М. І. Шахрайчук
Н. В. Шинкарчук
С. В. Петренко

Інтерфейси користувача та системні інтерфейси

Навчальний посібник
Лекційні матеріали

2022

РІВНЕНСЬКИЙ ДЕРЖАВНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА МОДЕЛЮВАННЯ

Шахрайчук М.І.
Шинкарчук Н.В.
Петренко С.В.

Інтерфейси користувача та системні інтерфейси

Навчальний посібник

Лекції

- **для студентів спеціальностей:**

121 «Інженерія програмного забезпечення»
122 «Комп'ютерні науки»
123 «Комп'ютерна інженерія»

- **лекційний курс з чисельними прикладами кодів програм з детальними коментарями**

2022

Н15
УДК 004.5 + 004.451.84 (075.8)

Інтерфейси користувача та системні інтерфейси: навч. посіб. для дисципліни: лекції / [уклад. М.І. Шахрайчук, Н.В. Шинкарчук, С.В. Петренко]. Рівне: РДГУ, 2022. 270 с.

Укладачі: кандидат фізико-математичних наук, професор кафедри інформаційних технологій та моделювання **Шахрайчук М. І.**

кандидат технічних наук, доцент кафедри інформаційних технологій та моделювання **Шинкарчук Н. В.**

кандидат педагогічних наук, доцент кафедри інформаційних технологій та моделювання **Петренко С. В.**

Рецензенти: доктор технічних наук, професор, завідувач кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування **Турбал Ю. В.**

доктор педагогічних наук, професор, завідувач кафедри інформаційно-комунікаційних технологій та методики викладання інформатики Рівненського державного гуманітарного університету **Войтович І. С.**

У цьому посібнику паралельно викладені два підходи програмування, на яких базується Windows Presentation Foundation (WPF). Студент циклічно переходитиме від особливостей застосування XAML (Extensible Application Markup Language) до тонкощів програмування на C#, що дозволить зрозуміти їх взаємозв'язаність при програмуванні інтерфейсів. На початку пояснюється синтаксис XAML і принципи створення взаємозв'язаного коду.

Велике число наочних прикладів допоможуть зрозуміти, як зв'язуються коди на XAML і C#. Усе це плюс роз'яснення концепцій інтерфейсів користувача (UI) навчить студентів – Windows-розробників – тому, як створювати для своїх застосунків інтерфейси нового покоління.

Розглянуто та схвалено на засіданні кафедри інформаційних технологій та моделювання, протокол №6 від 21.06.2022 р.

Розглянуто та рекомендовано до друку навчально-методичною комісією факультету математики та інформатики Рівненського державного гуманітарного університету, протокол № 5-2022 від 23.06.2022 р.

Зміст 3
Вступ 9
Лекція 1. Вступ у Windows Presentation Foundation і XAML 14
Мотивація, яка лежить в основі WPF 14
Уніфікація несхожих API-інтерфейсів 15
Забезпечення розмежування обов'язків через XAML 15
Забезпечення оптимізованої моделі візуалізації 16
Спрощення програмування складних інтерфейсів користувача 17
Дослідження збірок WPF 18
Роль класу Application 19
Роль класу Window 20
Роль класу System.Windows.Controls.ContentControl 21
Роль класу System.Windows.Controls.Control 22
Роль класу System.Windows.FrameworkElement 23
Роль класу System.Windows.UIElement 24
Роль класу System.Windows.Media.Visual 24
Роль класу System.Windows.DependencyObject 25
Роль класу System.Windows.Threading.DispatcherObject 25
Синтаксис XAML для WPF 25
Вступ у Kaxaml 25
Простори імен XML і «ключові слова» XAML 26
Управління видимістю класів і змінних-членів 29
Елементи XAML, атрибути XAML і перетворювачі типів 29
Поняття синтаксису "властивість-елемент" у XAML 31
Поняття приєднуваних властивостей XAML 31
Поняття розширень розмітки XAML 32
Резюме 35
Контрольні питання 35

Лекція 2. Побудова застосунків WPF з використанням Visual Studio 37
Побудова застосунків WPF з використанням Visual Studio 37
Шаблони проектів WPF 37
Панель інструментів і візуальний конструктор/редактор XAML 40
Встановлення властивостей з використанням вікна Properties 40
Обробка подій з використанням вікна Properties 42
Обробка подій в редакторі XAML 42
Вікно Document Outline 43
Включення і відключення відлагодження XAML 44
Дослідження файла App.xaml 44
Відображення XAML-розмітки вікна на код C# 45
Роль BAML 48
Розгадування загадки Main() 49

Взаємодія з даними рівня застосунку 50
Обробка закриття об'єкта Window 51
Перехоплення подій миші 52
Перехоплення подій клавіатури 54
Вивчення документації WPF 55
Резюме 55
Контрольні питання 55

Лекція 3. Елементи управління у WPF 57
Огляд основних елементів управління WPF 57
Елементи управління Ink API 57
Елементи управління документів WPF 58
Загальні діалогові вікна WPF 58
Детальні відомості в документації 59
Короткий огляд візуального конструктора WPF у Visual Studio 60
Робота з елементами управління WPF у Visual Studio 60
Робота з вікном Document Outline 61
Управління компонуванням вмісту з використанням панелей 62
Позиціонування вмісту всередині панелей Canvas 64
Позиціонування вмісту всередині панелей WrapPanel 65
Позиціонування вмісту всередині панелей StackPanel 67
Позиціонування вмісту всередині панелей Grid 68
Позиціонування вмісту всередині панелей DockPanel 71
Включення прокручування в типах панелей 72
Конфігурування панелей з використанням візуальних конструкторів Visual Studio 73
Резюме 77
Контрольні питання 77

Лекція 4. Побудова вікон 78
Побудова вікна з використанням вкладених панелей 78
Побудова системи меню 78
Побудова панелі інструментів 81
Побудова рядка стану 81
Завершення проектування інтерфейсу користувача 82
Реалізація обробників подій MouseEnter/MouseLeave 83
Реалізація логіки перевірки правопису 83
Поняття команд WPF 85
Чотири основні поняття в системі команд WPF 86
Внутрішні об'єкти команд 88
Під'єднання команд до властивості Command 88
Під'єднання команд до довільних дій 89
Робота з командами Open і Save 90

Резюме 93
Контрольні питання 93

Лекція 5. Події у WPF 95
Поняття маршрутизованих подій 95
Роль маршрутизованих бульбашкових подій 96
Продовження або припинення бульбашкового поширення 98
Роль тунельних маршрутизованих подій 98
Глибший погляд на API-інтерфейси та елементи управління WPF 100
Робота з елементом управління TabControl 101
Побудова вкладки Ink API 102
Проектування панелі інструментів 102
Елемент управління RadioButton 104
Додавання кнопок збереження, завантаження і видалення 104
Додавання елемента управління InkCanvas 104
Попередній перегляд вікна 104
Обробка подій для вкладки Ink API 105
Додавання елементів управління в панель інструментів 106
Елемент управління InkCanvas 106
Елемент управління ComboBox 108
Збереження, завантаження та очищення даних InkCanvas 110
Резюме 111
Контрольні питання 111

Лекція 6. Модель прив’язки даних WPF 112
Вступ в модель прив'язки даних WPF 112
Побудова вкладки Data Binding 112
Встановлення прив'язки даних 113
Властивість DataContext 113
Форматування прив'язки даних 115
Перетворення даних з використанням інтерфейсу IValueConverter 115
Встановлення прив'язки даних в коді 117
Роль властивостей залежності 118
Дослідження існуючої властивості залежності 121
Важливі зауваження відносно оболонок властивостей CLR 124
Побудова спеціальної властивості залежності 125
Додавання процедури перевірки достовірності даних 129
Реагування на зміну властивості 129
Резюме 131
Контрольні питання 131

Лекція 7. Служби візуалізації графіки WPF 132

Служби графічної візуалізації WPF 132
Варіанти графічної візуалізації WPF 133
Візуалізація графічних даних з використанням фігур 134
Додавання прямокутників, еліпсів і ліній на поверхню Canvas 136
Видалення прямокутників, еліпсів і ліній з поверхні Canvas 139
Робота з елементами Polyline і Polygon 140
Робота з елементом Path 141
Кисті і пера WPF 145
Конфігурування кистей з використанням Visual Studio 145
Конфігурування кистей в коді 148
Конфігурування пер 149
Застосування графічних трансформацій 149
Перший погляд на трансформації 150
Трансформація даних Canvas 152
Робота з редактором трансформацій Visual Studio 154
Побудова початкового компонування 154
Застосування трансформацій на етапі проектування 155
Трансформація клієнтської частини вікна в коді 156
Резюме 157
Контрольні питання 157

Лекція 8. Візуалізація графічних даних 159
Візуалізація графічних даних з використанням малюнків і геометричних об'єктів 159
Побудова кисті DrawingBrush з використанням геометричних об'єктів 160
Малювання за допомогою DrawingBrush 161
Включення типів Drawing в DrawingImage 162
Робота з векторними зображеннями 162
Перетворення файла з векторною графікою у файл XAML 162
Імпортування графічних даних в проект WPF 163
Взаємодія із зображенням 164
Візуалізація графічних даних з використанням візуального рівня 165
Базовий клас Visual і похідні дочірні класи 165
Перший погляд на клас DrawingVisual 165
Візуалізація графічних даних у спеціальному диспетчері компонування 168
Перевірка попадання у Visual 171
Реагування на операції перевірки попадання 171
Резюме 173
Контрольні питання 173

Лекція 9. Ресурси WPF 174

Система ресурсів WPF 174
Робота з двійковими ресурсами 174
Програмне завантаження зображення 177
Робота з об'єктними (логічними) ресурсами 182
Роль властивості Resources 182
Визначення ресурсів рівня вікна 183
Розширення розмітки {StaticResource} 185
Розширення розмітки {DynamicResource} 186
Ресурси рівня застосунку 186
Визначення об'єднаних словників ресурсів 188
Визначення збірки, яка включає тільки ресурси 189
Резюме 192
Контрольні питання 193

Лекція 10. Реалізація анімації WPF 194
Служби анімації WPF 194
Роль класів анімації 194
Властивості To, From і By 196
Роль базового класу Timeline 196
Реалізація анімації в коді C# 196
Управління темпом анімації 198
Запуск у зворотному порядку і циклічне виконання анімації 199
Реалізація анімації в розмітці XAML 199
Роль розкадровувань 201
Роль тригерів подій 201
Анімація з використанням дискретних ключових кадрів 202
Резюме 203
Контрольні питання 203

Лекція 11. Реалізація стилів WPF 204
Роль стилів WPF 204
Визначення і використання стилю 204
Перевизначення налаштувань стилю 205
Вплив атрибуту TargetType на стилі 206
Створення підкласів існуючих стилів 207
Визначення стилів з тригерами 208
Визначення стилів з множиною тригерів 209
Анімовані стилі 210
Застосування стилів в коді 210
Резюме 212
Контрольні питання 212

Лекція 12. Логічні і візуальні дерева та шаблони 213
Логічні дерева, візуальні дерева і стандартні шаблони 213

Програмна інспекція логічного дерева 213
Програмна інспекція візуального дерева 215
Програмна інспекція стандартного шаблону елемента управління 216
Побудова шаблону елемента управління за допомогою інфраструктури тригерів 219
Шаблони як ресурси 220
Вбудовування візуальних підказок з використанням тригерів 221
Роль розширення розмітки {TemplateBinding} 222
Роль класу ContentPresenter 223
Вбудовування шаблонів в стилі 224
Резюме 225
Контрольні питання 225

Лекція 13. Побудова бізнес-застосунків за допомогою WPF 227
Прив'язка даних 227
Застосунок BooksDemo 228
Прив'язка за допомогою XAML 230
Прив'язка простого об'єкта 234
Повідомлення про зміни 236
Об'єктний постачальник даних 239
Прив'язка списка 241
Прив'язка типу "головна-деталі" 244
Множинна прив'язка 245
Пріоритетна прив'язка 247
Перетворення значення 249
Динамічне додавання елементів списка 251
Селектор шаблону даних 252
Прив'язка до XML 254
Перевірка достовірності прив'язки 257
Обробка виключень 257
Інформація про помилку в даних 262
Спеціальні правила перевірки достовірності 264
Зв'язування з командами 265
Визначення команд 266
Визначення джерел команд 267
Прив'язки команд 268
Резюме 269

Рекомендована література (основна, допоміжна) 270

ВСТУП

Windows Presentation Foundation (**WPF**) – найсучасніша з пропонованих корпорацією Microsoft технологій створення графічних інтерфейсів користувача в ОС Windows, на зразок, простих форм, документо-орієнтованих вікон, анімованих зображень, відео, 3D-середовища з ефектом занурення або все вищеперераховане. Технологія WPF дозволяє розробляти найрізноманітніші застосунки простіше, ніж будь-коли раніше. Крім того, вона лежить в основі технології Silverlight, яка поширює WPF на Мережу і мобільні пристрої, наприклад телефони на базі ОС Windows.

З моменту анонсування WPF у 2003 році (під кодовою назвою **Avalon**) ця технологія прикувала до себе пильну увагу завдяки революційній зміні процесу розробки ПЗ – особливо з боку програмістів Windows, які звиклися з Windows Forms і GDI. WPF порівняно легко дозволяє створювати цікаві і корисні застосунки, які демонструють різноманітні можливості, які важко реалізувати за допомогою інших технологій. У версії WPF 4, випущеній у квітні 2010 року, істотно поліпшені практично всі аспекти цієї технології.

WPF знаменує собою відхід від попередніх технологій в плані моделі програмування, засадничих ідей і базової термінології. Навіть перегляд початкового коду WPF-застосунку (наприклад, шляхом декомпіляції його компонентів за допомогою програми .NET Reflector[1] або їй подібній) може стати джерелом сюрпризів, тому що код, який цікавить вас, часто знаходиться не там, де ви його очікуєте. А якщо додати сюди ще і той факт, що будь-яке завдання у WPF можна вирішити декількома способами, то легко прийти до висновку, який поділяють багато розробників: вивчення WPF – не проста справа.

Тому для спрощення початку процесу опанування WPF-технології має прийти цей навчальний посібник. Розробляючи цей посібник ми прагнули досягнути наступних цілей:

- Ознайомити студентів з базовими концепціями в доступній формі, не відходячи від практичного використання цієї технології.
- Дати відповідь на питання, які виникають у більшості студентів, що вивчають технологію, і показати, як розв’язуються типові завдання (у цьому нам, також, допоможе цикл завдань лабораторного практикуму, розроблений нами, під назвою «Створення інтерфейсу користувача»).
- Чітко окреслити межі застосовності технології, без претензій на те, що вона є рішенням усіх проблем.
- Сподіватися на те, що цей посібник стане початковим довідковим керівництвом, до якого можна повертатися знову і знову.

Сподіваємося, що цей посібник відповідає всім заявленим критеріям.

---

[1] **.NET Reflector** – платна утиліта для Microsoft .NET, яка комбінує браузер класів, статичний аналізатор і декомпілятор, спочатку написана Lutz Roeder. 20 серпня 2008 Red Gate Software оголосили, що вони беруть відповідальність за подальшу розробку програми. MSDN Magazine назвав її однією з десяти "Must-Have" утиліт для розробників, Scott Hanselman включив її у свій "Big Ten Life and Work – Changing Utilities" (приблизний переклад: велика десятка утиліт, які змінюють життя і роботу). Дивись – https://www.red-gate.com/products/dotnet-development/reflector/

Цей посібник адресований студентам спеціальностей «Інформаційні технології». Неважливо, що саме розроблятимуть студенти: програми для бізнесу чи для масового споживача, повторно використовувані елементи управління – тут ви знайдете початкові відомості, які дозволяють отримати максимум користі з платформи WPF. Посібник може стати відправною точкою для опанування технології створення інтерфейсу користувача вчителями інформатики, аматорами програмістами, які вирішили в майбутньому стати розробниками програмного забезпечення.

Підведемо підсумки. У цьому посібнику:

- Міститься достатньо відомостей, які потрібні для розуміння технології декларативного створення інтерфейсів користувача, що допускають застосування стилів – технології, яка використовує мову **eXtensible Application Markup Language** (**XAML**), яка заснована на **XML**.
- Достатньо детально розглядаються різні функціональні можливості WPF: елементи управління, компонування, ресурси, прив'язка до даних, стилі, графіка, анімація і багато чого іншого.
- Демонструється створення популярних елементів інтерфейсу користувача, наприклад галерей, екранних підказок, нестандартних способів компонування елементів.
- Демонструється створення складніших механізмів організації інтерфейсів користувача: спливаючих панелей і пристикованих панелей, як у Visual Studio.
- Описується, як створювати повноцінні елементи управління WPF.
- Демонструється створення гібридних застосувань, в яких WPF поєднується з Windows Forms, DirectX, ActiveX та іншими технологіями.

Посібник не претендує на опис абсолютно всіх можливості WPF – це і неможливо в межах відведеного часу. Але ми намагалися надати вам технологію створення інтерфейсу користувача, для практичного використання при розробці застосунків у бакалаврських та магістерських дослідженнях.

Приклади, приведені в посібнику , написані на XAML і C#. Повсюдне використання мови XAML пояснюється рядом причин: частенько це найшвидший спосіб записати початковий код; фрагменти, написані на XAML, можна копіювати в інструментальні засоби і бачити результат, не прибігаючи до компіляції; засновані на WPF інструменти генерують код на XAML, а не на процедурних мовах; нарешті, XAML не залежить від того, на якій .NET-сумісній мові ви пишете: Visual Basic, C# або ще якійсь. У тих випадках, коли відповідність між XAML і C# неочевидна, наводяться еквівалентні представлення коду на обох мовах.

Вимоги до програмного забезпечення

Має бути встановлене наступне програмне забезпечення:

- Версія ОС Windows, не нижче 10.0.
- Бажано використовувати Microsoft Visual Studio 2019/2022 (краще 2022).

При інсталяції Visual Studio для початку встановіть конфігурацію за замовчуванням. У міру набуття досвіду нескладно встановити потрібні додаткові деталі, яких вам не вистачає.

Майже всі наведені приклади створювалися у версії 2019/2022.

Організація матеріалу

Посібник складається з двох модулів, в яких послідовно викладається матеріал, потрібний для ефективного використання WPF. При бажанні можна відразу перейти до конкретної теми, яка вас цікавить, наприклад графіки або анімації. Нижче коротко описаний зміст кожного модуля.

Модуль 1. Базові елементи та механізми Windows Presentation Foundation і XAML

Лекція 1. Вступ у Windows Presentation Foundation і XAML: Мотивація, яка лежить в основі WPF. Дослідження збірок WPF. Синтаксис XAML для WPF.

Лекція 2. Побудова застосунків WPF з використанням Visual Studio: Побудова застосунків WPF з використанням Visual Studio. Вивчення документації WPF.

Лекція 3. Елементи управління у WPF: Огляд основних елементів управління WPF. Короткий огляд візуального конструктора WPF у Visual Studio. Робота з вікном Document Outline. Управління компонуванням вмісту з використанням панелей.

Лекція 4. Побудова вікон: Побудова вікна з використанням вкладених панелей.

Лекція 5. Події у WPF: Поняття команд WPF. Поняття маршрутизованих подій. Глибший погляд на API-інтерфейси та елементи управління WPF. Побудова вкладки Ink API.

Лекція 6. Модель прив’язки даних WPF: Вступ в модель прив'язки даних WPF. Роль властивостей залежності. Побудова спеціальної властивості залежності.

У цьому модулі ми з’ясуємо, що інфраструктура Windows Presentation Foundation (WPF) є набором інструментів для побудови інтерфейсів користувача, який з'явився у версії .NET 3.0. Основна мета WPF полягає в інтеграції та уніфікації багатьох раніше розрізнених настільних технологій (двовимірна і тривимірна графіка, розробка вікон та елементів управління і тому подібне) в єдину уніфіковану програмну модель. Окрім цього в застосунках WPF зазвичай застосовується мова XAML, яка дозволяє визначати зовнішній вигляд і поведінку елементів WPF через розмітку.

Ми довідаємося, що мова XAML дозволяє описувати дерева об'єктів .NET з використанням декларативного синтаксису. Під час дослідження XAML у цьому модулі ви дізнаєтеся про декілька нових фрагментів синтаксису, включаючи синтаксис "*властивість-елемент*" і приєднувані властивості, а також про роль перетворювачів типів і розширень розмітки XAML.

Ми з’ясуємо, що розмітка XAML ключовий аспект будь-якого застосунку WPF виробничого рівня. У фінальному прикладі буде побудовано застосунок WPF, який продемонструє багато концепцій, обговорених в модулі.

Також ми розглянемо деякі аспекти елементів управління WPF, починаючи з огляду набору інструментів для елементів управління і ролі диспетчерів компонування (панелей). Ми побудуємо простий застосунок –

текстовий процесор. У ньому продемонструємо використання інтегрованої у WPF функціональності перевірки правопису, а також створення головного вікна з системою меню, рядком стану і панеллю інструментів.

Важливіше те, що ми навчимося будувати команди WPF. Ці незалежні від елементів управління події можна приєднувати до елемента інтерфейсу користувача або інших механізмів для автоматичного спадкування готової функціональності (наприклад, операцій з буфером обміну).

Крім того, ми довідаємося про побудову інтерфейсів користувача у XAML і попутно ознайомитесь з інтерфейсом Ink API, запропонованим WPF. Ви також отримаєте уявлення про операції прив'язки даних WPF.

Нарешті, ми з'ясуємо, що інфраструктура WPF додає унікальний аспект до традиційних програмних примітивів .NET, зокрема до властивостей і подій. Буде показано, механізм властивостей залежності, який дозволяє будувати властивість, яка може інтегруватися з набором служб WPF (анімації, прив'язки даних, стилі і т. д.). Також з'ясуємо, що механізм маршрутизованих подій, надає події спосіб поширюватися подіям вгору або вниз деревом розмітки.

Модуль 2. Розширені можливості Windows Presentation Foundation

Лекція 7. Служби візуалізації графіки WPF: Служби графічної візуалізації WPF. Візуалізація графічних даних з використанням фігур. Кисті і пера WPF. Застосування графічних трансформацій. Робота з редактором трансформацій Visual Studio.

Лекція 8. Візуалізація графічних даних: Візуалізація графічних даних з використанням малюнків і геометричних об'єктів. Робота з векторними зображеннями. Візуалізація графічних даних з використанням візуального рівня.

Лекція 9. Ресурси WPF: Система ресурсів WPF. Робота з об'єктними (логічними) ресурсами.

Лекція 10. Реалізація анімації WPF: Служби анімації WPF. Реалізація анімації в розмітці XAML.

Лекція 11. Реалізація стилів WPF: Роль стилів WPF.

Лекція 12. Логічні і візуальні дерева та шаблони: Логічні дерева, візуальні дерева і стандартні шаблони. Побудова шаблону елемента управління за допомогою інфраструктури тригерів.

Ми довідаємося, про те що Windows Presentation Foundation настільки насичена графікою інфраструктура для побудови графічних інтерфейсів користувача, що це не викликатиме здивувань наявністю декількох способів візуалізації графічного виводу. Розглянемо три підходи до візуалізації (фігури, малюнки і візуальні об'єкти), а також різноманітні примітиви візуалізації, такі як кисті, пера і трансформації.

З'ясуємо, що коли потрібно будувати інтерактивну двовимірну візуалізацію, то фігури роблять такий процес дуже простим. З іншого боку, статичні, не інтерактивні зображення можуть візуалізуватися в оптимальнішій манері з використанням малюнків і геометричних об'єктів, а візуальний рівень (доступний тільки в коді) забезпечить максимальний контроль і продуктивність.

Тут ми розглянемо систему управління ресурсами WPF. Почнемо з дослідження роботи з двійковими ресурсами і ролі об'єктних ресурсів. Ви довідаєтеся, що об'єктні ресурси є іменованими фрагментами розмітки XAML, які можуть бути збережені в різноманітних місцях з метою багатократного використання вмісту.

Потім опишемо API-інтерфейс анімації WPF. У наведених прикладах анімація буде створюватися за допомогою коду C#, а також за допомогою розмітки XAML. Для управління виконанням анімації, визначеної в розмітці, застосуємо елементи Storyboard і тригери. Далі буде продемонстрований механізм стилів WPF, який інтенсивно використовує графіку, об'єктні ресурси та анімацію.

Після цього з'ясовуємо відношення між логічним і візуальним деревами. У своїй основі логічне дерево є однозначною відповідністю розмітці, яка створена для опису кореневого елемента WPF. Позаду логічного дерева знаходиться набагато глибше візуальне дерево, яке містить детальні інструкції візуалізації.

Крім того, вивчимо роль стандартного шаблону, не забуваючи, що при побудові спеціальних шаблонів ви по суті замінюєте усе візуальне дерево елемента управління (чи частина дерева) власною реалізацією.

Лекція 1. Вступ у Windows Presentation Foundation і XAML

З випуском платформи .NET версії 1.0, програмісти у яких була потреба в побудові графічних настільних застосунків використовували два API-інтерфейси – Windows Forms і GDI+, запаковані переважно в збірки **System.Windows.Forms.dll** і **System.Drawing.dll**. API-інтерфейси Forms і GDI+ – досі дієздатні, та для побудови настільних графічних інтерфейсів користувача, починаючи з версії .NET 3.0, розроблено альтернативний API-інтерфейс, названий **Windows Presentation Foundation** (**WPF**).

Ознайомимося з основними мотивами, які лежать в основі нової інфраструктури, для побудови графічних інтерфейсів користувача, та познайомимося з відмінностями між моделями програмування Windows Forms/GDI+ і WPF.

Також, у цій лекції розглянемо граматику на основі XML, так звану *розширювану мову розмітки застосунків* (**Extensible Application Markup Language** – XAML). Ознайомимося із синтаксисом і семантикою XAML.

Мотивація, яка лежить в основі WPF

Тривалий час у Microsoft створювали інструменти для побудови графічних інтерфейсів користувача (для розробки на C/C++/Windows API, VB6, MFC і т. д.) настільних застосунків. Кожен інструмент надає свою кодову базу для створення основних компонентів застосунку з графічним інтерфейсом користувача, включаючи:

- головні вікна;
- діалогові вікна;
- елементи управління;
- системи меню;
- та інші базові компоненти.

Програмна модель Windows Forms дещо асиметрична, хоча з її використанням було створено багато повноцінних настільних застосунків. Простіше кажучи – збірки **System.Windows.Forms.dll** і **System.Drawing.dll** прямо не підтримують деяких технологій, які потрібні при побудові повнофункціонального настільного застосунку. Розглянемо спеціалізовану розробку графічних інтерфейсів користувача до появи WPF (табл. 1.1), яка демонструє поняття асиметричності програмної моделі.

Таблиця 1.1. Технології, використовувані для забезпечення бажаної функціональності, попередники WPF

Бажана функціональність	Технологія
Підтримка вікон з елементами управління	Windows Forms
Підтримка двовимірної графіки	GDI(System.Drawing.dll)
Підтримка тривимірної графіки	API-інтерфейси DirectX
Підтримка потокового відео	API-інтерфейси Windows Media Player
Підтримка документів нефіксованого формату	Програмне маніпулювання файлами PDF

Видно, що при використанні Windows Forms, розробник запозичує типи з декількох незв'язаних API-інтерфейсів та об'єктних моделей. Треба відмітити, що використання різних API-інтерфейсів синтаксично схоже (код C#), але кожна технологія вимагає радикально інших підходів. Створення тривимірної анімації з використанням DirectX потребує навичок, які сильно відрізняються від потрібних для прив'язування даних до екранної сітки. Зрозуміло, що використовуючи Windows Forms, розробнику не просто досконало опанувати природу кожного API-інтерфейсу.

Уніфікація несхожих API-інтерфейсів

Мета створення WPF – розробити таку інфраструктуру WPF яка об'єднала б раніше незв'язні завдання програмування в одну уніфіковану об'єктну модель. Це означає, що розробляючи тривимірну анімацію більше не знадобиться ручне кодування із використанням API-інтерфейсу **DirectX** (хоча можна і вручну кодувати), бо потрібна функціональність вже вбудована у WPF. Продемонструємо, наскільки все стало зрозумілішим, з допомогою табл. 1.2 в якій представлена модель розробки настільних застосунків, введена в .NET 3.0.

Таблиця 1.2. Технології, використовувані в .NET 3.0 для забезпечення бажаної функціональності

Бажана функціональність	Технологія
Підтримка вікон з елементами управління	WPF
Підтримка двовимірної графіки	WPF
Підтримка тривимірної графіки	WPF
Підтримка потокового відео	WPF
Підтримка документів нефіксованого формату	WPF

Очевидно, що тепер розробники .NET отримали єдиний уніфікований *симетричний* API-інтерфейс для всіх широко використовуваних потреб, які виникають при розробці графічних інтерфейсів користувача настільних застосунків. Дослідивши і вивчивши функціональність основних збірок WPF та опанувавши граматику XAML, ви зрозумієте, як не складно і швидко з їх допомогою можна створювати складні інтерфейси користувача.

Забезпечення розмежування обов'язків через XAML

Мабуть, одна з найзначніших переваг технології WPF полягає в тому, що її інфраструктура надає спосіб *відокремлення/розмежування* візуалізації і поведінки застосунку з графічним інтерфейсом користувача від управляючої програмної логіки. Використовуючи мову XAML, інтерфейс користувача застосунку можна визначати, застосовуючи *розмітку* XML. Таку розмітку (вона може бути згенерована за допомогою середовища на зразок Microsoft Visual Studio або Microsoft Expression Blend) потім можна об’єднати з *пов'язаним* файлом коду, який забезпечить функціональність програми.

Інтерфейс користувача, створений з використанням мови XAML – це не тільки:

- прості елементи:
  - кнопки;
  - таблиці;
  - вікна зі списками;
  - і т. д.

але також:

- інтерактивна двовимірна графіка;
- тривимірна графіка;
- анімація;
- логіка прив'язки даних;
- функціональність мультимедіа (на зразок відтворення відео).

Також, XAML спрощує налаштування візуалізації елемента управління. WPF надає засоби для модифікації елементів управління за допомогою стилів і шаблонів, докладаючи мінімальні зусилля. Єдиною вагомою причиною для побудови елемента управління WPF з нуля є потреба в зміні поведінки елемента управління. Це може бути щось на зразок:

- додавання спеціальних методів;
- властивостей;
- подій;
- створення підкласу існуючого елемента управління з метою перевизначення віртуальних членів.

Просту зміну зовнішнього вигляду елемента управління можна здійснити повністю, використовуючи лише розмітку.

Забезпечення оптимізованої моделі візуалізації

Набори інструментів, такі як Windows Forms, MFC або VB6 використовуються для побудови графічних інтерфейсів користувача. Вони виконують всі запити щодо графічної візуалізації (візуалізуючи елементи управління на зразок кнопок і вікон зі списком) із використанням низькорівневого API-інтерфейсу на основі C (GDI). Цей API-інтерфейс тривалий час був частиною Windows. Він забезпечує адекватну продуктивність для типових бізнес-застосунків або простих графічних програм, але якщо застосунку потрібний був інтерфейс користувача з високопродуктивною графікою, то він цього на забезпечував. У такому випадку доводилося використовувати DirectX.

Програмна модель WPF при візуалізації графічних (двовимірна і тривимірна графіка, анімація, візуалізація елементів управління і т. д.) даних не використовує GDI, а працює з API-інтерфейсом DirectX. Очевидна вигода такого підходу в тому, що застосунки WPF автоматично отримуватимуть переваги апаратної і програмної оптимізації. З використанням API-інтерфейсу DirectX застосунки WPF використовують розвинені графічні служби (ефекти розмиття, згладжування, прозорості і т. п.) без складнощів, притаманних програмуванню безпосередньо.

Спрощення програмування складних інтерфейсів користувача

Підсумок сказаного досі: Windows Presentation Foundation (WPF) – це API-інтерфейс, призначений для побудови застосунків. Він інтегрує різноманітні настільні API-інтерфейси в єдину об'єктну модель. Також цей API-інтерфейс забезпечує чітке розмежування обов'язків завдяки використанню XAML. Крім того застосунки WPF також виграють від простого способу інтеграції з досить складними службами. Перерахуємо основні функціональні можливості WPF:

- *Диспетчери компонування*, які забезпечують досить гнучкий контроль над розміщенням і зміною позицій вмісту.
- *Розширений механізм прив'язки* даних для зв'язування вмісту з елементами інтерфейсу користувача різноманітними способами.
- *Вбудований механізм стилів*, який дозволяє визначати «теми» для застосунків WPF.
- *Автоматична зміна розмірів вмісту* аби цей розмір відповідав розмірам і роздільній здатності екрану, який відображає інтерфейс користувача застосунку. Все це реалізується завдяки підтримки векторної графіки.
- *Підтримка двовимірної* і тривимірної графіки, анімації, а також відтворення відео та аудіо.
- *Розвинений типографський API-інтерфейс*, який підтримує документи **XML Paper Specification** (XPS[2]), фіксовані[3] документи (WYSIWYG[4]), документи нефіксованого[5] формату та анотації в документах (наприклад, API-інтерфейс Sticky Notes). Детальніше за адресою https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/advanced/documents-in-wpf?view=netframeworkdesktop-4.8.
- *Підтримка взаємодії з успадкованими моделями графічних інтерфейсів* (такими як Windows Forms, ActiveX і HWND-дескриптори Win32). Наприклад, в застосунок WPF можна вбудовувати спеціальні елементи управління Windows Forms і навпаки.

---

[2] XPS (XML Paper Specification) – відкритий графічний формат фіксованої розмітки на базі XML від компанії Microsoft. Функціональність спрямована виключно на документообіг – документ простіший і легший PDF, використовується векторна непослідовна розмітка, аналогічна XAML, взаємодіє з .NET Framework, підтримує багатопотокову роботу і представлення, безпечний, підтримує шифрування, цифрові сертифікати.

[3] Фіксовані документи призначені для застосувань, для яких потрібно точне уявлення про те, що ви отримуєте (WYSIWYG), незалежно від використовуваного дисплея або принтера. Використовуються в комп'ютерній верстці, текстовій обробці і макетах форм, де строга відповідність дизайну початкової сторінки має критичне значення.

[4] WYSIWYG (***W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et*, «що бачиш, те і отримаєш») – властивість прикладних програм або веб-інтерфейсів, в яких зміст відображається в процесі редагування і виглядає максимально близько схожим на кінцеву продукцію, яка може бути друкарським документом, веб-сторінкою або презентацією..

[5] Документи нефіксованого формату призначені для підвищення зручності перегляду і читання; їх краще всього використовувати, коли в роботі з документом найважливіше зручність читання. Веб-сторінка – це простий приклад документа нефіксованого формату, вміст на якій динамічно форматується, щоб уміщатися у вікні.

Тепер, маючи уявлення про те, що інфраструктура WPF додає до платформи .NET, розглянемо різноманітні типи застосунків, які можуть бути створені завдяки використанню цього API-інтерфейсу. Багато з перерахованих вище можливостей будуть детальніше досліджені в наступних лекціях.

Дослідження збірок WPF

Зрештою інфраструктура WPF – це не що інше ніж колекція типів, вбудованих у збірки .NET. У табл. 1.3 описані основні збірки, використовувані при розробці застосунків WPF, на кожну з яких має бути додане посилання, коли створюється новий проект. Передбачувано, проекти WPF у Visual Studio посилаються на ці обов'язкові збірки автоматично.

Таблиця 1.3. Основні збірки WPF

Збірка	Опис
PresentationCore.dll	У цій збірці визначені багаточисельні простори імен, які утворюють фундамент рівня графічного інтерфейсу користувача у WPF. Наприклад, вона включає підтримку **API**-інтерфейсу **WPF Ink**, примітиви анімації і багато типів графічної візуалізації.
PresentationFramework.dl	У цій збірці містяться більшість елементів управління WPF, класи **Application** і **Window**, підтримка інтерактивної двовимірної графіки і численні типи, використовувані для прив'язки даних.
System.Xaml.dll	Ця збірка надає простори імен, які дозволяють програмно взаємодіяти з документами XAML під час виконання. Загалом і в цілому вона корисна тільки при розробці інструментів підтримки WPF або коли потрібен абсолютний контроль над розміткою XAML під час виконання.
WindowsBase.dll	У цій збірці визначені типи, які формують інфраструктуру **API**-інтерфейсу WPF, включаючи типи потоків WPF, типи, пов'язані з безпекою, різноманітні перетворювачі типів і підтримку властивостей залежності і маршрутизованих подій.

Описані в табл. 1.3 збірки визначають декілька нових просторів імен і сотні нових класів, інтерфейсів, структур, перерахувань і делегатів .NET. Детальні відомості можна знайти в документації за адресою https://docs.microsoft.com/ru-ru/dotnet/api/?view=netframework-4.7.2, а в табл. 1.4 описана роль деяких важливих просторів імен.

Таблиця 1.4. Основні простори імен WPF

Простір імен	Опис
System.Windows	Кореневий простір імен WPF. Тут розміщені основні класи (такі як **Application** і **Window**), які потрібні у будь-якому проекті настільного застосунку WPF.
System.Windows.Controls	Містить всі очікувані графічні елементи (віджети) WPF, включаючи типи для побудови систем меню, спливаючі підказки і багаточисельні диспетчери компонування.
System.Windows.Data	Містить типи для роботи з механізмом прив'язки даних WPF, а також для підтримки шаблонів прив'язки даних.
System.Windows.Documents	Містить типи для роботи з **API**-інтерфейсом документів, який дозволяє інтегрувати в застосунки WPF функціональність в стилі PDF через протокол XML Paper Specification (XPS).
System.Windows.Ink	Надає підтримку **Ink API**-інтерфейсу, який дозволяє отримувати ввід від

	пера або миші, реагувати на вхідні жести і т. д. Цей **API**-інтерфейс корисний при програмуванні для Tablet PC, але може використовуватися також у будь-яких додатках WPF.
System.Windows.Markup	Тут визначена множина типів, які забезпечують програмний аналіз та обробку розмітки XAML (та еквівалентного двійкового формату BAML).
System.Windows.Media	Кореневий простір імен для декількох пов'язаних з мультимедіа просторів імен, усередині яких визначені типи для роботи з анімацією, візуалізацією тривимірної графіки, візуалізацією тексту та іншими мультимедійними примітивами.
System.Windows.Navigation	Надає типи для забезпечення логіки навігації, використовуваної браузерними застосунками XAML (XAML browser application – XBAP), а також настільними застосунками, які вимагають сторінкової моделі навігації.
System.Windows.Shapes	Тут визначені класи, які дозволяють візуалізувати двовимірну графіку, яка автоматично реагує на ввід за допомогою миші.

Вивчення програмної моделі WPF розпочнемо з дослідження двох членів простору імен **System.Windows**, які є загальними при традиційній розробці будь-якого настільного застосунку: **Application** і **Window**.

Роль класу Application

Клас **System.Windows.Application** є *глобальним виконуваним екземпляром* застосунку WPF. У ньому є метод **Run** (для запуску застосунку), комплект подій, які можна обробляти для взаємодії із застосунком упродовж часу його життя (на зразок **Startup** та **Exit**), і набір членів, специфічних для браузерних застосунків XAML (таких як ініційовані події, під час навігації користувача сторінками). У табл. 1.5 описані основні властивості класу **Application**.

Таблиця 1.5. Основні властивості класу **Application**

Властивість	Опис
Current	Ця статична властивість дозволяє отримувати доступ до працюючого об'єкта **Application** з будь-якого місця коду. Може бути корисною, коли звичайному або діалоговому вікну потрібний доступ до об'єкта **Application**, який створив його, зазвичай, для взаємодії зі змінними або функціональністю рівня застосунку.
MainWindow	Ця властивість дозволяє програмно отримувати або встановлювати головне вікно застосування.
Properties	Ця властивість дозволяє встановлювати та отримувати дані, доступні через всі елементи застосунку WPF (вікна, діалогові вікна і т. д.).
StartupUr	Ця властивість отримує або встановлює URI, який вказує вікно або сторінку для автоматичного відкриття при запуску застосунку.
Windows	Ця властивість повертає об'єкт типу **WindowCollection**, який надає доступ до всіх вікон, що створені в потоці, створених об'єктом **Application**. Може бути корисною, коли треба пройтися по всіх відкритих вікнах застосунку і змінити їх стан (скажімо, згорнути всі вікна).

Побудова класу Application

У будь-якому застосунку WPF знадобиться визначити клас, який розширює **Application**. Усередині такого класу буде визначена точка входу програми (метод **Main()**), яка створює екземпляр цього підкласу і зазвичай обробляє події **Startup** та **Exit** (при потребі). Ось приклад:

```
// Визначте глобальний об'єкт застосунку для цієї програми WPF.
  class MyApp : Application
  {
    [STAThread]
    static void Main(string [ ] args)
    {
      // Створити об'єкт застосунку
      MyApp app = new MyApp() ;
      // Зареєструвати події Startup/Exit.
      app.Startup += (s, e) => { /* Запуск застосунку */ };
      app.Exit += (s, e) => { /* Завершення застосунку */ };
    }
  }
```

Обробник події **Startup** найчастіше оброблятиме вхідні аргументи командного рядка і запускатиме головне вікно програми. Обробник події **Exit** є місцем, куди можна помістити будь-яку потрібну логіку завершення програми (наприклад, збереження уподобань користувача).

> **Увага!** Метод **Main()** застосунку WPF повинен супроводжуватися атрибутом [**STAThread**], який гарантує, що будь-які успадковані об'єкти COM, використовувані застосунком, безпечні щодо потоків. Якщо не задавати цей атрибут, то під час виконання методу **Main()** виникне виключення.

Перерахування елементів колекції Windows

Цікавою властивістю класу **Application** є **Windows**. Вона забезпечує доступ до колекції, яка представляє всі вікна, завантажені в пам'ять для поточного застосунку WPF. Створювані нові об'єкти **Window** автоматично додаються в колекцію **Application.Windows**. Нижче наведений приклад методу, який згортає всі вікна застосунку (можливо у відповідь на натиснення певної комбінації клавіш або вибір пункта меню кінцевим користувачем):

```
static void MinimizeAllWindows()
{
  foreach(Window wnd in Application.Current.Windows)
  {
    wnd.WindowState = WindowState.Minimized;
  }
}
```

Роль класу Window

Клас **System.Windows.Window** (збірка **PresentationFramework.dll**) представляє одиночне вікно, яким володіє похідний від **Application** клас, включаючи всі відображувані головним вікном діалогові вікна. Тип **Window** має декілька батьківських класів з додатковою функціональністю. Рис. 1.1 демонструє ланцюжок спадкоємства, разом з інтерфейсами для класу **System.Windows.Window**, як він виглядає у браузері об'єктів Visual Studio.

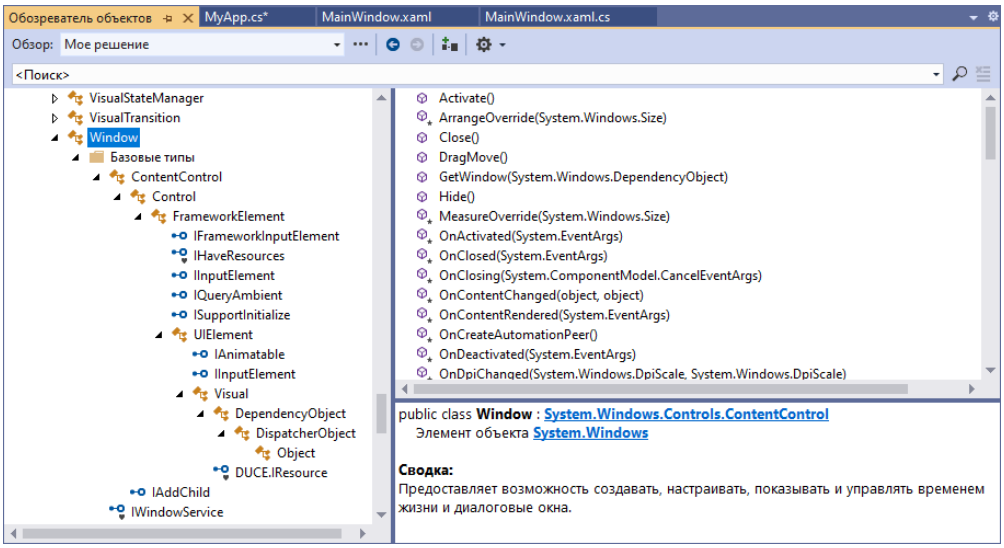


Рис. 1.1. Ієрархія спадкоємства класу **Window**

У міру опрацювання цієї і наступних лекцій прийде розуміння функціональності, пропонованої багатьма базовими класами WPF. Далі представлений короткий огляд функціональності кожного базового класу (повні відомості шукайте в документації за адресою https://docs.microsoft.com/ru-ru/dotnet/api/?view=netframework-4.7.2).

Роль класу System.Windows.Controls.ContentControl

Безпосереднім батьківським класом **Window** є – один із найважливіших – клас **ContentControl**. Базовий клас **ContentControl** забезпечує похідні типи здатністю розміщувати в собі одиночний фрагмент вмісту. Цей фрагмент відноситься до візуальних даних, поміщених всередину області елемента управління завдяки властивості **Content**. Модель вмісту WPF дозволяє легко налаштовувати базовий вигляд і поведінку елемента управління **ContentControl**.

Наприклад, коли мова йде про звичайний «кнопковий» елемент управління, то зазвичай передбачається, що його вмістом буде простий рядковий літерал (**ОК**, **Cancel**, **Abort** і т. д.). Якщо для опису елемента управління WPF застосовується XAML, а значення, присвоюване властивості **Content**, це простий рядок, тоді встановити властивість **Content** всередині відкриваючого визначення елемента можна так:

```
<!-- Встановлення значення Content у відкриваючому елементі -->
  <Button Height="80" Width="100" Content="OK"/>
```

На замітку! Властивість **Content** можна також встановлювати в коді C#, що дозволяє змінювати вміст елемента управління під час виконання.

Проте вміст може бути практично будь-яким. Наприклад, нехай потрібна «кнопка», яка містить в собі щось цікавіше, ніж простий рядок – можливо спеціальну графіку або текстовий фрагмент.

Коли властивості **Content** має бути присвоєне значення, яке неможливе виразити у вигляді простого масиву символів, то його не можна присвоїти з використанням атрибуту у відкриваючому визначенні елемента управління. Натомість знадобиться визначити дані вмісту неявно всередині зони дії елемента. Наприклад, наступний елемент <**Button**> включає як вміст елемент <**StackPanel**>, який сам має унікальні дані (а саме – <**Ellipse**> і <**Label**>):

```
<!-- Неявне встановлення для властивості Content складних даних -->
<Button Height="80" Width="100">
  <StackPanel>
    <Ellipse Fill="Red" Width="25" Height="25"/>
    <Label Content ="OK!"/>
  </StackPanel>
</Button>
```

Для встановлення складного вмісту можна також застосовувати *синтаксис* "*властивість-елемент*" мови XAML. Подивіться на, показане далі, функціонально еквівалентне попередньому визначення <**Button**>, яке *явно* встановлює властивість **Content** за допомогою синтаксису "властивість-елемент" (додаткова інформація про XAML буде викладена пізніше):

```
<!-- Встановлення властивості Content з використанням синтаксису "властивість-елемент"-->
  <Button Height="80" Width="100">
    <Button.Content>
      <StackPanel>
        <Ellipse Fill="Red" Width="25" Height="25"/>
        <Label Content ="OK!"/>
      </StackPanel>
    </Button.Content>
  </Button>
```

Зазначимо, що не всі елементи WPF похідні від класу **ConentConrtol**. З цієї причини не всі елементи підтримують таку унікальну модель вмісту (хоча більшість підтримує). Крім того, деякі елементи управління WPF вносять декілька удосконалень у щойно розглянуту базову модель вмісту. В ***Лекція 3. Елементи управління у WPF*** роль вмісту WPF розкривається детальніше.

Роль класу System.Windows.Controls.Control

На відміну від **ContentControl** всі елементи управління WPF розділяють як спільний батьківський клас базовий клас **Control**. Він надає багато членів, які потрібні для забезпечення основної функціональності інтерфейсу користувача. Наприклад, в класі **Control** визначені властивості для встановлення:

- розмірів елемента управління;
- прозорості;
- порядку обходу при натисненні клавіші <TAB>;
- відображуваного курсора;
- кольору фону і т. д.

Більше того, цей батьківський клас пропонує підтримку *шаблонних служб*. Як пояснюється – ***Лекція 11. Реалізація стилів WPF*** та ***Лекція 12.***

Логічні і візуальні дерева та шаблони – елементи управління WPF можуть повністю змінювати спосіб візуалізації свого зовнішнього вигляду, використовуючи *шаблони* і *стилі*. У табл. 1.6 коротко описані основні члени типу **Control**, згруповані за пов'язаною функціональністю.

Таблиця 1.6. Основні члени класу **Control**

Член	Опис
Background, Foreground, BorderBrush, BorderThickness, Padding, HorizontalContentAlignment, VerticalContentAlignment	Ці властивості дозволяють встановлювати базові налаштування, які стосуються того, як елемент управління візуалізуватиметься і позиціонуватиметься.
FontFamily, FontSize, FontStretch, FontWeight	Ці властивості управляють різноманітними налаштуваннями шрифтів.
IsTabStop, TabIndex	Ці властивості застосовуються для встановлення порядку обходу елементів управління у вікні при натисненні клавіші <TAB>.
MouseDoubleClick, PreviewMouseDoubleClick	Ці події обробляють дію подвійного клацання на віджеті.
Template	Ця властивість дозволяє отримувати і встановлювати шаблон елемента управління, який може бути використаний для зміни виводу візуалізації віджета.

Роль класу System.Windows.FrameworkElement

Базовий клас **FrameworkElement** надає декілька членів, які використовуються всюди в інфраструктурі WPF, у тому числі для підтримки:

- розкадровування (в анімації) і прив'язки даних;
- можливість іменування членів (через властивість **Name**);
- отримання будь-яких ресурсів, визначених похідним типом;
- встановлення загальних вимірів похідного типу.

Основні члени класу **FrameworkElement** коротко описані в табл. 1.7.

Таблиця 1.7. Основні члени класу **FrameworkElement**

Член	Опис
ActualHeight, ActualWidth, MaxHeight, MaxWidth, MinHeight, MinWidth, Height, Width	Ці властивості управляють розмірами похідного типу.
ContextMenu	Цю властивість отримує або встановлює спливаюче меню, асоційоване з похідним типом.
Cursor	Цю властивість отримує або встановлює курсор миші, асоційований з похідним типом.
HorizontalAlignment, VerticalAlignment	Ця властивість управляє позиціонуванням типу всередині контейнера (такого як панель або вікно зі списком).
Name	Ця властивість дозволяє призначати ім'я типу, щоб звертатися до його функціональності у файлі коду.
Resources	Ця властивість надає доступ до будь-яких ресурсів, які визначені типом (система ресурсів WPF пояснюється в ***Лекція 7. Ресурси WPF***).
ToolTip	Ця властивість отримує або встановлює спливаючу підказку, асоційовану з похідним типом.

Роль класу System.Windows.UIElement

З усіх типів у ланцюжку спадкоємства класу **Window** найбільший обсяг функціональності забезпечує базовий клас **UIElement**. Його основне завдання – надати похідному типу численні події, щоб він міг отримувати фокус та обробляти вхідні запити. Наприклад, в класі **UIElement** передбачена багато *подій* для обслуговування:

- операцій перетягування;
- переміщень курсора миші;
- клавіатурного вводу;
- вводу за допомогою пера (для **Pocket PC** і **Tablet PC**).

Модель подій WPF буде детально описана в ***Лекція 5. Події у WPF*** проте, багато основних подій вам вже відомі (**MouseMove**, **MouseDown**, **MouseEnter**, **MouseLeave**, **KeyUp** і т. д.). На додаток до десятка подій батьківський клас **UIElement** пропонує властивості, призначені для:

- управління фокусом;
- станом доступності;
- видимістю;
- логікою перевірки попадань (табл. 1.8).

Таблиця 1.8. Основні члени класу **UIElement**

Член	Опис
Focusable, sFocused	Ці властивості дозволяють встановлювати фокус на заданий похідний тип.
IsEnabled	Ця властивість дозволяє управляти доступністю заданого похідного типу.
IsMouseDirectlyOver, IsMouseOver	Ці властивості надають простий спосіб виконання логіки перевірки попадання.
IsVisible, Visibility	Ці властивості дозволяють працювати з налаштуванням видимості похідного типу.
RenderTransform	Ця властивість дозволяє встановлювати трансформацію, яка використовуватиметься при візуалізації похідного типу.

Роль класу System.Windows.Media.Visual

Клас **Visual** пропонує основну підтримку візуалізації у WPF, яка включає:

- перевірку попадання для графічних даних;
- координатну трансформацію;
- обчислення обмежуючих прямокутників.

Насправді при малюванні даних на екрані клас **Visual** взаємодіє з підсистемою **DirectX**. Як буде показано в ***Лекція 7. Служби візуалізації графіки WPF*** інфраструктура WPF підтримує три можливі способи візуалізації графічних даних, кожен з яких відрізняється в плані функціональності і продуктивності. Використання типу **Visual** (та його нащадків на зразок **DrawingVisual**) є найбільш легковагим шляхом візуалізації графічних даних, але потребує написання вручну великого обсягу коду для врахування всіх потрібних служб. Детальніше про це піде мова в ***Лекція 8. Візуалізація графічних даних***.

Роль класу System.Windows.DependencyObject

Інфраструктура WPF підтримує окремий різновид властивостей .NET під назвою *властивості залежності*. Говорячи простіше, цей стиль властивостей надає додатковий код, щоб дозволити властивості реагувати на певні технології WPF, такі як стилі, прив'язки даних, анімація і т. д. Щоб тип підтримував подібну схему властивостей, він має бути похідним від базового класу **DependencyObject**. Незважаючи на те що властивості залежності є ключовим аспектом розробки WPF, велику частину часу їх деталі приховані від очей. В ***Лекція 6. Модель прив'язки даних WPF*** ми розглянемо властивості залежності детальніше.

Роль класу System.Windows.Threading.DispatcherObject

Останнім базовим класом для типу **Window** (за винятком **System.Object**, який тут не вимагає додаткових пояснень) є **DispatherObject**. У ньому визначена одна цікава властивість **Dispatcher**, яка повертає асоційований об'єкт **System.Windows.Threading.Dispatcher**. Клас **Dispatcher** – це точка входу у чергу подій застосунку WPF, і він надає базові конструкції для організації *паралелізму* і *багатопоточності*.

Синтаксис XAML для WPF

На рівні розробки застосунки WPF зазвичай використовуватимуть окремі інструменти для генерації потрібної розмітки XAML. Які б не були зручні такі інструменти, важливо розуміти загальну структуру мови XAML. Для сприяння процесу вивчення доступний популярний (і безкоштовний) інструмент, який дозволяє легко експериментувати з XAML. Коли ви тільки приступаєте до вивчення граматики XAML, може виявитися зручним у використанні безкоштовний інструмент під назвою **Kaxaml**. Інсталяцію **Kaxaml** можна завантажити з адреси https://kaxaml.software.informer.com/1.8/.

Редактор Kaxaml не працює з початковим кодом C#, обробниками помилок або логікою реалізації. Він не претендує на тестування фрагментів XAML, як повноцінний шаблон проекту WPF у Visual Studio. Kaxaml володіє набором інтегрованих інструментів, зокрема: засобом вибору кольору, диспетчером фрагментів XAML і навіть засобом «очищення XAML», який форматує розмітку XAML на основі заданих налаштувань. Відкривши Kaxaml вперше, ви знайдете в ньому просту розмітку для елемента управління <**Page**>:

```
<Page
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>

  </Grid>
</Page>
```

Об'єкт **Page** як і **Window**, містить різноманітні диспетчери компонування та елементи управління. Але, на відміну від **Window** об'єкти **Page** не можна запускати як окремі сутності. Об'єкти **Page** повинні поміщатися всередину

відповідного хоста, такого як **NavigationWindow** або **Frame**. В елементах <**Page**> і <**Window**> можна вводити ідентичну розмітку.

Як початковий тест введемо наступну розмітку в панелі XAML, яка знаходиться в нижній частині вікна Kaxaml:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка зі спеціальним вмістом -->
    <Button Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

У верхній частині вікна Kaxaml з'явиться візуалізована сторінка (рис. 1.2).

Інструмент Kaxaml не дозволяє створювати розмітку з можливістю компіляції коду, визначення атрибуту x:Class (для задання файла коду), ввід імен обробників подій в розмітці або використання будь-яких ключових слів XAML, які також передбачають компіляцію коду (на зразок **FieldModifier** або **ClassModifier**). Спроба зробити так спричинить помилку розмітки.

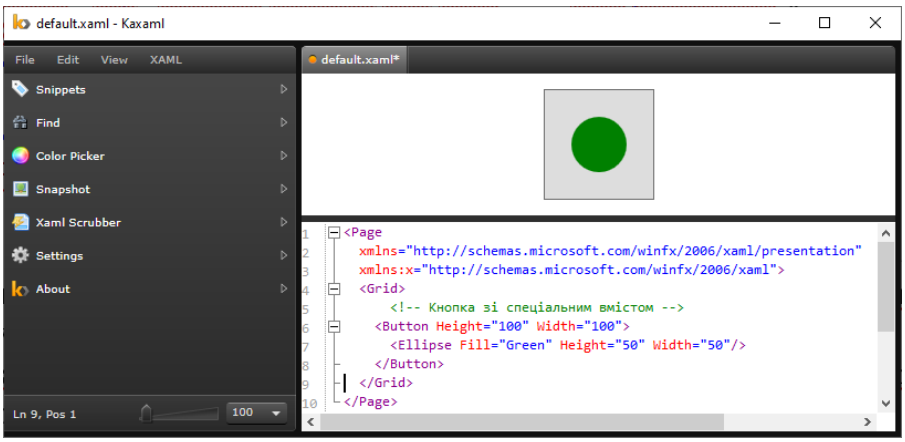


Рис. 1.2. Редактор Kaxaml зручний (і безкоштовний) інструмент, використовуваний при вивченні граматики XAML

Простори імен XML і «ключові слова» XAML

Кореневий елемент XAML-документа WPF (<**Window**>, <**Page**>, <**UserControl**> або <**Application**>) майже завжди посилатиметься на два заздалегідь визначені простори імен XML:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
```

```
  </Grid>
</Page>
```

Простір імен XML, http://schemas.microsoft.com/winfx/2006/xaml/presentation, відображає множину пов'язаних з WPF просторів імен .NET для використання поточним файлом *.xaml (**System.Windows, System.Windows.Controls, System.Windows.Data, System.Windows.Ink, System.Windows.Media, System.Windows.Navigation** і т. д.).

Це відображення «один до багатьох» насправді жорстко закодовано всередині збірок WPF (**WindowsBase.dll, PresentationCore.dll** і **PresentationFramework.dll**) із використанням атрибуту [**XmlnsDefinition**] рівня збірки. Наприклад, якщо відкрити браузер об'єктів Visual Studio і вибрати збірку **PresentationCore.dll**, то можна побачити списки, подібні до показаного нижче, в якому імпортується простір імен **System.Windows**:

```
[assembly:XmlnsDefinition("http://schemas.microsoft.com/wmfx/2006/xaml/presentation",
"System.Windows")]
```

Другий простір імен XML, http://schemas.microsoft.com/winfx/2006/xaml, використовується для додавання специфічних для XAML «ключових слів» (термін вибраний за відсутності кращого), а також простори імен **System.Windows.Markup**:

```
[assembly:XmlnsDefinition("http://schemas.microsoft.com/winfx/2006/xaml",
"System.Windows.Markup")]
```

Одне з правил будь-якого коректно сформованого документа XML полягає в тому, що відкриваючий кореневий елемент призначає один простір імен XML як первинний простір імен, який зазвичай є простором імен, що містить найчастіше використовувані елементи. Якщо кореневий елемент вимагає включення додаткових вторинних просторів імен (як бачимо тут), то вони мають бути визначені з використанням *унікального префікса* (щоб усунути можливі конфлікти імен). За угодою для префікса використовується просто **x**, проте він може бути будь-яким унікальним маркером, наприклад, таким як **XamlSpecificStuff**:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:XamlSpecificStuff="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Кнопка зі спеціальним вмістом -->
    <Button XamlSpecificStuff:Name="button1" Height="100" Width="100">
      <Ellipse Fill="Green" Height="50" Width="50"/>
    </Button>
  </Grid>
</Page>
```

Очевидний недолік визначення довгих префіксів для просторів імен XML пов'язаний з тим, що **XamlSpecificStuff** доведеться набирати всякий раз, коли у файлі XAML треба послатися на один з елементів, визначених в цьому просторі

імен XML. Через те, що префікс **XamlSpecificStuff** надто довгий, давайте обмежимося **x**.

Простір імен http://schemas.microsoft.com/winfx/2006/xaml окрім ключових слів x:Name, x:Class і x:Code також надає доступ до додаткових ключових слів XAML, найпоширеніші з яких коротко описані в табл. 1.9.

Таблиця 1.9. Ключові слова XAML

Ключове слово XAML	Опис
x:Array	Представляє у XAML тип масиву .NET.
x:ClassModifier	Дозволяє визначати видимість класу C# (**internal** або **public**), позначеного ключовим словом **Class**.
x:FieldModifier	Дозволяє визначати видимість члена типу (**internal**, **public**, **private** або **protected**) для будь-якого іменованого піделемента кореня (наприклад, <**Button**> всередині елемента <**Window**>). Іменований елемент визначається з використанням ключового слова **Name** у XAML.
x:Key	Дозволяє встановлювати значення ключа для елемента XAML, яке буде поміщене в елемент словника.
x:Name	Дозволяє задавати згенероване ім'я C# заданого елемента XAML.
x:Null	Представляє посилання **null**.
x:Static	Дозволяє посилатися на статичний член типу.
x:Type	Еквівалент XAML операції **typeof** мови C# (вона видаватиме об'єкт **System.Type** на основі заданого імені).
x:TypeArguments	Дозволяє встановлювати елемент як узагальнений тип з певним параметром типу (наприклад, **List<int>** або **List<bool>**).

На додаток до двох зазначених оголошень просторів імен XML можна (а іноді і треба) визначити додаткові префікси дескрипторів у відкриваючому елементі документа XAML. Зазвичай так чинять, коли потрібно описати у XAML клас .NET, визначений у зовнішній збірці.

Наприклад, припустимо, що було побудовано декілька спеціальних елементів управління WPF, які запаковані у бібліотеку на ім'я **MyControls.dll**. Якщо тепер вимагається створити новий об'єкт **Window**, в якому використовуються створені елементи, то можна встановити спеціальний простір імен XML, який відображається на бібліотеку **MyControls.dll**, з використанням маркерів **clr-namespace** та **assembly**. Нижче наведений приклад розмітки, яка створює префікс дескриптора на ім'я **myCtrls**, який може використовуватися для доступу до елементів управління в цій бібліотеці:

```
<Window x:Class="Lk01Ex01.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:myCtrls="clr-namespace:MyControls;assembly=MyControls"
  Title="MainWindow" Height="350" Width="525">
  <Grid>
    <myCtrls:MyCustomControl />
  </Grid>
</Window>
```

Маркеру **clr-namespace** призначається назва простору імен .NET у збірці, в той час як маркер **assembly** встановлюється у відповідне ім'я зовнішньої збірки ***.dll**. Такий синтаксис можна використати для будь-якої зовнішньої бібліотеки .NET, якою потрібно скористатися всередині розмітки. Нині в цьому немає потреби, але в наступних лекціях знадобиться визначати спеціальні оголошення просторів імен XML для опису типів в розмітці.

> **На замітку!** Якщо треба визначити в розмітці клас, який є частиною поточної збірки, але знаходиться в іншому просторі імен .NET, то префікс дескриптора xmlns визначається без атрибута assembly=:
> xmlns:myCtrls="clr-namespace:SomeNamespaceInMyApp"

Управління видимістю класів і змінних-членів

Багато ключових слів ви побачите у дії в наступних лекціях там, де вони знадобляться; проте, як простий приклад подивіться на наступне XAML-визначення <**Window**>, у якому застосовуються ключові слова **ClassModifier** і **FieldModifier**, а також x:Name і x:Class:

```
<!-- Цей клас тепер буде оголошений як internal у файлі *.g.cs -->
<Window x:Class="Lk01Ex01.MainWindow" x:ClassModifier="internal"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <!-- ця кнопка буде оголошена як public у файлі *.g.cs -->
  <Button x:Name="myButton" x:FieldModifier="public" Content="OK"/>
</Window>
```

За замовчуванням всі визначення типів C#/XAML є відкритими (**public**), а члени – внутрішніми (**internal**). Проте на основі показаного визначення XAML результуючий автоматично згенерований файл містить внутрішній тип класу з відкритою змінною-членом **Button**:

```
internal partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
  public System.Windows.Controls.Button myButton;
    ...
}
```

Елементи XAML, атрибути XAML і перетворювачі типів

Після встановлення кореневого елемента і потрібних просторів імен XML наступне завдання полягає в наповненні кореня *дочірнім елементом*. У реальному застосунку WPF дочірнім елементом буде диспетчер компонування (такий як **Grid** або **StackPanel**), який у свою чергу містить довільне число додаткових елементів, які описують інтерфейс користувача. Такі диспетчери компонування розглядаються в ***Лекція 3. Елементи управління у WPF***, а поки припустимо, що елемент **<Window>** міститиме єдиний елемент **Button**.

Як було показано раніше, *елементи* XAML відображаються на типи класів або структур всередині заданого простору імен .NET, тоді як *атрибути* у відкриваючому дескрипторі елемента відображаються на властивості або події

конкретного типу. В цілях ілюстрації введемо в редакторові Kaxaml наступне визначення <**Button**>:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml''>
  <Grid>
    <!-- Сконфігурувати зовнішній вигляд елемента Button -->
    <Button Height="50" Width="100" Content="OK""
    FontSize="20" Background="Green" Foreground="Yellow"/>
  </Grid>
</Page>
```

Зауважимо, що значення, присвоєні властивостям, представлені за допомогою простого тексту. Це може виглядати як повна невідповідність типам даних, бо після створення такого елемента **Button** в коді C# цим властивостям присвоюватимуться не рядкові об'єкти, а значення специфічних типів даних. Наприклад, нижче показано, як та ж сама кнопка описана в коді:

```
public void MakeAButton()
{
  Button myBtn = new Button();
  myBtn.Height = 50;
  myBtn.Width = 100;
  myBtn.FontSize = 20;
  myBtn.Content = "OK!";
  myBtn.Background = new SolidColorBrush(Colors.Green);
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow) ;
}
```

Виявляється, що інфраструктура WPF постачається з декількома класами *перетворювачів типів*, які використовуватимуться для трансформації простих текстових значень в коректні типи даних. Такий процес відбувається прозоро та автоматично.

Проте, нерідко виникає потреба у присвоєнні атрибуту XAML набагато складнішого значення, яке неможливо виразити за допомогою простого рядка. Наприклад, нехай потрібно побудувати спеціальну кисть для встановлення властивості **Background** елемента **Button**. Створити кисть подібного роду в коді не складно:

```
public void MakeAButton()
{
  // Незвичайна кисть для фону.
  LinearGradientBrush fancyBruch = new LinearGradientBrush(Colors.DarkGreen, Colors.LightGreen,
45);
  myBtn.Background = fancyBruch;
  myBtn.Foreground = new SolidColorBrush(Colors.Yellow);
}
```

Але чи можна зобразити цю складну кисть у вигляді рядка? Ні, не можна. На щастя, у XAML передбачений спеціальний синтаксис, який можна

використати всякий раз, коли треба присвоїти складний об'єкт як значення властивості; він називається синтаксисом "*властивість-елемент*".

Поняття синтаксису «властивість-елемент» у XAML

Синтаксис «властивість-елемент» *дозволяє присвоювати властивості складні об'єкти*. Нижче показаний опис XAML елемента **Button**, в якому для встановлення властивості **Background** застосовується об'єкт **LinearGradientBrush**:

```
<Button Height="50" Width="100" Content="OK!" FontSize="20" Foreground="Yellow">
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Color="DarkGreen" Offset="0"/>
      <GradientStop Color="LightGreen" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Зверніть увагу, що всередині дескрипторів <**Button**> і </**Button**> визначена вкладена область за іменем <**Button.Background**>, а в ній – спеціальний елемент <**LinearGradientBrush**>. (Поки не турбуйтеся про код кисті; ви розглянете графіку WPF в ***Лекція 7. Служби візуалізації графіки WPF*** та ***Лекція 8. Візуалізація графічних даних***.)

Взагалі кажучи, будь-яка властивість може бути встановлена з використанням синтаксису "властивість-елемент", який завжди зводиться до наступного шаблону:

```
<ВизначальнийКлас>
  <ВизначальнийКлас.ВластивістьВизначальногоКласу>
    <!-- Значення для властивості визначального класу -->
  </ВизначальнийКлас.ВластивістьВизначальногоКласу>
</ВизначальнийКлас>
```

Хоча будь-яка властивість може бути встановлена з використанням такого синтаксису, задання значення у вигляді простого рядка, коли подібне можливе, економитиме час вводу. Наприклад, набагато багатослівніший спосіб встановлення властивості **Width** елемента **Button**:

```
<Button Height="50" Content="OK!" FontSize="20" Foreground="Yellow">
  …
<Button Height="50" Content="OK!" FontSize="20" Foreground^'Yellow">
  <Button.Width>100</Button.Width>
</Button>
```

Поняття приєднуваних властивостей XAML

На додаток до синтаксису "властивість-елемент" у XAML підтримується спеціальний синтаксис, використовуваний для встановлення значення *приєднуваної властивості*. Насправді приєднувана властивість дозволяє дочірньому елементу встановлювати значення властивості, яке насправді

визначена у батьківському елементі. Загальний шаблон, якого треба дотримуватися, виглядає так:

```
<БатьківськийЕлемент>
  <ДочірнійЕлемент БатьківськийЕлемент.ВластивістьБатьківськогоЕлемента =
    "Значення">
</БатьківськийЕлемент >
```

Найпоширеніше використання синтаксису приєднуваних властивостей пов'язане з позиціонуванням елементів інтерфейсу користувача всередині одного з класів диспетчерів компонування (**Grid**, **DockPanel** і т. д.). Диспетчери компонування детальніше розглядаються в ***Лекція 3. Елементи управління у WPF***, а поки введемо в редакторові Kaxaml наступну розмітку:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Canvas Height="200" Width="200" Background="LightBlue">
    <Ellipse Canvas.Top="40" Canvas.Left="40" Height="20" Width="20" Fill="DarkBlue"/>
  </Canvas>
</Page>
```

Тут заданий диспетчер компонування **Canvas**, який містить елемент **Ellipse**. Зауважимо, що за допомогою синтаксису приєднуваних властивостей елемент **Ellipse** може інформувати свій батьківський елемент (**Canvas**) про те, де розташовувати позицію його лівого верхнього кута.

Щодо приєднуваних властивостей треба мати на увазі декілька моментів. Передусім, це *не універсальний синтаксис*, який може застосовуватися до будь-якої властивості будь-якого батьківського елемента. Наступна розмітка XAML містить помилку:

```
<!-- Спроба встановлення властивості Background в Canvas через приєднувані властивості.
Помилка! -->
<Canvas Height="200" Width="200">
  <Ellipse Canvas.Background="LightBlue"
      Canvas.Top="40" Canvas.Left="90"
      Height="20" Width="20" Fill="DarkBlue"/>
</Canvas>
```

Насправді приєднувані властивості є спеціалізованою формою специфічної для WPF концепції, яка називається *властивістю залежності*. Якщо тільки властивість не була реалізована в надто спеціалізованій манері, то її значення не може бути встановлене з використанням синтаксису приєднуваних властивостей. Властивості залежності детально досліджуються в ***Лекція 6. Модель прив'язки даних***.

Поняття розширень розмітки XAML

Як вже пояснювалося, значення властивостей найчастіше представляються у вигляді простого рядка або через синтаксис "властивість-елемент". Проте існує ще один спосіб задати значення атрибуту XAML –

використання *розширень розмітки*. Розширення розмітки дозволяють аналізатору XAML отримувати значення для властивості з виділеного зовнішнього класу. Це може забезпечити великі переваги, бо для отримання значень деяких властивостей потрібне виконання багато операторів коду.

Розширення розмітки пропонують спосіб акуратного *розширення граматики* XAML новою функціональністю. Розширення розмітки внутрішньо представлене як клас, похідний від **MarkupExtension**. Треба зазначити, що потреба в побудові спеціального розширення розмітки виникає украй рідко. Проте, деякі ключові слова XAML (на зразок **x:Array**, **x:Null**, **x:Static** і **x:Type**) – це замасковані розширення розмітки.

Розширення розмітки поміщається між фігурними дужками:
<Елемент ВстановлюванаВластивість = "{РозширенняРозмітки}"/>

Щоб побачити розширення розмітки у дії, введемо в редакторі Kaxaml наступний код:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:CorLib="clr-namespace:System;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <StackPanel>
   <!-- Розширення розмітки Static дозволяє отримувати значення статичного члена класу -->
       <Label Content="{x:Static CorLib:Environment.OSVersion}"/>
       <Label Content="{x:Static CorLib:Environment.ProcessorCount}"/>
   <!-- Розширення розмітки Type – це версія XAML операції typeof мови C# -->
       <Label Content="{x:Type Button}"/>
       <Label Content="{x:Type CorLib:Boolean}"/>
   <!-- Наповнення елемента ListBox масивом рядків -->
       <ListBox Width="200" Height="50">
          <ListBox.ItemsSource>
             <x:Array Type="CorLib:String">
                <CorLib:String>Sun Kil Moon</CorLib:String>
                <CorLib:String>Red House Painters</CorLib:String>
                <CorLib:String>Besnard Lakes</CorLib:String>
             </x:Array>
          </ListBox.ItemsSource>
       </ListBox>
   </StackPanel>
</Page>
```

Передусім, зверніть увагу, що визначення <**Page**> містить нове оголошення простору імен XML, яке дозволяє отримувати доступ до простору імен **System** збірки **mscorlib.dll**. Після встановлення цього простору імен XML насамперед за допомогою розширення розмітки **x:Static** отримуються значення властивостей **OSVersion** і **ProcessorCount** класу **System.Environment**.

Розширення розмітки **x:Type** забезпечує доступ до опису метаданих вказаного елемента. Тут вмісту елементів **Label** просто присвоюються повністю задані імена типів **Button** і **System.Boolean** з WPF.

Найцікавіша частина показаної вище розмітки пов'язана з елементом **ListBox**. Його властивість **ItemSource** встановлюється в масив рядків, повністю оголошений в розмітці. Бачимо, що розширення розмітки **x:Array** дозволяє задавати набір піделементів в зоні своєї дії:

```
<x:Array Type="CorLib:String">
   <CorLib:String>Sun Kil Moon </CorLib:String>
        <CorLib:String>Red House Painters</CorLib:String>
        <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

На рис. 1.3 представлена розмітка цього елемента <**Page**> у редакторі Kaxaml.

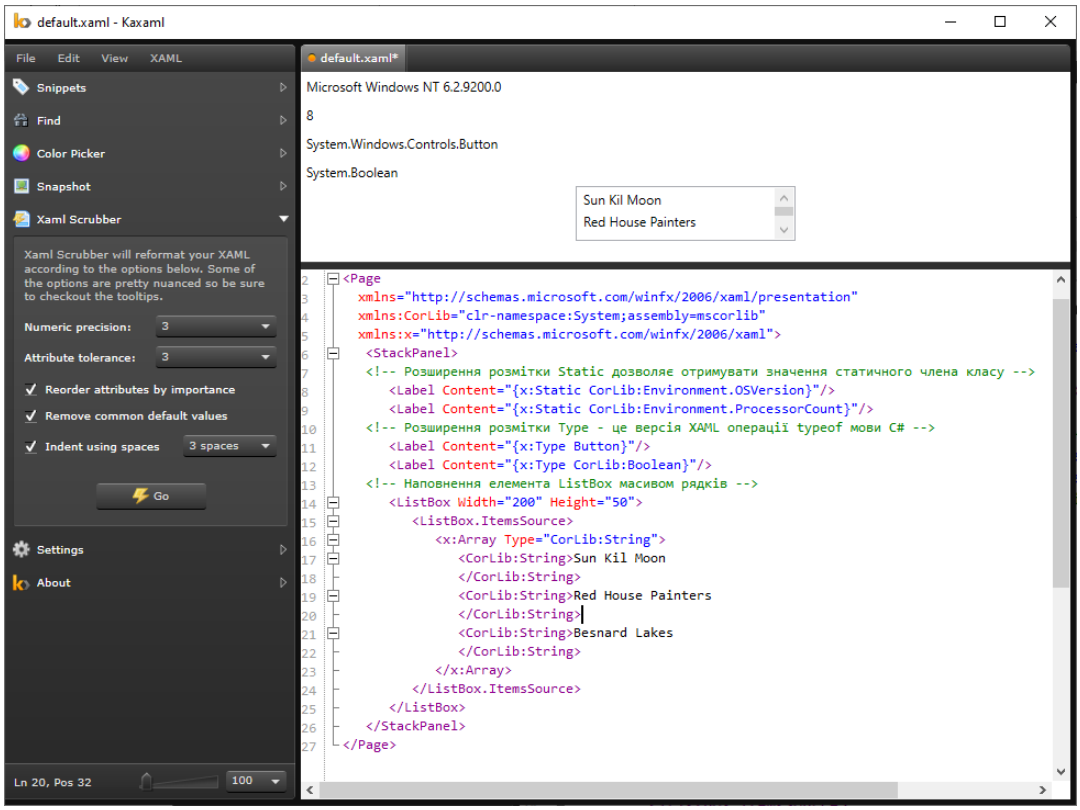


Рис. 1.3. Розширення розмітки дозволяють задавати значення через функціональність виділеного класу

Ви вже побачили багато прикладів, які демонстрували основні аспекти синтаксису XAML. Ви напевно погодитеся, що XAML цікавий своєю можливістю описувати дерева об'єктів .NET в декларативній манері. Хоча це винятково корисно при конфігурації графічних інтерфейсів користувача, не забувайте про те, що за допомогою XAML можна описувати будь-який тип з будь-якої збірки за умови, що він є неабстрактним і містить стандартний конструктор.

Резюме

Інфраструктура WPF це набір інструментів для побудови інтерфейсів, що з'явився у версії .NET 3.0. Основна мета WPF – інтеграція та уніфікація раніше розрізнених настільних технологій (двовимірна і тривимірна графіка, розробка вікон та елементів управління і т. п.) в єдину уніфіковану програмну модель. Окрім цього в застосунках WPF зазвичай використовується мова XAML, яка визначає зовнішній вигляд і поведінку елементів WPF через розмітку.

Мова XAML дозволяє описувати дерева об'єктів .NET з використанням декларативного синтаксису. Під час дослідження XAML у цій лекції ви довідалися про декілька нових фрагментів синтаксису, включаючи синтаксис "*властивість-елемент*" і *приєднувані властивості*, а також про *роль перетворювачів типів* і *розширень розмітки* XAML.

Контрольні питання.

1. Які графічні інструменти Microsoft створила для побудови графічних інтерфейсів користувача (ГІК)?
2. В чому суть асиметричності програмної моделі Windows Forms? Наведіть приклади.
3. В чому суть уніфікованої об’єктної моделі WPF? Наведіть приклади.
4. Яка одна з найзначніших переваг технології WPF?
5. Призначення мови XAML?
6. Призначення пов’язаного файла коду?
7. В чому суть програмної моделі WPF при візуалізації (двовимірна і тривимірна графіка, анімація, візуалізація елементів управління і т. д.) даних на відміну від Windows Forms (двовимірна і тривимірна графіка, анімація, візуалізація елементів управління і т. д.) даних?
8. Які основні функціональні можливості WPF?
9. Чи входять до основних функціональних можливостей WPF: *диспетчери компонування, розширений механізм прив’язки, вбудований механізм стилів, автоматична зміна розмірів вмісту, підтримка графіки (двовимірної, тривимірної), анімації, відтворення відео та аудіо*?
10. Чи підтримує WPF розвинені типографські API-інтерфейси?
11. Чи підтримує WPF взаємодію з успадкованими моделями графічних інтерфейсів?
12. Перерахуйте основні збірки WPF?
13. Що входить до складу збірки?
14. Назвіть основні простори імен WPF.
15. Охарактеризуйте основні простори імен WPF.
16. Яке основне призначення класу Application?
17. Основні властивості класу Application.
18. Яке основне призначення класу Window?
19. Яка роль класу Window?
20. Який клас є безпосереднім батьківським класом Window?
21. Основні властивості та члени класу Control.
22. Основні властивості та члени класу FrameworkElement.

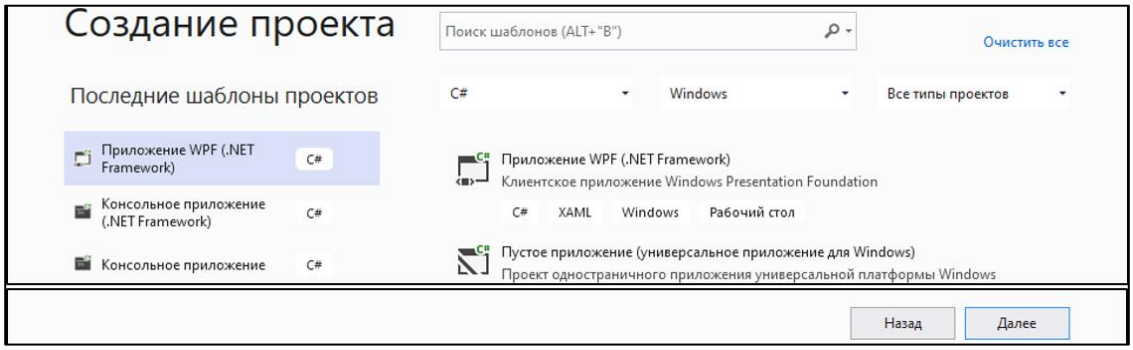
23. Основні властивості та члени класу UIElement.
24. Яку основну функціональність надає клас Visual?
25. Для підтримки яких властивостей використовується клас DependencyObject?
26. Для реалізації яких можливостей використовується клас DispatcherObject?
27. Основні простори імен XML та ключові слова XAML?
28. Для чого призначені префікси просторів імен?
29. У яких випадках використовують додаткові префікси?
30. У яких випадках використовуються ключові слова ClassModifier і FieldModifier, а також x:Name і x:Class?
31. Поясніть суть поняття синтаксису «властивість-елемент» у XAML?
32. Поясніть суть поняття синтаксису «приєднуваної властивості»?
33. Поясніть такий спосіб задання значення атрибуту XAML як «розширення розмітки»?

Лекція 2. Побудова застосунків WPF з використанням Visual Studio

Давайте з'ясуємо, як Visual Studio може спростити створення програм WPF. За цією адресою https://docs.microsoft.com/ru-ru/visualstudio/xaml-tools/xaml-code-editor?view=vs-2022 ви знайдете інформацію про: засоби XAML у Visual Studio; загальні відомості про XAML; редактор коду XAML; конструктор XAML; налаштування XAML; Windows Presentation Foundation (WPF).

Шаблони проектів WPF

У діалоговому вікні **Створення проекту** середовища Visual Studio визначений набір робочих просторів проектів WPF. Тут можна вибрати з декількох шаблонів проектів, включаючи **Застосунок WPF**, **Застосунок Windows Forms** і багато інших. Спершу створимо новий проект застосунка WPF на ім'я **WpfTestApp** (рис. 2.1). Все сказане стосується Visual Studio 2019.



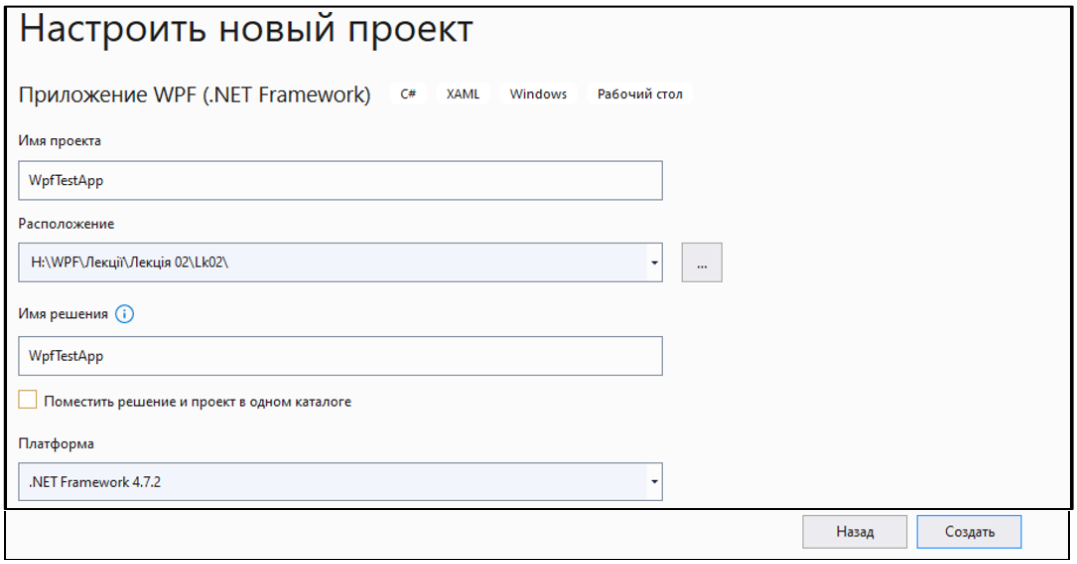


Рис. 2.1. Шаблони проектів WPF

Застосунок WpfTestApp

Код розмітки **MainWindow.xaml**

```
<Window x:Class="WpfTestApp.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:WpfTestApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800" Background="#FF1F4CE3"
          MouseMove="Window_MouseMove" Closing="Window_Closing" Closed="Window_Closed"
          KeyDown="Window_KeyDown" >
  <Canvas>
    <Canvas.Background>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="Black" Offset="0"/>
        <GradientStop Color="#FF1F4CE3" Offset="1"/>
      </LinearGradientBrush>
    </Canvas.Background>
    <Button x:Name="ClickMe" Content="Button" HorizontalAlignment="Left" VerticalAlignment="Top"
      Width="75" Click="Button_Click"/>
    <Calendar x:Name="MyCalendar" HorizontalAlignment="Left" VerticalAlignment="Top"
      SelectedDatesChanged="MyCalendar_SelectedDatesChanged" Canvas.Left="75"
      Canvas.Top="54"/>
  </Canvas>
</Window>
```

Доповнюючий **C#-код**

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;

namespace WpfTestApp
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
      this.Closed += Window_Closed;
      this.Closing += Window_Closing;
      this.MouseMove += Window_MouseMove;
      this.KeyDown += Window_KeyDown;
    }
```

```
    private void Button_Click(object sender, RoutedEventArgs e)
    {
      MessageBox.Show("You clicked the button!");
      // Ви клацнули на кнопці!
    }
    private void Window_MouseMove(object sender, MouseEventArgs e)
    {
      this.Title = e.GetPosition(this).ToString();
    }
    private void MyCalendar_SelectedDatesChanged(object sender, SelectionChangedEventArgs e)
    {

    }
    private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
    {
      // З'ясувати, чи справді користувач хоче закрити вікно.
      string msg = "Do you want to close without saving?";
      MessageBoxResult result = MessageBox.Show(msg,
        "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
      if (result == MessageBoxResult.No)
      {
        // Якщо користувач не хоче закривати вікно, то відмініть закриття.
        e.Cancel = true;
      }
    }
    private void Window_Closed(object sender, EventArgs e)
    {
      MessageBox.Show("See ya!");
    }
    private void Window_KeyDown(object sender, KeyEventArgs e)
    {
      // Відобразити на кнопці інформацію про натиснуту клавішу.
      ClickMe.Content = e.Key.ToString();
    }
  }
}
```

Результат запуску застосунку **WpfTestApp** зображений на рис. 2.7.

Окрім встановлення посилань на всі збірки WPF (**PresentationCore.dll**, **PresentationFramework.dll**, **System.Xaml.dll** і **WindowsBase.dll**) ви отримаєте початкові класи, похідні від **Window** та **Application**, кожен з яких представлений із використанням XAML і файла коду C#.

Далі ми ретельно проаналізуємо файли, які входять до проекту **WpfTestApp**. Таке ім'я передбачуване, бо ми розмістили рішення і проект в одному каталозі.

Вам пропонується ретельно дослідити склад файлів, які входять до складу рішення згідно викладу матеріалу пропонованого далі в лекції, починаючи з наступного пункту.

Панель інструментів і візуальний конструктор/редактор XAML

У Visual Studio є панель інструментів, яка відкривається через меню **View** (Вид), що містить численні елементи управління WPF. У верхній частині панелі розташовані найчастіше використовувані елементи управління, а в нижній частині – всі елементи управління (рис. 2.2).

Використовуючи стандартну операцію перетягування за допомогою миші елемент управління можна помістити у візуальний конструктора елемента **Window** або перетягнути його на область редактора розмітки XAML внизу вікна візуального конструктора. Після цього початкова розмітка XAML згенерується автоматично. Перетягнемо за допомогою миші елементи управління **Button** і **Calendar** на візуальний конструктор. Є можливість зміни позиції і розміру елементів управління (перегляньте результуючу розмітку XAML, згенеровану на основі змін).

На додаток до побудови інтерфейсу з використанням миші і панелі інструментів, розмітку можна також вводити вручну із застосуванням інтегрованого редактора XAML. Як показано на рис. 2.2, ви отримуєте підтримку засобу **IntelliSense**, який допомагає спростити написання розмітки. Для прикладу додамо властивість **Background** у відкриваючий елемент <**Window**>.

Приділіть деякий час на додавання значень властивостей безпосередньо в редакторі XAML. Обов'язково опануйте цей аспект візуального конструктора WPF.

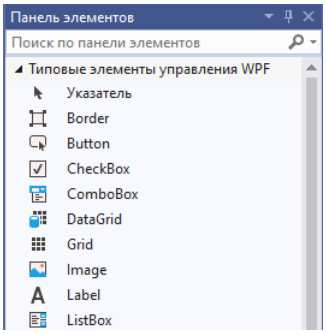


Рис. 2.2. Панель інструментів з елементами управління WPF

Встановлення властивостей з використанням вікна Properties

Після поміщення декількох елементів управління на поверхню візуального конструктора (або визначення їх в редакторі коду вручну) можна відкрити вікно **Properties** (Властивості) для встановлення *значень* властивостей виділеного елемента управління, а також для створення пов'язаних з ним *обробників* подій. Як простий приклад виберемо у візуальному конструкторі раніше доданий елемент управління **Button**. Із використанням вікна **Properties** змінимо колір у властивості **Background** елемента **Button**, використовуючи вбудований редактор кистей (рис. 2.3); редактор кистей детальніше розглядатиметься в ***Лекція 7. Служби візуалізації графіки WPF*** та ***Лекція 8. Візуалізація графічних даних*** під час дослідження графіки WPF.

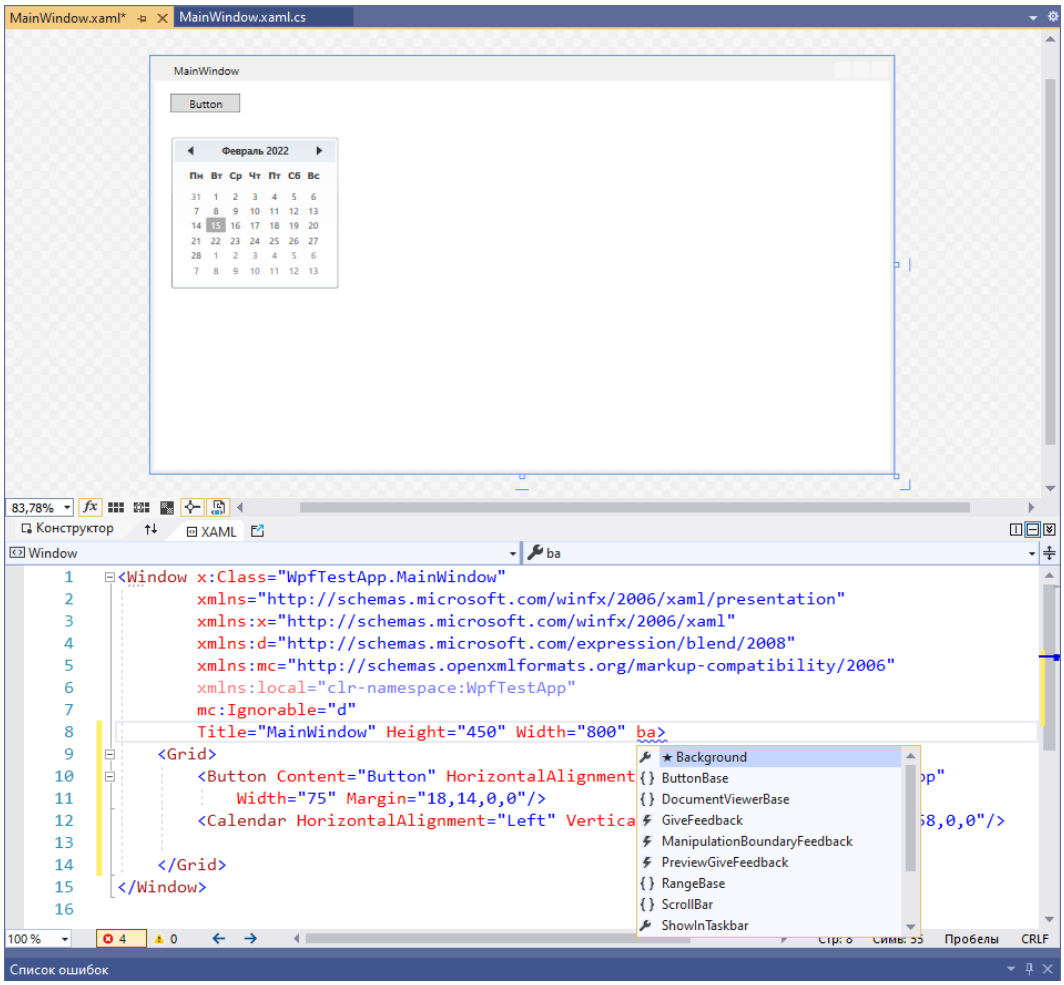


Рис. 2.2. Візуальний конструктор елемента **Window**

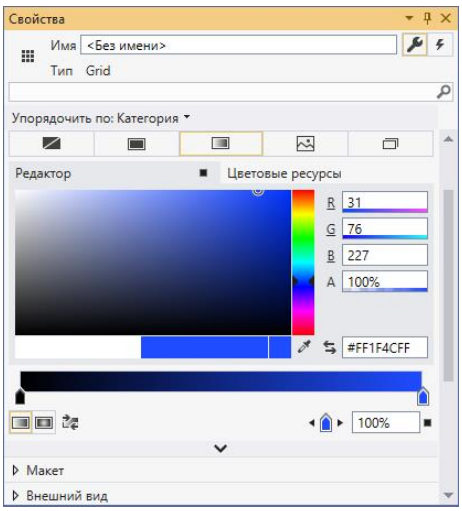


Рис. 2.3. Вікно **Properties** може застосовуватися для конфігурації елемента управління WPF інтерфейсу користувача

Обробка подій з використанням вікна Properties

Для організації обробки подій, пов'язаних з певним елементом управління, також можна застосовувати вікно **Properties**, але цього разу знадобиться клацнути на кнопці **Events** (Події), розташованій справа зверху вікна (кнопка із зображенням блискавки). На поверхні візуального конструктора виберемо елемент **Button**, якщо він ще не вибраний, клацнемо на кнопці **Events** у вікні **Properties** і двічі клацнемо на поле для події **Click**. Середовище Visual Studio автоматично побудує обробник подій, ім'я якого має наступну загальну форму:

Ім'яЕлементаУправління_Ім`яПодії

Через те, що кнопка не була перейменована, у вікні **Properties** відображається згенерований обробник подій на ім'я **Button_Click** (рис. 2.4).

Крім того, Visual Studio згенерує відповідний обробник події C# у файлі коду для вікна. У нього можна помістити будь-який код, який повинен виконуватися, коли на кнопці здійснене клацання. Як простий приклад додамо наступний оператор коду:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show ("You clicked the button!");
  // Ви клацнули на кнопці!
}
```

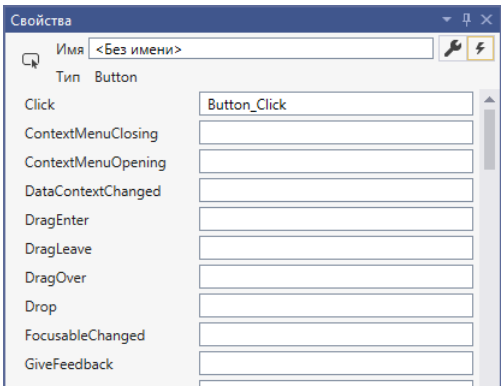


Рис. 2.4. Обробка подій з використанням вікна **Properties**

Обробка подій в редакторі XAML

Обробляти події можна і безпосередньо в редакторі XAML. Для прикладу помістимо курсор миші всередину елемента <**Window**> і введемо ім'я події **MouseMove**, а за ним знак рівності. Середовище Visual Studio відобразить всі сумісні обробники з файла коду (якщо вони існують), а також пункт **Create method** (Створити метод), як показано на рис. 2.5.

Дозволимо IDE-середовищу створити обробник події **MouseMove**, введемо наступний код і запустимо застосунок, щоб побачити кінцевий результат:

```
private void Window_MouseMove(object sender, MouseEventArgs e)
{
   this.Title = e.GetPosition(this).ToString();
}
```

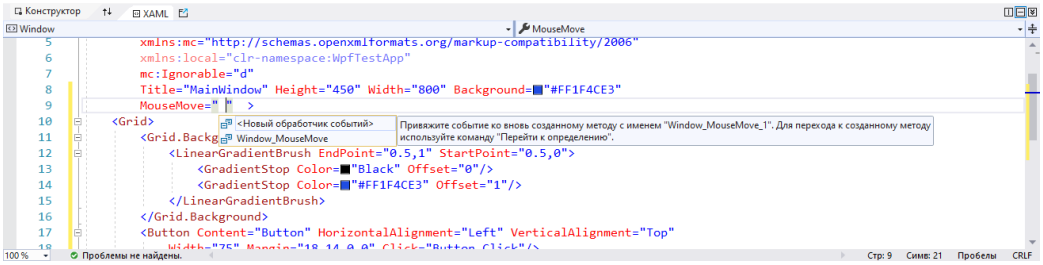


Рис. 2.5. Обробка подій із використанням редактора XAML

Вікно Document Outline

Під час роботи з будь-яким проектом, який базується на XAML ви безперечно використовуватимете значний обсяг розмітки для представлення інтерфейсу користувача. Коли ви почнете стикатися зі складнішою розміткою XAML, може виявитися зручною візуалізація розмітки для швидкого вибору елементів з метою редагування у візуальному конструкторі Visual Studio.

Нині наша розмітка досить проста, бо було визначено лише декілька елементів управління всередині початкового елемента <**Grid**>. Проте, потрібно знайти вікно **Document Outline** (Структура документу), яке за замовчуванням розташовується в лівій частині вікна IDE-середовища (якщо виявити його не вдається, то це вікно можна відкрити через пункт меню **View|Other Windows** (Вид|Інші вікна)). При активному вікні візуального конструктора XAML (не у вікні з файлом коду C#) в IDE-середовищі можна помітити, що у вікні **Document Outline** (Структура документа) відображаються вкладені елементи (рис. 2.6).

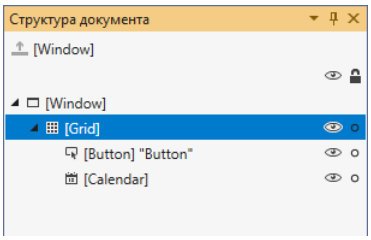


Рис. 2.6. Візуалізація розмітки XAML у вікні **Структура документа**

Цей інструмент також надає спосіб тимчасового приховування заданого елемента (або набору елементів) на поверхні візуального конструктора, а також блокування елементів з метою запобігання їх подальшого редагування. В «***Лекції 3. Елементи управління у WPF***» та «***Лекції 4. Побудова вікон***» ви побачите, що вікно **Document Outline** пропонує багато інших можливостей для групування вибраних елементів усередині нових диспетчерів компонування (окрім інших засобів).

Включення і відключення відлагодження XAML

Після запуску застосунку на екрані з'являється вікно **MainWindow**. Крім того, можна також бачити інтерактивний відлагоджувач (рис. 2.7).

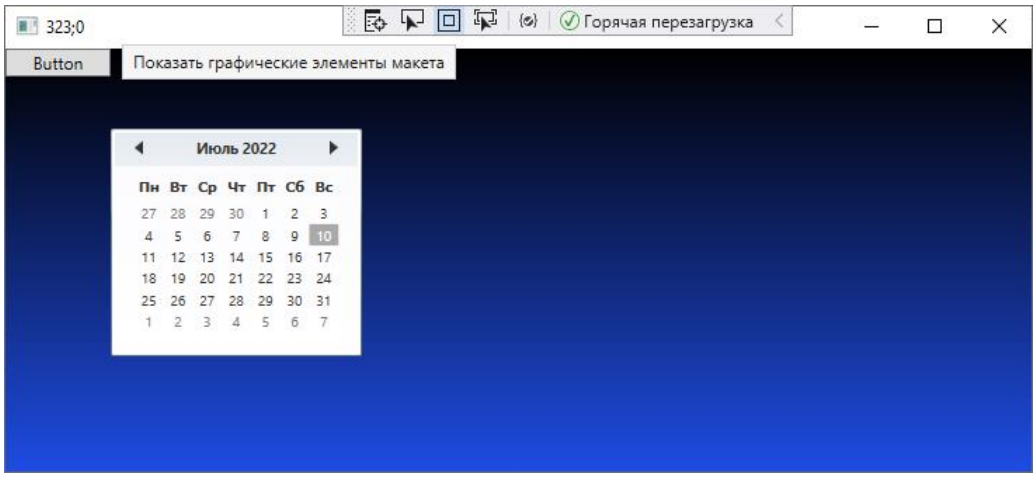


Рис. 2.7. Налаштування, пов'язані з налаштуванням інтерфейсу користувача XAML

Дослідження файла App.xaml

Як проект знає про те, яке вікно відображати? Ще більша інтрига в тому, що в результаті дослідження файлів коду, які відносяться до застосунку, **метод Main()** виявити не вдасться. Ви вже знаєте, що застосунки повинні мати точку входу, бо як же інакше інфраструктурі .NET стає відомо, як запускати застосунок. На щастя, обидва зв'язуючі елементи автоматично підтримуються через шаблони Visual Studio та інфраструктуру WPF.

Щоб розгадати загадку, яке вікно відкривати, у файлі **App.xaml** за допомогою розмітки визначений клас застосунку. На додаток до визначень просторів імен він визначає властивості застосунку, такі як **StartupUri**, ресурси рівня застосунку (див. «***Лекція 9. Ресурси WPF***») і специфічні обробники для подій застосунку на зразок **Startup** та **Exit**. У **StartupUri** вказане вікно, яке підлягає завантаженню при запуску. Відкриємо файл **App.xaml** і проаналізуємо розмітку в ньому:

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

Використовуючи візуальний конструктор XAML додамо обробники для подій **Startup** та **Exit**. Оновлена розмітка XAML повинна виглядати приблизно так (зміна виділена напівжирним):

```
<Application x:Class="WpfTestApp.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WpfTestApp"
             StartupUri="MainWindow.xaml"
             Startup="App_OnStartup" Exit="App_OnExit">
  <Application.Resources>
    …
  </Application.Resources>
</Application>
```

Вміст файла **App.xaml.cs** має бути схожим на приведений нижче:

```
public partial class App : Application
{
  private void App_OnStartup(object sender, StartupEventArgs e)
  {
    …
  }
  private void App_OnExit(object sender, ExitEventArgs e)
  {
    …
  }
}
```

Зверніть увагу, що клас помічений як *частковий* (**partial**). Насправді всі віконні класи у відокремленому коді для файлів XAML позначаються як часткові. Тут і знаходиться вирішення питання, де знаходиться метод **Main()**. Але спочатку з'ясуємо, що відбувається при обробці файлів XAML утилітою **msbuild.exe**[6], яка за замовчуванням включена до складу Visual Studio.

Відображення XAML-розмітки вікна на код C#

Коли утиліта **msbuild.exe** обробляє файл ***.csproj**, вона створює для кожного файла XAML у проекті три файли:

- ***.g.cs** (де **g** означає **autogenerated** (автоматично згенерований));
- ***.g.i.cs** (де **i** означає **IntelliSense**);
- ***.baml** (для BAML (**Binary Application Markup Language** – двійкова мова розмітки застосувань)).

[6] **MSBuild** (**Microsoft Build Engine**) – платформа для збирання застосунків, яку використовує за замовчуванням Visual Studio для збирання застосунків на основі технології .NET. Правила збирання проекту визначаються у формі XML-схеми, для управління збиранням надається інструментарій для запуску з командного рядка.
Компанія Microsoft надала MSBuild статус відкритого проекту у березні 2015. Сирцевий код відкритий під ліцензією MIT та опублікований на GitHub. Опубліковані сирцеві тексти відповідають MSBuild з кодової бази Visual Studio 2015, яка розширена для відокремленого використання системи збирання, незалежно від Visual Studio.

Ці файли зберігаються в каталозі **\obj\Debug** (їх можна побачити у вікні **Solution Explorer**, клацнувши на кнопці **Show All Files**).

Щоб зробити процес більше осмисленим, елементам управління корисно призначити імена **x:Name**. Призначимо імена елементам управління **Button** і **Calendar**, як показано нижче:

```
<Button x:Name="ClickMe" Content="Button" HorizontalAlignment="Left" VerticalAlignment="Top"
    Width="75" Margin="10,19,0,0" Click="Button_Click"/>
<Calendar x:Name="MyCalendar" HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="10,55,0,0"/>
```

Тепер перебудуємо рішення (або проект) і відновимо файли у вікні **Solution Explorer**. Якщо відкрити файл **MainWindow.g.cs** у текстовому редакторові, то у цьому файлі виявиться клас на ім'я **MainWindow** (див. рис. 2.8) ,який розширює базовий клас **Window**. Ім'я цього класу є прямим результатом дії атрибуту **x:Class** у початковому дескрипторі <**Window**>.

У класі **MainWindow** визначена закрита змінна-член типу **bool** (з ім'ям **_contentLoaded**), яка не була безпосередньо представлена в розмітці XAML. Зазначений член даних використовується для того, щоб визначити (і гарантувати) присвоєння вмісту вікна тільки один раз. Клас також містить змінну-член типу **System.Windows.Controls.Button** на ім'я **ClickMe**. Ім'я елемента управління базується на значенні атрибуту **x:Name** у відкриваючому оголошенні <**Button**>. У класі не наявна змінна для елемента управління **Calendar**. Причина в тому, що утиліта **msbuild.exe** створює змінну для кожного іменованого елемента управління в розмітці XAML, який має пов'язаний код у відокремленому коді. Коли такого коду немає, потреба в змінній відпадає. Якби елементу управління **Button** не призначалося ім'я, то і для нього не було б передбачено змінної. Це частина спроможності WPF, яка пов'язана з реалізацією інтерфейсу **IComponentConnector**.

Згенерований компілятором клас також явно реалізує інтерфейс **IComponentConnector** з WPF, визначений у просторі імен **System.Windows.Markup**. В інтерфейсі **IComponentConnector** наявний єдиний метод **Connect()**, який реалізований для підготовки кожного елемента управління, визначеного в розмітці, і забезпечення логіки подій, як задано у початковому файлі **MainWindow.xaml**. Бачимо, що обробник, налаштований для події клацання на кнопці **ClickMe**. Перед завершенням методу змінна-член **_contentLoaded** встановлюється в **true**. Ось так виглядає цей метод:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target)
{
  switch (connectionId)
  {
    case 1:

    #line 9 "..\..\MainWindow.xaml"
    ((WpfTestApp.MainWindow)(target)).MouseMove +=
      new System.Windows.Input.MouseEventHandler(this.Window_MouseMove);

    #line default
```

```
        #line hidden
        return;
        case 2:
        this.ClickMe = ((System.Windows.Controls.Button)(target));

        #line 18 "..\..\MainWindow.xaml"
        this.ClickMe.Click += new System.Windows.RoutedEventHandler(this.Button_Click);

        #line default
        #line hidden
        return;
        case 3:
        this.MyCalendar = ((System.Windows.Controls.Calendar)(target));
        return;
    }
    this._contentLoaded = true;
}
```

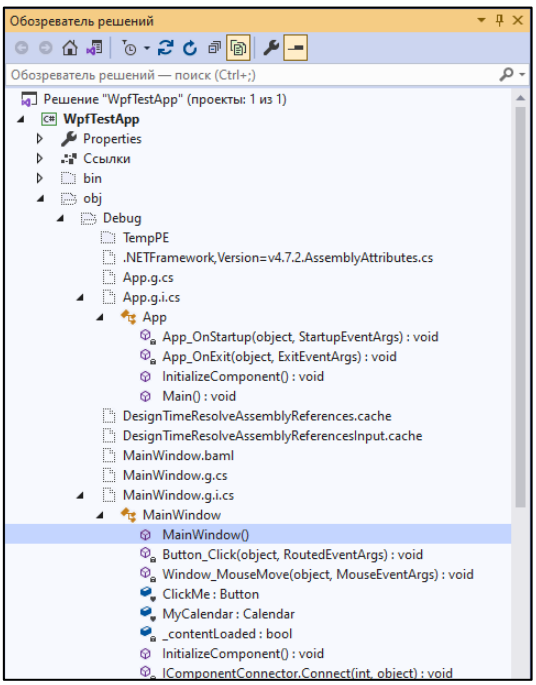


Рис. 2.8. Набір файлів у вікні **Solution Explorer**, які входять до складу рішення **WpfTestApp**.

Щоб продемонструвати вплив неіменованих елементів управління на код, додамо до календаря обробник події **SelectedDatesChanged**. Перебудуємо застосунок, відновимо файли і наново завантажимо файл **MainWindow.g.cs**. Тепер в методі **Connect()** присутній наступний блок коду:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target)
{
    ...
```

```
    #line 20 "..\..\MainWindow.xaml"
    this.MyCalendar.SelectedDatesChanged +=
        new System.EventHandler<System.Windows.Controls.SelectionChangedEventArgs>
        (this.MyCalendar_SelectedDatesChanged);
  ...
}
```

Він повідомляє інфраструктуру про те, що елементу управління в рядку **20** файла XAML призначений обробник події **SelectedDatesChanged**, як показано у коді вище.

Останнє, але не менш важливе: клас **MainWindow** визначає і реалізує метод з іменем **InitializeComponent()**. Можна було б сподіватися, що цей метод містить код, який налаштовує зовнішній вигляд і поведінку кожного елемента управління, встановлюючи його різноманітні властивості (**Height**, **Width**, **Content** і т. д.). Проте це не зовсім так. Як тоді елементи управління отримують коректний інтерфейс користувача? Логіка в методі **InitializeComponent()** з'ясовує місце розташування вбудованого у збірку ресурсу, з ім'ям ідентичним початковому файлу ***.xaml**:

```
public void InitializeComponent()
{
  if (_contentLoaded)
  {
    return;
  }
  _contentLoaded = true;
  System.Uri resourceLocater = new System.Uri("/WpfTestApp;component/mainwindow.xaml",
    System.UriKind.Relative);

  #line 1 "..\..\MainWindow.xaml"
  System.Windows.Application.LoadComponent(this, resourceLocater);

  #line default
  #line hidden
}
```

Тут виникає питання: що є цим вбудованим ресурсом?

Роль BAML

Формат BAML – це компактне двійкове представлення початкових даних XAML. Файл ***.baml** вбудовується у вигляді ресурсу (через файл ***.g.resources**) у скомпільовану збірку[7]. Ресурс BAML містить всі дані, потрібні для

[7] **Збірка** (assembly) – це логічна одиниця, яка містить скомпільований код для .NET Framework. Збірка – це повністю самодостатній і, швидше, логічний, ніж фізичний елемент. Це означає, що він може бути збережений в більш ніж одному файлі (хоча динамічні збірки зберігаються в пам'яті, а зовсім не у файлах). Якщо збірка зберігається більше ніж одному файлі, то має бути один головний файл, який містить точку входу та описує інші файли.

налаштування зовнішнього вигляду і поведінки віджетів[8] інтерфейсу користувача (властивості на зразок **Height** і **Width**).

Тут важливо розуміти, що застосунок WPF містить всередині себе двійкове представлення (BAML) розмітки. Під час виконання ресурс BAML добувається з контейнера ресурсів і використовується для налаштування зовнішнього вигляду і поведінки всіх вікон та елементів управління.

На додаток варто пам'ятати, що імена таких двійкових ресурсів ідентичні іменам написаних автономних файлів ***.xaml**. Проте, звідси зовсім не випливає потреба у поширенні файлів ***.xaml** разом із скомпільованою програмою WPF. Якщо тільки не будується застосунок WPF, який повинен динамічно завантажувати та аналізувати файли ***.xaml** під час виконання, то постачати початкову розмітку ніколи не доведеться.

Розгадування загадки Main()

Тепер, коли відомо, як працює процес **msbuild.exe**, відкриємо файл **App.g.cs**. У ньому знаходиться автоматично згенерований метод **Main()**, який ініціалізує і запускає наш об'єкт застосунку.

```
public static void Main()
{
  WpfTestApp.App app = new WpfTestApp.App();
  app.InitializeComponent();
  app.Run();
}
```

Метод **InitializeComponent()** конфігурує властивості застосунку, включаючи **StartupUri** та обробники подій **Startup** та **Exit**.

```
public void InitializeComponent()
{
  #line 6 "..\..\App.xaml"
  this.Startup += new System.Windows.StartupEventHandler(this.App_OnStartup);
  #line default
  #line hidden
  #line 6 "..\..\App.xaml"
  this.Exit += new System.Windows.ExitEventHandler(this.App_OnExit);
  #line default
  #line hidden
  #line 5 "..\..\App.xaml"
  this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
  #line default
  #line hidden
}
```

[8] **Віджет** – це невеликий додаток для виконання простої дії або для показу якоїсь інформації (годинник/календар на робочому столі, дозволяє швидко отримати інформацію про погоду і т. д.). Часто віджет це не сам додаток, а графічний або текстовий елемент, який відкриває доступ до нього або запускає роботу додатка. Віджети сильно спрощують користування складними електронними пристроями. Зовсім не важко здогадатися, що натиснувши на значок із зображенням, наприклад, сонечка і хмаринки, отримаємо інформацію про погоду, і т. п.

Взаємодія з даними рівня застосунку

В класі **Application**[9] (файл **App.xaml**) є властивість **Properties** (див. рис. 2.9), яка дозволяє визначити колекцію пар "*ім'я/значення*" через індексатор типу. Він призначений для операції на типі **System.Object**, тому в колекцію можна зберігати елементи будь-якого вигляду (у тому числі екземпляри спеціальних класів) з метою подальшого отримання за відповідним іменем. Використовуючи такий підхід, легко розділяти дані між всіма вікнами в застосунку WPF.

В цілях ілюстрації ми відновимо поточний обробник події **Startup**, щоб він перевіряв вхідні аргументи командного рядка на присутність значення **/GODMODE** (поширений шахрайський код у багатьох іграх). Якщо воно знайдене, тоді значення **bool** на ім'я **GodMode** всередині колекції властивостей встановлюється в **true** (інакше воно встановлюється в **false**).

Звучить досить просто, але як передати обробникові події **Startup** вхідні аргументи командного рядка (зазвичай отримувані методом **Main()**)? Один з підходів передбачає виклик статичного методу **Environment.GetCommandLineArgs()**. Проте ті ж самі аргументи автоматично додаються у вхідний параметр **StartupEventArgs** і доступні через властивість **Args**. Отже, приведемо першу модифікацію поточної кодової бази:

```
private void App_OnStartup(object sender, StartupEventArgs e)
{
  Application.Current.Properties ["GodMode"] = false;
  // Перевірити вхідні аргументи командного рядка
  // на предмет наявності прапорця /GODMODE.
  foreach (string arg in e.Args)
  {
    if (arg.Equals("/godmode", StringComparison.OrdinalIgnoreCase))
    {
      Application.Current.Properties["GodMode"] = true;
      break;
    }
  }
}
```

Дані застосунку доступні з будь-якого місця всередині застосунку WPF. Для звернення до них треба лише отримати точку доступу до глобального об'єкта застосунку (через **Application.Current**) і переглянути колекцію. Наприклад, обробник події **Click** для кнопки можна було б змінити так:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  // Чи вказав користувач /godmode?
  if ((bool)Application.Current.Properties["GodMode"])
  {
    MessageBox.Show("Cheater!");
    // Шахрай!
  }
}
```

[9] https://docs.microsoft.com/en-us/dotnet/api/system.windows.application?view=windowsdesktop-6.0

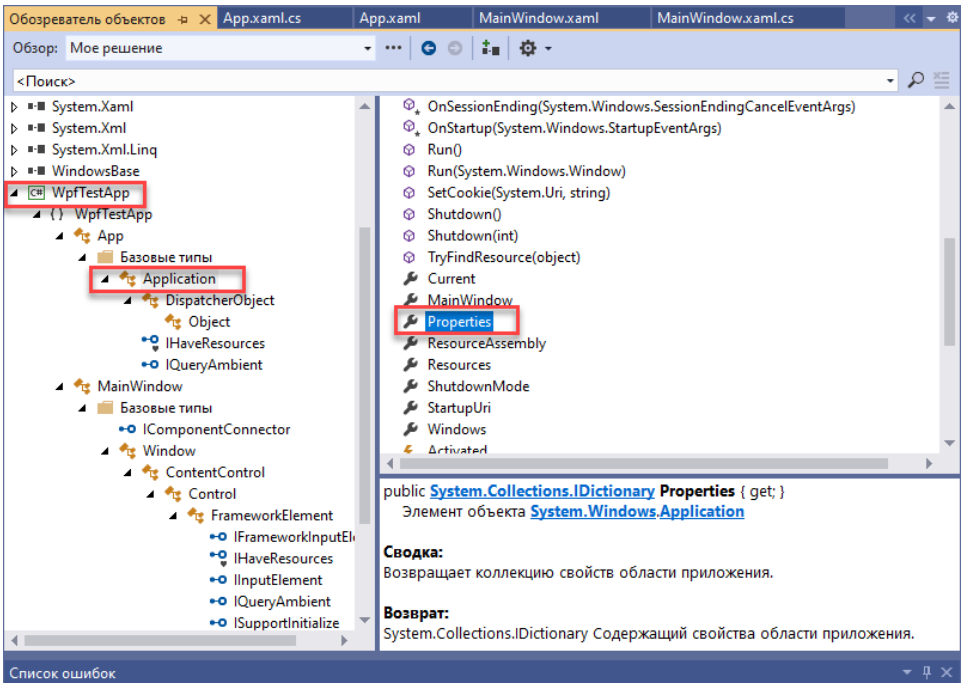


Рис. 2.9. Вікно **Оглядача об'єктів** з класом **Application** та властивістю **Properties**

Якщо тепер ввести аргумент командного рядка **/godmode** на вкладці **Debug** (Відлагодження) у вікні властивостей проекту і запустити програму, то відобразиться вікно повідомлення і програма завершиться. Можна також запустити програму з командного рядка за допомогою показаної нижче команди (заздалегідь відкривши вікно командного рядка і перейшовши в каталог **bin/debug**):

```
WpfTestApp.exe /godmode
```

Відобразиться вікно повідомлення, а програма завершиться.

Обробка закриття об'єкта Window

Кінцеві користувачі можуть завершувати роботу вікна за допомогою численних вбудованих прийомів рівня системи (наприклад, клацнувши на кнопці закриття × всередині рамки вікна) або викликавши метод **Close()** у відповідь на деяку дію з інтерактивним елементом (скажімо, вибір пункту меню **File|Exit** (Файл|Вихід)). Інфраструктура WPF пропонує дві події, які можна перехоплювати для з'ясування, чи справді користувач має намір закрити вікно і видалити його з пам'яті. Перша така подія – **Closing**, яка працює у поєднанні з *делегатом* **CancelEventHandler**.

Цей делегат чекає цільові методи, які приймають тип **System.ComponentModel.CancelEventArgs** у другому параметрі. Клас **CancelEventArgs** надає властивість **Cancel**, встановлення якої в **true** запобігає

фактичному закриттю вікна (що зручно, коли користувачеві має бути поставлене питання про те, чи він справді бажає закрити вікно, чи спочатку треба зберегти виконану роботу).

Якщо користувач дійсно хоче закрити вікно, тоді властивість **CancelEventArgs.Cancel** можна встановити у **false** (стандартне значення). У результаті згенерується подія **Closed** (яка працює з делегатом **System.EventHandler**), що є точкою, де вікно повністю і безповоротно готове до закриття.

Модифікуємо клас **MainWindow.xaml.cs** для обробки згаданих двох подій, додавши у поточний код конструктора такі оператори:

```
public MainWindow()
{
  InitializeComponent();
  this.Closed += Window_Closed;
  this.Closing += Window_Closing;
}
```

Тепер реалізуємо відповідні обробники подій:

```
private void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
  // З'ясувати, чи справді користувач хоче закрити вікно.
  string msg = "Do you want to close without saving?";
  MessageBoxResult result = MessageBox.Show(msg,
    "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
  if (result == MessageBoxResult.No)
  {
    // Якщо користувач не хоче закривати вікно, то відмініть закриття.
    e.Cancel = true;
  }
}
private void Window_Closed(object sender, EventArgs e)
{
  MessageBox.Show("See ya!");
}
```

Запустимо програму і спробуємо закрити вікно, клацнувши або на значку × у правому верхньому куті вікна, або на кнопці. Повинне з'явитися діалогове вікно із запитом підтвердження. Клацання на кнопці **Yes** (Так) приведе до відображення вікна з прощальним повідомленням, а клацання на кнопці **No** (Ні) залишить вікно в пам'яті.

Перехоплення подій миші

Інфраструктура WPF надає декілька подій, які можна перехоплювати, щоб взаємодіяти з мишею. Зокрема, базовий клас **UIElement** визначає такі пов'язані з мишею події, як **MouseMove**, **MouseUp**, **MouseDown**, **MouseEnter**, **MouseLeave** і т. д. (див. рис. 2.10).

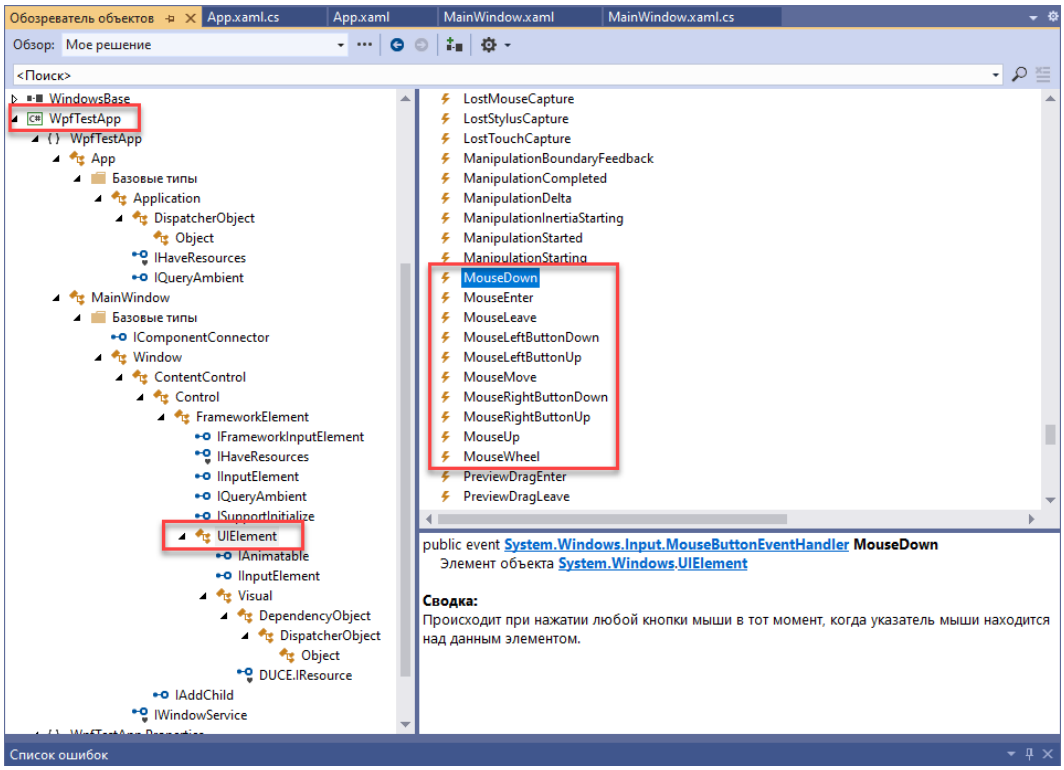


Рис. 2.10. Демонстрація наявності в класі **UIElement** «мишиних» подій.

Як приклад обробимо подію **MouseMove**. Ця подія працює у поєднанні з делегатом **System.Windows.Input.MouseEventHandler**, який чекає, що його цільовий метод прийматиме в другому параметрі об'єкт типу **System.Windows.Input.MouseEventArgs**. Використавши клас **MouseEventArgs** можна отримувати позицію (**x, y**] курсора миші та інші важливі деталі.

Погляньте на наступне часткове визначення:

```
public class MouseEventArgs : InputEventArgs
{
  …
  public Point GetPosition(IInputElement relativeTo);
  public MouseButtonState LeftButton { get; }
  public MouseButtonState MiddleButton { get; }
  public MouseDevice MouseDevice { get; }
  public MouseButtonState RightButton { get; }
  public StylusDevice StylusDevice { get; }
  public MouseButtonState XButton1 { get; }
  public MouseButtonState XButton2 { get; }
}
```

На замітку! Властивості **XButton1** і **XButton2** дозволяють взаємодіяти з «розширеними кнопками миші» (на зразок кнопок «вперед» і «назад», які є в деяких пристроях). Вони часто використовуються для взаємодії з

хронологією навігації браузера, щоб переміщатися між відвіданими сторінками.

Метод **GetPosition()** дозволяє отримати значення (**x, y**) відносно елемента інтерфейсу користувача у вікні. Якщо цікавить позиція відносно активного вікна, то треба просто передати **this**. Забезпечимо обробку події **MouseMove** у конструкторі класу **MainWindow**:

```
public MainWindow()
{
   InitializeComponent();
   this.Closed += Window_Closed;
   this.Closing += Window_Closing;
   this.MouseMove += Window_MouseMove;
}
```

Нижче приведений обробник події **MouseMove**, який відобразить місце розташування курсора миші в області заголовка вікна (зверніть увагу, що повернений тип **Point** транслюється в рядкове значення за допомогою виклику **ToString()**):

```
private void Window_MouseMove(object sender, System.Windows.Input.MouseEventArgs e)
{
   // Відобразити у заголовку вікна поточну позицію (x, y) курсора миші.
   this.Title = e.GetPosition(this).ToString();
}
```

Перехоплення подій клавіатури

Обробка клавіатурного вводу для вікна, на якому знаходиться фокус, також не складна. У класі **UIElement** визначені декілька подій, які можна перехоплювати для відстежування натиснень клавіш клавіатури на активному елементі (наприклад, **KeyUp** і **KeyDown**). Події **KeyUp** і **KeyDown** працюють з делегатом **System.Windows.Input.KeyEventHandler**, який чекає в другому параметрі тип **KeyEventArgs**, який визначає набір важливих відкритих властивостей:

```
public class KeyEventArgs : KeyboardEventArgs
{
   public bool IsDown { get; }
   public bool IsRepeat { get; }
   public bool IsToggled { get; }
   public bool IsUp { get; }
   public Key Key { get; }
   public KeyStates KeyStates { get; }
   public Key SystemKey { get; }
}
```

Щоб проілюструвати організацію обробки події **KeyDown** в конструкторі **MainWindow** (як робилося для попередніх подій), ми реалізуємо наступний

обробник події, який змінює вміст кнопки на інформацію про поточну натиснуту клавішу:

```
private void Window_KeyDown(object sender, System.Windows.Input.KeyEventArgs e)
{
  // Відобразити на кнопці інформацію про натиснуту клавішу.
  ClickMe.Content = e.Key.ToString();
}
```

До цього моменту WPF може здатися всього лише черговою інфраструктурою для побудови графічних інтерфейсів користувача, яка пропонує (у більшій або меншій мірі) ті ж самі служби, що і Windows Forms, MFC або VB6. Якби це було саме так, тоді виникло б питання про сенс у потребі ще одного інструментального набору, орієнтованого на створення інтерфейсів користувача. Щоб реально оцінити унікальність WPF, потрібно буде опанувати граматику XAML, яка базується на XML.

Вивчення документації WPF

На завершення лекції треба зазначити, що тематиці WPF у документації .NET Framework SDK присвячений цілий розділ. У міру дослідження API-інтерфейсу WPF і читання інших лекцій, які розкривають інфраструктуру WPF, ви виявите наполегливу потребу у зверненні до довідкової системи. Там ви знайдете багато прикладів розмітки XAML і детальне повчальне керівництво з широкого спектру тем, починаючи з програмування тривимірної графіки і закінчуючи складними операціями прив'язки даних.

Документація WPF доступна через меню **Docs|.NET|.NET Framework|Windows Presentation Foundation** (вона знаходиться за адресою https://docs.microsoft.com/en-us/dotnet/desktop/wpf/?view=netdesktop-6.0.

Резюме

Мова XAML дозволяє описувати дерева об'єктів .NET з використанням декларативного синтаксису. Під час дослідження XAML у цій лекції ви довідалися про декілька нових фрагментів синтаксису, включаючи синтаксис "*властивість-елемент*" і *приєднувані властивості*, а також про роль *перетворювачів типів* і *розширень розмітки* XAML.

Розмітка XAML є ключовим аспектом будь-якого застосунку WPF виробничого рівня. Використовуючи як приклад рішення **WpfTestApp** було побудовано застосунок WPF, який продемонстрував багато концепцій, обговорених в лекції. У наступних лекціях ці і багато інших концепцій розглядатимуться детальніше.

Контрольні питання.

1. Які основні характеристики середовища розробки застосунків Microsoft Visual Studio?
2. Опишіть схему створення шаблону проекту?
3. Охарактеризуйте основні складові компоненти початкового вікна Microsoft Visual Studio відразу після створення проекту?

4. Для чого потрібна панель інструментів?
5. Для чого потрібне вікно Properties?
6. Які функції візуального конструктора?
7. Як реалізувати створення обробника подій, використовуючи вікно Properties?
8. Як обробляти події безпосередньо в редакторі XAML?
9. Для чого потрібне вікно Document Outline?\
10. Дослідивши файл App.xaml, обґрунтуйте його обов'язкову наявність у складі проекту.
11. Що ви розумієте під поняттям відображення XAML-розмітки вікна на код C#?
12. Яке функціональне призначення вікна Solution Explorer?
13. Як працює метод Main()?
14. Як можна використовувати Object Browser?
15. Розкажіть про нюанси закриття об'єкта Window?
16. Які події пов'язані з мишкою?
17. Яке відношення до подій пов'язаних з мишею має клас UIElement?
18. Який механізм перехоплення подій від клавіатури?
19. Яке відношення до подій пов'язаних з клавіатурою має клас UIElement?

Лекція 3. Елементи управління у WPF

У ***Лекція 1. Вступ у Windows Presentation Foundation і XAML*** та ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio*** була представлена основа моделі програмування WPF, включаючи класи **Window** та **Application**, граматику **XAML** і використання файлів коду. Окрім цього в них зроблено вступ в процес побудови застосунків WPF з використанням візуальних конструкторів (**Integrated Development Environment** – інтегроване середовище розробки) **IDE**-середовища Visual Studio. У цій лекції ми заглибимося в конструювання складніших графічних інтерфейсів користувача з використанням декількох нових елементів управління і диспетчерів компонування.

Огляд основних елементів управління WPF

Опишемо основні елементи управління WPF (табл. 3.1).

Таблиця 3.1. Основні елементи управління WPF

Категорія елементів управління WPF	Приклади членів	Опис
Основні елементи управління для вводу користувача	Button, RadioButton, ComboBox, CheckBox, Calendar, DatePicker, Expander, DataGrid, ListBox, ListView, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBlock, TextBox, RepeatButton, RichTextBox, Label	Інфраструктура WPF пропонує повне сімейство елементів управління, які можна задіяти при побудові інтерфейсів користувача.
Бічні елементи вікон та елементів управління	Menu, ToolBar, StatusBar, ToolTip, ProgressBar	Ці елементи інтерфейсу користувача служать для декорування рамки об'єкта **Window** компонентами для вводу (на кшталт **Menu**) та елементами інформування користувача (скажімо, **StatusBar** і **ToolTip**).
Елементи управління мультимедіа	Image, MediaElement, SoundPlayerAction	Ці елементи управління надають підтримку відтворення аудіо/відео і виводу зображень.
Елементи управління компонуванням	Border, Canvas, DockPanel, Grid, GridView, GridSplitter, GroupBox, Panel, TabControl, StackPanel, Viewbox, WrapPanel	Інфраструктура WPF пропонує багато елементів управління, які дозволяють групувати та організовувати інші елементи з метою управління компонуванням.

Елементи управління Ink API

На додаток до елементів управління WPF (табл. 3.1), в інфраструктурі WPF є елементи управління для роботи з інтерфейсом **Ink API**. Вони корисні при побудові застосунків для **Tablet PC**, бо дозволяє реалізовувати ввід від пера. Але, це не означає, що стандартний настільний застосунок не може задіяти **Ink API**, бо ті ж самі елементи управління можуть працювати з вводом від миші.

Простір імен **System.Windows.Ink** зі збірки **PresentationCore.dll** містить різноманітні підтримуючі типи **Ink API** (наприклад, **Stroke** і **StrokeCollection**).

Проте більшість елементів управління **Ink API** (на зразок **InkCanvas** і **InkPresenter**) запаковані разом із загальними елементами управління WPF у просторі імен **System.Windows.Controls** всередині збірки **PresentationFramework.dll**. Ми будемо працювати з інтерфейсом **Ink API** про який детальніше буде розказано в ***Лекції 05. Події у WPF***.

Елементи управління документів WPF

На додаток інфраструктура WPF надає елементи управління для розширеної обробки документів, дозволяючи будувати застосунки, які включають функціональність у стилі **Adobe PDF**. Із використанням типів з простору імен **System.Windows.Documents** (який також знаходиться у збірці **PresentationFramework.dll**) можна створювати готові до друку документи, які підтримують масштабування, пошук, анотації користувача («липкі» замітки[10]) та інші розвинені засоби роботи з текстом.

Проте, насправді елементи управління документів не використовують API-інтерфейси Adobe PDF, а натомість працюють з API-інтерфейсом **XML Paper Specification** (**XPS**). Кінцеві користувачі ніякої різниці не помітять, тому що документи PDF і XPS мають практично ідентичний вигляд і поведінку. Насправді у доступі є багато безкоштовних утиліт, які дозволяють виконувати перетворення між вказаними двома файловими форматами на льоту. Ми такі елементи управління не розглядаємо.

Загальні діалогові вікна WPF

Інфраструктура WPF також пропонує декілька загальних діалогових вікон, таких як **OpenFileDialog** і **SaveFileDialog**, які визначені в просторі імен **Microsoft.Win32** всередині збірки **PresentationFramework.dll** (див. рис. 3.1).

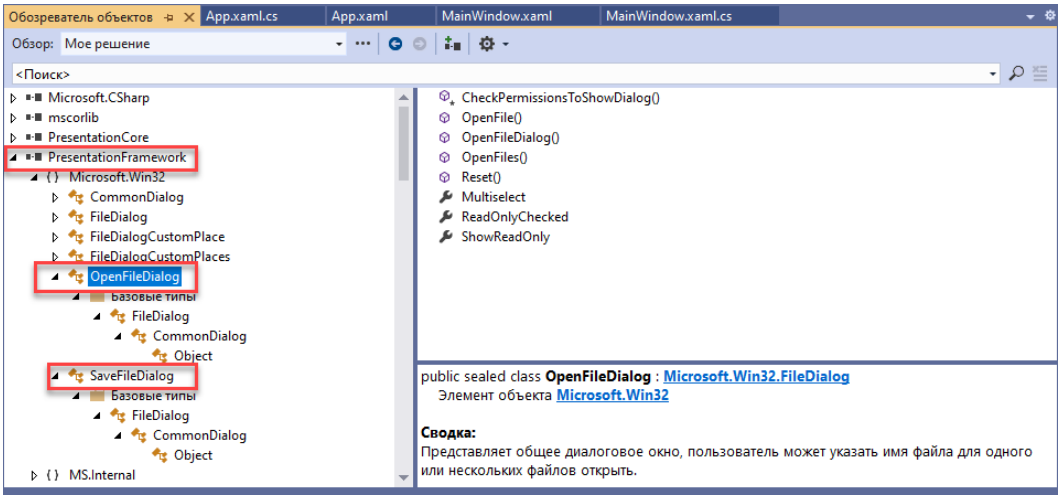


Рис. 3.1. Збірка **PresentationFramework.dll** в **Оглядачі об'єктів**

[10] **Sticky Notes** (Липкі замітки, Записки, Стікери) – застосунок для створення заміток на Робочому столі, вбудований в ОС Windows 10.

Робота з будь-яким з вказаних діалогових вікон зводиться до створення об'єкта і виклику методу **ShowDialog()**:

```
using Microsoft.Win32;
// Для скорочення код не показаний.
private void btnShowDlg_Click(object sender, RoutedEventArgs e)
{
    // Відобразити діалогове вікно збереження файла.
    SaveFileDialog saveDlg = new SaveFileDialog();
    saveDlg.ShowDialog();
}
```

В цих класах підтримуються різноманітні члени, які дозволяють встановлювати фільтри файлів і шляху до каталогів, та отримувати доступ до файлів. Деякі діалогові вікна використовуються у прикладах далі; крім того покажемо, як будувати діалогові вікна для отримання вводу користувача.

Детальні відомості в документації

Мета лекції не в описі кожного члена всіх елементів WPF (рис. 3.2).

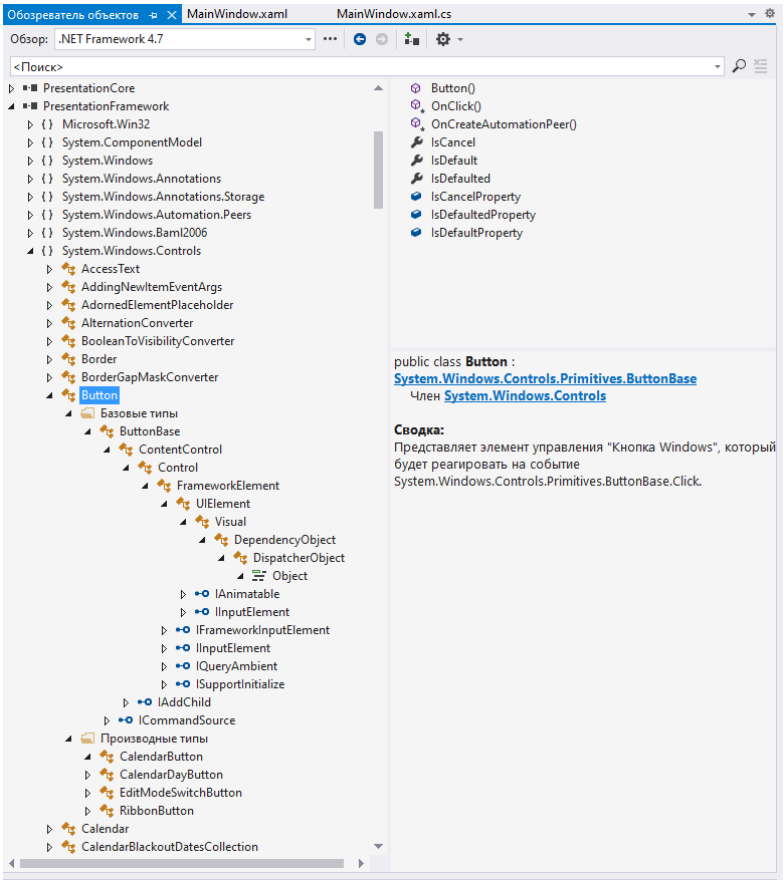


Рис. 3.2. Стандартні елементи управління WPF запаковані в простір імен

System.Windows.Controls всередині збірки **PresentationFramework.dll**.

Натомість буде здійснений огляд різних елементів управління з акцентом на програмній моделі, яка лежить в основі, і ключових службах, загальних для більшості елементів управління WPF.

Повне уявлення про конкретну функціональність заданого елемента управління дає документація, зокрема розділ «**Control Library**», довідкової системи доступної за адресою https://docs.microsoft.com/ru-ru/dotnet/framework/wpf/controls/index.,

Тут ви знайдете вичерпні описи кожного елемента управління, різноманітні приклади коду (на XAML і C#), а також інформацію про ланцюжок спадкоємства, реалізованих інтерфейсах і використовуваних атрибутах для будь-якого елемента управління. Обов'язково приділіть час на дослідження елементів управління, які розглядаються у цій лекції.

Короткий огляд візуального конструктора WPF у Visual Studio

Більшість стандартних елементів управління WPF запакована в простір імен **System.Windows.Controls** в збірці **PresentationFramework.dll** (рис. 3.2). При побудові застосунку WPF у Visual Studio багато загальних елементів управління знаходиться в панелі інструментів за умови, що активним вікном є візуальний конструктор WPF (див. ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio***, рис. 2.2).

При побудові інтерфейсів користувача у Visual Studio стандартні елементи управління можна перетягувати на поверхню візуального конструктора WPF і конфігурувати їх у вікні **Properties** (Властивості), як було показано в ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio***. Хоча Visual Studio згенерує пристойний обсяг XAML автоматично, немає нічого незвичайного в тому, щоб потім редагувати розмітку вручну. Давайте розглянемо основи.

Робота з елементами управління WPF в Visual Studio

Ви можете згадати з ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio***, що після поміщення елемента управління WPF на поверхню візуального конструктора Visual Studio у вікні **Properties** (або прямо в розмітці XAML) потрібно встановити властивість **x:Name**, бо це дозволяє звертатися до об'єкта у пов'язаному файлі коду C#. Крім того, на вкладці **Events** (Події) вікна **Properties** можна генерувати обробники подій для вибраного елемента управління. Отже, за допомогою Visual Studio можна було б згенерувати наступну розмітку для простого елемента управління **Button**:

```
<Button x:Name="btnMyButton" Content="Click Me!" Height="23" Width="140"
Click="btnMyButton_Click" />
```

Тут властивість **Content** елемента **Button** задається звичайним рядком "Click Me!". Проте завдяки *моделі вмісту* елементів управління WPF можна створити елемент **Button** з наступним складним вмістом:

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <Button.Content>
    <StackPanel Height="95" Width="128" Orientation="Vertical">
      <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
      <Label Width="59" FontSize="20" Content="Click!" Height="36" />
    </StackPanel>
  </Button.Content>
</Button>
```

Ви можете також згадати, що безпосереднім дочірнім елементом похідного від **ContentControl** класу є запропонований вміст, а тому при заданні складного вмісту визначати область **Button.Content** явно не вимагається. Можна було б написати таку розмітку:

```
<Button x:Name="btnMyButton" Height="121" Width="156" Click="btnMyButton_Click">
  <StackPanel Height="95" Width="128" Orientation="Vertical">
    <Ellipse Fill="Red" Width="52" Height="45" Margin="5"/>
    <Label Width="59" FontSize="20" Content="Click!" Height="36" />
  </StackPanel>
</Button>
```

Тут властивість **Content** кнопки встановлюється в елемент **StackPanel** з пов'язаними елементами. Створювати складний вміст такого роду можна також із застосуванням візуального конструктора Visual Studio. Після визначення диспетчера компонування для елемента управління вмістом його можна вибирати у візуальному конструкторі як цільовий компонент, на який перетягуватимуться внутрішні елементи управління. Кожен з них можна редагувати у вікні **Properties**. Якщо вікно **Properties** використовувалося для обробки події **Click** елемента управління **Button** (як було показано в попередніх оголошеннях XAML), то IDE-середовище згенерує порожній обробник події, куди можна буде додати спеціальний код, наприклад:

```
private void btnMyButton_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("You clicked the button!");
  // Ви клацнули на кнопці!
}
```

Робота з вікном Document Outline

В ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio*** ви довідалися, що вікно **Document Outline** (Схема документа) у Visual Studio (яке відкривається через меню **View|Other Windows**) зручне при проектуванні елемента управління WPF зі складним вмістом. Для створюваного елемента **Window** відображається логічне дерево XAML, а клацання на будь-якому вузлі в дереві спричиняє його автоматичний вибір у візуальному конструкторі і в редакторі XAML для редагування.

У цій версії Visual Studio вікно **Document Outline** має декілька додаткових засобів, які будуть для вас надзвичайно корисними. Праворуч від будь-якого вузла знаходиться значок, який нагадує очне яблуко. Клацання на

ньому дозволяє приховувати або відображати елемент у візуальному конструкторі, що буває зручним, коли потрібно зосередити увагу на окремому сегменті, який підлягає редагуванню (треба відмітити, що елемент буде прихований тільки на поверхні візуального конструктора, але не під час виконання).

Праворуч від значка з очним яблуком є ще один значок, який дозволяє блокувати елемент у візуальному конструкторі. Як і можна було здогадатися, це зручно, коли треба перешкодити випадковій зміні розмітки XAML для заданого елемента вами або колегами з розробки. Насправді блокування елемента переводить його в режим тільки читання на етапі проектування (що цілком очевидно не заважає змінювати стан об'єкта під час виконання).

Управління компонуванням вмісту з використанням панелей

Застосунок WPF містить певне число елементів інтерфейсу користувача (наприклад, елементів вводу, графічного вмісту, систем меню і рядків стану), які мають бути добре організовані всередині різноманітних вікон. Після розміщення елементів інтерфейсу користувача потрібно гарантувати їх передбачувану поведінку, коли кінцевий користувач змінює розмір вікна або його частини (як у разі вікна з роздільником). Щоб забезпечити збереження елементами управління WPF своїх позицій всередині вікна, в якому вони знаходяться, можна задіяти багато *типів панелей* (також відомих як *диспетчери компонування*).

За замовчуванням новий елемент **Window**, використовуватиме диспетчер компонування типу **Grid**. Проте, поки що розглянемо елемент **Window** без будь-яких оголошених диспетчерів компонування:

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Fun with Panels!" Height="350" Width="525">
</Window>
```

Коли елемент управління оголошується прямо всередині вікна, в якому панелі не використовуються, він позиціонується по центру контейнера. Розглянемо показане далі просте оголошення вікна, яке містить єдиний елемент управління **Button**. Незалежно від того, як змінюються розміри вікна, цей віджет інтерфейсу користувача завжди знаходитиметься на рівному віддаленні від усіх чотирьох меж клієнтської області. Розмір елемента **Button** визначається встановленими значеннями його властивостей **Height** і **Width**.

```
<!-- Ця кнопка завжди знаходиться в центрі вікна -->
<Window x:Class=" WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Дослідження панелей!" Height="285" Width="325">
  <Button x:Name="btnOK" Height = "100"
      Width="80" Content="OK"/>
</Window>
```

Запам'ятайте: Спроба поміщення всередину області **Window** декількох елементів спричинить помилки розмітки і компіляції. Причина в тому, що властивості **Content** вікна (або по суті будь-якого нащадка **ContentControl**) може бути присвоєний тільки один об'єкт. Наступна розмітка XAML помилкова:

```
<!-- Помилка! Властивість Content неявно встановлюється більше одного разу! -->
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title=" Дослідження панелей!" Height="285" Width="325">
  <!-- Помилка! Два безпосередніх дочірніх елементи у <Window> -->
  <Label x:Name="lblInstructions" Width="328" Height="25"
      FontSize="15" Content="Enter Information"/>
  <Button x:Name="btnOK" Height = "100" Width="80" Content="OK"/>
</Window>
```

Зрозуміло, що від вікна, тільки з одним елементом управління, мало користі. Коли вікно повинне містити декілька елементів, їх потрібно буде розташувати всередині панелей. У панель будуть поміщені всі елементи інтерфейсу користувача, які представляють вікно, після чого сама панель виступає єдиним об'єктом, який присвоюється властивості **Content** вікна.

Простір імен **System.Windows.Controls** (див. рис. 3.2) пропонує панелі, кожна з яких по-своєму обслуговує внутрішні елементи. За допомогою панелей можна встановлювати поведінку елементів управління при зміні розмірів вікна користувачем – будуть вони залишатися в тих же місцях, де були розміщені на етапі проектування, розташовуватися вільним потоком зліва направо або згори вниз і т. д.

Елементи управління типу панелей також дозволено поміщати всередину інших панелей (наприклад, елемент управління **DockPanel** може містити **StackPanel** зі своїми елементами), щоб забезпечити більшу гнучкість і ступінь управління.

У табл. 3.2 коротко описані деякі поширені елементи управління типу панелей WPF.

Таблиця 3.2. Основні елементи управління типу панелей WPF

Елемент управління типу панелі	Опис
Canvas	Надає класичний режим розміщення вмісту. Елементи залишаються точнісінько там же, куди були поміщені на етапі проектування.
DockPanel	Прив'язує вміст до заданої сторони панелі (**Top** (верхня), **Bottom** (нижня), **Left** (ліва) або **Right** (права)).
Grid	Розташовує вміст всередині серії комірок, створених всередині табличної сітки.
StackPanel	Розміщає вміст вертикально або горизонтально, згідно заданої властивості **Orientation**.
WrapPanel	Позиціонує вміст зліва направо, переносячи його на наступний рядок після досягнення межі панелі. Подальше впорядкування відбувається послідовно зверху вниз або зліва направо залежно від значення властивості **Orientation**

Позиціонування вмісту всередині панелей Canvas

Панель **Canvas** позиціонує вміст інтерфейсу користувача абсолютно. При зменшенні розміру вікна до розміру меншого ніж обслуговується панеллю компонування **Canvas**, внутрішній вміст буде невидимим доти, поки контейнер не збільшать до розміру, який дорівнює або перевищує розмір області **Canvas**.

Щоб додати вміст до **Canvas**, спочатку потрібно:

1. Визначити потрібні елементи управління всередині області між відкриваючим і закриваючим дескрипторами **Canvas**.
2. Потім для кожного елемента потрібно задати лівий верхній кут, використовуючи властивості **Canvas.Top** і **Canvas.Left**.
3. Нижній правий кут кожного елемента управління можна задати:
   - *неявно*, встановлюючи властивості **Canvas.Height** і **Canvas.Width**;
   - *явно* із використанням властивостей **Canvas.Right** і **Canvas.Bottom**.

Продемонструємо **Canvas** у дії створивши рішення **SimpleCanvas**:

```
<Window x:Class="SimpleCanvas.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:SimpleCanvas"
        mc:Ignorable="d"
        Title="Дослідження панелей!" Height="285" Width="325">
  <Canvas Background="LightSteelBlue">
    <Button x:Name="btnOK" Canvas.Left="212" Canvas.Top="203" Width="80" Content="OK"/>
    <Label x:Name="lblInstructions" Canvas.Left="17" Canvas.Top="14" Width="328" Height="27"
        FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Canvas.Left="17" Canvas.Top="60" Content="Make"/>
    <TextBox x:Name="txtMake" Canvas.Left="94" Canvas.Top="60" Width="193" Height="25"/>
    <Label x:Name="lblColor" Canvas.Left="17" Canvas.Top="109" Content="Color"/>
    <TextBox x:Name="txtColor" Canvas.Left="94" Canvas.Top="107" Width="193" Height="25"/>
    <Label x:Name="lblPetName" Canvas.Left="17" Canvas.Top="155" Content="Pet Name"/>
    <TextBox x:Name="txtPetName" Canvas.Left="94" Canvas.Top="153" Width="193" Height="25"/>
  </Canvas>
</Window>
```

У верхній половині екрана відобразиться вікно, показане на рис. 3.3.

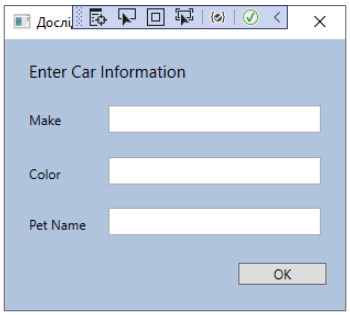


Рис. 3.3. Диспетчер **Canvas** здійснює абсолютне позиціонування вмісту

Зауважимо, що порядок оголошення елементів вмісту у **Canvas** не впливає на розрахунок місця розташування; насправді місце розташування базується на розмірі елемента управління і значеннях його властивостей **Canvas.Top**, **Canvas.Bottom**, **Canvas.Left** і **Canvas.Right**.

> **На замітку!** Якщо піделементи у **Canvas** не визначають специфічне місце розташування з використанням синтаксису *приєднуваних властивостей* (наприклад, **Canvas.Left** і **Canvas.Top**), тоді вони автоматично прикріпляються до верхнього лівого кута **Canvas**.

Використання типу **Canvas** може здатися переважним способом організації вмісту (бо він для вас виглядає звично), але цьому підходу властиві деякі обмеження.

1. По-перше, елементи усередині **Canvas** не змінюють свої розміри динамічно при використанні стилів або шаблонів (скажемо, їх шрифти залишаються недоторканими).
2. По-друге, панель **Canvas** не намагається зберігати елементи видимими, коли кінцевий користувач зменшує розмір вікна.

Мабуть, найкращим використанням типу **Canvas** є позиціонування графічного вмісту. Наприклад, при побудові зображення з використанням XAML безперечно вимагається зробити так, щоб всі лінії, фігури і текст залишалися на своїх місцях, а не динамічно переміщалися у разі зміни користувачем розміру вікна. Ми ще повернемося до **Canvas** в ***Лекція 7. Служби візуалізації графіки WPF*** при обговоренні служб візуалізації графіки WPF.

Позиціонування вмісту всередині панелей WrapPanel

Панель **WrapPanel** дозволяє визначати вміст, який буде безперервно продовжуватися в панелі, коли розмір вікна змінюється. При позиціонуванні елементів всередині **WrapPanel** їх координати верхнього лівого і нижнього правого кутів не задаються, як це зазвичай робиться в **Canvas**. Проте для кожного піделемента допускається визначення значень властивостей **Height** і **Width** (разом з іншими властивостями), щоб управляти їх сумарним розміром в контейнері.

Через те, що вміст всередині **WrapPanel** не пристиковується до заданої сторони панелі, порядок оголошення елементів відіграє важливу роль (вміст візуалізується від першого елемента до останнього). У рішенні **SimpleWrapPanel** знаходиться наступна розмітка:

```
<Window x:Class="SimpleWrapPanel.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:SimpleWrapPanel"
        mc:Ignorable="d"
        Title="Дослідження панелей!" Height="284" Width="323">
```

```
  <WrapPanel Background="LightSteelBlue">
    <Label Name="lblInstruction" Width="328" Height="27" FontSize="15"
        Content="Enter Car Information"/>
    <Label Name="lblMake" Content="Make"/>
    <TextBox Name="txtMake" Width="193" Height="25"/>
    <Label Name="lblColor" Content="Color"/>
    <TextBox Name="txtColor" Width="193" Height="25"/>
    <Label Name="lblPetName" Content="Pet Name"/>
    <TextBox Name="txtPetName" Width="193" Height="25"/>
    <Button Name="btnOK" Width="80" Content="OK"/>
  </WrapPanel>
</Window>
```

Коли ця розмітка завантажена, при зміні ширини вікна вміст виглядає не особливо привабливо, бо вміст продовжується зліва направо всередині вікна (рис. 3.4)

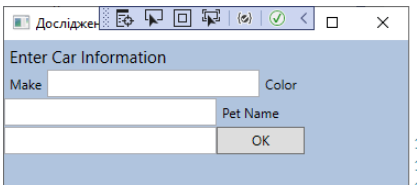


Рис. 3.4. Вміст в панелі **WrapPanel** поводиться багато в чому подібно до традиційної сторінки HTML

За замовчуванням вміст **WrapPanel** продовжується зліва направо. Проте, змінивши значення властивості **Orientation** на **Vertical**, можна змусити вміст продовжуватися зверху вниз:

```
<WrapPanel Background="LightSteelBlue" Orientation="Vertical">
```

Панель **WrapPanel** (як і ряд інших типів панелей) може бути оголошена із заданням значень **ItemWidth** і **ItemHeight**, які управляють стандартним розміром кожного елемента. Якщо піделемент задає власні значення **Height** і/або **Width**, то він позиціонуватиметься відносно розміру, встановленого для нього панеллю. Погляньте на наступну розмітку:

```
<Window x:Class="SimpleWrapPanel.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:SimpleWrapPanel"
        mc:Ignorable="d"
        Title="Дослідження панелей!" Height="150" Width="650">
  <WrapPanel Background="LightSteelBlue" Orientation="Horizontal" ItemWidth ="200"
      ItemHeight ="30">
    <Label x:Name="lblInstruction" FontSize="15" Content="Enter Car Information"/>
    <Label x:Name="lblMake" Content="Make"/>
```

```
    <TextBox x:Name="txtMake"/>
    <Label x:Name="lblColor" Content="Color"/>
    <TextBox x:Name="txtColor"/>
    <Label x:Name="lblPetName" Content="Pet Name"/>
    <TextBox x:Name="txtPetName"/>
    <Button x:Name="btnOK" Width ="80" Content="OK"/>
  </WrapPanel>
</Window>
```

У результаті візуалізації отримаємо вікно, показане на рис. 3.5 (зверніть увагу на розмір і позицію елемента управління **Button**, для якого було задано унікальне значення **Width**).

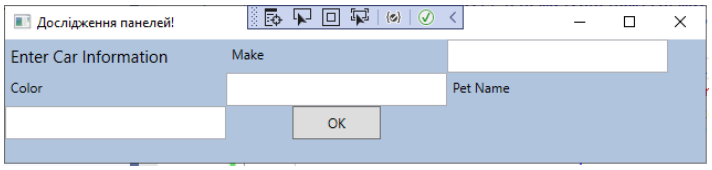


Рис. 3.5. Панель **WrapPanel** може встановлювати ширину і висоту окремого елемента

Після перегляду рис. 3.5 ви напевно погодитеся з тим, що панель **WrapPanel** – зазвичай не кращий вибір для організації вмісту безпосередньо у вікні, бо її елементи можуть безладно змішуватися, коли користувач змінює розмір вікна. У більшості випадків **WrapPanel** буде піделементом панелі іншого типу, дозволяючи невеликій зоні вікна переносити свій вміст при зміні розміру (як, наприклад, елемент управління **ToolBar**).

Позиціонування вмісту всередині панелей StackPanel

Подібно до **WrapPanel** елемент управління **StackPanel** організовує вміст всередині одинокого рядка, який може бути орієнтований горизонтально або вертикально (за замовчуванням) залежно від значення, присвоєного властивості **Orientation**. Проте відмінність між ними полягає в тому, що **StackPanel** не намагається переносити вміст при зміні розміру вікна користувачем. Натомість елементи в **StackPanel** просто розтягуються (згідно вибраної орієнтації), пристосовуючись до розміру самої панелі **StackPanel**. Наприклад, у рішенні **SimpleStackPanel** міститься розмітка, яка в результаті дає вивід, показаний на рис. 3.6:

```
<Window x:Class="SimpleStackPanel.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:SimpleStackPanel"
        mc:Ignorable="d"
        Title="Дослідження панелей" Height="200" Width="300">
  <StackPanel Background="LightSteelBlue" Orientation ="Vertical">
```

```
    <Label Name="lblInstruction" FontSize="15" Content="Enter Car Information"/>
    <Label Name="lblMake" Content="Make"/>
    <TextBox Name="txtMake"/>
    <Label Name="lblColor" Content="Color"/>
    <TextBox Name="txtColor"/>
    <Label Name="lblPetName" Content="Pet Name"/>
    <TextBox Name="txtPetName"/>
    <Button Name="btnOK" Width ="80" Content="OK"/>
  </StackPanel>
</Window>
```

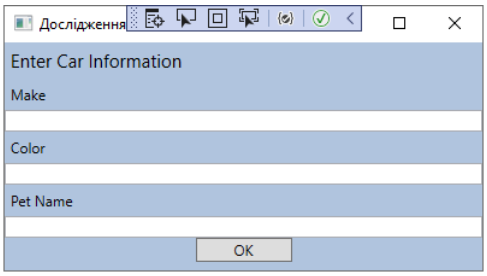


Рис. 3.6. Вертикальне розміщення вмісту

Якщо присвоїти властивості **Orientation** значення **Horizontal**, тоді візуалізований вивід стане таким, як на рис. 3.7:

```
<StackPanel Background="LightSteelBlue" Orientation ="Horizontal">
```

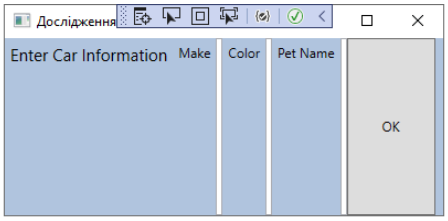


Рис. 3.7. Горизонтальне розміщення вмісту

Подібно до **WrapPanel** панель **StackPanel** теж рідко застосовується для організації вмісту прямо всередині вікна. Панель **StackPanel** повинна використовуватися як вкладена панель в якійсь іншій головній панелі.

Позиціонування вмісту всередині панелей Grid

З усіх панелей, API-, які надаються інтерфейсами WPF, панель **Grid**, поза сумнівом, є найгнучкішою. Аналогічно таблиці HTML панель **Grid** може складатися з набору комірок, кожна з яких має свій вміст. При визначенні **Grid** виконуються перераховані нижче кроки.

1. Визначення і конфігурація кожної колонки.
2. Визначення і конфігурація кожного рядка.

3. Призначення вмісту кожній комірці сітки з використанням синтаксису *приєднуваних властивостей*.

На замітку! Якщо не визначити якісь рядки і стовпчики, то за замовчуванням елемент **Grid** складатиметься з єдиної комірки, яка заповнює всю поверхню вікна. Крім того, якщо не встановити комірку (стовпчик і рядок) для піделемента всередині **Grid**, тоді він автоматично розміститься у стовпчику **0** і рядку **0**.

Перші два кроки (визначення стовпчиків і рядків) виконуються з використанням елементів **Grid.ColumnDefinitions** і **Grid.RowDefinitions**, які містять колекції елементів **ColumnDefinition** і **RowDefinition** відповідно. Кожна комірка всередині сітки насправді є справжнім об'єктом .NET, так що можна за бажанням налаштовувати зовнішній вигляд і поведінку кожного елемента.

Нижче представлено просте визначення **Grid** (застосунок **SimpleGrid**), яке організовує вміст інтерфейсу користувача, як показано на рис. 3.8:

```
<Window x:Class="SimpleGrid.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:SimpleGrid"
        mc:Ignorable="d"
        Title="Дослідження панелей" Height="186" Width="501">
  <Grid ShowGridLines ="True" Background ="AliceBlue">
    <!-- Визначення рядків/стовпчиків -->
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>

    <!-- Додати елементи в комірки сітки -->
    <Label Name="lblInstruction" Grid.Column ="0" Grid.Row ="0" FontSize="15">
        Enter Car Information</Label>
    <Button Name="btnOK"  Height ="30" Grid.Column ="0" Grid.Row ="0" >OK</Button>
    <Label Name="lblMake" Grid.Column ="1" Grid.Row ="0">Make</Label>
    <TextBox Name="txtMake" Grid.Column ="1" Grid.Row ="0" Width="193" Height="25"/>
    <Label Name="lblColor" Grid.Column ="0" Grid.Row ="1" >Color</Label>
    <TextBox Name="txtColor" Width="193" Height="25" Grid.Column ="0" Grid.Row ="1" />
    <!-- Додати кольори до іменованої комірки, просто щоб зробити картину цікавішою -->

    <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
    <Label Name="lblPetName" Grid.Column ="1" Grid.Row ="1" >Pet Name</Label>
    <TextBox Name="txtPetName" Grid.Column ="1" Grid.Row ="1" Width="193" Height="25"/>
```

```
  </Grid>
</Window>
```

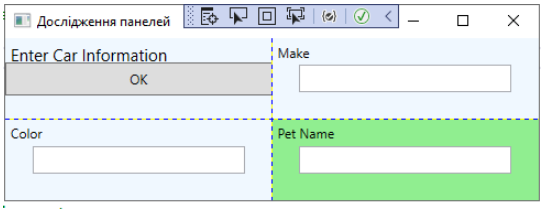


Рис. 3.8. Панель **Grid** у дії

Зверніть увагу, що кожен елемент (включаючи елемент **Rectangle** яскраво-зеленого кольору) прикріпляється до комірки сітки із використанням приєднуваних властивостей **Grid.Row** і **Grid.Column**. За замовчуванням порядок комірок починається з лівої верхньої комірки, яка задається з використанням **Grid.Column="0"** і **Grid.Row="0"**. Враховуючи, що сітка складається всього з чотирьох комірок, права нижня комірка може бути ідентифікована як **Grid.Column="1"** і **Grid.Row="1"**.

Встановлення розмірів стовпців і рядків в панелі Grid

Задавати розміри стовпців і рядків в панелі **Grid** можна одним з трьох способів:

- встановлення *абсолютних* розмірів (наприклад, **100**);
- встановлення *автоматичних* розмірів;
- встановлення *відносних* розмірів (наприклад, **3***).

Встановлення абсолютних розмірів – саме те, на що і можна було розраховувати; для розміру стовпчика (або рядка) задається специфічне число одиниць, незалежних від пристрою. При встановленні автоматичних розмірів розмір кожного стовпчика або рядка визначається на основі розмірів елементів управління, які містяться в стовпчику або рядку. Встановлення відносних розмірів практично еквівалентна заданню розмірів у відсотках всередині стилю CSS. Загальна сума чисел в стовпчиках або рядках з відносними розмірами розподіляється на загальний обсяг доступного простору.

У наступному прикладі перший рядок отримує **25%** простору, а другий – **75%** простору:

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="1*"/>
  <ColumnDefinition Width="3*"/>
</Grid.ColumnDefinitions>
```

Панелі Gride з типами GridSplitter

Панелі **Grid** також здатні підтримувати *роздільники*. Роздільники дозволяють кінцевому користувачу змінювати розміри стовпчиків і рядків сітки. При цьому вміст кожної комірки зі змінюваним розміром реорганізує себе на основі елементів, які знаходяться в ньому. Додати роздільники до **Grid** досить

просто: потрібно визначити елемент управління **GridSplitter** та із застосуванням синтаксису приєднуваних властивостей вказати рядок або стовпчик, на який він впливає.

Майте на увазі, що для того, щоб роздільник був видний на екрані, знадобиться присвоїти значення його властивості **Width** або **Height** (залежно від вертикального чи горизонтального розділення). Нижче показана проста панель **Grid** з роздільником на першому стовпчику (**Grid.Column="0"**) з файла **GridWithSplitter.xaml**:

```
<Window x:Class="SimpleGrid.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:SimpleGrid"
        mc:Ignorable="d"
        Title="Дослідження панелей" Height="100" Width="350">
  <Grid Background ="AliceBlue">
    <!-- Визначити стовпчики -->
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width ="Auto"/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <!-- Додати мітку в комірку 0 -->
    <Label Name="lblLeft" Background ="GreenYellow" Grid.Column="0" Content ="Left!"/>
    <!-- Визначити роздільник -->
    <GridSplitter Background ="White" Grid.Column ="0" Width ="5"/>
    <!-- Додати мітку в комірку 1 -->
    <Label Name="lblRight" Grid.Column ="1" Content ="Right!"/>
  </Grid>
</Window>
```

Передусім, зверніть увагу, що стовпчик, який підтримуватиме роздільник, має властивість **Width**, встановлену в **Auto**. На додаток елемент **GridSplitter** використовує синтаксис приєднуваних властивостей для зазначення, з якою колонкою він працює. У виводі (рис. 3.9) можна помітити **5**-піксельний роздільник, який дозволяє змінювати розмір кожного елемента **Label**. Через те, що для елементів **Label** не було задано властивість **Height** або **Width**, вони заповнюють всю комірку.

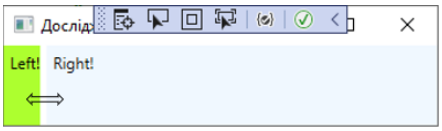


Рис. 3.9. Панель **Grid** з роздільником

Позиціонування вмісту всередині панелей DockPanel

Панель **DockPanel** зазвичай використовується як контейнер, який містить будь-яке число додаткових панелей для групування пов'язаного вмісту. Панелі **DockPanel** використовують синтаксис *приєднуваних властивостей* (як було показано в типах **Canvas** і **Grid**) для управління місцем, куди пристиковуватиметься кожен елемент всередині **DockPanel**.

У рішенні **SimpleDockPanel** визначена наступна проста панель **DockPanel**, яка дає результат, показаний на рис. 3.10:

```
<Window x:Class="SimpleDockPanel.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:SimpleDockPanel"
        mc:Ignorable="d"
        Title="Дослідження панелей!" Height="186" Width="300">
  <DockPanel LastChildFill ="True" Background="AliceBlue">
    <!-- Пристикувати елементи до панелі -->
    <Label DockPanel.Dock ="Top" Name="lblInstruction" FontSize="15"
        Content="Enter Car Information"/>
    <Label DockPanel.Dock ="Left" Name="lblMake" Content="Make"/>
    <Label DockPanel.Dock ="Right" Name="lblColor" Content="Color"/>
    <Label DockPanel.Dock ="Bottom" Name="lblPetName" Content="Pet Name"/>
    <Button Name="btnOK" Content="OK"/>
  </DockPanel>
</Window>
```

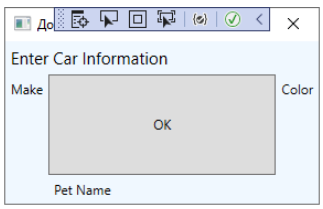


Рис. 3.10. Проста панель **DockPanel**

На замітку! Якщо додати багато елементів до однієї сторони **DockPanel**, то вони вишикуються уздовж заданої грані за порядком їх оголошення.

Перевага використання типів **DockPanel**: зміна користувачем розміру вікна залишає кожен елемент прикріпленим до заданої (за допомогою **DockPanel.Dock**) сторони панелі. Також зверніть увагу, на те, що всередині відкриваючого дескриптора **DockPanel** у цьому прикладі атрибут **LastChildFill** встановлений в **true**. Через те, що елемент **Button** «останній дочірній» елемент в контейнері, він буде розтягнутий, щоб зайняти весь залишковий простір.

Включення прокручування в типах панелей

Корисно згадати, що у рамках інфраструктури WPF постачається клас **ScrollViewer**, який забезпечує автоматичну поведінку прокручування даних всередині об'єктів панелей. В проекті **SimpleScrollViewer** він визначається так:

```
<Window x:Class="SimpleScrollViewer.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:SimpleScrollViewer"
        mc:Ignorable="d"
        Title="Дослідження панелей" Height="150" Width="401">
  <ScrollViewer>
    <StackPanel>
      <Button Content ="First" Background = "Green" Height ="40"/>
      <Button Content ="Second" Background = "Red" Height ="40"/>
      <Button Content ="Third" Background = "Pink" Height ="40"/>
      <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
      <Button Content ="Fifth" Background = "Blue" Height ="40"/>
    </StackPanel>
  </ScrollViewer>
</Window>
```

Результат візуалізації приведеного визначення XAML представлений на рис. 3.11 (видно, що справа у вікні відображається лінійка прокручування, бо розміру вікна бракує, щоб показати всі п'ять кнопок).

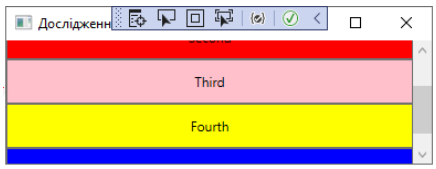


Рис. 3.11. Робота з класом **ScrollViewer**

Як і можна було очікувати, кожен клас панелі пропонує багато членів, які дозволяють точно налаштовувати розміщення вмісту. Як зв'язане зауваження: багато елементів управління WPF підтримують дві зручні властивості (**Padding** і **Margin**), які надають елементу управління можливість самостійного інформування панелі про те, як з цим елементом потрібно поводитися. Зокрема, властивість **Padding** управляє тим, скільки вільного простору повинно оточувати внутрішній елемент управління, а властивість **Margin** контролює обсяг додаткового простору поза елементом управління.

На цьому короткий екскурс в основні типи панелей WPF і різні способи позиціонування їх вмісту завершений. Далі ми покажемо, як використати візуальні конструктори Visual Studio для створення компонувань.

Конфігурування панелей з використанням візуальних конструкторів Visual Studio

Тепер, коли ви ознайомилися з розміткою XAML, яка використовувалася при визначенні ряду загальних диспетчерів компонування, корисно знати, що

IDE-середовище Visual Studio пропонує дуже хорошу підтримку для конструювання компонувань. Ключовим компонентом є вікно **Document Outline** (Структура документу), описаний раніше в лекції. Щоб проілюструвати деякі основи, ми створимо новий проект застосування WPF з іменем **VisualLayoutTester**.

У початковій розмітці для **Window** за замовчуванням використовується диспетчер компонування **Grid**:

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

  </Grid>
</Window>
```

Використавши диспетчер компонування **Grid** (рис. 3.12) ви помітите, що можна легко розділяти і міняти розміри комірок сітки, використовуючи візуальний конструктор.

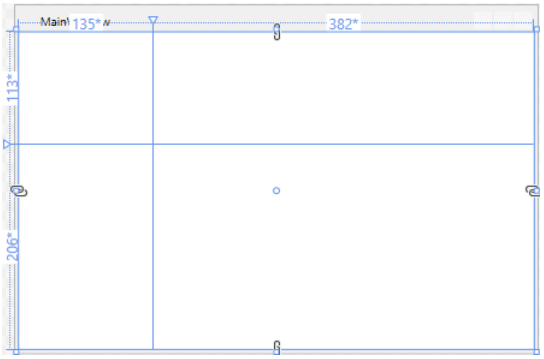


Рис. 3.12. Використовуючи візуальний конструктора IDE-середовища, елемент управління **Grid** може бути розділений на комірки

Тепер припустимо, що визначена сітка з якимсь числом комірок. Далі можна перетягувати елементи управління в комірку сітки, яка нас цікавить і тоді IDE-середовище буде автоматично встановлювати їх властивості **Grid.Row** і **Grid.Column**. Ось як може виглядати розмітка, згенерована IDE-середовищем після перетягування елемента **Button** у передбачувану комірку:

```
<Button Content="Button" Grid.Column="1" HorizontalAlignment="Left" Margin="174,101,0,0"
      Grid.Row="1" VerticalAlignment="Top" Width="75"/>
```

Нехай, наприклад, було вирішено взагалі не використовувати елемент **Grid**. Клацання правою кнопкою миші на будь-якому вузлі розмітки у вікні **Document Outline** спричиняє відкриття контекстного меню, яке містить пункт,

що дозволяє замінити поточний контейнер іншим (рис. 3.13). Треба усвідомлювати, що така дія (з високою вірогідністю) радикально змінить позиції елементів управління, тому що вони стануть задовольняти правилам нового типу панелі.

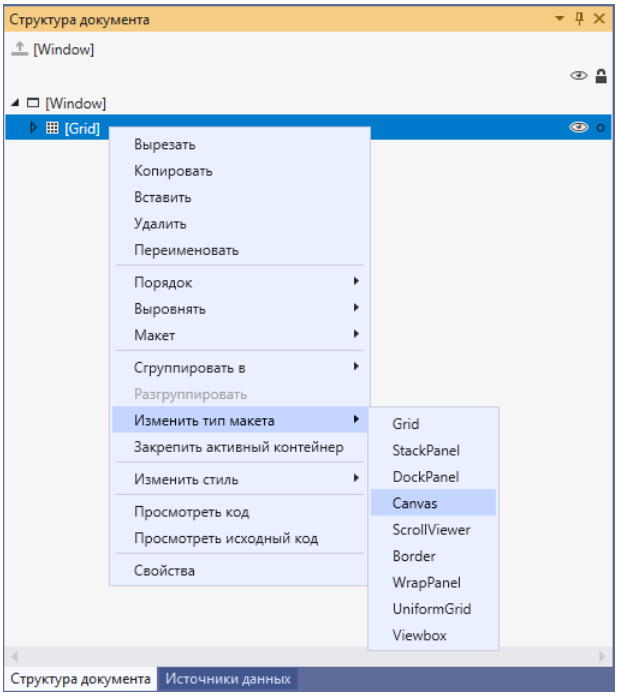


Рис. 3.13. Вікно **Document Outline** дозволяє виконувати перетворення в інші типи панелей

Ще один зручний трюк пов'язаний з можливістю вибору у візуальному конструкторі набору елементів управління і наступного їх групування всередині нового вкладеного диспетчера компонування. Припустимо, що є панель **Canvas**, в якій визначений набір довільних об'єктів (за бажання первинну панель **Grid** можна перетворити в **Canvas** із застосуванням прийому, продемонстрованого на рис. 3.13). Виділимо декілька елементів на поверхні візуального конструктора, клацаючи на кожному елементі лівою кнопкою миші при натиснутій клавіші <**Ctrl**>. Якщо потім клацнути правою кнопкою миші на виділеному наборі елементів, то за допомогою контекстного меню, що відкрилося, їх можна згрупувати в нову вкладену панель. Для цієї демонстрації використаємо проект **WpfTestApp** із ***Лекція 02. Побудова застосунків WPF з використанням Visual Studio*** (рис. 3.14).

Після цього знову треба подивитися у вікно **Document Outline**, щоб проконтролювати вкладену систему компонування. Через те, що будуються повнофункціональні вікна WPF, швидше за все, завжди треба буде використовувати систему вкладених компонувань, а не просто вибирати єдину панель для відображення всього інтерфейсу користувача (фактично у всіх наступних прикладах застосунків WPF, зазвичай так і буде робитися).

Як фінальне зауваження зазначимо, що всі вузли у вікні **Document Outline** підтримують перетягування. Наприклад, якщо вимагається перемістити у батьківську панель елемент управління, який у нинішній момент знаходиться усередині **Canvas**, тоді можна вчинити так, як ілюструється на рис. 3.15.

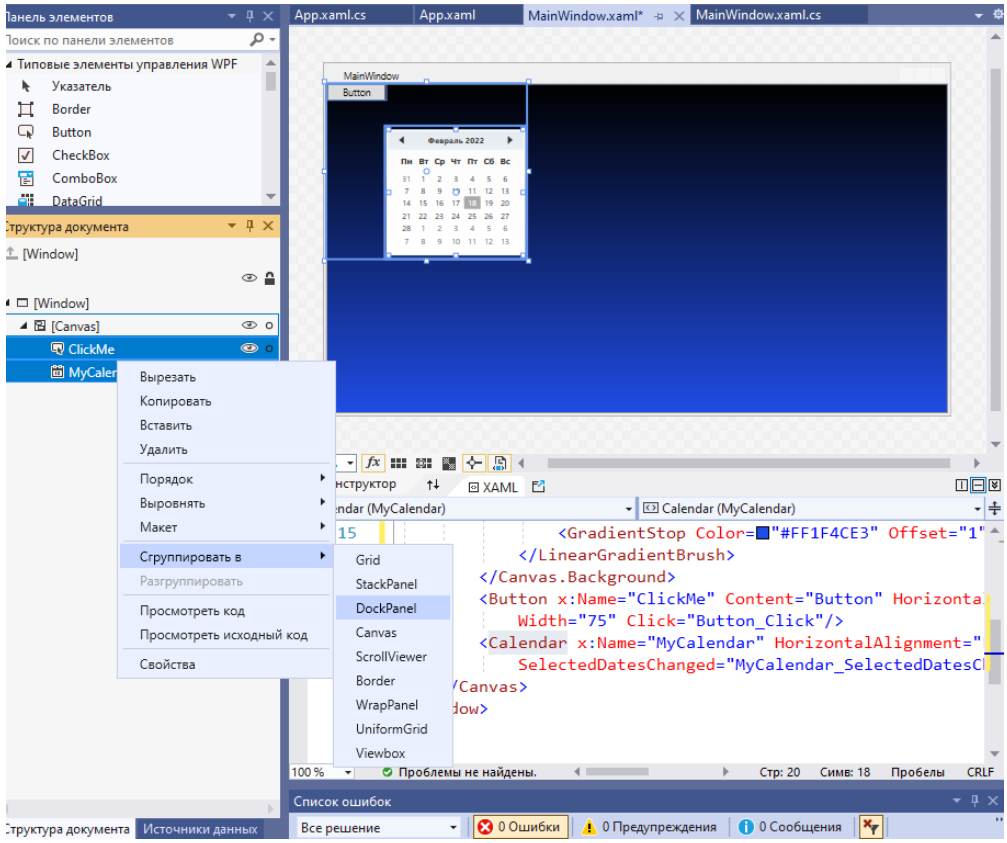


Рис. 3.14. Групування елементів в нову вкладену панель

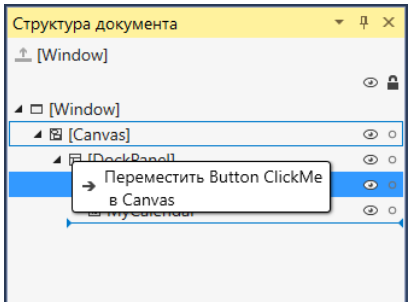


Рис. 3.15. Переміщення елементів за допомогою вікна **Document Outline**

У наступних лекціях, присвячених WPF, будуть представлені додаткові прискорені прийоми для роботи з компонуванням там, де вони можливі. Проте, безперечно корисно присвятити деякий час самостійному експериментуванню і перевірці різноманітних засобів і способів. У наступному прикладі цієї лекції

демонструватиметься побудова вкладеного диспетчера компонування для спеціального застосунку обробки тексту (з перевіркою правопису).

Резюме

У цій лекції розглядалися деякі аспекти елементів управління WPF, починаючи з огляду набору інструментів для елементів управління і ролі диспетчерів компонування (панелей).

Контрольні питання.

1. Приведіть приклади основних елементів управління WPF і їх призначення.
2. В який простір імен і з якої збірки запаковані стандартні елементи управління?
3. Яке призначення візуального конструктора?
4. Для чого використовується властивість x:Name елемента управління розміщеного на поверхні візуального конструктора?
5. Як зв’язати елемент управління, який розміщений у візуальному конструкторі з пов’язаним кодом у файлі C#?
6. Як вікно Document Outline пов’язане з візуальним конструктором і для чого воно призначене?
7. За рахунок чого здійснюється управління компонуванням вмісту всередині вікна?
8. Навіщо у WPF-елементі Window потрібно розміщувати панелі компонування або їх ще називають диспетчерами компонування?
9. Перелічіть та охарактеризуйте основні елементи управління типу панелей WPF.
10. У яких випадках найчастіше використовується панель Canvas?
11. У яких випадках найчастіше використовується панель WrapPanel?
12. У яких випадках найчастіше використовується панель StackPanel?
13. У яких випадках найчастіше використовується панель Grid?
14. Який механізм додавання роздільника GridSplitter до панелі Grid?
15. Як зробити роздільник видимим на екрані?
16. Який синтаксис властивості потрібно використати для додавання роздільника?
17. Як розділити панель Grid на рядки та стовпці?
18. Який механізм об’єднання суміжних рядків або стовпців?
19. У яких випадках найчастіше використовується панель DockPanel?
20. Як організувати прокручування вмісту всередині об’єктів панелей?
21. Як можна конфігурувати панелі у візуальному конструкторі Visual Studio? Під конфігуруванням розуміються зміни елементів управління у візуальному конструкторі при потребі.

Лекція 4. Побудова вікон

Побудова вікна з використанням вкладених панелей

Як згадувалося раніше, у типовому вікні WPF для отримання бажаної системи компонування використовується не єдиний елемент управління типу панелі, а одні панелі вкладаються всередину інших. Розпочнемо зі створення нового проекту застосунку WPF на ім'я **MyWordPad**.

Нашою метою є конструювання компонування, в якому головне вікно має розташовану:

- у верхній частині систему меню;
- під нею – панель інструментів;
- у нижній частині вікна – рядок стану. Він міститиме область для підказок, відображуваних при виборі пункта меню (або кнопки в панелі інструментів).

Система меню і панель інструментів нададуть *тригери* інтерфейсу користувача для закриття застосунку і відображення варіантів правопису у віджеті **Expander**[11].

Щоб приступити до побудови інтерфейсу користувача модифікуємо початкове визначення XAML типу **Window** для використання дочірнього елемента **DockPanel** замість стандартного елемента управління **Grid**:

```
<Window x:Class="MyWordPad.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:MyWordPad"
        mc:Ignorable="d"
        Title="Перевірка орфографії" Height="350" Width="525">
  <!-- Ця панель встановлює вміст вікна -->
  <DockPanel>

  </DockPanel>
</Window>
```

Побудова системи меню

Системи меню у WPF представлені класом **Menu**, який підтримує *колекцію об'єктів* **MenuItem**. При побудові системи меню у XAML кожен об'єкт **MenuItem** здатний обробляти різноманітні події, найважливішою з яких є **Click**, яка виникає при виборі піделемента **MenuItem** кінцевим користувачем. У цьому прикладі створюються два пункти меню верхнього рівня (**File** (Файл) і **Tools** (Сервіс); *пізніше* буде додано меню **Edit** (Правка)), які містять в собі піделементи **Exit** (Вихід) і **Spelling Hints** (Підказки з правопису) відповідно.

---

[11] https://drive.google.com/drive/u/1/folders/1caBi6m3Sb3erig9wS4Z9pruhiTucvgxH. Відкрити файл – 14.Expander.pdf.

На додаток до обробки події **Click** для кожного піделемента потрібно також обробити події **MouseEnter** і **MouseExit**, які використовуються для встановлення тексту в рядку стану. Додамо в контекст елемента **DockPanel**[12] наступну розмітку:

```
<Window x:Class="MyWordPad.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:MyWordPad"
        mc:Ignorable="d"
        Title="Перевірка орфографії" Height="350" Width="525">
  <!-- Ця панель встановлює вміст вікна -->
  <DockPanel>
    <!-- Стикувати систему меню до верхньої частини -->
    <Menu DockPanel.Dock="Top" HorizontalAlignment="Left" Background="White"
          BorderBrush="Black">
      <MenuItem Header="_File">
      <Separator/>
      <MenuItem Header="_Exit" MouseEnter="MouseEnterExitArea"
          MouseLeave="MouseLeaveArea" Click="FileExit_Click"/>
      </MenuItem>
      <MenuItem Header="_Tools">
        <MenuItem Header="_Spelling Hints" MouseEnter="MouseEnterToolsHintsArea"
          MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click"/>
      </MenuItem>
    </Menu>
  </DockPanel>
</Window>
```

Зверніть увагу, що система меню пристикована до верхньої частини **DockPanel**. Крім того, елемент **Separator** використовується для додавання в систему меню тонкої горизонтальної лінії прямо перед пунктом **Exit**. Значення **Header** для кожного **MenuItem** містять символ підкреслення (наприклад, **_Exit**). Так само задається символ, який підкреслюватиметься, коли кінцевий користувач натисне клавішу <**Alt**> (для вводу клавіатурного скорочення).

Після побудови системи меню потрібно реалізувати різні обробники подій. Передусім, є обробник пункту меню **File|Exit** (Файл|Вихід), **FileExit_Click()**, який просто закриває вікно, що у свою чергу спричиняє завершення застосунку, бо це вікно найвищого рівня. Обробники подій **MouseEnter** і **MouseExit** для кожного піделемента у результаті оновлюватимуть рядок стану; проте наразі ми просто залишимо ці обробники порожніми. Нарешті, обробник **ToolsSpellingHints_Click()** для пункта меню **Tools|Spelling Hints** також залишимо поки порожнім. Нижче показані поточні оновлення

[12] https://drive.google.com/drive/u/1/folders/1caBi6m3Sb3erig9wS4Z9pruhiTucvgxH. Відкрити файл – 20. Елемент управління DockPanel.pdf.

файла відокремленого коду (у тому числі додані оператори **using** для просторів імен **Microsoft.Win32** і **System.IO**):

```
using System;
using System.Windows;
using System.Windows.Input;
using Microsoft.Win32;
using System.IO;

namespace MyWordPad
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }
    private void FileExit_Click(object sender, RoutedEventArgs e)
    {
      // Закрити це вікно.
      this.Close();
    }
    private void ToolsSpellingHints_Click(object sender, RoutedEventArgs e)
    {

    }
    private void MouseEnterExitArea(object sender, MouseEventArgs e)
    {

    }
    private void MouseEnterToolsHintsArea(object sender, MouseEventArgs e)
    {

    }
    private void MouseLeaveArea(object sender, MouseEventArgs e)
    {

    }
  }
}
```

Візуальна побудова меню

Зауважимо, що корисно знати, як вручну визначати елементи у XAML, та така робота може бути набридливою. У Visual Studio підтримується можливість візуального конструювання систем меню, панелей інструментів, рядків стану і багатьох інших елементів управління інтерфейсу користувача. Клацання правою кнопкою миші на елементі управління **Menu** спричиняє відкриття контекстного

меню, яке містить **Add MenuItem** (Додати **MenuItem**), що дозволяє додати новий пункт меню в елемент управління **Menu**. Після додавання набору пунктів верхнього рівня можна зайнятися додаванням пунктів підменю, роздільників, розгортання і згортання самого меню і виконання інших пов'язаних з меню операцій за допомогою другого клацання правою кнопкою миші.

У залишковій частині прикладу **MyWordPad** ви побачите фінальну згенеровану розмітку XAML; проте, потратьте деякий час на експерименти з візуальними конструкторами.

Побудова панелі інструментів

Панелі інструментів (у WPF – це клас **ToolBar**) зазвичай пропонують альтернативний спосіб активізації пунктів меню. Помістіть наступну розмітку безпосередньо після закриваючого дескриптора визначення **Menu**:

```
<!-- Помістіть панель інструментів під областю меню -->
<ToolBar DockPanel.Dock ="Top" >
  <Button Content="Exit" MouseEnter="MouseEnterExitArea" MouseLeave="MouseLeaveArea"
      Click="FileExit_Click"/>
  <Separator/>
  <Button Content="Check" MouseEnter="MouseEnterToolsHintsArea"
      MouseLeave="MouseLeaveArea" Click="ToolsSpellingHints_Click" Cursor="Help" />
</ToolBar>
```

Як бачимо елемент **ToolBar**[13] утворений з двох елементів управління **Button**, які обробляють одні і ті ж події тими ж методами з файла коду. За допомогою такого прийому можна дублювати обробники для обслуговування і пунктів меню, і кнопок панелі інструментів. Хоча в цій панелі використовуються типові кнопки, які натискаються, ви повинні брати до уваги, що тип **ToolBar «***є***» ContentControl**, а тому на його поверхню можна поміщати будь-які типи (скажімо, розкривні списки, зображення і графіку). Ще один цікавий аспект пов'язаний з тим, що кнопка **Check** (Перевірити) підтримує спеціальний курсор миші через властивість **Cursor**.

> **На замітку!** Елемент **ToolBar** може бути додатково поміщений всередину елемента **ToolBarTray**[14], який управляє компонуванням, стикуванням і перетягуванням для набору об'єктів **ToolBar**.

Побудова рядка стану

Елемент управління рядком стану (**StatusBar**) стикується з нижньою частиною **DockPanel** і містить єдиний елемент управління **TextBlock**, який раніше в лекції не використовувався. Елемент **TextBlock** можна використовувати для зберігання тексту з форматуванням на зразок виділення напівжирним і підкресленням, додавання розривів рядків і т. д. Помістимо

---

[13] https://drive.google.com/drive/u/1/folders/1caBi6m3Sb3erig9wS4Z9pruhiTucvgxH. Відкрити файл – 21. Елемент управління ToolBar.pdf.

[14] https://drive.google.com/drive/u/1/folders/1caBi6m3Sb3erig9wS4Z9pruhiTucvgxH. Відкрити файл – 22. Елементи управління ToolBarTray.pdf і StatusBar.pdf.

приведену нижче розмітку відразу після попереднього визначення елемента управління **ToolBar**:

```
<!-- Розмістити рядок стану внизу -->
<StatusBar DockPanel.Dock="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText" Text="Ready"/>
  </StatusBarItem>
</StatusBar>
```

Завершення проектування інтерфейсу користувача

Фінальний аспект проектування інтерфейсу користувача пов'язаний з визначенням підтримуючого роздільники елемента **Grid**, в якому визначені дві колонки. Ліворуч знаходиться елемент управління **Expander**, поміщений всередину **StackPanel**, який відображатиме список передбачуваних варіантів правопису, а справа – елемент **TextBox** з підтримкою багаторядкового тексту, лінійок прокручування і включеною перевіркою орфографії. Елемент **Grid** може бути цілком розміщений у лівій частині батьківської панелі **DockPanel**. Щоб завершити визначення інтерфейсу користувача вікна, додамо наступну розмітку XAML, розташувавши її безпосередньо під розміткою, яка описує **StatusBar**:

```
<Grid DockPanel.Dock="Left" Background="AliceBlue">
  <!-- Визначити рядки і стовпчики -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column="0" Width="5" Background="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment="Stretch" >
    <Label Name="lblSpellingInstructions" FontSize="14" Margin="10,10,0,0">
          Spelling Hints
    </Label>
    <Expander Name="expanderSpelling" Header="Try these!" Margin="10,10,10,10">
      <!-- Буде заповнюватися програмою -->
      <Label Name="lblSpellingHints" FontSize="12"/>
    </Expander>
  </StackPanel>
  <!-- Це буде областю для вводу -->
  <TextBox Grid.Column="1"
    SpellCheck.IsEnabled="True"
    AcceptsReturn="True"
    Name="txtData"
    FontSize="14"
    BorderBrush="Blue"
    VerticalScrollBarVisibility="Auto"
    HorizontalScrollBarVisibility="Auto">
  </TextBox>
</Grid>
```

Реалізація обробників подій MouseEnter/MouseLeave

Отже, інтерфейс користувача вікна готовий. Потрібно лише реалізувати обробники подій, які залишилися. Розпочнемо з оновлення файла коду C# так, щоб кожен з обробників подій **MouseEnter** і **MouseLeave** встановлював в текстовій панелі рядка стану відповідне повідомлення, яке надасть допомогу кінцевому користувачеві:

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

  private void FileExit_Click(object sender, RoutedEventArgs e)
  {
    // Закрити це вікно.
    this.Close();
  }

  private void ToolsSpellingHints_Click(object sender, RoutedEventArgs e)
  {

  }

  private void MouseEnterExitArea(object sender, MouseEventArgs e)
  {
    statBarText.Text = "Exit the Application";
  }

  private void MouseEnterToolsHintsArea(object sender, MouseEventArgs e)
  {
    statBarText.Text = "Show Spelling Suggestions";
  }

  private void MouseLeaveArea(object sender, MouseEventArgs e)
  {
    statBarText.Text = "Ready";
  }
}
```

Тепер застосунок можна запустити. Текст в рядку стану повинен змінюватися залежно від того, над яким пунктом меню або кнопкою панелі інструментів знаходиться курсор.

Реалізація логіки перевірки правопису

Інфраструктура WPF має вбудовану підтримку перевірки правопису, незалежну від продуктів Microsoft Office. Це означає, що використовувати рівень взаємодії COM для звернення до функції перевірки правопису Microsoft

Word не знадобиться: та ж сама функціональність додається за допомогою всього декількох рядків коду.

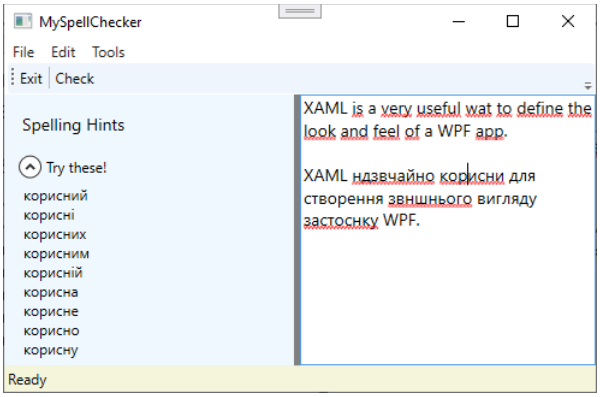


Рис. 4.1. Приклад застосунку **MyWordPad**

Згадайте, що при визначенні елемента управління **TextBox** властивість **Spellcheck.IsEnabled** встановлюється в **true**. У результаті неправильно написані слова підкреслюються червоною хвилястою лінією, як це відбувається в Microsoft Office. Більше того, програмна модель, яка лежить в основі, надає доступ до механізму перевірки правопису, який дозволяє отримати список передбачуваних варіантів для слів, написаних з помилкою. Додамо в метод **ToolsSpellingHints_Click()** наступний код:

```
private void ToolsSpellingHints_Click(object sender, RoutedEventArgs e)
{
  string spellingHints = string.Empty;
  // Спробувати отримати помилку правопису в поточному положенні курсора вводу.
  SpellingError error = txtData.GetSpellingError(txtData.CaretIndex);
  if (error != null)
  {
    // Побудувати рядок з передбачуваними варіантами правопису.
    foreach (string s in error.Suggestions)
    {
      spellingHints += $"{s}\n";
    }
    // Відобразити передбачувані варіанти і розкрити елемент Expander.
    lblSpellingHints.Content = spellingHints;
    expanderSpelling.IsExpanded = true;
  }
}
```

Приведений вище код досить простий. Із використанням властивості **CaretIndex** отримується об'єкт **SpellingError** та обчислюється поточне положення курсора вводу у текстовому полі. Якщо у вказаному місці наявна помилка (тобто значення **error** не дорівнює **null**), тоді здійснюється прохід в циклі за списком передбачуваних варіантів з використанням властивості **Suggestions**. Після того, як всі передбачувані варіанти для неправильно

написаного слова отримані, вони поміщаються в елемент **Label** всередині елемента **Expander**.

Після вивчення управляючих команд, ми додамо додаткові можливості.

Поняття команд WPF

Команди мають декілька цілей. Перша з них полягає в тому, щоб відокремити *семантику* та *об'єкт*, який викликає команду, від *логіки*, яка виконує команду. Це дозволяє декільком розрізненим джерелам викликати одну і ту ж логіку команди, а також дозволяє налаштовувати логіку команди для різних цілей. Наприклад, операції редагування **Copy**, **Cut** і **Paste**, наявні у багатьох застосунках, можна викликати за допомогою різних дій користувача, якщо вони реалізовані за допомогою команд. Застосунок може дозволити користувачеві вирізувати вибрані об'єкти або текст, натиснувши кнопку, вибравши елемент в меню або використовуючи комбінацію клавіш, наприклад **CTRL+X**. Використовуючи команди, ви можете зв'язати кожен тип дій користувача з однією і тією ж логікою.

Інша мета команд – попередити про те, чи доступна дія. Продовжуючи приклад вирізування об'єкта або тексту, дія має сенс тільки тоді, коли щось виділене. Якщо користувач спробує вирізувати об'єкт або текст, який не виділений, то спроба буде невдалою. Щоб повідомити про це користувача, багато застосунків деактивують кнопки і пункти меню, т. т. роблять їх неактивними. Команда може вказати, чи можлива дія, реалізовуючи метод **CanExecute**. Кнопка може підписатися на подію **CanExecuteChanged** і бути відключеною, якщо **CanExecute** повертає значення **false**, або бути включеною, якщо **CanExecute** повертає значення **true**.

Семантика команди може бути однаковою для всіх застосунків і класів, але логіка дії специфічна для конкретного об'єкта, на який впливають. Комбінація клавіш **CTRL+X** викликає команду «Вирізати» в текстових класах, класах зображень і веб-браузерах, але фактична логіка виконання операції «Вирізати» визначається застосунком, який виконує вирізування. Клас **RoutedCommand** дозволяє клієнтам реалізувати логіку. Текстовий об'єкт може вирізувати виділений текст у буфер обміну, а об'єкт зображення може вирізувати виділене зображення. Коли застосунок обробляє подію **Executed**, він має доступ до мети команди і може зробити відповідні дії залежно від типу мети.

Отже, інфраструктура WPF підтримує *незалежний механізм від елементів управління подіями*, через *архітектуру команд*. Звичайна подія .NET визначається всередині деякого базового класу і може використовуватися тільки цим класом або його нащадками. Отже, стандартні події .NET тісно *прив'язані* до класу, в якому вони визначені.

Навпаки *команди* WPF схожі на події сутності, які не залежать від специфічного елемента управління і у багатьох випадках можуть успішно застосовуватися до численних (і на вигляд незв'язаних) типів елементів управління. Ось лише декілька прикладів: WPF підтримує команди *копіювання*, *вирізування* і *вставки*, які можуть використовуватися в різноманітних елементах

інтерфейсу користувача (на зразок пунктів меню, кнопок панелі інструментів і спеціальних кнопок), а також клавіатурні комбінації (скажімо, <**Ctrl+C**> і <**Ctrl+V**>).

Тоді як інші інструментальні набори для побудови інтерфейсів користувача (на зразок Windows Forms) пропонують для таких цілей стандартні події, їх застосування, зазвичай, дає в результаті надмірний і важкий у супроводі код. Всередині моделі WPF як альтернативу можна використати команди. Підсумком зазвичай є компактніша і гнучкіша кодова база.

Чотири основні поняття в системі команд WPF

Модель команд WPF можна розділити на чотири основні поняття:

- **Команди.** Команда представляє *задачу* застосунка й стежить за тим, коли вона може бути виконана. Однак коду, який виконує задачу застосунка, команди насправді не містять.
- **Прив'язка команд.** Кожна *прив'язка* (**binding**) має на увазі з'єднання команди застосунка з логікою, яка має до неї відношення, відповідальною за обслуговування певної області інтерфейсу користувача. Такий факторизований дизайн дуже важливий, тому що та сама команда може використовуватися в декількох місцях у застосунку і мати в кожному з них різне призначення. Для забезпечення подібної поведінки саме й служать різні прив'язки однієї і тієї ж команди.
- **Джерела команд.** Джерело команди ініціює команду. Наприклад, і елемент керування **MenuItem**, і елемент керування **Button** можуть служити джерелами команд. Клацання на них у такому випадку буде спричиняти виконання прив'язаної команди.
- **Цільові об'єкти команд.** Цільовий об'єкт команди – це елемент, для якого призначена ця команда, тобто елемент, на якому вона виконується. Наприклад, команда **Paste** може *вставляти* текст в елемент **TextBox**, а команда **OpenFile** – *відображати* документ в елементі **DocumentViewer**. Цільовий об'єкт може бути важливий, а може бути й неважливий, що залежить від природи команди.

Серцем моделі команд WPF безумовно є інтерфейс **System.Windows.Input.ICommand** (інтерфейс **ICommand** (див. рис. 4.2), який визначає спосіб, у відповідності з яким працюють команди. Цей інтерфейс включає два методи і подію:

```
public interface ICommand
{
  // Подія виникає, коли відбуваються зміни, які впливають
  // на те, чи повинна виконуватися команда чи ні.
  event EventHandler CanExecuteChanged;
  // Визначає метод, який з'ясовує, чи може
  // команда виконуватися в її поточному стані.
  bool CanExecute(object parameter);
  // Визначає метод для виклику при звертання до команди.
```

```
    void Execute(object parameter);
}
```

Інфраструктура WPF постачається з багаточисельними вбудованими командами, кожну з яких можна асоціювати з певною клавіатурною комбінацією або іншими конструкціями (іноді використовується термін «жест»). З погляду програмування команда WPF – це будь-який об'єкт, який підтримує властивість **Command**, яка повертає об'єкт, що реалізовує показаний нижче інтерфейс **ICommand** – деталі за адресою https://docs.microsoft.com/en-us/dotnet/api/system.windows.input.icommand?view=net-6.0.

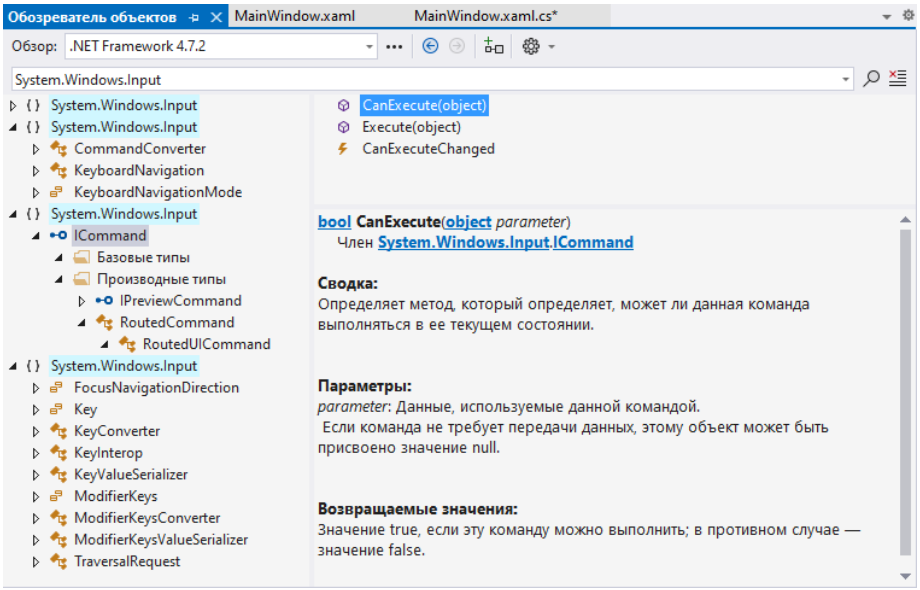


Рис. 4.2. Ілюстрація до **ICommand** в **Оглядачі об'єктів**

При створенні своїх власних команд реалізовувати інтерфейс **ICommand** прямо не обов'язково. Клас **System.Windows.Input.RoutedCommand**, який реалізує цей інтерфейс, автоматично можна використовувати замість інтерфейсу **ICommand**. Клас **RoutedCommand** є єдиним класом у WPF, який реалізує інтерфейс **ICommand**. Інакше кажучи, всі команди WPF є екземплярами класу **RoutedCommand** (або похідного від нього класу). Одна із ключових концепцій, яка лежать в основі моделі команд у WPF, полягає в тому, що клас **RoutedCommand** *не містить* ніякої логіки застосунку. Він просто представляє команду. Це означає, що один об'єкт **RoutedCommand** має такі самі можливості, як і інший. Якщо інтерфейс **ICommand** інкапсулює ідею команди – дії, яка може ініціюватися й бути або не бути доступною, то клас **RoutedCommand** змінює команду так, щоб вона могла подібно бульбашці підніматися нагору ієрархією елементів WPF до відповідного обробника подій.

Більшість команд, з якими доведеться мати справу, будуть не об'єктами **RoutedCommand**, а екземплярами класу **RoutedUICommand**, який

успадковується від класу **RoutedCommand**. Насправді всі заготовлені команди, які надає WPF є саме об'єктами **RoutedUICommand**.

Внутрішні об'єкти команд

У WPF пропонуються різноманітні класи команд, які відкривають доступ до приблизно сотні готових об'єктів команд. У цих класах визначені численні властивості, які представляють специфічні об'єкти команд, кожен з яких реалізує інтерфейс **ICommand**. У табл. 4.1 коротко описані вибрані стандартні об'єкти команд (детальніша інформація за адресою https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/commanding-overview?view=netframeworkdesktop-4.8.

Таблиця 4.1. Внутрішні об'єкти команд WPF

Клас WPF	Об'єкти команд	Опис
ApplicationCommands	Close, Copy, Cut, Delete, Find, Open, Paste, Save, SaveAs, Redo, Undo	Різноманітні команди рівня застосунку
ComponentCommands	MoveDown, MoveFocusBack, MoveLeft, MoveRight, ScrollToEnd, ScrollToHome	Різноманітні команди, які є загальними для компонентів інтерфейсу користувача
MediaCommands	BoostBase, ChannelUp, ChannelDown, FastForward, NextTrack, Play, Rewind, Select,Stop	Різноманітні команди, пов'язані з мультимедіа
NavigationCommands	BrowseBack, BrowseForward, Favorites, LastPage, NextPage, Zoom	Різноманітні команди, пов'язані з навігаційною моделлю WPF
EditingCommands	AlignCenter, CorrectSpellingError, DecreaseFontSize, EnterLineBreak, EnterParagraphBreak, MoveDownByLine, MoveRightByWord	Різноманітні команди, пов'язані з інтерфейсом Documents API у WPF

Під'єднання команд до властивості Command

Для під'єднання будь-якої властивості команд WPF до елемента інтерфейсу користувача, який підтримує властивість **Command** (на зразок **Button** або **MenuItem**), потрібно буде виконати зовсім незначну роботу. Як приклад модифікуємо поточну систему меню, додавши новий пункт верхнього рівня на ім'я **Edit** (Правка) з трьома піделементами (*піделемент* – наступний в ієрархії елемент меню), які дозволяють копіювати, вставляти і вирізувати текстові дані:

```
<Menu DockPanel.Dock="Top" HorizontalAlignment="Left" Background="White" BorderBrush ="Black">
  <MenuItem Header="_File"
            Click ="FileExit_Click" >
    <MenuItem Header ="_Exit"
              MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea"
              Click ="FileExit_Click"/>
    </MenuItem>
    <!-- Нові пункти меню з командами -->
    <MenuItem Header="_Edit">
      <MenuItem Command ="ApplicationCommands.Copy"/>
      <MenuItem Command ="ApplicationCommands.Cut"/>
```

```
      <MenuItem Command ="ApplicationCommands.Paste"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
                MouseEnter ="MouseEnterToolsHintsArea"
                MouseLeave ="MouseLeaveArea"
                Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Зверніть увагу, що властивості **Command** кожного піделемента в меню **Edit** присвоєно деяке значення. У результаті пункти меню автоматично отримують коректні імена і гарячі клавіші (зокрема, <**Ctrl+C**> – операція вирізування) в інтерфейсі користувача меню. Тепер застосунок зможе копіювати, вирізувати і вставляти текст *без написання* процедурного коду.

Запустивши застосунок і виділивши якусь частину тексту, нові пункти меню можна використати відразу ж. На додаток застосунок також оснащений можливістю реагування на стандартну операцію клацання правою кнопкою миші, пропонуючи користувачеві ті ж самі пункти в контекстному меню.

Під'єднання команд до довільних дій

Якщо об'єкт команди треба під'єднати до певної події (специфічної для застосунку), то доведеться написати процедурний код. Завдання нескладне, але вимагає трохи більше логіки, чим можна бачити у XAML. Наприклад, нехай потрібно, щоб все вікно реагувало на натиснення клавіші <**F1**>, активізуючи асоційовану з ним довідкову систему. Також припустимо, що у файлі коду для головного вікна визначений новий метод на ім'я **SetF1CommandBinding()**, який викликається всередині конструктора після виклику **InitializeComponent()**:

```
public MainWindow()
{
  InitializeComponent();
  SetF1CommandBinding();
}
```

Метод **SetF1CommandBinding()** програмно створюватиме новий об'єкт **CommandBinding**, який можна застосовувати всякий раз, коли треба прив'язати об'єкт команди до заданого обробника подій. Конфігуруємо об'єкт **CommandBinding** для роботи з командою **ApplicationCommands.Help**, яка автоматично запускається після натиснення клавіші <**F1**>:

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding = new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

Більшість об'єктів **CommandBinding** оброблятимуть подію **CanExecute** (яка дозволяє вказати, чи активується команда для конкретної операції програми) і подію **Executed** (де можна визначити код, який має бути виконаний після того, як команда сталася). Додамо до нашого похідного від **Window** типу наступні обробники подій (формати методів регламентуються асоційованими делегатами):

```
private void CanHelpExecute(object sender, CanExecuteRoutedEventArgs e)
{
  // Якщо потрібно запобігти виконанню команди,
  // то можна встановити CanExecute у false.
  e.CanExecute = true;
}
private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Дивись, це не так складно. Просто напиши що-небудь!", "Довідка!");
}
```

У попередньому фрагменті коду метод **CanHelpExecute()** реалізований так, що довідка при натисненні <**F1**> завжди дозволена; це здійснюється шляхом повернення **true**. Проте якщо в деяких ситуаціях довідкова система відображатися не повинна, то потрібно зробити відповідну перевірку і повертати **false**. Наша «довідкова система», яка відображається всередині **HelpExecute()**, є всього лише звичайне вікно повідомлення. Тепер можна запустити застосунок. Після натиснення <**F1**> з'являється наше вікно повідомлення.

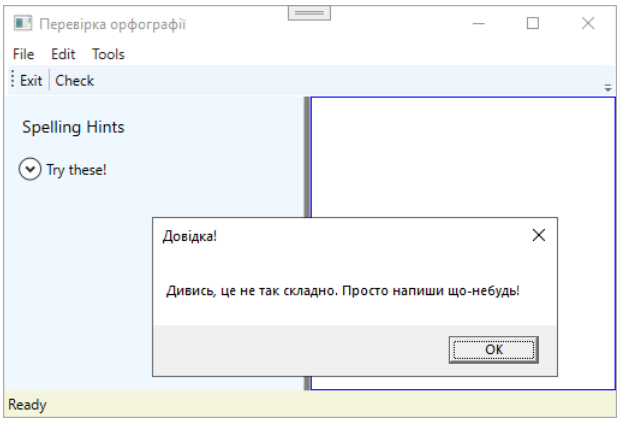


Рис. 4.3. Вікно з довідковою системою

Робота з командами Open і Save

Щоб завершити поточний приклад, ми додамо функціональність збереження текстових даних у зовнішньому файлі і відкриття файлів ***.txt** для редагування. Можна піти довшим шляхом, вручну додавши програмну логіку, яка включає і відключає пункти меню залежно від того, чи є дані всередині **TextBox**. Але ми звернемося до послуг команд для скорочення зусиль.

Розпочнемо з оновлення елемента **MenuItem**, який представляє меню **File** верхнього рівня, шляхом додавання двох нових підменю, які використовують об'єкти **Save** та **Open** класу **ApplicationCommands**:

```
<MenuItem Header="_File">
  <MenuItem Command ="ApplicationCommands.Open"/>
  <MenuItem Command ="ApplicationCommands.Save"/>
  <Separator/>
  <MenuItem Header ="_Exit"
      MouseEnter ="MouseEnterExitArea"
      MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
</MenuItem>
```

Згадайте – всі об'єкти команд реалізують інтерфейс **ICommand**, в якому визначені дві події (**CanExecute** та **Executed**). Тепер дозволимо вікну виконувати вказані команди, переконавшись, що це можна робити в поточних обставинах; отже, визначимо обробник події для запуску спеціального коду.

Знадобиться наповнити колекцію **CommandBindings**, підтримувану вікном. У розмітці XAML потрібно буде застосувати синтаксис "*властивість-елемент*" для визначення області **Window.CommandBindings**, в яку поміщаються два визначення **CommandBinding**. Модифікуємо визначення **Window**, як показано нижче:

```
<Window x:Class="MenuTest.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:MenuTest"
        mc:Ignorable="d"
        Title="MySpellChecker" Height="331" Width="508"
          WindowStartupLocation ="CenterScreen">
        <!-- Це інформує елемент управління Window про те, які
          обробники викликати при надходженні команд Open і Save -->
        <Window.CommandBindings>
            <CommandBinding Command="ApplicationCommands.Open"
                Executed="OpenCmdExecuted"
                CanExecute="OpenCmdCanExecute"/>
            <CommandBinding Command="ApplicationCommands.Save"
                Executed="SaveCmdExecuted"
                CanExecute="SaveCmdCanExecute"/>
        </Window.CommandBindings>
  <!-- Ця панель встановлює вміст вікна -->
  <DockPanel>
  …
  </DockPanel>
</Window>
```

Клацнемо правою кнопкою миші на кожному з атрибутів **Executed** і **CanExecute** в редакторі XAML і виберемо в контекстному меню пункт **Navigate**

to Event Handler (Перейти до обробника події). Як пояснювалося у ***Лекція 2. Побудова застосунків WPF з використанням Visual Studio*** автоматично згенерується шаблон коду для обробника події. Тепер у файлі коду C# для вікна мають з'явитися чотири порожні обробники подій.

Реалізація обробників події **CanExecute** повідомлятимуть вікно, що можна ініціювати відповідні події **Executed** у будь-який момент, для чого властивість **CanExecute** вхідного об'єкта **CanExecuteRoutedEventArgs** встановлюється в **true**:

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
private void SaveCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = true;
}
```

Обробники відповідної події **Executed** виконують роботу з відображення діалогових вікон, відкриття і збереження файла; вони також відправляють дані з **TextBox** у файл. Спочатку імпортуємо простори імен **System.IO** і **Microsoft.Win32** у файл коду. Остаточний код наступний:

```
private void OpenCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
  // Створити діалогове вікно відкриття файла і показати в ньому тільки текстові файли.
  var openDlg = new OpenFileDialog { Filter = "Text Files | *.txt" };
  // Чи було здійснено клацання на кнопці ОК?
  if (true == openDlg.ShowDialog())
  {
    // Завантажити вміст вибраного файла.
    string dataFromFile = File.ReadAllText(openDlg.FileName);
    // Відобразити рядок в TextBox.
    txtData.Text = dataFromFile;
  }
}
private void SaveCmdExecuted(object sender, ExecutedRoutedEventArgs e)
{
  var saveDlg = new SaveFileDialog { Filter = "Text Files | *.txt" };
  // Чи було здійснено клацання на кнопці ОК?
  if (true == saveDlg.ShowDialog())
  {
    // Зберегти дані із TextBox у заданому файлі.
    File.WriteAllText(saveDlg.FileName, txtData.Text);
  }
}
```

На замітку! Систему команд WPF детальніше розглянемо пізніше, де створюватимемо спеціальні команди на основі **ICommand** і **RelayCommands**.

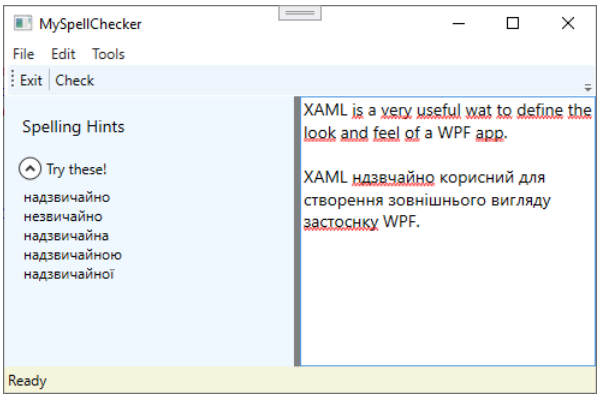


Рис. 4.4. Приклад застосунку **MyWordPadModif**

Отже, приклад і початкове знайомство з елементами управління WPF завершені. Ви довідалися, як працювати з базовими командами, системами меню, рядками стану, панелями інструментів, вкладеними панелями і декількома основними елементами інтерфейсу користувача (на зразок **TextBox** та **Expander**). Наступний приклад буде мати справу з більш екзотичними елементами управління, а також з рядом важливих служб WPF.

Резюме

Ми побудували простий застосунок – текстовий процесор. У ньому демонструвалося використання інтегрованої у WPF функціональності перевірки правопису, а також створення головного вікна з системою меню, рядком стану і панеллю інструментів.

Контрольні питання

1. Для чого використовуються тригери при побудові інтерфейсу користувача?
2. Що таке віджет?
3. З використанням якого класу будується система меню у WPF?
4. Який клас підтримує колекцію об’єктів MenuItem?
5. Для чого використовується елемент управління Separator?
6. У чому суть альтернативного способу активізації пунктів меню?
7. Для чого використовується елемент управління ToolBar?
8. Для чого використовується елемент управління StatusBar?
9. Яке функціональне призначення елемента управління Expander?
10. Як реалізовується функція перевірки правопису у WPF?
11. Які цілі досягаються за рахунок використання команд WPF?
12. Що собою представляє команда?
13. Що розуміється під поняттям прив’язка команд?
14. Що розуміється під поняттям джерела команд?
15. Що розуміється під поняттям цільові об’єкти команд?

16. Яку роль відіграє інтерфейс System.Windows.Input.ICommand (ICommand)? Що він визначає?
17. Назвіть класи WPF в яких зосереджені внутрішні об’єкти команд, які реалізують інтерфейс ICommand.
18. Як здійснюється під’єднання команд до властивості Command?

Лекція 5. Події у WPF

У ***Лекція 4. Побудова вікон*** ми конструювали складніші графічні інтерфейси, використовуючи деякі нові елементи управління і диспетчери компонування. У цій лекції розглянемо питання щодо подій та їх обробки.

Поняття маршрутизованих подій

Ви могли помітити, що у прикладі коду з ***Лекція 4. Побудова вікон*** передавався параметр **RoutedEventArgs**, а не **EventArgs**. Модель маршрутизованих подій, є удосконаленням стандартної моделі подій CLR і спроектована для того, щоб забезпечити можливість обробки подій у спосіб, який підходить для опису XAML-дерева об'єктів. Створимо новий проект застосунку WPF з іменем **WPFRoutedEvents**. Модифікуємо опис XAML початкового вікна, додавши наступний елемент управління **Button**, який визначає *складний* вміст:

```
<Button Name="btnClickMe" Height="75" Width = "250" Click ="btnClickMe_Clicked">
  <StackPanel Orientation="Horizontal">
    <Label Height="50" FontSize ="20">Незвичайна кнопка!</Label>
    <Canvas Height ="50" Width ="100" >
      <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
          Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
          Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

Зауважимо, що у відкриваючому визначенні **Button** обробляється подія **Click** за рахунок задання імені методу, який повинен викликатися при виникненні події. Подія **Click** працює з делегатом **RoutedEventHandler**, якого чекає обробник події, що приймає **object** у першому параметрі і **System.Winodws.RoutedEventArgs** у другому. Реалізовується такий обробник:

```
public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
  // Робити будь-що, коли на кнопці здійснили клацання.
  MessageBox.Show("Клацання кнопки");
}
```

Після запуску застосунку (рис. 5.1) вікно повідомлення відображатиметься незалежно від того, на якій частині вмісту кнопки було виконано клацання (зелений елемент **Ellipse**, жовтий елемент **Ellipse**, елемент **Label** або поверхня елемента **Button**). У принципі це нормально. Тільки уявіть, наскільки громіздкою виявилася б обробка подій WPF, якби довелося обробляти подію **Click** для кожного із згаданих піделементів. Справа не лише в тому, що створення окремих обробників подій для кожного компонента **Button** – трудомістке завдання, а ще і в тому, що у результаті ми отримали б складний у супроводі код.

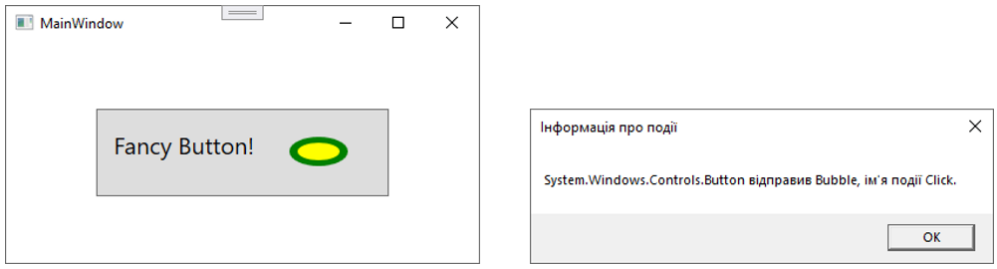


Рис. 5.1. Результат виконання частини застосунку **WPFRoutedEvents** з інформацією про подію

На щастя, *маршрутизовані* події WPF потурбуються про *автоматичний* виклик єдиного обробника події **Click** незалежно від того, на якій частині кнопки було здійснено клацання. Кажучи простіше, *модель* маршрутизованих подій, автоматично поширює подію *вгору* (або *вниз*) по дереву об'єктів (рис. 5.2) у пошуках відповідного обробника.

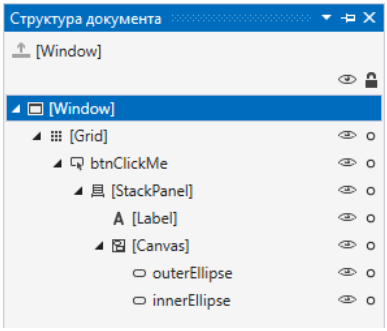


Рис. 5.2. Дерево об’єктів, отримане з використанням **Структура документа**

Точніше кажучи, маршрутизована подія може використати три стратегії маршрутизації.

1. Якщо подія переміщається від точки виникнення *вгору* до інших областей визначень всередині дерева об'єктів, то її називають *бульбашковою подією*.
2. І навпаки, якщо подія переміщається від самого зовнішнього елемента (наприклад, **Window**) *вниз* до точки виникнення, то її називають *тунельною подією*.
3. Нарешті, якщо подія ініціюється та обробляється тільки елементом, всередині якого вона виникла (так звана – нормальна подія CLR), то її називають *прямою подією*.

Роль маршрутизованих бульбашкових подій

У поточному прикладі, коли користувач клацає на внутрішньому овалі жовтого кольору, подія **Click** піднімається на наступний рівень області визначення (**Canvas**), потім на **StackPanel** і у результаті на рівень **Button**, де

обробляється. Точнісінько так же, якщо користувач клацає на **Label**, то подія спливає на рівень **StackPanel** і, врешті-решт, потрапляє в елемент **Button**.

Завдяки такому шаблону *маршрутизованих бульбашкових подій*, не потрібно турбуватися про реєстрацію специфічних обробників події **Click** для всіх членів складеного елемента управління. Проте якщо виникає потреба виконати спеціальну логіку обробки клацань для декількох елементів усередині того ж самого дерева об'єктів, то це цілком можна робити.

З метою ілюстрації припустимо, що клацання на елементі управління **outerEllipse** має бути оброблене в унікальний спосіб. Спочатку обробимо подію **MouseDown** для цього піделемента (типи, які графічно візуалізується, на зразок **Ellipse** не підтримують подію **Click**, але можуть відстежувати дії кнопки миші через події **MouseDown**, **MouseUp** і т. д.):

```
<Button Name="btnClickMe" Height="75" Width = "250" Click ="btnClickMe_Clicked">
  <StackPanel Orientation ="Horizontal">
    <Label Height="50" FontSize ="20">Fancy Button!</Label>
    <Canvas Height ="50" Width ="100" >
      <Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
          MouseDown="outerEllipse_MouseDown" Width ="50" Cursor="Hand"
            Canvas.Left="25" Canvas.Top="12"/>
      <Ellipse Name = "innerEllipse" Fill ="Yellow" Height = "15" Width ="36"
          Canvas.Top="17" Canvas.Left="32"/>
    </Canvas>
  </StackPanel>
</Button>
```

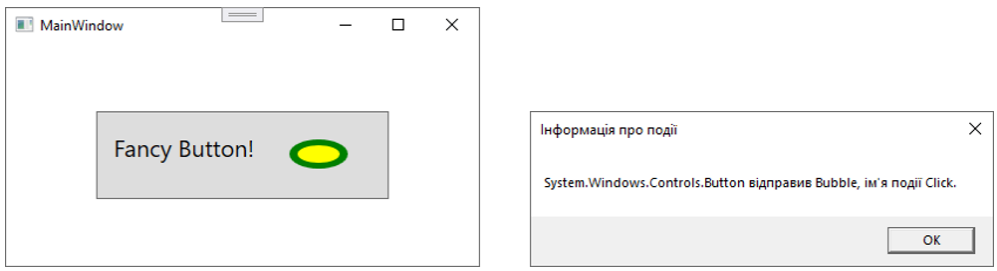


Рис. 5.3. Результат виконання частини застосунку **WPFRoutedEvents** з інформацією про подію **Click** оброблену *звичайним* способом

Потім реалізуємо відповідний обробник подій, який в демонстраційних цілях просто змінюватиме властивість **Title** головного вікна:

```
private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Змінити заголовок вікна.
  this.Title = "Ви натиснули на зовнішній еліпс!";
}
```

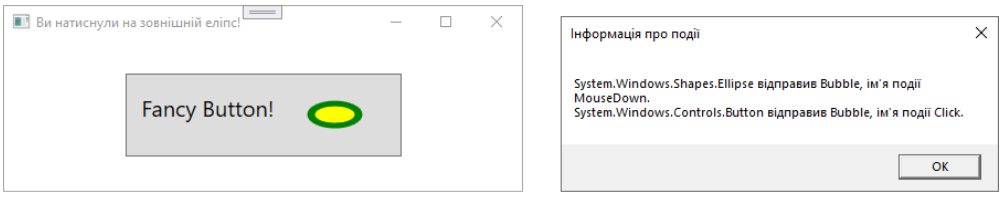


Рис. 5.4. Результат виконання частини застосунку **WPFRoutedEvents** з інформацією про подію **MouseDown** оброблену *незвичайним* способом при клацанні на зовнішньому еліпсі

Далі можна виконувати різні дії залежно від того, на якому елементі конкретно клацнув кінцевий користувач (на зовнішньому еліпсі або у будь-якому іншому місці всередині кнопки).

> **На замітку!** Маршрутизовані бульбашкові події завжди переміщаються з точки виникнення до *наступної визначальної області*. Отже, в розглянутому прикладі клацання на елементі **innerEllipse** привело б до потрапляння події в контейнер **Canvas**, а не в елемент **outerEllipse**, тому що обидва елементи є типами **Ellipse** всередині області визначення **Canvas**.

Продовження або припинення бульбашкового поширення

Наразі, коли користувач клацає на об'єкті **outerEllipse**, запускається зареєстрований обробник події **MouseDown** для цього об'єкта **Ellipse**, після чого подія спливе до події **Click** кнопки (див. рис. 5.4). Щоб інформувати WPF про потребу зупинки бульбашкового поширення деревом об'єктів, властивість **Handled** параметра **MouseButtonEventArgs** треба встановити в **true**:

```
private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Змінити заголовок вікна.
  this.Title = "Ви натиснули на зовнішній еліпс!";
  // Зупинити бульбашкове розповсюдження.
  e.Handled = true;
}
```

У такому випадку з’ясується, що заголовок вікна змінився, але вікно **MessageBox**, яке відображається обробником події **Click** елемента **Button**, не з'являється. По суті маршрутизовані бульбашкові події дозволяють складній групі вмісту діяти або як єдиний логічний елемент (наприклад, **Button**), або як окремі елементи (скажімо, **Ellipse** всередині **Button**).

Роль маршрутизованих тунельних подій

Строго кажучи, маршрутизовані події за своєю природою можуть бути *бульбашковими* (як було описано щойно) або *тунельними*. Тунельні події (імена яких розпочинаються з префікса **Preview** – на кшталт **PreviewMouseDown**) *спускаються* від самого верхнього елемента до внутрішніх областей визначення

дерева об'єктів. Загалом і в цілому для кожної бульбашкової події у бібліотеках базових класів WPF передбачена *пов'язана* тунельна подія, яка виникає перед її бульбашковим аналогом. Наприклад, перед виникненням бульбашкової події **MouseDown** спочатку ініціюється тунельна подія **PreviewMouseDown**.

Обробка тунельних подій виглядає дуже схожою на обробку будь-яких інших подій: треба просто вказати ім'я обробника події у розмітці XAML (або при потребі застосувати відповідний синтаксис обробки подій C# у файлі коду) і реалізувати такий обробник в коді. Для демонстрації взаємодії тунельних і бульбашкових подій розпочнемо з організації обробки події **PreviewMouseDown** для об'єкта **outerEllipse**:

```
<Ellipse Name = "outerEllipse" Fill ="Green" Height ="25"
    MouseDown ="outerEllipse_MouseDown"
    PreviewMouseDown ="outerEllipse_PreviewMouseDown"
    Width ="50" Cursor="Hand" Canvas.Left="25" Canvas.Top="12"/>
```

Після цього модифікуємо поточне визначення класу C#, відновивши обробники подій (для всіх об'єктів) за рахунок додавання даних про подію в змінну-член **mouseActivity** типу **string** з використанням вхідного об'єкта аргументів події. У результаті з'явиться можливість спостерігати за потоком подій, які з'являються у фоновому режимі.

```
public partial class MainWindow : Window
{
  string _mouseActivity = string.Empty;
  public MainWindow()
  {
    InitializeComponent();
  }
  private void btnClickMe_Clicked(object sender, RoutedEventArgs e)
  {
    AddEventInfo(sender, e);
    MessageBox.Show(_mouseActivity, "Інформація про вашу подію");
    // Очистити рядок для наступного циклу.
    _mouseActivity = "";
    // Робити будь-що, коли на кнопці здійснили клацання.
    // MessageBox.Show("Клацання кнопки");
  }
  private void AddEventInfo(object sender, RoutedEventArgs e)
  {
    _mouseActivity += string.Format(
    "{0} відправив {1}, ім'я події {2}.\n", sender,
    e.RoutedEvent.RoutingStrategy,
    e.RoutedEvent.Name);
  }
  private void outerEllipse_MouseDown(object sender, MouseButtonEventArgs e)
  {
    AddEventInfo(sender, e);
  }
  private void outerEllipse_PreviewMouseDown(object sender, MouseButtonEventArgs e)
```

```
  {
    AddEventInfo(sender, e);
  }
}
```

Зверніть увагу, що ні в одному обробнику подій бульбашкове поширення не зупиняється. Після запуску застосунку відобразиться вікно з унікальним повідомленням, яке залежить від місця на кнопці, де було здійснене клацання. На рис. 5.5 показаний результат клацання на зовнішньому об'єкті **Ellipse**.

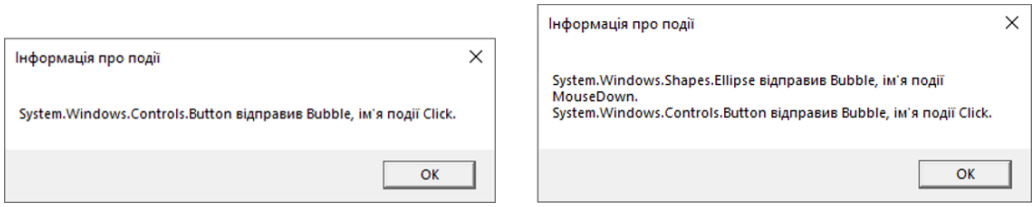


Рис. 5.5. Тунельна подія виникає першою, бульбашкова – другою

Отже, чому події WPF зазвичай зустрічаються парами (одна тунельна та одна бульбашкова)? Відповідь можна сформулювати так: завдяки попередньому перегляду подій з'являється можливість виконання будь-якої спеціальної логіки (перевірка достовірності даних, відключення бульбашкового поширення і т. п.) перед запуском бульбашкового аналога подій.

Як приклад, припустимо, що створюється елемент **TextBox**, який повинен містити тільки числові дані. У ньому можна було б обробити подію **PreviewKeyDown**; якщо з'ясовується, що користувач ввів нечислові дані, то бульбашкову подію легко відмінити, встановивши властивість **Handled** в **true**.

Як нескладно було припустити, при побудові спеціального елемента управління, який підтримує спеціальні події, подію дозволяється реалізувати так, щоб вона могла поширюватися бульбашковим (або тунельним) способом деревом розмітки XAML. У цій лекції ми не розглядаємо процес створення спеціальних маршрутизованих подій (хоча він не особливо відрізняється від побудови спеціальної властивості залежності). Детальніше можна ознайомитися за адресою https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/routed-events-overview?view=netframeworkdesktop-4.8 та https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/advanced/preview-events?view=netframeworkdesktop-4.8.

Глибший погляд на API-інтерфейси та елементи управління WPF

Побудуємо новий застосунок WPF із використанням Visual Studio. Мета – створення інтерфейсу користувача, який складається з віджета **TabControl**, що містить набір вкладок. Кожна вкладка ілюструватиме декілька нових елементів управління WPF і цікаві API-інтерфейси, які можуть бути задіяні в розроблюваних проектах. Попутно ви також довідаєтеся про додаткові можливості візуальних конструкторів WPF з Visual Studio.

Робота з елементом управління TabControl

Насамперед створимо новий проект застосунку WPF з іменем **WpfControlsAndAPIs**. Початкове вікно міститиме елемент управління **TabControl** з двома вкладками, кожна з яких відображає набір пов'язаних елементів управління і/або API-інтерфейсів WPF. Встановимо властивість **Width** вікна в **800**, а властивість **Height** вікна в **350**.

Перетягнемо елемент управління **TabControl** з панелі інструментів Visual Studio на поверхню візуального конструктора і обновимо його розмітку так:

```
<TabControl Name="MyTabControl" HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch">
   <TabItem Header="TabItem">
        <Grid Background="#FFE5E5E5"/>
   </TabItem>
   <TabItem Header="TabItem">
        <Grid Background="#FFE5E5E5"/>
   </TabItem>
</TabControl>
```

На рис. 5.6 зображено макет вікна згідно коду розмітки.

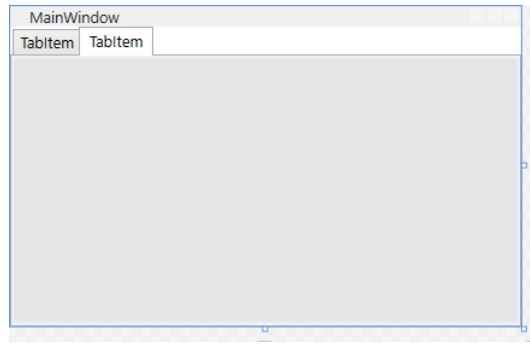


Рис. 5.6. Макет вікна згідно попереднього коду розмітки

Ми бачимо, що два елементи типу вкладок створюються автоматично. Щоб додати додаткові вкладки, потрібно клацнути правою кнопкою миші на вузлі **TabControl** у вікні **Document Outline** та вибрати у контекстному меню пункт **Add TabItem** (Додати **TabItem**). Можна також клацнути правою кнопкою миші елемент **TabControl** у візуальному конструкторі і вибрати той же пункт меню або просто ввести розмітку у редакторі XAML. Дослідіть описані способи створення додаткових вкладок самостійно.

Обновимо розмітку кожного елемента керування **TabItem** у редакторі XAML і змінимо його властивість **Header**, вказуючи **Ink API**, **Data Binding**. Вікно візуального конструктора має виглядати так як показано на рис. 5.7.

Майте на увазі, що вкладка, яка вибрано для редагування, стає активною, та її вміст можна формувати, перетягуючи елементи керування з панелі інструментів. Маючи в своєму розпорядженні визначення основного елемента

управління **TabControl**, можна опрацювати деталі кожної вкладки, одночасно вивчаючи додаткові засоби API-інтерфейсу WPF.

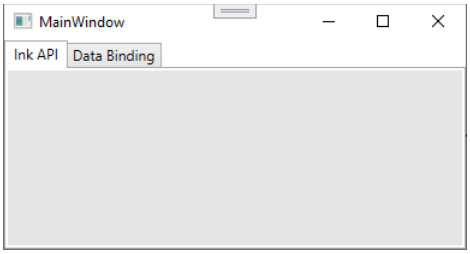


Рис. 5.7. Макет початкового компонування системи вкладок

Побудова вкладки Ink API

Перша вкладка, якою ми наразі займемося, призначена для розкриття загальної ролі інтерфейсу **Ink API**, який дозволяє легко вбудовувати в програму функціональність рисування. Звичайно, його використання не обмежується програмами для рисування; **Ink API** можна використовувати з різноманітними цілями, включаючи фіксацію рукописного вводу.

> **На замітку!** Надалі замість використання різноманітних вікон головним чином безпосередньо редагуватиметься розмітка XAML. Хоча процедура перетягування елементів управління працює нормально, найчастіше компонування виявляється небажаним (Visual Studio додає межі та заповнення на основі того, де розміщений елемент), а тому доводиться витрачати значний час на очищення розмітки XAML.

Почнемо із заміни дескриптора **Grid** в елементі керування **TabItem**, поміченому як **Ink API**, дескриптором **StackPanel** і додавання закриваючого дескриптора. Розмітка повинна мати такий вигляд:

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">

  </StackPanel>
</TabItem>
```

Проектування панелі інструментів

Додамо (використовуючи редактор XAML) новий елемент керування **ToolBar** на ім'я **InkToolbar** з властивістю **Height**, встановленою в **60**:

```
<TabItem Header="Ink API">
  <StackPanel Background="#FFE5E5E5">
    <ToolBar Name="InkToolBar" Height="60">
    </ToolBar>
  </StackPanel>
</TabItem>
```

Додамо три елементи керування **RadioButton** у панель **WrapPanel** всередині елемента управління **Border**:

```
<ToolBar Name="InkToolBar" Height="60">
  <Border Margin="0,2,0,2.4" Width="280" VerticalAlignment="Center">
    <WrapPanel>
      <RadioButton x:Name="inkRadio" Margin="5,10" Content="Ink Mode!" IsChecked="True"
          GroupName="InkMode" Click="RadioButtonClicked"/>
      <RadioButton x:Name="eraseRadio" Margin="5,10" Content="Erase Mode!"
          GroupName="InkMode" Click="RadioButtonClicked"/>
      <RadioButton x:Name="selectRadio" Margin="5,10" Content="Select Mode!"
          GroupName="InkMode" Click="RadioButtonClicked" />
    </WrapPanel>
  </Border>
</ToolBar>
```

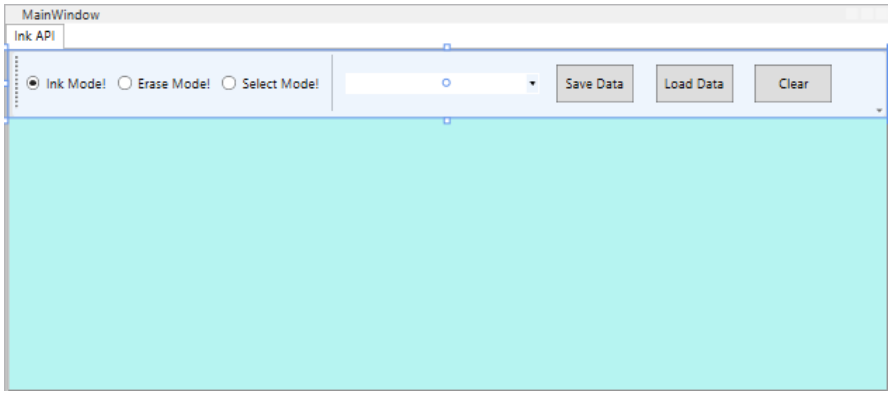


Рис. 5.8. Черговий макет вікна після додавання **RadioButton**

Коли елемент керування **RadioButton** поміщається не всередину батьківської панелі, він отримує інтерфейс користувача, ідентичний інтерфейсу користувача елемента управління **Button**. Саме тому елементи керування **RadioButton** були запаковані у панель **WrapPanel**. Далі додамо елемент **Separator** та елемент **ComboBox**, властивість **Width** якого встановлена в **175**, а властивість **Margin** – в **10**, **0**, **0**, **0**. Додамо три дескриптори **ComboBoxItem** з вмістом **Red**, **Green** і **Blue**, і додамо на весь елемент управління **ComboBox** ще один елемент **Separator**:

```
<ToolBar Name="InkToolBar" Height="60">
  <Border Margin="0,2,0,2.4" Width="520" VerticalAlignment="Center">
    <WrapPanel >
.....
      <Separator/>
      <ComboBox x:Name="comboColors" Width="175" Margin="10,0,0,0"
          SelectionChanged="comboColors_SelectionChanged">
        <ComboBoxItem Content="Red"/>
        <ComboBoxItem Content="Green"/>
        <ComboBoxItem Content="Blue"/>
      </ComboBox>
```

```
      <Separator/>
    </WrapPanel>
  </Border>
</ToolBar>
```

Елемент управління RadioButton

У цьому прикладі потрібно, щоб три додані елементи управління **RadioButton** були взаємовиключними. В інших інфраструктурах для побудови графічних інтерфейсів такі пов'язані елементи вимагають поміщення в одну групову рамку. Робити так у WPF не обов'язково. Натомість елементам управління просто призначається те ж саме групове ім'я, що дуже зручно, бо пов'язані елементи не повинні фізично розміщатися всередині однієї області, а можуть розташовуватися будь-де у вікні.

Клас **RadioButton** має властивість **IsChecked**, значення якої перемикаються між **true** і **false**, коли кінцевий користувач клацає на елементі інтерфейсу користувач. До того ж елемент управління **RadioButton** надає дві події (**Checked** та **Unchecked**), які можна використати для перехоплення такої зміни стану.

Додавання кнопок збереження, завантаження та видалення

Фінальним елементом управління всередині **ToolBar** буде **Grid**, з трьом елементами управління **Button**. Додамо наступну розмітку після останнього елемента управління **Separator**:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Button Grid.Column="0" x:Name="btnSave" Margin="10,10" Width="70"
      Content="Save Data"/>
  <Button Grid.Column="1" x:Name="btnLoad" Margin="10,10" Width="70"
      Content="Load Data"/>
  <Button Grid.Column="2" x:Name="btnClear" Margin="10,10" Width="70" Content="Clear"/>
</Grid>
```

Додавання елемента керування InkCanvas

Фінальним елементом **TabControl** є **InkCanvas**. Помістимо показану нижче розмітку після закриваючого дескриптора **ToolBar**, але перед закриваючим дескриптором **StackPanel**:

```
<InkCanvas x:Name="MyInkCanvas" Background="#FFB6F4F1" />
```

Попередній перегляд вікна

Тепер все готове до тестування програми. Повинні відобразитися три взаємовиключні перемикачі, розкривний список з трьома елементами і три кнопки (рис. 5.9).

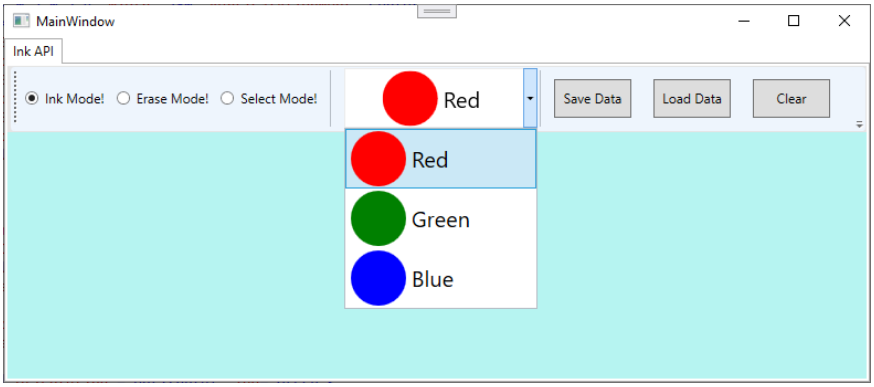


Рис. 5.9. Завершене компонування вкладки **Ink API**

Обробка подій для вкладки Ink API

Наступне завдання для вкладки **Ink API** пов'язане з організацією обробки події **Click** для кожного елемента керування **RadioButton**. Як ми робили в інших проектах WPF, потрібно просто клацнути на значку із зображенням блискавки у вікні **Properties** середовища Visual Studio та ввести імена обробників подій. За допомогою такого прийому зв'яжемо подію **Click** кожного елемента управління **RadioButton** з тим же самим обробником на ім'я **RadioButtonClicked**. Після обробки всіх трьох подій **Click** обробимо подію **SelectionChanged** елемента управління **ComboBox**, використовуючи обробник на ім'я **ColorChanged**. У результаті отримаємо наступний код:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    // Вставити тут код, потрібний при створенні об'єкта.
  }
  private void comboColors_SelectionChanged(object sender, SelectionChangedEventArgs e)
  {

  }
  private void RadioButtonClicked(object sender, RoutedEventArgs e)
  {
    // TODO: Додати сюди реалізацію обробника подій.
  }
  private void ColorChanged(object sender, SelectionChangedEventArgs e)
  {
    // TODO: Додати сюди реалізацію обробника подій.
  }
}
```

Обробники подій будуть реалізовані пізніше, так що залишимо їх поки що порожніми.

Додавання елементів керування до панелі інструментів

Елемент керування **InkCanvas** додається шляхом прямого редагування розмітки XAML. Зважайте на те, що панель інструментів Visual Studio за замовчуванням не відображає всі можливі компоненти WPF, але вміст панелі інструментів можна оновлювати.

Клацніть правою кнопкою миші десь в області панелі елементів і виберіть в контекстному меню пункт **Choose Items** (Вибрати елементи). Незабаром з'явиться список можливих компонентів для додавання до панелі інструментів. Нас цікавить додавання елемента керування **InkCanvas** (рис. 5.10). Після цього клацнути двічі на елементі **InkCanvas**. Подивіться на код розмітки і ви переконаєтеся, що він дійсно з’явився.

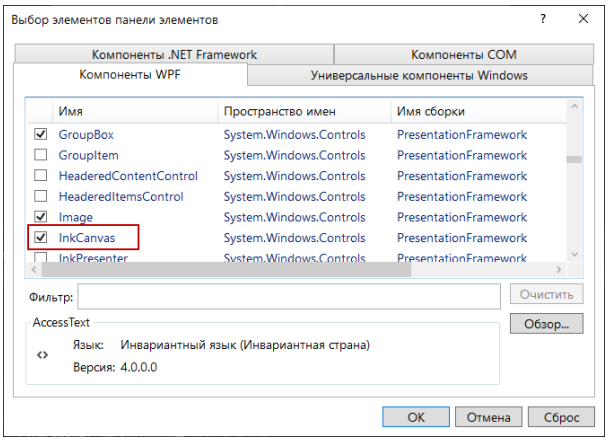


Рис. 5.10. Додавання нових компонентів до панелі інструментів Visual Studio

Елемент управління InkCanvas

Просте додавання **InkCanvas** дозволяє малювати у вікні. Малювати можна з допомогою миші або за наявності пристрою, який сприймає дотики пальця або цифрового пера. Запустимо програму і намалюємо щось (рис. 5.11).

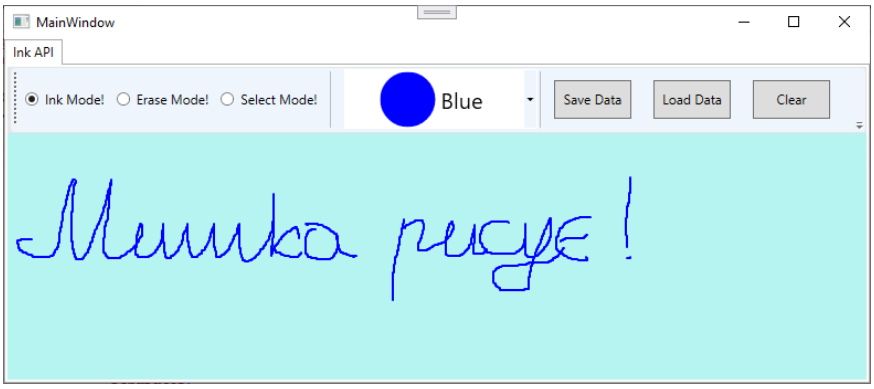


Рис. 5.11. Елемент управління **InkCanvas** у дії

Елемент управління **InkCanvas** забезпечує щось більше ніж просто малювання штрихів за допомогою миші (або пера); він також підтримує декілька унікальних режимів редагування, керованих властивістю **EditingMode**, якій можна присвоїти будь-яке значення з пов'язаного перерахування **InkCanvasEditingMode**. У цьому прикладі нас цікавить режим:

- **Ink**, прийнятий за замовчуванням, який щойно демонструвався;
- **Select**, який дозволяє користувачеві вибирати за допомогою миші область для переміщення або зміни розміру;
- **EraseByStroke**, який видаляє попередній штрих миші.

На замітку! Штрих – це візуалізація, яка відбувається під час одиночної операції натискання та відпускання кнопки миші. Елемент керування **InkCanvas** зберігає всі штрихи в об'єкті **StrokeCollection**, який доступний при використанні властивості **Strokes**.

Оновимо обробник **RadioButtonClicked()** наступною логікою, яка поміщає **InkCanvas** в коректний режим залежно від вибраного перемикача **RadioButton**:

```
private void RadioButtonClicked(object sender, RoutedEventArgs e)
{
  // Залежно від того, яка кнопка відправила подію,
  // помістити InkCanvas в потрібний режим оперування.
  switch ((sender as RadioButton)?.Content.ToString())
  {
    // Ці рядки повинні співпадати зі значеннями властивості Content
    // кожного елемента RadioButton.
    case "Ink Mode!":
      this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
      break;
    case "Erase Mode!":
      this.MyInkCanvas.EditingMode = InkCanvasEditingMode.EraseByStroke;
      break;
    case "Select Mode!":
      this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Select;
      break;
  }
}
```

На додачу встановимо **Ink** як стандартний режим у конструкторі вікна. Там же встановимо стандартний вибір для **ComboBox** (про **ComboBox** далі):

```
public MainWindow()
{
  InitializeComponent();
  // Встановити режим Ink як стандартний.
  this.MyInkCanvas.EditingMode = InkCanvasEditingMode.Ink;
  this.inkRadio.IsChecked = true;
  this.comboColors.SelectedIndex = 0;
}
```

Тепер запустимо програму ще раз, натиснувши <**F5**>. Увійдемо в режим **Ink** і нарисуємо щось. Потім перейдемо в режим **Erase** і зітремо раніше намальоване (курсор миші автоматично набуде вигляд ластика). Нарешті, перейдемо в режим **Select** і виберемо кілька ліній, використовуючи мишу як ласо.

Охопивши елемент, його можна переміщати поверхнею полотна, а також змінювати розміри. На рис. 5.12 демонструються різні режими дії.

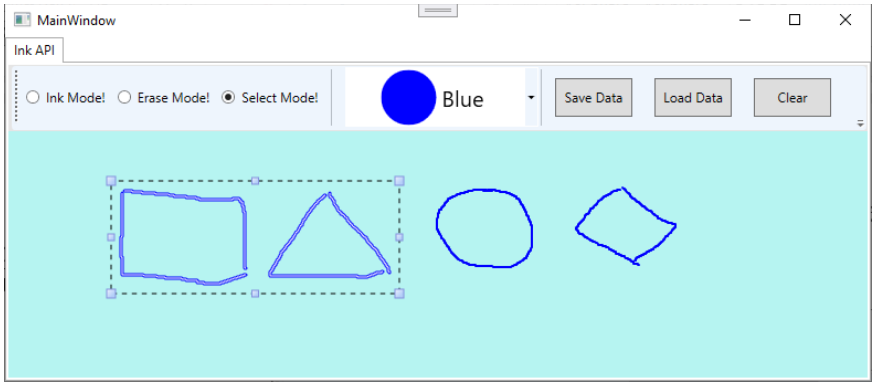


Рис. 5.12. Різні режими редагування елемента **InkCanvas** у дії

Елемент управління ComboBox

Після заповнення елемента управління **ComboBox** (або **ListBox**) є три способи визначення вибраного в них елемента:

1. По-перше, коли потрібно знайти числовий індекс вибраного елемента, повинна застосовуватися властивість **SelectedIndex** (відлік починається з нуля; значення **-1** це відсутність вибору).
2. По-друге, якщо потрібно отримати об'єкт, вибраний всередині списка, то згодиться властивість **SelectedItem**.
3. По-третє, властивість **SelectedValue** дозволяє отримати значення вибраного об'єкта (зазвичай, за допомогою виклику **ToString()**).

Останній фрагмент коду, який потрібно додати для цієї вкладки відповідає за зміну кольору штрихів, намальованих в **InkCanvas**. Властивість **DefaultDrawingAttributes** елемента **InkCanvas** повертає об'єкт **DrawingAttributes**, який дозволяє конфігурувати численні атрибути пера, включаючи його розмір та колір (крім інших налаштувань). Модифікуємо код C# наступною реалізацією методу **ColorChanged()**:

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
  // Отримати вибраний елемент в розкривному списку.
  string colorToUse = (this.comboColors.SelectedItem as ComboBoxItem)?.Content.ToString();
  // Змінити колір, використовуваний для візуалізації штрихів.
  this.MyInkCanvas.DefaultDrawingAttributes.Color =
(Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Згадайте, що **ComboBox** містить колекцію **ComboBoxItems**. Попередньо згенеровану розмітку XAML незначно змінимо:

```
<ComboBox x:Name="comboColors" Width="145" SelectionChanged="ColorsChanged">
        <ComboBoxItem Content="Red"/>
        <ComboBoxItem Content="Green"/>
     <ComboBoxItem Content="Blue"/>
</ComboBox>
```

У результаті звернення до властивості **SelectedItem** отримаємо вибраний елемент **ComboBoxItem**, який зберігається як екземпляр загального типу **Object**. Після приведення **Object** до **ComboBoxItem** отримується значення **Content**, яке буде рядком **Red**, **Green** або **Blue**. Цей рядок потім перетвориться в об'єкт **Color** із застосуванням зручного службового класу **ColorConverter**. Знову запустимо програму. Тепер повинна з'явитися можливість перемикання між кольорами при візуалізації зображення.

Зверніть увагу, що елементи управління **ComboBox** і **ListBox** також можуть мати складний вміст, а не тільки список текстових даних. Щоб отримати уявлення про деякі можливості, відкриємо редактор XAML для вікна і змінимо визначення елемента управління **ComboBox**, помістивши в нього набір елементів **StackPanel**, кожен з яких містить **Ellipse** і **Label** (властивість **Width** елементу **ComboBox** встановлене в **175**):

```
<ComboBox x:Name="comboColors" Width="145" SelectionChanged="ColorChanged">
  <StackPanel Orientation ="Horizontal" Tag="Red">
      <Ellipse Fill ="Red" Height ="55" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center" VerticalAlignment="Center"
          Content="Red"/>
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag="Green">
      <Ellipse Fill ="Green" Height ="55" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center" VerticalAlignment="Center"
          Content="Green"/>
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag="Blue">
      <Ellipse Fill ="Blue" Height ="55" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center" VerticalAlignment="Center"
          Content="Blue"/>
  </StackPanel>
</ComboBox>
```

У визначенні кожного елемента **StackPanel** присвоюється значення властивості **Tag**, що є швидким і зручним способом з’ясування, який стек елементів був вибраний користувачем (для цього існують і кращі способи, але поки вистачить такого). Із заданою поправкою потрібно змінити реалізацію методу **ColorChanged()**:

```
private void ColorChanged(object sender, SelectionChangedEventArgs e)
{
  // Отримати властивість Tag вибраного елемента StackPanel.
```

```
  string colorToUse = (this.comboColors.SelectedItem as StackPanel).Tag.ToString();

  // Змінити колір, використовуваний для візуалізації штрихів.
  this.MyInkCanvas.DefaultDrawingAttributes.Color                                    =
(Color)ColorConverter.ConvertFromString(colorToUse);
}
```

Після запуску програми елемент управління **ComboBox** виглядатиме так, як показано на рис. 5.13.

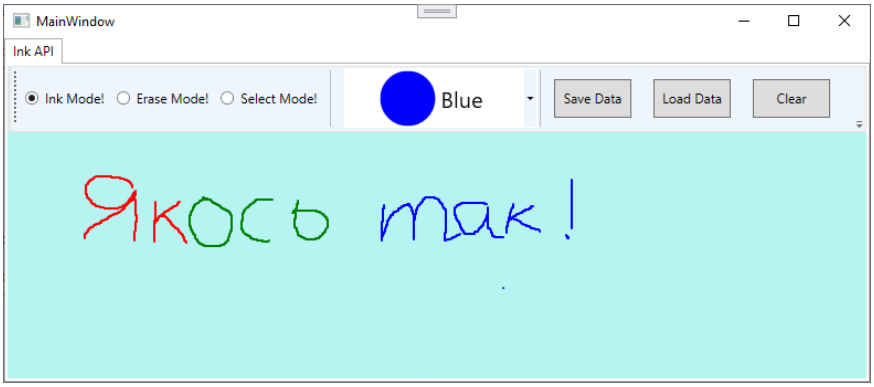


Рис. 5.13. Спеціальний елемент **ComboBox**,
реалізований за допомогою моделі вмісту WPF

Збереження, завантаження та очищення даних InkCanvas

Остання частина вкладки **Ink API** дозволить зберігати і завантажувати дані контейнера **InkCanvas**, а також очищати його вміст, додаючи обробники подій для кнопок в панелі інструментів. Потрібно додати наступні простори імен:

- using System.Windows.Ink;
- using System.IO;

Тепер модифікуємо розмітку XAML для кнопок за рахунок додавання розмітки, яка відповідає за події клацань:

```
private void SaveData(object sender, RoutedEventArgs e)
{
  // Зберегти всі дані InkCanvas в локальному файлі.
  using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Create))
  {
    this.MyInkCanvas.Strokes.Save(fs);
    fs.Close();
  }
}
private void LoadData(object sender, RoutedEventArgs e)
{
  // Наповнити StrokeCollection з файлf.
  using (FileStream fs = new FileStream("StrokeData.bin", FileMode.Open, FileAccess.Read))
```

```
  {
    StrokeCollection strokes = new StrokeCollection(fs);
    this.MyInkCanvas.Strokes = strokes;
  }
}
private void Clear(object sender, RoutedEventArgs e)
{
  // Очистити всі штрихи.
  this.MyInkCanvas.Strokes.Clear();
}
```

Тепер повинна з'явитися можливість збереження даних у файлі, завантаження з файла та очищення **InkCanvas** від усіх даних. Отже, робота з першою вкладкою елемента управління **TabControl** завершена, так само як і дослідження інтерфейсу **Ink API**. Звичайно, про технологію **Ink API** можна розповісти ще багато чого, але тепер ви маєте достатньо знань, аби продовжити вивчення теми самостійно. Далі ви довідаєтеся, як застосовувати прив'язку даних WPF.

Резюме

У цій лекції ви поповнили свої знання новими відомостями про побудову інтерфейсів користувача у XAML і попутно ознайомилися з інтерфейсом Ink API, пропонованим WPF.

Нарешті, ви з'ясували, що інфраструктура WPF додає унікальну можливість до традиційних програмних примітивів .NET, зокрема до властивостей і подій. Як попутне зауваження: механізм маршрутизованих подій надає події спосіб поширюватися вгору або вниз деревом розмітки.

Контрольні питання

1. Розкрийте суть поняття маршрутизовані події?
2. Що ви розумієте під поняттям нормальної події CLR?
3. Скільки стратегій маршрутизації може використати маршрутизована подія? Розкрийте їх суть.
4. Що таке маршрутизована бульбашкова подія?
5. Чи можна продовжити або припинити бульбашкове поширення?
6. Що таке маршрутизована тунельна подія?
7. Як формуються імена маршрутизованих тунельних подій?
8. Як пов’язані між собою бульбашкові і тунельні події?
9. Для чого використовується елемент управління TabControl?
10. З якою метою використовується інтерфейс Ink API?
11. Для чого використовується елемент управління RadioButton?
12. Для чого призначений елемент управління InkCanvas і як його додати до панелі інструментів?
13. Для чого призначений елемент управління ComboBox?

Лекція 6. Модель прив’язки даних WPF

Вступ в модель прив'язки даних WPF

Елементи управління часто служать ціллю для різноманітних операцій *прив'язки даних*. Кажучи простіше, *прив'язка даних* є дією з підключення властивостей елемента управління до значень даних, які можуть змінюватися упродовж життєвого циклу застосунку. Це дозволяє елементу інтерфейсу користувача відображати стан змінної в коді. Наприклад, прив'язку даних можна використати для вирішення наступних завдань:

- позначати прапорець елемента управління **CheckBox** на основі булевої властивості заданого об'єкта;
- відображати в елементах **TextBox** інформацію, отриману з реляційної бази даних;
- підключати елемент **Label** до цілого числа, яке представляє число файлів у теці.

При роботі із вбудованим механізмом прив'язки даних WPF важливо пам'ятати про *різницю* між *джерелом* і місцем *призначення* операції прив'язки. Як і можна було очікувати, *джерелом* операції прив'язки даних є самі дані (булева властивість, реляційні дані і т. д.), а місцем *призначення* (або метою) – властивість елемента управління інтерфейсу користувача, в якому задіюється *вміст* даних (на зразок властивості елемента управління **CheckBox** або **TextBox**).

Простий сценарій прив'язки даних має на увазі ситуацію, коли *початковий об'єкт* – елемент WPF, а *початкова властивість* – властивість залежності. Причина в тому, що властивість залежності має вбудовану підтримку повідомлень про зміни. У результаті, коли *значення властивості залежності змінюється* у початковому об'єкті, *прив'язана властивість цільового об'єкта негайно оновлюється*. Це саме те, що вимагається, і відбувається воно без потреби побудови будь-якої додаткової інфраструктури.

На додаток до прив'язки традиційних даних інфраструктура WPF робить можливою прив'язку елементів, як стверджувалося в попередніх прикладах. Це значить, що можна прив'язати (скажімо) видимість властивості до властивості стану позначки прапорця. Така дія була безперечно можливою у Windows Forms, але вимагала реалізації через код. Інфраструктура WPF пропонує розвинену систему прив'язки даних, яка здатна майже цілком підтримуватися в розмітці. Вона також дозволяє забезпечувати *синхронізацію* джерела і *цілі* у разі зміни значень даних.

Побудова вкладки Data Binding

Використаємо проект **WpfControlsAndAPIs** з попередньої лекції – ***Лекція 05. Події у WPF***. У вікні **Document Outline** замінимо елемент управління **Grid** у другій вкладці (**Data Binding**) панеллю **StackPanel**. Із використанням панелі елементів і вікна **Properties** середовища Visual Studio створимо наступне початкове компонування:

```
<TabItem x:Name="tabDataBinding" Header="Data Binding">
  <StackPanel Width="250">
    <Label Content="Перемістіть полосу прокручування, щоб побачити поточне значення"/>
    <!-- Значення лінійки прокручування є джерелом цього прив'язування даних -->
    <ScrollBar x:Name="mySB" Orientation="Horizontal" Height="30"
        Minimum = "1" Maximum = "100" LargeChange="1" SmallChange="1"/>
    <!-- Вміст мітки буде прив'язаний до лінійки прокручування -->
    <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
        BorderThickness="2" Content = "0"/>
  </StackPanel>
</TabItem>
```

Зверніть увагу, що об'єкт **ScrollBar** (названий тут **mySB**) конфігурований з діапазоном від **1** до **100**. Мета полягає в тому, щоб при зміні положення повзунка лінійки прокручування (або по клацанню на стрілці вліво або вправо) елемент **Label** автоматично оновлювався поточним значенням повзунка. Зараз значення властивості **Content** елемента управління **Label** встановлено в «**0**»; проте, ми змінимо його за допомогою операції прив'язки даних.

Встановлення прив'язки даних

Механізмом, який забезпечує визначення прив'язки у розмітці XAML, є розширення розмітки **{Binding}**. Хоча прив'язку можна визначати використовуючи Visual Studio, це не складно зробити прямо в розмітці. Відредагуємо розмітку XAML властивості **Content** елемента **Label** на ім'я **labelSBThumb** так:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "{Binding Path=Value, ElementName=mySB}"/>
```

Зверніть увагу на значення, присвоєне властивості **Content** елемента **Label**. Конструкція **{Binding}** означає *операцію прив'язки* даних. Значення **ElementName** представляє *джерело* операції прив'язки даних (об'єкт **ScrollBar**), а **Path** вказує *властивість*, до якої *здійснюється прив'язка* (властивість **Value** лінійки прокручування).

Запустивши програму знову, можна виявити, що вміст мітки оновлюється на основі значення лінійки прокручування у міру переміщення повзунка.

Властивість DataContext

Для задання операції прив'язки даних у XAML може використовуватися альтернативний формат, при якому допускається *розбивати* значення, вказані розширенням розмітки **{Binding}**, за рахунок явного встановлення властивості **DataContext** в джерело операції прив'язки:

```
<!-- Розбиття об'єкта і значення шляхом використання DataContext -->
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
      DataContext="{Binding ElementName=mySB}"
      Content="{Binding Path=Value}" />
```

З використанням поточного фрагменту вивід буде ідентичним попередньому. З врахуванням цього цілком імовірно вас цікавить, в яких випадках потрібно встановлювати властивість **DataContext** явно. Вчиняти так може бути зручно через те, що піделементи здатні спадкувати свої значення в дереві розмітки.

Так само можна легко встановлювати одне і те ж джерело даних сімейства елементів управління, не повторюючи надмірні XAML-фрагменти **"{Binding ElementName=X, Path=Y}"** у багатьох елементах управління. Наприклад, нехай в панель **StackPanel** вкладки доданий новий елемент **Button** (незабаром ви побачите, чому він має настільки великий розмір):

```
<Button Content="Click" Height="200"/>
```

Щоб згенерувати прив'язку даних для множини елементів управління, можна було б використати Visual Studio, але натомість ми введемо модифіковану розмітку у редакторі XAML:

```
<!-- Зверніть увагу, що StackPanel встановлює властивість DataContext -->
<StackPanel Background="#FFE5E5E5" DataContext="{Binding ElementName=mySB}">
  <Label Content="Перемістіть полосу прокручування, щоб побачити поточне значення"/>
  <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
    Maximum = "100" LargeChange="1" SmallChange="1"/>
  <!-- Тепер обидва елементи інтерфейсу користувача працюють зі
    значенням лінійки прокручування унікальними шляхами -->
  <Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
      BorderThickness="2" Content = "{Binding Path=Value}"/>
  <Button Content="Click" Height="200" FontSize = "{Binding Path=Value}"/>
</StackPanel>
```

Тут властивість **DataContext** панелі **StackPanel** встановлюється безпосередньо. Отже, при переміщенні повзунка не лише відображається поточне значення в елементі **Label**, але також збільшується розмір шрифта елемента **Button** у відповідність з тим же значенням (на рис. 6.1 показаний можливий вивід).

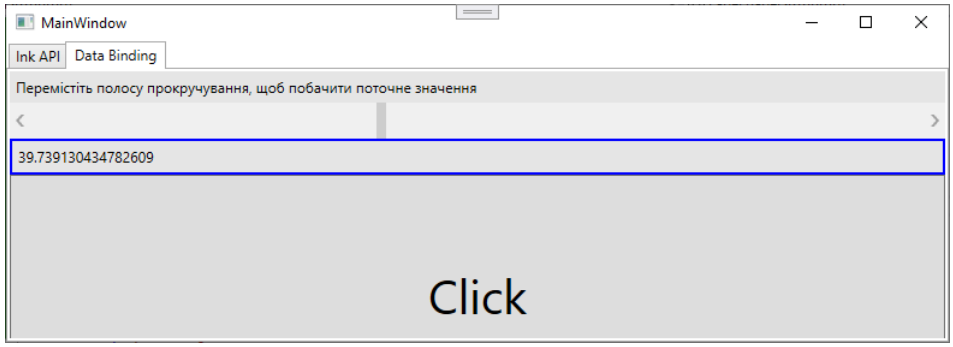


Рис. 6.1. Прив'язування значення **ScrollBar** до елементів **Label** і **Button**

Форматування прив'язаних даних

Замість очікуваного цілого числа для представлення положення повзунка тип **ScrollBar** використовує значення **double**. Отже, у міру переміщення повзунка всередині елемента **Label** відображатимуться різноманітні значення з плаваючою крапкою (на зразок **61.057 692 307 6923**), які виглядають не надто інтуїтивно зрозумілими для кінцевого користувача, майже напевно очікуючого побачити цілі числа (такі як **61**, **62**, **63** і т. д.).

За бажання форматувати дані можна додати властивість **ContentStringFormat**, передавши їй спеціальний рядок і специфікатор формату.

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" Content = "{Binding Path=Value}"
       ContentStringFormat="Тепер значення таке: {0:F0}"/>
```

Якщо в специфікації форматування відсутній який-небудь текст, тоді специфікатору повинен передувати порожній набір фігурних дужок, який є керуючою послідовністю для XAML. Такий прийом повідомляє процесор про те, що символи, які йдуть за **{}** є літералами, а не, скажімо, конструкцією прив'язки. Ось оновлена розмітка XAML (див. рис. 6.2):

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue"
       BorderThickness="2" Content = "{Binding Path=Value}" ContentStringFormat="{} {0:F0}"/>
```

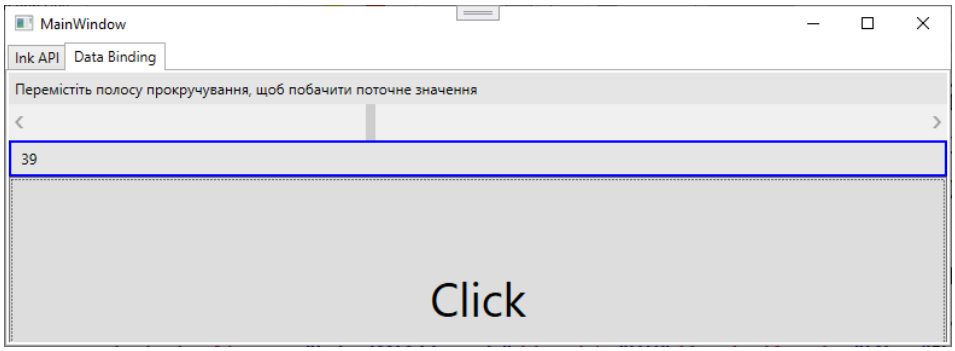


Рис. 6.2. Приведення **double** до **int**, використовуючи форматну властивість **ContentStringFormat**

<u>На замітку!</u> При прив'язці властивості **Text** елемента управління, пару "*ім'я-значення*" об'єкта **StringFormat** можна додавати прямо в конструкцію прив'язки. Вона має бути окремою тільки для властивостей **Content**.

Перетворення даних з використанням інтерфейсу IValueConverter

Якщо потрібно щось більше, ніж просто форматування даних, тоді можна створити спеціальний клас, який реалізовує інтерфейс **IValueCVonverter** з простору імен **System.Windows.Data**. В інтерфейсі **IValueCVonverter** визначені два члени, які дозволяють виконувати перетворення між *джерелом* і *ціллю* (у

разі двонапрямленої[15] прив'язки). Після визначення такий клас можна застосовувати для подальшого уточнення процесу прив'язки даних.

Замість використання властивості форматування можна застосовувати перетворювач значень для відображення цілих чисел всередині елемента управління **Label**. Додамо в проект новий клас (на ім'я **MyDoubleConverter**) з наступним кодом (див. рис. 6.3):

```
class MyDoubleConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
      System.Globalization.CultureInfo culture)
  {
    // Перетворити значення double в int.
    double v = (double)value;
    return (int)v;
  }
  public object ConvertBack(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
  {
    // Через те, що піклуватися про "двонапрямлену" прив'язку
    // не потрібно, просто повернемо значення value.
    return value;
  }
}
```

Метод **Convert()** викликається при передачі значення від *джерела* (**ScrollBar**) до *цілі* (властивість **Content** елемента **Label**). Хоча у нього багато вхідних аргументів, для такого перетворення треба маніпулювати тільки вхідним аргументом типу **object**, який представляє поточне значення **double**. Цей тип можна використати для приведення до цілого і повернення нового числа.

Метод **ConvertBack()** викликатиметься, коли значення передається від *цілі* до *джерела* (якщо включений двонапрямлений режим прив'язки). Тут ми просто повертаємо значення **value**. Це дозволяє вводити в **TextBox** значення з плаваючою крапкою (наприклад, **99.9**) і автоматично перетворювати його в цілочисельне значення (**99**), коли користувач переміщає фокус з елемента управління. Таке перетворення відбувається через те, що метод **Convert()** викликатиметься ще раз після виклику **ConvertBack()**. Якщо просто повернути **null** з **ConvertBack()**, то синхронізація прив'язки виглядатиме порушеною, бо елемент **TextBox** як і раніше відображатиме число з плаваючою крапкою. Щоб використати побудований перетворювач у розмітці, спочатку треба створити локальний ресурс, який представляє створений клас. Тема додавання ресурсів буде детально розкрита в ***Лекція 9. Система ресурсів WPF***. Помістимо показану нижче розмітку відразу після відкриваючого дескриптора **Window**:

```
<Window.Resources>
  <local:MyDoubleConverter x:Key="DoubleConverter"/>
```

[15] Про моделі прив’язки даних детальніше можна прочитати за адресою – https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/data/?view=netdesktop-6.0

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

Далі оновимо конструкцію прив'язки для елемента управління **Label**:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2"
    Content="{Binding Path=Value, Converter{StaticResource DoubleConverter}}" />
```

Після запуску застосунку тепер можна бачити тільки цілі числа.

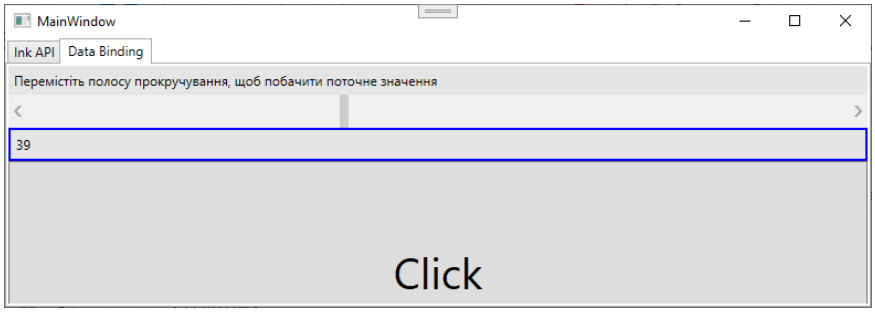


Рис. 6.3. Приведення **double** до **int**, використовуючи клас **MyDoubleConverter**

Встановлення прив'язок даних в коді

Спеціальний перетворювач даних можна також реєструвати в коді. Почнемо з очищення поточного визначення елемента управління **Label** всередині вкладки **DataBinding**, щоб розширення розмітки {**Binding**} більше не використовувалося:

```
<Label x:Name="labelSBThumb" Height="30" BorderBrush="Blue" BorderThickness="2" />
```

Якщо потрібно, то додайте простір імен **System.Windows.Data** і в конструкторі вікна викличемо новий закритий допоміжний метод на ім'я **SetBindings()**, код якого показаний нижче:

```
private void SetBindings()
{
  // Створити об'єкт Binding.
  Binding b = new Binding();
  // Зареєструвати перетворювач, джерело і шлях.
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");
  // Викликати метод SetBinding() об'єкта Label.
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Єдина частина методу **SetBindings()**, яка може виглядати дещо незвичайною – виклик **SetBinding()**. Зверніть увагу, що перший параметр звертається до статичного, доступного тільки для читання поля **ContentProperty** класу **Label**. Як ви довідаєтеся далі, така конструкція називається *властивістю залежності*. Поки просто майте на увазі, що при встановленні прив'язки в коді

перший аргумент майже завжди вимагає зазначення імені класу, який потребує прив'язки (**Label** у цьому випадку), за яким йде звернення до внутрішньої властивості з додаванням до його імені суфікса **Property**. Запустивши застосунок, можна переконатися в тому, що елемент **Label** відображає тільки цілі числа (див. рис. 6.4).

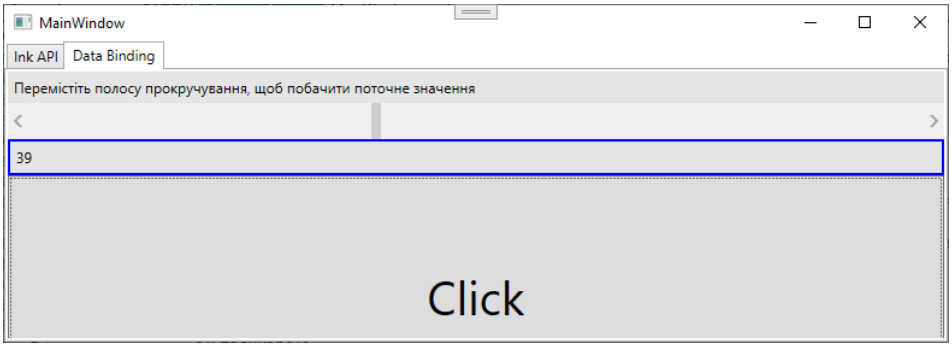


Рис. 6.4. Приведення **double** до **int**, без використання розширення розмітки {**Binding**}

Роль властивостей залежності

Подібно до будь-якого API-інтерфейсу .NET, всередині реалізації WPF використовуються всі *члени системи типів* .NET (класи, структури, інтерфейси, делегати, перерахування) і кожен *член типу* (властивості, методи, події, константні дані, поля тільки для читання і т. п.). Проте WPF також підтримує *унікальну* програмну концепцію під назвою ***властивості залежності*** (**dependency property**).

Властивості залежності є абсолютно новою, значно кориснішою, реалізацією властивостей. Без них ви не зможете працювати з основними засобами WPF, такими як *анімація*, *прив'язка* даних і *стилі*.

Більшість властивостей у елементів WPF є властивостями залежності. У всіх прикладах, які були приведені до цього моменту, ви використовували властивості залежності, навіть не підозрюючи про це. Це пояснюється тим, що властивості залежності розроблені так, щоб з ними можна було працювати як із звичайними властивостями.

Та все ж властивості залежності *не є* звичайними властивостями. Краще всього уявляти собі ці властивості як звичайні (які визначені в .NET звичайним способом), але з додатковим набором можливостей WPF. У концептуальному відношенні поведінка властивостей залежності не відрізняється від поведінки звичайних властивостей, проте *реалізовані вони по-іншому*. Причина проста: *продуктивність*. Якби розробники WPF просто внесли додаткові можливості в систему властивостей .NET, то їм довелося б створити складний і громіздкий шар для вашого коду. Звичайні властивості не можуть підтримувати всі характеристики властивостей залежності, не перевантажуючи при цьому систему.

Властивості залежності є специфічним дітищем WPF. Проте у бібліотеках WPF вони завжди поміщені в оболонки звичайних процедур властивостей .NET. Це дозволяє використати їх звичайним способом навіть в тому коді, який не має поняття про систему властивостей залежності WPF. На перший погляд дивно, що нова технологія запакована в стару, проте тільки так WPF може змінити такий фундаментальний інгредієнт, як властивості, не порушуючи структуру решти світу .NET.

Подібно до «*звичайної*» властивості .NET (яку в літературі, присвяченій WPF, часто називають властивістю CLR (**CLR property**)), властивості залежності можуть *встановлюватися декларативно*, з використанням XAML або програмно у файлі коду. Більше того, властивості залежності (подібно до властивостей CLR) зрештою призначені для інкапсуляції полів даних класу і можуть бути конфігуровані як доступні тільки для читання, тільки для запису або для читання і запису.

Що цікавіше: майже у будь-якому випадку ви перебуватимете в щасливому невіданні відносно того, що насправді встановлюєте (або читаєте) властивість залежності, а не властивість CLR! Наприклад, властивості **Height** і **Width**, які елементи управління WPF спадкують від **FrameworkElement**, а також член **Content**, успадкований від **ControlContent** – все це фактично властивості залежності:

```
<!-- Встановити три властивості залежності -->
<Button x:Name = "btnMyButton" Height = "50" Width = "100" Content = "OK"/>
```

З врахуванням всієї цієї схожості виникає питання: навіщо треба було визначати у WPF новий термін для такої знайомої концепції? Причина криється в способі реалізації властивості залежності всередині класу. На вищому рівні всі властивості залежності створюються так:

- Передусім, клас, який визначає властивість залежності, повинен мати у своєму ланцюжку спадкування **DependencyObject**.
- Єдина властивість залежності представляється як загальнодоступне, статичне, призначене тільки для читання поле в класі типу **DependencyProperty**. За існуючою угодою ім'я цього поля формується додаванням слова **Property** до імені оболонки CLR.
- Змінна **DependencyProperty** зареєстрована за допомогою статичного виклику **DependencyProperty.Register()**, що зазвичай відбувається в статичному конструкторі або вбудованим чином, при оголошенні змінної.
- Нарешті, клас визначить дружню до XAML властивість CLR, яка здійснює виклики методів, наданих **DependencyObject**, для отримання і встановлення значення.

Будучи реалізованими, властивості залежності надають багато потужних засобів, використовуваних різними технологіями WPF, включаючи *прив'язку* даних, служби *анімації*, *стилі*, *шаблонами* і т. д. В основі своїй мотивація властивостей залежності полягає в наданні способу обчислення значення

властивості на основі значень інших джерел. Нижче приведений список деяких основних переваг, які виходять за рамки простої інкапсуляції даних, наявної у властивості CLR:

1. Властивості залежності можуть спадкувати свої значення від XAML-визначення батьківського елемента. Наприклад, якщо ви визначите значення атрибуту **FontSize** у відкриваючому дескрипторі <**Window**>, то всі елементи управління в цьому **Window** за замовчуванням матимуть цей розмір шрифта.
2. Властивості залежності підтримують можливість отримувати значення, які встановлені елементами всередині їх області визначення XAML, наприклад, у випадку встановлення елементом **Button** властивості **Dock** батьківського контейнера **DockPanel**. (Згадайте, що *приєднувані властивості* роблять саме це, бо є різновидом *властивостей залежності*.)
3. Властивості залежності дозволяють WPF обчислювати значення на основі декількох зовнішніх значень, що може бути дуже важливо для *анімації* і служб *прив'язки* даних.
4. Властивості залежності надають інфраструктуру підтримки для *тригерів* WPF (які, так само досить часто використовуються при роботі з *анімацією* і *прив'язкою* даних).

Запам'ятайте, що у багатьох випадках ви взаємодіятимете з існуючою властивістю залежності в манері, ідентичній звичайній властивості CLR (завдяки оболонці XAML).

Врахуйте, що у багатьох випадках ви взаємодіятимете з існуючою властивістю залежності в манері, ідентичній роботі зі звичайною властивістю CLR (завдяки оболонці XAML). У попередньому розділі, присвяченому прив'язці даних, ви довідалися, що якщо потрібно встановити прив'язку даних в коді, то має бути викликаний метод **SetBinding()** на цільовому об'єкті операції і вказана властивість залежності, з якою працюватиме прив'язка:

```
private void SetBindings()
{
  // Створити об'єкт Binding.
  Binding b = new Binding();
  // Зареєструвати перетворювач, джерело і шлях.
  b.Converter = new MyDoubleConverter();
  b.Source = this.mySB;
  b.Path = new PropertyPath("Value");
  // Викликати метод SetBinding() об'єкта Label,
  // т. т. задати властивість залежності.
  this.labelSBThumb.SetBinding(Label.ContentProperty, b);
}
```

Ви побачите схожий код в ***Лекція 10. Реалізація анімації WPF*** під час дослідження запуску анімації в коді:

```
// Задати властивість залежності.
rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
```

Потреба в побудові спеціальної властивості залежності виникає тільки тоді, коли ви розробляєте власний елемент управління WPF. Наприклад, коли створюється клас **UserControl** з чотирма спеціальними властивостями, які повинні тісно інтегруватися з API-інтерфейсом WPF, вони мають бути реалізовані з використанням логіки властивостей залежності.

Зокрема, якщо потрібно, щоб властивість була метою операції прив'язки даних або анімації, якщо вона зобов'язана повідомляти про свою зміну, якщо властивість повинна бути спроможною працювати установником в стилі WPF або отримувати свої значення від батьківського елемента, то можливостей звичайної властивості CLR буде не достатньо. У разі використання звичайної властивості іншими програмістами, вони, дійсно можуть отримувати і встановлювати її значення, але якщо вони пробуватимуть застосувати таку властивість всередині контексту служби WPF, то вона не працюватиме так як очікується. Через те, що заздалегідь не можна знати, як інші захочуть взаємодіяти з властивостями спеціальних класів **UserControl**, треба виробити в собі звичку при побудові спеціальних елементів управління *завжди* визначати властивості залежності.

Дослідження існуючої властивості залежності

Перш ніж ви навчитеся створювати спеціальні властивості залежності, давайте розглянемо внутрішню реалізацію властивості **Height** класу **FrameworkElement**. Нижче приведений відповідний код (з коментарями):

```
// FrameworkElement "є" DependencyObject.
public class FrameworkElement : UIElement, IFrameworkInputElement,
      IInputElement, ISupportInitialize, IHaveResources, IQueryAmbient
{
  …
  // Статичне поле тільки для читання типу DependencyProperty.
  public static readonly DependencyProperty HeightProperty;

  // Поле DependencyProperty часто реєструється
  // у статичному конструкторі класу.
  static FrameworkElement()
  {
    …
    HeightProperty = DependencyProperty.Register(
                     "Height",
                     typeof(double),
                     typeof(FrameworkElement),
                     new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
                     FrameworkPropertyMetadataOptions.AffectsMeasure,
                     new PropertyChangedCallback),
                     new ValidateValueCallback);
  // Оболонка CLR, реалізована з використанням
  // успадкованих методів GetValue()/SetValue().
  public double Height
  {
```

```
    get { return (double)base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
  }
}
```

Як бачите, порівняно із звичайними властивостями CLR властивості залежності вимагають чималого обсягу додаткового коду. У реальності залежність може виявитися навіть ще складнішою, ніж показано тут (на щастя, багато реалізацій простіші властивості **Height**).

У першу чергу згадайте, що якщо в класі потрібно визначити властивість залежності, то вона повинна мати у своєму ланцюжку спадкування **DependencyObject**, бо саме цей клас визначає методи **GetValue()** і **SetValue()**, використовувані в оболонці CLR. Через те, що клас **FrameworkElement** "є" **DependencyObject**, зазначена вимога задоволена.

Далі згадайте, що сутність, де дійсно зберігається значення властивості (значення **double** у разі **Height**), представляється як відкрите, статичне, представляє поле типу **DependencyProperty**, тільки читання. За угодою ім'я цієї властивості повинне завжди формуватися з імені зв'язаної оболонки CLR з додаванням суфікса **Property**:

```
public static readonly DependencyProperty HeightProperty;
```

Враховуючи, що властивості залежності оголошуються як статичні поля, вони зазвичай створюються (і реєструються) всередині статичного конструктора класу. Об'єкт **DependencyProperty** створюється за допомогою виклику статичного методу **DependencyProperty.Register()**. Цей метод має багато перевантажуваних версій, але у разі властивості **Height** він викликається так:

```
HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata((double)0.0,
    FrameworkPropertyMetadataOptions.AffectsMeasure,
    new PropertyChangedCallback(FrameworkElement.OnTransformDirty)),
    new ValidateValueCallback(FrameworkElement.IsWidthHeightValid));
```

Першим аргументом, який передається методу **DependencyProperty.Register()**, є ім'я звичайної властивості CLR класу (**Height**), а другий аргумент містить інформацію про тип даних, який його інкапсулює (**double**). Третій аргумент задає інформацію про тип класу, якому належить властивість (**FrameworkElement**). Хоча такі відомості можуть здатися надмірними (врешті-решт, поле **HeightProperty** вже визначено всередині класу **FrameworkElement**), це дуже продуманий аспект WPF, бо він дозволяє одному класу реєструвати властивості в іншому класі (навіть якщо його визначення було запечатане).

Четвертий аргумент, який передається методу **DependencyProperty.Register()** в розглянутому прикладі, є те, що дійсно робить властивості залежності унікальними. Тут передається об'єкт **FrameworkPropertyMetadata**, який описує різноманітні деталі відносно того,

як інфраструктура WPF повинна обробляти цю властивість в плані повідомлень за допомогою зворотних викликів (якщо властивості потрібно сповіщати інших, коли його значення змінюється). Крім того, об'єкт **FrameworkPropertyMetadata** задає різні параметри (перерахування **FrameworkPropertyMetadataOptions**), які управляють тим, на що властивість впливає (чи працює вона з прив'язкою даних, чи може спадкувати і т. д.). У цьому випадку аргументи конструктора **FrameworkPropertyMetadata** можна описати так:

```
new FrameworkPropertyMetadata(
  // Стандартне значення властивості.
  (double) 0.0,
  // Параметри метаданих.
  FrameworkPropertyMetadataOptions.AffectsMeasure,
  // Делегат, який вказує на метод, що викликається при зміні властивості,
  new PropertyChangedCallback(FrameworkElement.OnTransformDirty)
)
```

Через те, що останній аргумент конструктора **FrameworkPropertyMetadata** є делегатом, зверніть увагу, що він вказує на статичний метод **OnTransformDirty()** класу **FrameworkElement**. Код методу **OnTransformDirty()** тут відсутній, але майте на увазі, що при створенні спеціальної властивості залежності завжди можна вказувати делегат **PropertyChangeCallback**, націлений на метод, який викликатиметься у разі зміни значення властивості.

Звернемося до фінального параметра методу **DependencyProperty.Register()** – другому делегату типу **ValidateValueCallback**, який вказує на метод класу **FrameworkElement**, що викликається для перевірки достовірності значення, присвоюваного властивості:

```
new ValidateValueCallback(FrameworkElement.IsWidthHeightValid)
```

Метод **IsWidthHeightValid()** містить логіку, яку зазвичай очікують знайти у блоці встановлення значення властивості (як детальніше пояснюється в наступному розділі):

```
private static bool IsWidthHeightValid(object value)
{
  double num = (double)value;
  return ((!DoubleUtil.IsNaN(num) && (num >= 0.0)) && !double.IsPositiveInfinity(num));
  public MainWindow()
}
```

Після того, як об'єкт **DependencyProperty** зареєстрований, залишається запакувати поле у звичайну властивість CLR (**Height** у цьому випадку). Проте, зверніть увагу, що блоки **get** і **set** не просто повертають або встановлюють значення **double** змінної-члена рівня класу, а роблять це опосередковано з використанням методів **GetValue()** і **SetValue()** базового класу **System.Windows.DependencyObject**:

```
public double Height
{
```

```
    get { return (double)base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```

Важливі зауваження щодо оболонок властивостей CLR

Отже, підводячи підсумки сказаному досі, властивості залежності виглядають як звичайні властивості, коли ви отримуєте або встановлюєте їх значення в розмітці XAML або в коді, але "за кулісами" вони реалізовані з допомогою набагато більш хитромудрих прийомів кодування. Згадайте, що основним призначенням цього процесу є побудова спеціального елемента управління зі спеціальними властивостями, що мають бути інтегровані зі службами WPF, які вимагають взаємодії через властивості залежності (наприклад, з анімацією, прив'язкою даних і стилями).

Незважаючи на те що частина реалізації властивості залежності передбачає визначення оболонки CLR, *ви ніколи не повинні поміщати логіку перевірки достовірності у блок* **set**. До того ж оболонка CLR властивості залежності не повинна робити нічого окрім викликів **GetValue()** або **SetValue()**.

Виконавче середовище WPF сконструйоване так, що якщо написати розмітку XAML, яка виглядає як встановлення властивості, наприклад:

```
<Button x:Name = "btnMyButton" Height="100" .../>,
```

то виконавче середовище взагалі обійде блок встановлення властивості **Height** і безпосередньо викличе метод **SetValue()**. Причина такої незвичайної поведінки пов'язана з простим прийомом оптимізації. Якби виконавче середовище WPF зверталося до блоку встановлення властивості **Height**, то йому довелося б під час виконання з'ясовувати за допомогою рефлексії, де знаходиться поле **DependencyProperty** (вказане в першому аргументі **SetValue()**), посилатися на нього в пам'яті і т. д. Те ж саме залишається справедливим і при написанні розмітки XAML, яка отримує значення властивості **Height** – метод **GetValue()** викликатиметься безпосередньо.

Але раз так, тоді навіщо взагалі будувати оболонку CLR? Річ у тому, що XAML у WPF не дозволяє викликати функції в розмітці, тому наступний фрагмент приведе до помилки:

На замітку! Перерахуємо терміни, використовувані для властивостей залежностей.

- ***Властивість залежностей*** – властивість, підтримувана **DependencyProperty**;
- ***Ідентифікатор властивості залежностей.*** Екземпляр **DependencyProperty**, який отримується в результаті реєстрації властивості залежностей і потім зберігається як статичний член класу. Цей ідентифікатор використовується як параметр для багатьох API, що взаємодіють з системою властивостей WPF.
- **CLR-оболонка.** Фактичні реалізації отримання і задання властивості. Ці реалізації включають ідентифікатор властивості залежностей, використовуючи його у викликах **GetValue** і **SetValue** і, отже,

забезпечуючи підтримку для властивості за допомогою системи властивостей WPF.

Насправді встановлення або набуття значення в розмітці із використанням оболонки CLR треба вважати способом повідомлення виконавчого середовища WPF про потребу виклику методів **GetValue()/SetValue()**, бо безпосередньо викликати їх в розмітці неможливо. А що, якщо звернутися до оболонки CLR в коді, як показано нижче?

```
Button b = new Button();
b.Height = 10;
```

У такому разі, якщо блок **set** властивості **Height** містить якийсь код окрім виклику **SetValue()**, то він повинен виконатися, тому що оптимізація синтаксичного аналізатора XAML у WPF не задіюється.

Запам'ятайте *основне правило*: при реєстрації властивості залежності використайте делегат **ValidateValueCallback** для вказівки на метод, який виконує перевірку достовірності даних. Такий підхід гарантує коректну поведінку незалежно від того, що використовується для отримання/встановлення властивості залежності – розмітка XAML або код.

Побудова спеціальної властивості залежності

Якщо до цього моменту ви злегка заплуталися, то така реакція абсолютно нормальна. Створення властивостей залежності може вимагати деякого часу звикання. Як би то не було, але це частина процесу побудови багатьох спеціальних елементів управління WPF, так що давайте розглянемо, як створюється властивість залежності.

Розпочнемо зі створення нового проекту застосунку WPF на ім'я **CustomDepProp**. Виберемо в меню **Project** (Проект) пункт **Add New Item** (Додати новий елемент) і, вказавши **User Control** (Елемент управління (WPF) користувача) для типу елемента (див. рис. 6.5), створимо елемент з ім'ям **ShowNumberControl.xaml**.

<u>На замітку!</u> Детальніші відомості про клас **UserControl** у WPF дивіться в ***Лекція 9. Ресурси WPF***, а поки просто дотримуйтесь вказівок у міру опрацювання прикладу.

Подібно до вікна типи **UserControl** у WPF мають файл XAML і пов'язаний файл коду. Модифікуємо розмітку XAML елемента управління користувача, щоб визначити простий елемент **Label** усередині **Grid**:

```
<UserControl x:Class="CustomDepProp.ShowNumberControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:locals="clr-namespace:CustomDepProp"
             mc:Ignorable="d"
```

```
            d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Label x:Name="numberDisplay" Height="50" Width="200" Background="LightBlue"/>
    </Grid>
</UserControl>
```

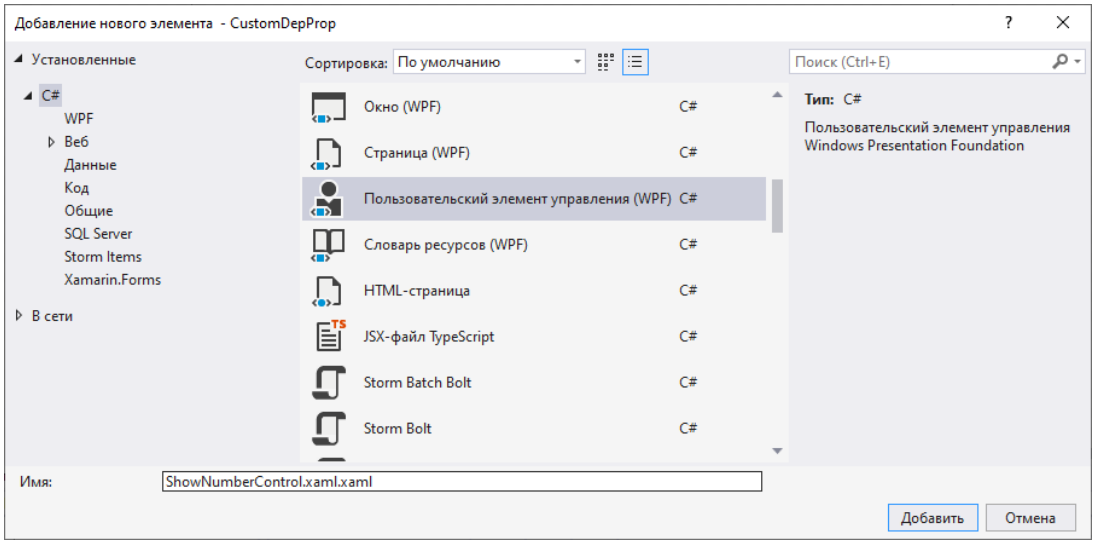


Рис. 6.5. Елемент управління (WPF) користувача

У файлі коду для цього елемента створимо звичайну властивість .NET, яка запаковує поле типу **int** і встановлює нове значення для властивості **Content** елемента **Label**:

```
public partial class ShowNumberControl : UserControl
{
    public ShowNumberControl()
    {
        InitializeComponent();
    }

    // Звичайна властивість .NET.
    private int _currNumber = 0;
    public int CurrentNumber
    {
        get => _currNumber;
        set
        {
            _currNumber = value;
            numberDisplay.Content = CurrentNumber.ToString();
        }
    }
}
```

Тепер відновимо визначення XAML, оголосивши екземпляр спеціального елемента управління всередині диспетчера компонування **StackPanel**. Через те, що спеціальний елемент управління не входить до складу основних збірок WPF, знадобиться визначити спеціальний простір імен XML, який відображається на нього. Ось потрібна розмітка:

```
<Window x:Class="CustomDepProp.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:myCtrls="clr-namespace:CustomDepProp"
        xmlns:local="clr-namespace:CustomDepProp"
        mc:Ignorable="d"
        Title="Простий застосунок властивостей залежностей" Height="150" Width="250"
            WindowStartupLocation="CenterScreen">
  <StackPanel>
    <myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100"/>
  </StackPanel>
</Window>
```

Схоже, що візуальний конструктор Visual Studio коректно відображає значення, встановлене у властивості **CurrentNumber** (рис. 6.6).

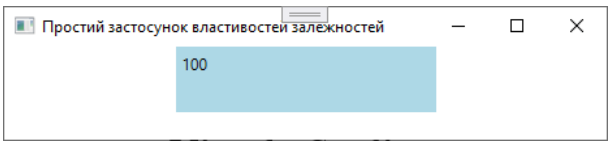


Рис. 6.6. Відображення правильного результату в **CurrentNumber**

А що, якщо до властивості **CurrentNumber** потрібно застосувати об'єкт анімації, який забезпечить зміну значення властивості від **100** до **500** впродовж **10** секунд? Якщо це бажано зробити в розмітці, тоді область **myCtrls:ShowNumberControl** можна змінити так:

```
<myCtrls:ShowNumberControl x:Name="myShowNumberCtrl" CurrentNumber="100">
  <myCtrls:ShowNumberControl.Triggers>
    <EventTrigger RoutedEvent = "myCtrls:ShowNumberControl.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "CurrentNumber">
            <Int32Animation From = "100" To = "500" Duration = "0:0:10"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions >
    </EventTrigger>
  </myCtrls:ShowNumberControl.Triggers>
</myCtrls:ShowNumberControl>
```

Після запуску застосунку об'єкт анімації не зможе знайти відповідну ціль і тому буде проігнорований, т. т. буде згенеровано виключення (див. рис. 6.7).

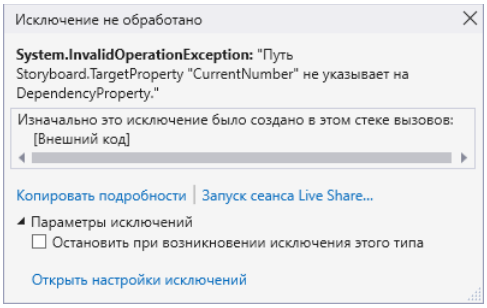


Рис. 6.7. Не зареєстрована властивість незалежності викликала виключення

Причина в тому, що властивість **CurrentNumber** *не* була *зареєстрована* як властивість залежності. Щоб усунути проблему, звернемося до файла коду для *спеціального елемента управління* і повністю закоментуємо поточну логіку властивості (включаючи закрите підтримуюче поле). Тепер помістимо курсор всередину області визначення класу і введемо фрагмент коду **propdp**. Після вводу **propdp** двічі натиснемо клавішу <**Tab**>. Фрагмент коду розгорнеться у базовий *шаблон* властивості залежності:

```
// Закоментований шаблон після вводу фрагмента коду propdp
  /*public int MyProperty
  {
    get { return (int)GetValue(MyPropertyProperty); }
    set { SetValue(MyPropertyProperty, value); }
  }

  // Використати DependencyProperty як підтримуюче сховище для MyProperty.
  // Це включає анімацію, стилі, прив'язку і т. д.
  public static readonly DependencyProperty MyPropertyProperty =
    DependencyProperty.Register("MyProperty", typeof(int),
    typeof(ownerclass), new PropertyMetadata(0));*/
```

Модифікуємо шаблон, як показано нижче:

```
public int CurrentNumber
{
  get => (int)GetValue(CurrentNumberProperty);
  set => SetValue(CurrentNumberProperty, value);
}

// Використати DependencyProperty як підтримуюче сховище для CurrentNumber.
// Це включає анімацію, стилі, прив'язку і т. д.
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber",
    typeof(int),
    typeof(ShowNumberControl),
    new UIPropertyMetadata(0));
```

Робота схожа на прооблену в реалізації властивості **Height**; проте, фрагмент коду **propdp** реєструє властивість безпосередньо в тілі, а не в статичному конструкторі (що добре). Також зверніть увагу, що об'єкт **UIPropertyMetadata** використовується для визначення стандартного цілочисельного значення (**0**) замість складнішого об'єкта **FrameworkPropertyMetadata**. У результаті отримується проста версія **CurrentNumber** як властивості залежності.

Додавання процедури перевірки достовірності даних

Незважаючи на наявність властивості залежності на ім'я **CurrentNumber**, анімація поки ще не спостерігається. Наступним *коригуванням* буде задання функції, яка викликається для виконання перевірки достовірності даних. У цьому прикладі ми припустимо, що треба забезпечити знаходження значення властивості **CurrentNumber** в діапазоні між **0** і **500**.

Додамо в метод **DependencyProperty.Register()** останній аргумент типу **ValidateValueCallback**, який вказує на метод на ім'я **ValidateCurrentNumber**.

Тут **ValidateValueCallback** є делегатом, який може вказувати тільки на методи, що повертають тип **bool** і приймають єдиний аргумент типу **object**. Екземпляр **object** представляє присвоюване нове значення. Реалізація **ValidateCurrentNumber** повинна повертати **true**, якщо вхідне значення знаходиться в очікуваному діапазоні, і **false** інакше:

```
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber",
    typeof(int),
    typeof(ShowNumberControl),
    new UIPropertyMetadata(100),
    new ValidateValueCallback(ValidateCurrentNumber));

public static bool ValidateCurrentNumber(object value)
{
  // Просте бізнес-правило: значення повинне знаходитися в діапазоні між 0 і 500.
  return Convert.ToInt32(value) >= 0 && Convert.ToInt32(value) <= 500;
}
```

Реагування на зміну властивості

Отже, допустиме число вже є, але анімація як і раніше відсутня. Остання зміна, яку потрібно буде внести – передати в другому аргументі конструктора **UIPropertyMrtadata** об'єкт **PropertyChangedCallback**. Цей делегат може вказувати на будь-який метод, приймаючий **DependencyObject** у першому параметрі і **DependencyPropertyChangeEventArgs** в другому. Модифікуємо код так:

```
public static readonly DependencyProperty CurrentNumberProperty =
  DependencyProperty.Register("CurrentNumber",
    typeof(int),
    typeof(ShowNumberControl),
    // Зверніть увагу на другий параметр конструктора UIPropertyMetadata.
```

```
    new UIPropertyMetadata(100, new PropertyChangedCallback(CurrentNumberChanged)),
    new ValidateValueCallback(ValidateCurrentNumber));
```

Кінцевою метою всередині методу **CurrentNumberChanged()** буде зміна властивості **Content** об'єкта **Label** на нове значення, присвоєне властивості **CurrentNumber**. Проте ми стикаємося з серйозною проблемою: метод **CurrentNumberChanged()** статичний, бо він повинен працювати зі статичним об'єктом **DependencyProperty**. Як тоді отримати доступ до об'єкта **Label** для поточного екземпляра **ShowNumberControl**? Потрібне посилання міститься в першому параметрі **DependencyObject**. Нове значення можна знайти із застосуванням вхідних аргументів події. Нижче показаний потрібний код, який змінюватиме властивість **Content** об'єкта **Label**:

```
private static void CurrentNumberChanged(DependencyObject depObj,
    DependencyPropertyChangedEventArgs args)
{
  // Привести DependencyObject до ShowNumberControl.
  ShowNumberControl c = (ShowNumberControl) depObj;
  // Отримати елемент управління Label в ShowNumberControl.
  Label theLabel = c.numberDisplay;
  // Встановити для Label нове значення.
  theLabel.Content = args.NewValue.ToString();
}
```

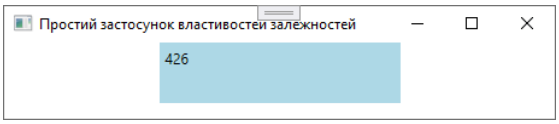


Рис. 6.8. Остаточний результат в **CurrentNumber**

Бачите, наскільки довгий шлях довелося пройти, щоб усього лише змінити вміст мітки. Перевага полягає в тому, що тепер властивість залежності **CurrentNumber** може бути ціллю для стилю WPF, об'єкта анімації, операції прив'язки даних і т. д. Знову запустивши застосунок, легко помітити, що значення змінюється під час виконання.

На цьому огляд властивостей залежності WPF завершений. Хоча тепер ви набагато краще розумієте, що вони дозволяють робити, і як створювати власні властивості такого роду, майте на увазі, що багато деталей тут не було розкрито.

Якщо вам одного разу знадобиться створювати багато власних елементів управління, які підтримують спеціальні властивості, то загляньте в підрозділ «**Properties**» (Властивості) вузла в документації по .NET Framework за адресою https://docs.microsoft.com/en-us/dotnet/framework/.

Там ви знайдете набагато більше прикладів побудови властивостей залежності, приєднуваних властивостей, різноманітних способів конфігурації метаданих і масу інших детальних відомостей.

Резюме

У цій лекції ви з'ясували, що інфраструктура WPF додає унікальний аспект до традиційних програмних примітивів .NET, зокрема до властивостей і подій. Як було показано, механізм властивостей залежності дозволяє будувати властивість, яка може інтегруватися з набором служб WPF (анімації, прив'язки даних, стилі і т. д.). Як попутне зауваження: механізм маршрутизованих подій надає події спосіб поширюватися вгору або вниз деревом розмітки.

Контрольні питання

1. Що означає дія з прив’язки даних?
2. Що є джерелом прив’язки даних?
3. Що означає поняття місце прив’язки даних?
4. Приведіть приклад простого сценарію прив’язки?
5. Яку роль грає властивість залежності у сценарії прив’язки даних?
6. Чи можлива прив’язка елементів?
7. Який механізм забезпечує визначення прив’язки у розмітці XAML?
8. Що означає конструкція {Binding}? Приведіть приклад. У цьому прикладі вкажіть де джерело операції прив’язки, а де властивість до якої здійснюється прив’язка.
9. Яку роль грає властивість DataContext при організації прив’язки?
10. Що означає поняття двонапрямленої прив’язки?
11. Приведіть простий приклад встановлення прив’язки у зв’язаному C#-коді.
12. Яку роль грає властивість залежності в організації прив’язки?
13. Що таке звичайна або CLR властивість залежності?
14. Що таке елемент управління користувача?
15. Для чого використовується процедура перевірки достовірності даних?

Лекція 7. Служби візуалізації графіки WPF

Розглянемо можливості графічної візуалізації WPF. Ви побачите, що інфраструктура WPF надає три окремих способи візуалізації графічних даних: фігури, малюнки і візуальні об'єкти. Розібравшись у перевагах і недоліках кожного підходу, ми приступимо до дослідження світу інтерактивної двовимірної графіки з використанням класів з простору імен **System.Windows.Shapes**. Потім буде показано, як за допомогою малюнків і геометричних об'єктів візуалізувати двовимірні дані в легковагій манері. І, нарешті, з'ясуєте, як домогтися від візуального рівня максимальної функціональності і продуктивності.

Разом з тим ми торкнемося багатьох пов'язаних тем, такі як створення спеціальних кистей і пер, застосування графічних трансформацій до візуалізації і виконання операцій перевірки попадання. Зокрема ви побачите, як можна спростити рішення завдань кодування графіки за допомогою інтегрованих інструментів Visual Studio і додаткового засобу під назвою **Inkscape**.

> **На замітку!** Графіка є ключовим аспектом розробки WPF. Навіть якщо ви не будуєте застосунок з інтенсивною графікою (на зразок відеогри або мультимедійного застосунку), то теми, які ми розглянемо, критично важливі при роботі з такими службами, як шаблони елементів управління, анімація і налаштування прив'язки даних.

Служби графічної візуалізації WPF

Інфраструктура WPF використовує особливий різновид графічної візуалізації, який відомий під назвою *графіка режиму збереження* (**retained mode**). Кажучи простіше, це означає, що після застосування розмітки XAML або процедурного коду для генерування графічної візуалізації інфраструктура WPF несе відповідальність за збереження візуальних елементів і забезпечення їх коректного перемальовування та оновлення в оптимальний спосіб. Отже, графічні дані, які візуалізуються, присутні постійно, навіть коли кінцевий користувач приховує зображення, змінюючи розмір вікна або згортаючи його, перекриваючи одне вікно іншим і т. д.

Разючий контраст попередніх версій API-інтерфейсів графічної візуалізації від Microsoft (включаючи GDI+ у Windows Forms) в тому, що вони були графічними системами *прямого режиму* (**immediate mode**). У такій моделі відповідальність за коректне «*запам'ятовування*» та оновлення візуалізованих елементів, упродовж часу життя застосунку покладалася на програміста. Наприклад, в застосунку Windows Forms візуалізація фігури на зразок прямокутника передбачала обробку події **Paint** (або перевизначення віртуального методу **OnPaint()**), отримання об'єкта **Graphics** для малювання прямокутника і, що найважливіше, додавання інфраструктури, яка забезпечує збереження зображення за ситуації, коли користувач змінив розміри вікна (наприклад, за рахунок створення змінних-членів для представлення позиції прямокутника і виклику методу **Invalidate()** у багатьох місцях коду).

Перехід від графіки *прямого режиму* до графіки *режиму збереження* – дійсно хороше рішення, бо програмістам доводиться писати і супроводжувати набагато менший обсяг рутинного коду для підтримки графіки. Проте це не означає, що API-інтерфейс графіки WPF повністю відрізняється від попередніх інструментальних наборів візуалізації. Наприклад, як і GDI+, інфраструктура WPF підтримує різноманітні типи об'єктів кистей і пер, прийоми перевірки попадання, області відсікання, графічні трансформації і т. д. Тому, якщо у вас є досвід роботи з GDI (чи GDI на мові C/C++), то ви вже маєте непогане уявлення про спосіб виконання базової візуалізації у WPF.

Варіанти графічної візуалізації WPF

Як і з іншими аспектами розробки застосунків WPF, існує вибір з декількох способів виконання графічної візуалізації після ухвалення рішення робити це за допомогою розмітки XAML або процедурного коду C# (або їх комбінації). Зокрема, інфраструктура WPF пропонує наступні три індивідуальні підходи до візуалізації графічних даних.

- ***Фігури.*** Інфраструктура WPF надає простір імен **System.Windows.Shapes**, в якому визначене невелике число класів для візуалізації двовимірних геометричних об'єктів (прямокутників, еліпсів, багатокутників і т. п.). Хоча такі типи прості у використанні і дуже потужні, у випадку непродуманого їх застосування, вони можуть спричиняти значні накладні витрати пам'яті.
- ***Малюнки і геометричні об'єкти.*** Другий спосіб візуалізації графічних даних у WPF припускає роботу з класами, похідними від абстрактного класу **System.Windows.Media.Drawing**. Використовуючи класи, подібні **GeometryDrawing** або **ImageDrawing** (на додаток до різних геометричних об'єктів), можна візуалізувати графічні дані у менш витратний (але менш функціональний) спосіб.
- ***Візуальні об'єкти.*** Найшвидший і менш витратний спосіб візуалізації графічних даних у WPF передбачає роботу з візуальним рівнем, який доступний тільки через код C#. Із застосуванням класів, похідних від **System.Windows.Media.Visual**, можна взаємодіяти *безпосередньо* з графічною підсистемою WPF.

Причина надання різних способів рішення однієї тієї ж самої задачі (тобто візуалізація графічних даних) пов'язана з витратою пам'яті і зрештою з продуктивністю застосунку. Через те, що WPF є системою, яка інтенсивно використовує графіку, немає нічого незвичайного в тому, що застосунку потрібно візуалізовувати сотні або навіть тисячі різних зображень на поверхні вікна, і вибір реалізації (фігури, малюнки або візуальні об'єкти) може на цей вибір радикально впливати. Важливо розуміти, що при побудові застосунку WPF висока вірогідність використання всіх трьох підходів. Як *емпіричне правило* запам'ятайте: якщо потрібний помірний обсяг *інтерактивних* графічних даних, якими може маніпулювати користувач (приймаючих ввід від миші, відображаючих спливаючі підказки і т. д.), то треба застосовувати члени з простору імен **System.Windows.Shapes**.

Навпаки, малюнки і геометричні об'єкти краще підходять, коли потрібно моделювати *складні* і здебільшого *не інтерактивні* векторні графічні дані з використанням розмітки XAML або коду C#. Хоча малюнки і геометричні об'єкти здатні реагувати на події миші, а також підтримують перевірку попадання та операції перетягування, для виконання таких дій зазвичай доводиться писати більше коду.

Нарешті, якщо потрібен *найшвидший* спосіб візуалізації значних обсягів графічних даних, то треба вибирати візуальний рівень. Припустимо, що інфраструктура WPF використовується для побудови наукового застосунку, який повинен відображати тисячі точок на графіку даних. Використовуючи візуальний рівень, точки на графіку візуалізуватимуться оптимальним способом. Далі покажемо, візуальний рівень доступний лише з коду C#, але не з розмітки XAML.

Незалежно від вибраного підходу (фігури, малюнки, геометричні об'єкти чи візуальні об'єкти) завжди застосовуватимуться найчастіше використовувані графічні примітиви, такі як *кисті* (заповнення обмежених областей), *пера* (малювання контурів) та *об'єкти трансформацій* (які видозмінюють дані). Спочатку дослідимо класи з простору **System.Windows.Shapes**.

> **На замітку!** Інфраструктура WPF постачається також з повнофункціональним API-інтерфейсом, який можна використати для візуалізації і маніпулювання тривимірною графікою, але ми його не розглядатимемо. Детальніше про тривимірну графіку за адресою https://docs.microsoft.com/en-us/dotnet/framework/ .

Візуалізація графічних даних з використанням фігур

Члени простору імен **System.Windows.Shapes** пропонують найбільш прямолінійний, інтерактивний і найвитратніший (витрати пам'яті) спосіб візуалізації двовимірного зображення. Цей простір імен (у збірці **PresentationFramework.dll**) складається всього з шести запечатаних класів, які розширюють абстрактний базовий клас **Shape**: **Ellipse**, **Rectangle**, **Line**, **Polygon**, **Polyline** і **Path**.

Абстрактний клас **Shape** спадкоємець класу **FrameworkElement**, який сам нащадок **UIElement**. У вказаних класах визначені члени для роботи зі зміною розмірів, спливаючими підказками, курсорами миші і т. п. Завдяки такому ланцюжку спадкування при візуалізації графічних даних із застосуванням класів, похідних від **Shape**, об'єкти отримуються майже такими ж функціональними (з погляду взаємодії з користувачем), як елементи управління WPF.

Скажімо, для з'ясування, чи клацнув користувач на візуалізованому зображенні, досить обробити подію **MouseDown**. Наприклад, якщо написати наступну розмітку XAML для об'єкта **Rectangle** всередині елемента управління **Grid** початкового вікна **Window**:

```
<Rectangle x:Name="myRect" Height="30" Width="30" Fill="Green"
    MouseDown="myRect_MouseDown"/>
```

то можна реалізувати обробник події **MouseDown**, який змінює колір фону прямокутника в результаті клацання на ньому:

```
private void myRect_MouseDown(object sender, MouseButtonEventArgs e)
{
  // Змінити колір прямокутника в результаті клацання на ньому.
  myRect.Fill = Brushes.Pink;
}
```

На відміну від інших інструментальних наборів для роботи з графікою, вам не доведеться писати громіздкий код інфраструктури, в якому вручну зіставляються координати миші з геометричним об'єктом, з'ясовується попадання курсора всередину меж, виконується візуалізація у невідображуваний буфер і т. д. Члени простору імен **System.Windows.Shapes** просто реагують на зареєстровані вами події подібно до типового елемента управління WPF (**Button** і т. д.).

Недолік всієї готової функціональності пов'язаний з тим, що фігури потребують досить багато пам'яті. Якщо будується науковий застосунок, який малює тисячі точок на екрані, то використання фігур буде невдалим вибором (по суті таким же марнотратним щодо використання пам'яті, як візуалізація тисяч об'єктів **Button**). Тим не менш, коли треба згенерувати інтерактивне двовимірне векторне зображення, фігури виявляються прекрасним варіантом.

Окрім функціональності, успадкованої від батьківських класів **UIElement** і **FrameworkElement**, в класі **Shape** визначено багато власних членів, найбільш корисні з яких коротко описані в табл. 7.1.

Таблиця 7.1. Ключові властивості базового класу **Shape**

Властивості	Опис
DefiningGeometry	Повертає об'єкт **Geometry**, який представляє загальні розміри поточної фігури. Цей об'єкт містить тільки точки, використовувані для візуалізації даних, і не має ніяких слідів функціональності з класу **UIElement** або **FrameworkElement**.
Fill	Дозволяє задати об'єкт кисті для заповнення внутрішньої області фігури.
GeometryTransform	Дозволяє застосовувати трансформацію до фігури до її візуалізації на екрані. Успадкована (від **UIElement**) властивість **RenderTransform** застосовує трансформацію після візуалізації фігури на екрані.
Stretch	Описує, як розмістити фігуру у виділеному їй просторі, наприклад, за її позицією всередині диспетчера компонування. Це здійснюється з використанням відповідного перерахування **System.Windows.Media.Stretch**.
Stroke	Визначає об'єкт кисті або у ряді випадків об'єкт пера (який насправді є замаскованим об'єктом кисті), використовуваний для малювання межі фігури.
StrokeDashArray, StrokeEndLineCap, StrokeStartLineCap, StrokeThickness	Ці (та інші) властивості, пов'язані з штрихами, управляють тим, як конфігуровані лінії при рисуванні меж фігури. У більшості випадків ці властивості конфігуруватимуть кисть, використовувану для малювання межі або лінії.

На замітку! Якщо ви забудете встановити властивості **Fill** і **Stroke**, то WPF надасть «*невидимі*» кисті, внаслідок чого фігуру не буде видно на екрані!

Додавання прямокутників, еліпсів і ліній на поверхню Canvas

Побудуємо застосунок, який здатний візуалізувати фігури, із використанням XAML і C#, і попутно дослідимо процес перевірки попадання. Створимо новий проект застосунку на ім'я **RenderingWithShapes** і змінимо заголовок головного вікна у **MainWindow.xaml** на **Робота з фігурами!**. Модифікуємо первинну розмітку XAML для елемента **Window**, замінивши **Grid** панеллю **DockPanel**, яка містить (поки порожні) елементи **ToolBar** і **Canvas**. Зверніть увагу, що кожному розміщеному елементу, за допомогою властивості **Name** призначається відповідне ім'я.

```
<DockPanel LastChildFill="True">
  <ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  </ToolBar>
  <Canvas Background="LightBlue" Name="canvasDrawingArea"/>
</DockPanel>
```

Заповнимо елемент **ToolBar** об'єктами **RadioButton**, кожен з яких містить об'єкт специфічного класу, похідного від **Shape**. Кожному елементу **RadioButton** призначене *групове* ім'я **GroupName** (щоб забезпечити взаємне виключення) і також *індивідуальне* ім'я.

```
<ToolBar DockPanel.Dock="Top" Name="mainToolBar" Height="50">
  <RadioButton Name="circleOption" GroupName="shapeSelection">
      <Ellipse Fill="Green" Height="35" Width="35" />
  </RadioButton>
  <RadioButton Name="rectOption" GroupName="shapeSelection">
      <Rectangle Fill="Red" Height="35" Width="35" RadiusY="10" RadiusX="10" />
  </RadioButton>
  <RadioButton Name="lineOption" GroupName="shapeSelection">
      <Line Height="35" Width="35" StrokeThickness="10"
          Stroke="Blue" X1="10" Y1="10" Y2="25" X2="25"
          StrokeStartLineCap="Triangle" StrokeEndLineCap="Round" />
  </RadioButton>
</ToolBar>
```

Як бачите, оголошення об'єктів **Rectangle**, **Ellipse** і **Line** в розмітці XAML досить прямолінійне і вимагає лише мінімальних коментарів. Згадайте, що властивість **Fill** дозволяє задати *кисть* для фарбування всередині фігури. Коли потрібна кисть суцільного кольору, можна просто задати жорстко закодований рядок відомих значень, а відповідний перетворювач типу згенерує коректний об'єкт. Цікава характеристика типу **Rectangle** пов'язана з тим, що в ньому визначені властивості **RadiusX** і **RadiusY**, які дозволяють візуалізувати округлені кути.

Об'єкт **Line** представлений своїми початковою і кінцевою точками з використанням властивостей **X1**, **X2**, **Y1** і **Y2** (враховуючи, що *висота* і *ширина* при описі лінії мають мало сенсу). Тут встановлюється декілька додаткових управляючих властивостей, відповідальних за те, як візуалізується початкова і кінцева точки об'єкта **Line**, а також налаштовують параметри штриха. На рис.

7.1 показана візуалізована панель інструментів у візуальному конструкторі WPF Visual Studio.

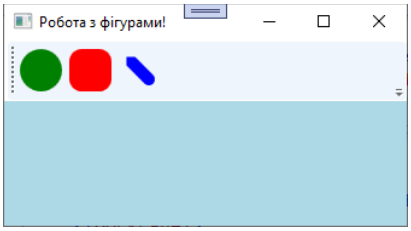


Рис. 7.1. Застосування об'єктів **Shape** у якості вмісту для набору елементів **Radiobutton**

За допомогою вікна **Properties** (Властивості) середовища Visual Studio створимо обробник події **MouseLeftButtonDown** для **Canvas** та обробник події Click для кожного елемента **RadioButton**. Мета полягає в тому, щоб в коді C# візуалізувати вибрану фігуру (круг, квадрат або лінію), коли користувач клацає усередині **Canvas**. Насамперед визначимо наступне вкладене перерахування (і відповідну змінну-член) всередині класу, похідного від **Window**:

```
public partial class MainWindow : Window
{
   private enum SelectedShape { Circle, Rectangle, Line }
   private SelectedShape_currentShape;
   public MainWindow()
   {
      InitializeComponent();
   }
}
```

У кожному обробнику **Click** встановимо змінну-член **currentShape** в коректне значення **SelectedShape**:

```
private void circleOption_Click(object sender, RoutedEventArgs e)
{
   _currentShape = SelectedShape.Circle;
}
private void rectOption_Click(object sender, RoutedEventArgs e)
{
   _currentShape = SelectedShape.Rectangle;
}
private void lineOption_Click(object sender, RoutedEventArgs e)
{
   _currentShape = SelectedShape.Line;
}
```

За допомогою обробника події **MouseLeftButtonDown** елемента **Canvas** візуалізуватиметься відповідна фігура (визначеного розміру) в початковій точці,

яка відповідає позиції **X**, **Y** курсора миші. Нижче приведена повна реалізація з наступним аналізом:

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  Shape shapeToRender = null;
  // Конфігурувати коректну фігуру для малювання.
  switch (_currentShape)
  {
    case SelectedShape.Circle:
      shapeToRender = new Ellipse() { Fill = Brushes.Green, Height = 35, Width = 35 };
        break;
    case SelectedShape.Rectangle:
      shapeToRender = new Rectangle() { Fill = Brushes.Red, Height = 35, Width = 35, RadiusX = 10,
RadiusY = 10 };
        break;
    case SelectedShape.Line:
      shapeToRender = new Line()
      {
        Stroke = Brushes.Blue,
        StrokeThickness = 10,
        X1 = 0,
        X2 = 50,
        Y1 = 0,
        Y2 = 50,
        StrokeStartLineCap = PenLineCap.Triangle,
        StrokeEndLineCap = PenLineCap.Round
      };
        break;
      default:
        return;
  }

  // Встановити верхній лівий кут для малювання на клієнтській частині вікна.
  Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
  Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
  // Намалювати фігуру.
  canvasDrawingArea.Children.Add(shapeToRender);
}
```

> **<u>На замітку!</u>** У методі **canvasDrawingArea_MouseLeftButtonDown()**, створюються об’єкти **Ellipse**, **Rectangle** і **Line**, властивості яких налаштовані так само як і відповідні визначення XAML. Як і можна було очікувати, код можна спростити, але це вимагає розуміння *об'єктних ресурсів* WPF, які розглядатимуться в ***Лекція 9. Ресурси WPF***.

У коді перевіряється змінна-член **currentShape** з метою створення коректного об'єкта, похідного від **Shape**. Потім встановлюються координати лівого верхнього кута всередині **Canvas** з використанням вхідного об'єкта **MouseButtonEventArgs**. Нарешті, в колекцію об'єктів **UIElement**, підтримувану

Canvas, додається новий похідний від **Shape** об'єкт. Якщо запустити програму прямо зараз, то вона повинна дозволити клацати лівою кнопкою миші де завгодно на полотні і візуалізувати у позиції клацання вибрану фігуру (рис. 7.2).

Рис. 7.2. Демонстрація візуалізації вибраних фігур

Видалення прямокутників, еліпсів і ліній з поверхні Canvas

Маючи в розпорядженні елемент **Canvas** з колекцією об'єктів, може виникнути питання: як динамічно видалити елемент, скажімо, у відповідь на клацання користувача правою кнопкою миші на фігурі? Це робиться за допомогою класу **VisualTreeHelper** з простору імен **System.Windows.Media**. Роль «*візуальних дерев*» і «*логічних дерев*» детальніше пояснюється в ***Лекція 12. Логічні і візуальні дерева та шаблони***», а наразі обробимо подію **MouseRightButtonDown** об'єкта **Canvas** і реалізуємо відповідний обробник:

```
private void canvasDrawingArea_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
{
  // Спочатку отримати координати X, Y позиції, де користувач виконав клацання.
  Point pt = e.GetPosition((Canvas)sender);
  // Використати метод HitTest() класу VisualTreeHelper, щоб
  // з'ясувати, чи клацнув користувач на елементі dсередині Canvas.
  HitTestResult result = VisualTreeHelper.HitTest(canvasDrawingArea, pt);
  // Якщо змінна result не дорівнює null, то клацання здійснене на фігурі,
  if (result != null)
  {
    // Отримати фігуру, на якій здійснено клацання, і видалити її з Canvas.
    canvasDrawingArea.Children.Remove(result.VisualHit as Shape);
  }
}
```

Метод **canvasDrawingArea_MouseRightButtonDown()** розпочинається з отримання точних координат **X**, **Y** позиції, де користувач клацнув всередині **Canvas**, і перевірки попадання за допомогою статичного методу **VisualTreeHelper.HitTest()**. Повернуте значення – об'єкт **HitTestResult** – буде встановлено в **null**, якщо користувач виконав клацання не на **UIElement** всередині **Canvas**. Якщо значення **HitTestResult** не дорівнює **null**, тоді за допомогою властивості **VisualHit** можна отримати об'єкт **UIElement**, на якому було здійснено клацання, і привести його до типу, похідного від **Shape** (згадайте, що **Canvas** може містити будь-який **UIElement**, а не тільки фігури).

Деталі, пов'язані з «*візуальним деревом*», будуть викладені в ***Лекція 12. Логічні і візуальні дерева та шаблони***.

> **На замітку!** За замовчуванням метод **VisualTreeHelper.HitTest()** повертає об'єкт **UIElement** самого верхнього рівня, на якому здійснено клацання, і не надає інформацію про інші об'єкти, розташовані під ним (тобто перекритих в **Z**-порядку).

У результаті внесених модифікацій повинна з'явитися можливість додавання фігури на **Canvas** клацанням лівою кнопкою миші і її видалення клацанням правою кнопкою миші.

До цього моменту ми застосовували об'єкти типів, похідних від **Shape**, для візуалізації вмісту елементів **RadioButton** з використанням розмітки XAML і заповнювали **Canvas** в коді C#. Під час дослідження ролі кистей і графічних трансформацій в цей приклад буде додана додаткова функціональність. До речі, в іншому прикладі ілюструватимуться прийоми перетягування на об'єктах **UIElement**. А поки давайте розглянемо інші члени простору імен **System.Windows.Shapes**.

Робота з елементами Polyline і Polygon

У поточному прикладі використовуються тільки три класи, похідні від **Shape**. Інші дочірні класи (**Polyline**, **Polygon** і **Path**) надзвичайно важко коректно візуалізувати без інструментальної підтримки (наприклад, Microsoft Blend, наявний в складі Visual Studio і призначений для розробників WPF, або інші інструменти, які можуть створювати векторну графіку) – просто тому, що вони вимагають визначення великого числа точок для свого остаточного представлення. Вам варто з'ясувати роль інструменту Microsoft Blend, а поки здійснимо короткий огляд інших типів **Shapes**.

Тип **Polyline** дозволяє визначити колекцію координат **X**, **Y** (через властивість **Points**) для малювання послідовності лінійних сегментів, які не вимагають замикання. Тип **Polygon** схожий, але запрограмований так, що завжди замикає контур, сполучаючи початкову точку з кінцевою, і заповнює внутрішню область з допомогою заданої кисті. Припустимо, що в редакторі Kaxaml створений наступний елемент **StackPanel**:

```
<StackPanel>
<!-- Елемент Polyline не замикає автоматично кінцеві точки -->
<Polyline Stroke ="Red" StrokeThickness ="20" StrokeLineJoin ="Round"
    Points ="10, 10 40,40 10,90 300,50"/>
<!-- Елемент Polygon завжди замикає кінцеві точки -->
<Polygon Fill ="AliceBlue" StrokeThickness ="5" Stroke ="Green"
    Points ="40, 10 70, 80 10, 50" />
</StackPanel>
```

На рис. 7.3 показано візуалізований вивід в Kaxaml.

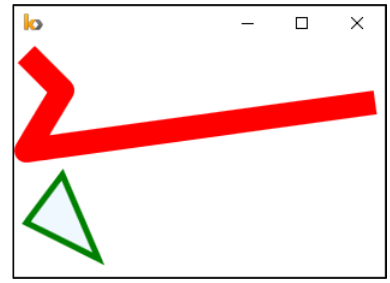


Рис. 7.3. Елементи **Polyline** і **Polygon**

Робота з елементом Path

Застосовуючи тільки типи **Rectangle**, **Ellipse**, **Polygon**, **Polyline** і **Line**, намалювати деталізоване двовимірне векторне зображення було б надто важко, бо згадані примітиви не дозволяють легко фіксувати графічні дані, подібні до кривих, об'єднань даних, які перекриваються, і т. д. Останній похідний від **Shape** клас, **Path**, надає можливість визначення складних двовимірних графічних даних у вигляді колекції незалежних геометричних об'єктів. Після того, як колекція таких геометричних об'єктів визначена, її можна присвоїти властивості **Data** класу **Path**, де вона використовуватиметься для візуалізації складного двовимірного зображення.

Властивість **Data** отримує об'єкт похідного від **System.Windows.Media.Geometry** класу, який містить ключові члени, коротко описані в табл. 7.2.

Таблиця 7.2. Вибрані члени класу **System.Windows.Media.Geometry**

Член	Опис
Bounds	Встановлює поточний обмежуючий прямокутник, який містить геометричний об'єкт.
FillContains()	З'ясовує, чи знаходиться заданий об'єкт **Point** (або інший об'єкт **Geometry**) всередині меж певного класу, похідного від **Geometry**. Це корисно при обчисленнях для перевірки попадання.
GetArea()	Повертає загальну область, зайняту об'єктом похідного від **Geometry** типу.
GetRenderBounds()	Повертає об'єкт **Rect**, який містить найменший з можливих прямокутник, який може бути застосований для візуалізації об'єкта класу, похідного від **Geometry.**
Transform	Призначає геометричному об'єкту екземпляр класу **Transform** для зміни візуалізації.

Класи, які розширюють клас **Geometry** (табл. 7.3), майже схожі на свої аналоги, похідні від **Shape**. Наприклад, клас **EllipseGeometry** має члени, подібні до членів класу **Ellipse**. Значна відмінність пов'язана з тим, що похідні від **Geometry** класи не знають, як візуалізувати себе безпосередньо, бо вони не є **UIElement**. Натомість класи, похідні, від **Geometry** представляють всього лише колекцію даних про точки, яка вказує об'єкту **Path**, як їх візуалізувати.

Таблиця 7.3. Класи, похідні від класу **Geometry**

Клас	Опис
LineGeometry	Представляє пряму лінію.
RectangleGeometry	Представляє прямокутник.
EllipseGeometry	Представляє еліпс.

GeometryGroup	Дозволяє групувати разом декілька об'єктів **Geometry.**
CombinedGeometry	Дозволяє об'єднувати два різні об'єкти **Geometry** в єдину фігуру.
PathGeometry	Представляє фігуру, утворену з ліній і кривих.

У показаній далі розмітці для елемента **Path** використовується декілька похідних від **Geometry** типів. Зверніть увагу, що властивість **Data** об'єкта **Path** встановлюється в об'єкт **GeometryGroup**, який містить об'єкти інших похідних від **Geometry** класів, таких як **EllipseGeometry**, **RectangleGeometry** і **LineGeometry**. Результат представлений на рис. 7.4.

```
<StackPanel>
  <!-- Елемент Path містить набір об'єктів Geometry, встановлений у властивості Data -->
  <Path Fill = "Orange" Stroke = "Blue" StrokeThickness = "3">
    <Path.Data>
      <GeometryGroup>
        <EllipseGeometry Center = "75, 70" RadiusX = "30" RadiusY = "30" />
        <RectangleGeometry Rect = "25, 55 100 30" />
        <LineGeometry StartPoint="0, 0" EndPoint="70, 30" />
        <LineGeometry StartPoint="70, 30" EndPoint="0, 30" />
      </GeometryGroup>
    </Path.Data>
  </Path>
</StackPanel>
```

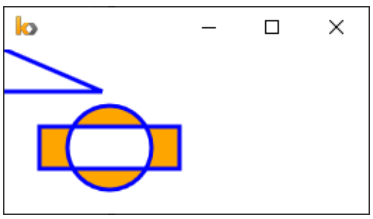


Рис. 7.4. Елемент **Path**, з різноманітними об'єктами **Geometry**

Зображення на рис. 7.4 може бути візуалізовано із використанням розглянутих раніше класів, **Line**, **Ellipse** і **Rectangle**. Проте це вимагало б поміщення різних об'єктів **UIElement** в пам'ять. Коли для моделювання точок мальованого зображення використовуються геометричні об'єкти, а потім колекція геометричних об'єктів поміщається в контейнер, який здатний візуалізувати дані (**Path** у цьому випадку), то тим самим скорочується витрата пам'яті.

Згадаймо, що клас **Path** має той же ланцюжок спадкування, як і будь-який член простору **System.Windows.Shapes**, і може відправляти такі ж повідомлення про події, як інші елементи **UIElement**. Отже, якщо визначити той же самий елемент <**Path**> у проекті Visual Studio, тоді з'ясувати, що користувач клацнув у будь-якому місці лінії, можна буде за рахунок обробки події миші (не забувайте, що редактор Kaxaml не дозволяє обробляти події для написаної розмітки).

«Міні-мова» моделювання шляхів

З усіх класів, перерахованих в табл. 7.3, клас **PathGeometry** найскладніший для конфігурації в термінах XAML і коду. Це пояснюється тим фактом, що кожен сегмент **PathGeometry** складається з об'єктів, які містять різноманітні сегменти і фігури (скажімо, **PolyQuadraticBezierSegment**, **ArcSegment**, **BezierSegment**, **LineSegment**, **PolyBezierSegment**, **PolyLineSegment**, і т. д.). Ось приклад об'єкта **Path**, властивість **Data** якого було встановлено в елемент **PathGeometry**, що складається з різних фігур і сегментів:

```
<StackPanel>
  <Path Stroke="Black" StrokeThickness="1">
    <Path.Data>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="10,50">
            <PathFigure.Segments>
              <BezierSegment Point1="100,0" Point2="200,200" Point3="300,100"/>
              <LineSegment Point="400,100" />
              <ArcSegment Size="50,50" RotationAngle="45" IsLargeArc="True"
                  SweepDirection="Clockwise" Point="200,100"/>
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
</StackPanel>
```

Результат виконання коду розмітки зображений на рис. 7.5.

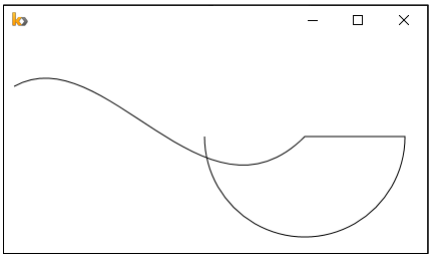


Рис. 7.5. Результат роботи коду вище

Насправді, лише небагатьом програмістам доведеться коли-небудь вручну будувати складні двовимірні зображення, безпосередньо описуючи об'єкти похідних від **Geometry** або **PathSegment** класів. Пізніше ви довідаєтеся, як перетворювати векторну графіку в оператори «*міні-мови*» моделювання шляхів, які можна застосовувати в розмітці XAML.

Навіть з врахуванням сприяння, з боку згаданих раніше інструментів, обсяг розмітки XAML, потрібної для задання складних об'єктів **Path**, буває надто великим, бо дані складаються з повних описів різних об'єктів класів,

похідних від **Geometry** або **PathSegment**. Для того, щоб створювати лаконічнішу розмітку, в класі **Path** підтримується спеціалізована «*міні-мова*».

Наприклад, замість встановлення властивості **Data** об'єкта **Path** в колекцію об'єктів похідних від **Geometry** і **PathSegment** класів їх можна задати в одиночний рядковий літерал, який містить набір відомих символів і різних значень, що визначають фігуру, яка підлягає візуалізації. Нижче наведений простий приклад, а його результуючий вивід показаний на рис. 7.6:

```
<Path Stroke="Black" StrokeThickness="3" Data="M 10,75 C 70,15 250, 270 300,175 H 240" />
```

Команда **M** (**move** – перемістити) приймає координати **X**, **Y**, які задають початкову точку малювання. Команда **C** (**curve** – крива) приймає послідовність точок для візуалізації кривої (кубічної кривої **Безьє**), а команда **H** (**horizontal** – горизонталь) малює горизонтальну лінію.

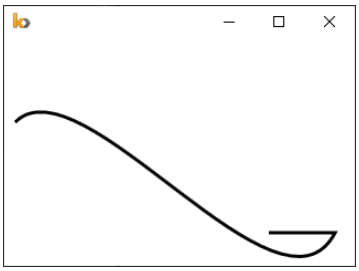


Рис. 7.6. «Міні-мова» моделювання шляхів дозволяє компактно описувати об'єктну модель **Geometry/PathSegment**

На замітку! Криві Безьє (названі на честь інженера П'єра Безьє) всюди застосовуються в комп'ютерній графіці для зображення гладких кривих. Вони навіть використовуються в шрифтах для математичного опису форми гліфів[16].

Основна ідея полягає в тому, що окрім двох кінцевих точок у кривої Безьє є одна або декілька управляючих точок, які і надають прямолінійному відрізку форми кривої. Управляючі точки, невидимі (і можуть не лежати на самій кривій), а лише входять у формулу, яка описує криву. Інтуїтивно можна уявляти собі, що управляюча точка, є центром тяжіння, тобто відрізок як би «*притягується*» до цих точок.

Незважаючи на лякаючу назву, клас **QuadraticBezierSegment** насправді простіший, ніж **BezierSegment**, і менш затратний з погляду обсягу обчислень. Але квадратична крива Безьє може приймати тільки **U**-подібну форму (або бути відрізком прямої), тоді як кубічна – **S**-подібну форму.

[16] **Гліф** – елемент письма, конкретне графічне представлення *графеми*, іноді кількох пов'язаних *графем* (складений гліф), або тільки частини *графеми* (наприклад, *діакритичний знак*). Два або більше гліфи, представляючи один і той же символ, використовувані попеременно або вибрані, залежно від контексту, називаються *алографами* один одного. І якщо *графема* – це одиниця тексту, то *гліф* – одиниця графіки.

І знову треба зазначити, що вам дуже рідко доведеться вручну будувати або аналізувати рядковий літерал, який містить інструкції міні-мови моделювання шляхів. Проте, мета в тому, щоб розмітка XAML, генерована спеціалізованими інструментами, не здавалася абсолютно незрозумілою. Деталі цієї конкретної граматики, можна отримати в розділі «**Path Markup Syntax**» («Синтаксис розмітки шляхів») документації https://docs.microsoft.com/en-us/dotnet/desktop/wpf/graphics-multimedia/path-markup-syntax?view=netframeworkdesktop-4.8.

Кисті і пера WPF

Кожен спосіб графічної візуалізації (фігури, малювання і геометричні об'єкти, а також візуальні об'єкти) інтенсивно використовує кисті, які дозволяють управляти заповненням внутрішньої області двовимірної фігури. У WPF надаються шість різних типів кистей, і всі вони розширюють клас **System.Windows.Media.Brush**. Незважаючи на те що клас **Brush** абстрактний, його нащадки, описані в табл. 7.4, можуть застосовуватися для заповнення області вмістом майже будь-якого мислимого вигляду.

Таблиця 7.4. Класи, похідні від **Brush**

Клас	Опис
DrawingBrush	Заповнює область за допомогою об'єкта похідного від **Drawing** класу (**GeometryDrawing**, **ImageDrawing** або **VideoDrawing**).
OmageBrush	Заповнює область зображенням (представленим за допомогою об'єкта **ImageSource**).
LinearGradientBrush	Заповнює область лінійним градієнтом.
RadialGradientBrush	Заповнює область радіальним градієнтом.
SolidColorBrush	Заповнює область суцільним кольором, вказаним у властивості **Color.**
VisualBrush	Заповнює область за допомогою об'єкта похідного від **Visual** класу (**DrawingVisual**, **Viewport3DVisual** і **ContentVisual**).

Класи **DrawingBrush** і **VisualBrush** дозволяють будувати кисть на основі існуючого класу, похідного від **Drawing** або **Visual**. Такі класи кистей використовуються при роботі з двома іншими способами візуалізації графіки WPF (малюнками або візуальними об'єктами) і пояснюватимуться далі в лекції.

Клас **ImageBrush** дозволяє будувати кисть, яка відображає дані зображення із зовнішнього файла або вбудованого ресурсу застосунку, який заданий в його властивості **ImageSource**. Інші типи кистей (**LinearGradientBrush** і **RadialGradientBrush**) досить прості у використанні, хоча потрібна розмітка XAML може виявитися багатослівною. На щастя, в середовищі Visual Studio підтримуються інтегровані редактори кистей, які полегшують завдання генерації стилізованих кистей.

Конфігурування кистей з використанням Visual Studio

Давайте тепер оновимо застосунок WPF для малювання **RenderingWithShapes**, щоб використати в ньому цікавіші кисті. У трьох фігурах, які були задіяні досі при візуалізації даних в панелі інструментів, застосовуються прості суцільні кольори, так що їх значення можна зафіксувати за допомогою простих рядкових літералів. Щоб зробити завдання трохи

цікавішим, тепер ми використовуватимемо інтегрований редактор кистей. Переконавшись в тому, що редактор XAML початкового вікна відкритий в IDE-середовищі, виберемо елемент **Ellipse**. У вікні **Properties** відшукаємо категорію **Brush** (Кисть) і клацнемо на властивості **Fill** (рис. 7.7).

У верхній частині редактора кистей знаходиться набір властивостей, які «*сумісні з кистю*» для вибраного елемента (т. т. **Fill**, **Stroke** та **OpacityMask**). Під ними розміщений набір вкладок, які дозволяють конфігурувати різні типи кистей, включаючи поточну *одноколірну кисть* (кисть з суцільним кольором). Для управління кольором поточної кисті можна використовувати інструмент вибору кольору, а також повзунки **ARGB** (**alpha**, **red**, **green**, **blue** – прозорість, червоний, зелений, синій). За допомогою цих повзунків і пов'язаній з ними області вибору кольору можна створювати суцільний колір будь-якого виду. Використаємо вказані інструменти для зміни кольору у властивості **Fill** елемента **Ellipse** і переглянемо результуючу розмітку XAML. Як бачите, колір зберігається у вигляді шістнадцяткового значення:

```
<Ellipse Fill="#FF47CE47" Height="35" Width="35" />
```

Цей же редактор дозволяє конфігурувати і *градієнтні кисті*, які використовуються для визначення послідовностей кольорів і точок переходу кольорів. Редактор кистей пропонує набір вкладок, перша з яких дозволяє встановити *порожню кисть* для відсутнього візуалізованого виводу. Інші чотири дають можливість встановити кисть *суцільного кольору* (щойно продемонстровану), *градієнтну* кисть, *мозаїчну* кисть і кисть із *зображенням*.

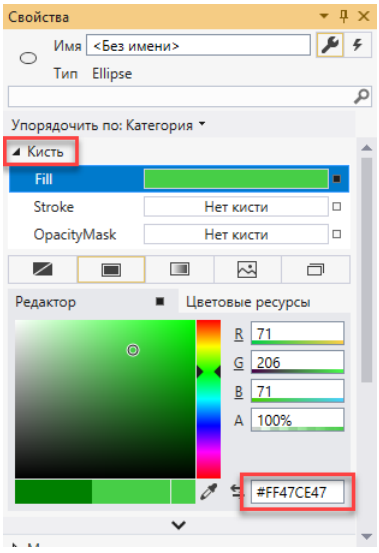


Рис. 7.7. Будь-яка властивість, яка вимагає кисті, може бути конфігурована за допомогою інтегрованого редактора кистей

Клацнемо на вкладці градієнтної кисті; редактор відобразить нові налаштування (рис. 7.8).

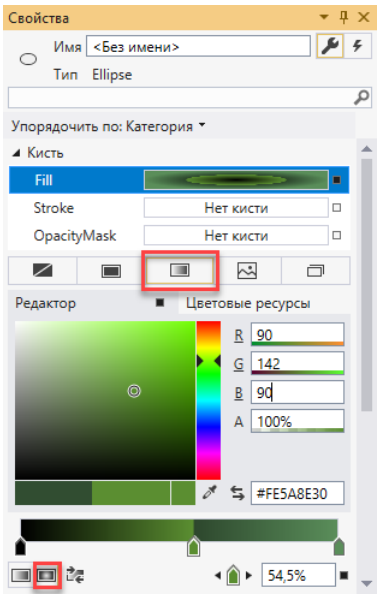


Рис. 7.8. Редактор кистей Visual Studio
дозволяє будувати базові градієнтні кисті

Три кнопки в нижньому лівому кутку дозволяють вибрати *лінійний* градієнт, *радіальний* градієнт або здійснити *реверс* градієнтних переходів. Смуга внизу покаже поточний колір кожного градієнтного переходу, який буде представлений спеціальним повзунком. Перетягуючи повзунок по смузі градієнта, можна управляти *зміщенням* градієнта. Крім того, клацаючи на конкретному повзунку, можна змінювати колір певного градієнтного переходу за допомогою *селектора* кольору. Нарешті, клацання прямо на смузі градієнта дозволяє додавати додаткові градієнтні переходи.

Потратьте деякий час на опанування цього редактора. Ми побудуємо радіальну градієнтну кисть, яка містить три градієнтні переходи, і встановимо їх кольори. На рис. 7.8 показаний приклад кисті, яка використовує три різні відтінки зеленого кольору.

У результаті IDE-середовище оновить розмітку XAML, додавши набір спеціальних кистей і присвоївши їх сумісним з кистями властивостям (властивість **Fill** елемента **Ellipse** у цьому прикладі) із застосуванням синтаксису "*властивість-елемент*":

```
<Ellipse Height="35" Width="35" >
  <Ellipse.Fill>
    <RadialGradientBrush>
      <GradientStop Color="Black"/>
      <GradientStop Color="#FF5A8E5A" Offset="1"/>
      <GradientStop Color="#FF2D472D" Offset="0.547"/>
      <GradientStop Color="#FE5A8E5A" Offset="0.545"/>
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Конфігурування кистей в коді

Тепер, коли ми побудували спеціальну кисть для визначення XAML-елемента **Ellipse**, відповідний код C# застарів, бо він як і раніше візуалізуватиме круг з суцільним зеленим кольором. Дня відновлення синхронізації треба модифікувати потрібний оператор **case**, щоб використати щойно створену кисть. Нижче показано потрібне оновлення, яке виглядає складнішим, ніж можна було очікувати, бо шістнадцяткове значення перетвориться у відповідний об'єкт **Color** за допомогою класу **System.Windows.Media.ColorConverter** (результат зміни представлений на рис. 7.9):

```
case SelectedShape.Circle:
  shapeToRender = new Ellipse() { Height = 35, Width = 35 };
  // Створити кисть RadialGradientBrush в коді.
  RadialGradientBrush brush = new RadialGradientBrush();
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF77F177"), 0));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF11E611"), 1));
  brush.GradientStops.Add(new GradientStop(
    (Color)ColorConverter.ConvertFromString("#FF5A8E5A"), 0.545));
  shapeToRender.Fill = brush;
  break;
```

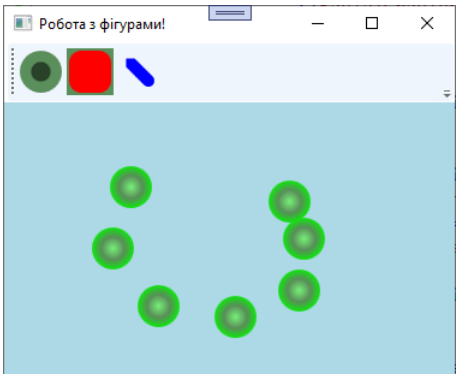


Рис. 7.9. Малювання цікавіших кругів

Об'єкти **GradientStop** можна будувати, вказуючи простий колір як перший параметр конструктора із застосуванням перерахування **Colors**, яке дає конфігурований об'єкт **Color**:

```
GradientStop g = new GradientStop(Colors.Aquamarine, 1);
```

Якщо потрібний тонший контроль, то можна передавати об'єкт **Color**, конфігурований в коді, наприклад так:

```
Color myColor = new Color() { R = 200, G = 100, B = 20, A = 40 };
GradientStop g = new GradientStop(myColor, 34);
```

Зрозуміло, використання перерахування **Colors** і класу **Color** не обмежується градієнтними кистями. Їх можна застосовувати всякий раз, коли потрібно представити значення кольору в коді.

Конфігурування пер

Порівняно з кистями *перо* є об'єктом для малювання меж геометричних об'єктів або у разі класу **Line** або **PolyLine** – справжнього лінійного геометричного об'єкта. Зокрема, клас **Pen** дозволяє малювати лінію заданої товщини, представленої значенням типу **double**. На додаток об'єкт **Pen** може бути конфігурований за допомогою того ж самого виду властивостей, що і в класі **Shape**, таких як початковий і кінцевий кінці пера, шаблони точок-тире і т. д. Наприклад, для визначення атрибутів пера до визначення фігури можна додати наступну розмітку:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle" StartLineCap="Round" />
```

У багатьох випадках створювати об'єкт **Pen** безпосередньо не доведеться, тому що це здійснюється опосередковано, коли присвоюється значення властивостям на зразок **StrokeThickness** похідного від **Shape** типу (а також інших типів **UIElement**). Проте будувати спеціальний об'єкт **Pen** зручно при роботі з типами, похідними від **Drawing** (які розглядаються трохи пізніше). Середовище Visual Studio не має в розпорядженні редактора пер як таких, але дозволяє конфігурувати всі властивості, пов'язані із штрихами, для вибраного елемента з використанням вікна **Properties**.

> **На замітку!** Через те, що кисть **Brush** і перо **Pen** задаються на рівні **Drawing**, а не **Geometry**, група **GeometryGroup** не дозволяє об'єднувати фігури з різними заливками або контурами. Щоб досягти цього, треба скористатися класом **DrawingGroup**, який дозволяє об'єднувати декілька малюнків (кожен з яких може складатися з одного або декількох геометричних об'єктів).

> **На замітку!** На відміну від об'єктів **UIElement**, які можуть мати єдиного батька, об'єкти класів **Geometry**, **PathFigure** і пов'язаних з ними допустимо використовувати спільно. Їх усуспільнення може дати помітний виграш в продуктивності, особливо для складних геометричних об'єктів. Якщо не передбачається ніяких змін, то є сенс їх заморозити і тим самим ще підвищити продуктивність.

Застосування графічних трансформацій

На завершення обговорення фігур розглянемо тему трансформацій. Інфраструктура WPF постачається з численними класами, які розширюють абстрактний базовий клас **System.Winodws.Media.Transform**. У табл. 7.5 описані основні класи, похідні від **Transform**.

Таблиця 7.5. Основні класи, похідні від **System.Windows.Media.Transform**

Клас	Опис
MatrixTransform	Створює довільну матричну трансформацію, яка використовується для маніпулювання об'єктами або координатними системами на двовимірній площині.
RotateTransform	Повертає об'єкт за годинниковою стрілкою навколо заданої точки в двовимірній системі координат (**X**, **Y**).
ScaleTransform	Масштабує об'єкт в двовимірній системі координат (**X**, **Y**).
SkewTransform	Здійснює нахил об'єкта в двовимірній системі координат (**X**, **Y**).
TranslateTransform	Перетворить (переміщає) об'єкт в двовимірній системі координат (**X**, **Y**).
TransformGroup	Представляє комбінований об'єкт **Transform**, який складається з інших об'єктів **Transform**.

Трансформації можуть застосовуватися до будь-яких об'єктів **UIElement** (наприклад, до об'єктів похідних від **Shape** класів, а також до елементів управління **Button**, **TextBox** і т. п.). Використовуючи класи трансформацій, можна візуалізувати графічні зображення, нахиляючи їх під заданим кутом, розтягуючи, стискаючи або повертаючи цільовий елемент різними способами.

> **На замітку!** Хоча об'єкти трансформацій можна застосовувати всюди, вони можуть бути найзручнішими при роботі з анімацією WPF і спеціальними шаблонами елементів управління. Анімацію WPF можна використати для включення в спеціальний елемент управління візуальних підказок, призначених кінцевому користувачеві.

Призначати цільовому об'єкту (**Button**, **Path** і т. д.) трансформацію (або їх набір) можна за допомогою двох загальних властивостей, **LayoutTransform** і **RenderTransform**.

Властивість **LayoutTransform** зручна тим, що трансформація відбувається перед візуалізацією елементів в диспетчері компонування і тому не впливає на операції **Z**-впорядковування (тобто трансформовані дані зображень не перекриваються).

З іншого боку, трансформація з властивості **RenderTransform** ініціюється після того, як елементи потрапили у свої контейнери, тому цілком можливо, що елементи будуть трансформовані з перекриттям один одного залежно від того, як вони організовані в контейнері.

Перший погляд на трансформації

Незабаром ми додамо до проекту **RenderingWithShapes** деяку, трансформуючу логіку. Щоб побачити об'єкт трансформації у дії, відкриємо редактор Kaxaml, визначимо всередині кореневого елемента **Page** або **Window** групуючий елемент **StackPanel** і встановимо властивість **Orientation** в **Horizontal**. Далі додамо наступний елемент **Rectangle**, який буде намальований під кутом в **45** градусів із застосуванням об'єкта **RotateTransform**:

```
<StackPanel Orientation="Horizontal">
  <!-- Елемент Rectangle з трансформацією поворотом -->
  <Rectangle Height ="100" Width ="40" Fill ="Red" >
    <Rectangle.LayoutTransform>
```

```
      <RotateTransform Angle ="45"/>
    </Rectangle.LayoutTransform>
  </Rectangle>
</StackPanel>
```

Наступний елемент **Button** скошується на поверхні на **20** градусів за допомогою трансформації **SkewTransform**:

```
<!-- Елемент Button з трансформацією скошуванням -->
<Button Content ="Click Me!" Width="95" Height="40">
  <Button.LayoutTransform>
    <SkewTransform AngleX ="20" AngleY ="20"/>
  </Button.LayoutTransform>
</Button>
```

Для повноти нижче приведений елемент **Ellipse**, масштабований на **20%** за допомогою трансформації **ScaleTransform** (зверніть увагу на значення, встановлені у властивостях **Height** і **Width**), а також елемент **TextBox**, до якого застосована група об'єктів трансформації:

```
<!-- Елемент Ellipse, масштабований на 20% -->
<Ellipse Fill="Blue" Width="5" Height="5">
  <Ellipse.LayoutTransform>
    <ScaleTransform ScaleX ="20" ScaleY ="20"/>
  </Ellipse.LayoutTransform>
</Ellipse>
<!-- Елемент TextBox, повернений і скошений -->
<TextBox Text ="Me Too!" Width="50" Height="40">
  <TextBox.LayoutTransform>
    <TransformGroup>
      <RotateTransform Angle ="45"/>
      <SkewTransform AngleX ="5" AngleY ="20"/>
    </TransformGroup>
  </TextBox.LayoutTransform>
</TextBox>
```

Треба зазначити, що у разі застосування трансформації виконувати якінебудь ручні обчислення для реагування на перевірку попадання, переміщення фокуса вводу та аналогічні дії не потрібно. Графічний механізм WPF самостійно вирішує такі завдання. Наприклад, на рис. 7.10 можна бачити, що елемент **TextBox** як і раніше реагує на клавіатурний ввід.

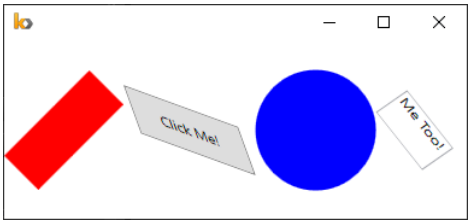


Рис. 7.10. Результат застосування об'єктів графічних трансформацій

Трансформація даних Canvas

Застосуємо в проекті **RenderingWithShapes** логіку трансформації. Окрім застосування об'єктів трансформації до одиночного елемента (**Rectangle**, **TextBox** і т. д.) їх можна також застосовувати до диспетчера компонування, щоб трансформувати всі внутрішні дані. Наприклад, всю панель **DockPanel** головного вікна можна було б візуалізувати під кутом:

```
<DockPanel LastChildFill="True">
  <DockPanel.LayoutTransform>
    <RotateTransform Angle="45"/>
  </DockPanel.LayoutTransform>
</DockPanel>
```

Цей приклад дещо незвичний, тому додамо останню (не радикальну) можливість, яка дозволить користувачеві дзеркально відобразити цілий контейнер **Canvas** і всю його графіку. Розпочнемо з додавання в **ToolBar** фінального елемента **ToggleButton** з наступним визначенням:

```
<ToggleButton Name="flipCanvas" Click="flipCanvas_Click" Content="Flip Canvas!" />
```

Всередині обробника події **Click** для нового елемента **ToggleButton** створимо об'єкт **RotateTransform** і підключимо його до об'єкта **Canvas** через властивість **LayoutTransform**, якщо елемент **ToggleButton** відмічений. Якщо ж елемент **ToggleButton** не відмічений, тоді ми видалимо трансформацію, встановивши властивість **LayoutTransform** в **null**.

```
private void flipCanvas_Click(object sender, RoutedEventArgs e)
{
  if (flipCanvas.IsChecked == true)
  {
    RotateTransform rotate = new RotateTransform(-180);
    canvasDrawingArea.LayoutTransform = rotate;
  }
  else
  {
    canvasDrawingArea.LayoutTransform = null;
  }
}
```

Запустивши застосунок, додамо кілька графічних фігур в область **Canvas**, стежачи за тим, щоб вони знаходилися упритул до її країв. Після клацання на новій кнопці виявиться, що фігури виходять за межі **Canvas** (рис. 7.11). Причина – не був визначений прямокутник відсікання.

Виправити проблему легко. Замість того щоб вручну писати складну логіку відсікання, просто встановимо властивість **ClipToBounds** елемента **Canvas** в **true**, запобігши візуалізації дочірніх елементів поза межами

батьківського елемента. Після запуску застосунку можна помітити, що графічні дані більше не покидають меж відведеної області.

```
<Canvas ClipToBounds="True" Background="LightBlue" Name="canvasDrawingArea"
    MouseRightButtonDown="canvasDrawingArea_MouseRightButtonDown"
    MouseLeftButtonDown="canvasDrawingArea_MouseLeftButtonDown" />
```

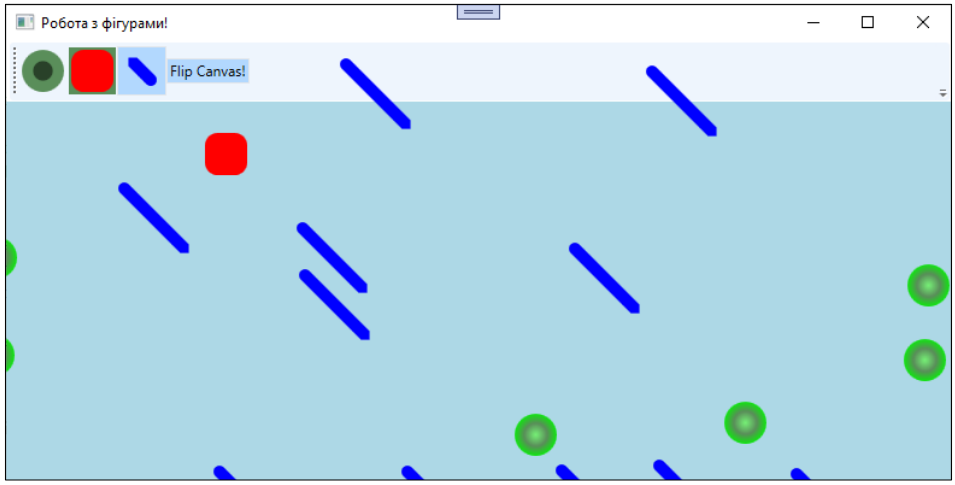


Рис. 7.11. Після трансформації фігури виходять за межі **Canvas**

Остання незначна модифікація, яку потрібно внести, пов'язана з тим фактом, що коли користувач дзеркально відображає клієнтську частину вікна, клацаючи на кнопці перемикання ("Flip Canvas!"), а потім клацає у клієнтській частині вікна для малювання нової фігури, то точка, де було здійснено клацання, *не є* тією позицією, куди потраплять графічні дані. Натомість вони з'являться в місці знаходження курсора миші.

Щоб усунути проблему, застосуємо той же самий об'єкт трансформації до мальованої фігури перед виконанням візуалізації (через **RenderTransform**). Ось основний фрагмент коду:

```
private void canvasDrawingArea_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  …
  if (flipCanvas.IsChecked == true)
  {
    RotateTransform rotate = new RotateTransform(-180);
    shapeToRender.RenderTransform = rotate;
  }
  // Встановити верхній лівий кут для малювання у вікні.
  Canvas.SetLeft(shapeToRender, e.GetPosition(canvasDrawingArea).X);
  Canvas.SetTop(shapeToRender, e.GetPosition(canvasDrawingArea).Y);
  // Намалювати фігуру.
  canvasDrawingArea.Children.Add(shapeToRender);
}
```

Дослідження простору імен **System.Windows.Shapes**, кистей і трансформацій завершено. Перед аналізом ролі візуалізації графіки з малюнками і геометричними об'єктами, з'ясуємо, як IDE-середовище Visual Studio може спростити роботу з примітивними графічними елементами.

Робота з редактором трансформацій Visual Studio

У попередньому прикладі трансформації виконувалися за рахунок ручного вводу розмітки і написання коду C#. Але, треба відмітити, не заперечуючи, можливість ручного вводу, останні версії Visual Studio постачаються із вбудованим редактором трансформацій. Згадайте, що одержувачем служб трансформацій може бути будь-який елемент інтерфейсу, у тому числі і диспетчер компонування, з різними елементами управління. Для демонстрації роботи з редактором трансформацій, ми створимо проект застосунку WPF по імені **FunWithTransforms**.

Побудова початкового компонування

Насамперед розділимо первинний елемент **Grid** на дві колонки із застосуванням вбудованого редактора сітки (точні розміри колонок ролі не грають). Далі додамо елемент управління **StackPanel**, щоб він зайняв увесь простір першої колонки **Grid**; потім додамо в панель **StackPanel** три елементи управління **Button**:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <StackPanel Grid.Row="0" Grid.Column="0">
    <Button Name="btnSkew" Content="Skew" Click="Skew"/>
    <Button Name="btnRotate" Content="Rotate" Click="Rotate"/>
    <Button Name="btnFlip" Content="Flip" Click="Flip"/>
  </StackPanel>
</Grid>
```

Додамо обробники для кнопок:

```
private void Skew(object sender, RoutedEventArgs e)
{

}
private void Rotate(object sender, RoutedEventArgs e)
{

}
private void Flip(object sender, RoutedEventArgs e)
{

}
```

Завершимо інтерфейс, створивши в другій колонці елемента **Grid**, довільну графіку (використавши будь-який прийом, розглянутий раніше). Нехай, розмітка, буде така:

```
<Canvas x:Name="myCanvas" Grid.Column="1" Grid.Row="0">
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top" Height="186" Width="92"
       Stroke="Black" Canvas.Left="20" Canvas.Top="31">
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="#FF951ED8" Offset="0.215"/>
        <GradientStop Color="#FF2FECB0" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse HorizontalAlignment="Left" VerticalAlignment="Top" Height="101" Width="110"
       Stroke="Black" Canvas.Left="122" Canvas.Top="126">
    <Ellipse.Fill>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FFB91DDC" Offset="0.355"/>
        <GradientStop Color="#FFB0381D" Offset="1"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Canvas>
```

Остаточне компонування показане на рис. 7.12.

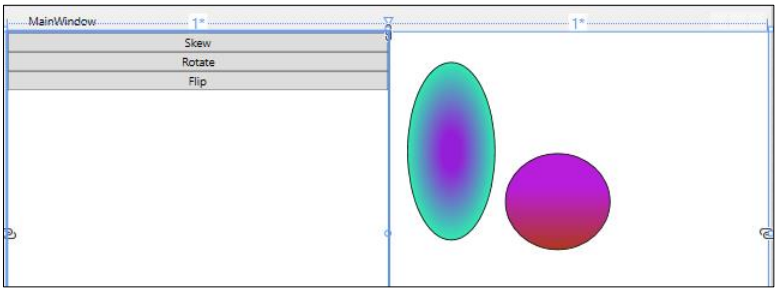


Рис. 7.12. Остаточне компонування в прикладі трансформації

Застосування трансформацій на етапі проектування

IDE-середовище Visual Studio надає вбудований редактор трансформацій, який можна знайти у вікні **Properties**. Розкриємо розділ **Transform** (Трансформація), щоб відобразити області **RenderTransform** і **LayoutTransform** редактора (рис. 7.13).

Подібно до розділу **Brush** розділ **Transform** пропонує декілька вкладок, призначених для конфігурації різноманітних типів графічної трансформації поточного вибраного елемента. У табл. 7.6 описані варіанти трансформації, доступні на цих вкладках (в порядку зліва направо).

Таблиця 7.6. Варіанти трансформації

Трансформація	Опис
Translate (Трансляція)	Дозволяє зрушити місце розташування елемента в позицію **X**,**Y**.
Rotate (Поворот)	Дозволяє повернути елемент на кут до **360** градусів.
Scale (Масштабування)	Дозволяє збільшити або зменшити елемент на масштабний коефіцієнт в напрямах **X** і **Y**.
Skew (Перекошування)	Дозволяє нахилити обмежуючий прямокутник, який містить вибраний елемент, на масштабний коефіцієнт в напрямах **X** і **Y**.
Center Point (Центральна точка)	При повороті або дзеркальному відображенні об'єкта елемент переміщається відносно фіксованої точки, яка називається центральною точкою об'єкта. За замовчуванням центральна точка об'єкта розташована в центрі об'єкта; проте, ця трансформація дозволяє змінювати центральну точку об'єкта для виконання повороту або дзеркального відображення відносно іншої точки.
Flip (Дзеркальне відображення)	Дозволяє дзеркально відобразити вибраний елемент на основі координати **X** або Y центральної точки.

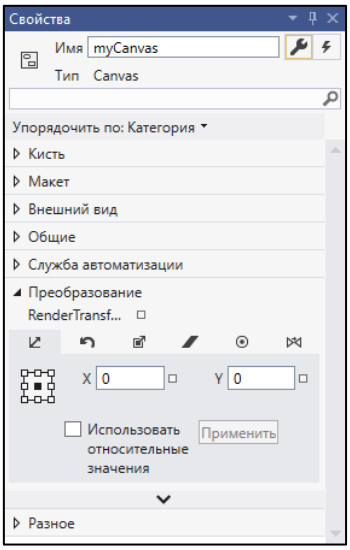


Рис. 7.13. Редактор трансформацій

Випробуйте кожну з описаних трансформацій, використовуючи як мету спеціальну фігуру (для відміни виконаної операції просто натискайте <**Ctrl+Z**>). Як і багато інших аспектів розділу **Transform** вікна **Properties**, кожна трансформація має унікальний набір параметрів конфігурації, які повинні стати цілком зрозумілими, як тільки ви переглянете їх. Наприклад, редактор трансформації **Skew** дозволяє встановлювати значення *нахилу* **X** і **Y**, а редактор трансформації **Flip** дає можливість *дзеркально* відображати відносно осі **X** або **Y** і т. д.

Трансформація клієнтської частини вікна в коді

Реалізації обробників для всіх кнопок будуть більш-менш схожими. Ми сконфігуруємо об'єкт трансформації і присвоїмо його об'єкту **myCanvas**. Після запуску застосунку можна буде клацати на кнопці, щоб переглядати результат

застосування трансформації. Нижче приведений повний код обробників (зверніть увагу на встановлення властивості **LayoutTransform**, що дозволяє даним фігури позиціонуватися відносно батьківського контейнера):

```
private void Skew(object sender, RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new SkewTransform(40, -20);
}

private void Rotate(object sender, RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new RotateTransform(180);
}

private void Flip(object sender, RoutedEventArgs e)
{
  myCanvas.LayoutTransform = new ScaleTransform(-1, 1);
}
```

Резюме

Ми переконалися, що WPF насичена графічною інфраструктурою для побудови графічних інтерфейсів користувача. В цій лекції ми розглянули один зі способів візуалізації графічного виводу. Тут ми розглянули два підходи до візуалізації – фігури, малюнки і візуальні об'єкти – і використання примітивів візуалізації, таких як кисті, пера і трансформації.

Контрольні питання

1. Які способи візуалізації графічних даних надає інфраструктура WPF?
2. Графіка режиму збереження – що це означає в контексті графічної візуалізації інфраструктури WPF?
3. Яка різниця між графічними системами прямого режиму Windows Forms та режимом збереження WPF?
4. В якому класі зосереджені класи для візуалізації двовимірних геометричних фігур?
5. В яких класах зосереджені графічні дані похідні від абстрактного класу System.Windows.Media.Drawing? В чому перевага їх використання?
6. Із застосуванням якого класу можна реалізувати візуалізацію шляхом взаємодії безпосередньо з графічною підсистемою WPF?
7. З яких причин інфраструктура WPF пропонує три індивідуальних підходи до візуалізації графічних даних?
8. Які ключові властивості базового класу Shape нащадка класів UIElement та FrameworkElement?
9. Нащадками якого класу є дочірні класи Rectangle, Ellipse, Polygon, Polyline, Line і Path?
10. Для чого використовується клас Path?
11. Які ключові члени містить клас System.Windows.Media.Geometry?
12. Які класи похідні від класу Geometry?
13. В чому відмінність між класами Shape і Geometry?

14. У яких випадках використовується «міні-мова» моделювання шляхів?
15. Для чого потрібні кисті і пера?
16. Скільки різних типів кистей наявні у WPF?
17. Яким вмістом можуть заповнюватися області нащадками класу Brush?
18. Як конфігуруються кисті з використанням можливостей Visual Studio?
19. Який об’єкт використовують для рисування меж геометричних об’єктів?
20. Які основні класи похідні від System.Windows.Media.Transform реалізують трансформацію?
21. Де найчастіше використовують об’єкти трансформацій?
22. Редактор трансформацій – як ним користуватися?
23. Які варіанти трансформацій наявні в редакторі трансформацій?

Лекція 8. Візуалізація графічних даних

Незважаючи на те що типи **Shape** дозволяють генерувати інтерактивну двовимірну поверхню будь-якого виду, через насичений ланцюжок спадкування вони споживають досить багато пам'яті. І хоча клас **Path** може допомогти зменшити накладні витрати за рахунок застосування включених геометричних об'єктів (замість великої колекції інших фігур), інфраструктура WPF надає розвинений API-інтерфейс рисування і геометрії, який візуалізує ще менш затратні двовимірні векторні зображення.

Вхідною точкою в цей API-інтерфейс є абстрактний клас **System.Windows.Media.Drawing** (зі збірки **PresentationCore.dll**), який сам по собі всього лише визначає обмежуючий прямокутник для зберігання результатів візуалізації.

Інфраструктура WPF пропонує різні класи, які розширюють **Drawing**, кожен з яких представляє окремий спосіб малювання вмісту (табл. 8.1).

Таблиця 8.1. Класи, похідні від **Drawing**

Клас	Опис
DrawingGrdoup	Використовується для комбінування колекції окремих об'єктів, похідних від **Drawing**, в єдину об'єднану візуалізацію.
GeometryDrawing	Застосовується для візуалізації двовимірних фігур в дуже економний, щодо витрат пам'яті, спосіб.
GlyphRunDrawing	Використовується для візуалізації текстових даних із застосуванням служб графічної візуалізації WPF.
ImageDrawing	Використовується для візуалізації файла зображення, або набору геометричних об'єктів, всередині обмежуючого прямокутника.
VideoDrawing	Застосовується для відтворення аудіо- або відеофайла. Цей тип може повноцінно використовуватися тільки в *процедурному* коді. Для відтворення відео в розмітці XAML набагато краще підходить тип **MediaPlayer**.

Будучи менш затратними, похідні від **Drawing** типи не мають вбудованої можливості обробки подій, бо вони не є **UIElement** або **FrameworkElement** (хоча допускають програмну реалізацію логіки перевірки попадання).

Інша ключова відмінність між типами, похідними від **Drawing**, і типами, похідними від **Shape**, полягає в тому, що похідні від **Drawing** типи не уміють візуалізувати себе, бо не успадковані від **UIElement**. Для відображення вмісту похідні типи повинні поміщатися в якийсь контейнерний об'єкт (зокрема **DrawingImage**, **DrawingBrush** або **DrawingVisual**).

Клас **DrawingImage** дозволяє поміщати малюнки і геометричні об'єкти всередину елемента управління **Image** з WPF, який зазвичай використовується для відображення даних із зовнішнього файла. Клас **DrawingBrush** дає можливість будувати кисть на основі малюнків і геометричних об'єктів, яка призначена для встановлення властивості, яку вимагає кисть. Нарешті, клас **DrawingVisual** використовується тільки на "*візуальному*" рівні графічної візуалізації, повністю керованій з коду C#.

Хоча працювати з малюнками трохи складніше, ніж з простими фігурами, відділення графічної композиції від графічної візуалізації робить типи, похідні

від **Drawing**, набагато менш витратними щодо пам'яті, ніж похідні від **Shape** типи, одночасно зберігаючи їх ключові служби.

> **На замітку!** Спочатку важко розібратися, в чому різниця між класом **DrawingImage** і згадуваним раніше класом **ImageDrawing**. Обидва вони цікаві тим, що дозволяють змішувати векторну і растрову графіку.
> **DrawingImage** – підклас **ImageSource** і в якості вмісту допускає типовий векторний малюнок **Drawing**, а не растрове зображення. Навпаки, **ImageDrawing** – підклас **Drawing**, і його вмістом може бути растровий об'єкт **ImageSource**, а не векторний.
> Зрозуміти різницю дозволяє нехитрий прийом: майже для всіх графічних класів у WPF (двовимірних і тривимірних) складене ім'я виду ***FooBar*** означає, що цей клас *є* підкласом **Bar**, який може *містити* або *працювати* як ***Foo***. Отже, **DrawingImage** – це **ImageSource**, який містить **Drawing**, а **ImageDrawing** – це **Drawing**, який містить **ImageSource**.

Побудова кисті DrawingBrush з використанням геометричних об'єктів

Раніше в ***Лекція 07.Служби візуалізації графіки WPF*** елемент **Path** заповнювався групою геометричних об'єктів приблизно так:

```
<Path Fill = "Orange" Stroke="Blue" StrokeThickness = "3">
  <Path.Data>
    <GeometryGroup>
      <EllipseGeometry Center ="75,70" RadiusX = "30" RadiusY = "30" />
      <RectangleGeometry Rect ="25,55 100 30" />
      <LineGeometry StartPoint="0,0" EndPoint="70,30" />
      <LineGeometry StartPoint="70,30" EndPoint="0,30" />
    </GeometryGroup>
  </Path.Data>
</Path>
```

Вчиняючи так само, ми досягаємо інтерактивності **Path** при надзвичайній легковагості[17], властивій геометричним об'єктам. Проте якщо потрібно візуалізувати аналогічний вивід і відсутня потреба у будь-якій (готовій) інтерактивності, тоді той же самий елемент <**GeometryGroup**> можна помістити всередину **DrawingBrush**:

```
<DrawingBrush>
  <DrawingBrush.Drawing>
    <GeometryDrawing>
      <GeometryDrawing.Geometry>
        <GeometryGroup>
          <EllipseGeometry Center ="75,70" RadiusX = "30" RadiusY = "30" />
          <RectangleGeometry Rect ="25,55 100 30" />
          <LineGeometry StartPoint="0,0" EndPoint="70,30" />
          <LineGeometry StartPoint="70,30" EndPoint="0,30" />
```

[17] Під **легковаговістю** будемо розуміти менші накладні витрати, наприклад, щодо пам'яті.

```
        </GeometryGroup>
      </GeometryDrawing.Geometry>
      <!-- Спеціальне перо для малювання меж -->
      <GeometryDrawing.Pen>
        <Pen Brush="Blue" Thickness="3"/>
      </GeometryDrawing.Pen>
      <!-- Спеціальна кисть для заповнення внутрішньої області -->
      <GeometryDrawing.Brush>
        <SolidColorBrush Color="Orange"/>
      </GeometryDrawing.Brush>
    </GeometryDrawing>
  </DrawingBrush.Drawing>
</DrawmgBrush>
```

При поміщенні групи геометричних об'єктів всередину **DrawingBrush** також знадобиться встановити об'єкт **Pen**, використовуваний для малювання меж, бо властивість **Stroke** більше не спадкує від базового класу **Shape**. Тут був створений елемент **Pen** з тими ж налаштуваннями, які використовувалися в значеннях **Stroke** і **StrokeThickness** з попереднього прикладу **Path**.

Крім того, через те, що властивість **Fill** більше не спадкує від класу **Shape**, то потрібно застосовувати синтаксис "*елемент-властивість*" для визначення об'єкта кисті, призначеного елементу **DrawingGeometry**, з суцільним помаранчевим кольором, як в попередніх налаштуваннях **Path**.

Малювання за допомогою DrawingBrush

Тепер об'єкт **DrawingBrush** можна використати для встановлення значення будь-якої властивості, яка вимагає об'єкта кисті. Наприклад, після підготовки наступної розмітки у редакторі Kaxaml за допомогою синтаксису "*елемент-властивість*" можна малювати зображення по всій поверхні **Page**:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Background>
    <DrawingBrush>
      <!-- Той же самий об'єкт DrawingBrush, що і раніше -->
    </DrawingBrush>
  </Page.Background>
</Page>
```

Або ж елемент **DrawingBrush** можна застосовувати для встановлення іншої сумісної з кистю властивості, такої як властивість **Background** елемента **Button**:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<Button Height="100" Width="100">
  <Button.Background>
    <DrawingBrush>
```

```
        <!-- Той же самий об'єкт DrawingBrush, що і раніше -->
      </DrawingBrush>
    </Button.Background>
</Button>
</Page>
```

Незалежно від того, яка, сумісна з кистю властивість встановлюється з використанням спеціального об'єкта **DrawingBrush**, візуалізація двовимірного зображення отримується з набагато меншими накладними витратами, чим у випадку візуалізації того ж зображення за допомогою фігур.

Включення типів Drawing в DrawingImage

Тип **DrawingImage** дозволяє підключати мальований геометричний об'єкт до елемента управління **Image** з WPF. Погляньте на наступну розмітку:

```
<Image>
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <!-- Той же самий об'єкт DrawingBrush, що і раніше -->
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

Тут елемент **GeometryDrawing** був поміщений всередину елемента **DrawingImage**, а не **DrawingBrush**. Із застосуванням елемента **DrawingImage** можна встановити властивість **Source** елемента управління **Image**.

Робота з векторними зображеннями

Ви напевно погодитеся з тим, що художнику досить не просто створювати складне векторне зображення з використанням інструментів і прийомів, які надаються середовищем Visual Studio. У розпорядженні художників є власні набори інструментів, які дозволяють здійснювати чудову векторну графіку. Образотворчих можливостей подібного роду не має ні IDE-середовище Visual Studio, ні супроводжуючий її інструмент Microsoft Blend. Перед тим, як векторні зображення можна буде імпортувати в застосунок WPF, вони мають бути перетворені у вирази шляхів. Після цього можна програмувати із використанням згенерованої об'єктної моделі, використовуючи Visual Studio.

> **На замітку!** Зображення знака небезпеки лазерного випромінювання взяте за адресою https://ru.wikipedia.org/wiki/Символы_опасности.

Перетворення файла з векторною графікою у файл XAML

Перш ніж можна буде імпортувати складні графічні дані (такі як векторна графіка) в застосунок WPF, графіку треба перетворити в *дані шляхів*. Робиться це так: візьмемо приклад файла зображення **.svg** із згаданим вище знаком небезпеки лазерного випромінювання. Потім завантажимо і встановимо

інструмент з відкритим кодом під назвою **Inkscape** (з веб-сайта www.inkscape.org). За допомогою **Inkscape** відкриємо файл **LaserSign.svg**. Можна здійснити модернізацію формату використовуючи головне меню: **Файл|Відкрити недавній**. Встановимо налаштування, як показано на рис. 8.1.

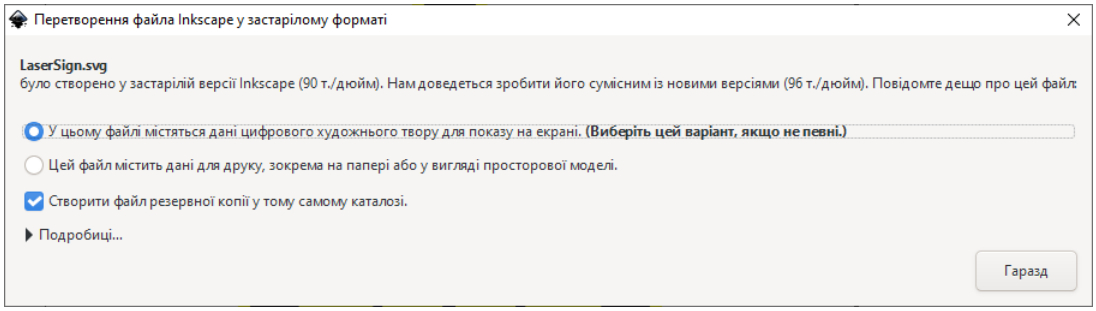


Рис. 8.1. Модернізація файла **SVG** до самого останнього формату в **Inkscape**

Наступні кроки спочатку здадуться дещо дивними, але насправді вони є простим способом перетворення векторних зображень в розмітку XAML. Коли зображення набуло бажаного вигляду, треба вибрати пункт меню **File|Print**. Далі потрібно буде вказати **Microsoft XPS Document Writer** як цільовий принтер і клацнути на кнопці **Print** (Друк). У вікні, яке відкрилося, треба ввести ім'я файла і вибрати місце, де він має бути збережений, після чого клацнути на кнопці **Save** (Зберегти). У результаті отримується файл ***.xps** (або ***.oxps**).

Формати ***.xps** і ***.oxps** насправді є *архівами* ZIP. Перейменувавши розширення в **.zip**, файл можна відкрити у провіднику файлів (або в утиліті архівації). Файл містить ієрархію тек у **Провіднику**, приведену на рис. 8.2.

Потрібний файл знаходиться в теці **Pages(Documents/1/Pages)** і називається **1.fpage**. Відкриємо його в текстовому редакторі і скопіюємо у буфер всі дані окрім відкриваючого і закриваючого дескрипторів **FixedPage**. Дані шляхів потім можна помістити всередину елемента **Canvas** головного вікна у Kaxaml. У результаті зображення буде показане у вікні XAML.

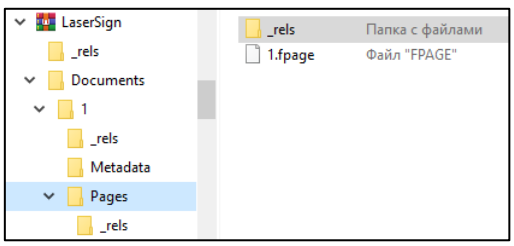


Рис. 8.2. Ієрархія тек у файлі ***.xps** або ***.oxps**

Імпортування графічних даних в проект WPF

Створимо новий проект застосування WPF на ім'я **InteractiveLaserSign**. Змінимо значення властивостей **Height** і **Width** елемента **Window** відповідно на **625** і **675** і замінимо елемент **Grid** елементом **Canvas**:

```
<Window x:Class="InteractiveLaserSign.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:InteractiveLaserSign"
        mc:Ignorable="d"
        Title="MainWindow" Height="625" Width="675">
  <Canvas>

  </Canvas>
</Window>
```

Скопіюємо повну розмітку XAML з файла **1.fpage** (виключаючи зовнішній дескриптор **FixedPage**) і вставимо її в елемент управління **Canvas** всередині **MainWindow**. Переглянувши вікно в режимі проектування, легко впевнитися в тому, що знак небезпеки лазерного випромінювання успішно відтворюється в застосунку.

Заглянувши у вікно **Document Outline**, можна побачити, що кожна частина зображення представлена як XAML-елемент **Path**. Якщо змінити розміри елемента **Window**, то якість зображення залишиться такою ж безвідносно до того, наскільки великим зроблено вікно. Причина в тому, що зображення, представлені з допомогою елементів **Path**, візуалізуються із використанням механізму малювання і математики, а не за рахунок маніпулювання пікселями.

Взаємодія із зображенням

Згадайте, що маршрутизована подія поширюється тунельним і бульбашковим способом (див. ***Лекція 05. Події у WPF***), тому клацання на будь-якому елементі **Path** всередині **Canvas** може бути оброблене обробником подій клацання на **Canvas**. Модифікуємо розмітку **Canvas** так:

```
<Canvas MouseLeftButtonDown="Canvas_MouseLeftButtonDown">
```

Додамо обробник подій з таким кодом:

```
private void Canvas_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
  if (e.OriginalSource is Path p)
  {
    p.Fill = new SolidColorBrush(Colors.Red);
  }
}
```

Запустимо застосування і клацнемо на лініях, щоб побачити ефекти.

Тепер ви розумієте процес генерації даних шляхів для складної графіки і знаєте, як взаємодіяти з графічними даними в коді. Ви напевно погодитеся, що наявність у професійних художників можливості генерувати складні графічні дані та експортувати їх у вигляді розмітки XAML виключно важлива. Після

того, як графічні дані збережені у файлі XAML, розробники можуть імпортувати розмітку і писати код для взаємодії з об'єктною моделлю.

Візуалізація графічних даних з використанням візуального рівня

Останній варіант візуалізації графічних даних за допомогою WPF називається *візуальним рівнем*. Раніше вже згадувалося, що доступ до нього можливий тільки з коду (він не дружній по відношенню до розмітки XAML). Незважаючи на те що переважна більшість застосунків WPF добре працюватимуть із використанням фігур, малюнків і геометричних об'єктів, візуальний рівень забезпечує *найшвидший* спосіб візуалізації великих обсягів графічних даних. Візуальний рівень також може бути корисний, коли потрібно візуалізувати єдине зображення у великій області. Наприклад, якщо вимагається заповнити фон вікна простим статичним зображенням, тоді візуальний рівень буде найшвидшим способом рішення такої задачі. Крім того, він зручний, коли треба дуже швидко міняти фон вікна залежно від вводу користувача або чогось іншого.

Давайте побудуємо невелику програму, яка ілюструє основи використання візуального рівня.

Базовий клас Visual і похідні дочірні класи

Абстрактний клас **System.Windows.Media.Visual** пропонує мінімальний набір служб (візуалізацію, перевірку попадання, трансформації) для візуалізації графіки, але не надає підтримку додаткових невізуальних служб, які можуть стати причиною розбухання коду (події вводу, служби компонування, стилі і прив'язка даних). Клас **Visual** є абстрактним базовим класом. Для виконання справжніх операцій візуалізації повинен застосовуватися один з його похідних класів. У WPF визначені декілька підкласів **Visual**, у тому числі **DrawingVisual**, **Viewport3DVisual** і **ContainerVisual**. У розглядуваному далі прикладі, ми зосередимося тільки на **DrawingVisual**-легковагому класі малювання, який використовується для візуалізації фігур, зображень або тексту.

Перший погляд на клас DrawingVisual

Щоб візуалізувати дані на поверхні із застосуванням класу **DrawingVisual**, знадобиться виконати наступні основні кроки:

- *отримати* об'єкт **DrawingContext** з **DrawingVisual**;
- *використати* об'єкт **DrawingContext** для візуалізації графічних даних.

Ці два кроки представляють абсолютний мінімум, необхідний для візуалізації якихось даних на поверхні. Проте, коли треба, щоб графічні дані, які візуалізуються реагували на обчислення при перевірці попадання (що важливо для додавання взаємодії з користувачем), потрібно буде також виконати *додаткові* кроки:

- *оновити* логічне і візуальне дерева, підтримувані контейнером, на якому здійснюється візуалізація;

- *перевизначити* два віртуальні методи з класу **FrameworkElement**, дозволивши контейнеру отримувати створені візуальні дані.

Давайте дослідимо останні два кроки детальніше. Щоб продемонструвати застосування класу **DrawingVisual** для візуалізації двовимірних даних, створимо у Visual Studio новий проект застосунку WPF на ім'я **RenderingWithVisuals**. Нашою першою метою буде використання класу **DrawingVisual** для динамічного присвоєння даних елементу управління **Image** з WPF. Розпочнемо з наступного оновлення розмітки XAML вікна для обробки події **Loaded**:

```
<Window x:Class="RenderingWithVisuals.MainWindow"
        ...
        Title="Дослідження візуального прошарку " Height="350" Width="525"
            Loaded="MainWmdow_Loaded">
  <Grid>

  </Grid>
</Window>
```

Замінимо елемент **Grid** панеллю **StackPanel** і додамо в неї елемент **Image**:

```
<StackPanel Background="AliceBlue" Name="myStackPanel">
  <Image Name="mylmage" Height="80"/>
</StackPanel>
```

Елемент управління **Image** поки не має значення у властивості **Source**, бо вона буде встановлюватися під час виконання. З подією **Loaded** пов'язана робота з побудови графічних даних в пам'яті із застосуванням об'єкта **DrawingBrush**. Ось реалізація обробника події **Loaded**:

```
private void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
  const int TextFontSize = 30;
  // Створити об'єкт System.Windows.Media.FormattedText.
  FormattedText text = new FormattedText("Hello Visual Layer!",
      new System.Globalization.CultureInfo("en-us"),
      FlowDirection.LeftToRight,
      new Typeface(this.FontFamily, FontStyles.Italic, FontWeights.DemiBold,
      FontStretches.UltraExpanded),
      TextFontSize,
      Brushes.Green,
      null,
      VisualTreeHelper.GetDpi(this).PixelsPerDip);

  // Створити об'єкт DrawingVisual та отримати об'єкт DrawingContext.
  DrawingVisual drawingVisual = new DrawingVisual();
  using (DrawingContext drawingContext = drawingVisual.RenderOpen())
  {
    // Викликати будь-який з методів DrawingContext для візуалізації даних.
      drawingContext.DrawRoundedRectangle(Brushes.Yellow,
```

```
            new Pen(Brushes.Black, 5),
            new Rect(5, 5, 450, 100), 20, 20);
        drawingContext.DrawText(text, new Point(20, 20));
    }
    // Динамічно створити бітове зображення,
    // використовуючи дані в об'єкті DrawingVisual.
    RenderTargetBitmap bmp = new RenderTargetBitmap(500, 100, 100, 90,
            PixelFormats.Pbgra32);
    bmp.Render(drawingVisual);
    // Встановити джерело для елементу управління Image.
    myImage.Source = bmp;
}
```

У коді задіяні декілька нових класів WPF, які будуть коротко описані нижче (детальніші відомості можна отримати в документації по .NET Framework за адресою https://docs.microsoft.com/ru-ru/dotnet/framework/). Метод розпочинається зі створення нового об'єкта **FormattedText**, який представляє текстову частину конструйованого зображення в пам'яті. Як бачите, конструктор дозволяє задавати численні атрибути, у тому числі розмір шрифту, сімейство шрифтів, колір переднього плану і сам текст.

Потім через виклик методу **RenderOpen()** на екземплярі **DrawingVisual** отримується потрібний об'єкт **DrawingContext**. Тут у **DrawingVisual** візуалізується кольоровий прямокутник із заокругленими кутами, за яким йде форматований текст. В обох випадках графічні дані поміщаються в **DrawingVisual** із застосуванням жорстко закодованих значень, що не надто добре у бізнесовому застосунку, але цілком підходить для нашого простого тесту.

> **На замітку!** Обов'язково перегляньте опис класу **DrawingContext** в документації .NET Framework, щоб ознайомитися з усіма членами, пов'язаними з візуалізацією. Якщо у минулому ви працювали з об'єктом **Graphics** з Windows Forms, то **DrawingContext** повинен бути схожим.

Декілька останніх операторів відображають **DrawingVisual** на об'єкт **RenderTargetBitmap**, який є членом простору імен **System.Windows.Media.Imaging**. Цей клас приймає візуальний об'єкт і трансформує його в растрове зображення, що знаходиться в пам'яті. Потім встановлюється властивість **Source** елемента управління **Image** та отримується вивід, показаний на рис. 8.3.

> **На замітку!** Простір імен **System.Windows.Media.Imaging** містить додаткові класи кодування, які дозволяють зберігати об'єкт, що знаходиться в пам'яті **RenderTargetBitmap** у фізичний файл з різноманітними форматами. Деталі шукайте в описі **JpegBitmapEncoder** і пов'язаних з ним класів.

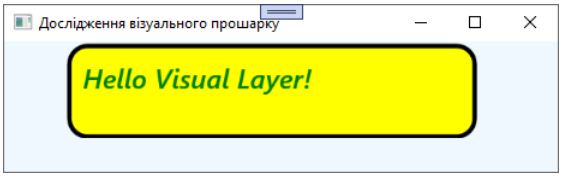


Рис. 8.3. Використання візуального рівня для візуалізації растрового зображення, розміщеного в пам'яті

Візуалізація графічних даних у спеціальному диспетчері компонування

Хоча використання **DrawingVisual** для малювання на тлі елемента управління WPF викликає цікавість, можливо частіше доведеться будувати спеціальний диспетчер компонування (**Grid**, **StackPanel**, **Canvas** і т. д.), який внутрішньо використовує візуальний рівень для візуалізації свого вмісту. Після створення такого спеціального диспетчера компонування його можна підключити до звичайного елемента **Window** (а також **Page** або **UserControl**) і дозволити частині інтерфейсу користувача використати високо оптимізований компонент візуалізації, тоді як для візуалізації некритичних графічних даних застосовуватимуться фігури і малюнки.

Якщо додаткова функціональність, пропонована спеціалізованим диспетчером компонування, не потрібна, то можна просто розширити клас **FrameworkElement**, який має необхідну інфраструктуру, що дозволяє містити також і візуальні елементи. У цілях ілюстрації вставимо в проект новий клас на ім'я **CustomVisualFrameworkElement** (рис. 8.4). Успадкуємо його від **FrameworkElement** та імпортуємо простори імен **System.Windows**, **Systern.Windows.Input** і **System.Windows.Media**.

Клас **CustomVisualFrameworkElement** підтримуватиме змінну-член типу **VisualCollection**, яка містить два фіксовані об'єкти **DrawingVisual** (звичайно, в цю колекцію можна було б додавати члени за допомогою миші, але ми вирішили зберегти приклад простим). Модифікуємо код класу так:

```
class CustomVisualFrameworkElement : FrameworkElement
{
  // Колекція всіх візуальних об'єктів.
  VisualCollection theVisuals;
  public CustomVisualFrameworkElement()
  {
    // Заповнити колекцію VisualCollection декількома об'єктами DrawingVisual.
    // Аргумент конструктора представляє власника візуальних об'єктів.
    theVisuals = new VisualCollection(this) { AddRect(), AddCircle() };
  }
  private Visual AddCircle()
  {
    DrawingVisual drawingVisual = new DrawingVisual();
    // Отримати об'єкт DrawingContext для створення нового вмісту.
    using (DrawingContext drawingContext = drawingVisual.RenderOpen())
```

```
        {
            // Створити коло і намалювати його в DrawingContext.
            Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
            drawingContext.DrawEllipse(Brushes.DarkBlue, null,
                new Point(70, 90), 40, 50);
        }
        return drawingVisual;
    }
    private Visual AddRect()
    {
        DrawingVisual drawingVisual = new DrawingVisual();
        using (DrawingContext drawingContext = drawingVisual.RenderOpen())
        {
            Rect rect = new Rect(new Point(160, 100), new Size(320, 80));
            drawingContext.DrawRectangle(Brushes.Tomato, null, rect);
        }
        return drawingVisual;
    }
}
```

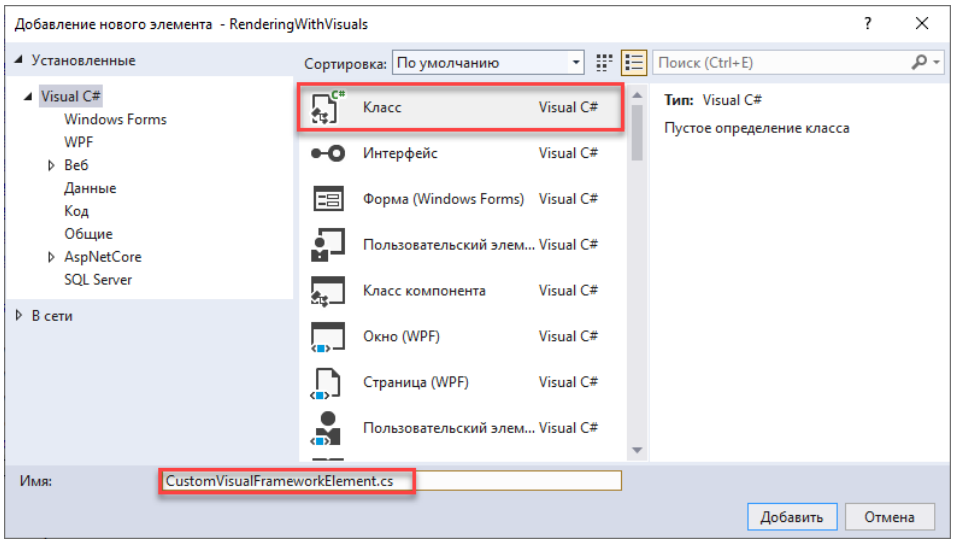


Рис. 8.4. Додавання класу до проекту

Перш ніж спеціальний елемент **FrameworkElement** можна буде використати всередині **Window**, потрібно буде перевизначити два згаданих раніше ключових віртуальних члени, які викликаються внутрішньо інфраструктурою WPF під час процесу візуалізації. Метод **GetVisualChild()** повертає з колекції дочірніх елементів дочірній елемент за заданим індексом. Властивість **VisualChildrenCount**, яка дозволяє тільки читання, повертає число візуальних дочірніх елементів всередині візуальної колекції. Обидва члени легко реалізувати, бо всю реальну роботу можна делегувати змінній-члену типу **VisualCollection**:

```
protected override int VisualChildrenCount => theVisuals.Count;
protected override Visual GetVisualChild(int index)
{
   // Значення має бути більше нуля, тому розумно це перевірити,
   if (index < 0 || index >= theVisuals.Count)
   {
      throw new ArgumentOutOfRangeException();
   }
   return theVisuals[index];
}
```

Тепер ми маємо в розпорядженні достатню функціональність, щоб протестувати наш спеціальний клас. Модифікуємо опис XAML елемента **Window**, додавши в існуючий контейнер **StackPanel** один об'єкт **CustomVisualFrameworkElement**. Це буде вимагати створення спеціального простору імен XML, який відображається на простір імен .NET.

```
<Window x:Class="RenderingWithVisuals.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:RenderingWithVisuals"
        mc:Ignorable="d"
        Title="Дослідження візуального прошарку" Height="350" Width="525"
            Loaded="MainWmdow_Loaded">
  <StackPanel Background="AliceBlue" Name="myStackPanel">
    <Image Name="myImage" Height="80"/>
    <local:CustomVisualFrameworkElement/>
  </StackPanel>
</Window>
```

Результат виконання програми показаний на рис. 8.5.

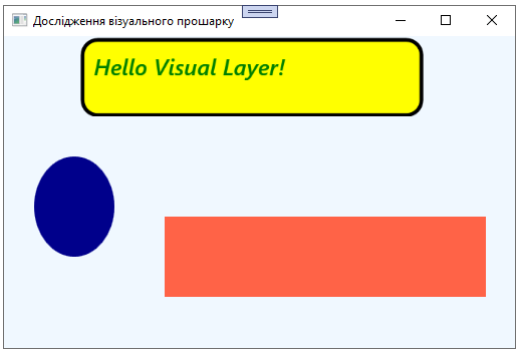


Рис. 8.5. Використання візуального рівня для візуалізації даних у спеціальному елементі **FrameworkElement**

Перевірка попадання у Visual

Під *перевіркою попадання* (**hit testing**) розуміють відповідь на питання, чи знаходиться деяка точка (або множина точок) всередині заданого об'єкта. Зазвичай це питання виникає при обробці подій миші, стилуса або торкання, коли треба довідатися, де знаходиться покажчик миші, кінчик стилуса або палець.

У WPF є два види перевірки попадання: перевірка попадання у **Visual**, яка підтримується всіма класами, похідними від **Visual**, і перевірка попадання при вводі, підтримувана тільки підкласами **UIElement**. Тут ми опишемо лише перевірку попадання у **Visual**, а перевірку попадання при вводі розглянуто в ***Лекція 07.Служби візуалізації графіки WPF*** де розглядається клас **Shape**.

Без механізму перевірки попадання об'єкти **Visual** не мали б змоги реагувати на такі дії користувача, як клацання мишею, торкання або наведення покажчика миші, тому що в цьому класі немає подій вводу, які має в розпорядженні клас **UIElement** (**MouseLeftButtonDown**, **MouseEnter**, **MouseLeave**, **MouseMove** та ін.). Обробивши таку подію в елементі-власнику **UIElement** і скориставшись потім перевіркою попадання у **Visual**, ви зможете довідатися, чи «*зачеплені*» якісь дочірні об'єкти **Visual**, які цікавлять вас, і відповідно відреагувати на подію.

Реагування на операції перевірки попадання

Через те, що клас **DrawingVisual** не має в розпорядженні інфраструктури **UIElement** або **FrameworkElement**, потрібно програмно додати можливість реагування на операції перевірки попадання. Завдяки концепції *логічного* і *візуального* дерев на візуальному рівні робити це дуже просто. Виявляється, що в результаті написання блоку XAML по суті будується логічне дерево елементів. Проте з кожним логічним деревом пов'язаний набагато розвиненіший опис, відомий як візуальне дерево, яке містить низькорівневі інструкції візуалізації.

Згадані дерева детально розглядаються в ***Лекція 12. Логічні і візуальні дерева та шаблони***, а зараз досить знати, що доти, поки спеціальні візуальні об'єкти не будуть зареєстровані в таких структурах даних, виконувати операції перевірки попадання неможливо. На щастя, контейнер **VisualCollection** забезпечує реєстрацію автоматично (ось чому в аргументі конструктора потрібно передавати посилання на спеціальний елемент **FrameworkElement**).

Змінимо код класу **CustomVisualFrameworkElement** для обробки події **MouseDown** в конструкторі класу із застосуванням стандартного синтаксису C#:

```
this.MouseDown += CustomVisualFrameworkElement_MouseDown;
```

Реалізація цього обробника викликатиме метод **VisualTreeHelper.HitTest()** з метою з'ясування, чи знаходиться курсор миші всередині меж одного з візуальних об'єктів. Для цього в одному з параметрів методу **HitTest()** вказується делегат **HitTestResultCallback**, який виконуватиме обчислення. Додамо в клас **CustomVisualFrameworkElement** наступні методи:

```
void CustomVisualFrameworkElement_MouseDown(object sender, MouseButtonEventArgs e)
{
  // З'ясувати, де користувач виконав клацання.
```

```
    Point pt = e.GetPosition((UIElement)sender);
    // Викликати допоміжну функцію делегат, щоб
    // подивитися, чи було здійснено клацання на візуальному об'єкті.
    VisualTreeHelper.HitTest(this, null,
        new HitTestResultCallback(myCallback), new PointHitTestParameters(pt));
}
public HitTestResultBehavior myCallback(HitTestResult result)
{
    // Якщо клацання було здійснене на візуальному об'єкті, то
    // перемкнутися між скошеною і нормальною візуалізацією.
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
        if (((DrawingVisual)result.VisualHit).Transform == null)
        {
            ((DrawingVisual)result.VisualHit).Transform = new SkewTransform(7, 7);
        }
        else
            ((DrawingVisual)result.VisualHit).Transform = null;
    }
    // Повідомити метод HitTest() про припинення поглиблення у візуальне дерево.
    return HitTestResultBehavior.Stop;
}
```

Запустимо програму знову. Тепер клацнувши на будь-якому з візуальних зображень об'єктів спостерігаємо виконання трансформації (див. рис. 8.6). Розглядуваний нами приклад взаємодії з візуальним рівнем WPF дуже простий, бо тут можна використати ті ж самі кисті, трансформації, пера і диспетчери компонування, які зазвичай застосовуються в розмітці XAML. Отже, ви вже знаєте досить багато про роботу з класами, похідними від **Visual**.

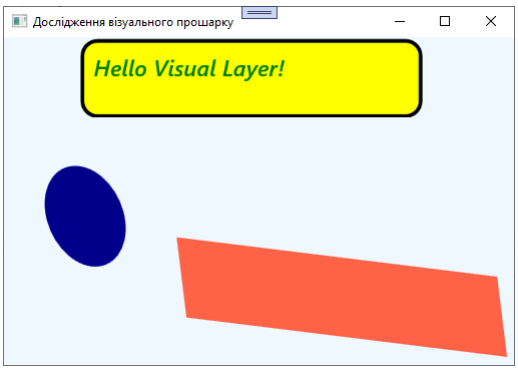


Рис. 8.6. Демонстрація візуалізації деяких трансформацій

На цьому дослідження служб графічної візуалізації WPF завершене. Незважаючи на розкриття ряду цікавих тем, насправді ми лише злегка торкнулися великої області графічних можливостей інфраструктури WPF. Подальше вивчення фігур, малюнків, кистей, трансформацій і візуальних об'єктів ви можете продовжити самостійно.

Резюме

Згадайте, що коли потрібно будувати інтерактивну двовимірну візуалізацію, то фігури роблять такий процес дуже простим. З іншого боку, статичні, не інтерактивні зображення можуть візуалізуватися в оптимальнішій спосіб з використанням малюнків і геометричних об'єктів, а візуальний рівень (доступний тільки в коді) забезпечить максимальний контроль і продуктивність.

Контрольні питання

1. Які переваги надає розвинений API-інтерфейс WPF з використанням класу System.Windows.Media.Drawing зі збірки PresentationCore.dll порівняно з використанням типів з класу Shape?
2. Які класи, що розширюють Drawing пропонує інфраструктура WPF та для чого вони використовуються?
3. Які відмінності між типами, похідними від Drawing, і типами, похідними від Shape?
4. У чому полягає різниця між класами DrawingImage та ImageDrawing?
5. У чому суть варіанту візуалізації графічних даних за допомогою WPF? Як називають цей рівень?
6. Коли переважно використовують, так званий, візуальний рівень?
7. Як здійснюється візуалізація графічних даних з використанням спеціального диспетчера компонування?
8. Що розуміється під поняттям перевірки попадання?

Лекція 9. Ресурси WPF

У цій лекції ми розглянемо три важливих (і взаємозв'язаних) теми, які дозволять поглибити розуміння API-інтерфейсу WPF. Насамперед, ми вивчимо роль *логічних ресурсів*. Ви побачите, що система логічних ресурсів (також званих *об'єктними ресурсами*), які представляють собою спосіб посилання на часто використовувані об'єкти всередині застосунку WPF. Хоча логічні ресурси нерідко реалізуються в розмітці XAML, вони можуть бути визначені і в процедурному коді.

Далі ви з'ясуєте, як визначати, виконувати та управляти анімаційною послідовністю. Використання анімації WPF не обмежується відеоіграми або мультимедіа-застосунками. В API-інтерфейсі WPF анімація може використовуватися, наприклад, для підсвічування кнопки, коли вона отримує фокус, або збільшення розміру вибраного рядка і т. п. Розуміння анімації є ключовим аспектом побудови спеціальних шаблонів елементів управління (див. ***Лекція 10. Реалізаця анімації WPF***).

Потім ми з'ясуємо роль стилів і шаблонів WPF у ***Лекція 11. Реалізація стилів WPF*** та ***Лекція 12. Логічні і візуальні дерева та шаблони***. Подібно до веб-сторінки, в якій застосовуються стилі CSS або механізм тем ASP.NET, використання WPF може визначати загальний вигляд і поведінку для набору елементів управління. Такі стилі можна визначати в розмітці і зберігати їх у вигляді об'єктних ресурсів для подальшого використання, а також динамічно застосовувати під час виконання.

Система ресурсів WPF

Першим завданням буде дослідження теми *вбудовування* і *доступу* до ресурсів застосунку. Інфраструктура WPF підтримує два види ресурсів:

1. Перший – *двійкові ресурси*; ця категорія зазвичай включає елементи, які більшість програмістів вважають ресурсами в традиційному сенсі (вбудовані файли зображень або звукових кліпів, значки, використовувані застосунками, і т. д.).
2. Друга – *об'єктні* або *логічні ресурси*, представляють іменовані об'єкти .NET, які можна пакувати і багаторазово використовувати всюди в застосунку. Незважаючи на те що запаковувати у вигляді об'єктного ресурсу дозволено будь-який об'єкт .NET, логічні ресурси особливо зручні при роботі з графічними даними довільного роду, бо можна визначити часто використовувані графічні примітиви (кисті, пера, анімації і т. д.) і посилатися на них у міру потреби.

Робота з двійковими ресурсами

Перш ніж перейти до теми об'єктних ресурсів, коротко проаналізуємо, як запаковувати *двійкові ресурси* на зразок значків і файлів зображень (наприклад, логотипів компанії або зображень для анімації) всередині застосунків. Створимо у Visual Studio новий проект застосунку WPF на ім'я **BinaryResourcesApp**. Модифікуємо розмітку початкового вікна для обробки події **Loaded** елемента **Window** і використаємо **DockPanel** як корінь компонування:

```
<Window x:Class="BinaryResourcesApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Робота з бінарними ресурсами" Height="500" Width="649"
            Loaded="MainWindow_OnLoaded">
  <DockPanel LastChildFill="True">

  </DockPanel>
</Window>
```

Припустимо, що застосунок повинен відображати всередині частини вікна один з трьох файлів зображень, ґрунтуючись на використанні вводу користувача. Елемент управління **Image** з WPF може використовуватися для відображення не лише типового файла зображення (***.bmp**, ***.gif**, ***.ico**, ***.jpg**, ***.png**, ***.wdp** або ***.tiff**), але також даних об'єкта **DrawingImage** (як було показано в ***Лекція 07.Служби візуалізації графіки WPF*** та ***Лекція 08. Візуалізація графічних даних***). Побудуємо інтерфейс користувача вікна, який підтримує диспетчер компонування **DockPanel**, що містить просту панель інструментів з кнопками **Next** (Вперед) і **Previous** (Назад). Під панеллю інструментів розташований елемент управління **Image**, властивість **Source** якого в поточний момент не встановлена:

```
<DockPanel LastChildFill="True">
  <ToolBar Height="60" Name="picturePickerToolbar" DockPanel.Dock="Top">
    <Button x:Name="btnPreviousImage" Height="40" Width="100" BorderBrush="Black"
      Margin="5" Content="Previous" Click="btnPreviousImage_Click"/>
    <Button x:Name="btnNextImage" Height="40" Width="100" BorderBrush="Black"
      Margin="5" Content="Next" Click="btnNextImage_Click"/>
  </ToolBar>
  <!-- Цей елемент Image буде заповнюватися в коді -->
  <Border BorderThickness="2" BorderBrush="Green">
    <Image x:Name="imageHolder" Stretch="Fill" />
  </Border>
</DockPanel>
```

Додамо наступні порожні обробники подій:

```
private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
{

}
private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{

}
private void btnNextImage_Click(object sender, RoutedEventArgs e)
{

}
```

Коли вікно завантажується, зображення додадуться в колекцію, по якій здійснюватиметься прохід за допомогою кнопок **Next** і **Previous**. Тепер, маючи в розпорядженні інфраструктуру застосунку, займімося дослідженням різних варіантів її реалізації.

Включення в проект незв'язаних файлів ресурсів

Один з варіантів передбачає постачання файлів зображень у вигляді набору незв'язаних файлів в якомусь підкаталозі *всередині шляху встановлення* застосунку. Для цього:

1. Додамо в теку проекту (у нас – **BinaryResourcesApp**) нову теку на ім'я **Images**.
2. Додамо в теку **Images** декілька файлів із зображеннями, викачаних з інтернету. У мене це файли з іменами **Deer.jpg**, **Dogs.jpg**, **Dogs1.jpg**, і **Welcome.jpg**.
3. У вікні проекту **Solution Explorer** (Оглядач рішень) створимо таку ж теку **Images**.
4. Клацнувши правою кнопкою миші цю теку, виберемо у контекстному меню пункт **Add|Existing Item** (Додати|Існуючий елемент).
5. У діалоговому вікні **Add Existing Item** (Додавання існуючого елемента), яке відкрилося змінимо фільтр файлів на ***.*** (рис. 9.1), щоб стали видними файли зображень. Ви можете додавати власні файли зображень або викачати з інтернету (див. п. 2).
6. Після групового вибору зображень і натиснення кнопки **Додати** ми побачимо файли цих зображень у вікні **Solution Explorer** (Оглядач рішень) у теці **Images**.

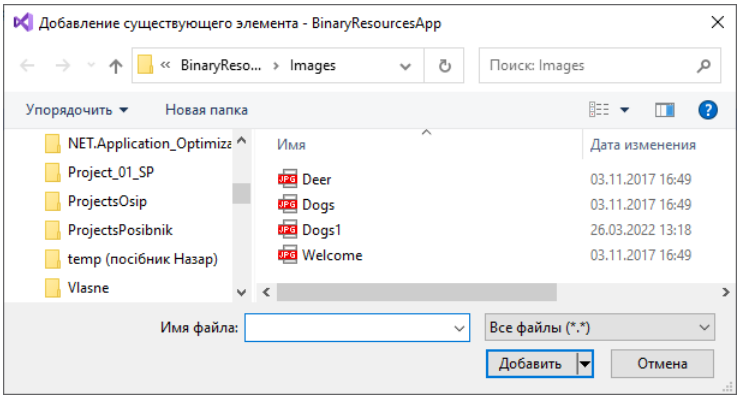


Рис. 9.1. Додавання існуючих зображень

Конфігурування незв'язаних ресурсів

Для копіювання вмісту теки **\Images** в теку **\bin\Debug** при побудові проекту виберемо всі зображення у вікні **Solution Explorer**, клацнемо правою кнопкою миші і виберемо в контекстному меню пункт **Properties** (Властивості), щоб відкрити вікно **Properties** (Властивості (рис. 9.2.)) Встановимо властивість **Build Action** (Дії при збиранні) в **Content** (Вміст), а властивість **Copy Output**

Directory (Копіювати у вихідний каталог) в **Copy always** (Копіювати завжди), як показано на рис. 9.2.

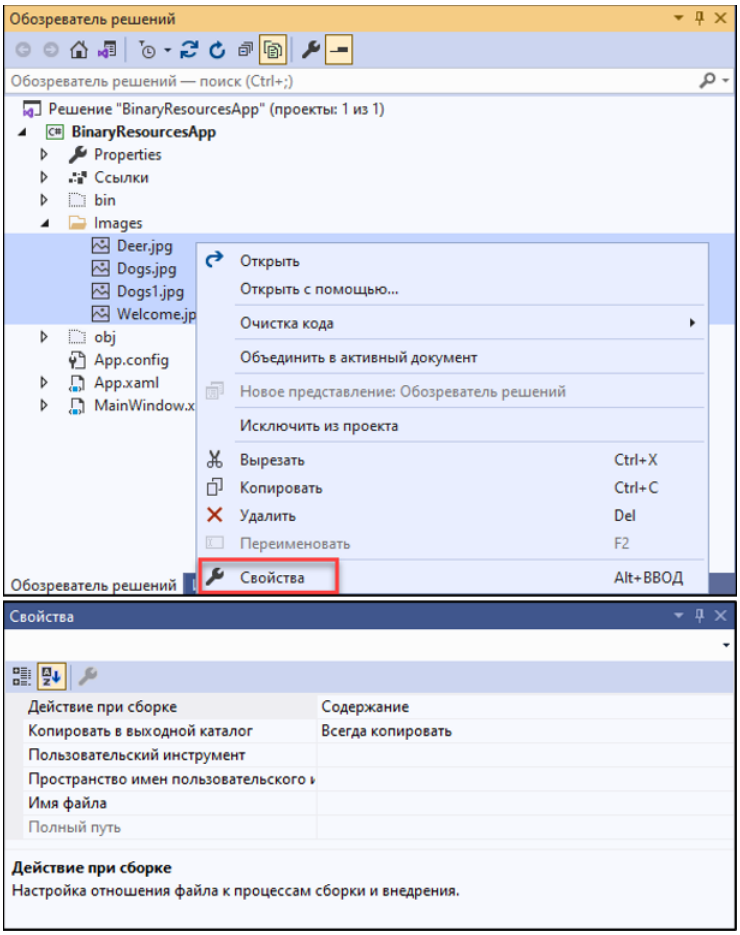


Рис. 9.2. Конфігурація цих зображень для копіювання у вихідний каталог

> **На замітку!** Для властивості **Copy Output Directory** (Копіювати у вихідний каталог) можна було б також вибрати варіант **Copy if Newer** (Копіювати, якщо новіше), що дозволить скоротити час копіювання при побудові великих проектів з великим обсягом вмісту. У цьому прикладі варіанту **Copy always** (Завжди копіювати) цілком достатньо.

Після побудови проекту можна клацнути на кнопці **Show all Files** (Показати всі файли) у вікні **Solution Explorer** і переглянути скопійовану папку **\Images** всередині **\bin\Debug** (може також знадобитися клацнути на кнопці **Refresh** (Оновити)).

Програмне завантаження зображення

Інфраструктура WPF надає клас на ім'я **BitmapImage**, визначений в просторі імен **System.Windows.Media.Imaging**. Він дозволяє завантажувати дані

з файла зображення, місце розташування якого представлене об'єктом **System.Uri**. Додамо поле типу **List<BitmapImage>** для зберігання всіх зображень, а також поле типу **int** для зберігання індексу зображення, показаного у нинішній момент.

В обробнику події **Loaded** вікна заповнимо список зображень і встановимо властивість **Source** елемента управління **Image** в перше зображення зі списка:

```
private void MainWindow_OnLoaded(object sender, RoutedEventArgs e)
{
  try
  {
    string path = Environment.CurrentDirectory;
    // Завантажте ці зображення з диска під час завантаження вікна.
    _images.Add(new BitmapImage(new Uri($@"{path}\Images\Deer.jpg")));
    _images.Add(new BitmapImage(new Uri($@"{path}\Images\Dogs.jpg")));
    _images.Add(new BitmapImage(new Uri($@"{path}\Images\Dogs1.jpg")));
    _images.Add(new BitmapImage(new Uri($@"{path}\Images\Welcome.jpg")));
    // Показати перше зображення у списку.
    imageHolder.Source = _images[_currImage];
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

<u>Пригадаємо!</u> Визначення дослівних рядків

За рахунок додавання до рядкового літерала *префікса* **@** можна створювати так звані *дослівні рядки*. Використовуючи дослівні рядки, ви відключаєте обробку управляючих послідовностей в літералах і примушуєте виводити значення **string** таким як він є. Така можливість найкорисніша при роботі з рядками, які представляють шляхи до каталогів і мережевих ресурсів. Отже, замість використання управляючої послідовності \\, можна зробити так:

```
// Наступний рядок відтворюється дослівно,
// так що відображаються всі управляючі символи.
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Також зверніть увагу, що дослівні рядки можуть використовуватися для зберігання символів пропуску в рядках, рознесених по декількох рядках виводу:

```
// При використанні дослівних рядків пробільні символи оберігаються,
string myLongString = @"This is a very
        very
                very
                        long string";
Console.WriteLine(myLongString);
```

Застосовуючи дослівні рядки, в літеральний рядок можна також безпосередньо вставляти символи подвійної лапки, просто дублюючи знак **"**:

```
Console.WriteLine(@"Cerebus said ""Darrr! Pret - ty sun - sets""");
```

Пригадаємо! Інтерполяція рядків

Синтаксис з фігурними дужками, (**{0}**, **{1}** і т. д.), існував всередині платформи .NET ще з часів версії 1.0. Починаючи з випуску C# **6**, при побудові рядкових літералів, які містять заповнювачі для змінних, програмісти C# можуть використати альтернативний синтаксис. Формально він називається *інтерполяцією рядків*. Незважаючи на те що вивід операції ідентичний виводу, який отримується за допомогою традиційного синтаксису форматування рядків, новий підхід дозволяє безпосередньо впроваджувати самі змінні, а не поміщати їх в список з роздільниками-комами. Поглянемо на показаний нижче додатковий метод в класі **Program** (**StringInterpolation()**), який будує змінну типу **string** із використанням обох підходів:

```
static void StringInterpolation()
{
  // Деякі локальні змінні будуть включені у великий рядок.
  int age = 4;
  string name = "Soren";
  // Використання синтаксису з фігурними дужками.
  string greeting = string.Format("Hello {0} you are {1} years old,", name, age);
  // Використання інтерполяції рядків.
  string greeting2 = $"Hello {name} you are {age} years old.";
}
```

У змінній **greeting2** зверніть увагу на те, що сконструйований рядок розпочинається з *префікса* **$**. Крім того, фігурні дужки як і раніше використовуються для позначення заповнювача під змінну; проте, замість використання числової мітки є можливість вказувати безпосередньо змінну. Передбачувана перевага полягає в тому, що новий синтаксис дещо легше читати в лінійній манері (зліва направо) з врахуванням того, що не вимагається «*перескакувати в кінець*» для перегляду списку значень, які підлягають вставлянню під час виконання.

З новим синтаксисом пов'язаний ще один цікавий аспект: фігурні дужки, використовувані в інтерполяції рядків, означають *допустиму зону видимості*. Отже, зі змінними можна використовувати операцію *крапка*, щоб змінювати їх стан. Модифікуємо код присвоєння змінних **greeting** і **greeting2**:

```
string greeting = string.Format("Hello {0} you are {1} years old.", name.ToUpper(), age);
string greeting2 = $"Hello {name.ToUpper()} you are {age} years old.";
```

Тут за допомогою виклику **ToUpper()** здійснюється перетворення **name** у верхній регістр. Зверніть увагу, що при підході з інтерполяцією рядків завершальна пара круглих дужок до виклику цього методу не додається.

Враховуючи це, використати зону видимості, визначувану фігурними дужками, як повноцінну зону видимості методу, яка містить численні рядки виконуваного коду, неможливо. Натомість допускається тільки виклик одиночного методу на об'єкті із використанням операції крапка, а також визначати простий загальний вираз на зразок **{age += 1}**.

Корисно також відмітити, що у рамках нового синтаксису як і раніше можна використати управляючі послідовності всередині рядкового літерала. Отже, для вставляння символа табуляції потрібно застосовувати послідовність **\t:**

```
string greeting = string.Format("\tHello {0} you are {1} years old.", name.ToUpper(), age);
string greeting2 = $"\tHello {name.ToUpper()} you are {age} years old.";
```

Очікувано, що при побудові змінних типу **string** на льоту ви маєте право використати будь-який з двох підходів. Проте будьте уважні, бо у разі роботи з більш ранньою версією платформи .NET використання синтаксису інтерполяції рядків приведе до помилки на етапі компіляції. Отже, якщо потрібно забезпечити успішну компіляцію коду C# за допомогою багатьох версій компілятора, то безпечніше дотримуватися традиційного підходу з нумерованими заповнювачами.

Реалізуємо обробники для кнопок **Previous** і **Next**, щоб забезпечити прохід зображеннями. Коли користувач досягає кінця списку, здійснюється перехід у початок і навпаки.

```
private void btnPreviousImage_Click(object sender, RoutedEventArgs e)
{
  if (--_currImage < 0)
  {
    _currImage = _images.Count - 1;
  }
  imageHolder.Source = _images[_currImage];
}
private void btnNextImage_Click(object sender, RoutedEventArgs e)
{
  if (++_currImage >= _images.Count)
  {
    _currImage = 0;
  }
  imageHolder.Source = _images[_currImage];
}
```

Тепер можна запустити програму і перемикатися між всіма зображеннями.

Вбудовування ресурсів застосунку

Якщо файли зображень потрібно вбудувати прямо в збірку .NET як двійкові ресурси, тоді виберемо ці файли у вікні **Solution Explorer** (з теки

\Images, а не **\bin\Debug\Images**) і встановимо властивість **Build Action** (Дії при збиранні) в **Resource** (Ресурс), а властивість **Copy to Output Directory** – в **Do not copy** (Не копіювати), як показано на рис. 9.3.

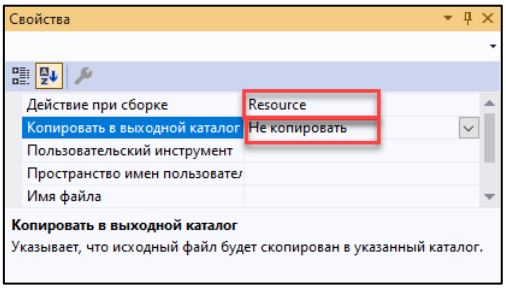


Рис. 9.3. Конфігурація зображень як вбудованих ресурсів

У меню **Build** (Збирання (рис. 9.4)) середовища Visual Studio виберемо пункт **Clean Solution** (Очистити рішення), щоб очистити поточний вміст теки **\bin\Debug\Images**, і повторно побудуємо проект. Відновимо вікно **Solution Explorer** і упевнимося в тому, що дані в каталозі **\bin\Debug\Images** відсутні.

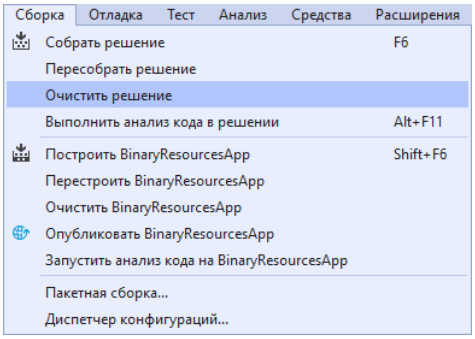


Рис. 9.4. Очищення рішення

При поточних параметрах збірки графічні дані більше не копіюються у вихідну теку, а вбудовуються в саму збірку. Прийом забезпечує наявність ресурсів, але також спричиняє збільшення розміру скомпільованої збірки.

Треба модифікувати код для завантаження зображень в список, отримуючи їх зі скомпільованої збірки:

```
// Отримайте зображення зі збірки, а потім завантажте їх
_images.Add(new BitmapImage(new Uri(@"/Images/Deer.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"/Images/Dogs.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"/Images/Dogs1.jpg", UriKind.Relative)));
_images.Add(new BitmapImage(new Uri(@"/Images/Welcome.jpg", UriKind.Relative)));
```

У цьому випадку більше не доведеться визначати шлях встановлення і можна просто задавати ресурси за іменами, які враховують назву початкового

підкаталогу. Також зверніть увагу, що при створенні об'єктів **Uri** вказується значення **Relative** перерахування **UriKind**. У цей момент виконувана програма є автономна сутність, яка може бути запущена з будь-якого місця розташування на машині, бо всі скомпільовані дані знаходяться всередині збірки.

Робота з об'єктними (логічними) ресурсами

При побудові застосунку WPF часто доводиться визначати великий обсяг розмітки XAML для використання у багатьох місцях вікна або можливо в множині вікон або проектів. Наприклад, нехай створена бездоганна кисть з лінійним градієнтом, визначення якої в розмітці займає **10** рядків. Тепер кисть потрібно застосувати як фоновий колір для кожного елемента **Button** у проекті, який складається з **8** вікон, тобто всього отримується **16** елементів **Button**.

Гірше, що можна було б зробити – копіювати і вставляти одну і ту ж розмітку XAML в кожен елемент управління **Button**. Очевидно, у результаті це могло б стати справжнім кошмаром при супроводі, бо всякий раз, коли треба скоректувати зовнішній вигляд і поведінку кисті, доводилося б вносити зміни у багато місць.

На щастя, *об'єктні ресурси* дозволяють:

- визначити фрагмент розмітки XAML;
- призначити йому ім'я;
- зберегти його у відповідному словнику для використання в майбутньому.

Подібно до двійкових ресурсів об'єктні ресурси часто компілюються у збірки, де вони потрібні. Проте в такій ситуації немає потреби возитися з властивістю **Build Action**. За умови, що розмітка XAML поміщена в коректне місце розміщення, компілятор подбає про все інше.

Взаємодія з об'єктними ресурсами є великою частиною процесу розробки застосунків WPF. Ви побачите, що об'єктні ресурси можуть бути набагато складніші, ніж спеціальна кисть. Допускається визначати анімацію на основі XAML, тривимірну візуалізацію, спеціальний стиль елемента управління, шаблон даних, шаблон елемента управління і багато чого іншого, і запаковувати кожну сутність у багаторазово використовуваний ресурс.

Роль властивості Resources

Як уже згадувалося, для використання у застосунку об'єктних ресурсів вони мають бути поміщені у відповідний об'єкт словника. Кожен похідний від **FrameworkElement** клас підтримує властивість **Resources**, яка інкапсулює об'єкт **ResourceDictionary**, що містить певні об'єктні ресурси. Об'єкт **ResourceDictionary** може зберігати елементи будь-якого типу, тому що оперує екземплярами **System.Object** і допускає маніпуляції з розмітки XAML або процедурного коду.

У WPF всі елементи управління плюс елементи **Window**, **Page** (використовувані при побудові навігаційних застосувань) і **UserControl** розширюють клас **FrameworkElement**, так що майже всі віджети надають

доступ до **ResourceDictionary**. Більше того, клас **Application**, хоча і не розширює **FrameworkElement**, але підтримує властивість з ідентичним ім'ям **Resources**, яка призначена для тієї ж мети.

Визначення ресурсів рівня вікна

Щоб приступити до дослідження ролі об'єктних ресурсів, створимо у Visual Studio новий проект застосунку WPF на ім'я **ObjectResourcesApp** і замінимо первинний елемент **Grid** горизонтально вирівняним диспетчером компонування **StackPanel**, всередині якого визначимо два елементи управління **Button** (чого цілком досить для пояснення ролі об'єктних ресурсів):

```
<StackPanel Orientation="Horizontal">
  <Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"/>
  <Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"/>
</StackPanel>
```

Виберемо кнопку **OK** і встановимо у властивості **Background** спеціальний тип кисті з використанням інтегрованого редактора кистей (див. ***Лекція 07.Служби візуалізації графіки WPF***). Вбудуємо всередину області між дескрипторами **<Button>** і **</Button>** цей тип кисті для кнопки **OK**:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20">
  <Button.Background>
    <RadialGradientBrush>
      <GradientStop Color="#FFC44EC4" Offset="0" />
      <GradientStop Color="#FF829CEB" Offset="1" />
      <GradientStop Color="#FF793879" Offset="0.669" />
    </RadialGradientBrush>
  </Button.Background>
</Button>
```

Щоб дозволити використати цю кисть також і для кнопки **Cancel** (Відміна), область визначення **RadialGradientBrush** має бути розширена до словника ресурсів батьківського елемента. Наприклад, якщо перемістити **RadialGradientBrush** в **StackPanel**, то обидві кнопки зможуть застосовувати одну і ту ж кисть, бо вони є дочірніми елементами того ж самого диспетчера компонування. Ще краще, кисть можна було б запакувати в словник ресурсів самого вікна, внаслідок чого її могли б вільно використовувати всі елементи вмісту вікна.

Коли потрібно визначити ресурс, для встановлення властивості **Resources** власника застосовується синтаксис "*властивість-елемент*". Крім того, елементу ресурсу призначається значення **x:Key**, яке використовуватиметься іншими частинами вікна для посилання на об'єктний ресурс. Майте на увазі, що атрибути **x:Key** і **x:Name** – не одне і те ж. Атрибут **x:Name** дозволяє *отримувати* доступ до об'єкта як до змінної-члена у *файлі коду*, тоді як атрибут **x:Key** дає можливість *посилатися* на елемент у *словнику ресурсів*.

Середовище Visual Studio дозволяє перемістити ресурс на вищий рівень використовуючи відповідне вікно **Properties**. Щоб зробити це, спочатку знадобиться ідентифікувати властивість, що має складний об'єкт, який потрібно

запакувати у вигляді ресурсу (тут, властивість **Background**). Праворуч від властивості знаходиться невеликий квадрат, клацання на якому спричиняє відкриття спливаючого меню. Виберемо в ньому пункт **Convert to New Resource** (Перетворити в новий ресурс), як продемонстровано на рис. 9.5.

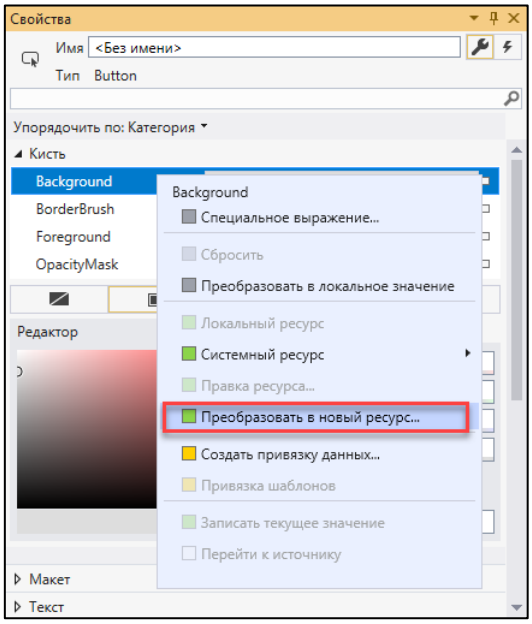


Рис. 9.5. Переміщення складного об'єкта в контейнер ресурсів

Буде запрошено ім'я ресурсу (**myBrush**) і запропоновано вказати, куди ресурс помістити. Залишимо відміченим перемикач **This document** (Цей документ) – за замовчуванням (рис. 9.6).

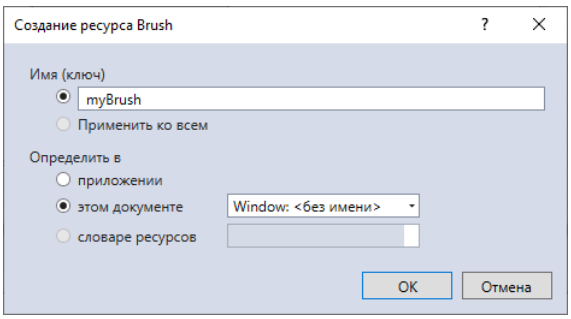


Рис. 9.6. Призначення імені об'єктному ресурсу

У результаті, створена кисть переміститься всередину дескриптора **Window.Resources**:

```
<Window.Resources>
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF829CEB" Offset="1" />
```

```
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</Window.Resources>
```

Властивість **Background** елемента **Button** (**OK**) оновлюється для роботи з новим ресурсом:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
    Background="{DynamicResource myBrush}"/>
```

Майстер створення ресурсів визначив новий ресурс як динамічний {**DynamicResource**}. Динамічні ресурси розглядаються пізніше, а наразі змінимо тип ресурсу на статичний {**StaticResource**}:

```
<Button Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
    Background="{StaticResource myBrush}"/>
```

Для оцінки переваги, модифікуємо властивість **Background** кнопки **Cancel** (Відміна), задавши в ньому той же самий ресурс {**StaticResource**}, після чого можна буде побачити повторне використання у дії:

```
<Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"
    Background="{StaticResource myBrush}"/>
```

Розширення розмітки {StaticResource}

Розширення розмітки {**StaticResource**} використовує ресурс один раз (при ініціалізації) і залишається «*підключеним*» до первинного об'єкта протягом часу життя застосунку. Деякі властивості (на зразок градієнтних переходів) оновлюватимуться, але у разі створення нового елемента **Brush**, наприклад, елемент управління не оновиться. Щоб поглянути на таку поведінку у дії, додамо властивість **Name** та обробник події **Click** до кожного елемента управління **Button**:

```
<Button Name="Ok" Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
    Background="{StaticResource myBrush}" Click="Ok_Click"/>
<Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"
    Background="{StaticResource myBrush}" Click="Cancel_Click"/>
```

Потім помістимо в обробник події **Ok_Click()** наступний код:

```
private void Ok_Click(object sender, RoutedEventArgs e)
{
  // Отримати кисть і внести зміну.
  var b = (RadialGradientBrush) Resources["myBrush"];
  b.GradientStops[1] = new GradientStop(Colors.Black, 0.0);
}
```

На замітку: Тут для пошуку ресурсу за іменем використовується *індексатор* **Resources**. Але, треба зважати, що якщо ресурс не знаходиться, тоді згенерується *виключення* часу виконання. Можна також застосовувати метод **TryFindResource()**, який не спричиняє генерацію виключення, а просто повертає **null**, якщо вказаний ресурс не знайдений.

Запустивши програму і клацнувши на кнопці **OK**, можна помітити, що градієнти відповідним чином змінюються (рис. 9.7).

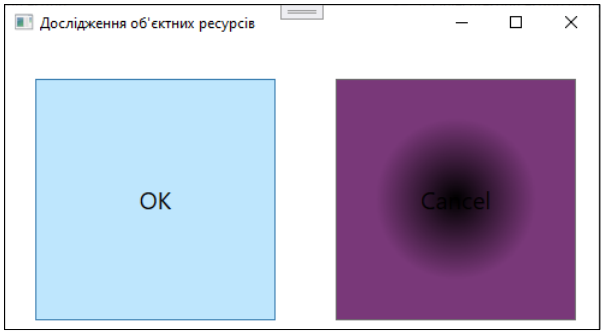


Рис. 9.7. Клацання на кнопці **OK**, спричиняє зміну градієнта

Додамо в обробник події **Cancel_Click()** такий код:

```
private void Cancel_Click(object sender, RoutedEventArgs e)
{
   // Помістити в комірку myBrush абсолютно нову кисть.
   Resources["myBrush"] = new SolidColorBrush(Colors.Red);
}
```

Запустивши програму знову і клацнувши **Cancel**, з'ясуємо, що нічого не відбувається.

Розширення розмітки {DynamicResource}

Для властивості також можна використати розширення розмітки **DynamicResource**. Щоб з'ясувати різницю, змінимо розмітку для кнопки **Cancel**, як показано нижче:

```
<Button Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"
    Background="{DynamicResource myBrush}" Click="Cancel_Click"/>
```

Цього разу в результаті клацання на кнопці **Cancel** колір фону для кнопки **Cancel** змінюється, колір фону для кнопки **OK** залишається тим самим. Причина в тому, що розширення розмітки {**DynamicResource**} здатне виявляти заміну внутрішнього об'єкта, заданого за допомогою ключа, новим об'єктом. Як і можна було припустити, така можливість вимагає додаткової інфраструктури часу виконання, так що {**StaticResource**} зазвичай треба використовувати, тільки якщо *не планується* замінювати об'єктний ресурс іншим об'єктом під час виконання з повідомленням всіх елементів, які використовують цей ресурс (див. рис. 9.8).

Ресурси рівня застосунку

Якщо у словнику ресурсів вікна є об'єктні ресурси, їх можуть використовувати всі елементи цього вікна, але не інші вікна застосунку. Щоб *розділяти* ці ресурси у застосунку в цілому, треба визначити їх на рівні

застосунку, а не на рівні вікна. У Visual Studio відсутні способи автоматизації такої дії, а тому ми просто *виріжемо* наявне визначення об'єкта кисті з області **Windows.Resource** і помістимо його в область **Application.Resources** файла **App.xaml**.

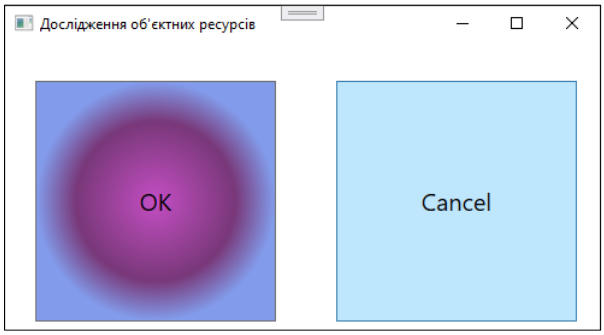


Рис. 9.8. Розширення розмітки {**DynamicResource**} кнопки **Cancel**, спричиняє зміну градієнта

Тепер будь-яке *додаткове вікно* або *елемент управління* в застосунку здатен працювати з цим об'єктом кисті. Для того, щоб ресурси рівня застосунку стали *доступними* додатковим вікнам або елементам управління потрібно встановити властивість **Background** додаткового елемента управління (рис. 9.9). Схема встановлення властивості **Background** описана вище, у пункті **Визначення ресурсів рівня вікна**, але із заданням її значення як «**Локальний ресурс**».

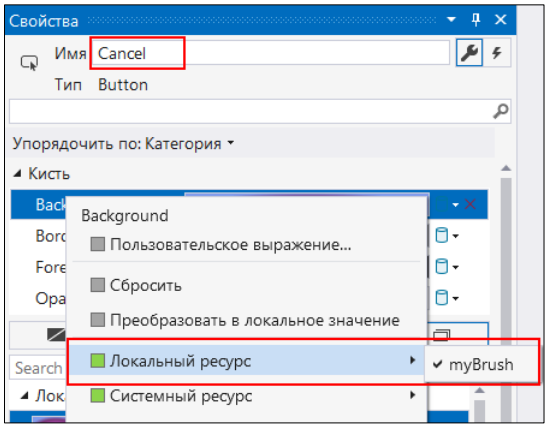


Рис. 9.9. Використання ресурсів рівня застосунку

Після цього код розмітки в диспетчері компонування **StackPanel** стане наступним:

```
<StackPanel Orientation="Horizontal" Background="{DynamicResource myBrush}">
  <Button Name="Ok" Margin="25" Height="200" Width="200" Content="OK" FontSize="20"
      Background="{StaticResource myBrush}" Click="Ok_Click"/>
```

```
  <Button Name="Cancel" Margin="25" Height="200" Width="200" Content="Cancel" FontSize="20"
      Background="{DynamicResource myBrush}" Click="Cancel_Click" />
</StackPanel>
```

Визначення об'єднаних словників ресурсів

Ресурсів рівня застосунку часто буває цілком достатньо, але вони нічим не допоможуть, якщо ресурси *розділяти* між проектами. У такому разі потрібно визначити *об'єднаний словник ресурсів*. Вважайте його *бібліотекою класів для ресурсів* WPF; вона є файлом **.xaml**, який містить колекцію ресурсів. Єдиний проект може мати будь-яке потрібне число таких файлів (один для кистей, один для анімації і т. д.), кожен з яких може бути доданий в діалоговому вікні **Add New Item**, яке відкривається через меню **Project** (рис. 9.10).

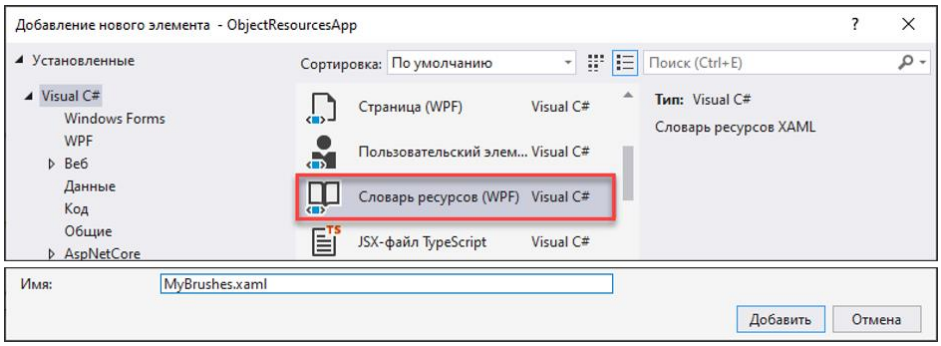


Рис. 9.10. Додавання нового об'єднаного словника ресурсів

Виріжемо поточні ресурси з області визначення **Application.Resources** і перенесемо їх у словник в новому файлі **MyBrushes.xaml**:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:local="clr-namespace:ObjectResourcesApp"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF827CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</ResourceDictionary>
```

Хоча цей словник ресурсів є частиною проекту, всі словники ресурсів мають бути об'єднані (зазвичай на рівні застосунку) в єдиний словник ресурсів, щоб їх можна було використати. Для цього застосовується наступний формат у файлі **App.xaml** (зверніть увагу, що багато словників ресурсів об'єднуються за рахунок додавання елементів **ResourceDictionary** в область **ResourceDictionary.MergedDictionaries**):

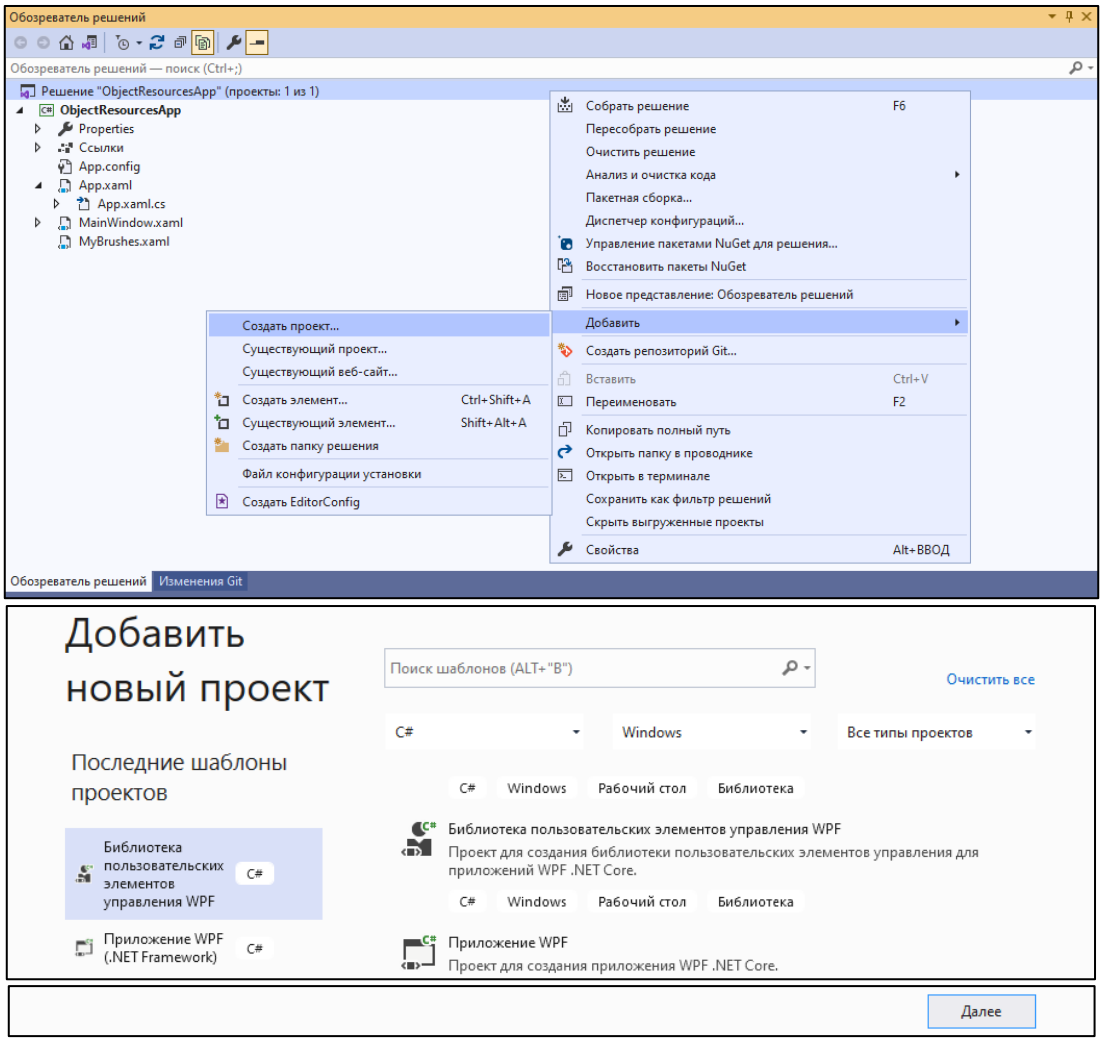
```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionanes>
      <ResourceDictionary Source="MyBrushes.xaml"/>
```

```
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Проблема такого підходу в тому, що кожен файл ресурсів потрібно буде додавати в кожен проект, який потребує ресурсів. Вдаліший підхід до розділення ресурсів полягає у визначенні *бібліотеки класів* .NET для спільного використання проектами, чим ми і займемося.

Визначення збірки, яка включає тільки ресурси

Найлегший спосіб побудови збірки з одних ресурсів передбачає створення проекту WPF **User Control Library** (Бібліотека елементів управління WPF користувача). Додамо такий проект (на ім'я **MyBrushesLibrary**) в поточне рішення, вибравши пункт меню **Add|New Project** (Додати|Створити проект) у Visual Studio (рис. 9.11).



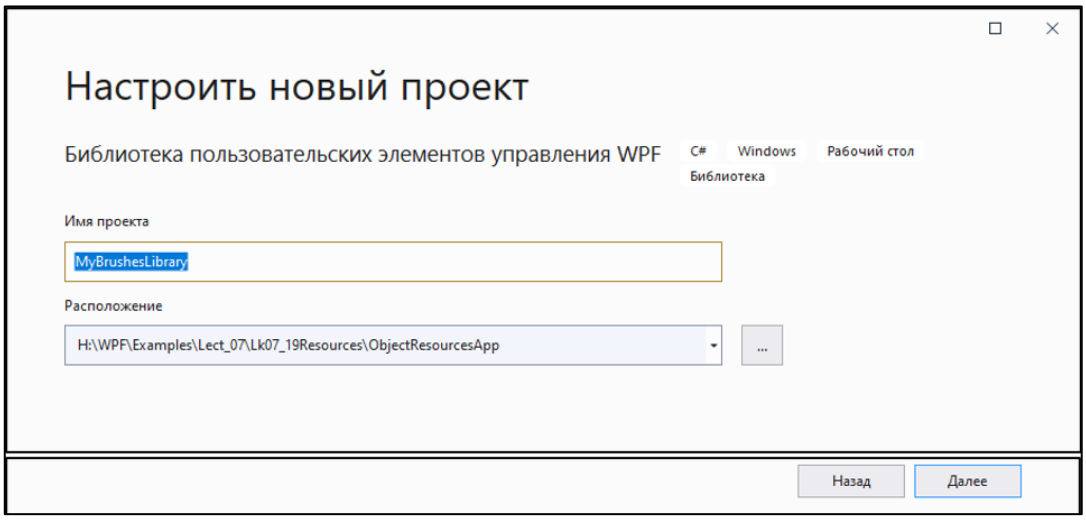


Рис. 9.11. Додавання проекту WPF **User Control Library**
як початкової точки для побудови бібліотеки з одних ресурсів

Тепер з отриманого проекту:

1. Видалимо файл **UserControl1.xaml**.
2. Перетягнемо файл **MyBrushes.xaml** в проект **MyBrushesLibrary** і видалимо його з проекту **ObjectResourcesApp**.
3. Відкриємо файл **MyBrushes.xaml** в проекті **MyBrushesLibrary** і змінимо простір імен xmlns:local="clr-namespace:ObjectResourcesApp" на xmlns:local="clr-namespace:MyBrushesLibrary">.

Так тепер повинен виглядати вміст файла **MyBrushes.xaml**:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:local="clr-namespace:MyBrushesLibrary"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <RadialGradientBrush x:Key="myBrush">
    <GradientStop Color="#FFC44EC4" Offset="0" />
    <GradientStop Color="#FF827CEB" Offset="1" />
    <GradientStop Color="#FF793879" Offset="0.669" />
  </RadialGradientBrush>
</ResourceDictionary>
```

Скомпілюємо проект WPF **User Control Library** – під компіляцією розуміємо процес отримання бібліотечного файла **MyBrushesLibrary.dll**.

Зробимо це так:

1. У вікні **Оглядач рішень**, клацнувши на бібліотечному проекті **MyBrushesLibrary** активуємо контекстне меню.
2. У ньому активуємо пункт **Перезібрати**. (див. рис. 9.12)
3. Переконайтеся в тому, файл **MyBrushesLibrary.dll** отримано.

Потім встановимо посилання на бібліотеку **MyBrushesLibrary.dll** з проекту **ObjectResourcesApp** із використанням діалогового вікна **Add Reference** (Додавання посилання).

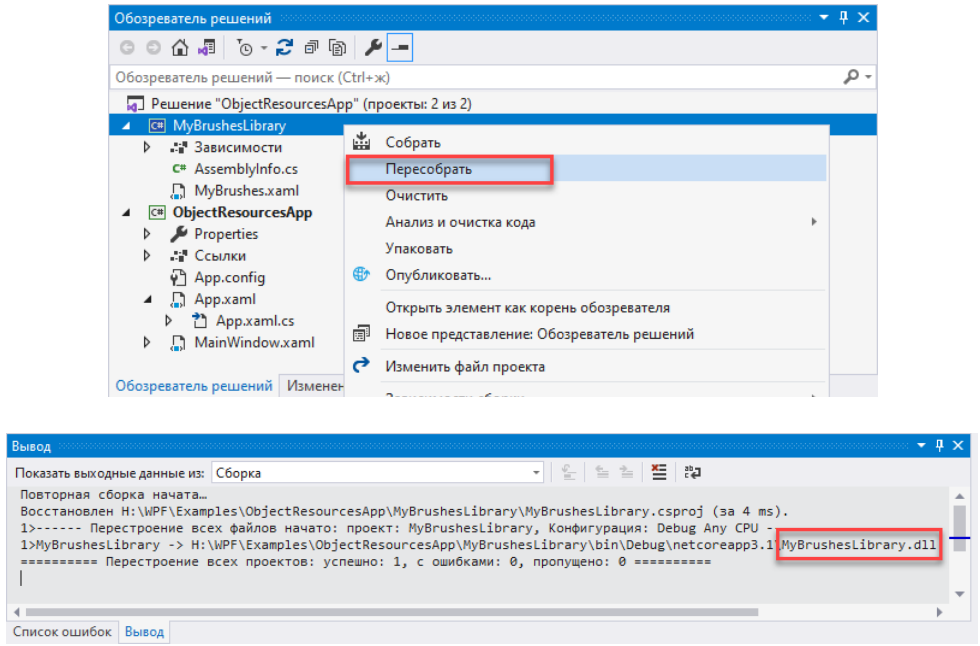


Рис. 9.12. Процес компіляції проекту WPF **User Control Library** (**MyBrushesLibrary**)

Зробимо це так:

1. Клацніть пункт **Посилання** проекту **ObjectResourcesApp** для активації діалогового вікна (див. рис. 9.13).

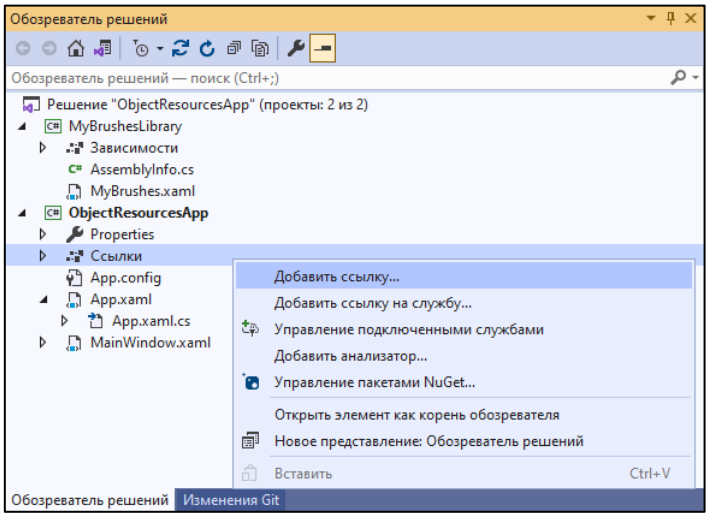


Рис. 9.13. Діалогове вікно **Додавання посилання**

2. У вікні **Менеджер посилань** – **ObjectResourcesApp**, яке з'явилося виберіть пункт **Проекти|Рішення** відмітьте галочкою збірку **MyBrushesLibrary** і натисніть **OK**. (див. рис. 9.14).

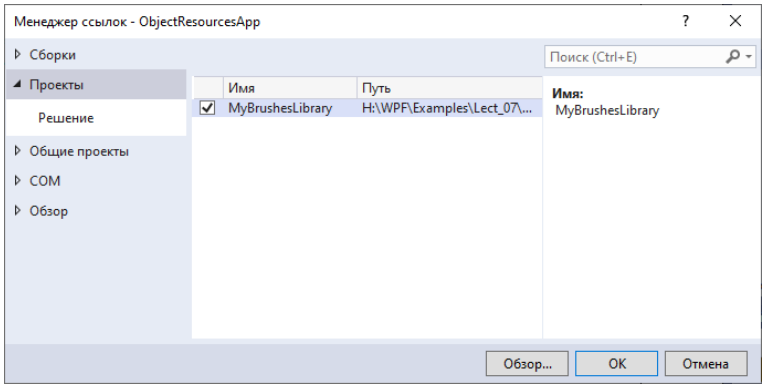


Рис. 9.14. Вікно **Менеджер посилань**

3. Перевірте наявність файла **MyBrushesLibrary** у пункті **Посилання** проекту **ObjectResourcesApp**.

Тепер об'єднаємо наявні двійкові ресурси зі словником ресурсів рівня застосунку з проекту **ObjectResourcesApp**. Проте така дія вимагає використання досить дивного синтаксису:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionanes>
        <!-- Синтаксис виглядає як /ІмяЗбірки;Component/ІмяФаилаXamlyЗбірці.xaml -->
        <ResourceDictionary Source = "/MyBrushesLibrary;Component/MyBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

Врахуйте, що цей рядок чутливий до пропусків. Якщо біля двокрапки чи похилих рисок будуть присутніми зайві пропуски, то виникнуть помилки часу виконання. Перша частина рядка є дружнім ім'ям зовнішньої бібліотеки (без файлового розширення). Після двокрапки йде слово **Component**, а за ним ім'я скомпільованого двійкового ресурсу, яке буде ідентичним імені початкового словника ресурсів XAML.

Отже, знайомство з системою управління ресурсами WPF завершене. Описані тут прийоми доведеться часто застосовувати у більшості застосунків, які розробляються (а можливо і у всіх).

Резюме

Ми розглянули систему управління ресурсами WPF. Розпочали дослідження роботи з двійковими ресурсами і ролі об'єктних ресурсів. Ви

довідалися, що об'єктні ресурси є іменованими фрагментами розмітки XAML, які можуть бути збережені в різноманітних місцях з метою багатократного використання вмісту.

Контрольні питання

1. Які види ресурсів підтримує інфраструктура WPF?
2. Як ви розумієте поняття ресурсу інфраструктури WPF, званого двійковим ресурсом?
3. Як ви розумієте поняття ресурсу інфраструктури WPF, званого об’єктним або логічним ресурсом?
4. Яка схема включення в проект незв’язаних файлів ресурсів?
5. Яка схема конфігурування незв’язаних ресурсів?
6. Коли використовується вбудовування ресурсів прямо в збірку .NET? В чому перевага такого способу включення ресурсів?
7. Як ви розумієте поняття багаторазового використання ресурсу?
8. Яке відношення мають об’єктні ресурси до багаторазово використовуваних ресурсів?
9. Який взаємозв’язок між FrameworkElement, властивістю Resources та об’єктом ResourceDictionary?
10. Чи може об’єкт ResourceDictionary зберігати елементи будь-якого типу? Якщо так, то за рахунок чого?
11. Чи підтримує клас Application властивість Resource?
12. Який механізм визначення ресурсів рівня вікна? Приведіть приклад.
13. Як перемістити складний об’єкт в контейнер ресурсів? Наведіть приклад.
14. Яка роль розширення розмітки {StaticResource} у створенні під’єднання до первинного об’єкта на проміжок часу життя застосунку? Наведіть приклад.
15. Яка роль розширення розмітки {DynamicResource} у створенні під’єднання до первинного об’єкта на проміжок часу життя застосунку? Наведіть приклад.
16. В чому полягає різниця між розширеннями розмітки {StaticResource} та {DynamicResource}?
17. Яка схема створення ресурсу рівня застосунку?
18. В чому різниця між ресурсами рівня вікна та ресурсами рівня застосунку?
19. За якою схемою створюється об’єднаний словник ресурсів?
20. Скільки в єдиному проекті може бути словників ресурсів?
21. За якою схемою будується збірка (файл з розширенням .dll), яка складається з одних ресурсів?
22. Як реалізувати посилання на створену збірку ресурсів?

Лекція 10. Реалізація анімації WPF

Служби анімації WPF

На додаток до служб графічної візуалізації, які розглядалися в ***Лекція 07.Служби візуалізації графіки WPF*** та ***Лекція 08. Візуалізація графічних даних***, інфраструктура WPF пропонує API-інтерфейс для підтримки служб анімації.

Не заперечуючи те, що API-інтерфейси анімації WPF безперечно могли б використовуватися для згаданих вище цілей, анімація може застосовуватися всякий раз, коли застосунку потрібно надати особливий стиль. Наприклад, можна було б побудувати анімацію для кнопки на екрані, щоб вона злегка збільшувалася, коли курсор миші знаходиться всередині її меж (і поверталася до попередніх розмірів, коли курсор покидає межі). Або ж можна було б передбачити анімацію для вікна, забезпечивши його закриття з використанням певного візуального ефекту, такого як поступове зникнення до повної прозорості. Застосункам, більше орієнтованим на бізнес-застосунки, може бути реалізоване поступове збільшення чіткості відображення повідомлень про помилки на екрані, покращуючи сприйняття інтерфейсу користувача. Фактично підтримка анімації WPF може використовуватися в застосунках будь-якого роду (бізнес-застосунки, мультимедіа-програми, відеоігри і т. д.) всякий раз, коли треба створити привабливіше враження у користувачів.

Як і з багатьма іншими аспектами WPF, з побудовою анімації не пов'язано нічого нового. Єдина особливість в тому, що на відміну від інших API-інтерфейсів, які ви могли використовувати у минулому (включаючи Windows Forms), розробники не повинні створювати потрібну інфраструктуру вручну. У WPF не доведеться:

- заздалегідь створювати фонові потоки або таймери, використовувані для просування вперед анімаційної послідовності;
- визначати спеціальні типи для представлення анімації;
- очищати і перемальовувати зображення;
- реалізовувати стомливі математичні обчислення.

Подібно до інших аспектів WPF-анімацію можна будувати цілком в розмітці XAML, цілком в коді C# або з використанням комбінації того й іншого.

> **<u>На замітку!</u>** У середовищі Visual Studio відсутня підтримка створення анімації за допомогою будь-яких графічних інструментів і тому потрібно вводити розмітку XAML вручну. Тим не менш, у складі Visual Studio 2019 наявний продукт Microsoft Blend має вбудований редактор анімації, який здатний істотно спростити рішення завдань.

Роль класів анімації

Щоб з’ясувати суть підтримки анімації WPF, потрібно розпочати з розгляду класів анімації з простору імен **System.Windows.Media.Animation**

збірки **PresentationCore.dll**. Тут ви знайдете понад **100** різних класів, які містять слово **Animation** у своїх іменах.

Усі класи такого роду можуть бути віднесені до однієї з трьох великих категорій:

1. По-перше, будь-який клас, який спадкує угоду про іменування вигляду ***ТипДаних*Animation** (**ByteAnimation**, **ColorAnimation**, **DoubleAnimation**, **Int32Animation** і т. п.), дозволяє працювати з анімацією *лінійною інтерполяцією*. Вона забезпечує плавну зміну значення в часі від початкового до кінцевого.
2. По-друге, класи, які спадкують угоду про іменування вигляду ***ТипДаних*AnimationUsingKeyFrames** (**StringAnimationUsingKeyFrames**, **DoubleAnimationUsingKeyFrames**, **PointAnimationUsingKeyFrames** і т. п.), представляють анімацію *ключовими кадрами*, яка дозволяє проходити в циклі набором певних значень за заданий період часу. Наприклад, ключові кадри можна застосовувати для зміни напису на кнопці, проходячи в циклі послідовністю окремих символів.
3. По-третє, класи, які спадкують угоду про іменування вигляду ***ТипДаних*AnimationUsingPath** (**DoubleAnimationUsingPath**, **PointAnimationUsingPath** і т. п.), представляють анімацію на *основі шляху*, що дозволяє переміщати об'єкти по заданому шляху. Наприклад, в застосунку глобального позиціонування (**GPS**) анімацію на основі шляху можна використовувати для переміщення елемента найкоротшим маршрутом до місця, заданого користувачем.

Цілком очевидно, згадані класи не застосовуються для того, щоб якось безпосередньо надати анімаційну послідовність змінній певного типу даних (врешті-решт, як можна було б виконати анімацію значення **9**, використовуючи об'єкт **Int32Animation**?).

За приклад візьмемо властивості **Height** і **Width** типу **Label**, які є властивостями залежності, які запаковують значення **double**. Щоб визначити анімацію, яка збільшуватиме висоту мітки з часом, можна підключити об'єкт **DoubleAnimation** до властивості **Height** і дозволити WPF потурбуватися про деталі виконання справжньої анімації. Або інший приклад: якщо вимагається реалізувати перехід кольору кисті від зеленого до жовтого впродовж **5** секунд, то це можна зробити із застосуванням типу **ColorAnimation**.

Треба зрозуміти, що класи **Animation** можуть підключатися до будь-якої властивості залежності заданого об'єкта, яка має відповідний тип. Як пояснювалося в ***Лекція 06. Модель прив'язки даних WPF***, властивості залежності є спеціальною формою властивостей, яку потребують багато служб WPF, включаючи анімацію, прив'язку даних і стилі.

За угодою властивість залежності визначається як статичне, доступне тільки для читання поле класу, ім'я якого утворюється додаванням слова **Property** до нормального імені властивості. Наприклад, для звернення до властивості залежності для властивості **Height** класу **Button** в коді використовуватиметься **Button.HeightProperty**.

Властивості To, From і By

У всіх класах **Animation** визначені наступні ключові властивості, які управляють початковими і кінцевими значеннями, використовуваними для виконання анімації представляє:

- **To** – кінцеве значення анімації;
- **From** – початкове значення анімації;
- **By** – загальну величину, на яку анімація змінює початкове значення.

Незважаючи на той факт, що всі класи підтримують властивості **To**, **From** і **By**, вони не отримують їх через віртуальні члени базового класу. Причина в тому, що типи, які лежать в основі, запаковані всередині вказаних властивостей, варіюються в широких межах (цілі числа, кольори, об'єкти **Thickness** і т. д.), і надання всіх можливостей через єдиний базовий клас привело б до дуже складних кодових конструкцій.

У зв'язку із сказаним може виникнути питання: чому не використовувалися узагальнення .NET для визначення єдиного узагальненого класу анімації з одиночним параметром типу (скажімо, **Animate<T>**)? Знову-таки, через те, що існує величезне число типів даних (кольори, вектори, цілі числа, рядки і т. д.), використовуваних для анімації властивостей залежності, рішення виявилося б не настільки зрозумілим, як можна було б очікувати (не кажучи вже про те, що XAML забезпечує лише обмежену підтримку узагальнених типів).

Роль базового класу Timeline

Хоча для визначення віртуальних властивостей **To**, **From** і **By** не використовувався єдиний базовий клас, класи **Animation** все ж розділяють загальний базовий клас – **System.Windows.Media.Animation.Timeline**. Цей тип пропонує набір додаткових властивостей, які управляють темпом просування анімації (табл. 10.1).

Таблиця 10.1. Основні властивості базового класу **Timeline**

Властивості	Опис
AccelerationRatio, DecelerationRatio, SpeedRatio	Ці властивості застосовуються для управління загальним темпом просування анімаційної послідовності.
AutoReverse	Ця властивість отримує або встановлює значення, яке вказує, чи повинна часова шкала відтворюватися у зворотному напрямі після завершення ітерації вперед (стандартним значенням є **false**).
BeginTime	Ця властивість отримує або встановлює час запуску часової шкали. Стандартним значенням є **0**, яке запускає анімацію негайно.
Duration	Ця властивість дозволяє встановлювати тривалість відтворення часової шкали.
FileBehavior, RepeatBehavior	Ці властивості використовуються для управління тим, що повинно статися після завершення часової шкали (повторення анімації, нічого і т. д.).

Реалізація анімації в коді C#

Ми побудуємо вікно, яке містить елемент **Button**, що має досить дивну поведінку: коли на нього наводиться курсор миші, він обертається навколо

свого лівого верхнього кута. Розпочнемо зі створення у Visual Studio нового проекту застосунку WPF на ім'я **SpinningButtonAnimationApp**. Модифікуємо початкову розмітку, як показано нижче (зверніть увагу на обробку події **MouseEnter** кнопки):

```
// Додайте до просторів імен, пов'язаний з анімацією: using System.Windows.Media.Animation.
private bool _isSpinning = false;
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
  if (!_isSpinning)
  {
    _isSpinning = true;
    // Створити об'єкт DoubleAnimation і зареєструвати
    // його з подією Completed.
    var dblAnim = new DoubleAnimation();
    dblAnim.Completed += (o, s) => { _isSpinning = false; };
    // Встановити початкове і кінцеве значення.
    dblAnim.From = 0;
    dblAnim.Te = 360;
    // Створити об'єкт RotateTransform і присвоїти
    // його властивості RenderTransform кнопки.
    var rt = new RotateTransform();
    btnSpinner.RenderTransform = rt;
    // Виконати анімацію об'єкта RotateTransform.
    rt.BeginAnimation(RotateTransform.AngleProperty, dblAnim);
  }
}
```

Створимо шаблон обробника події:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{

}
```

Перша значна задача методу **btnSpinner_MouseEnter()** пов'язана з конфігуруванням об'єкта **DoubleAnimation**, який розпочинатиме зі значення **0** і закінчувати значенням **360**. Зверніть увагу, що для цього об'єкта також обробляється подія **Completed**, де здійснюється перемикання булевої змінної рівня класу, яка використовується для того, щоб виконувана анімація, не була скинута в початок.

Потім створюється об'єкт **RotateTransform**, який підключається до властивості **RenderTransform** елемента управління **Button (btnSpinner)**. Нарешті, об'єкт **RenderTransform** інформується про початок анімації його властивості **Angle** з використанням об'єкта **DoubleAnimation**. Реалізація анімації в коді зазвичай здійснюється шляхом виклику методу **BeginAnimation()** і передачі йому лежачої в основі *властивості залежності*, до якої потрібно застосувати анімацію (згадайте, що за угодою вона визначена як статичне поле класу), і пов'язаного об'єкта анімації.

Давайте додамо в програму ще одну анімацію, яка змусить кнопку після клацання плавно ставати невидимою. Спершу створимо обробник події **Click** кнопки **btnSpinner** з приведеним нижче кодом:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  var dblAnim = new DoubleAnimation
  {
    From = 1.0,
    To = 0.0
  };
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

У коді обробника події **btnSpinner_Click()** змінюється властивість **Opacity**, щоб поступово приховати кнопку з поля зору. Проте зараз це не просто, тому що кнопка обертається занадто швидко. Управління темпом анімації пояснюється в наступному пункті.

Управління темпом анімації

За замовчуванням анімація займатиме приблизно одну секунду для переходу між значеннями, які присвоєні властивостям **From** і **To**. Отже, кнопка має в розпорядженні одну секунду, щоб здійснити оберт на **360** градусів, і в той же час впродовж однієї секунди вона поступово сховається із виду (після клацання на ній).

Визначити інший період часу для переходу анімації можна за допомогою властивості **Duration** об'єкта анімації, якому присвоюється об'єкт **Duration**. Зазвичай проміжок часу встановлюється шляхом передачі об'єкта **TimeSpan** конструктору класу **Duration**. Погляньте на показану далі зміну, в результаті якої кнопці буде виділено чотири секунди на обертання:

```
private void btnSpinner_MouseEnter(object sender, MouseEventArgs e)
{
  if (!_isSpinning)
  {
    _isSpinning = true;
    // Створити об'єкт DoubleAnimation і зареєструвати
    // його з подією Completed.
    var dblAnim = new DoubleAimation();
    dblAnim.Completed += (o, s) => { _isSpinning = false; };
    // Встановимо час завершення оберту кнопки чотири секунди.
    dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
    …
}
```

Завдяки такій модифікації з'явиться реальний шанс клацнути на кнопці під час її обертання, після чого вона плавно зникне.

На замітку! Властивість **BeginTime** класу **Animation** також приймає об'єкт **TimeSpan**. Згадайте, що цю властивість можна встановлювати для задання часу очікування перед запуском анімаційної послідовності.

Запуск у зворотному порядку і циклічне виконання анімації

За рахунок встановлення в **true** властивості **AutoReverse** об'єктам **Animation** вказується про потребу запуску анімації в зворотному напрямку після її завершення. Наприклад, якщо потрібно, щоб кнопка знову стала видимою після зникнення, можна написати наступний код:

```
private void btnSpinner_Click(object sender, RoutedEventArgs e)
{
  DoubleAnimation dblAnim = new DoubleAnimation
  {
    From = 1.0,
    To = 0.0
  };
  // Після зникнення кнопки, викликаного клацанням на кнопці, вона запуститься
  // у зворотному порядку, т. т. вона поступово з'явиться на екрані.
  dblAnim.AutoReverse = true;
  btnSpinner.BeginAnimation(Button.OpacityProperty, dblAnim);
}
```

Якщо потрібно, щоб анімація повторювалася кілька разів (або ніколи не припинялася), тоді можна скористатися властивістю **RepeatBehavior**, загальною для всіх класів **Animation**. Передаючи конструктору просте числове значення, можна задати жорстко закодоване число повторень. З іншого боку, якщо передати конструктору об'єкт **TimeSpan**, то можна задати час, впродовж якого анімація повинна повторюватися. Нарешті, щоб виконувати анімацію нескінченно, властивість **RepeatBehavior** можна встановити в **RepeatBehavior.Forever**. Розглянемо наступні способи зміни поведінки повтору одного з двох об'єктів **DoubleAnimation**, використовуваних в прикладі:

```
// Повторювати нескінченно.
dblAmm.RepeatBehavior = RepeatBehavior.Forever;
// Повторювати три рази.
dblAmm.RepeatBehavior = new RepeatBehavior(3);
// Повторювати впродовж 30 секунд.
dblAmm.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(30));
```

Дослідження прийомів додавання анімації до аспектів якогось об'єкта з використанням коду C# та API-інтерфейсу анімації WPF завершене. Тепер подивимося, як робити те ж саме за допомогою розмітки XAML.

Реалізація анімації в розмітці XAML

Реалізація анімації в розмітці подібна до її реалізації в коді, принаймні, для простих анімаційних послідовностей. Коли потрібно створити складнішу анімацію, яка включає зміну значень багатьох властивостей одночасно, обсяг розмітки може помітно збільшитися. Навіть у разі застосування якогось

інструменту для генерування анімації, яка базується на розмітці XAML, важливо знати основи представлення анімації у XAML, бо тоді полегшується завдання модифікації і налаштування згенерованого інструментом вмісту.

Переважно створення анімації подібно до того, що ви вже бачили: як і раніше здійснюється конфігурування об'єкта **Animation**, який потім асоціюється з властивістю об'єкта. Проте, велика відмінність пов'язана з тим, що розмітка XAML не підтримує викликів методів. У результаті замість виклику **BeginAnimation()** використовується *розкадровування* як проміжний рівень.

Давайте розглянемо повний приклад анімації, визначеної в термінах XAML, і детально її проаналізуємо. Приведене далі визначення XAML відображатиме вікно, яке містить єдину мітку. Після того, як об'єкт **Label** завантажився в пам'ять, він починає анімаційну послідовність, під час якої розмір шрифту збільшується від **12** до **100** точок за період в чотири секунди. Анімація повторюватиметься стільки часу, скільки об'єкт залишається завантаженим в пам'ять. Розмітка знаходиться у файлі **GrowLabelFont.xaml**, так що його вміст потрібно скопіювати в редактор **Kaxaml**, натиснути клавішу <**F5**> і постежити за поведінкою.

Файл **GrowLabelFont.xaml**:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="200" Width="600" Title="Growing Label Font!">
  <StackPanel>
    <Label Content = "Interesting...">
      <Label.Triggers>
        <EventTrigger RoutedEvent = "Label.Loaded">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard TargetProperty = "FontSize">
                <DoubleAnimation From = "12" To = "100" Duration = "0:0:4"
                    RepeatBehavior = "Forever"/>
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Label.Triggers>
    </Label>
  </StackPanel>
</Window>
```

А тепер детально розберемо приклад.

Роль розкадровувань

Рухаючись від самого глибоко вкладеного елемента назовні, першим ми зустрічаємо елемент <**DoubleAnimation**>, який звертається до тих же самих властивостей, які встановлювалися у процедурному коді (**From**, **To**, **Duration** і **RepeatBehavior**):

```
<DoubleAnimation From = "12" To = "100" Duration = "0:0:4" RepeatBehavior = "Forever"/>
```

Як згадувалося раніше, елементи **Animation** поміщаються всередину елемента **Storyboard**, використовуваного для відображення об'єкта анімації на задану властивість батьківського типу через властивість **TargetProperty**, якою у цьому випадку є **FontSize**. Елемент **Storyboard** завжди знаходиться всередині батьківського елемента з іменем **BeginStoryboard**:

```
<BeginStoryboard>
  <Storyboard TargetProperty = "FontSize">
    <DoubleAnimation From = "12" To = "100" Duration = "0:0:4" RepeatBehavior = "Forever"/>
  </Storyboard>
</BeginStoryboard>
```

Роль тригерів подій

Після того, як елемент **BeginStoryboard** визначений, має бути задана дія якогось виду, яка приведе до запуску анімації. Інфраструктура WPF пропонує декілька різних способів реагування на умови часу виконання в розмітці, один з яких називається *тригером*. З високорівневого погляду тригер можна вважати способом реагування на подію в розмітці XAML без потреби у написанні процедурного коду.

Зазвичай коли відповідь на подію реалізується в C#, пишеться спеціальний код, який буде виконаний при настанні події. Проте тригер – всього лише спосіб отримати повідомлення про те, що деяка подія сталася (завантаження елемента в пам'ять, наведення на нього курсора миші, отримання їм фокусу і т. д.).

Отримавши повідомлення про появу події, можна запускати розкадровування. У показаному нижче прикладі ми реагуємо на факт завантаження елемента **Label** в пам'ять.

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "FontSize">
            <DoubleAnimation From = "12" To = "100" Duration = "0:0:4" RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

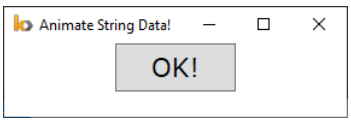
Анімація з використанням дискретних ключових кадрів

На відміну від об'єктів анімації лінійною інтерполяцією, які забезпечують тільки переміщення між початковою і кінцевою точками, об'єкти анімації ключовими кадрами дозволяють створювати колекції спеціальних значень, які повинні досягатися в певні моменти часу.

Щоб проілюструвати використання типу дискретного ключового кадру, припустимо, що потрібно побудувати елемент управління **Button**, який виконує анімацію свого вмісту так, що упродовж трьох секунд з'являється значення **OK!** по одному символу за раз. Представлена далі розмітка знаходиться у файлі **AnimationString.xaml**. Її можна скопіювати в редактор **Kaxaml** і переглянути результати.

Файл **AnimationString.xaml**:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="100" Width="300"
  WindowStartupLocation="CenterScreen" Title="Animate String Data!">
  <StackPanel>
    <Button Name="myButton" Height="40"
         FontSize="16pt" FontFamily="Verdana" Width = "100">
      <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Loaded">
          <BeginStoryboard>
            <Storyboard>
              <StringAnimationUsingKeyFrames RepeatBehavior = "Forever"
                   Storyboard.TargetName="myButton" Storyboard.TargetProperty="Content"
                        Duration="0:0:3">
                   <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
                   <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
                   <DiscreteStringKeyFrame Value="OK" KeyTime="0:0:1.5" />
                   <DiscreteStringKeyFrame Value="OK!" KeyTime="0:0:2" />
              </StringAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Button.Triggers>
    </Button>
  </StackPanel>
</Window>
```



Насамперед зверніть увагу, що для кнопки визначається тригер події, який забезпечує запуск розкадровування при завантаженні кнопки в пам'ять. Клас **StringAnimationUsingKeyFrames** відповідає за зміну вмісту кнопки через значення **Storyboard.TargetProperty**.

Всередині елемента **StringAnimationUsingKeyFrames** визначені чотири елементи **DiscreteStringKeyFrame**, які змінюють властивість **Content** упродовж двох секунд (**StringAnimationUsingKeyFrames** тривалість встановлена об'єктом, яка складає в сумі три секунди, тому між фінальним символом **!** і наступною появою **O** буде помітна невелика пауза).

Тепер, коли ви отримали деяке уявлення про те, як будуються анімації у коді C# і розмітці XAML, давайте з'ясуємо роль стилів WPF, які інтенсивно задіюють графіку, об'єктні ресурси та анімацію.

Резюме

Ми описали API-інтерфейс анімації WPF. У наведених прикладах анімація створювалася за допомогою коду C#, а також за допомогою розмітки XAML. Для управління виконанням анімації, визначеної в розмітці, застосовуються елементи **Storyboard** і *тригери*.

Контрольні питання

1. Для чого використовується анімація в інфраструктурі WPF?
2. Чим відрізняється API-інтерфейс анімації у WPF від API-інтерфейсу анімації у WindowsForms?
3. Згідно якої угоди класи анімації з простору імен System.Windows.Media.Animation зі збірки PresentationCore.dll, який спадкує угоду про іменування дозволяє працювати з анімацією лінійною інтерполяцією?
4. Згідно якої угоди класи анімації з простору імен System.Windows.Media.Animation зі збірки PresentationCore.dll, який спадкує угоду про іменування дозволяє представляє анімацію ключовими кадрами?
5. Згідно якої угоди класи анімації з простору імен System.Windows.Media.Animation зі збірки PresentationCore.dll, який спадкує угоду про іменування представляють анімацію на основі шляху?
6. Яким вимогам повинен відповідати об’єкт для того, щоб під’єднатися до класу Animation?
7. Які ключові властивості у класах Animation управляють початковими і кінцевими значеннями та загальною величиною на яку анімація змінює початкове значення?
8. Який набір додаткових властивостей, які управляють темпом просування анімації пропонує клас System.Windows.Media.Animation.Timeline?
9. Чи можна реалізувати анімацію в розмітці XAML?

Лекція 11. Реалізація стилів WPF

Роль стилів WPF

При побудові інтерфейсу користувача застосунку WPF нерідко вимагається забезпечити загальний вигляд і поведінку для цілого сімейства елементів управління. Наприклад, може знадобитися зробити так, щоб всі типи кнопок мали однакову висоту, ширину, колір і розмір шрифту для свого рядкового вмісту. Хоча розв'язати завдання можна було б встановленням однакових значень в індивідуальних властивостях, такий підхід затрудняє внесення змін, бо при кожній зміні доведеться встановлювати заново один і той же набір властивостей у багатьох об'єктів.

На щастя, інфраструктура WPF пропонує простий спосіб реалізації зовнішнього вигляду і поведінки зв'язаних елементів управління з використанням *стилів*. Кажучи простіше, стиль WPF – це об'єкт, який підтримує колекцію пар "*властивість-значення*". З погляду програмування окремий стиль представляється за допомогою класу **System.Windows.Style**. Клас **Style** має властивість на ім'я **Setters**, яка відкриває доступ до колекції строго типізованих об'єктів **Setter**. Саме об'єкт **Setter** забезпечує можливість визначення пар "*властивість-значення*".

На додаток до колекції **Setter** клас **Style** також визначає декілька інших важливих властивостей, які дозволяють вбудовувати тригери, обмежувати місце застосування стилю і навіть створювати новий стиль на основі існуючого (сприймайте такий прийом як "*спадкування стилів*"). Нижче перераховані властивості класу **Style**:

Таблиця 11.1. Основні властивості класу **Style**

Властивість	Опис
BasedOn	Повертає або задає певний стиль, який є основою поточного стилю, т. т. дозволяє будувати новий стиль на основі існуючого.
Dispatcher	Повертає об'єкт **Dispatcher**, з яким пов'язаний цей об'єкт **DispatcherObject**. (Нащадок **DispatcherObject**)
IsSealed	Повертає значення, яке вказує, чи доступний стиль тільки для читання.
Resources	Повертає або задає колекцію ресурсів, які можуть використовуватися в зоні видимості цього стилю.
Setters	Повертає колекцію об'єктів **Setter** та **EventSetter**.
TargetType	Повертає або задає тип, для якого призначений цей стиль, т. т. відкриває доступ до колекції об'єктів тригерів, яка робить можливою фіксацію умов виникнення різноманітних подій в стилі.
Triggers	Повертає колекцію об'єктів **TriggerBase**, властивостей, які застосовують значення, на основі заданих умов, т. т. відкриває доступ до колекції об'єктів тригерів, яка робить можливою фіксацію умов виникнення різноманітних подій в стилі

Визначення і використання стилю

Майже в кожному випадку об'єкт **Style** пакується як об'єктний ресурс. Подібно до будь-якого об'єктного ресурсу його можна пакувати на рівні вікна

або на рівні застосунку, а також всередині виділеного словника ресурсів (це робить об'єкт **Style** легко доступним у всіх місцях застосунку). Згадайте, що мета полягає у визначенні об'єкта **Style**, який наповнює (мінімум) колекцію **Setters** набором пар "*властивість-значення*".

Давайте побудуємо стиль, який фіксує базові характеристики шрифту елемента управління в нашому застосунку. Розпочнемо зі створення у Visual Studio нового проекту застосунку WPF на ім'я **WpfStyles**. Відкриємо файл **App.xaml** і визначимо наступний іменований стиль:

```
<Application.Resources>
  <Style x:Key ="BasicControlStyle">
    <Setter Property = "Control.FontSize" Value ="14"/>
    <Setter Property = "Control.Height" Value = "40"/>
    <Setter Property = "Control.Cursor" Value = "Hand"/>
  </Style>
</Application.Resources>
```

Зверніть увагу, що об'єкт **BasicControlStyle** додає у внутрішню колекцію три об'єкти **Setter**. Тепер застосуємо отриманий стиль, до декількох елементів управління у головному вікні. Через те, що стиль є *об'єктним ресурсом*, елементи управління, яким він потрібний, як і раніше повинні використати розширення розмітки {**StackResource**} або {**DynamicResource**} для знаходження стилю. Коли вони знаходять стиль, то встановлюють елемент ресурсу в ідентично іменовану властивість **Style**. Замінимо стандартний елемент управління **Grid** наступною розміткою:

```
<StackPanel>
  <Label x:Name="lblInfo" Content=" Цей стиль скучний..."
      Style="{StaticResource BasicControlStyle}" Width="150"/>
  <Button x:Name="btnTestButton" Content=" Так, але ми повторно використовуємо налаштування!"
      Style="{StaticResource BasicControlStyle}" Width="400"/>
</StackPanel>
```

Якщо переглянути елемент **Window** у візуальному конструкторі Visual Studio (або запустити застосунок), то виявиться, що обидва елементи управління підтримують ті ж самі курсор, висоту і розмір шрифту.

Перевизначення налаштувань стилю

Тоді як обидва елементи управління підкоряються стилю, після застосування стилю до елемента управління цілком допустимо змінювати деякі з визначених налаштувань. Наприклад, елемент **Button** тепер використовує курсор **Help** (замість курсора **Hand**, визначеного в стилі):

```
<Button x:Name="btnTestButton" Content="Так, але ми повторно використовуємо налаштування!"
    Cursor="Help" Style="{StaticResource BasicControlStyle}" Width="400"/>
```

Стилі обробляються перед налаштуваннями індивідуальних властивостей елемента управління, до якого застосований стиль; отже, елементи управління можуть «*перевизначати*» налаштування від випадку до випадку.

Вплив атрибуту TargetType на стилі

Зараз наш стиль визначений так, що його може задіяти будь-який елемент управління (і він повинен робити це явно, встановлюючи свою властивість **Style**), бо кожна властивість уточнена за допомогою класу **Control**. Для програми, яка визначає десятки налаштувань, у результаті ми отримали б значний обсяг повторюваного коду. Один із способів дещо поліпшити ситуацію передбачає використання атрибуту **TargetType**. Додавання атрибуту **TargetType** до відкриваючого дескриптора **Style** дозволяє точно вказати, де стиль може бути застосований (у цьому прикладі всередині файла **App.xaml**):

```
<Style x:Key ="BasicControlStyle" TargetType="Control">
  <Setter Property = "Control.FontSize" Value ="14"/>
  <Setter Property = "Control.Height" Value = "40"/>
  <Setter Property = "Control.Cursor" Value = "Hand"/>
</Style>
```

> **На замітку!** При побудові стилю, який використовує базовий клас, немає потреби турбуватися про те, що значення присвоюється властивості залежності, яка не підтримується похідними типами. Якщо похідний тип не підтримує задану властивість залежності, то вона ігнорується.

Прийом незначно допоміг, але все одно ми маємо стиль, який може застосовуватися до будь-якого елемента управління. Атрибут **TargetType** зручніший, коли потрібно визначити стиль, який може бути застосований тільки до окремого типу елементів управління. Додамо в словник ресурсів застосунку наступний стиль:

```
<Style x:Key ="BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value ="20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

Такий стиль працюватиме тільки з елементами управління **Button** (або підкласами **Button**). Якщо застосувати його до несумісного елемента, тоді виникнуть помилки розмітки і компіляції. Додамо елемент управління **Button**, який використовує новий стиль:

```
<Style x:Key ="BigGreenButton" TargetType="Button">
  <Setter Property = "FontSize" Value ="20"/>
  <Setter Property = "Height" Value = "100"/>
  <Setter Property = "Width" Value = "100"/>
  <Setter Property = "Background" Value = "DarkGreen"/>
  <Setter Property = "Foreground" Value = "Yellow"/>
</Style>
```

Результуючий вивід представлений на рис. 11.1.

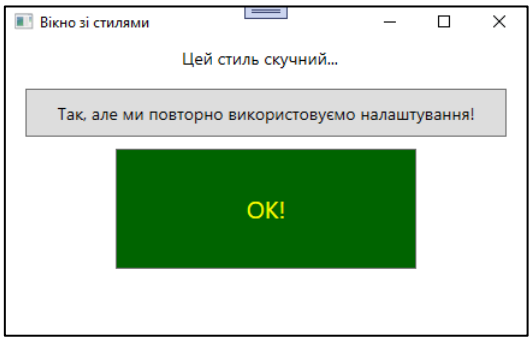


Рис. 11.1. Елементи управління з різними стилями

Ще один ефект від атрибуту **TargetType** полягає в тому, що стиль буде застосований до всіх елементів цього типу всередині області визначення стилю за умови, що властивість **x:Key** *відсутня*.

Ось ще один стиль рівня застосунку, який автоматично застосовуватиметься до всіх елементів управління **TextBox** у поточному застосунку:

```
<!-- Стандартний стиль для всіх текстових полів -->
<Style TargetType="TextBox">
   <Setter Property = "FontSize" Value ="14"/>
   <Setter Property = "Width" Value ="100"/>
   <Setter Property = "Height" Value ="30"/>
   <Setter Property = "BorderThickness" Value = "5"/>
   <Setter Property = "BorderBrush" Value = "Red"/>
   <Setter Property = "FontStyle" Value = "Italic"/>
</Style>
```

Тепер можна визначати будь-яке число елементів управління **TextBox**, і всі вони автоматично отримають встановлений зовнішній вигляд. Якщо якомусь елементу управління **TextBox** не потрібний такий стандартний зовнішній вигляд, тоді він може відмовитися від нього, встановивши властивість **Style** в {**x:Null**}. Наприклад, елемент **txtTest** матиме неіменований стандартний стиль, а елемент **txtTest2** зробить все самостійно:

```
<TextBox x:Name="txtTest"/>
<TextBox x:Name="txtTest2" Style="{x:Null}" BorderBrush="Black"
    BorderThickness="5" Height="60" Width="100" Text="Ha!"/>
```

Створення підкласів існуючих стилів

Нові стилі можна також будувати на основі існуючого стилю за допомогою властивості **BasedOn**. *Розширюваний* стиль повинен мати відповідний атрибут **x:Key** у словнику, бо похідний стиль посилатиметься на нього за іменем, використовуючи розширення розмітки {**StaticResource**} або {**DynamicResource**}. Нижче представлений новий стиль, який базується на стилі **BigGreenButton**, що повертає елемент управління **Button** на **20** градусів (рис. 11.2):

```
<!-- Цей стиль базується на BigGreenButton -->
```

```
<Style x:Key ="TiltButton" TargetType="Button" BasedOn = "{StaticResource BigGreenButton}">
  <Setter Property = "Foreground" Value = "White"/>
  <Setter Property = "RenderTransform">
    <Setter.Value>
      <RotateTransform Angle = "20"/>
    </Setter.Value>
  </Setter>
</Style>
```

Щоб застосувати новий стиль, модифікуємо розмітку для кнопки так:

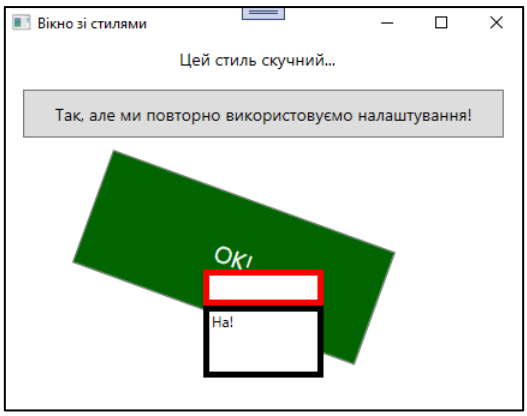


Рис. 11.2. Використання похідного стилю

Визначення стилів з тригерами

Стилі WPF можуть також містити тригери за рахунок пакування об'єктів **Trigger** в колекцію **Triggers** об'єкта **Style**. Використання тригерів в стилі дозволяє визначати деякі елементи **Setter** так, що вони застосовуватимуться тільки у разі істинності заданої умови тригера. Наприклад, можливо вимагається збільшувати розмір шрифту, коли курсор миші знаходиться над кнопкою. Чи, скажімо, треба підсвітити текстове поле з фокусом, з використанням фону заданого кольору. Тригери корисні в ситуаціях подібного роду, тому що вони дозволяють робити специфічні дії при зміні властивості, не вимагаючи написання явної логіки C# у файлі відокремленого коду.

Далі приведена модифікована розмітка для стилю елементів управління типу **TextBox**, де забезпечується встановлення фону жовтого кольору, коли елемент **TextBox** отримує фокус:

```
<!-- Стандартний стиль для всіх текстових полів -->
 <Style TargetType="TextBox">
   <Setter Property = "FontSize" Value ="14"/>
   <Setter Property = "Width" Value = "100"/>
   <Setter Property = "Height" Value = "30"/>
   <Setter Property = "BorderThickness" Value = "5"/>
   <Setter Property = "BorderBrush" Value = "Red"/>
   <Setter Property = "FontStyle" Value = "Italic"/>
```

```
  <!-- Наступне встановлення буде застосоване, тільки
       коли текстове поле знаходиться у фокусу -->
  <Style.Triggers>
    <Trigger Property = "IsFocused" Value = "True">
      <Setter Property = "Background" Value = "Yellow"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

При тестуванні цього стилю виявиться, що при переході за допомогою клавіші <**Tab**> між елементами **TextBox** поточний вибраний **TextBox** отримує фон жовтого кольору (якщо тільки стиль не відключений шляхом присвоєння {**x:Null**} властивості **Style**).

Тригери властивостей також дуже інтелектуальні в тому сенсі, що коли умова тригера не істинна, властивість автоматично набуває стандартного значення. Отже, щойно **TextBox** втрачає фокус, він також автоматично набуває стандартного кольору без зусиль з вашого боку. На противагу, *тригери подій* не повертаються автоматично в попередній стан.

Визначення стилів з множиною тригерів

Тригери можуть бути спроектовані так, що певні елементи **Setter** будуть застосовуватися, коли істинними повинні виявитися багато умов. Нехай потрібно встановлювати фон елемента **TextBox** в **Yellow** тільки у разі, якщо він має *активний фокус* і *курсор миші* знаходиться всередині його меж. Для цього можна скористатися елементом **MultiTrigger** і визначити в ньому кожну умову:

```
<!-- Стандартний стиль для всіх текстових полів -->
<Style TargetType="TextBox">
  <Setter Property = "FontSize" Value ="14"/>
  <Setter Property = "Width" Value ="100"/>
  <Setter Property = "Height" Value ="30"/>
  <Setter Property = "BorderThickness" Value = "5"/>
  <Setter Property = "BorderBrush" Value = "Red"/>
  <Setter Property = "FontStyle" Value = "Italic"/>

  <!-- Наступне встановлення буде застосоване, тільки
       коли текстове поле знаходиться у фокусі -->
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition Property = "IsFocused" Value = "True"/>
        <Condition Property = "IsMouseOver" Value = "True"/>
      </MultiTrigger.Conditions>
      <Setter Property = "Background" Value = "Yellow"/>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

Анімовані стилі

Стилі також можуть містити в собі тригери, які запускають анімаційну послідовність. Нижче показаний останній стиль, який після застосування до елементів управління **Button** змусить їх збільшуватися і зменшуватися в розмірах, коли курсор миші знаходиться всередині меж кнопки:

```
<!-- Стиль кнопки, яка збільшується -->
<Style x:Key = "GrowingButtonStyle" TargetType="Button">
  <Setter Property = "Height" Value = "40"/>
  <Setter Property = "Width" Value = "100"/>
  <Style.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
      <Trigger.EnterActions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "Height">
            <DoubleAnimation From = "40" To = "200"
                  Duration = "0:0:2" AutoReverse="True"/>
          </Storyboard>
        </BeginStoryboard>
      </Trigger.EnterActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

Тут колекція **Triggers** спостерігає за тим, коли властивість **IsMouseOver** поверне значення **true**. Після того, як це станеться, визначається елемент **Trigger.EnterActions** для виконання простого розкадровування, яке примушує кнопку за дві секунди збільшитися до значення **Height**, яке дорівнює **200** (і потім повернутися до значення **Height**, яке дорівнює **40**). Щоб відстежувати інші зміни властивостей, можна також додати область **Trigger.ExitActions** і визначити в ній будь-які спеціальні дії, які мають бути виконані, коли **IsMouseOver** змінюється на **false**.

Застосування стилів в коді

Згадайте, що стиль може застосовуватися також під час виконання. Прийом зручний, коли у кінцевих користувачів має бути можливість вибору зовнішнього вигляду для їх інтерфейсу користувача, якщо вимагається примусово встановлювати зовнішній вигляд і поведінку на основі налаштувань безпеки (наприклад, стиль **DisableAllButton**) або ще в якійсь ситуації.

У поточному проекті було визначено немале число стилів, багато з яких можна застосовувати до елементів управління **Button**. Давайте переробимо інтерфейс користувача головного вікна, щоб дозволити користувачеві вибирати імена наявних стилів в елементі управління **ListBox**. На основі вибраного імені буде застосований відповідний стиль. Ось фінальна розмітка для елемента **DockPanel**:

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Top" Margin="0,0,0, 50">
  <Label Content="Please Pick a Style for this Button" Height="50"/>
  <ListBox x:Name ="lstStyles" Height ="80" Width ="150" Background="LightBlue"
```

```
        SelectionChanged ="comboStyles_Changed" />
    </StackPanel>
    <Button x:Name="btnStyle" Height="40" Width="100" Content="OK!"/>
</StackPanel>
```

Елемент управління **ListBox** (на ім'я **lstStyles**) динамічно заповнюватиметься всередині конструктора вікна:

```
public MainWindow()
{
    InitializeComponent();
    // Заповнити вікно зі списком всіма стилями для елементів Button.
    lstStyles.Items.Add("GrowingButtonStyle");
    lstStyles.Items.Add("TiltButton");
    lstStyles.Items.Add("BigGreenButton");
    lstStyles.Items.Add("BasicControlStyle");
}
```

Останнім завданням є обробка події **SelectionChanged** у пов'язаному файлі коду. Зверніть увагу, що в наступному коді є можливість отримання поточного ресурсу за іменем з використанням успадкованого методу **TryFindResouce()**:

```
private void comboStyles_Changed(object sender, SelectionChangedEventArgs e)
{
    // Отримати ім'я стилю, вибране у вікні зі списком.
    var currStyle = (Style) TryFindResource(lstStyles.SelectedValue);
    if (currStyle == null) return;
    // Встановити стиль для типу кнопки.
    this.btnStyle.Style = currStyle;
}
```

Після запуску застосування з'являється можливість вибору одного з чотирьох стилів кнопок на льоту. На рис. 11.3 показано готовий застосунок у дії.

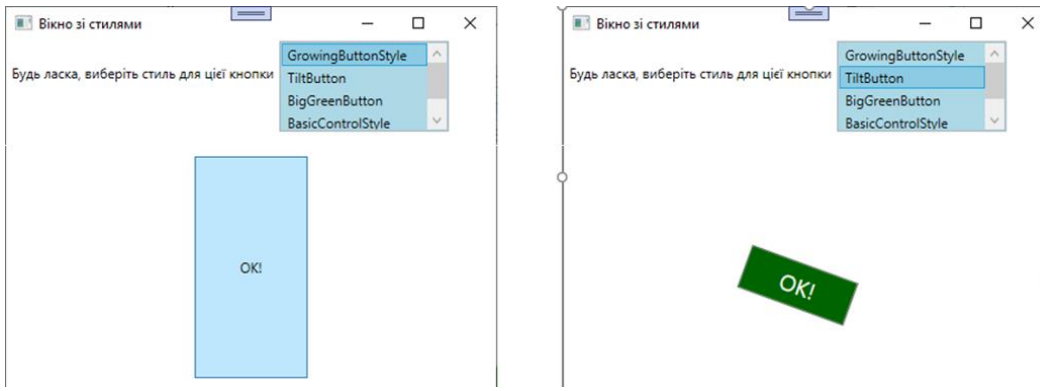


Рис. 11.3. Елементи управління з різними стилями

Резюме

Продемонстрований механізм стилів WPF, який інтенсивно використовує графіку, об'єктні ресурси та анімацію.

Контрольні питання

1. Яку роль в інфраструктурі виконують стилі?
2. Чи підтримує стиль WPF колекцію пар «властивість-значення»?
3. За допомогою якого класу підтримується стиль у WPF?
4. Яка властивість класу Style відкриває доступ до колекції строго типізованих об’єктів Setter?
5. Які ще важливі властивості на додаток до колекції Setter визначені в класі Style?
6. Що дозволяють реалізовувати ці нові члени?
7. Чи можна стверджувати, що стиль це об’єктний ресурс і до нього можна використати риторику стосовно ресурсів з попередньої лекції?
8. Що означає обробка стилю перед налаштуванням індивідуальних властивостей елемента управління? Чи означає це, що елементи управління можуть «перевизначати» налаштування від випадку до випадку?
9. Як атрибут TargetType впливає на стилі?
10. Як зробити так, щоб вплив атрибута TargetType здійснювався тільки в межах області визначення стилю?
11. Як створити підкласи існуючих стилів, використовуючи властивості класу Style описані в табл. 11.1?
12. Як ви розумієте поняття визначення стилів з тригерами?
13. В чому полягає суть «інтелектуальності» тригерів? Наведіть приклад.
14. Як вирозумієте поняття визначення стилів з множиною тригерів? Наведіть приклад.
15. Що таке анімовані стилі?
16. Які можливості щодо інтерфейсу користувача надає використання стилів в коді? Наведіть приклад.

Лекція 12. Логічні і візуальні дерева та шаблони

Логічні дерева, візуальні дерева і стандартні шаблони

Тепер, коли ви розумієте, що таке стилі і ресурси, є ще декілька тем, які потрібно буде розкрити, перш ніж приступати до вивчення побудови спеціальних елементів управління. Зокрема, потрібно з'ясувати різницю між *логічним деревом*, *візуальним деревом* і *стандартним шаблоном*. Використовуючи розмітку XAML у Visual Studio, розмітка є *логічним представленням* документа XAML. У разі написання коду C#, який додає в елемент управління **StackPanel** нові елементи, вони вставляються в логічне дерево. По суті логічне представлення відображає те, як вміст буде позиціонуватися всередині різноманітних диспетчерів компонування для головного елемента **Window** (або іншого кореневого елемента, такого як **Page** або **NavigationWindow**).

Проте за кожним логічним деревом стоїть набагато складніше представлення, яке називається *візуальним деревом* і внутрішньо використовується інфраструктурою WPF для коректної візуалізації елементів на екрані. Всередині будь-якого візуального дерева знаходитимуться всі деталі шаблонів і стилів, використовуваних для візуалізації кожного об'єкта, включаючи всі потрібні *малюнки*, *фігури*, *візуальні об'єкти* та *об'єкти анімації*.

Корисно зрозуміти *різницю* між логічним і візуальним деревами, тому що при побудові спеціального шаблону елемента управління насправді здійснюється заміна всього або частини стандартного візуального дерева елемента управління власним варіантом. Отже, якщо потрібно, щоб елемент управління **Button** візуалізувався у вигляді зіркоподібної фігури, тоді можна визначити новий шаблон такого роду і підключити його до візуального дерева **Button**. Логічно тип залишається тим же типом **Button**, який підтримує всі очікувані властивості, методи і події. Але *візуально він виглядає абсолютно по-іншому*. Один лише згаданий факт робить WPF виключно корисним API-інтерфейсом, бо інші інструментальні набори для створення кнопки зіркоподібної форми зажадали б побудови абсолютно нового класу. В інфраструктурі WPF знадобиться просто визначити нову розмітку.

> **На замітку!** Елементи управління WPF часто описують як *позбавлені зовнішності*. Це відноситься до того факту, що зовнішній вигляд елемента управління WPF абсолютно не залежить від його поведінки і допускає налаштування.

Програмна інспекція логічного дерева

Хоча аналіз логічного дерева вікна під час виконання – не надто поширена дія при програмуванні з використанням WPF, корисно згадати про те, що в просторі імен **System.Windows** визначений клас **LogicalTreeHelper**, який дозволяє інспектувати структуру логічного дерева під час виконання. Для ілюстрації зв'язку між логічними деревами, візуальними деревами і шаблонами

елементів управління створимо новий проект застосунку WPF по імені **TreesAndTemplatesApp**.

Замінимо елемент **Grid** приведеною нижче розміткою, яка містить два елементи управління **Button** і досить великий елемент **TextBox**, який допускає тільки читання з включеними лінійками прокручування. Створимо в IDE-середовищі обробники подій **Click** для кожної кнопки. Ось результуюча розмітка XAML:

```
<DockPanel LastChildFill="True">
  <Border Height="50" DockPanel.Dock="Top" BorderBrush="Blue">
    <StackPanel Orientation="Horizontal">
      <Button x:Name="btnShowLogicalTree" Content="Логічне дерево вікна" Margin="4"
          BorderBrush="Blue" Height="40" Click="btnShowLogicalTree_Click"/>
      <Button x:Name="btnShowVisualTree" Content="Візуальне дерево вікна"
          BorderBrush="Blue" Height="40" Click="btnShowVisualTree_Click"/>
    </StackPanel>
  </Border>
  <TextBox x:Name="txtDisplayArea" Margin="10"
        Background="AliceBlue" IsReadOnly="True"
        BorderBrush="Red" VerticalScrollBarVisibility="Auto"
        HorizontalScrollBarVisibility="Auto" />
</DockPanel>
```

Всередині файла коду C# визначимо *змінну-член* **dataToShow** типу **string**. В обробнику події **Click** об'єкта **btnShowLogicalTree** викличемо допоміжну функцію, яка продовжить викликати себе рекурсивно з метою заповнення рядкової змінної логічним деревом **Window**. Для цього буде викликаний статичний метод **GetChildren()** об'єкта **LogicalTreeHelper**. Нижче показаний потрібний код:

```
private void btnShowLogicalTree_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  BuildLogicalTree(0, this);
  txtDisplayArea.Text = _dataToShow;
}

void BuildLogicalTree(int depth, object obj)
{
  // Додати ім'я типу до змінної-члена _dataToShow.
  _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
  // Якщо елемент - не DependencyObject, тоді пропустити його.
  if ((obj is DependencyObject))
    return;
  // Виконати рекурсивний виклик для кожного логічного дочірнього елемента,
  foreach (var child in LogicalTreeHelper.GetChildren((DependencyObject)obj))
  {
    BuildLogicalTree(depth + 5, child);
  }
}
```

Після запуску застосунку і клацання на кнопці «**Логічне дерево вікна**» у текстовій області відобразиться деревовидне представлення, яке виглядає як майже точна копія початкової розмітки XAML (рис. 12.1).

Програмна інспекція візуального дерева

Візуальне дерево об'єкта **Window** також можна інспектувати під час виконання з використанням класу **VisualTreeHelper** з простору імен **System.Windows.Media**. Далі приведена реалізація обробника події **Click** для другого елемента управління **Button** (**btnShowVisualTree**), яка виконує схожу рекурсивну логіку з метою побудови текстового представлення візуального дерева:

```
private void btnShowVisualTree_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  BuildVisualTree(0, this);
  txtDisplayArea.Text = _dataToShow;
}

void BuildVisualTree(int depth, DependencyObject obj)
{
  // Додати ім'я типу до змінної-члена _dataToShow.
  _dataToShow += new string(' ', depth) + obj.GetType().Name + "\n";
  // Виконати рекурсивний виклик для кожного візуально дочірнього елемента,
  for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
  {
    BuildVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
  }
}
```

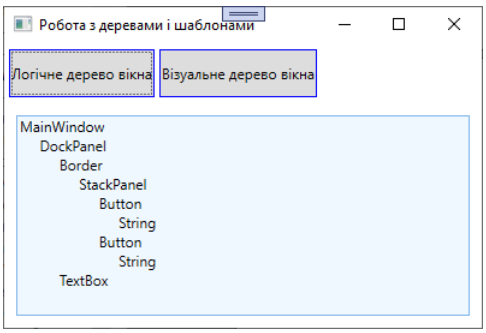


Рис. 12.1. Перегляд логічного дерева під час виконання

На рис. 12.2 видно, що візуальне дерево відкриває доступ до декількох низькорівневих агентів візуалізації, таких як **ContentPresenter**, **AdornerDecorator**, **TextBoxLineDrawingVisual** і т. д.

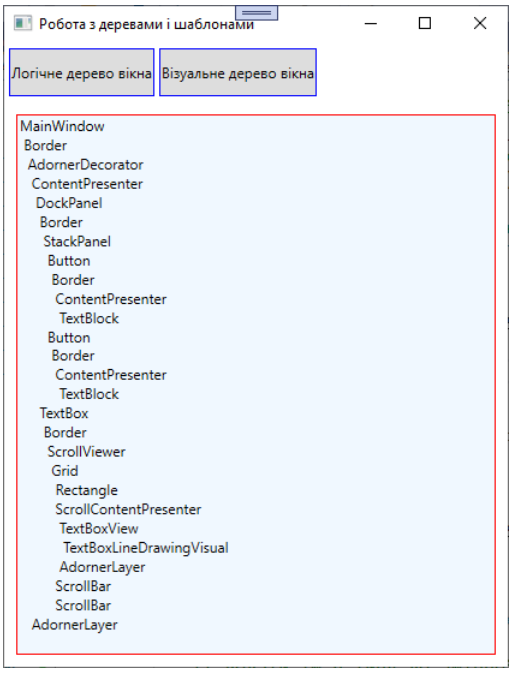


Рис. 10.2. Перегляд візуального дерева під час виконання

Програмне інспектування стандартного шаблону елемента управління

Згадайте, що візуальне дерево використовується інфраструктурою WPF для з'ясування, того як візуалізувати елемент **Window** і всі елементи, які містяться в ньому. Кожен елемент управління WPF зберігає власний набір команд візуалізації всередині свого стандартного шаблону. З погляду програмування будь-який шаблон може бути представлений як екземпляр класу **ControlTemplate**. Крім того, стандартний шаблон елемента управління можна отримати через властивість **Template**:

```
// Отримати стандартний шаблон елемента Button.
Button myBtn = new Button();
ControlTemplate template = myBtn.Template;
```

Аналогічно можна створити в коді новий об'єкт **ControlTemplate** і підключити його до властивості **Template** елемента управління:

```
// Підключити новий шаблон для використання в кнопці.
Button myBtn = new Button();
ControlTemplate customTemplate = new ControlTemplate();

// Припустимо, що цей метод додає увесь код для зіркоподібного шаблону.
MakeStarTemplate(customTemplate);
myBtn.Template = customTemplate;
```

Разом з тим, не зважаючи на те, що новий шаблон можна будувати в коді, набагато частіше це здійснюється в розмітці XAML. Проте, перш ніж

приступити до побудови власних шаблонів, давайте завершимо поточний приклад і додамо можливість перегляду стандартного шаблона для елемента управління WPF під час виконання, що може виявитися по-справжньому корисним способом ознайомлення із загальною структурою шаблона. Додамо в розмітку вікна нову панель **StackPanel** з елементами управління; вона стикована з лівою стороною головної панелі **DockPanel** (знаходиться прямо перед елементом <**TextBox**>) і визначена так:

```
<Border DockPanel.Dock="Left" Margin="10" BorderBrush="DarkGreen" BorderThickness="4"
Width="358">
  <StackPanel>
    <Label Content="Enter Full Name of WPF Control" Width="340" FontWeight="DemiBold" />
    <TextBox x:Name="txtFullName" Width="340" BorderBrush="Green"
        Background="BlanchedAlmond" Height="22" Text="System.Windows.Controls.Button" />
    <Button x:Name="btnTemplate" Content="See Template" BorderBrush="Green"
        Height="40" Width="100" Margin="5" Click="btnTemplate_Click" HorizontalAlignment="Left" />
    <Border BorderBrush="DarkGreen" BorderThickness="2" Height="260"
        Width="301" Margin="10" Background="LightGreen" >
      <StackPanel x:Name="stackTemplatePanel" />
    </Border>
  </StackPanel>
</Border>
```

Додамо порожній обробник події **btnTemplate_Click()**:

```
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{

}
```

Текстова область ліворуч вгорі дозволяє вводити повністю задане ім'я елемента управління WPF, розташованого у збірці **PresentationFramework.dll**. Після того як бібліотека завантажена, екземпляр елемента управління динамічно створюється і відображається у великому квадраті ліворуч внизу. Нарешті, в текстовій області справа відображатиметься стандартний шаблон елемента управління. Додамо в клас C# нову змінну-член типу **Control**:

```
private Control _ctrlToExamine = null;
```

Нижче показаний інший код, який вимагає імпортування просторів імен **System.Reflection**, **System.Xml** і **System.Windows.Markup**:

```
private Control _ctrlToExamine = null;
private void btnTemplate_Click(object sender, RoutedEventArgs e)
{
  _dataToShow = "";
  ShowTemplate();
  txtDisplayArea.Text = _dataToShow;
}

private void ShowTemplate()
{
```

```
    // Видалити елемент, який у нинішній момент знаходиться
    // в області попереднього перегляду
    if (_ctrlToExamine != null)
        stackTemplatePanel.Children.Remove(_ctrlToExamine);
    try
    {
      // Завантажити PresentationFramework і створити екземпляр
      // вказаного елемента управління. Встановити його розміри
      // відображення, а потім додати в порожній контейнер StackPanel.
      Assembly asm = Assembly.Load("PresentationFramework, Version=4.0.0.0," +
                                   "Culture=neutral, PublicKeyToken=31bf3856ad364e35");
      _ctrlToExamine = (Control)asm.CreateInstance(txtFullName.Text);
      _ctrlToExamine.Height = 200;
      _ctrlToExamine.Width = 200;
      _ctrlToExamine.Margin = new Thickness(5);
      stackTemplatePanel.Children.Add(_ctrlToExamine);

      // Визначити налаштування XML для оберігання відступів
      var xmlSettings = new XmlWriterSettings { Indent = true };

      // Створити об'єкт StringBuilder для зберігання розмітки XAML.
      var strBuilder = new StringBuilder();

      // Створити об'єкт XmlWriter на основі наявних налаштувань.
      var xWriter = XmlWriter.Create(strBuilder, xmlSettings);

      // Зберегти розмітку XAML в об'єкті XmlWriter на основі ControlTemplate.
      XamlWriter.Save(_ctrlToExamine.Template, xWriter);

      // Відобразити розмітку XAML в текстовому полі.
      _dataToShow = strBuilder.ToString();
    }
    catch (Exception ex)
    {
      _dataToShow = ex.Message;
    }
}
```

Велика частина роботи пов'язана з відображенням скомпільованого ресурсу BAML на рядок розмітки XAML. На рис. 12.3 демонструється фінальний застосунок в дії на прикладі виводу стандартного шаблону для елемента управління **System.Windows.Controls.DatePicker**. Тут відображається календар, який доступний після клацання на кнопці в правій частині елемента управління.

Тепер ви повинні краще розуміти взаємозв'язок між логічними деревами, візуальними деревами і стандартними шаблонами елементів управління. Далі займемося побудовою спеціальних шаблонів та елементів управління користувача.

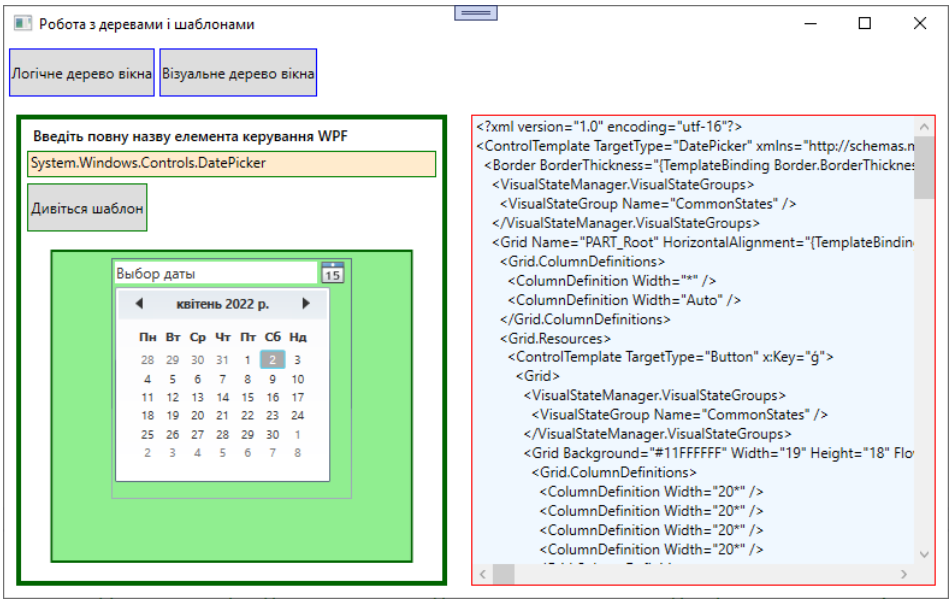


Рис. 12.3. Перегляд стандартного шаблону елемента управління під час виконання

Побудова шаблону елемента управління за допомогою інфраструктури тригерів

Спеціальний шаблон для елемента управління можна створювати з допомогою тільки коду C#. Такий підхід передбачає додавання даних до об'єкта **ControlTemplate** і потім присвоєння його властивості **Template** елемента управління. Проте більшу частину часу зовнішній вигляд і поведінка **ControlTemplate** визначатимуться з використанням розмітки XAML і фрагментів коду (дрібних або великих) для управління поведінкою під час виконання.

Ми навчимося будувати спеціальні шаблони застосунку Visual Studio. Попутно ознайомимося з інфраструктурою тригерів WPF і навчимося використовувати анімацію для вбудовування візуальних підказок кінцевим користувачам. Використання при побудові складних шаблонів тільки IDE-середовища Visual Studio може бути пов'язане з досить великим обсягом клавіатурного набору і важкої роботи.

Спершу створимо новий проект застосунку WPF на ім'я **ButtonTemplate**. Основна цікавинка цього проекту механізми створення і використання шаблонів, так що замінимо елемент **Grid** наступною розміткою:

```
<StackPanel Orientation="Horizontal">
  <Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"/>
</StackPanel>
```

В обробнику події **Click** ми просто відображаємо вікно повідомлення (за допомогою виклику **MessageBox.Show()**) з підтвердженням клацання на елементі управління. При побудові спеціальних шаблонів пам'ятайте, що

поведінка елемента управління незмінна, але його зовнішній вигляд може змінюватися.

Нині цей елемент **Button** візуалізується з використанням стандартного шаблону, який є ресурсом BAML всередині заданої збірки WPF, як було проілюстровано у прикладах з попередніх лекцій. Визначення власного шаблону по суті зводиться до заміни стандартного візуального дерева своїм варіантом. Спершу модифікуємо визначення елемента **Button**, задавши новий шаблон із застосуванням синтаксису "*властивість-елемент*". Шаблон надасть елементу управління округлий вигляд.

```
<Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid x:Name="controlLayout">
        <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
        <Label x:Name="buttonCaption" VerticalAlignment = "Center" HorizontalAlignment = "Center"
            FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
      </Grid>
    </ControlTemplate>
  </Button.Template>
</Button>
```

Тут визначено шаблон, який складається з іменованого елемента **Grid**, що містить іменовані елементи **Ellipse** і **Label**. Через те, що в **Grid** не визначені рядки і стовпці, кожен дочірній елемент поміщається поверх попереднього елемента управління, дозволяючи центрувати вміст. Якщо тепер запустити застосунок, то можна помітити, що подія **Click** ініціюватиметься тільки за ситуації, коли курсор миші знаходиться всередині меж елемента **Ellipse** (тобто не на кутах, які оточують еліпс). Це чудова можливість архітектури шаблонів WPF, бо немає потреби повторно обчислювати попадання курсора, перевіряти граничні умови або здійснювати інші низькорівневі дії. Отже, якщо шаблон використовує об'єкт **Polygon** для відображення якоїсь незвичайної геометрії, тоді можна бути впевненим в тому, що деталі перевірки попадання курсора відповідатимуть формі елемента управління, а не більшого обмежуючого прямокутника.

Шаблони як ресурси

Наразі наш шаблон впроваджений в специфічний елемент управління **Button**, який обмежує можливості його багатократного застосування. В ідеалі шаблон круглої кнопки треба було б помістити в словник ресурсів, щоб його можна було використати в різних проектах, або як мінімум перенести в контейнер ресурсів застосунку для багаторазового використання всередині проекту. Давайте перемістимо локальний ресурс **Button** на рівень застосунку, вирізавши визначення шаблону з розмітки **Button** і вставивши його в дескриптор **Application.Resources** всередині файла **App.xaml**. Додамо атрибути **Key** і **TargetType**:

```
<Application.Resources>
  <ControlTemplate x:Key="RoundButtonTemplate" TargetType="{x:Type Button}">
```

```
    <Grid x:Name="controlLayout">
      <Ellipse x:Name="buttonSurface" Fill = "LightBlue"/>
      <Label x:Name="buttonCaption" VerticalAlignment = "Center" HorizontalAlignment = "Center"
          FontWeight = "Bold" FontSize = "20" Content = "OK!"/>
    </Grid>
  </ControlTemplate>
</Application.Resources>
```

Модифікуємо розмітку для **Button** так:

```
<Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"
    Template="{StaticResource RoundButtonTemplate}">
</Button>
```

Через те, що цей ресурс доступний всьому застосунку, можна визначати будь-яке число круглих кнопок, просто застосовуючи наявний шаблон. В цілях тестування створимо два додаткові елементи управління **Button**, які використовують цей шаблон (обробляти подію **Click** для них не треба):

```
<StackPanel Orientation="Horizontal">
  <Button x:Name="myButton" Width="100" Height="100" Click="myButton_Click"
      Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton2" Width="100" Height="100"
      Template="{StaticResource RoundButtonTemplate}"></Button>
  <Button x:Name="myButton3" Width="100" Height="100"
      Template="{StaticResource RoundButtonTemplate}"></Button>
</StackPanel>
```

Вбудовування візуальних підказок з використанням тригерів

При визначенні спеціального шаблону також *видаляються* всі візуальні підказки стандартного шаблону. Наприклад, стандартний шаблон кнопки містить розмітку, яка задає зовнішній вигляд елемента управління при виникненні певних подій інтерфейсу користувача, таких як отримання фокусу, клацання кнопкою миші, включення (або відключення) і т. д. Користувачі звикли до візуальних підказок подібного роду, бо вони надають елементу управління деяку відчутну реакцію. Проте, в шаблоні **RoundButtonTemplate** розмітка такого типу не визначена і тому зовнішній вигляд елемента управління залишається однаковим незалежно від дій миші. В ідеальному випадку елемент повинен виглядати трохи по-іншому, коли на ньому здійснюється клацання (можливо, за рахунок зміни кольору або відкидання тіні), щоб повідомити користувача про зміну візуального стану.

Завдання можна вирішити із використанням тригерів. Для простих операцій тригери працюють просто прекрасно. Існують додаткові способи досягнення цілі, які виходять за рамки лекційного курсу, але більше інформації доступно за адресами: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/controls/creating-a-control-that-has-a-customizable-appearance?view=netframeworkdesktop-4.8 – стаття **«Створення елемента управління із налаштовуваним зовнішнім виглядом»** та

https://docs.microsoft.com/en-us/dotnet/desktop/wpf/controls/styles-templates-overview?view=netframeworkdesktop-4.8 – стаття «**Стилі і шаблони WPF».**

Як приклад оновимо шаблон **RoundButtonTemplate** розміткою, яка додає два тригери. Перший тригер змінюватиме колір фону кнопок на синій, а колір переднього плану на жовтий (т. т. напис **OK!**), коли курсор знаходиться на поверхні елемента управління. Другий тригер зменшить розміри елемента **Grid** (а також його дочірніх елементів) при натисненні кнопки миші, курсор якої розміщений в межах елемента.

```
<ControlTemplate x:Key="RoundButtonTemplate" TargetType="Button" >
  <Grid x:Name="controlLayout">
    <Ellipse x:Name="buttonSurface" Fill="LightBlue" />
    <Label x:Name="buttonCaption" Content="OK!" FontSize="20" FontWeight="Bold"
        HorizontalAlignment="Center " VerticalAlignment="Center" />
  </Grid>
  <ControlTemplate.Triggers>
    <Trigger Property = "IsMouseOver" Value = "True">
        <Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/>
        <Setter TargetName = "buttonCaption" Property = "Foreground" Value = "Yellow"/>
    </Trigger>
    <Trigger Property = "IsPressed" Value="True">
        <Setter TargetName="controlLayout" Property="RenderTransformOrigin" Value="0.5,0.5"/>
        <Setter TargetName="controlLayout" Property="RenderTransform">
          <Setter.Value>
            <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
          </Setter.Value>
        </Setter>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Роль розширення розмітки {TemplateBinding}

Проблема з шаблоном елемента управління в тому, що кожна кнопка виглядає так само і містить той же текст. Наступне оновлення розмітки ніяк не впливає на попередній результат:

```
<Button x:Name="myButton" Width="100" Height="100" Background="Red" Content="Howdy!"
    Click="myButton_Click" Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton2" Width="100" Height="100" Background="LightGreen" Content="Cancel!"
    Template="{StaticResource RoundButtonTemplate}" />
<Button x:Name="myButton3" Width="100" Height="100" Background="Yellow" Content="Format"
    Template="{StaticResource RoundButtonTemplate}" />
```

Причина в тому, що стандартні властивості елемента управління (такі як **BackGround** і **Content**) перевизначаються в шаблоні. Щоб вони стали доступними, їх потрібно буде відобразити на зв'язані властивості в шаблоні. Вирішити ці проблеми можна за рахунок використання розширення розмітки {**TemplateBinding**} при побудові шаблону. Розширення дозволяє захоплювати налаштування властивостей, які визначені елементом управління, що

використовує шаблон, і використати їх при встановленні значень у самому шаблоні.

Нижче приведена перероблена версія шаблону **RoundButtonTemplate**, в якій розширення розмітки {**TemplateBinding**} застосовується для відображення властивості **Background** елемента **Button** на властивість **Fill** елемента **Ellipse**: тут також забезпечується справжня передача значення **Content** елемента **Button** властивості **Content** елемента **Label**:

```
<Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
<Label x:Name="buttonCaption" Content="{TemplateBinding Content}"
    FontSize="20" FontWeight="Bold" HorizontalAlignment="Center"
        VerticalAlignment="Center" />
```

Рядок розмітки `<!-- Setter TargetName="buttonSurface" Property="Fill" Value="Blue"/ -->` закоментуйте або видаліть.

Після такого оновлення створюються кнопки з різними кольорами і текстом. Результат оновлення розмітки XAML представлений на рис. 12.4.

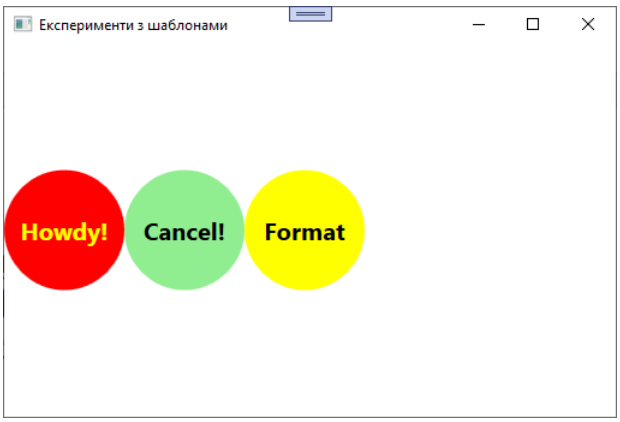


Рис. 12.4. Прив'язки шаблону дозволяють передавати значення внутрішнім елементам управління

Роль класу ContentPresenter

При проектуванні шаблону для відображення текстового значення елемента управління використовувався елемент **Label**. Подібно до **Button** він підтримує властивість **Content**. Отже, якщо використовується розширення розмітки {**TemplateBinding**}, тоді можна визначати елемент **Button** зі складним вмістом, а не тільки з простим рядком.

Але що, якщо потрібно передати складний вміст членові шаблона, який не має властивості **Content**? Коли в шаблоні вимагається визначити узагальнену область відображення вмісту, то замість елемента управління специфічного типу (**Label** або **TextBox**) можна використати клас **ContentPresenter**. Хоча у прикладі, який ми розглядаємо немає потреби, нижче показана проста розмітка, яка ілюструє спосіб побудови спеціального шаблону, який використовує клас **ContentPresenter** для відображення значення властивості **Content** елемента управління, який використовує шаблон:

```
<!-- Цей шаблон кнопки відобразить те, що встановлено
     у властивості Content розміщуваної кнопки -->
<ControlTemplate x:Key="NewRoundButtonTemplate" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}"/>
    <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```

Вбудовування шаблонів в стилі

На цей момент наш шаблон просто визначає базовий зовнішній вигляд і поведінку елемента управління **Button**. Проте, за процес встановлення базових властивостей елемента управління (вмісту, розміру шрифту, ваги шрифту і т. д.) відповідає сам елемент **Button**:

```
<Button x:Name ="myButton" Foreground ="Black" FontSize ="20" FontWeight ="Bold"
    Template ="{StaticResource RoundButtonTemplate}" Click ="myButton_Click"/>
```

За бажання значення базових властивостей можна встановлювати в шаблоні. По суті, у такий спосіб фактично створюються стандартний зовнішній вигляд і поведінка. Як вам вже повинно бути зрозуміло, це робота стилів WPF. Коли будується стиль (для обліку налаштувань базових властивостей), можна визначити шаблон всередині стилю. Нижче показаний змінений ресурс застосунку всередині файла **App.xaml**, якому призначений ключ **RoundButtonSyle**:

```
<!-- Стиль, який містить шаблон -->
<Style x:Key ="RoundButtonStyle" TargetType ="Button">
  <Setter Property ="Foreground" Value ="Black"/>
  <Setter Property ="FontSize" Value ="14"/>
  <Setter Property ="FontWeight" Value ="Bold"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="100"/>

  <!-- Далі йде сам шаблон -->
  <Setter Property ="Template">
    <Setter.Value>
      <ControlTemplate TargetType ="Button">
        <Grid x:Name="controlLayout">
          <Ellipse x:Name="buttonSurface" Fill="{TemplateBinding Background}"/>
          <Label x:Name="buttonCaption" Content="{TemplateBinding Content}" FontSize="20"
            FontWeight="Bold" HorizontalAlignment="Center" VerticalAlignment="Center" />
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property = "IsMouseOver" Value = "True">
            <!-- Setter TargetName = "buttonSurface" Property = "Fill" Value = "Blue"/ -->
            <Setter TargetName = "buttonCaption" Property = "Foreground" Value = "Yellow"/>
          </Trigger>
            <Trigger Property = "IsPressed" Value="True">
              <Setter TargetName="controlLayout" Property="RenderTransformOrigin"
                  Value="0.5,0.5"/>
```

```
            <Setter TargetName="controlLayout" Property="RenderTransform">
              <Setter.Value>
                <ScaleTransform ScaleX="0.8" ScaleY="0.8"/>
              </Setter.Value>
            </Setter>
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Після такого оновлення кнопкові елементи управління можна створювати зі встановленням властивості **Style** так:

```
<Button x:Name="myButton" Background="Red" Content="Howdy"
    Click="myButton_Click" Style="{StaticResource RoundButtonStyle}"/>
```

Незважаючи на те що зовнішній вигляд і поведінка кнопки залишаються такими ж, перевага впровадження шаблонів всередину стилів пов'язана з тим, що з'являється можливість надати готовий набір значень для загальних властивостей. На цьому огляд використання Visual Studio та інфраструктури тригерів при побудові спеціальних шаблонів для елемента управління завершений. Хоча про інфраструктуру WPF можна ще багато чого сказати, тепер у вас є хороший фундамент для подальшого самостійного вивчення.

Резюме

У цій лекції ми з'ясовували відношення між логічним і візуальним деревами. У своїй основі логічне дерево є однозначною відповідністю розмітці, яка створена для опису кореневого елемента WPF. Позаду логічного дерева знаходиться набагато глибше візуальне дерево, яке містить детальні інструкції візуалізації.

Крім того, була вивчена роль стандартного шаблона. Не забувайте, що при побудові спеціальних шаблонів ви по суті замінюєте все візуальне дерево елемента управління (або частину дерева) власною реалізацією.

Контрольні питання

1. Що є логічним представленням документа XAML?
2. Що по суті відображає логічне дерево?
3. Яке представлення називається візуальним деревом?
4. Чим відрізняється візуальне дерево від логічного?
5. Який клас та з якого простору імен дозволяє інспектувати структуру логічного дерева пі час виконання застосунку?
6. Який клас та з якого простору імен дозволяє інспектувати візуальне дерево об’єкта Window?
7. Для чого інфраструктурою WPF використовується візуальне дерево?

8. Який набір команд кожен елемент управління WPF зберігає всередині свого стандартного шаблону?
9. Чи може будь-який шаблон представлений як екземпляр класу ControlTemplate?
10. Використовуючи яку властивість можна отримати стандартний шаблон елемента управління?
11. Де частіше будують новий шаблон: в коді чи в розмітці XAML?
12. Яка схема створення спеціального шаблону управління?
13. Чи можна використати шаблон як ресурс? Якщо так, то яка схема реалізації цього механізму?
14. Як відновити візуальні підказки для спеціального шаблону управління?
15. Яка роль класу ContentPresenter при організації передачі складного вмісту членові шаблону?
16. Як визначити шаблон всередині стилю?

Лекція 13. Побудова бізнес-застосунків за допомогою WPF

Цю лекцію ми присвятимо побудові бізнес-застосунків за допомогою WPF, використовуючи знання отримані при опануванні лекцій **1** – **12**. Крім того, зважаючи на неможливість викладу всіх можливостей технології WPF, ми доповнимо деякі, розглянуті раніше можливості технології новими деталями. Зокрема, розглянемо важливі *аспекти* створення готових застосунків, такі як *прив'язка даних* та *обробка команд*. Прив'язка даних – важлива концепція для виводу даних з класів .NET в інтерфейс користувача і забезпечення користувача можливістю змінювати дані. Зв'язування з командами дозволяє відображати події інтерфейсу користувача на код. На відміну від моделі подій, це забезпечує краще розділення між XAML і кодом.

Розпочнемо розгляд з прив’язки даних.

Прив'язка даних

Прив'язка даних WPF також зробила величезний крок вперед порівняно з колишніми технологіями. Прив'язка даних бере дані з об'єктів .NET і переносить їх в інтерфейс, або навпаки. Прості об'єкти можуть бути прив'язані до елементів інтерфейсу користувача, списків об'єктів, а також самих елементів XAML. У прив'язці даних WPF *ціллю* може бути будь-яка *властивість залежності* елемента WPF, а будь-яка властивість об'єкта CLR може бути *джерелом*. Оскільки елементи WPF реалізовані у вигляді класів .NET, кожен елемент WPF також може бути джерелом. На рис. 13.1 показано з'єднання між джерелом і метою прив'язки[18]. Об'єкт **Binding** визначає з'єднання.

Прив'язка підтримує декілька режимів взаємодії між *метою* і *джерелом*. Прив'язка може бути *однонапрямленою* (**one-way**), коли інформація джерела надходить в ціль, але якщо користувач змінює інформацію в інтерфейсі, то джерело при цьому не оновлюється. Для оновлення джерела потрібна *двонапрямлена* (**two-way**) прив'язка. У табл. 13.1 перераховані режими прив'язки та описані їх вимоги.

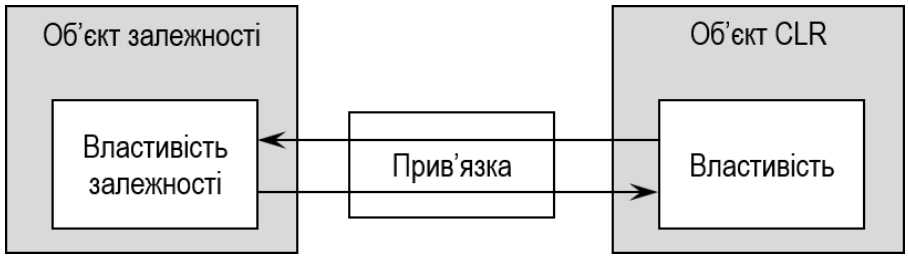


Рис. 13.1. Схема прив'язки

[18] Про моделі прив’язки даних детальніше можна прочитати за адресою – https://docs.microsoft.com/ru-ru/dotnet/desktop/wpf/data/?view=netdesktop-6.0

Таблиця 13.1. Режими прив'язки

Режим прив'язки	Опис
Однократний (one-time)	Прив'язка спрямована від джерела до цілі та активізується тільки при запуску застосунку або зміні контексту даних. У цьому випадку отримуємо знімок даних.
Однонапрямлений (one-way)	Прив'язка спрямована від джерела до цілі. Це зручно у випадку даних тільки для читання, тому що такі дані змінити через інтерфейс користувача неможливо. Щоб отримати оновлення в інтерфейсі користувача, в джерелі має бути реалізований інтерфейс **INotifyPropertyChanged**.
Двонапрямлений (two-way)	При двонапрямленій прив'язці користувач може вносити зміни в дані через інтерфейс користувача. Прив'язка працює в двох напрямах – від джерела до цілі і від цілі до джерела. Щоб зміни можна було здійснювати з інтерфейсу користувача в джерело, в останньому мають бути встановлені властивості для читання/запису.
Однонапрямлений до джерела (one-way-to-source)	При однонапрямленій прив'язці до джерела, якщо міняється цільова властивість, то початковий об'єкт оновлюється.

Розглянемо детальніше прив'язку до:

- елементів XAML;
- простих об'єктів .NET;
- списків.

За допомогою механізму *повідомлення* про зміни інтерфейс користувача оновлюється тими змінами, які сталися у прив'язаних об'єктах. Розглянемо отримання даних від об'єктного постачальника даних і безпосередньо з коду. Множинна і пріоритетна прив'язки демонструють різні можливості прив'язки, які відрізняються від прив'язки за замовчуванням. Опишемо динамічний вибір шаблонів даних і перевірку достовірності прив'язаних значень.

Давайте розпочнемо з прикладу застосунка **BooksDemo**.

Застосунок BooksDemo

У цьому розділі створимо застосунок WPF під назвою **BooksDemo**, який будемо використовувати упродовж розгляду прив'язки даних і зв'язування з командами. Суть завдання полягає у створенні застосунку для обліку літературних джерел з різними функціональними можливостями. Детальніше опишіть створення такого програмного продукту, використовуючи знання і навики почерпнуті з лабораторного практикуму, самостійно.

Змініть XAML-файл **MainWindow.xaml** і додайте в нього елементи управління **DockPanel**, **ListBox**, **Hyperlink** і **TabControl**.

```
<Window x:Class="BooksDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:BooksDemo "
    Title="Main Window" Height="400" Width="600">
  <DockPanel>
    <ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
```

```
        <ListBoxItem>
          <Hyperlink Click="OnShowBook">
            Show Book
          </Hyperlink>
        </ListBoxItem>
      </ListBox>
      <TabControl Margin="5" x:Name="tabControl1">
      </TabControl>
    </DockPanel>
</Window>
```

Тепер додайте елемент управління WPF користувача на ім'я **BookUC**. Це робиться так:

- клацніть правою кнопкою миші у будь-якому місці **Панелі елементів**;
- у контекстному меню, яке з'явилося, виберіть пункт **Вибрати елементи…**;
- у вікні **Вибір елементів панелі елементів** виберіть елемент **UserControl**;
- натисніть кнопку **OK** (див. рис. 13.2).

Після цього на панелі елементів з'явиться елемент управління UserControl, який у панелі елементів, яка завантажується за замовчуванням він відсутній (див. рис. 13.2, скріншот справа).

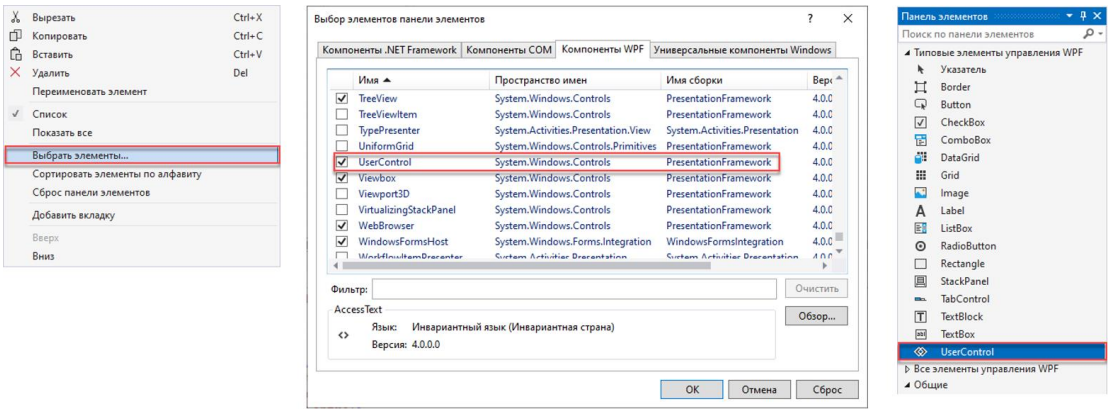


Рис. 13.2. Створення **Елемента управління користувача** – **UserControl**

Елемент управління **UserControl** повинен містити в собі елементи управління **DockPanel**, **Grid** з декількома рядками і стовпцями, а також **Label** і **TextBox**.

```
<UserControl x:Class="BooksDemo.BookUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:BooksDemo"
        mc:Ignorable="d"
```

```
        d:DesignHeight="450" d:DesignWidth="800">
  <DockPanel>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
        HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
        Margin="10,0,5,0" HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Isbn" Grid.Row="2" Grid.Column="0"
        Margin="10,0,5,0" HorizontalAlignment="Left" VerticalAlignment="Center" />
      <TextBox Grid.Row="0" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="1" Grid.Column="1" Margin="5" />
      <TextBox Grid.Row="2" Grid.Column="1" Margin="5" />
      <StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
        <Button Content="Show Book" Margin="5" Click="OnShowBook" />
      </StackPanel>
    </Grid>
  </DockPanel>
</UserControl>
```

Створіть обробник **OnShowBook()** всередині якого в **MainWindow.xaml** створіть екземпляр елемента управління користувача **BookUC** і додайте новий **TabItem** до **TabControl**. Потім змініть властивість **SelectedIndex** елемента **TabControl** для відкриття нової вкладки:

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
  var bookUI = new BookUC();
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header = "Book", Content = bookUI });
}
```

Після збирання проекту можна запустити застосунок і відкрити елемент управління користувача всередині **TabControl**, клацнувши на *гіперпосиланні*. (див. рис. 13.3).

Прив'язка за допомогою XAML

Елемент WPF може бути не лише *ціллю* прив'язки даних, але також і *джерелом*. Властивість-джерело одного елемента WPF можна прив'язати до цілі в іншому елементі WPF.

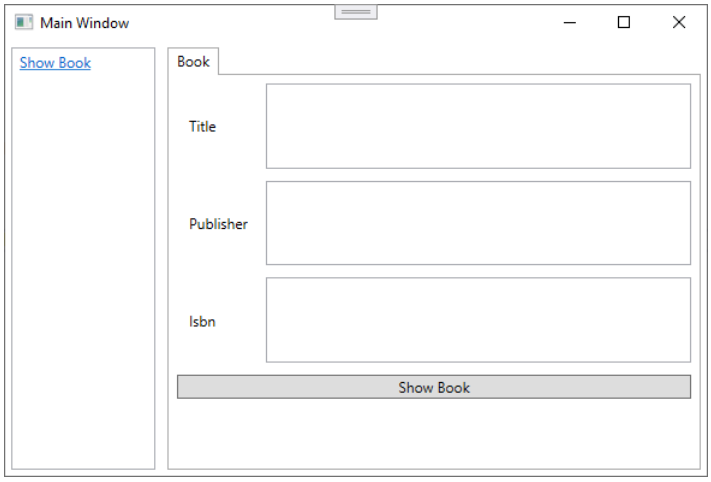


Рис. 13.3. Застосунок з відкритим
елементом управління користувача всередині **TabControl**

У наступному прикладі прив'язка даних (дані про книги) використовується для зміни розміру елементів управління всередині елемента управління користувача за допомогою *повзунка*. Додайте елемент управління **StackPanel**, який містить елементи **Label** і **Slider**, до елемента користувача **BookUC**. В елементі **Slider** визначені значення **Minimum** і **Maximum**, які задають масштаб, і властивості **Value** присвоюється початкове значення **1**.

```
<StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
    HorizontalAlignment="Right">
  <Label Content="Resize" />
  <Slider x:Name="slider1" Value="1" Minimum="0.4" Maximum="3"
    Width="150" HorizontalAlignment="Right" />
</StackPanel>
```

Встановіть властивість **LayoutTransform** елемента управління **Grid** і додайте елемент **ScaleTransform**. В елементі **ScaleTransform** властивості **ScaleX** і **ScaleY** прив'язані до даних. Обидві властивості встановлюються за допомогою розширення розмітки **Binding**. У розширенні розмітки **Binding** властивість **ElementName** встановлюється в **slider1** для посилання на раніше створений елемент управління **Slider**. Властивість **Path** встановлюється значення властивості **Value** для набуття значення від повзунка.

```
<Grid>
  <Grid.LayoutTransform>
    <ScaleTransform x:Name="scale1"
        ScaleX="{Binding Path=Value, ElementName=slider1} "
        ScaleY="{Binding Path=Value, ElementName=slider1} " />
  </Grid.LayoutTransform>
```

Запустивши застосунок, можна переміщати повзунок і так змінювати розміри елементів управління всередині **Grid**, як показано на рис. 13.4 і 13.5.

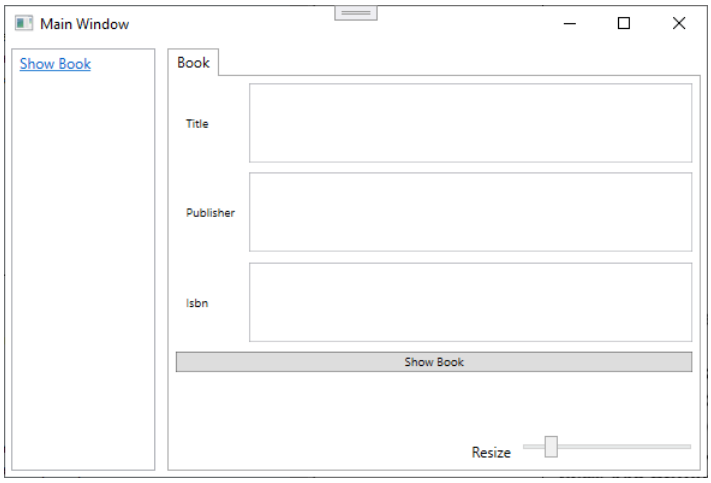


Рис. 13.4. Зменшення розмірів елементів управління всередині **Grid**

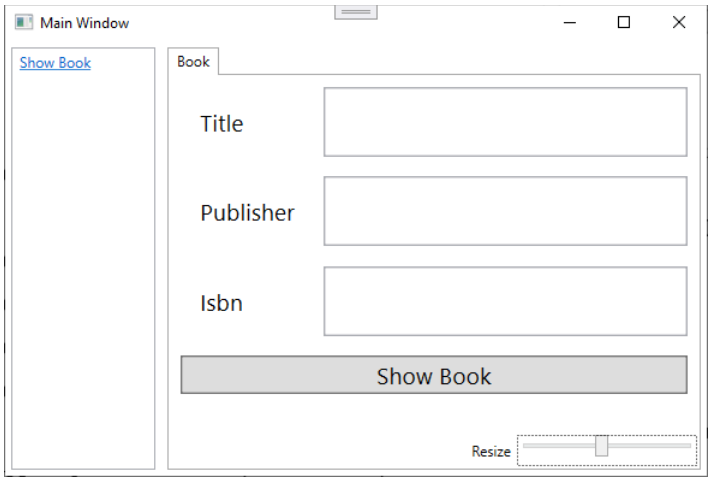


Рис. 13.5. Збільшення розмірів елементів управління всередині **Grid**

Замість визначення інформації прив'язки у XAML за допомогою розширення метаданих **Binding**, як це було зроблено у попередньому коді, це можна зробити у відокремленому коді. Тут потрібно створити новий об'єкт **Binding** і встановити властивості **Path** і **Source**. Властивість **Source** має бути встановлена в об'єкт-джерело; у прикладі це WPF-об'єкт **slider1**. Властивість **Path** встановлюється в екземпляр **PropertyPath**, який ініціалізується ім'ям властивості об'єкта-джерела – **Value**.

З елементами управління, успадкованими від **FrameworkElement**, можна викликати метод **SetBinding()** для визначення прив'язки. Проте **ScaleTransform** спадкує не від елемента **FrameworkElement**, а від базового класу **Freezable**. Для прив'язки таких елементів управління використовується допоміжний клас **BindingOperations**. Метод **SetBinding()** класу **BindingOperations** вимагає об'єкт **DependencyObject**, яким у цьому прикладі є екземпляр **ScaleTransform**. Метод **SetBinding()** також вимагає для прив'язки властивості залежності цілі та об'єкта

Binding. Зауважимо, що всі класи, успадковані від **DependencyObject**, можуть мати властивості залежності. Додайте наступний код до конструктора **public BookUC()**.

```
public BookUC()
{
  InitializeComponent();
  var binding = new Binding
  {
    Path = new PropertyPath("Value"),
    Source = slider1
  };
  BindingOperations.SetBinding(scale1, ScaleTransform.ScaleXProperty, binding);
  BindingOperations.SetBinding(scale1, ScaleTransform.ScaleYProperty, binding);
}
```

У класі **Binding** можна конфігурувати багато опцій прив'язки, які описані в табл. 13.2.

Таблиця 13.2. Члени класу **Binding**

Член	Опис
Source	За допомогою властивості **Source** визначається об'єкт-джерело для прив'язки даних.
RelativeSource	За допомогою властивості **RelativeSource** задається джерело відносно цільового об'єкта. Це зручно для відображення повідомлень про помилки, коли джерело помилки знаходиться в тому ж елементі управління.
ElementName	Якщо джерелом є елемент WPF, то джерело можна задати у властивості **ElementName**.
Path	За допомогою властивості **Path** задається шлях до об'єкта-джерела. Це може бути властивість об'єкта-джерела, але індексатори[19] і властивості дочірніх елементів також підтримуються.
XPath	У разі джерела даних XML для отримання даних прив'язки можна задати вираз запиту **XPath**.
Mode	Режим визначає напрям прив'язки. Властивість **Mode** має тип перерахування **BindingMode**, яка набуває наступних значень: **Default**, **OneTime**, **OneWay**, **TwoWay** та **OneWayToSource**. Режим за замовчуванням залежить від цілі. Для **TextBox** за замовчуванням використовується двонапрямлена прив'язка. **OneTime** означає, що дані тільки спочатку одноразово завантажуються з джерела. **Oneway** означає, що оновлення здійснюється від джерела до цілі. **OneWayToSource** означає, що дані ніколи не читаються, але завжди записуються з цілі в джерело.
Converter	За допомогою властивості **Converter** можна задати клас конвертера, який перетворить дані з інтерфейсу користувача і назад. Клас конвертера повинен реалізувати інтерфейс **IValueConverter**, який визначає методи **Convert()** і **ConvertBack()**. Через властивість **ConverterParameters** методам конвертера можна передавати параметри. Конвертер може бути залежним від культури; культура встановлюється у властивості **ConverterCulture**.
FallbackValue	За допомогою властивості **FallbackValue** можна визначати значення за замовчуванням, використовується, якщо прив'язка не повертає значення.
ValidationRules	За допомогою властивості **ValidationRules** можна задавати колекцію об'єктів

[19] **Індексатор** – це засіб мови C#, який дозволяє індексувати об'єкт, точнісінько так само, як масив за допомогою прямокутних дужок [ ]. За допомогою індексаторів можна реалізовувати власні спеціалізовані масиви, на які можуть накладатися різні обмеження. Детальніше – https://docs.microsoft.com/ru-ru/dotnet/csharp/indexers та https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/indexers/using-indexers.

	ValidationRule, які перевіряються перед тим, як джерело оновлюється з цільового елемента WPF. Клас **ExceptionValidationRule** успадкований від класу **ValidationRule** і перевіряє наявність виключень.

Прив'язка простого об'єкта

Для прив'язки до об'єктів CLR в класі .NET мають бути визначені властивості, як показано в прикладі класу **Book** з властивостями **Title**, **Publisher**, **Isbn** і **Authors**. Цей клас знаходиться в теці **Data** проекту **BooksDemo**.

```
using System.Collections.Generic;

namespace BooksDemo.Data
{
  public class Book
  {
    public Book(string title, string publisher, string isbn,
    params string[] authors)
    {
      this.Title = title;
      this.Publisher = publisher;
      this.Isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book()
      : this("unknown", "unknown", "unknown")
    {
    }
    public string Title { get; set; }
    public string Publisher { get; set; }
    public string Isbn { get; set; }
    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
      get
      {
        return authors.ToArray();
      }
    }
    public override string ToString()
    {
      return Title;
    }
  }
}
```

У коді XAML елемента управління користувача **BookUC** визначено декілька елементів **Label** і **Text Box** для відображення інформації про книги. Використовуючи розширення розмітки **Binding**, елементи управління **TextBox** прив'язуються до властивостей класу **Book**. У розширенні розмітки **Binding** для

прив'язки властивості класу **Book** не визначається нічого окрім властивості **Path**. Немає потреби вказувати джерело, тому що воно визначене присвоєнням **DataContext**, як показано у відокремленому коді, представленому нижче. Режим для елемента **TextBox** визначений за замовчуванням, і це режим двонапрямленої прив'язки.

```
<TextBox Text="{Binding Path=Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
```

У відокремленому коді створюється новий об'єкт **Book**, який присвоюється властивості **DataContext** елемента управління користувача. **DataContext** – властивість залежності, визначена базовим класом **FrameworkElement**. Присвоєння **DataContext** в елементі управління користувача означає, що кожен елемент в елементі користувача має прив'язку за замовчуванням до одного і того ж контексту даних.

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace BooksDemo
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    public MainWindow()
    {
      InitializeComponent();
    }

    private void OnShowBook(object sender, RoutedEventArgs e)
    {
      var bookUI = new BookUC();
      bookUI.DataContext = new Data.Book
        { Title = "Professional C# 2008",
        Publisher = "Wrox Press",
        Isbn = "978-0-470-19137-8" };

      this.tabControl1.SelectedIndex =
        this.tabControl1.Items.Add(
          new TabItem { Header = "Book", Content = bookUI });
    }
  }
}
```

Після запуску застосунку можна побачити прив'язані дані, як показано на рис. 13.6.

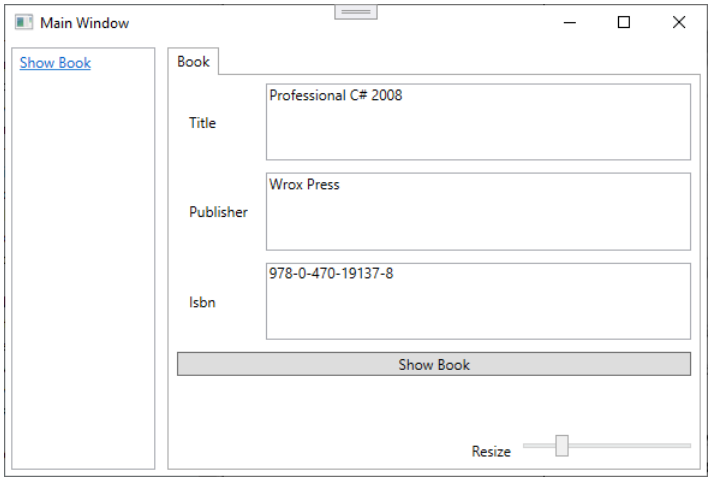


Рис. 13.6. Результат прив'язки даних

Щоб подивитися на двонапрямлену прив'язку у дії (коли зміни, внесені в елементі WPF, відображаються всередині об'єкта CLR), реалізований обробник події **Click** кнопки в елементі управління користувача – метод **OnShowBook()**. Він відкриває вікно повідомлення, в якому відображається поточний заголовок і номер **ISBN** об'єкта **book1** в заголовку вікна. На рис. 13.7 показане вікно повідомлення після внесення змін під час виконання.

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
   Data.Book theBook = this.DataContext as Data.Book;
   if (theBook != null)
      MessageBox.Show(theBook.Title, theBook.Isbn);
}
```

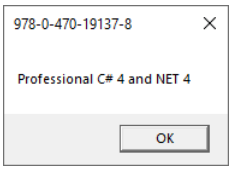


Рис. 13.7. Вікно повідомлення після внесення змін під час виконання

Повідомлення про зміни

При поточній двонапрямленій прив'язці дані зчитуються з об'єкта і записуються назад. Проте якщо всередині коду дані змінюються, інтерфейс користувача не отримує інформації про зміну. У цьому легко переконатися, додавши кнопку до *елемента управління користувача* (**BookUC.xaml.cs**) і реалізувавши обробник події **Click** по імені **OnChangeBook()**. Книга всередині контексту даних зміниться, але інтерфейс користувача не відобразить ці зміни.

```
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
   <Button Content="Show Book" Margin="5" Click="OnShowBook" />
   <Button Content="Change Book" Margin="5" Click="OnChangeBook" />
```

```
</StackPanel>

private void OnChangeBook(object sender, RoutedEventArgs e)
{
  Data.Book theBook = this.DataContext as Data.Book;
  if (theBook != null)
  {
    theBook.Title = "Professional C# 4 with .NET 4";
    theBook.Isbn = "978-0-470-50225-9";
  }
}
```

Щоб передати інформацію про зміну в інтерфейс користувача, в сутнісному класі (**Book**) має бути реалізований інтерфейс **INotifyPropertyChanged**. Клас **Book** модифікований для реалізації цього інтерфейсу. Цей інтерфейс визначає подію **PropertyChanged**, яка також вимагає зміни реалізації властивостей для генерації події.

```
using System.ComponentModel;
using System.Collections.Generic;

namespace BooksDemo.Data
{
  public class Book : INotifyPropertyChanged
  {
    public Book(string title, string publisher, string isbn,
      params string[] authors)
    {
      this.Title = title;
      this.Publisher = publisher;
      this.Isbn = isbn;
      this.authors.AddRange(authors);
    }
    public Book(): this("unknown", "unknown", "unknown")
    {
    }
    public event PropertyChangedEventHandler PropertyChanged;
    private string title;
    public string Title
    {
      get
      {
        return title;
      }

      set
      {
        title = value;
        if (PropertyChanged != null)
```

```
            PropertyChanged(this, new
            PropertyChangedEventArgs("Title"));
        }
    }
    private string publisher;
    public string Publisher
    {
        get
        {
            return publisher;
        }
        set
        {
            publisher = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs("Publisher"));
        }
    }

    private string isbn;
    public string Isbn
    {
        get
        {
            return isbn;
        }
        set
        {
            isbn = value;
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs("Isbn"));
        }
    }

    private readonly List<string> authors = new List<string>();
    public string[] Authors
    {
        get
        {
            return authors.ToArray();
        }
    }
    public override string ToString()
    {
        return this.title;
    }
  }
}
```

Запустіть застосунок знову і переконайтеся, що інтерфейс користувача тепер оновлюється відповідно до зміни даних в обробнику подій (див. рис. 13.8).

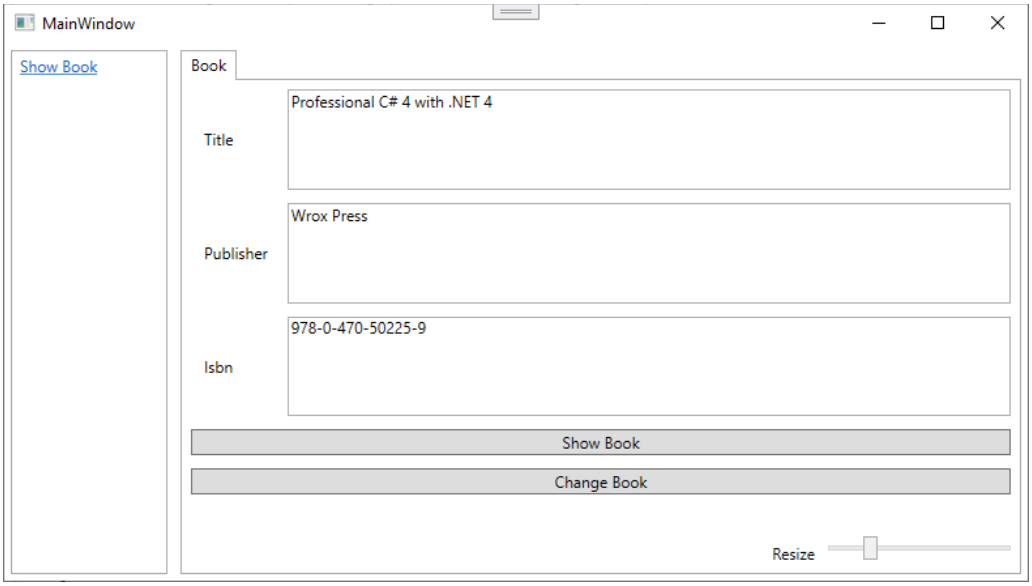


Рис. 13.8. Оновлений інтерфейс користувача відповідно до зміни даних в обробнику подій **OnChangeBook**

Об'єктний постачальник даних

Замість створення екземпляра об'єкта у відокремленому коді це можна зробити у XAML. Щоб послатися на клас відокремленого коду всередині XAML, потрібно буде послатися на простір імен в оголошенні простору імен кореневого елемента XML. Атрибут XML xmlns:local="clr-namespace:BooksDemo" призначає простір імен .NET BooksDemo псевдоніму простору імен XML local.

Один об'єкт класу **Book** тепер визначений в елементі всередині ресурсів **DockPanel**. Присвоєння значень XML-атрибутам **Title**, **Publisher** та **Isbn** спричиняє встановлення значень властивостей з класу **Book**. Конструкція x:Key="theBook" визначає ідентифікатор для ресурсу, так що можна посилатися на об'єкт **book**.

```
<UserControl x:Class="BooksDemo.BookUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:BooksDemo.Data"
        mc:Ignorable="d"
        d:DesignHeight="450" d:DesignWidth="800">
  <DockPanel>
    <DockPanel.Resources>
      <local:Book x:Key="theBook" Title="Professional C# 2010"
        Publisher="Wrox Press" Isbn="978-0-470-50225-9" />
    </DockPanel.Resources>
```

Зауваження: Якщо простір імен, на який треба послатися, знаходиться в іншій збірці, цю збірку треба додати до оголошення XML: xmlns:sys="clr-namespace:System;assembly=mscorlib"

В елементі **TextBox** за допомогою розширення розмітки **Binding** визначено джерело **Source**, яке посилається на ресурс **theBook**:

```
<TextBox Text="{Binding Path=Title, Source={StaticResource theBook}}"
    Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher, Source={StaticResource theBook}}"
    Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn, Source={StaticResource theBook}}"
    Grid.Row="2" Grid.Column="1" Margin="5" />
```

Оскільки всі ці елементи **TextBox** містяться всередині одного і того ж елемента управління, можна присвоїти властивість **DataContext** батьківського елемента управління і встановити властивість **Path** з елементами прив'язки **TextBox**. Через те, що властивість **Path** за замовчуванням, є розширення розмітки **Binding** попередню розмітку можна скоротити до наступного коду:

```
<Grid x:Name="grid1" DataContext="{StaticResource theBook}">
  <!-- ... -->
  <TextBox Text="{Binding Path=Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
  <TextBox Text="{Binding Path=Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
  <TextBox Text="{Binding Path=Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
```

Створимо клас **BookFactory** і помістимо його у папку **Data**. Тепер замість визначення екземпляра об'єкта безпосередньо всередині коду XAML можна визначити *об'єктний постачальник даних*, який посилається на клас, що викликав метод. Для використання **ObjectDataProvider** краще створити *клас-фабрику*, яка повертає об'єкт для відображення, як показано в класі **BookFactory**.

```
using System.Collections.Generic;

namespace BooksDemo.Data
{
  public class BookFactory
  {
    private List<Book> books = new List<Book>();
    public BookFactory()
    {
      books.Add(new Book
      {
        Title = "Professional C# 2010",
        Publisher = "Wrox Press",
        Isbn = "978-0-470-50225-9"
      });
    }
```

```
    public Book GetTheBook()
    {
      return books[0];
    }
  }
}
```

Елемент **ObjectDataProvider** може бути визначений в розділі ресурсів елемента управління користувача **BookUC.xaml**. У XML-атрибуті **ObjectType** вказується ім'я класу, а в **MethodName** – ім'я методу, який викликається для отримання об'єкта **Book**.

```
<DockPanel.Resources>
  <!-- local:Book x:Key="theBook" Title="Professional C# 2010"
    Publisher="Wrox Press" Isbn="978-0-470-50225-9" / -->
  <ObjectDataProvider x:Key="theBook" ObjectType="local:BookFactory" MethodName="GetTheBook"
/>
</DockPanel.Resources>
```

Властивості класу **ObjectDataProvider**, перераховані в табл. 13.3.

Таблиця 13.3. Властивості класу **ObjectDataProvider**

Властивість	Опис
ObjectType	Властивість **ObjectType** визначає тип для створення екземпляра.
ConstructorParameters	Використовуючи колекцію **ConstructorParameters**, до класу можна додавати параметри для створення екземпляра.
MethodName	Властивість **MethodName** визначає ім'я методу, який викликається об'єктним постачальником даних.
MethodParameters	За допомогою властивості **MethodParameters** можна задавати параметри для методу, визначеного властивістю **MethodName**.
ObjectInstance	За допомогою властивості **ObjectInstance** можна отримувати і встановлювати об'єкт, використовуваний класом **ObjectDataProvider**. Наприклад, замість визначення **ObjectType** можна присвоїти існуючий об'єкт програмно, так що екземпляр об'єкта створюється об'єктом **ObjectDataProvider**.
Data	За допомогою властивості **Data** отримується доступ до лежачого в основі об'єкта, використовуваного для прив'язки даних. Якщо визначено властивість **MethodName**, у властивості **Data** можна звернутися до об'єкта, поверненого з певного методу.

Прив'язка списка

Прив'язка до списка виконується *частіше*, ніж прив'язка до простого об'єкта. Прив'язка до списка дуже схожа на прив'язку простого об'єкта. Повний список можна присвоювати об'єкту **DataContext** у відокремленому коді або скористатися **ObjectDataProvider**, який звертається до фабрики об'єктів, що повертає список. До елементів, які підтримують прив'язку до списка (наприклад, **ListBox**), прив'язується повний список. До елементів, які підтримують прив'язку одиночного об'єкта (наприклад, **TextBox**), прив'язується поточний елемент.

Клас **BookFactory** тепер повертає список об'єктів **Book**.

```
public class BookFactory
{
   private List<Book> books = new List<Book>();
   public BookFactory()
   {
      books.Add(new Book("Professional C# 4 with .NET 4", "Wrox Press",
            "978-0-470-50225-9", "Christian Nagel", "Bill Evjen",
            "Jay Glynn", "Karli Watson", "Morgan Skinner"));
      books.Add(new Book("Professional C# 2008", "Wrox Press",
            "978-0-470-19137-8", "Christian Nagel", "Bill Evjen",
            "Jay Glynn", "Karli Watson", "Morgan Skinner"));
      books.Add(new Book("Beginning Visual C# 2010", "Wrox Press",
            "978-0-470-50226-6", "Karli Watson", "Christian Nagel",
            "Jacob Hammer Pedersen", "Jon D. Reid", "Morgan Skinner", "Eric White"));
      books.Add(new Book("Windows 7 Secrets", "Wiley", "978-0-470-50841-1",
            "Paul Thurrott", "Rafael Rivera"));
      books.Add(new Book("C# 2008 for Dummies", "For Dummies",
            "978-0-470-19109-5", "Stephen Randy Davis", "Chuck Sphar"));
   }
   public IEnumerable<Book> GetBooks()
   {
      return books;
   }
   public Book GetTheBook()
   {
      return books[0];
   }
}
```

Щоб використати цей список, створіть новий, *елемент управління користувача* **BooksUC**. Код XAML для цього елемента управління містить елементи **Label** і **TextBox**, які відображають значення однієї книги, та елемент **ListBox**, який відображає список книг. **ObjectDataProvider** викликає метод **GetBooks()** класу **BookFactory**, і цей постачальник використовується для присвоєння **DataContext** об'єкта **DockPanel**. Панель **DockPanel** включає *прив'язані* **ListBox** і **TextBox** як дочірні елементи.

```
<UserControl x:Class="BooksDemo.BooksUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:BooksDemo.Data"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="400">
  <UserControl.Resources>
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
      MethodName="GetBooks" />
  </UserControl.Resources>
  <DockPanel DataContext="{StaticResource books}">
```

```
    <ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
    MinWidth="120" />
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>
      <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
        HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Publisher" Grid.Row="1" Grid.Column="0" Margin="10,0,5,0"
        HorizontalAlignment="Left" VerticalAlignment="Center" />
      <Label Content="Isbn" Grid.Row="2" Grid.Column="0" Margin="10,0,5,0"
        HorizontalAlignment="Left" VerticalAlignment="Center" />
      <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
      <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
      <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
    </Grid>
  </DockPanel>
</UserControl>
```

Новий елемент управління користувача розпочинається з додавання **Hyperlink** у **MainWindow.xaml**. Він використовує обробник події **Click** на ім'я **OnShowBooks()** і реалізацію **OnShowBooks()** у файлі відокремленого коду **MainWindow.xaml.cs**:

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
  <ListBoxItem>
      <Hyperlink Click="OnShowBook">Show Book</Hyperlink>
  </ListBoxItem>
  <ListBoxItem>
      <Hyperlink Click="OnShowBooks">Show Books</Hyperlink>
  </ListBoxItem>
</ListBox>

private void OnShowBooks(object sender, RoutedEventArgs e)
{
  var booksUI = new BooksUC();
  this.tabControl1.SelectedIndex =
    this.tabControl1.Items.Add(
      new TabItem { Header = "Books", Content = booksUI });
}
```

Оскільки **DockPanel** має масив **Book**, присвоєний **DataContext**, і **ListBox** поміщений всередину **DockPanel**, елемент управління **ListBox** відображає всі книги з шаблоном за замовчуванням, як показано на рис. 13.9.

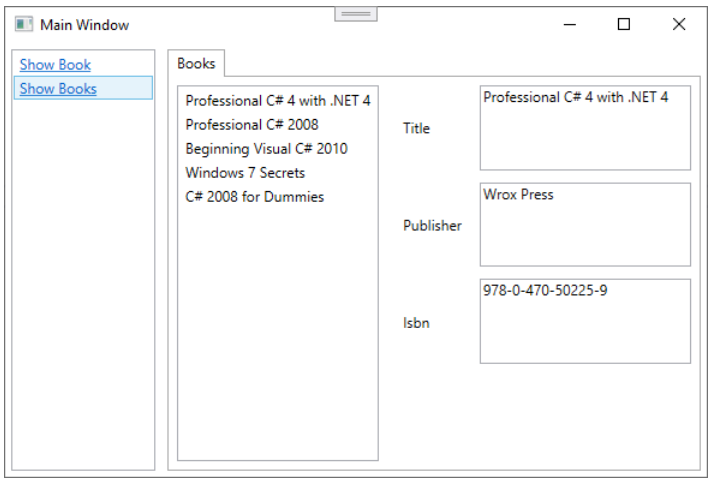


Рис. 13.9. Елемент управління **ListBox** з шаблоном за замовчуванням

Для гнучкішого компонування **ListBox** в *елементі управління користувача* **BooksUC.xaml** знадобиться визначити шаблон щодо стилізації **ListBox** – ***Лекція 11. Реалізація стилів WPF*** та ***Лекція 12. Логічні і візуальні дерева та шаблони***. Властивість **ItemTemplate** об'єкта **ListBox** визначає **DateTemplate** з елементом **Label**. Вміст **Label** прив'язаний до **Title**. Шаблон елемента повторюється для кожного елемента в списку. Зрозуміло, шаблон елемента можна також додати до стилю з ресурсами.

```
<DockPanel DataContext="{StaticResource books}">
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5" MinWidth="120" >
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Прив'язка типу «головна-деталі»

Замість простого відображення всіх елементів всередині списка може знадобитися показати детальну інформацію про вибраний елемент. Це не складно зробити. Елементи управління **Label** і **TextBox** вже визначені, але зараз вони відображають тільки перший елемент в списку.

У визначення **ListBox** потрібно внести одну важливу зміну. За замовчуванням мітки прив'язані тільки до першого елемента в списку. При встановленні властивості **IsSynchronizedWithCurrentItem=True** в **ListBox** вибір в списку встановлюється на поточний елемент. На рис. 13.10 показаний результат; вибраний елемент відображається в елементах **Label** розділу деталей.

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
    MinWidth="120" IsSynchronizedWithCurrentItem="True" >
  <ListBox.ItemTemplate>
```

```
    <DataTemplate>
        <Label Content="{Binding Title}" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

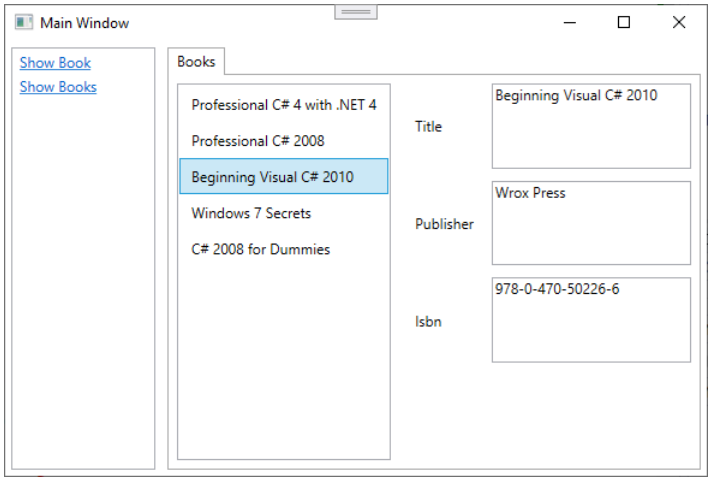


Рис. 13.10. Робота прив'язки «головна-деталі»

Множинна прив'язка

Binding – один з класів, які можуть використовуватися для прив'язки даних. **BindingBase** – абстрактний базовий клас для всіх прив'язок, який має набір конкретних реалізацій. Окрім **Binding** є ще **MultiBinding** і **PriorityBuilding**. Клас **MultiBinding** дозволяє прив'язувати один елемент WPF до багатьох джерел. Наприклад, в класі **Person**, який ми створимо і в якому є властивості **LastName** і **FirstName**, цікаво прив'язати обидві властивості до одного елемента WPF:

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
}
```

Для демонстрації множинних прив’язок створимо проект з іменем **MultiBindingDemo**.

Для **MultiBinding** розширення розмітки не передбачене – ця прив'язка має бути специфікована в синтаксисі елементів XAML. Дочірні елементи **MultiBinding** – це елементи **Binding**, які специфікують прив'язку до різних властивостей. Тут використовуються властивості **FirstName** і **LastName**. Контекст даних встановлюється в елементі **Grid** для посилання на ресурс **person1**.

Щоб з'єднати властивості разом, **MultiBinding** використовує **Converter** для перетворення декількох значень в одне. Цей конвертер використовує параметри, які доступні для різних перетворень на основі параметра.

```
<Window x:Class="MultiBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:MultiBindingDemo"
        Title="MainWindow" Height="240" Width="500">
  <Window.Resources>
    <local:Person x:Key="person1" FirstName="Tom" LastName="Turbo" />
    <local:PersonNameConverter x:Key="personNameConverter" />
  </Window.Resources>
  <Grid DataContext="{StaticResource person1}">
    <TextBox>
      <TextBox.Text>
        <MultiBinding Converter="{StaticResource personNameConverter}"  >
          <MultiBinding.ConverterParameter>
            <system:String>FirstLast</system:String>
          </MultiBinding.ConverterParameter>
          <Binding Path="FirstName" />
          <Binding Path="LastName" />
        </MultiBinding>
      </TextBox.Text>
    </TextBox>
  </Grid>
</Window>
```

Багатозначний конвертер реалізує інтерфейс **IMultiValueConverter**. Цей інтерфейс визначає два методи – **Convert()** і **ConvertBack()**. Метод **Convert()** приймає у першому аргументі декілька значень від *джерела* даних і повертає одне значення *цілі*. У цій реалізації, залежно від того, чи має параметр значення **FirstLast** чи **LastFirst**, результат формується по-різному.

```
using System;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF
{

  public class PersonNameConverter : IMultiValueConverter
  {
    public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)
    {
      switch (parameter as string)
      {
        case "FirstLast":
          return values[0] + " " + values[1];
        case "LastFirst":
          return values[1] + ", " + values[0];
        default:
```

```
            throw new ArgumentException(String.Format("invalid argument {0}", parameter));
        }
    }
    public object[] ConvertBack(object value, Type[] targetTypes, object parameter, CultureInfo culture)
    {
        throw new NotSupportedException();
    }
  }
}
```

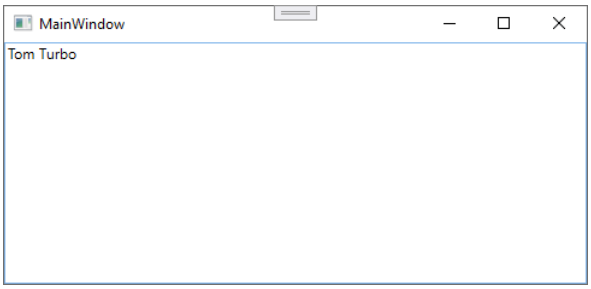


Рис. 13.11. Результат виконання застосунку **MultiBindingDemo**

Пріоритетна прив'язка

Пріоритетна прив'язка **PriorityBinding** дозволяє легко прив'язати дані, які на даний момент не підготовлені. Якщо отримання результату від **PriorityBinding** вимагає часу, можна інформувати користувача про хід виконання, щоб він знав, скільки ще треба чекати. Для ілюстрації пріоритетної прив'язки створимо проект **PriorityBindingDemo** і в ньому створимо клас **Data**. Доступ до його властивості **ProcessSomeData** вимагає деякого часу, і затримка емулюється за допомогою методу **Thread.Sleep()**.

```
using System.Threading;

namespace PriorityBindingDemo
{
  public class Data
  {
    public string ProcessSomeData
    {
      get
      {
        Thread.Sleep(8000);
        return "the final result is here";
      }
    }
  }
}
```

Створимо клас **Information**, який буде надавати інформацію користувачу. Інформація з властивості **Info1** повертається негайно, тоді як **Info2** повертає інформацію через декілька секунд. У реальних реалізаціях цей клас міг

бути асоційований з класом-обробником, щоб отримати передбачуваний час очікування для користувача.

```
using System.Threading;

namespace PriorityBindingDemo
{
  public class Information
  {
    public string Info1
    {
      get
      {
        return "please wait...";
      }
    }

    public string Info2
    {
      get
      {
        Thread.Sleep(5000);
        return "please wait a little more";
      }
    }
  }
}
```

У файлі **MainWindow.xaml** класи **Data** та **Information** ініціюються і посилаються всередині ресурсів **Window**:

```
<Window.Resources>
  <local:Data x:Key="data1" />
  <local:Information x:Key="info" />
</Window.Resources>
```

PriorityBinding замінює звичайну прив'язку всередині властивості **Content** елемента **Label**. Прив'язка **PriorityBinding** складається з декількох елементів **Binding**, кожен з яких, окрім останнього, має властивість **IsAsync**, встановлену в **True**. Якщо результат першого виразу прив'язки недоступний відразу, процес прив'язки переходить до наступного. Перша прив'язка, що посилається на властивість **ProcessSomeData** класу **Data**, вимагає деякого часу. Через це наступна прив'язка вступає в гру і посилається на властивість **Info2** класу **Information**. Властивість **Info2** не повертає результат негайно, і оскільки властивість **IsAsync** встановлена, процес прив'язки не чекає, а переходить до наступної прив'язки. Остання прив'язка використовує властивість **Info1** і негайно повертає результат. Якщо це не відбувається негайно, потрібно буде почекати, бо **IsAsync** встановлена в значення за замовчуванням – **False**.

```
<Label>
  <Label.Content>
```

```
    <PriorityBinding>
        <Binding Path="ProcessSomeData" Source="{StaticResource data1}" IsAsync="True" />
        <Binding Path="Info2" Source="{StaticResource info}" IsAsync="True" />
        <Binding Path="Info1" Source="{StaticResource info}" IsAsync="False" />
    </PriorityBinding>
  </Label.Content>
</Label>
```

При запуску застосунку в інтерфейсі користувача з'являється повідомлення **please wait...** . Після декількох секунд результат з властивості **Info2** повертається як **please wait a little more** (будь-ласка зачекайте ще трішки). Він замінює вивід з **Info1**. І, нарешті, результат від **ProcessSomeData** знову замінює вивід (див. рис. 13.12).

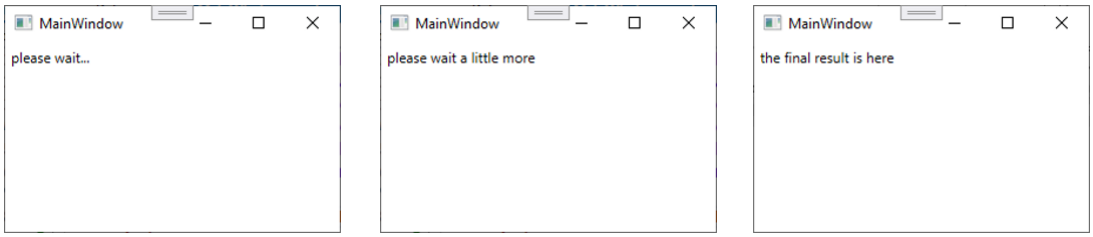


Рис. 13.12. Результат роботи пріоритетної прив'язки

Перетворення значення

Повернемося до застосунку **BooksDemo**. Бачимо, що автори книги відсутні в інтерфейсі користувача. Ідея може бути наступною: якщо прив'язати властивість **Authors** до елемента **Label**, і викликати метод **ToString()** класу **Array**, то він поверне ім'я типу. Одне з рішень цієї проблеми полягає в прив'язці властивості **Authors** до **ListBox**. Для **ListBox** можна задати шаблон для заданого представлення. Інше рішення полягає в перетворенні масиву рядків, повернених властивістю **Authors**, в рядок, і використанні цього рядка для прив'язки.

Виконаємо деякі попередні побудови для того щоб відобразити автора/авторів книги в інтерфейсі користувача. Створимо папку **Utilities** в проекті **BooksDemo**, а в ній створимо клас **StringArrayConverter**. Цей клас використаємо для відображення імен авторів в інтерфейсі користувача.

```
using System;
using System.Diagnostics.Contracts;
using System.Globalization;
using System.Windows.Data;

namespace BooksDemo.Utilities
{
  [ValueConversion(typeof(string[]), typeof(string))]
  class StringArrayConverter : IValueConverter
  {
    public object Convert(object value, Type targetType, object parameter,
      CultureInfo culture)
```

```
    {
      Contract.Requires(value is string[]);
      Contract.Requires(parameter is string);
      string[] stringCollection = (string[])value;
      string separator = (string)parameter;
      return String.Join(separator, stringCollection);
    }
    public object ConvertBack(object value, Type targetType, object parameter,
    CultureInfo culture)
    {
      throw new NotImplementedException();
    }
  }
}
```

Клас **StringArrayConverter** перетворить масив рядків в рядок. Класи конвертерів WPF повинні реалізовувати інтерфейс **IValueConverter** з простору імен **System.Windows.Data**. Цей інтерфейс визначає методи **Convert()** і **ConvertBack()**. У **StringArrayConverter** метод **Convert()** перетворить рядковий масив зі змінної **value** в рядок, використовуючи метод **String.Join()**. Параметр-роздільник **Join()** береться зі змінної **parameter**, отриманої методом **Convert()**.

У коді XAML клас **StringArrayConverter** може бути оголошений як ресурс для посилання з розширення розмітки **Binding**.

```
<UserControl x:Class="BooksDemo.BooksUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:BooksDemo.WPF.Data"
        xmlns:utils="clr-namespace:BooksDemo.Utilities"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="400">
  <UserControl.Resources>
    <utils:StringArrayConverter x:Key="stringArrayConverter" />
    <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
      MethodName="GetBooks" />
  </UserControl.Resources>
<!--...—->
```

Для множинного виводу елемент **TextBlock** оголошений з властивістю **TextWrapping**, встановленою в значення **Wrap**, щоб можна було відображати декількох авторів. У розширенні розмітки **Binding** властивість **Path** встановлена в **Authors**, яка повертає масив рядків. Цей масив рядків перетвориться з ресурсу **StringArrayConverter**, як визначено властивістю **Converter**. Метод **Convert()** реалізації конвертера приймає **', '** для параметра **ConverterParameter**, що і відділятиме прізвища авторів одне від одного.

```
<TextBlock Text="{Binding Authors, Converter={StaticResource stringArrayConverter},
  ConverterParameter=', '}" Grid.Row="3" Grid.Column="1" Margin="5"
```

```
VerticalAlignment="Center" TextWrapping="Wrap" />
```

Як показано на рис. 13.13, вікно тепер відображає детальну інформацію про книгу, включаючи авторів.

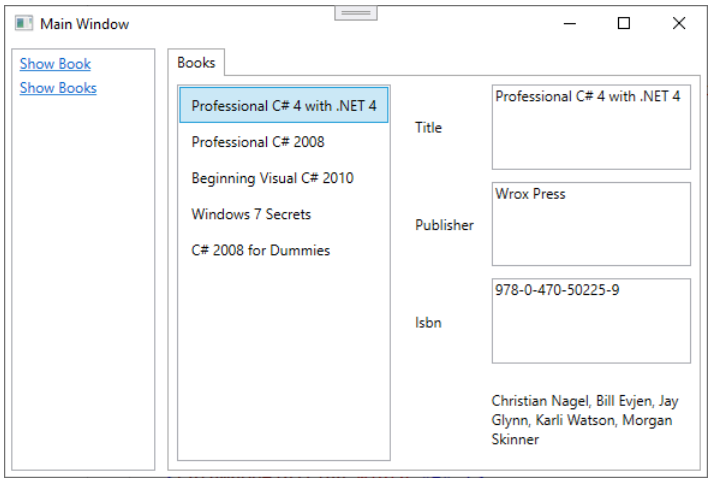


Рис. 13.13. Відображення детальної інформації про книгу

Динамічне додавання елементів списка

А що, якщо елементи списка додаються динамічно? Елемент WPF має бути повідомлений про додавання елементів до списка.

Додамо в код XAML *елемента управління користувача* **BooksUC.xaml** елемент **Button** всередину **StackPanel**. Події **Click** призначимо обробник **OnAddBook()**.

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
        HorizontalAlignment="Center">
  <Button Margin="5" Padding="4" Content="Add Book" Click="OnAddBook" />
</StackPanel>
```

У методі **OnAddBook()** в список додається новий об'єкт **Book**. Якщо запустити застосунок з **BookFactory**, як він реалізований зараз, елементи WPF не отримають повідомлень про додавання нового об'єкта в список:

```
private void OnAddBook(object sender, RoutedEventArgs e)
{
  ((this.FindResource("books") as ObjectDataProvider).Data as
    IList<Book>).Add(
      new Book(".NET 3.5 Wrox Box", "Wrox Press",
        "978-0470-38799-3"));
}
```

Об'єкт, присвоюваний **DataContext**, повинен реалізувати інтерфейс **INotifyCollectionChanged**.

Цей інтерфейс визначає подію **CollectionChanged**, яка використовується застосунком WPF. Замість власноручної реалізації цього інтерфейсу з

використанням спеціального класу колекції можна скористатися узагальненим класом колекції **ObservableCollection<T>**, який визначений в просторі імен **System.Collections.ObjectMode** зі збірки **WindowsBase**. Тепер при додаванні нового елемента в колекцію цей новий елемент негайно відображатиметься в **ListBox**.

```
public class BookFactory
{
  private ObservableCollection<Book> books = new ObservableCollection<Book>();
  // private List<Book> books = new List<Book>();
  public BookFactory()
  {
  …
  }
  public IEnumerable<Book> GetBooks()
  {
    return books;
  }
}
```

Результат додавання нового об’єкта **Book** відображено на рис. 13.13.

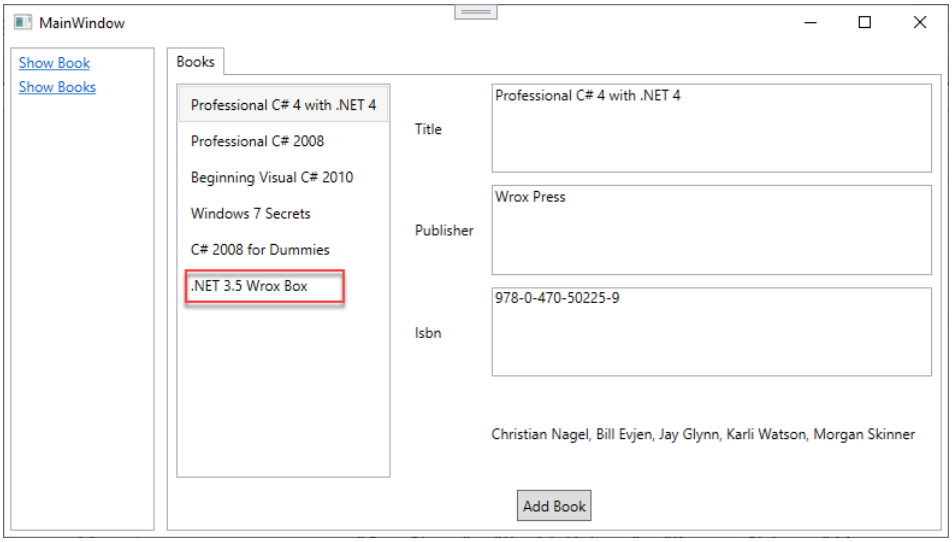


Рис. 13.14. Додавання нової книги до списка

Селектор шаблону даних

В попередніх лекціях – ***Лекція 09. Система ресурсів WPF, Лекція 10. Реалізаця анімації WPF, Лекція 11. Реалізація стилів WPF, Лекція 12. Логічні і візуальні дерева та шаблони*** – було розглянуто, як налаштовувати елементи управління з допомогою шаблонів. Ми ще раз продемонструємо створення шаблону даних, які визначають відображення специфічних типів даних. Селектор шаблону даних може створювати різні шаблони даних динамічно для

одного і того ж типу даних. Селектор шаблону даних реалізований як клас, успадкований від базового класу **DataTemplateSelector**.

Тут селектор шаблонів даних реалізований за рахунок вибору різних шаблонів на основі видавця. Цей шаблон визначений всередині ресурсів елемента управління користувача. Один шаблон може бути доступний згідно ключового імені **wroxTemplate**, інший – за ключовим іменем **dummiesTemplate**, а третій – згідно ключового імені **bookTemplate**.

```
<DataTemplate x:Key="wroxTemplate" DataType="{x:Type local:Book}">
  <Border Background="Red" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="dummiesTemplate" DataType="{x:Type local:Book}">
  <Border Background="Yellow" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>

<DataTemplate x:Key="bookTemplate" DataType="{x:Type local:Book}">
  <Border Background="LightBlue" Margin="10" Padding="10">
    <StackPanel>
      <Label Content="{Binding Title}" />
      <Label Content="{Binding Publisher}" />
    </StackPanel>
  </Border>
</DataTemplate>
```

Для вибору шаблону створимо клас **BookTemplateSelector** клас **BookDataTemplateSelector** перевизначає метод **SelectTemplate** з базового класу **DataTempleteSelector**. Ця реалізація вибирає шаблон на основі властивості **Publisher** класу **Book**.

```
using System.Windows;
using System.Windows.Controls;
using BooksDemo.Data;

namespace BooksDemo.Utilities
{
  public class BookTemplateSelector : DataTemplateSelector
  {
    public override System.Windows.DataTemplate
```

```
        SelectTemplate(object item, System.Windows.DependencyObject container)
    {
        if (item != null && item is Book)
        {
            var book = item as Book;
            switch (book.Publisher)
            {
                case "Wrox Press":
                    return (container as FrameworkElement).FindResource("wroxTemplate") as DataTemplate;
                case "For Dummies":
                    return (container as FrameworkElement).FindResource("dummiesTemplate")
                        as DataTemplate;
                default:
                    return (container as FrameworkElement).FindResource("bookTemplate") as DataTemplate;
            }
        }
        return null;
    }
  }
}
```

Для доступу до класу **BookDataTemplateSelector** з коду XAML цей клас визначений всередині ресурсу **Window**:

```
<utils:BookTemplateSelector x:Key="bookTemplateSelector" />
```

Тепер клас селектора може бути призначений властивості **ItemTemplateSelector** класу **ListBox**:

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
  MinWidth="120" IsSynchronizedWithCurrentItem="True"
    ItemTemplateSelector="{StaticResource bookTemplateSelector}">
```

Запустивши застосунок, можна бачити, що для кожного видавця застосовується свій шаблон даних (рис. 13.15).

Прив'язка до XML

Створимо застосунок **XmlBindingDemo** для демонстрації підтримки для даних XML.

Прив'язка даних WPF має спеціальну підтримку для даних XML. Як джерела даних можна використати **XmlDataProvider** і прив'язувати елементи за допомогою виразів **XPath**. Для ієрархічного відображення можна застосовувати елемент управління **TreeView** і створювати представлення для елементів, використовуючи шаблон **HierarchicalDataTemplate**. Наступний файл XML, який містить елементи **Book**, використовується як джерело в прикладах, що розглядаються нижче.

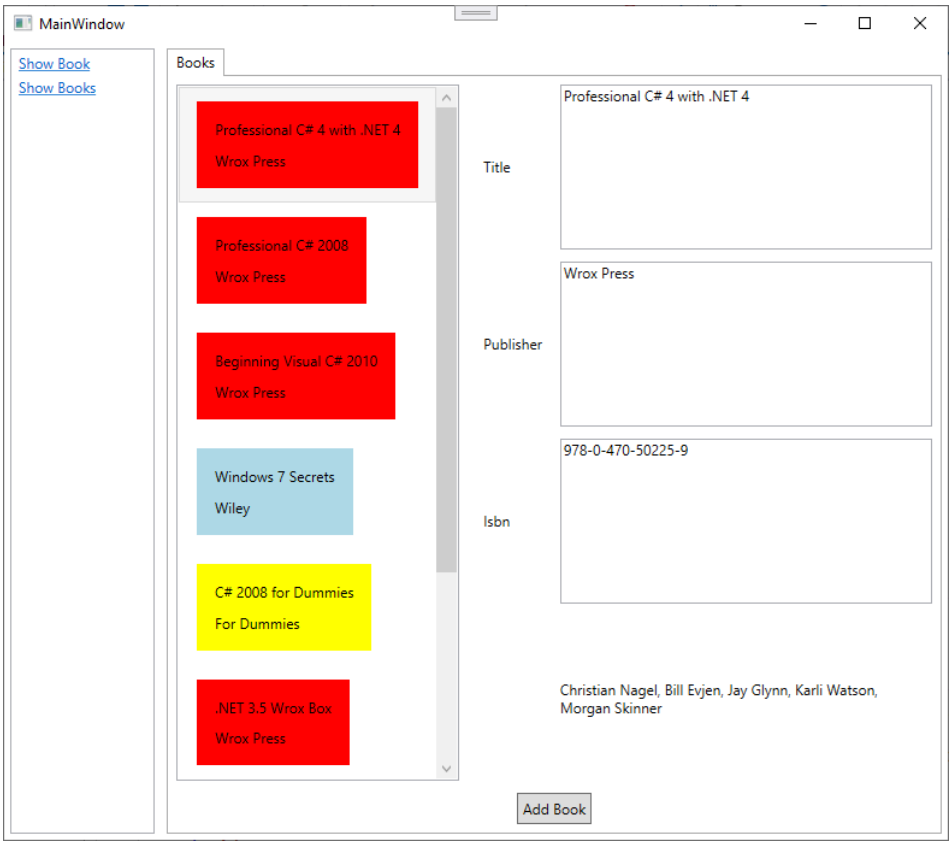


Рис. 13.15. Селектор шаблону даних у дії

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-0-470-12472-7">
    <Title>Professional C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Bill Evjen</Author>
    <Author>Jay Glynn</Author>
    <Author>Karli Watson</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-7645-4382-1">
    <Title>Beginning Visual C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>David Espinosa</Author>
    <Author>Zach Greenvoss</Author>
    <Author>Jacob Hammer Pedersen</Author>
    <Author>Christian Nagel</Author>
    <Author>John D. Reid</Author>
    <Author>Matthew Reynolds</Author>
    <Author>Morgan Skinner</Author>
```

```
    <Author>Eric White</Author>
  </Book>
</Books>
```

Аналогічно визначенню об'єктного постачальника даних, можна визначити постачальника даних XML. Як **ObjectDataProvider**, так і **XmlDataProvider** спадкують від одного базового класу – **DataSourceProvider**.

У **XmlDataProvider** із цього прикладу властивість **Source** встановлюється для посилання на XML-файл **books.xml**. Властивість **XPath** визначається як вираз **XPath** для посилання на кореневий елемент XML на ім'я **Books**. Елемент **Grid** посилається на джерело даних XML за допомогою властивості **DataContext**. Що стосується контексту даних для **Grid**, то тут всі елементи **Book** потрібні для прив'язки списка, тому вираз **XPath** встановлено в **Book**. Всередині **Grid** можна знайти елемент **ListBox**, який прив'язується до контексту даних за замовчуванням і використовує **DataTemplate** для включення заголовка в елементи **TextBlock**, які є елементами **ListBox**. Всередині **Grid** можна також побачити три елементи **Label** з даними, прив'язаними до виразів **XPath** для відображення заголовка, видавця і номера ISBN.

```
<Window x:Class="XmlBindingDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Main Window" Height="240" Width="500">
  <Window.Resources>
    <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
    <DataTemplate x:Key="listTemplate">
      <TextBlock Text="{Binding XPath=Title}" />
    </DataTemplate>
    <Style x:Key="labelStyle" TargetType="{x:Type Label}">
      <Setter Property="Width" Value="190" />
      <Setter Property="Height" Value="40" />
      <Setter Property="Margin" Value="5" />
    </Style>
  </Window.Resources>
  <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <ListBox IsSynchronizedWithCurrentItem="True" Margin="5"
      Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
      ItemTemplate="{StaticResource listTemplate}" />
    <Label Style="{StaticResource labelStyle}"
      Content="{Binding XPath=Title}"
```

```
      Grid.Row="0" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}"
      Content="{Binding XPath=Publisher}"
      Grid.Row="1" Grid.Column="1" />
    <Label Style="{StaticResource labelStyle}"
      Content="{Binding XPath=@isbn}"
      Grid.Row="2" Grid.Column="1" />
  </Grid>
</Window>
```

Результат прив'язки XML показаний на рис. 13.16.

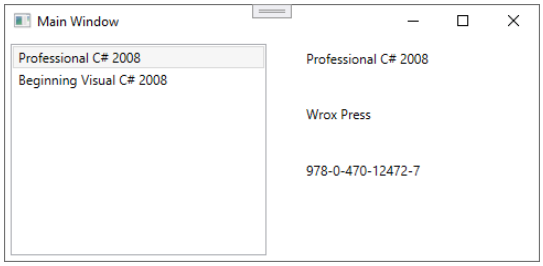


Рис. 13.16. Результат прив'язки XML

Для ієрархічного відображення даних XML можна використати елемент управління **TreeView**.

Перевірка достовірності прив'язки

Для перевірки достовірності даних, введених користувачем, яка здійснюється до їх поміщення в об'єкти .NET, передбачені декілька варіантів:

- обробка виключень;
- інформація про помилки в даних;
- правила перевірки достовірності користувача.

Обробка виключень

Давайте створимо застосунок з іменем **ValidationDemo** для демонстрації перевірки достовірності прив’язки, зокрема, обробки виключень.

Створимо макет проекту, який складатиметься з мітки **Label** в якій буде відображатися напис **Value1**. Для вводу значення помістимо елемент управління **TextBox** в якому буде відображатися введене значення. Також помістимо кнопку з написом **Show Value** при натисненні на яку повинне з’явитися вікно повідомлень з інформацією про правильність введеного значення.

Створимо макет проекту згідно опису вище (див. рис. 13.17).

```
<Window x:Class="ValidationDemo.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:ValidationDemo"
```

```
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
    <TextBox Margin="5" Grid.Row="0" Grid.Column="1" />
  </Grid>
</Window>
```

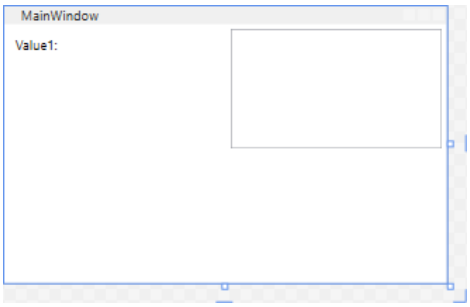


Рис. 13.17. Макет вікна застосунку **ValidationDemo**

Одна з опцій, які ми продемонструємо тут, полягає в тому, що клас .NET генерує виключення, якщо отримано неправильне значення, як показано в класі **SomeData**. Властивість **Value1** набуває значень більших або рівних **5** і менших **12**. Створимо клас **SomeData** у складі проекту **ValidationDemo**.

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

namespace ValidationDemo
{
  public class SomeData
  {
    private int value1;
    public int Value1
    {
      get
      {
        return value1;
      }
```

```
        set
        {
          if (value < 5 || value > 12)
            throw new ArgumentException(
            "Вік не повинен бути менше 0 або більше 80");
          value1 = value;
        }
      }
    }
}
```

У конструкторі класу **MainWindow** новий об'єкт класу **SomeData** ініціалізується і передається властивості **DataContext** для прив'язки:

```
using System;
using System.Windows;

namespace ValidationDemo
{
  /// <summary>
  /// Логика взаимодействия для MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
    private SomeData p1 = new SomeData { Value1 = 11 };
    public MainWindow()
    {
      InitializeComponent();
      this.DataContext = p1;
    }
  }
}
```

Прив'яжемо дані властивості **Text** елемента **TextBox** до властивості **Value1**. Також додамо до проекту елемент управління **Button** з іменем **Show Value**. Для цього потрібно в код розмітки внести зміни:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1" Text="{Binding Path=Value1}" />
<Button Grid.Row="1" Grid.ColumnSpan="2" Content="Show Value" Margin="5"
Click="OnShowValue" />
```

Обробник події – метод **OnShowValue** відображає у вікні повідомлення про справжнє значення екземпляра **SomeData**:

```
private void OnShowValue(object sender, RoutedEventArgs e)
{
  MessageBox.Show(p1.Value1.ToString());
}
```

Якщо ви запустите застосунок зараз і спробуєте змінити значення (**11**) на неправильне, то виявите, що значення не зміниться при натисненні кнопки **Show**

Value. WPF перехоплює та ігнорує виключення, згенероване засобом доступу **set** властивості **Value1** (див. рис. 13.18):

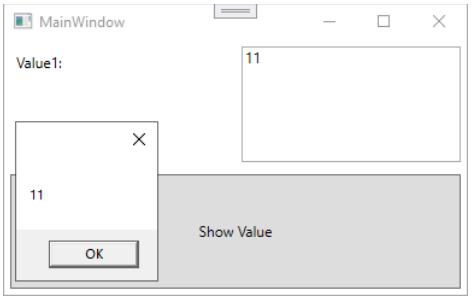


Рис. 13.18. Результат запуску застосунку **ValidationDemo** зі значенням **Value1=11**

Все нормально – значення **11** не виходить за межі діапазону можливих значень **Value1** (**5 ≤ Value1 ≤ 12**). А тепер змінимо значення на таке, що виходить за межі діапазону, наприклад **15**. Ми розуміємо, що повинне згенеруватися системне виключення (див. рис.13.19), яке попереджує нас про це.

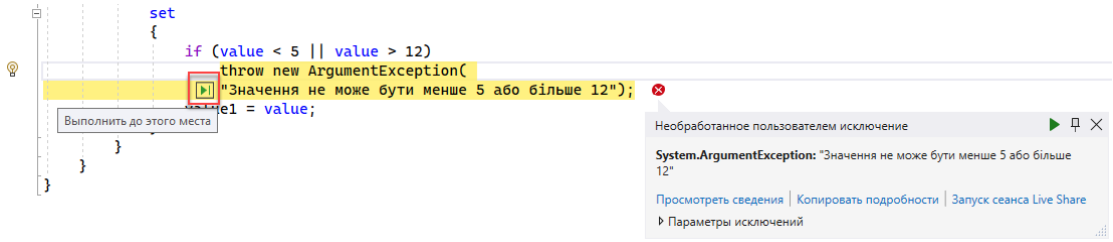


Рис. 13.19. Необроблене користувачем виключення

При натисненні на маркер обведений червоною рамкою ми побачимо у полі вводу **15**, але значення виведене у вікні повідомлення залишиться попереднім, т. т. **11**. (див. рис. 13.20)

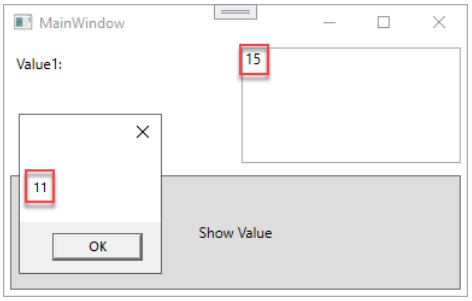


Рис. 13.20. Демонстрація незмінності значення з допустимого діапазону

Для відображення візуального повідомлення про помилку після зміни контексту поля вводу можна встановити в **True** властивість **ValidatesOnException** розширення розмітки **Binding**. При неправильному значенні, так само як і у попередньому випадку, згенерується виключення і **TextBox** оточується червоною рамкою (рис. 13.21).

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1" Text="{Binding Path=Value1,
ValidatesOnExceptions=True}" />
<Button Grid.Row="1" Grid.ColumnSpan="2" Content="Show Value" Click="OnShowValue" Margin="5"
/>
```

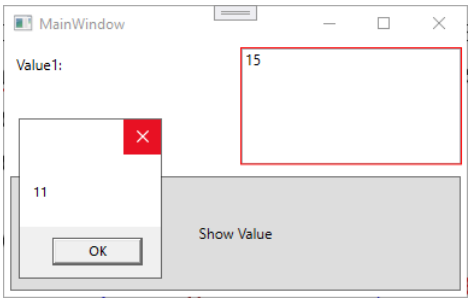


Рис. 13.21. Реакція на неправильне значення введене користувачем

Щоб іншим способом повернути інформацію про помилку користувачеві, можна присвоїти приєднаній властивості **ErrorTemplate**, визначеній класом **Validation**, шаблон, який визначає інтерфейс користувача для представлення помилок. Новий шаблон для позначки помилок показаний тут з ключем **validationTemplate**. Шаблон **ControlTemplate** поміщає червоний знак оклику перед існуючим вмістом елемента управління. Додамо наступний код до **MainWindow.xaml**.

```
<Window.Resources>
  <ControlTemplate x:Key="validationTemplate">
    <DockPanel>
      <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
      <AdornedElementPlaceholder/>
    </DockPanel>
  </ControlTemplate>
</Window.Resources>
```

Встановлення **validationTemplate** з приєднаною властивістю **Validation.ErrorTemplate** активізує шаблон з **TextBox**:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
    Validation.ErrorTemplate="{StaticResource validationTemplate}" />
<Button Grid.Row="1" Grid.ColumnSpan="2" Content="Show Value" Click="OnShowValue"
    Margin="5" />
```

Новий вигляд застосунку показаний на рис. 13.21.

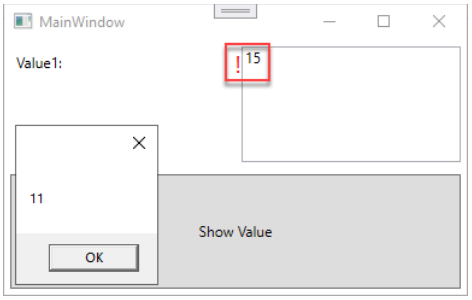


Рис. 13.21. Інша реакція на неправильне значення введене користувачем

Власне інформація про помилку може бути доступна через колекцію **Errors** класу **Validation**. Для відображення інформації про помилку в **ToolTip** (спливаюча підказка) елемента **TextBox** можна створити властивість **Trigger**, як показано нижче. Тригер активізується, щойно властивість **HasError** класу **Validation** встановлюється в **True**. Тригер же і встановлює властивість **ToolTip** елемента **TextBox**.

```
<Application x:Class="ValidationDemo.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:ValidationDemo"
             StartupUri="MainWindow.xaml">
  <Application.Resources>
    <Style TargetType="{x:Type TextBox}">
      <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="True">
          <Setter Property="ToolTip"
            Value="{Binding RelativeSource= {x:Static RelativeSource.Self},
              Path=(Validation.Errors)[0].ErrorContent}" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Application.Resources>
</Application>
```

Інформація про помилку в даних

Інший спосіб роботи з помилками полягає в реалізації об'єктом .NET інтерфейсу **IDataErrorInfo**.

Змінимо клас **SomeData** для реалізації інтерфейсу **IDataErrorInfo**. Цей інтерфейс визначає властивість **Error** та індексатор з рядковим аргументом. При перевірці достовірності WPF під час прив'язки даних викликається індексатор, а ім'я властивості, яка перевіряється, передається у вигляді аргумента **columnName**. Ця реалізація верифікує значення, якщо воно правильне, а інакше

передає рядок помилки. Нижче перевірка достовірності виконується на властивості **Value2**.

```
public class SomeData : IDataErrorInfo
{
  private int value1;
  public int Value1
  {
    get
    {
      return value1;
    }
    set
    {
      if (value < 5 || value > 12)
        throw new ArgumentException(
          "Значення не може бути менше 5 або більше 12");
    value1 = value;
    }
  }
  public int Value2 { get; set; }
  string IDataErrorInfo.Error
  {
    get
    {
      return null;
    }
  }

  string IDataErrorInfo.this[string columnName]
  {
    get
    {
      if (columnName == "Value2")
      {
        if (this.Value2 < 0 || this.Value2 > 80)
          return "Вік не має бути менше 0 або більший 80";
    }
    return null;
    }
  }
}
```

Зауваження: Маючи сутнісний клас .NET, не зовсім зрозуміло, що повинен повертати індексатор; наприклад, чого треба чекати від об'єкта типу **Person**, який викликає індексатор? Ось чому краще передбачати явну реалізацію інтерфейсу **IDataErrorInfo**. У результаті індексатор може бути доступний тільки за рахунок використання інтерфейсу, і клас .NET може мати іншу реалізацію для інших цілей.

Якщо встановити властивість **ValidatesOnDataErrors** класу **Binding** в **True**, інтерфейс **IDataErrorInfo** використовуватиметься під час прив'язки. Тут при зміні **TextBox** механізм прив'язки викликає індексатор інтерфейсу і передає **Value2** змінній **columnName**.

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value2:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />
<Button Grid.Row="1" Grid.ColumnSpan="2" Content="Show Value" Click="OnShowValue" Margin="5"
/>
```

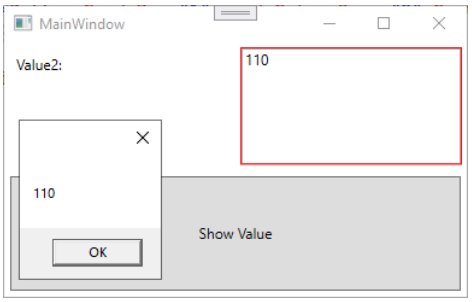


Рис. 13.22. Червона рамка навколо **TextBox** сигналізує про допущену помилку

Спеціальні правила перевірки достовірності

Щоб отримати повніший контроль над перевіркою достовірності даних, треба реалізувати *спеціальне правило перевірки достовірності*. Клас, який реалізовує спеціальне правило перевірки (**special authentication rules**) достовірності, має бути успадкований від базового класу **ValidationRule**. У попередніх двох прикладах також використовувалися правила перевірки достовірності. Два класи були успадковані від абстрактного базового класу **ValidationRule** – це **DataErrorValidationRule** та **ExceptionValidationRule**. Клас **DataErrorValidationRule** активізується встановленням властивостей **ValidatesOnDataErrors** і використовує інтерфейс **IDataErrorInfo**; клас **ExceptionValidationRule** має справу з виключеннями та активізується встановленням властивості **ValidatesOnException**.

Нижче реалізовано правило перевірки достовірності для перевірки регулярного виразу. Клас **RegularExpressionValidationRule** успадкований від базового класу **ValidationRule** і використовується для перевірки достовірності виразів, визначених властивістю **Expression**.

```
public class RegularExpressionValidationRule : ValidationRule
{
  public string Expression { get; set; }
  public string ErrorMessage { get; set; }
  public override ValidationResult Validate(object value,
  CultureInfo cultureInfo)
  {
    ValidationResult result = null;
    if (value != null)
```

```
    {
      Regex regEx = new Regex(Expression);
      bool isMatch = regEx.IsMatch(value.ToString());
      result = new ValidationResult(isMatch, isMatch ? null : ErrorMessage);
    }
    return result;
  }
}
```

Замість використання розширення розмітки **Binding** прив'язка тепер здійснюється у вигляді дочірнього компонента елемента **TextBox.Text**. Прив'язаний об'єкт визначає властивість **Email**, яка реалізована в простому синтаксисі властивостей.

Властивість **UpdateSourceTrigger** визначає, коли джерело має бути оновлене. Можливі варіанти оновлення джерела такі:

1. коли змінюється значення властивості, т. т. при вводі кожного символа користувачем;
2. при втраті елементом фокусу;
3. явне оновлення.

ValidationRules – властивість класу **Binding**, яка містить елементи **ValidationRule**. Тут правило перевірки достовірності використовується у вигляді спеціального класу **RegularExpressionValidationRule**, в якому властивість **Expression** встановлена в регулярний вираз, який перевіряє, чи є введене користувачем значення коректною адресою електронної пошти, а властивість **ErrorMessage** містить повідомлення про помилку, яке виводиться у разі некоректного значення введеного в **TextBox**.

```
<Label Margin="5" Grid.Row="2" Grid.Column="0">Email:</Label>
  <TextBox Margin="5" Grid.Row="2" Grid.Column="1">
    <TextBox.Text>
      <Binding Path="Email" UpdateSourceTrigger="LostFocus">
        <Binding.ValidationRules>
            <src:RegularExpressionValidationRule
                Expression="A([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.
                            [0-9]{l,3}\.) | (([\w-]+\.)+))([a-zA-Z]{2,4} | [0-9] {1,3}) (\]?)$"
                ErrorMessage="Email is not valid" />
        </Binding.ValidationRules>
      </Binding>
    </TextBox.Text>
  </TextBox>
```

Зв'язування з командами

Зв'язування з командами (**commanding**) – це концепція WPF, яка створює слабкий зв'язок між *джерелом дії* (наприклад, *кнопкою*) і *метою*, яка виконує роботу (наприклад, *методом-обробником*). Компіляція коду XAML, який включає посилання на події, вимагає наявності у відокремленому коді

реалізованого обробника, доступного під час компіляції. При використанні команд такий зв'язок ослаблений.

Виконувана дія визначається об'єктом команди. Команди реалізують інтерфейс **ICommand**. Класи команд, використовувані WPF – це **RoutedCommand** і його клас-нащадок **RoutedUICommand**. Клас **RoutedUICommand** визначає додатковий текст для інтерфейсу користувача, який не визначений в **ICommand**. В інтерфейсі **ICommand** визначені методи **Execute()** і **CanExecute()**, які виконуються на цільовому об'єкті.

Джерелом команди є об'єкт, який *викликає* команду. Джерела команд реалізують інтерфейс **ICommandSource**. Прикладами таких джерел команд можуть бути класи кнопок, успадковані від **ButtonBase**, **HyperLink** та **InputBinding**. Прикладами класів-нащадків **InputBinding** є **KeyBinding** і **MouseBinding**. Джерела команд мають властивість **Command**, якій може бути присвоєний об'єкт команди, що реалізовує **ICommand**. Ця властивість ініціює команду при використанні елемента управління, наприклад, при клацанні на кнопці.

Метою команди є об'єкт, який реалізовує обробник для *виконання* дії. За допомогою прив'язки команд визначається відображення обробника на команду. Прив'язка команд визначає обробник, який викликається по команді. Прив'язки команд визначаються властивістю **CommandBinding**, яка реалізована в класі **UIElement**. Кожен клас-нащадок **UIElement** має властивість **CommandBinding**. Це робить пошук відображеного обробника ієрархічним процесом. Наприклад, команду може ініціювати кнопка, визначена в **StackPanel**, яка знаходиться усередині **ListBox**, який, у свою чергу, розміщений всередині **Grid**. Обробник заданий прив'язкою команди десь в дереві, як у разі прив'язки команд **Window**.

Давайте змінимо реалізацію проекту **BooksDemo**, скориставшись командами замість моделі подій.

Визначення команд

У .NET пропонуються класи, які повертають обумовлені команди. У класі **ApplicationCommands** визначені статичні властивості **New**, **Open**, **Close**, **Print**, **Cut**, **Copy**, **Paste** і т. д. Ці властивості повертають об'єкти **RoutedUICommand**, які можуть використовуватися для певної мети. До інших класів, які представляють команди, відносяться **NavigationCommands** і **MediaCommands**. Клас **NavigationCommands** надає команди, призначені для навігації, на зразок **GoToPage**, **NextPage** і **PrevousPage**. Клас **MediaCommands** зручний для управління програвачем медіа за допомогою команд **Play**, **Pause**, **Stop**, **Rewind** і **Record**.

Неважко визначити спеціальні команди для виконання дій, специфічних для домена застосунку. Нижче в прикладі створюється клас **BooksCommands**, який поверне **RoutedUICommand** за допомогою властивості **ShowBooks**. Команді можна також призначити жест вводу, такий як **KeyGesture** або **MouseGesture**. Тут призначається **KeyGesture**, який задає клавішу **<B>** з модифікатором **<Alt>**. *Жест вводу* – це джерело команди, тому натиснення клавіатурної комбінації **<Alt+B>** приведе до активізації команди.

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

namespace BooksDemo
{
  public static class BooksCommands
  {
    private static RoutedUICommand showBooks;
    public static ICommand ShowBooks
    {
      get
      {
        if (showBooks == null)
        {
          showBooks = new RoutedUICommand("Show Books", "ShowBooks",
          typeof(BooksCommands));
          showBooks.InputGestures.Add(new KeyGesture(Key.B, ModifierKeys.Alt));
        }
        return showBooks;
      }
    }
  }
}
```

Визначення джерел команд

Кожен клас, який реалізовує інтерфейс **ICommandSource**, може бути джерелом команд, у тому числі, **Button** або **MenuItem**. Всередині головного вікна елемент управління **Menu** додається як дочірній в **DockPanel**. Елементи **MenuItem** містяться всередині елемента **Menu**, а властивості **Command** призначаються наперед визначені команди, такі як **ApplicationCommands.Close**, і спеціальна команда **BooksCommands.ShowBooks**.

```
<Menu DockPanel.Dock="Top">
  <MenuItem Header="_File">
    <MenuItem Header="_Show Books" Command="local:BooksCommands.ShowBooks" />
    <Separator />
    <MenuItem Header="E_xit" Command="ApplicationCommands.Close" />
  </MenuItem>
  <MenuItem Header="_Edit">
    <MenuItem Header="Undo" Command="ApplicationCommands.Undo" />
    <Separator />
    <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
    <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
    <MenuItem Header="Pastee" Command="ApplicationCommands.Paste" />
  </MenuItem>
</Menu>
```

Прив'язки команд

Прив'язки команд потрібні для встановлення зв'язку їх з методами-обробниками. У приведеному нижче коді прив'язки команд визначені всередині елемента **Window**, так що вони доступні всім елементам всередині вікна. Коли виконується команда **ApplicationCommands.Close**, викликається метод **OnClose()**. Коли виконується команда **BooksCommands.ShowBooks**, викликається метод **OnShowBooks**().

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Close" Executed="OnClose" />
  <CommandBinding Command="local:BooksCommands.ShowBooks"
    Executed="OnShowBooks" />
</Window.CommandBindings>
```

У прив'язці команди можна також задати властивість **CanExecute**, де викликається метод для перевірки доступності команди. Наприклад, якщо файл не змінювався, команда **ApplicationCommand.Save** може бути недоступною. Обробник має бути визначений з параметром типу **object** – *відправником* і параметром **ExecutedRoutedEventArgs**, в якому міститься *інформація* про команду:

```
private void OnClose(object sender, ExecutedRoutedEventArgs e)
{
  Application.Current.Shutdown();
}
```

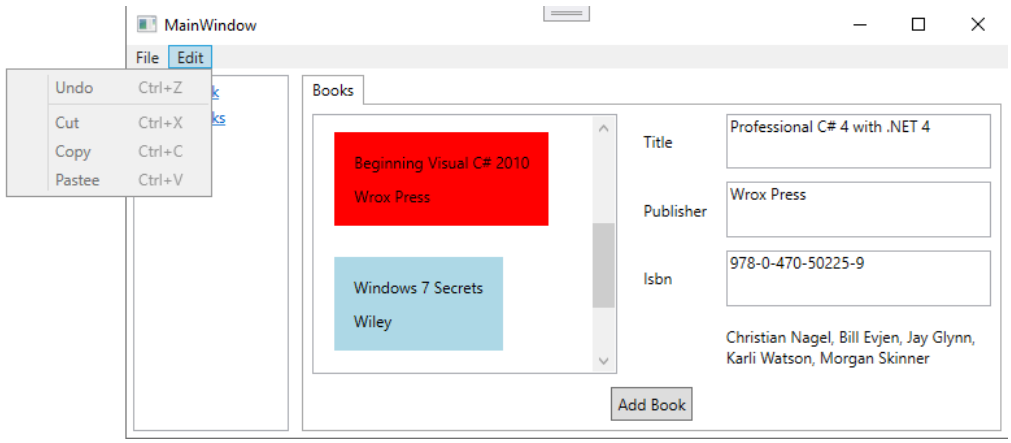


Рис. 13.23. Демонстрація зв’язування з командами

Зауваження: З командою можна також передавати параметри. Це робиться за рахунок задання властивості **CommandParameter** з джерелом команди, таким як **MenuItem**. До параметра можна звернутися через властивість **Parameter** об'єкта **ExecutedRoutedEventArgs**.

Елементи управління також можуть визначати прив'язки до команд. Елемент **TextBox** визначає прив'язки для **ApplicationCommands.Cut**, **ApplicationCommands.Copy**, **ApplicationCommands.Paste** та **ApplicationCommands.Undo**. У результаті знадобиться лише вказати джерело

команди і скористатися існуючою функціональністю всередині елемента управління **TextBox**.

Резюме

У цій лекції були описані засоби WPF, які надзвичайно важливі для бізнес-застосунків.

Прив'язка даних WPF зробила крок вперед. Тепер можна прив'язувати будь-яку властивість класу .NET до властивості елемента WPF. Режим прив'язки визначає її напрям. Допускається прив'язувати об'єкти .NET і списки, а також визначати шаблон даних, що дозволяє задати зовнішній вигляд за замовчуванням для класу .NET. Прив'язка команди дозволяє відобразити код обробника на меню і панелі інструментів. Було показано, наскільки легко копіювати і вставляти фрагменти тексту у WPF, оскільки обробник команд для цієї технології вже включений в елемент управління **TextBox**.

Рекомендована література (основна, допоміжна)

1. Натан А. WPF 4. Подробное руководство. Санкт-Петербург: Символ-Плюс, 2011. 880 с.
2. Мак-Дональд М. WPF 4: Windows Presentation Foundation в .NET 4.0 с примерами на C# 2010 для профессионалов. Москва.: «И. Д. Вильямс», 2011. 1024 с.
3. Петцольд Ч. Microsoft Windows Presentation Foundation. Базовый курс. Москва: Изд-во «Русская Редакция»; Санкт-Петербург: Питер, 2008. 944 с.

Допоміжна:

1. Албахари Дж., Албахари Б. C# 9.0. Карманный справочник. Киев: «Диалектика», 2021. 224 с.
2. Албахари Дж., Албахари Б. C# 9.0. Справочник. Полное описание языка. Киев: «Диалектика», 2021. 1056 с.
3. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. Москва: ООО «И.Д. Вильяме», 2011. 1392 с.
4. Нейгел К., Ивьен Б., Глинн Д., Уотсон К. C# 4.0 и платформа .NET 4 для профессионалов. Москва: ООО "И.Д. Вильяме", 2011. – 1440 с.
5. Шилдт Г. C# 4.0: полное руководство. Москва: ООО «И.Д. Вильямс», 2011. 1056 с.
6. Шахрайчук М. І., Шинкарчук Н. В., Шахрайчук А. М. Основи налаштування застосувань у Microsoft Visual Studio .NET: навчальний посібник. Рівне: РВВ РДГУ, 2018. 190 с.

Інформаційні довідкові системи:

1. Сховище документації Майкрософт для користувачів, розробників та IT-спеціалістів [Електронний ресурс]: інформаційно-довідкова онлайн система / Компанія Microsoft. URL: https://docs.microsoft.com/ru-ru/, Windows Presentation Foundation documentation – https://docs.microsoft.com/en-us/dotnet/desktop/wpf/?view=netdesktop-6.0

Навчальне видання

ІНТЕРФЕЙСИ КОРИСУВАЧА ТА СИСТЕМНІ ІНТЕРФЕЙСИ

Навчальний посібник

Лекції

Друкується в авторській редакції

Підписано до друку 23.06.2022 р.
Формат 60×90/16.
Ум. друк. арк. 11. Тираж 50 прим.
Замовлення №637/3

Відділ мережевого та інформаційного забезпечення
Рівненського державного гуманітарного університету.
33028, м. Рівне, вул. С. Бандери, 12.