter***, по одному для кожної властивості, яка підлягає встановленню. У кожному об'єкті ***Setter*** вказується ім'я властивості, на яку він впливає, та значення, яке він має застосовуватись до цієї властивості. Як і всі ресурси, об'єкт стилю має ключове ім'я, за яким до нього можна за необхідності звертатись. За замовчуванням цей стиль застосовується до всіх кнопок вікна.

Якщо не потрібно, щоб кнопка успадкувала стиль, це можна зазначити явно:

```
<Button Style="{x:Null}">Ok</Button>
```

Тригери

Простий тригер може бути приєднаний до будь-якої властивості залежності. Наприклад, реагуючи на зміни у властивостях ***IsFocused***, ***IsMouseOver*** та ***IsPressed*** класу **Control**, можна створити ефекти наведення курсору миші та отримання фокусу.

Тригери зв'язуються зі стилями через колекцію ***Style.Triggers***.

Приклад тригера, який очікує на отримання кнопкою фокуса з клавіатури і в цьому випадку встановлює для неї фон темно-червоного кольору:

```
<Style x:Key="BigFontButton">
    <Style.Setters>
        <Setter Property="Control.FontFamily" Value="Times New Roman" />
        <Setter Property="Control.FontSize" Value="18" />
    </Style.Setters>
    <Style.Triggers>
        <Trigger Property="Control. IsFocused" Value="True">
            <Setter Property="Control. Foreground" Value="DarkRed" />
        </Trigger>
    </Style.Triggers>
</Style>
```

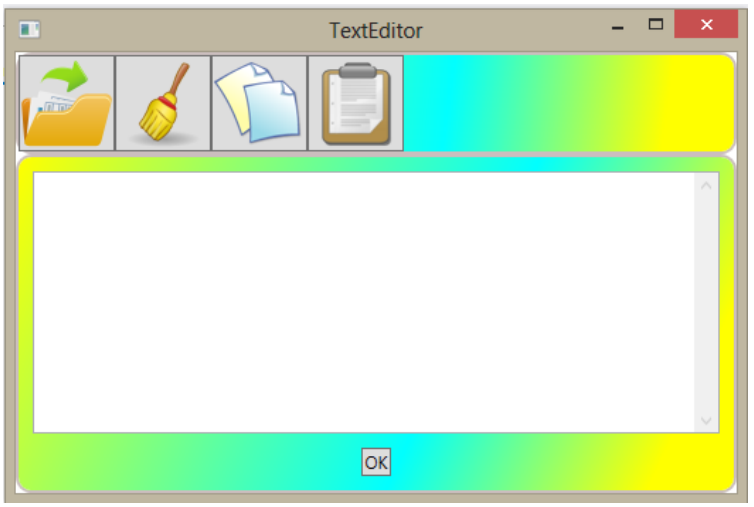
Хід виконання

1. У вікні XAML-розмітки у тегу < ***Window*** > задайте властивість ***Resources*** :

```
<Window>
...
    <Window.Resources>
    </Window.Resources>
...
</Window>
```

2. Опишіть ресурс під ім'ям ***LinearPanelBrush*** для створення градієнтної заливки жовто-блакитно-жовтої.
3. Використайте ресурс для створення градієнтної заливки панелі інструментів та оточення навколо текстового поля.

4. Створіть стиль для оформлення зовнішнього вигляду кнопок із такими параметрами: поле навколо кнопки – 5 одиниць, прозорий фон, границя кнопки біла.
5. Скасуйте використання стилю для кнопки **OK**.
6. Переконайтеся, що отримано наступний результат:



7. Фрагмент коду розмітки XAML створеного ресурсу скопіюйте у звіт про виконання завдання 3.

Завдання 4. Виконати прив'язку команд та елементів

Основні теоретичні відомості

Прив'язування елементів

Прив'язка даних - це відношення, яке повідомляє WPF про необхідність вилучення деякої інформації з вихідного об'єкта та використання його для встановлення властивості цільового об'єкта. Цільова властивість - це завжди властивість залежності.

Прив'язка елементів дозволяє налаштувати реакцію користувача елемента на зміну стану іншого елемента. Щоб показати, як прив'язувати один елемент до іншого, розглянемо просте вікно, яке містить два елементи керування: **Slider** (повзунок) та **TextBlock** (текстовий блок) з єдиним рядком тексту. Переміщення повзунка вправо повинне призводити до збільшення розміру тексту, а переміщення вліво — до зменшення розміру шрифту.

Таку поведінку не важко реалізувати в коді: треба просто реагувати на подію ***Slider.ValueChanged*** і копіювати поточне значення з повзунка в **TextBlock**. Проте прив'язка даних робить це ще простіше.

При використанні прив'язки даних жодних змін у вихідний об'єкт (**Slider**) вносити не потрібно. Тільки задається ім'я та потрібний діапазон значень:

<Slider Name="sliderFontSize" Margin="3" Minimum="1" Maximum="40" Value="10" ></Slider>

Прив'язка визначається в елементі **TextBlock**. Замість встановлення ***FontSize*** з використанням літерального значення застосовується вираз прив'язки:

<TextBlock FontSize="{Binding ElementName=sliderFontSize, Path=Value, UpdateSourceTrigger=PropertyChanged}"/></TextBlock>

Властивості ***FontSize*** призначається прив'язка. Вирази прив'язки даних поміщаються у фігурні дужки. На початку йде слово ***Binding***, в ***ElementName*** вказується той елемент, звідки береться значення, в

Path вказується властивість елемента, у якому зберігається значення, ***UpdateSourceTrigger*** призначає режим оновлення прив'язки (як тільки слайдер зрушить з місця, прив'язка спрацює біля текстового поля).

Прив'язка команд

У WPF весь функціонал програми має бути поділений на команди. До команд можна підключати елементи керування. Команди можуть визначати стан інтерфейсу користувача, і в залежності від поточного стану команди робити доступним або недоступним елемент керування.

Розробники WPF вже створили стандартний набір команд, який включає такі команди як **New**, **Exit**, **Cut**, **Copy** … Всі ці команди знаходяться у статичному класі ***ApplicationCommands***. Користувачу залишається реалізувати логіку роботи команди так, як потрібно застосунку. Будь-яка команда може бути активною та неактивною. У неактивному положенні елемент інтерфейсу, до якого прив'язана команда, повинен автоматично ставати недоступним для використання.

Для того, щоб команда сама розуміла, коли їй треба бути активною, а коли – ні, у команди передбачені дві поді - ***CanExecute*** та ***Execute***. Спочатку виникає подія ***CanExecute***, тут має бути визначена якась логіка, яка перевіряє правильність введення або щось інше. І лише коли ***CanExecute*** відпрацює до кінця, спрацює подія ***Execute***.

Для прив'язки команди необхідно виконати такі дії:

1. Перейти у файл відокремленого коду (*.cs) і в класі вікна задати для неї два приватні обробники подій ***CanExecute*** і ***Execute***:

```
void canExecute_Save(object sender, CanExecuteRoutedEventArgs e){
        if (inputTextBox.Text.Trim().Length > 0) e.CanExecute = true; else e.CanExecute = false; }
void execute_Save(object sender, ExecutedRoutedEventArgs e) {
        System.IO.File.WriteAllText("d:\\myFile.txt", inputTextBox.Text);
        MessageBox.Show("The file was saved!");}
```

2. У конструкторі вікна *MainWindow* створити прив'язку команди, приєднавши до неї ці обробники, та зареєструвати її:

```
//Створення прив'язки та приєднання обробників
CommandBinding saveCommand = new CommandBinding(ApplicationCommands.Save, execute_Save,
canExecute_Save);
//Реєстрація прив'язки
CommandBindings.Add(saveCommand);
```

3. Перейти в XAML-файл і зв'язати команду з елементом керування через його властивість ***Command*** :

```
<Button ToolTip="Зберегти" Command="Save">
```

Елементи керування убудованими командами

Деякі елементи керування введенням можуть обробляти події команд самостійно. Наприклад, клас **TextBox** обробляє команди **Cut**, **Copy** та **Paste** (а також команди **Undo** і **Redo** та частину команд класу **EditingCommands**, які дозволяють виділяти текст та переміщати курсор у різні позиції).

Якщо кнопки розміщені в іншому контейнері (відмінному від ***ToolBar*** і ***Menu***), то вони не працюватимуть доти, доки для них вручну не буде встановлено властивість ***CommandTarget***. Це вимагає використання прив'язки з ім'ям цільового елемента. Наприклад, якщо ім'ям текстового поля є ***txtDocument***, кнопки повинні бути визначені таким чином:

```
<Button Command="Cut" CommandTarget="{Binding ElementName=txtDocument}">Cut</Button>
```

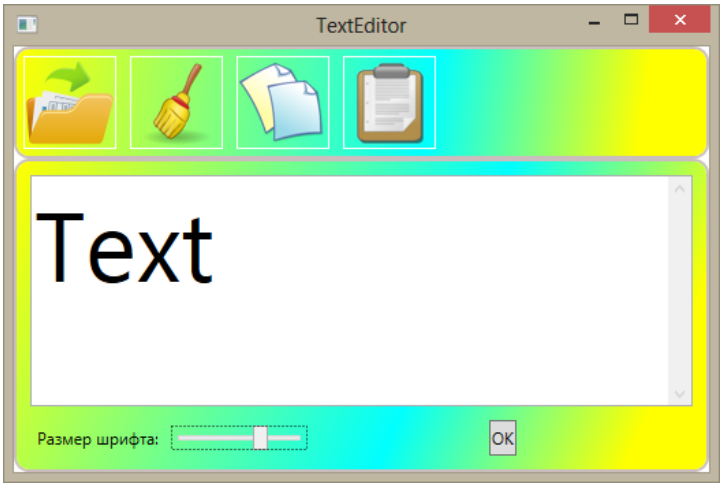
Інший, більш простий варіант передбачає створення нової сфери дії фокусу з використанням приєднаної властивості ***FocusManager.IsFocusScope***. Це змушує WPF при спрацьовуванні команди шукати елемент у дії фокусу батьківського елемента:

```
<StackPanel FocusManager.IsFocusScope="True">
        <Button Command="Cut">Cut</Button>
        <Button Command="Copy">Copy</Button>
        <Button Command="Paste">Paste</Button>
</StackPanel>
```

Такий підхід має додаткову перевагу, оскільки дозволяє застосовувати ті ж самі команди до кількох елементів керування.

Хід виконання

1. Додайте до створеного раніше контейнера **DockPanel** елемент **Slider** для керування розміром шрифту.
2. Задайте елементу **Slider** наступні атрибути: ***Margin="4", MinWidth="100", Name="fontSlider", Maximum="100", Minimum="12".***
3. Додайте в контейнер **DockPanel** до елемента **Slider** елемент ***Label*** для відображення підпису до нього у вигляді тексту ***"Розмір шрифту:".***
4. Прив'яжіть елемент **TextBlock** до **Slider** таким чином, щоб переміщення повзунка призводило до зміни розміру шрифту.
5. Переконайтеся, що отримано наступний результат:



6. Перейдіть до файлу відокремленого коду (*.cs)
7. У класі вікна задайте два приватні обробники подій ***CanExecute*** та ***Execute*** для команди ***Save*** (Зберегти).
8. У конструкторі вікна *MainWindow* виконайте прив'язку команди, приєднавши до неї ці обробники, та зареєструйте її.
9. Перейдіть до файлу XAML і зв'яжіть команду збереження через його властивість ***Command*** з кнопкою **ОК** .
10. Перекомпілюйте проект та перевірте правильність функціонування кнопки.
11. Повторіть пп.6-10 для прив'язування відповідних команд до кнопок **Відкрити**, **Стерти**.
12. Скористайтеся можливостями вбудованих команд для забезпечення функціональності кнопок **Копіювати**, **Вставити**.
13. Перекомпілюйте проект та перевірте функціональність програми.
14. Файли XAML-розмітки та програмного коду скопіюйте у звіт про виконання завдання 4.