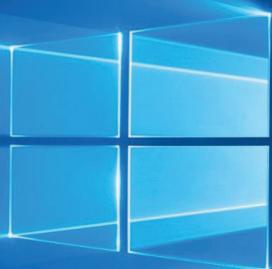


Сергій Байдачний  
Остапчук Маргарита

# Windows 10

## для C# розробників

### КНИГА 1



- Universal Windows Platform
- Основи XAML
- Елементи керування
- Стилі, ресурси і теми
- Зв'язування даних
- Графіка, трансформації та анімації
- Адаптивні інтерфейси
- Плитки та повідомлення
- Публікація в Магазині

# Про авторів



## **Сергій Байдачний**

працює у сфері розробки програмного забезпечення більше 14 років і останні 9 років є співробітником компанії Microsoft на позиції технічного євангеліста. В активі Сергія декілька книг, присвячених розробці програм на платформі .NET.

Блог Сергія доступний за адресою  
<http://en.baydachnyy.com>



## **Остапчук Маргарита,**

фахівець з інформаційних технологій у відділі стратегічних досліджень компанії Microsoft Ukraine. Основними сферами спеціалізації є розробка для платформи Windows та Microsoft Azure.

Блог Маргарити доступний за адресою  
<http://in4margaret.azurewebsites.net>

# Зміст

Розділ 1.

## Що таке Universal Windows Platform? ..... 1

Одне ядро, одна платформа програм, один Магазин .....	2
Інструменти і мови .....	6
Кросплатформена розробка і мости .....	10
Наша перша програма.....	13

Розділ 2.

## Основи XAML..... 19

Що таке XAML? .....	20
Базовий синтаксис .....	21
Простори імен в XAML.....	27
Пишемо код для сторінок і обробників подій .....	28
Розширення розмітки.....	32
Властивості залежностей .....	34

Розділ 3.

## Розмітка ..... 35

Canvas .....	36
StackPanel .....	37
Основні властивості та клас Panel .....	38
Grid.....	41
RelativePanel.....	45
ScrollViewer.....	49

Розділ 4.

**Поширені елементи керування..... 51**

Декілька слів про ієрархічну структуру .....	52
Кнопки.....	54
Робота з текстом .....	57
Елементи керування RangeBase.....	61
ProgressRing .....	62
Елемент керування ToolTip.....	63
Робота з колекціями.....	63
SplitView.....	70

Розділ 5.

**Взаємодія з користувачем ..... 73**

Розділ 6.

**Стилі, ресурси і теми ..... 81**

Основні відомості про стилі.....	82
Розширення стилів.....	85
Ресурси.....	86
Словники ресурсів.....	88
Теми.....	90
Як локалізувати програму .....	93

Розділ 7.

**Графіка, трансформації та анімації ..... 97**

Графічні примітиви .....	98
Пензлі .....	102
Геометричні фігури .....	106
Трансформації.....	108
Основи анімації.....	114
Вбудовані анімації.....	119

---

Розділ 8.**Зв'язування даних ..... 125**

Зв'язування двох елементів.....	126
Зв'язування в коді .....	130
Зв'язування елемента з об'єктом .....	130
Конвертери.....	142
Зв'язування з колекціями .....	145

## Розділ 9.

**Навігація й керування вікнами ..... 151**

Сторінки та навігація.....	152
Кнопка «Назад» є всюди .....	155
Життєвий цикл програм.....	157
Керування вікнами .....	161

## Розділ 10.

**Як створити адаптивні інтерфейси..... 171**

Що ми вже знаємо про адаптивні інтерфейси? .....	172
VisualStateManager.....	173
Сеттери.....	175
Адаптивні тригери .....	176
Як створити різні подання .....	180

## Розділ 11.

**Плитки та повідомлення ..... 185**

Динамічні плитки .....	186
Спливаючі сповіщення.....	195
Заплановані сповіщення.....	200
Періодичні сповіщення .....	201

Розділ 12.

**Як публікувати програми в Магазині..... 203**

Створення облікового запису для публікації.....	204
Підготовка до публікації програми .....	207
Публікація .....	216

Розділ 1.

# **Що таке Universal Windows Platform?**

## Одне ядро, одна платформа програм, один Магазин

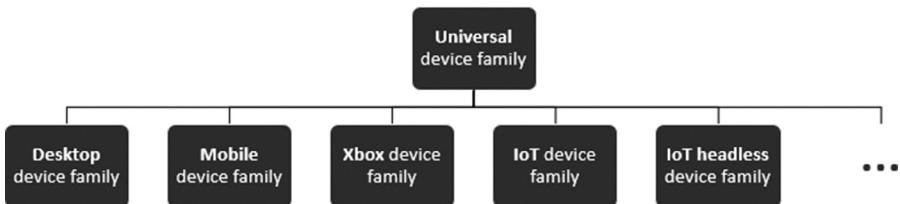
З часів Windows 95 найкраща операційна система – це Windows. Її можна інсталювати на будь-якому персональному комп'ютері, й відразу розробляти програми для Windows, які можуть запускатися на будь-якому ПК. Раніше ніхто не думав про інші пристрой, тому що їх просто не існувало.

Але відтоді корпорація Microsoft випустила багато різних версій Windows для багатьох типів пристрой. До Windows 10 ви могли розробляти Windows-програми для настільних комп'ютерів, планшетів, телефонів, Xbox, вбудованих пристрой. І зазвичай для різних типів пристрой вам потрібно було використовувати різні підходи. Наприклад, для розробки програм під Windows 8.x була необхідна бібліотека Windows Runtime; для розробки програм під Windows Phone 8.x – Silverlight і, пізніше, Windows Runtime, але з іншим набором елементів керування і класів; для розробки програм для Інтернету речей на платі Galileo – C++ і набір API з Win32; Xbox вимагав доступу до спеціальних приватних SDK. Тому розробити щось універсальне для всіх пристрой Windows було непростим завданням. Також Microsoft створив Магазини для Windows і Windows Phone, а отже, розробникам потрібно було підтримувати обидва ці Магазини замість одного.

Але випуск Windows 10 змінив усе, і тепер ми говоримо про програми для Windows 10, тобто про програми, що працюють на настільних комп'ютерах, планшетах, Xbox, телефонах, HoloLens та в Інтернеті речей. Прочитавши цю книгу, ви навчитеся розробляти програми для всіх типів пристрой і публікувати їх у Магазині Windows принаймні для комп'ютерів, планшетів і телефонів. Давайте поглянемо, як це можна реалізувати.

Передусім ми маємо згадати про ядро Windows, що є спільним для всіх версій цієї операційної системи. Тобто навіть якщо ви бачите певні відмінності між інтерфейсом пристрой IoT (Інтернету речей) та телефонів, вони все одно мають спільне ядро, що містить стандартні протоколи комунікацій, універсальну модель драйверів, спільну платформу для інтерфейсу програм та ін. Базуючись на одному ядрі, Microsoft може створювати різні версії Windows, додаючи компоненти, що є специфічними для певного типу пристрой. Саме тому Windows для персональних комп'ютерів підтримує режим комп'ютера і планшету, а версія для IoT дає змогу лише запускати ваші програми.

Сьогодні Microsoft здійснює підтримку декількох Windows SKU для різних наборів пристрой, наприклад Desktop – для персональних комп'ютерів і планшетів, Mobile – для телефонів і невеликих планшетів тощо.



Різні Windows SKU можна вважати різними наборами налаштувань тої самої Windows. Особливо це відчутно для розробників, тому що завдяки одному ядру ми можемо говорити про єдину платформу розробки для всіх можливих типів пристройів. Ця платформа називається Universal Windows Platform (UWP) і є основною темою цієї книги. Отже, якщо вам відомо, як будувати програмами, використовуючи Universal Windows Platform, ви відразу знаєте, як будувати програми для всіх пристройів з Windows 10.

Звичайно, якщо ви хочете використовувати деякі компоненти, що є лише на певному пристройі, ви також зможете це зробити, тому що Universal Windows Platform підтримує спеціальний механізм, що називається розширенням, і завдяки ньому можна включати до програми додаткові можливості, властиві для того чи іншого пристроя, і навіть перевіряти їх наявність під час виконання програми.

Universal Windows Platform – це нове покоління платформи Windows Runtime, що була представлена у складі Windows 8 і є нативно об'єктно-орієнтованою платформою для розробників програмного забезпечення. Починаючи з Windows Phone 8.1, Windows Runtime додано до Windows Phone, і зараз ми можемо використовувати цю платформу всюди. Але Universal Windows Platform – це не лише перейменування Windows Runtime через глобальне використання на всіх пристроях. UWP – не залежить від версії Windows. Перша версія Windows Runtime була представлена для Windows 8.0, а перше оновлення – для Windows 8.1. Однак UWP Microsoft може оновлювати незалежно від оновлень Windows. І деякі з оновлень ви можете знайти раніше, ніж оновлення Windows, але деякі і пізніше. Тобто ідея схожа до оновлень .NET Framework. Деякі розробники й досі створюють програми для .NET Framework 2.0 і ви можете знайти декілька версій .NET Framework на одному комп'ютері. Таким чином, можна орієнтуватися на стару версію UWP і ваша програма буде працювати скрізь, або ви можете орієнтуватися на останню версію, щоб використовувати деякі нові функції.

Для кращого розуміння UWP можна перейти в папку **<disk>:\Program Files (x86)\Windows Kits\10** і знайти всі файли, що стосуються SDK. Переглядаючи різні папки, ви можете знайти багато підпапок **10.0.10240.0**. Це збірка Windows,

що була випущена для загалу, і вона відповідає першій версії Universal Windows Platform. Як тільки виходить нова версія UWP, за вищезазначеним шляхом з'являються інші папки, які стосуються цієї версії. У папці **Platforms\UAP** ви можете знайти файл Platform.xml, у якому міститься інформація про платформу:

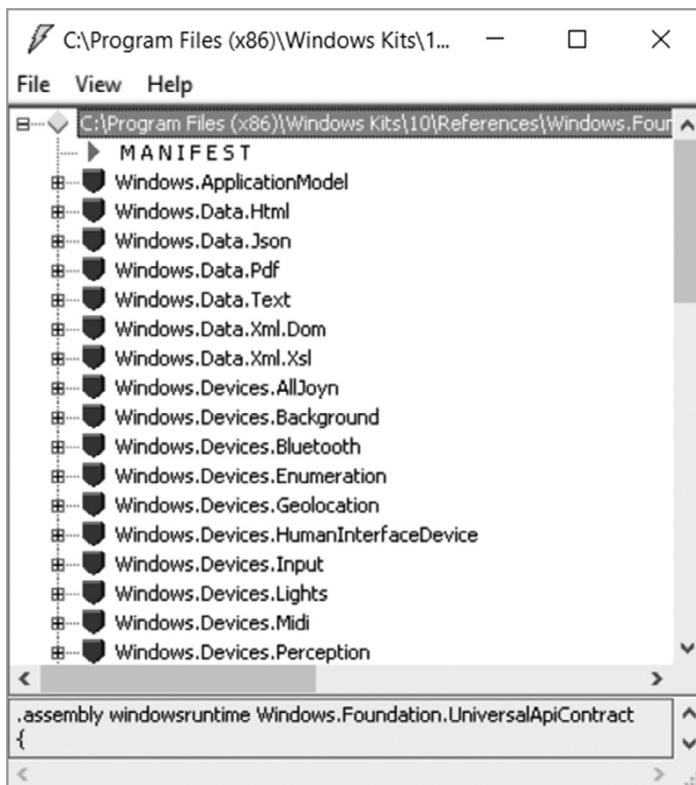
```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationPlatform name="UAP" friendlyName="Windows 10"
version="10.0.10240.0">
    <MinimumVisualStudioVersion>14.0.22213.01
    </MinimumVisualStudioVersion>
    <ContainedApiContracts>
        <ApiContract name="Windows.Foundation.FoundationContract"
            version="1.0.0.0" />
        <ApiContract name="Windows.Foundation.UniversalApiContract"
            version="1.0.0.0" />
        <ApiContract name="Windows.Networking.Connectivity.
            WwanContract" version="1.0.0.0" />
    </ContainedApiContracts>
</ApplicationPlatform>
```

Тут ви можете побачити повну назву версії, зрозумілі для користувача імена, основні контракти API та інше. Пізніше ми обговоримо, як обирати певну платформу у Visual Studio.

Використовуючи наведений вище XML-файл, ви можете побачити, що Universal Windows Platforms за замовчуванням включає три контракти. По суті, контракт – це бібліотека, що містить класи і метадані в окремих файлах. Множина контрактів утворює своєрідне розширення до програм UWP і платформи UWP загалом. Якщо ви хочете переглянути всі доступні контракти, можете відвідати папку **References**, щоб знайти їхні метадані.

Ідея метаданих проста. Оскільки ні Windows Runtime, ані пізніша UWP не є нативними платформами, існує та ж проблема, що й при роботі з СОМ-компонентами – ніхто не знає, що всередині компонента. Проблема була вирішена в .NET Framework шляхом включення інформації про метадані в усі компоненти/збірки. Розширення наявної СОМ-моделі є не простим завданням, як і додавання метаданих. Саме тому Microsoft вирішив перенести метадані в окремий файл. Тому в папці **References** ви можете знайти файл **WINMD**, який містить метадані. Завдяки метаданим легко зрозуміти, що містить вибраний контракт, а інструменти розробки, наприклад Visual Studio, можуть використовувати метадані для надання підтримки IntelliSense та ін.

Якщо ви хочете відкрити метадані, найкращий спосіб це зробити – використати інструмент **ildasm**. Він працює для .NET Framework, але Microsoft додав можливість читати також метадані Windows Runtime. Для запуску **ildasm** можна відкрити Developer Command Prompt для VS 2015 і ввести **ildasm**. Як тільки ви запустили даний інструмент, можете відкривати будь-які файли метаданих з папки **References**:



Отже, з Windows 10 ми отримали єдине ядро і платформу для програм. І останньою темою, яку хотілося б обговорити, є Магазин. Починаючи з Windows 10 Microsoft анонсував єдиний Магазин для всіх програм Windows 10. Розробники зараз можуть публікувати єдиний пакет в Магазині та інсталювати програму на будь-який пристрій Windows 10, зокрема на телефони, комп'ютери, планшети тощо. Завдяки цьому вам не потрібно витрачати час на просування програми в різних Магазинах, щоб охопити максимум користувачів.

У другій книзі ми розглянемо Магазин детальніше.

## Інструменти і мови

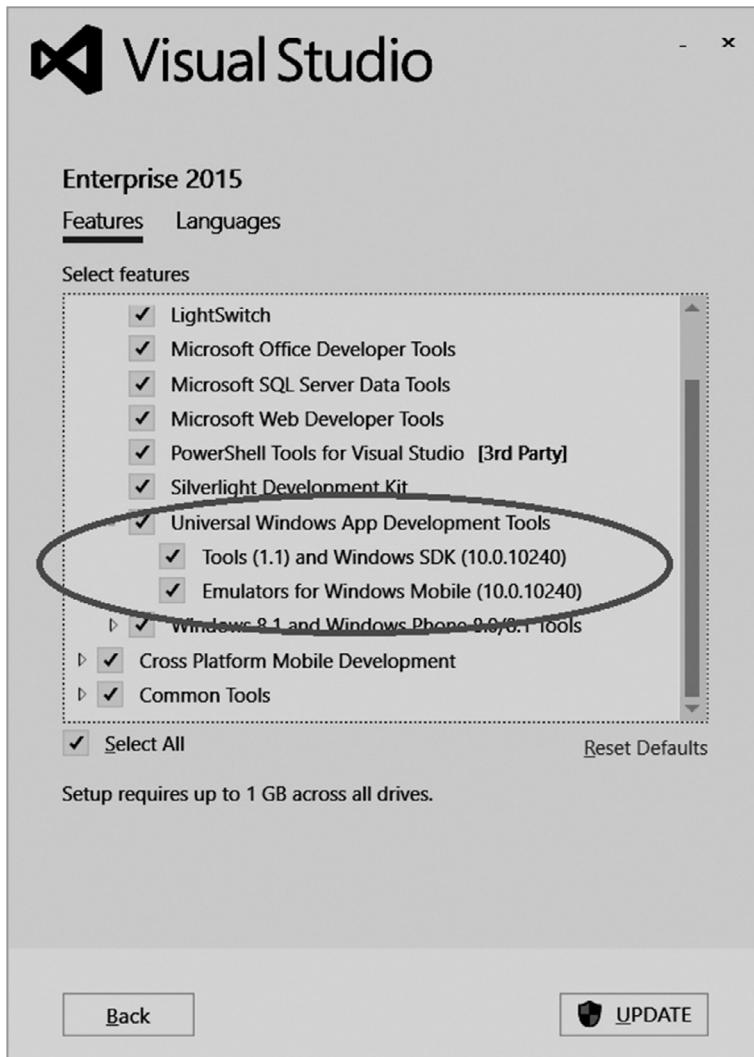
Звичайно, найкращим інструментом для розробки програм під Windows 10 є Visual Studio 2015. Є декілька різних версій Visual Studio, зокрема Microsoft пропонує безкоштовну версію – Visual Studio 2015 Community Edition. Ви можете просто відвідати сайт <http://visualstudio.com> і завантажити продукт з основної сторінки. Якщо у вашій команді до п'яти розробників, можна використовувати Community Edition безкоштовно і без будь-яких обмежень.

Версія Community Edition замінює версію Express, але по суті вона включає всі можливості Professional Edition, зокрема підтримку модулів plug-in, усіх мов і проектів, інтеграцію з Git та ін.

У тому разі, якщо ви не можете встановити Community Edition на робочий комп'ютер через можливе порушення ліцензійних угод, можна і надалі використовувати Express версію Visual Studio для Windows. Звичайно, у вас будуть певні обмеження, але цього буде достатньо для того, щоб почати вивчення розробки під Windows 10.

Загалом для розробки програм під Windows 10 вам потрібно інсталювати Windows 10 на ваш комп'ютер, але Visual Studio 2015 дає змогу писати програми під Windows 10 на Windows 8.x і навіть Windows 7. Звичайно, тоді у вас не буде зможи запускати і відразу тестувати програми на вашому комп'ютері, але у випадку з Windows 8.x ви можете використовувати Windows Phone Emulator і бачити, як ваша програма працює на мобільному пристрої. Також у багатьох випадках ви можете тестувати програму й налагоджувати її віддалено на пристрій з Windows 10.

Встановлюючи Visual Studio 2015, вам потрібно переконатись, що ви інсталювали інструменти розробки Universal Windows App Development. За замовчуванням вони можуть бути не вибрані, тому краще обирати вибіркове встановлення.



Коли Visual Studio 2015 інсталяовано, ви можете розпочинати розробляти програми для Windows 10. Перш ніж ви створите власну програму, вам потрібно вирішити, якою мовою її написати.

Universal Windows Platform підтримує чотири мови, що дають змогу створювати програми для Windows 10: C#, VB.NET, C++ і JavaScript.

C++ традиційно нативна для платформи Windows і ви можете створювати справді нативні програми для Windows, але у деяких випадках це може бути досить важко, навіть з використанням сучасних засобів. Якщо ви хочете писати програми для Windows 10, вам краще використовувати мову C++ з розширеннями (C++/CX), що дають змогу використати деякі стандартні можливості C#, такі як інтерфейси, властивості, події та інше.

Код на C# і VB.NET виглядає набагато зрозуміліше і ви можете використовувати C# також для багатьох інших типів програм, зокрема ігор. Звичайно, C# є керованою мовою, але у випадку програм для Windows 10 це перевага, тому що в C# ви можете користуватись основними класами .NET Framework, що дають можливість працювати з рядками, колекціями, LINQ та ін. Водночас у Visual Studio застосовується технологія .NET Native, що дає змогу компілювати програму для Windows 10 у хмарі. Перед створенням пакету для випуску Visual Studio здійснить оптимізацію до .NET Native і як тільки користувач надішле запит на вашу програму в Магазин, він буде скомпільований в нативний код. Якщо користуваєтеся C#, у вас не має виникнути жодних проблем з продуктивністю. Зверніть особливу увагу, що Visual Studio надає всі можливості налагодження завдяки використанню .NET Core CLR.

Коли йшлося про Universal Windows Platform, ми говорили про нативні API для Windows. Але навіть у тому разі, якщо ваш C#-код був скомпільований за допомогою компілятора .NET Native, ви не можете використовувати нативні бібліотеки, такі як .NET Framework Core. Саме тому Microsoft розробив проекції мови для C# і VB.NET. Загалом проекція мови – це прошарок між мовою та Windows Runtime, що дає змогу використовувати бібліотеки Windows Runtime, такі як нативні .NET Framework. Використовуючи керовані мови, ви не будете відчувати незручностей при взаємодії з ядром Windows.

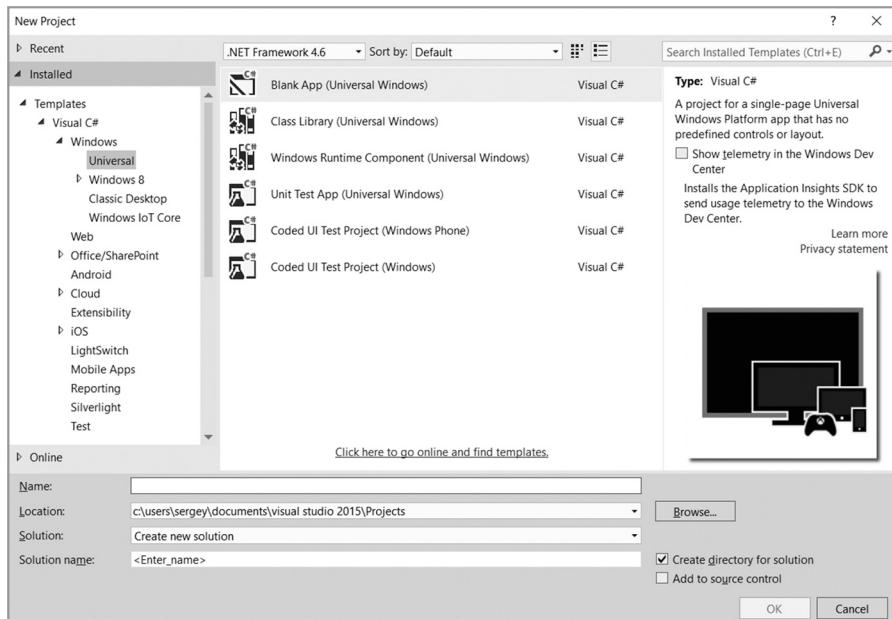
Розробляючи програми для Windows 10 на C++, C# чи VB.NET, ви будете приділяти багато часу реалізації користувачького інтерфейсу. Звичайно, мови програмування – не найкращий засіб для імплементації інтерфейсів. Саме тому другою мовою, яку розробники використовують при побудові програм для Windows 10, є XAML (eXtensible Application Markup Language). У наступній частині ми приділимо увагу саме XAML. Зазвичай якщо ви обираєте основну мову для програмування вашої програми поміж C++, C#, VB.NET, то маєте на увазі такі сполучення технологій: C++/XAML, VB.NET/XAML, C#/XAML (але це не стосується JavaScript).

У випадку JavaScript ви можете використовувати HTML для створення користувачьких інтерфейсів. Як і для C# і VB.NET, Microsoft розробив спеціальне відображення для мови JavaScript і додав усі потрібні сутності до бібліотеки WinJS. Із JavaScript і WinJS ви можете й надалі використовувати стандартні елементи керування на основі HTML і популярні бібліотеки JavaScript, але додатково можна

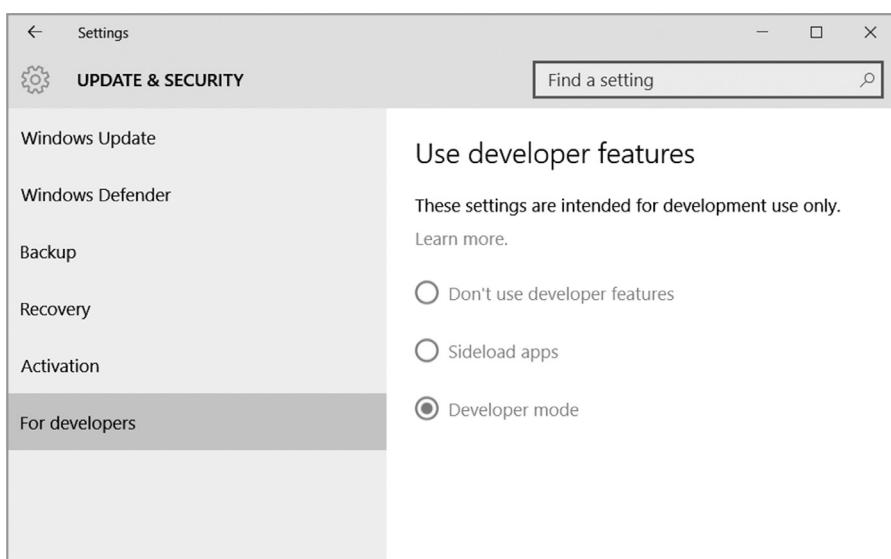
користуватися всіма можливостями Universal Windows Platform. Отже, якщо вам подобається JavaScript, ви можете використати цю мову для створення нативних програм для Windows 10.

У цій книзі для створення програм під Windows 10 ми обрали C#/XAML. Ви можете використати деякі з матеріалів книги, якщо будете писати на C++, але для JavaScript-розвробок інформація буде не дуже релевантна.

Працюючи з Visual Studio 2015, ми будемо використовувати шаблони категорії Visual C#. Якщо ви успішно інсталюєте інструменти Universal Windows App Development, то знайдете підкатегорію Windows/Universal. У більшості випадків ми будемо використовувати шаблон Blank App:



Зверніть увагу, що ви не зможете запустити вашу стандартну програму, тому що Windows не дозволяє встановлювати сучасні (UWP) програми не з Магазину. Для того, щоб мати змогу запускати власні програми на вашому комп’ютері, вам потрібно відкрити вікно **Settings** і ввімкнути на пристрої режим розробки (**Developer mode**):



Як тільки ви це зробите, можете починати писати вашу першу програму.

## Кросплатформена розробка і мости

Перед тим, як переходити до кодування, ми б хотіли обговорити ще одну тему – кросплатформену розробку. Це дуже актуальна тема, тому що на деяких із численних пристройів, які сьогодні використовуються, може бути інсталювана не ОС Windows, а Android чи iOS. Отже, ми маємо дві теми для обговорення: створення з нуля програм, що будуть працювати на всіх доступних платформах і міграція наявних програм для Android та iOS на Windows 10.

Якщо ви захочете створити кросплатформену програму і витратити менше часу на вивчення нових мов, інструментів та бібліотек, можете використати одну з трьох технологій, що інтегровані у Visual Studio 2015: Xamarin, Apache Cordova і C++.

Щоб увімкнути інтеграцію з Xamarin та/або Apache Cordova, вам потрібно інсталювати інструменти прямо з інсталятора Visual Studio 2015.



Інсталятор містить все, що вам потрібно. Загалом ви можете починати розробку програми відразу після інсталяції Visual Studio і вам не потрібно думати про Android SDK чи щось інше.

Перший інструмент, Xamarin, побудований на реалізації відкритого коду .NET Framework – Mono. Використовуючи Xamarin, ви можете і далі писати на C# програми під Android, iOS і Windows. Xamarin – це платний продукт, але ви маєте змогу користуватись Xamarin Starter Edition безкоштовно. Основна ідея проста – реалізувати проекцію мови для Android та iOS, як Microsoft зробив це для Windows Runtime і C#. Сьогодні ви можете побачити, що всі API для Android і iOS реалізовані (запаковані) у керованих класах, і немає нічого такого в Java та Objective-C/Swift, чого ви не могли б реалізувати на C#. Але більш важливо те, що Xamarin дає змогу створювати нативні програми.

Використовуючи Starter Edition для Xamarin, ви можете, по-перше, розробити все на C# і, по-друге, створити загальну бізнес-логіку. На додаток до цього Visual Studio підтримує емулятори Android та iOS і Android-програми, але вам усе одно доведеться створювати інтерфейси для кожної операційної системи окремо. Але якщо у вас є ліцензія Xamarin (починаючи від \$25 на місяць), ви можете створити справжній універсальну програму. Xamarin Forms дає змогу використовувати

універсальний набір елементів керування для побудови Universal UI для всіх платформ, зокрема для Windows 10.

Наступний інструмент – це Apache Cordova, що є абсолютно безкоштовним і дає змогу використовувати HTML і JavaScript для створення програм для Windows, iOS і Android. Програми, що ви створюєте завдяки Apache Cordova, не є повністю нативними. Взагалі-то Apache Cordova не надає багато бібліотек, але дозволяє використовувати той самий базовий код (HTML і JavaScript) і загортати його в один пакет програми, використовуючи весь вбудований контент веб-елемента керування, що доступний на всіх платформах. Наприклад, веб-елемент керування в UWP називається WebView і дає змогу подати веб-контент з пакету програми чи за допомогою Uri. Схожий елемент керування доступний в SDK для Android і iOS. Звичайно, є можливість поєднати контент, що був вбудований у пакет програми, і зовнішній контент.

Застосовуючи стандартний набір HTML-тегів, ви, звичайно, не можете користуватися специфічними можливостями і компонентами платформи. Саме тому Apache Cordova підтримує плагіни, що агрегують різні API і надають розробникам під Apache Cordova єдиний API, що не залежить від пристроїв. Саме тому, якщо ви надаєте перевагу JavaScript, то можете користуватись Apache Cordova, як універсальною платформою для основних операційних систем.

Нарешті, починаючи з Visual Studio 2015, Microsoft включає підтримку C++ для Android та iOS. Зараз ви можете прямо у Visual Studio користуватися тим самим кодом для створення на C++ бібліотек для Android і iOS.

Мова йшла про нові програми. Але якщо у вас уже є програми на іншій платформі, ви можете використати спеціальний набір інструментів і бібліотек, що називаються мостами. Microsoft представив три мости, що дають змогу перенести веб-, Android- і iOS-програми на Windows. Перший міст ми обговоримо у розділі 25. Наступні два мости досі в розробці, але ви можете знайти їхні ранні бета-версії.

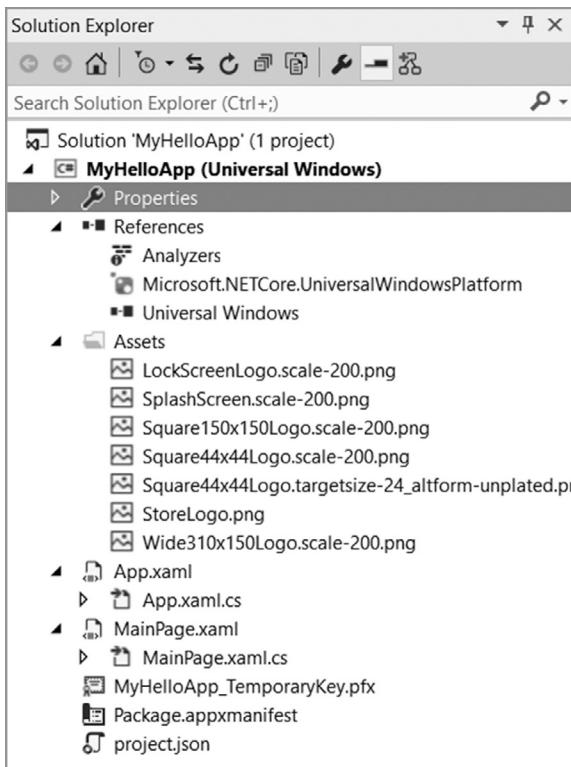
Ідея мосту для Android – це перенести до нативного Android IDE деякі інструменти, що будуть аналізувати програму для виявлення можливих проблем на Windows та рекомендувати, як усунути ці проблеми. Як тільки програма готова, ви можете створювати пакет програми прямо з Android IDE. І вам не потрібно переписувати ваш код.

У випадку iOS ви будете мати можливість відкрити проект і скомпілювати програму у Visual Studio, а також додати певну функціональність в Objective-C чи Swift. Тому, якщо у вас є програми, які ви хочете перенести на Windows 10, звертайтеся до представників Microsoft у вашому регіоні.

## Наша перша програма

Ми обговорювали Universal Windows Platform, інструменти і навіть кросплатформену розробку, і тепер настяг час створити першу програму за допомогою Universal Windows Platform.

Як ми вже говорили, для того, щоб почати, вам потрібно створити пустий проект – і Visual Studio заповнить його всім необхідним:



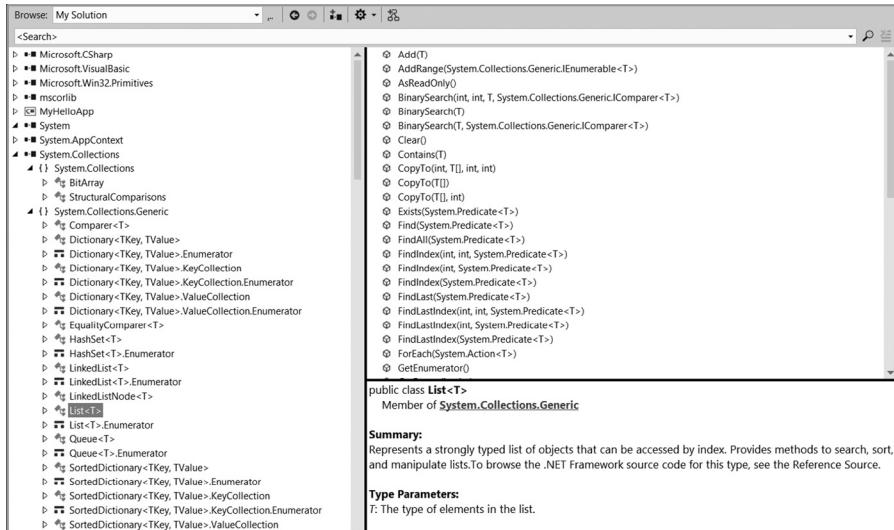
Давайте відкриємо Solution Explorer і побачимо, що Visual Studio для нас створила.

Спочатку звернемо увагу на папку References: там ви можете знайти два посилання на два фреймворки.

Перший – Universal Windows – це загальний контракт UWP, що реалізовує всі основні можливості. У більшості випадків – це все, що вам потрібно для створення універсальної програми під Windows.

Друге посилання – це Microsoft.NETCore.UniversalWindowsPlatform. Як ми й казали раніше, C# – це керована мова, яка дає змогу використовувати базові класи з .NET Framework. Саме тому Visual Studio додала посилання на .NET Core і завдяки цьому ви можете використовувати колекції, основні типи .NET, LINQ та інше.

Для того, щоб побачити усі класи, які включені до Universal Windows Platform і .NET Core, радимо використовувати Object Browser. Ви можете відкрити вікно, використовуючи меню **View**. У вас є також спосіб фільтрувати класи за простором імен, обираючи певний фреймворк чи просто **My Solution**:



Якщо ми виберемо **My Solution**, то зможемо побачити загадані вище контракти UWP і багато збірок .NET Framework – просто відкрийте деякі з них, щоб побачити простори імен і класи.

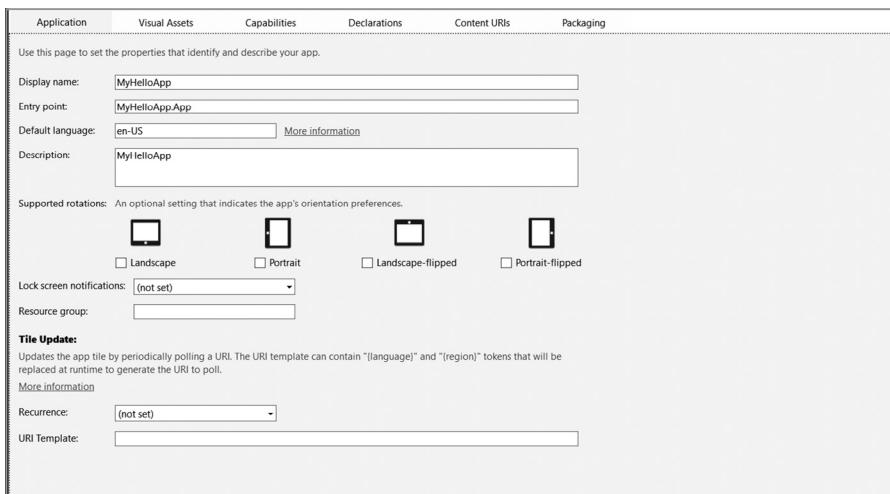
Наступна папка, яку ви можете знайти в Solution Explorer, – це Assets. Тут міститься картинки, що відображатимуться для вашої програми в Магазині або на стартовому екрані.

Зрештою ви можете побачити декілька файлів у кореневій папці:

- **MainPage.xaml** і **MainPage.xaml.cs** – ці два файли відображають основну сторінку програми. Перший файл містить три елементи інтерфейсу XAML, а другий – C#-код, куди ви можете додати обробники подій та інші сторінки, що стосуються коду. Сторінка поки порожня. Згодом саме тут ви почнете розробляти першу UI-сторінку.

- **App.xaml і App.xaml.cs** – файли, що містять саму програму. **App.xaml.cs** містить весь код, що утворює інфраструктуру вашої програми та спрямовує її на головну сторінку. Оскільки об'єкт програми сам по собі не містить елементів інтерфейсу, файл **App.xaml** порожній, але пізніше ви зможете додати в нього ресурси програми.
- **Project.json** – цей файл містить список пакетів, що були включені в програму за допомогою менеджера пакетів NuGet. За замовчуванням там не міститься жодних пакетів.
- **Package.appmanifest** – це дуже важливий файл, що містить усю інформацію про програму, зокрема про мови, що підтримуються, плитки, заставки, розширення та інше. У нашій книзі ми будемо часто працювати з цим файлом.
- **.pfx** – кожна програма має бути підписана сертифікатом того, хто публікує її в Магазині. Сертифікат ви можете отримати під час публікації. Але спочатку облікового запису в Магазині у вас немає і Visual Studio в цьому випадку створить за вас тимчасовий pfx-файл, щоб підписати ваші пакети. Як тільки ви асоціюєте вашу програму з Магазином, Visual Studio замінить pfx-файл справжнім сертифікатом.

Перед тим, як відкривати першу сторінку, рекомендуємо відкрити файл **Package.appmanifest**. Це файл формату XML, але Visual Studio підтримує дизайнер маніфестів і тому ви можете змінювати майже всі компоненти файлу:

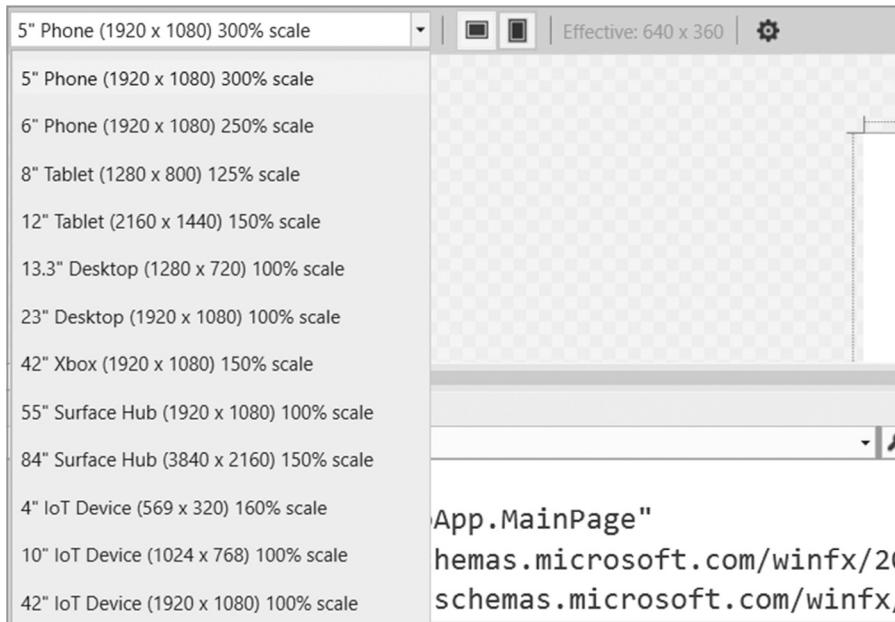


В дизайнери маніфестів ви побачите шість вкладок, що містять різну інформацію про пакети програми:

- **Application** – тут ви можете знайти основну інформацію про програму, таку як точка входу в програму, опис, орієнтація вікна тощо.
- **Visual Assets** – на цій сторінці міститься список плиток і посилань на зображення, що відображаються на плитках.
- **Capabilities** – усі сучасні програми для Windows 10 запускаються безпечно, тобто для них обмежено можливості доступу до нативних Win32 API і ресурсів системи, що містяться поза межами власної пісочниці. Якщо ви спробуєте використовувати заборонені API, ваша програма не зможе пройти сертифікацію в Магазині. І навіть для сертифікованих програм є певні правила. Наприклад, якщо ви хочете використовувати камеру, мікрофон чи API для визначення розташування, вам потрібно повідомити про це користувача. Усі ці можливості мають бути відображені користувачу на сторінці в Магазині, і, базуючись на цій інформації, користувач має вирішити, чи бажає він інсталювати програму. Але навіть за умови, що користувач установив вашу програму, вам потрібно перевірчитися, чи є у програми дозвіл на використання зазначених можливостей, оскільки більшість можливостей користувач має змогу вимкнути на сторінці Конфіденційності у вікні налаштувань.
- **Declarations** – використовуючи цю сторінку, ви можете описати і встановити певні можливості, що дають змогу інтегрувати вашу програму в операційну систему. Windows буде використовувати цю частину маніфесту для оновлення реєстру і ввімкнення інтеграції. Наприклад, ви можете асоціювати програму з деякими типами файлів і Windows буде використовувати його для того, щоб відкривати ці файли чи, наприклад, надавати контент для вікна блокування та ін.
- **Content URIs** – цю сторінку призначено для інтеграції з веб-контентом. Ми обговоримо це питання у розділі 25.
- **Packaging** – ця сторінка містить інформацію про ім'я програми в Магазині і сертифікат для програми. Зазвичай вам не потрібно буде модифікувати цю сторінку, тому що Visual Studio може заповнити її автоматично.

Перед публікацією програми в Магазині слід приділити певний час створенню маніфесту – це дасть змогу виконати інсталяцію коректно. Наразі в нашому проекті публікувати немає чого, тому нарешті ви можете відкрити **MainPage.xaml**, щоб побачити інтерфейс головної сторінки.

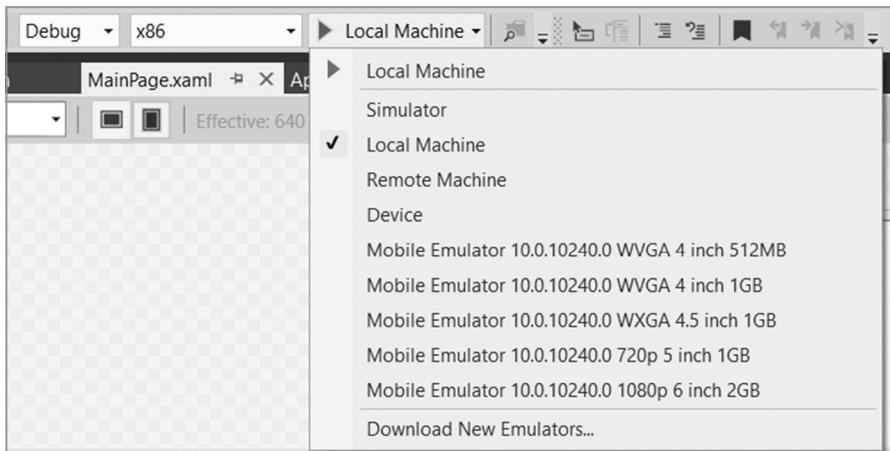
У Visual Studio ви можете редагувати файл XAML як текст або скористатися режимом дизайну. Зазвичай з елементами XAML працюють у редакторі коду, а режим дизайну використовують для перегляду інтерфейсу. Вкрай важливо перевірити, як виглядає ваш інтерфейс на різних видах пристройів, особливо якщо ви застосовуєте різну розмітку:



Давайте додамо всередину Grid-елемента такий рядок коду:

```
<TextBlock Text="Hello World!" HorizontalAlignment="Center"  
VerticalAlignment="Center" />
```

Для запуску програми у Visual Studio передбачено декілька можливостей. Ви можете запустити її локально, якщо у вас інсталявовано Windows 10, або обрати для запуску і налагодження віддалений пристрій. Можна використати зовнішній пристрій Windows Phone, що приєднаний до вашого ПК за допомогою кабелю USB, або емулятор Windows Phone:



Оберіть найкращий, на ваш погляд, спосіб для запуску програми. Ви побачите таке вікно:



Вітаємо! Вашу першу програму для Windows 10 створено.

Розділ 2.

## **Основи XAML**

## Що таке XAML?

Якщо у вас є певний досвід розробки користувацького інтерфейсу, ви знаєте, що такі мови, як C#, C++, Java і JavaScript, є не дуже вдалими для створення форм, стилів та ресурсів. Погляньмо на код програми Windows Forms для створення простої кнопки:

```
this.myButton = new System.Windows.Forms.Button();
this.myButton.Location = new System.Drawing.Point(120, 60);
this.myButton.Name = "myButton";
this.myButton.Size = new System.Drawing.Size(110, 40);
this.myButton.Text = "Hello";
this.Controls.Add(this.myButton);
```

Як бачите, створення простої кнопки вимагає написання шести рядків коду! Зазвичай розробники змушені присвоювати додаткові властивості кожному елементу керування, а додаткові форми можуть містити сотні елементів керування та контейнерів. У такому разі внесення змін до коду стає непростим завданням. Звичайно, Visual Studio та інші IDE підтримують різні типи візуальних редакторів, але здебільшого потрібно все-таки заглиблюватись у код та здійснювати модифікацію безпосередньо. Особливо якщо у вас виникла необхідність створити адаптивний інтерфейс, що змінює свій вигляд під час виконання.

Цікаво, що веб-розробники багато років мали рішення цієї проблеми – HTML та JavaScript, де HTML використовується для оголошення інтерфейсів, а JavaScript – для логіки програми. Ось чому майже 10 років тому Microsoft анонсував мову, що називається XAML.

XAML (eXtensible Application Markup Language) – це спеціальна декларативна мова, що базується на XML і здебільшого використовується для побудови інтерфейсів.

XAML було представлено в .NET Framework 3.0 як частину технології Windows Presentation Foundation. Але відтоді розробники мали можливість використовувати XAML для Silverlight, Windows Workflow і Windows 8.x/Windows Phone 8.x. Звичайно, ми застосовуватимемо XAML в усіх програмах Windows 10 у цій книзі.

Погляньмо на простий XAML-код, що створює ту саму кнопку всередині контейнера Canvas:

```
<Canvas x:Name="LayoutRoot">
    <Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
        Width="110" Height="40" x:Name="myButton"></Button>
</Canvas>
```

Цей код є набагато кращим для розуміння, оскільки елемент керування описується в одному тегу і завдяки XAML-редактору Visual Studio або Blend ви можете легко зрозуміти ієрархічну структуру будь-якого інтерфейсу. Знайти елемент в дереві інтерфейсу та дізнатися про його властивості з атрибутів XAML-тегів – це дуже простий і зрозумілий підхід.

Звичайно, за допомогою XAML описують не лише візуальні елементи керування. Розробники можуть використовувати його для оголошення ресурсів, стилів, станив інтерфейсу і об'єктів без візуального подання. Крім того, XAML має розширення розмітки і, на відміну від звичайного XML, дозволяє будувати зв'язки між XAML і кодом C# або між різними частинами XAML. У цьому розділі ми розглянемо всі аспекти цієї мови.

## Базовий синтаксис

Якщо у вас уже є певний досвід використання XML, тоді ви зможете легко зrozуміти XAML. Як і в XML, кожен елемент описується тегом, що може містити атрибути та дочірні елементи. Атрибути визначаються такими парами, як **ім'я="значення"**, де значення – це довільний рядок, і розміщуються всередині початкового тегу.

```
<Canvas Name="LayoutRoot">
    <Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
        Width="110" Height="40" Name="button1"></Button>
</Canvas>
```

Код вище містить два елементи: **Canvas** і **Button**. Елемент **Canvas** має тільки один атрибут, а от елемент **Button** має уже шість атрибутів, розташованих усередині **Canvas**. Як і в XML, ви можете описати елемент без будь-яких інших елементів усередині, використовуючи лише одинарний тег:

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
    Width="110" Height="40" Name="button1" />
```

У цьому випадку варто не застосовувати тег **</Button>**, а закривати початковий тег, записавши **</>** замість **<>**.

Як і XML, XAML чутливий до реєстру, і це правило стосується не тільки назв елементів і атрибутів, а й нерідко значень атрибутів. Наприклад, цей код описує дві кнопки з різними іменами **button1** і **Button1**.

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"  
       Width="110" Height="40" Name="button1" />  
<Button Content="Hello" Canvas.Top="160" Canvas.Left="120"  
       Width="110" Height="40" Name="Button1" />
```

Пізніше ми зможемо використовувати ці імена, щоб отримувати доступ до елементів керування з коду C#.

XAML дуже гнучкий і дає змогу писати код у кілька способів. Наприклад, ви можете задавати властивості, використовуючи не тільки атрибути, а й дочірні елементи. Наступні два блоки коду описують ту саму кнопку.

### Блок 1

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"  
       Width="110" Height="40" Name="button1"></Button>
```

### Блок 2

```
<Button Content="Hello" Canvas.Top="60" Canvas.Left="120"  
       Name="button1">  
    <Button.Width>  
        110  
    </Button.Width>  
    <Button.Height>  
        40  
    </Button.Height>  
</Button>
```

Отже, для того, щоб присвоїти значення властивості елемента керування з використанням дочірнього елемента, розробники повинні застосовувати таке позначення: **<Назва елемента>.<назва властивості>**. Звичайно, це має сенс, якщо ви хочете присвоїти значення складному об'єкту, що не може бути визначений за допомогою текстового рядка. Але навіть у випадку, якщо присвоєння досить складне, інколи можна використовувати атрибути. Погляньмо на такі блоки коду:

### Блок 1

```
<Rectangle Width="100" Height="50" Fill="Red"></Rectangle>
```

**Блок 2**

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
    </Rectangle.Fill>
</Rectangle>
```

Цей код присвоює значення властивості **Fill** прямокутника. Ця складна властивість може бути присвоєна об'єкту типу **SolidColorBrush**. Другий блок допомагає зрозуміти всі використані типи та їх властивості. У першому блоці ми застосовуємо легший спосіб визначення властивості **Fill** за допомогою атрибутів і простої рядка. Це можливо за рахунок конвертації властивостей. У випадку зв'язування даних ви можете створити власний конвертер, що дасть змогу змінювати об'єкт в пам'яті на об'єкт, що може бути присвоєний властивості елемента керування. Наприклад, нижчезаведений клас дає змогу перетворити логічне значення (Boolean) на значення перелічуваного типу **Visibility**:

```
public sealed class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, string language)
    {
        return (value is bool && (bool)value) ?
            Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, string language)
    {
        return value is Visibility && (Visibility)value ==
            Visibility.Visible;
    }
}
```

Створюючи свій елемент керування, ви будете мати можливість використовувати атрибути для визначення властивостей, що набувають рядкового значення. Про це мова піде згодом.

Погляньмо на два еквівалентні блоки коду:

## Блок 1

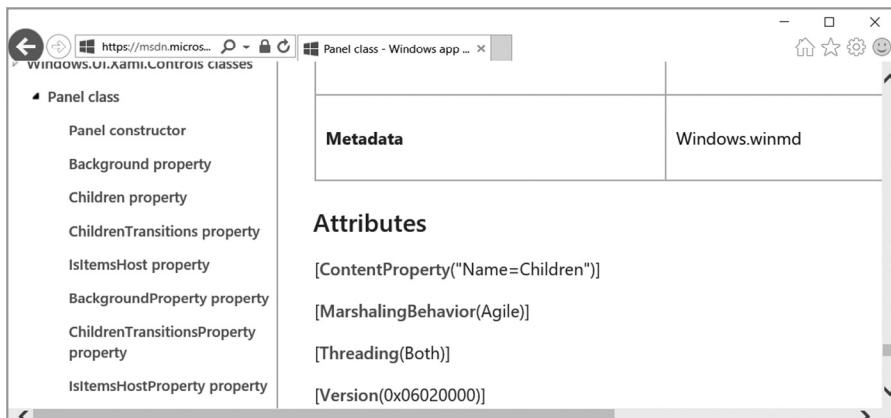
```
<TextBlock>
    Hello
</TextBlock>
```

## Блок 2

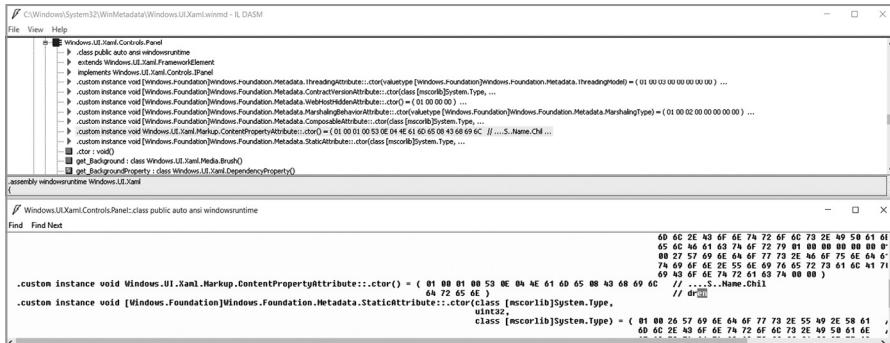
```
<TextBlock Text="Hello"></TextBlock>
```

Ми бачимо, що в першому випадку властивість **Text** було визначено неявно. Це можливо завдяки атрибуту **ContentPropertyAttribute**, що дає можливість визначати вміст елемента керування. Звичайно, кожен елемент керування може мати лише одну властивість вмісту, й у випадку з **TextBlock** це властивість **Text**, а у випадку з **Canvas** це властивість **Children**.

Визначити, яку властивість вмісту було зазначено, можна за допомогою MSDN. У випадку з **Canvas** нам потрібно перевірити базовий клас **Panel** для того, щоб переглянути всі наявні там атрибути.



Крім того, ви можете використати ILDasm і побачити всі метадані у файлах **winmd**. Цей підхід набагато кращий для компонентів сторонніх розробників.



На завершення наведемо приклад визначення колекції за допомогою XAML. Погляньте на код нижче:

```

<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0.0" Color="Red" />
                    <GradientStop Offset="1.0" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>

```

Цей код визначає градієнтний пензель для прямокутника. Очевидно, що **LinearGradientBrush** – складний об'єкт, і неможливо визначити властивість **Fill**, присвоюючи рядкове значення, оскільки для задання градіента необхідно створити колекцію об'єктів **GradientStop**. Але ми можемо спростити цей код.

По-перше, зазначимо, що **GradientStops** – це властивість вмісту об'єкта **LinearGradientBrush**. Таким чином, ми можемо видалити принаймні два рядки XAML:

```

<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStopCollection>
                <GradientStop Offset="0.0" Color="Red" />

```

```
<GradientStop Offset="1.0" Color="Green" />
</GradientStopCollection>
</LinearGradientBrush>
</Rectangle.Fill>
</Rectangle>
```

До того ж аналізатор XAML достатньо розумний для визначення того, що **GradientStops** – це колекція елементів **GradientStop**. Ми можемо модифікувати наш код так:

```
<Rectangle Width="100" Height="50">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Offset="0.0" Color="Red" />
            <GradientStop Offset="1.0" Color="Green" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Цей блок, як і попередній, створює той самий прямокутник, але насправді перший і третій блоки абсолютно різні. У першому варіанті ми створюємо **GradientStopCollection** явно. Ми визначаємо окремий об'єкт в пам'яті і приєднуємо його до властивості **GradientStops** нашого об'єкта **LinearGradientBrush**. Можна додати ім'я для **GradientStopCollection** і отримувати доступ до цієї колекції за допомогою коду C#. В останньому блоці ми не визначаємо об'єкт **GradientStopCollection** взагалі. Замість цього використовуємо існуючу пусту колекцію, що вже була створена всередині **LinearGradientBrush** і додаємо нові члени колекції, використовуючи метод **Add**.

У багатьох випадках ми можемо ініціалізувати колекцію тільки у другий спосіб. Наприклад, код нижче працювати не буде:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <StackPanel.Children>
        <UIElementCollection>
            <Button Width="100" Height="50"></Button>
        </UIElementCollection>
    </StackPanel.Children>
</StackPanel>
```

Цей код не працює, тому що властивість **Children** у класі **Panel** не має загально-доступного методу **set** для надання значення. Саме з цієї причини немає можли-

вості створити нову колекцію елементів і присвоїти їх властивості **Children**. Розробники у такому випадку можуть використовувати метод **Add** (з **ICollection**), що додає новий елемент до наявної колекції. Другий підхід працюватиме так:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <StackPanel.Children>
        <Button Width="100" Height="50"></Button>
    </StackPanel.Children>
</StackPanel>
```

Звичайно, ми можемо спростити цей код, відповідно використовуючи властивість **вмісту**:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Button Width="100" Height="50"></Button>
</StackPanel>
```

## Простори імен в XAML

Для розробників програм на C# простір імен – це фундаментальне поняття. Завдяки просторам імен можна створювати логічні контейнери, що містять вибрані класи. Наприклад, клас **Button** розташований у просторі імен **Windows.UI.Xaml.Controls**, де можна знайти інші класи, які визначають елементи керування для створення користувацьких інтерфейсів. Повна назва класу **Button** – це **Windows.UI.Xaml.Controls.Button**, але завдяки ключовому слову **using** розробники можуть зазначати простори імен у коді й уникати використання повних назв класів.

XML підтримує простори імен, як і C#. Але XML має дві важливі відмінності від C#. По-перше, потрібно використовувати уніфіковані ідентифікатори ресурсів (URI) для того, щоб задати простір імен. У деяких випадках це можуть бути мережеві уніфіковані ідентифікатори ресурсів (web uri), але за наявності власних класів використовуємо **using** uri, як у C#. Друга відмінність – це те, що немає можливості застосовувати більше одного простору імен за замовчуванням. Для решти просторів імен можна лише вказати ім'я, яке потрібно використати в XAML, щоб отримати доступ до елементів керування.

Можна вказати простір імен за замовчуванням чи за назвою завдяки атрибуту **xmlns**, який зазвичай розміщують у кореневому елементі документа. Наприклад, простір імен **my** описує такий код:

```
xmlns:my="http://baydachnyy.com/schemas"
```

Зазвичай за замовчуванням елемент **Page** містить п'ять просторів імен:

```
<Page
    x:Class="App33.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:App33"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
    markup-compatibility/2006"
    mc:Ignorable="d">
```

Перші два – це стандартні простори імен, які можна знайти у будь-якій програмі WPF, Silverlight або Windows Runtime. Завдяки ним розробники можуть користатися стандартними елементами керування для опису інтерфейсу користувача (UI). Зазвичай розробники використовують величезну кількість стандартних елементів, що вказані в першому просторі імен за замовчуванням.

Третій простір імен використовує **using**, щоб надати доступ до класів, які можна створити всередині вашої програми. Загалом це не потрібно, оскільки розробники люблять створювати додаткові простори імен усередині, але цей код наведено для зразка.

Останні два простори імен дають можливість застосовувати об'єкти й атрибути, що впливають на інтерфейс тільки в режимі дизайну. Це дуже важливо, коли ви використовуєте зв'язування даних, але хочете подивитися, як виглядає інтерфейс в режимі дизайну.

## Пишемо код для сторінок і обробників подій

Ми щойно опанували основи XAML-синтаксису. Час дізнатись, як поєднувати його із C#. Відкриємо будь-який XAML-код і перегляньмо елемент сторінки:

```
<Page
    x:Class="App33.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:App33"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/
    markup-compatibility/2006"
mc:Ignorable="d">

```

Видно, що елемент **Page** містить атрибут **x:Class**, завдяки якому можна оголосити клас у коді програмної частини (code-behind) для **Page**. Якщо ваша сторінка – це файл **MainPage.xaml**, клас у коді програмної частини можна знайти у файлі **MainPage.xaml.cs**. Завдяки атрибуту **x:Class** у файлі XAML ви можете використати будь-які обробники подій з файлу C#. Звичайно, цього недостатньо, тому що потрібно отримувати доступ з XAML-файлу до C# також. Додамо кнопку до сторінки:

```
<Button Name="myButton" Width="100" Height="50">Hello</Button>
```

Щойно кнопку буде додано, ви зможете відкрити файл **MainPage.xaml.cs** і побачити такий код:

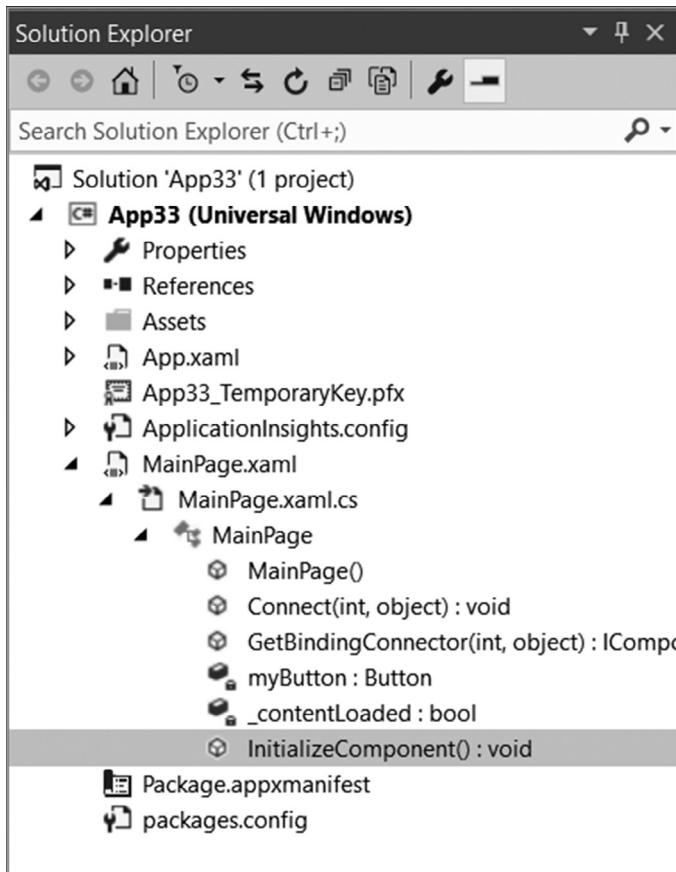
```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        InitializeComponent();
    }
}

```

Особливу увагу зверніть на те, що клас було визначено як **partial** і він містить виклик методу **InitializeComponent**. Водночас кнопки **myButton** не видно, але якщо ви спробуєте використати цю змінну, Visual Studio навіть підкаже її назву за допомогою засобу IntelliSense і даст змогу із нею працювати. Це можливо завдяки тому, що Visual Studio стежить за всіма змінами в XAML і вносить відповідні зміни в код програмної частини. Visual Studio використовує часткові класи, щоб уникнути «сміття», яке може заважати розробникам.

Погляньмо на клас **MainPage** в Solution Explorer:



Помітно, що клас **MainPage** містить декілька методів і полів, зокрема **myButton** та **InitializeComponent**. Якщо ви натиснете **myButton**, Visual Studio переспрямує вас до іншої частини класу **MainPage** – у **MainPage.g.i.cs**:

```
partial class MainPage : global::Windows.UI.Xaml.Controls.Page
{
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "Microsoft.Windows.UI.Xaml.Build.Tasks", " 14.0.0.0")]
    private global::Windows.UI.Xaml.Controls.Button myButton;
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "Microsoft.Windows.UI.Xaml.Build.Tasks", " 14.0.0.0")]
    private bool _contentLoaded;
```

```
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "Microsoft.Windows.UI.Xaml.Build.Tasks", " 14.0.0.0")]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
public void InitializeComponent()
{
    if (_contentLoaded)
        return;
    _contentLoaded = true;
    global::System.Uri resourceLocator =
        new global::System.Uri("ms-appx:///MainPage.xaml");
    global::Windows.UI.Xaml.Application.LoadComponent(
        this, resourceLocator,
        global::Windows.UI.Xaml.Controls.Primitives.
            ComponentResourceLocation.Application);
}
}
```

Там ви знайдете поле **myButton**, а також метод **LoadComponent**, що допомагає будувати логічне дерево XAML.

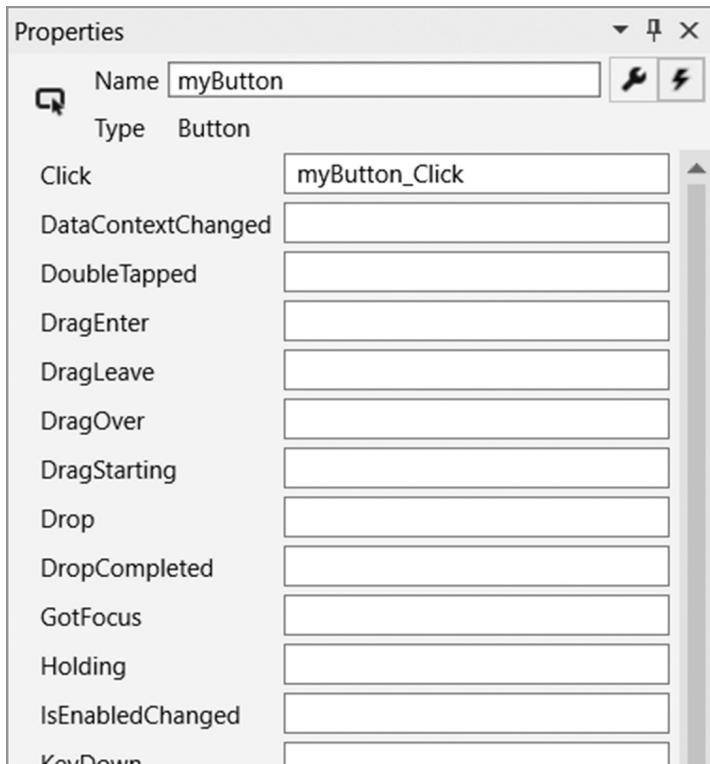
Перейдімо до опису обробників подій у XAML. Це просто: потрібно лише використовувати атрибути для визначення подій і значень, таких як ім'я методу, що оброблятиме подію.

```
<Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">
    <Button Name="myButton" Width="100"
        Height="50" Click="myButton_Click">Hello</Button>
</Grid>
```

Якщо ви вже визначили обробник події у коді C#, можна обрати метод за допомогою Visual Studio, що, використовуючи засіб IntelliSense, пропонує всі методи з потрібними прототипами. Крім того, можна надіслати Visual Studio запит на створення нового методу й скористатися **Peek Definition** або **Go To Definition** для того, щоб перейти до методу в cs-файлі.



Окрім того, можна скористатися вікном властивостей для того, щоб переглядати всі наявні події та обробники. Для цього перейдіть до вкладки подій:



Нарешті, якщо натиснути кнопку у вікні дизайну, Visual Studio також згенерує обробник події **Click**, тому що це стандартна подія в режимі дизайну.

## Розширення розмітки

Ми вже знаємо, як оголосити базовий інтерфейс і як використовувати в XAML обробники подій із C# і отримати доступ з C# до XAML-елементів. Проте багато сценаріїв не може бути реалізовано за наявних можливостей XML. Отже, XAML розширює XML і вводить розширення розмітки. Основною метою такого розширення є можливість використовувати додаткові асоціації між об'єктами C# і XAML, а також між різними елементами в XAML. Так, завдяки розширенням розмітки можна присвоювати значення атрибутів у нестандартний спосіб. Подивімось, які розширення XAML доступні для розробників Windows 10:

- **Binding** – дає змогу пов'язувати XAML-властивості з будь-яким об'єктом у момент виконання. Ви можете використовувати для посилання об'єкти, що створені в пам'яті, або XAML-об'єкти.
- **StaticResource** – завдяки цьому розширенню можна використовувати стилі, шаблони і об'єкти, які ви визначили в ресурсах.
- **TemplateBinding** – дає змогу пов'язати елементи керування з шаблонами, визначеними у ресурсах.
- **ThemeResource** – це розширення подібне до **StaticResource**, але реалізує додаткову логіку, завдяки якій можна вибрати ресурс на підставі поточної теми.
- **RelativeSource** – дає змогу визначити джерело зв'язування у поняттях реляційних відношень.
- **x:Bind** – нове розширення, що дає можливість створити зв'язування як Binding-розширення, але це статичне зв'язування, яке дає змогу підвищити продуктивність програм. Воно може бути використане, якщо ваші об'єкти строго типізовано.
- **CustomResource** – ви можете використовувати це розширення, якщо маєте власне сховище ресурсів і хочете реалізувати власну логіку.

У наведеному нижче прикладі показано, як використати розширення розмітки **StaticResource**, щоб застосувати до кнопки **myButton** стиль **btnStyle** зі сторінки ресурсів:

```
<Page.Resources>
    <Style x:Name="btnStyle" TargetType="Button">
        <Setter Property="Background" Value="Green"></Setter>
    </Style>
</Page.Resources>
<Grid x:Name="LayoutRoot" Background="White">
    <Button Name="myButton" Width="100" Height="50"
           Style="{StaticResource btnStyle}" Content="Hello">
    </Button>
</Grid>
```

Далі в блоці коду показано, як працювати з розширенням розмітки **x:Bind**. У цьому прикладі ми зв'язуємо властивість **Width** елемента керування **TextBox** із властивістю **Value** елемента керування **Slider**.

```
<StackPanel x:Name="LayoutRoot" >
    <Slider Name="sld1" Width="300" Height="50"
           Minimum="100" Maximum="200" Value="10">
    </Slider>
    <TextBox Text="Hello"
```

```
    Width="{x:Bind sld1.Value, Mode=TwoWay}">
</TextBox>
</StackPanel>
```

Детальніше про всі наявні розширення розмітки піде мова в наступних розділах.

## Властивості залежностей

Як приклад наведемо такий код:

```
<Canvas>
    <Button Content="Hello" Canvas.Top="60" Canvas.Left="120"
        Width="110" Height="40" Name="button1" />
</Canvas>
```

Із цього видно, що два атрибути елемента **Button** мають не такий вигляд, як інші. Ці атрибути – **Canvas.Top** і **Canvas.Left**, що не належать до класу **Button** взагалі. Загалом **Button** не повинен знати про можливі контейнери. Тож як ми можемо визначити дві властивості, що належать **Canvas**? Це можливо завдяки класам **DependencyObject** і **DependencyProperty**. Перший задає методи **SetValue** і **GetValue**, що отримують об'єкти **DependencyProperty** як параметри. Головна ідея цих методів – утримувати динамічний список об'єктів **DependencyProperty** всередині кожного користувацького елемента керування. Цей список допомагає визначати нові елементи в наведеному вище переліку (**Canvas.Top**). Можна також присвоювати залежності безпосередньо з коду C#:

```
button1.SetValue(Canvas.TopProperty, 60)
```

Щойно **Canvas** розмістить кнопку у правильному місці, для отримання значення властивостей **Top** і **Left** буде застосовано метод **GetValue**. Ми створюватимемо властивості залежностей, коли обговорюватимемо користувацькі елементи керування.

Отже, ми розглянули всі основні особливості XAML, і настав час починати використовувати XAML для побудови справжніх інтерфейсів. У наступних розділах ми обговоримо контейнери й основні користувацькі елементи керування, а також матимемо змогу більше попрактикуватися в XAML.

Розділ 3.

## **Розмітка**

Розпочнімо наш огляд із наявних спеціальних елементів керування, що дають можливість створити розмітку. Цей важливий набір елементів керування визначає розміщення усіх інших елементів керування та вигляд інтерфейсу вашої програми. Створення кожної нової сторінки розробники починають із розмітки.

Ви вже знаєте, що всі сторінки XAML містять кореневий елемент **Page**. Клас **Page** успадковується від **UserControl**, що містить властивість **Content**. Завдяки властивості **Content** ви можете розмістити будь-який елемент **UIElement** всередині сторінки, але не можете додавати безпосередньо більше одного елемента. Тому створення розмітки починається з контейнера, що міститиме всі інші елементи керування та інші контейнери.

Майже неможливо створити інтерфейс всередині лише одного контейнера або використовувати для цього контейнери лише одного типу. Саме тому вам необхідно буде ознайомитися з усіма типами контейнерів і, найважливіше, усвідомити, що саме ви маєте намір реалізувати. Головний принцип полягає в тому, щоб спроектувати дизайн усіх сторінок до того, як почнете власне розробку.

У цьому розділі ми обговоримо контейнери, що використовуються у просторі імен **Windows.UI.Xaml.Controls** і успадковані від класу **Page**: **Canvas**, **StackPanel**, **Grid** і **RelativePanel**.

## Canvas

**Canvas** це «найгірша» розмітка, тому що дозволяє розміщувати внутрішні елементи керування із використанням абсолютнох позицій всередині контейнера. Це порушує сучасну концепцію програм для Windows 10 – адаптивний інтерфейс. Якщо задати нерухомі позиції елементів керування, це перешкоджатиме внесенню змін у разі, якщо користувач змінить розмір екрана чи поверне пристрій із горизонтального у вертикальне положення.

Проте існує кілька сценаріїв, коли цей контейнер справді корисний. Наприклад, у розробці простої гри, де розміщення елементів є критично важливим, можна використати **Canvas** разом із **Viewbox**. **Canvas** дає можливість малювати будь-що, а за допомогою **Viewbox** можна змінювати **Canvas** відповідно до розмірів вікна.

Нижче наведено приклад того, як задавати елемент керування **Canvas** і кнопку всередині. Щоб розмістити елемент всередині **Canvas**, слід використовувати властивості **Canvas.Top** і **Canvas.Left**.

```
<Canvas>
    <Button Content="Hello" Canvas.Top="100" Canvas.
        Left="100"></Button>
</Canvas>
```

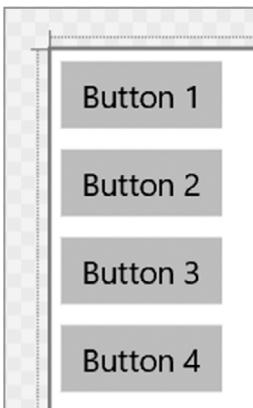
Зауважте, що не варто використовувати цей контейнер без вагомих причин.

## StackPanel

Однією з найпростіших розміток є **StackPanel**. Ця розмітка розташовує всі елементи керування в одному рядку чи стовпці:

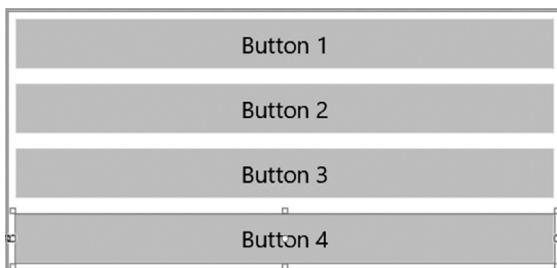
```
<StackPanel>
    <Button Content="Button 1"></Button>
    <Button Content="Button 2"></Button>
    <Button Content="Button 3"></Button>
    <Button Content="Button 4"></Button>
</StackPanel>
```

Якщо запустити цей код, StackPanel відобразить чотири кнопки, розташовані одна під одною.



Це орієнтація за замовчуванням для **StackPanel**. Не зважаючи на те, що **StackPanel** заповнює весь простір, кнопки не займають його повністю, а ви можете легко вирівняти їх до ширини **StackPanel**, застосувавши властивість **HorizontalAlignment**:

```
<StackPanel>
    <Button Content="Button 1"
        HorizontalAlignment="Stretch"></Button>
    <Button Content="Button 2"
        HorizontalAlignment="Stretch"></Button>
    <Button Content="Button 3"
        HorizontalAlignment="Stretch"></Button>
    <Button Content="Button 4"
        HorizontalAlignment="Stretch"></Button>
</StackPanel>
```



Значення **Stretch** вказує, що кнопки повинні заповнювати все доступне місце всередині контейнера. Якщо **StackPanel** має горизонтальну орієнтацію, можна використати властивість **VerticalAlignment** і значення **Stretch**. Додатково можна застосувати значення **Center**, **Right** і **Left** для властивості **HorizontalAlignment** і **Top**, **Bottom** і **Center** – для **VerticalAlignment**.

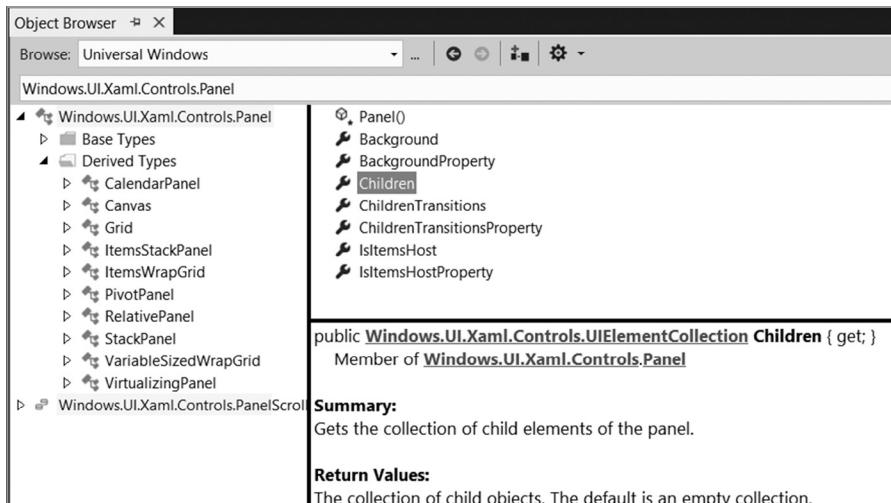
Щоб змінити розміщення елементів керування всередині **StackPanel**, можна використати властивість **Orientation**:

```
<StackPanel Orientation="Horizontal">
    <Button Content="Button 1"></Button>
    <Button Content="Button 2"></Button>
    <Button Content="Button 3"></Button>
    <Button Content="Button 4"></Button>
</StackPanel>
```

## Основні властивості та клас Panel

Під час роботи зі **StackPanel** використовуються дві властивості класу **Button**: **HorizontalAlignment** і **VerticalAlignment**. Звичайно, вони не можуть задоволінити

всі потреби, а поширені елементи керування додатково містять багато різних властивостей, які розробники можуть використовувати, щоб коригувати розміщення в контейнері. Перед оглядом цих властивостей відкриймо клас **Panel** в Object Browser у Visual Studio:



Як бачимо, усі класи для стандартних контейнерів, таких як **StackPanel**, **Canvas**, **Grid**, **RelativeLayout**, успадковуються від класу **Panel**. Цей клас має особливу властивість, що називається **Children**. Уже згадувалося, що в разі, коли розробник розміщує елемент керування в контейнері, це впливає на колекцію **Children**.

Зазвичай колекція **Children** використовується для того, щоб додавати нові елементи керування до контейнерів із C#. Давайте створимо пустий **StackPanel**:

```
<StackPanel Name="myPanel" />
```

Ми використали властивість **Name**, щоб отримати доступ до панелі із C#. Для того, щоб додати кнопку із C#, можна модифікувати наявний конструктор для головної сторінки програми:

```

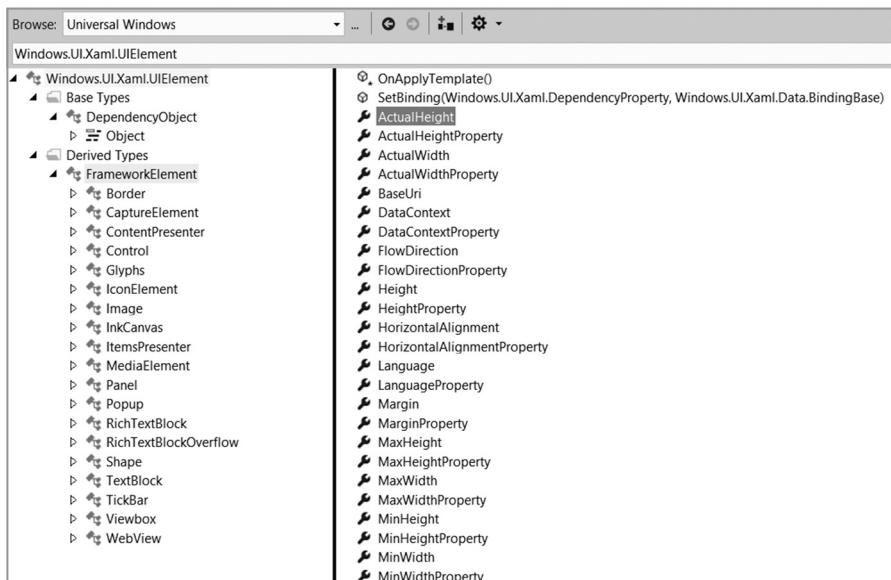
public MainPage()
{
    this.InitializeComponent();

    myPanel.Children.Add(new Button() { Content = "Hello" });
}

```

Якщо запустити цей код, буде видно, що панель містить кнопку зі стандартними властивостями.

Давайте поглянемо на властивість **Children**. Це колекція, що може містити об'єкти **UIElement**. Час ще раз відкрити Object Browser:



- OnApplyTemplate()
- SetBinding(Windows.UI.Xaml.DependencyProperty, Windows.UI.Xaml.Data.BindingBase)
- ActualHeight
- ActualHeightProperty
- ActualWidth
- ActualWidthProperty
- BaseUri
- DataContext
- DataContextProperty
- FlowDirection
- FlowDirectionProperty
- Height
- HeightProperty
- HorizontalAlignment
- HorizontalAlignmentProperty
- Language
- LanguageProperty
- Margin
- MarginProperty
- MaxHeight
- MaxHeightProperty
- MaxWidth
- MaxWidthProperty
- MinHeight
- MinHeightProperty
- MinWidth
- MinWidthProperty

Завдяки Object Browser ми знаємо, що **DependencyObject** є базовим класом **UIElement**. Це дуже важливий тип взаємозв'язків, але **DependencyObject** не містить нічого суттєвого для розмітки. **UIElement** має багато інших властивостей, але вони також не впливають на розмітку. Варто взяти до уваги лише властивість **Visibility**, що дає можливість видалити елемент керування з візуального дерева XAML, якщо має значення **Collapsed**. Тут видно, що **UIElement** має лише один похідний тип – **FrameworkElement**. Якщо ми працюємо зі стандартними елементами керування, це гарантує, що всі елементи керування мають **FrameworkElement** у своїй ієрархії й усі вони мають властивості, які має **FrameworkElement**. Найважливішими властивостями для розміток є:

- **Margin** – визначає простір між елементом керування та контентом, розташованим ззовні. **Margin** можна визначити, використовуючи лише одне число. У цьому разі між будь-якими елементами керування буде одна-кова відстань. Або можна визначити відстань від кожної сторони. **Margin** можна визначити за допомогою редактора властивостей у Visual Studio чи в самому XAML:

```
<Button Margin="10" Content="Hello"></Button>
<Button Margin="10, 5, 20, 5" Content="Hello"></Button>
```

- **MinWidth** – визначає мінімальне значення ширини елемента керування.
- **MinHeight** – визначає мінімальне значення висоти елемента керування.
- **MaxWidth** – визначає максимальне значення ширини елемента керування.
- **MaxHeight** – визначає максимальне значення висоти елемента керування.

Можна також використовувати властивості **Width** і **Height**, але цього варто уникати.

Крім того, у класах **StackPanel**, **Grid** і **RelativePanel** можна знайти властивість **Padding**. Вона дає змогу встановити відстань між межами панелі та контентом всередині. Оскільки властивість **Padding** по-різному реалізується для різних контейнерів, і деякі контейнери, такі як **Canvas**, взагалі не мають властивості **Padding**, розробники з Microsoft не включили її до класу **Panel**.

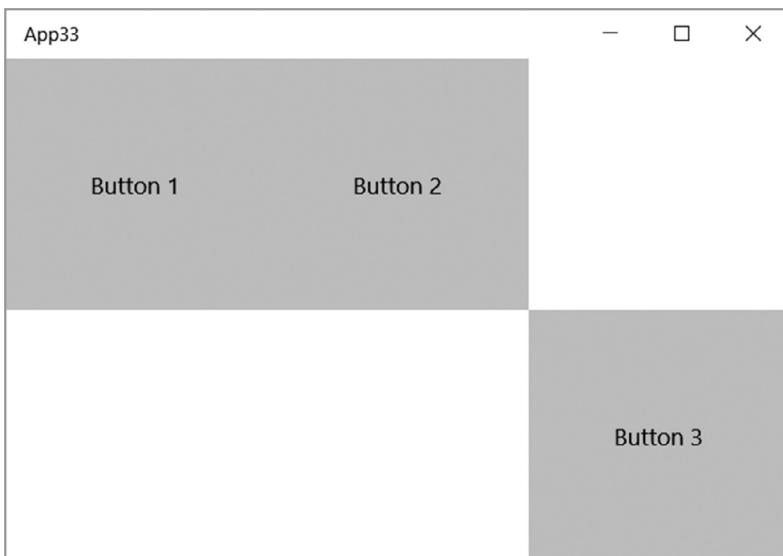
## Grid

Наступним контейнером, який ми розглянемо, є **Grid**. Він є найпотужнішим і за його допомогою можна створювати дуже складні розмітки. Завдяки **Grid** розробники можуть розділити увесь доступний простір на комірки, визначивши кількість рядків і стовпців. Як тільки стовпці і рядки задано, елементи керування можна розміщувати всередині, використовуючи рядки та стовпці як властивості залежностей для **Grid**. Нижченаведений код демонструє, як задавати рядки та стовпці й розміщувати в них елементи керування:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Button Content="Button 1" Grid.Row="0" Grid.Column="0"
        HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch"></Button>
```

```
<Button Content="Button 2" Grid.Row="0" Grid.Column="1"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"></Button>
<Button Content="Button 3" Grid.Row="1" Grid.Column="2"
    HorizontalAlignment="Stretch"
    VerticalAlignment="Stretch"></Button>
</Grid>
```

У цьому прикладі визначено два рядки та три стовпці й розміщено три кнопки всередині:



За замовчуванням **Grid** ділить весь доступний простір на рядки та стовпці рівномірно, але можна й задати властивості **Width** і **Height** для стовпців і рядків відповідно. Існує декілька способів це зробити:

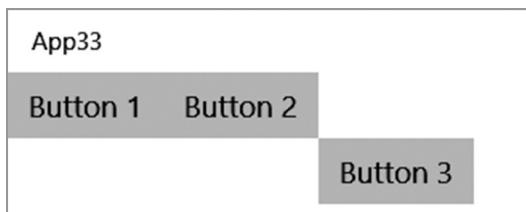
- Розробники можуть вказати розмір стовпців і рядків за допомогою ефективних пікселів. Цей підхід застосовується для визначення заголовків і колонтитулів або будь-якого елемента керування, що має фіксовану висоту й ширину.
- Кращий спосіб для визначення стовпців і рядків у разі адаптивного інтерфейсу – це використання значення **Auto** або пропорцій. У разі застосування **Auto** контейнер **Grid** надасть стільки місця для контенту, скільки буде необхідно. Нижче подано приклад.

```

<Grid >
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Button Content="Button 1" Grid.Row="0" Grid.Column="0"
        HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch"></Button>
    <Button Content="Button 2" Grid.Row="0" Grid.Column="1"
        HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch"></Button>
    <Button Content="Button 3" Grid.Row="1" Grid.Column="2"
        HorizontalAlignment="Stretch"
        VerticalAlignment="Stretch"></Button>
</Grid>

```

Як бачимо, зараз наш інтерфейс набагато компактніший, і властивості **HorizontalAlignment** і **VerticalAlignment** не спрацьовують взагалі, оскільки **Grid** задає місце для кнопок з огляду на їх стандартну ширину та довжину.



- У разі пропорційних значень можна використовувати такий приклад:

```

<Grid Background="{StaticResource
    ApplicationPageBackgroundBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*"/></RowDefinition>
        <RowDefinition Height="2*"/></RowDefinition>
    </Grid.RowDefinitions>

```

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/></ColumnDefinition>
    <ColumnDefinition Width="2*"/></ColumnDefinition>
    <ColumnDefinition Width="2*"/></ColumnDefinition>
</Grid.ColumnDefinitions>
<Button Content="Button 1" Grid.Row="0" Grid.Column="0">
</Button>
<Button Content="Button 2" Grid.Row="0" Grid.Column="1">
</Button>
<Button Content="Button 3" Grid.Row="1" Grid.Column="2">
</Button>
</Grid>
```

У цьому прикладі контейнер **Grid** поділяє весь доступний простір для рядків на три частини; одна із них відповідатиме першому рядку, а дві інші – другому. Що стосується стовпців, **Grid** поділяє весь простір на п'ять частин: одна частина для першого стовпця і по дві – для другого і третього.

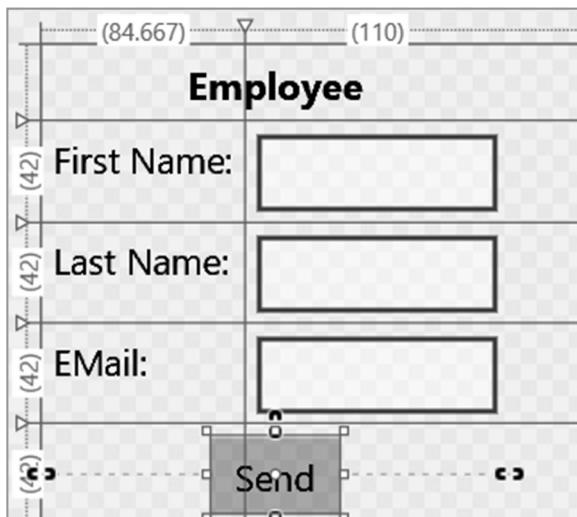
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition Height="Auto"/></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/></ColumnDefinition>
        <ColumnDefinition Width="Auto"/></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <TextBlock Text="Employee" Grid.Row="0" Grid.Column="0"
        Grid.ColumnSpan="2" HorizontalAlignment="Center"
        FontSize="16" FontWeight="Bold" Margin="5">
    </TextBlock>
    <TextBlock Text="First Name:" Grid.Row="1"
        Grid.Column="0" Margin="5"></TextBlock>
    <TextBox Grid.Column="1" Grid.Row="1" MinWidth="100"
        Margin="5"></TextBox>
    <TextBlock Text="Last Name:" Grid.Row="2" Grid.Column="0"
        Margin="5"></TextBlock>
    <TextBox Grid.Column="1" Grid.Row="2" MinWidth="100"
        Margin="5"></TextBox>
```

```

<TextBlock Text="EMail:" Grid.Row="3" Grid.Column="0"
Margin="5"></TextBlock>
<TextBox Grid.Column="1" Grid.Row="3" MinWidth="100"
Margin="5"></TextBox>
<Button Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
Content="Send" Margin="5"
HorizontalAlignment="Center">
</Button>
</Grid>

```

У цьому коді ми використали властивість **ColumnSpan**, що об'єднує два стовпці в рядку, де розташовано елемент керування. Так само можна використовувати властивість **RowSpan** для об'єднання рядків. Після запуску цього коду з'явиться така форма:

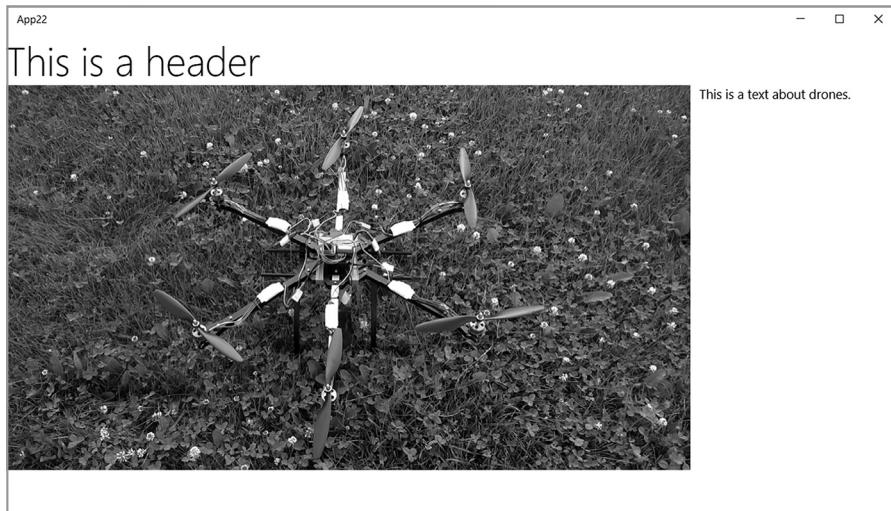


## RelativePanel

Останній (не за значущістю) елемент керування для розмітки – це **RelativePanel**. Це новий елемент керування, що був представлений у Windows 10. Він застосовується для створення адаптивних інтерфейсів. Його використання ілюструє такий приклад:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/></RowDefinition>
        <RowDefinition Height="*"/></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/></ColumnDefinition>
        <ColumnDefinition Width="*"/></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <TextBlock Text="This is a header" Grid.ColumnSpan="2"
        Style="{ThemeResource HeaderTextBlockStyle}"></TextBlock>
    <Image MaxWidth="800" Grid.Row="1"
        Source="Assets\drone.jpg" VerticalAlignment="Top"></Image>
    <TextBlock Text="This is a text about drones."
        Margin="10,0,10,0"
        TextWrapping="Wrap" Grid.Column="1" Grid.Row="1">
    </TextBlock>
</Grid>
```

У цьому прикладі було вибрано просту розмітку, що використовує таблицю для розміщення двох текстових блоків і зображення. Після запуску коду відобразиться малюнок на кшталт такого.



Однак Windows 10 дає змогу змінювати розмір вікна, а ця програма матиме мобільну версію. Як легко побачити, на пристроях із меншим екраном зображення й текст будуть відображатися не повністю. Оскільки нам потрібен універсальний інтерфейс, слід змінювати розмітку динамічно, якщо не вистачатиме місця для її відображення. Нам необхідно розмістити текст під зображенням, а для цього найкраще використати **VisualStateManager** (див. детальніше про цей елемент керування у наступному розділі), що дає змогу змінювати інтерфейс під час виконання за допомогою тригерів або коду C#. Досі ми використовували контейнер **Grid**, що містить два стовпці та два рядки, а для меншого екрана потрібна таблиця з одним стовпцем і трьома рядками. Це майже неможливо реалізувати у **VisualStateManager** навіть для нашого прикладу, а справжні інтерфейси ще складніші. Саме тому, коли розробники будують універсальні програми для Windows 8.1, вони дублюють деякі частини інтерфейсу й працюють із властивістю **Visibility** всередині **VisualStateManager**.

Починаючи з Windows 10 розробники мають можливість використовувати новий контейнер **RelativePanel**, який дає можливість уникнути проблем із **Grid**. Давайте розглянемо той самий приклад усередині **RelativePanel**.

```
<RelativePanel >
    <TextBlock Text="This is a header" Name="header"
        Style="{ThemeResource HeaderTextBlockStyle}"></TextBlock>
    <Image Name="image" MaxWidth="800"
        RelativePanel.Below="header"
        Source="Assets\drone.jpg"></Image>
    <TextBlock Text="This is a text about drones."
        Margin="10,0,10,0"
        TextWrapping="Wrap" RelativePanel.AlignTopWith="image"
        RelativePanel.RightOf="image" Name="text"></TextBlock>
</RelativePanel>
```

Як бачимо, в разі використання контейнера **RelativePanel** можна задавати позиції внутрішніх елементів керування за допомогою властивостей залежностей. Оскільки це лише властивості, легко переформатувати розмітку, використовуючи їх у **VisualStateManager**. Наприклад, можна скористатися наведеним нижче кодом, щоб відобразити той самий інтерфейс в одному рядку:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="Normal">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="900">
                    </AdaptiveTrigger>
                </VisualState.StateTriggers>
            </VisualState>
            <VisualState x:Name="Mobile">
                <VisualState.Setters>
                    <Setter Value="" Target="text.(RelativePanel.AlignTopWith)">
                    </Setter>
                    <Setter Value="image" Target="text.(RelativePanel.Below)"></Setter>
                    <Setter Value="" Target="text.(RelativePanel.RightOf)">
                    </Setter>
                    <Setter Value="0,10,0,10" Target="text.Margin">
                    </Setter>
                </VisualState.Setters>
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0">
                        </AdaptiveTrigger>
                </VisualState.StateTriggers>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

Як тільки роздільна здатність екрана стає меншою за 900 пікселів, текст відображається під картинкою. Особливу увагу слід приділити заданню властивостей залежностей в атрибуті **Target**.

Радимо використовувати цю розмітку; на сьогодні **RelativeLayout** є найпопулярнішою розміткою для створення універсальних інтерфейсів.

## ScrollView

Наступний елемент керування не стосується розмітки безпосередньо, але розглянемо його в цьому розділі, щоб охопити всю тему розмітки. Щоб побачити, як працює **ScrollView**, розмістимо багато звичайних елементів керування всередині **StackPanel**:

```
<StackPanel Orientation="Horizontal">
    <Button Content="Hello"></Button>
    . . .
</StackPanel>
```

Якщо додати достатню кількість елементів керування, стане видно, що **StackPanel** не має прокрутки і деякі елементи керування потрапляють за межі екрана, тож їх неможливо використовувати. Для додавання прокрутки всередині **ScrollView** потрібно розмістити елемент **StackPanel**:

```
<ScrollView HorizontalScrollMode="Enabled"
    HorizontalScrollBarVisibility="Auto"
    VerticalScrollBarVisibility="Hidden">
    <StackPanel Orientation="Horizontal">
        <Button Content="Hello"></Button>
        . . .
    </StackPanel>
</ScrollView>
```

Елемент керування **ScrollView** – це не просто рядок прокрутки. Завдяки ньому можна застосувати до контенту масштабування. Масштабування виконується із використанням таких властивостей:

- **ZoomMode** – активує функціональність масштабування;
- **MaxZoomFactor** – задає максимальний поріг наближення;
- **MinZoomFactor** – задає мінімальний поріг наближення;
- **ZoomSnapPoints** – задає опорні точки для реалізації покрокового масштабування (1x, 1,5x, 2,0x тощо).

На цьому завершимо розгляд розмітки та перейдемо до огляду поширеніх елементів керування.

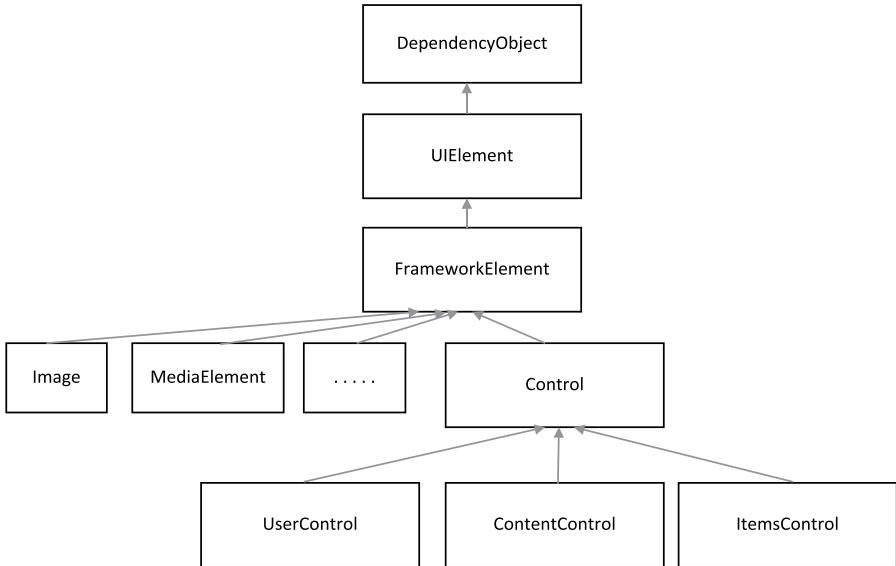


Розділ 4.

## **Поширені елементи керування**

## Декілька слів про ієрархічну структуру

Перш ніж почати працювати з елементами керування, розглянемо основні класи та спробуємо зрозуміти їхню ієрархічну структуру. Звичайно, краще відкрити Object Browser у Visual Studio і переглянути загальну ієрархічну структуру для будь-якого користувачького елемента керування. Проте цей підхід ми застосуємо пізніше, а зараз розгляньмо це зображення:



Тут задля скорочення не показано клас **System.Object**, оскільки зрозуміло, що для всіх класів у C#/NET Framework він є базовим класом, розташованим на вершині ієрархії.

Першим важливим класом на нашому малюнку є **DependencyObject**. Він не містить спеціальних властивостей для решти елементів керування, але завдяки цьому класу всі елементи керування можуть мати властивості залежностей. Пізніше ми поговоримо про користувачькі елементи керування і побачимо, наскільки важливим для розробників є клас **DependencyObject**.

Наступним класом є **UIElement**. Він описує події, властивості й методи основних типів введення даних, зокрема перетягування за допомогою миші, роботи із вказівником, введення з клавіатури та містить інфраструктуру для додаткових обробників подій. Ви не застосовуватимете цей клас напряму. Навіть серед стандартних класів в Universal Windows Platform можна знайти лише один, **FrameworkElement**, для якого **UIElement** є базовим класом. Окрім того,

**UIElement** описує декілька загальних властивостей, наприклад **Opacity** і **Visibility**, але більшість загальних властивостей визначені в класі **FrameworkElement**.

Наступним за ієрархією класом є **FrameworkElement**. Завдяки йому всі користувацькі елементи керування підтримують зв'язування даних, стилі та ресурси. Крім того, цей клас має деякі загальні властивості, наприклад **Width**, **Height**, **VerticalAlignment** і **HorizontalAlignment**. Таким чином, **FrameworkElement** має усе необхідне для подання у візуальному дереві XAML. Як видно, деякі користувацькі елементи керування успадковуються безпосередньо з **FrameworkElement**, наприклад **InkCanvas**, **Image**, **TextBlock**, **WebView** і **Panel**. Ці елементи відображають специфічний вміст і часом навіть не мають візуального подання. Якщо користувацький елемент керування успадковано безпосередньо від **FrameworkElement**, ви не матимете можливості змінити шаблон елемента керування, і він не підтримуватиме різні стани.

Далі за ієрархією розташовано клас **Control**. Завдяки цьому класу користувацькі елементи керування можуть мати стані і шаблони. Шаблони визначають вигляд і взаємодію елементів керування в XAML, відокремленому від коду C#. Це дає змогу змінювати візуальне подання будь-якого елемента керування, що підтримує шаблони (оскільки успадкований від **Control**). Користувацькі елементи керування, успадковані від **Control**, можуть бути активні чи неактивні, тож є змога опанувати цілком новий спосіб взаємодії з підтримкою різних станів. Ми обговоримо шаблони в окремому розділі, де покажемо, як створити власні майже нові елементи керування, просто змінюючи шаблони навіть без програмування на C#.

Клас **Control** – це базовий клас для багатьох різних елементів керування, наприклад **Hub**, **TextBox**, **TimePicker**. Є ще три класи, що успадковуються від **Control** і розширяють його функціональність: **UserControl**, **ContentControl** і **ItemsControl**. Перший ми використовуватимемо для створення власних складених елементів керування, що побудовані на основі наявного набору користувацьких елементів керування. Клас **ContentControl** є базовим для всіх користувацьких елементів керування, що можуть містити певний елемент контенту. В цьому разі можна застосовувати будь-який контент і власний шаблон. Крім того, **ItemsControl** є базовим для всіх користувацьких елементів керування, що працюють із колекціями. Він дає змогу визначити шаблон для елемента, встановити контейнер для елементів і виконати зв'язування з колекціями.

Детальніше про деякі властивості користувацьких елементів керування буде розказано далі.

## Кнопки

Усі кнопки в Universal Windows Platform успадковано від класу **BaseButton**, що в свою чергу є нащадком **ContentControl**. Ось скріншот з Object Browser у Visual Studio:



Тут перелічено такі 8 кнопок:

- **Button** – класична кнопка. Зазвичай має вигляд прямокутника з текстом усередині, але цей шаблон за бажання можна змінити. Подія, що найчастіше трапляється, – **Click**.
- **AppBarButton** – хороший приклад того, як можна змінити вигляд і взаємодію класичної кнопки, використовуючи шаблони та розширюючи функціональність. **AppBarButton** – це та сама кнопка, але з додатковими властивостями, а її вигляд і функції відповідають основним принципам дизайну програм UWP.
- **HyperlinkButton** – за замовчуванням ця кнопка має вигляд звичайного гіперпосилання в браузері. Зазвичай її використовують для переходу між сторінками всередині програми.
- **RepeatButton** – це та сама кнопка, але якщо її натиснути, вона генеруватиме подію **Click** постійно. Інтервал між подіями **Click** установлюють у мілісекундах.
- **ToggleButton** – це кнопка, що має два стани: **Checked** і **Unchecked**. Після кожного натискання стан кнопки змінюється.
- **AppBarToggleButton** – цей елемент керування подібний до **ToggleButton**, але має стилі, рекомендовані для кнопок панелі програм.

- **CheckBox** – ця кнопка містить простий прапорець, що показує, натиснутої чи ні. По суті це та сама **ToggleButton**, але вона використовує інший шаблон. **ToggleButton** і **CheckBox** можуть мати і третій стан – невизначений, який можна задати лише в коді. Це знадобиться, якщо необхідно перевірити, чи зробив користувач хоча б якийсь вибір, чи не проминув кнопку взагалі.
- **RadioButton** – цей елемент керування також має два стани, але працює разом із іншими об'єктами **RadioButton**. За його допомогою можна визначити певну кількість елементів керування в групі й без написання коду реалізувати вибір одного з них.

Окрім того, зверніть увагу на елемент керування **ToggleSwitch**, що не є успадкованим від **ButtonBase**, але по суті це теж кнопка. **ToggleSwitch** дає змогу виконувати функції **CheckBox**, але якщо використати текст, селектор матиме унікальний вигляд:

```
<ToggleSwitch OffContent="No" OnContent="Yes" />
```

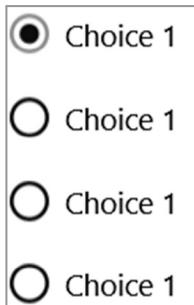
Кнопка відображає повідомлення **Yes** або **No** залежно від стану, у якому перебуває:



Розгляньмо приклади з кнопками. Нижче ми визначимо декілька елементів керування **RadioButton** у групі, щоб можна було вибрати один із них:

```
<StackPanel>
    <RadioButton Content="Choice 1" IsChecked="True"
        GroupName="Group 1" Margin="5">
    </RadioButton>
    <RadioButton Content="Choice 1" IsChecked="False"
        GroupName="Group 1" Margin="5">
    </RadioButton>
    <RadioButton Content="Choice 1" IsChecked="False"
        GroupName="Group 1" Margin="5">
    </RadioButton>
    <RadioButton Content="Choice 1" IsChecked="False"
        GroupName="Group 1" Margin="5">
    </RadioButton>
</StackPanel>
```

Запустивши цей приклад, отримаємо таку групу елементів керування:



Оскільки всі кнопки, крім **ToggleSwitch**, успадковано від **ContentControl**, вони підтримують усі шаблони для властивості **Content**. Погляньмо на код:

```
<Button Content="Hello">
    <Button.ContentTemplate>
        <DataTemplate>
            <Grid>
                <Ellipse Width="100" Height="50" Fill="Red" />
                <TextBlock Text="{Binding}"
                           HorizontalAlignment="Center"
                           VerticalAlignment="Center" />
            </Grid>
        </DataTemplate>
    </Button.ContentTemplate>
</Button>
```

У цьому прикладі ми застосували свій шаблон, використавши властивість **ContentTemplate**. Для цієї властивості потрібний дуже часто використовуваний об'єкт класу **DataTemplate**. Усередині **DataTemplate** можна розмістити лише один елемент керування, а решту – в контейнері. Наразі ми використовуємо **Grid** та розмістили **Ellipse** і **TextBlock** у **Grid**. Зверніть особливу увагу, що ми використали шаблон, але кнопка має властивість **Content**, яку можна присвоїти поза шаблоном. Щоб скористатися цією властивістю, потрібне розширення розмітки **Binding**. Цей синтаксис дає змогу використовувати властивість **Content** як джерело для властивості **Text** елемента керування **TextBox**. Після запуску коду з'явиться така кнопка:



## Робота з текстом

Насамперед розгляньмо найпоширеніший з-поміж елементів керування для редагування текстів – **TextBox**. Він, як і решта елементів для редагування тексту, є успадкованим від класу **Control** і має властивість **Template**. На сторінці [msdn.microsoft.com](https://msdn.microsoft.com/en-us/library/windows/apps/xaml/jj710191.aspx) є визначення для цього та інших елементів керування. Перейшовши за посиланням <https://msdn.microsoft.com/en-us/library/windows/apps/xaml/jj710191.aspx>, можна побачити, що **TextBox** містить **ContentPresenter**, **Border**, **ScrollViewer**, **ContentControl** і навіть **Button**. Завдяки **TextBox** користувачі матимуть змогу редагувати текст. Зазвичай цей елемент керування наявний у формах.

Стандартний **TextBox** заповнює весь простір контейнера, тож для його розміщення можна використовувати будь-яку доступну властивість **FrameworkElement**:

```
<TextBox Margin="10" HorizontalAlignment="Left"
VerticalAlignment="Top" Width="150"></TextBox>
```

Після запуску коду з'явиться типове текстове поле



Щойно ви почнете редагувати текст, наприкінці його відобразиться кнопка, натиснувши яку, можна очистити текстове поле все відразу. Саме тому кнопку було додано до шаблону **TextBox**. Спробуйте ввести багато тексту, і ви побачите, як працює **ScrollViewer**, що також є частиною шаблону **TextBox**.

Користувацький елемент керування **TextBox** придатний для редагування складніших текстів, що складаються з декількох рядків. У такому разі слід застосовувати деякі властивості. Основні з них такі:

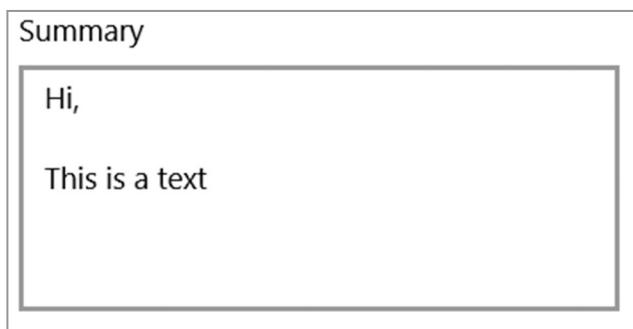
- **AcceptsReturn** – ця властивість дає змогу створити нові рядки всередині тексту.
- **IsReadOnly** – завдяки цій властивості можна визначати, чи відображатиметься текст в режимі «тільки для читання».
- **Header** – дає можливість створити заголовок, що може бути просто текстом або чимось складнішим, залежно від **HeaderTemplate**.
- **InputScope** – встановлює стандартну екранну клавіатуру. Наприклад, якщо **TextBox** призначено для введення чисел, можна використовувати **Number** як значення **InputScope**. Зауважте, що дані не перевіряються і користувач будь-коли може змінити тип введення з клавіатури.

- **SelectedText** – повертає текст, виділений у текстовому полі.
- **SelectionLength** – повертає кількість виділених символів.
- **SelectionHighlightColor** – завдяки цій властивості можна вибрати пензель для підсвічування виділення.
- **SelectionStart** – повертає положення першого символу з виділених.
- **TextWrapping** – дає можливість визначати поведінку тексту, якщо рядок тексту довший за ширину текстового поля.
- **IsSpellCheckEnabled** – активує/вимикає перевірку граматики.
- **PlaceholderText** – дає можливість заповнювати **TextBox** заповнювачем, який буде видалено, щойно користувач розпочне редагування.

Розглянемо короткий приклад з користувацьким елементом керування **TextBox**:

```
<TextBox Margin="10" HorizontalAlignment="Left"  
AcceptsReturn="True" VerticalAlignment="Top" Height="150"  
Width="300" Header="Summary"></TextBox>
```

У цьому полі можна ввести декілька рядків тексту, і воно матиме заголовок:



Але якщо ви введете багато тексту, смуга прокручування не з'явиться. Щоб її активувати, скористайтеся властивостями залежностей і застосуйте їх до внутрішнього **ScrollViewer** із шаблону.

```
<TextBox Margin="10" ScrollViewer.VerticalScrollBarVisibility="Auto"  
HorizontalAlignment="Left" AcceptsReturn="True"  
VerticalAlignment="Top" Height="150" Width="300"  
Header="Summary" />
```

Наступний елемент керування для редагування тексту – це **PasswordBox**. За його допомогою можна вводити прихований текст, що зазвичай потрібно для паролів. Цей елемент керування має такі важливі властивості:

- **Password** – надає доступ до тексту, введеного до **PasswordBox**;
- **PasswordChar** – дає змогу визначити символи, що будуть показані замість справжніх;
- **IsPasswordRevealButtonEnabled** – приховує чи показує кнопку, що допомагає користувачам бачити введений текст.

Розглянемо приклад використання **PasswordBox**:

```
<PasswordBox Width="150" HorizontalAlignment="Left"
VerticalAlignment="Top" PasswordChar="*" Margin="10"
IsPasswordRevealButtonEnabled="True"></PasswordBox>
```

Після запуску коду відобразиться такий користувацький елемент керування:



Далі мова піде про **SearchBox**. Цей елемент керування також допомагає вводити та редагувати текст, але його адаптовано до пошукових запитів. Звичайно, основною перевагою є підтримка історії пошуку, яка відображатиметься завдяки властивості **SearchHistoryEnabled**. Окрім того, цей елемент керування містить спеціальну кнопку пошуку і підтримує деякі важливі події, наприклад **QuerySubmitted**, **QueryChanged** і **ResultSuggestionChosen**.

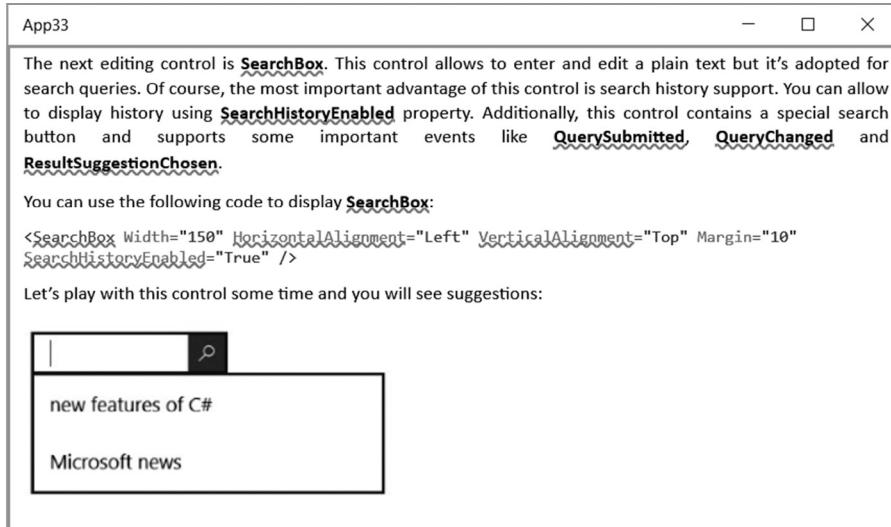
Для відображення **SearchBox** можна використати такий код:

```
<SearchBox Width="150" HorizontalAlignment="Left"
VerticalAlignment="Top" Margin="10"
SearchHistoryEnabled="True" />
```

Попрацюйте з цим елементом керування деякий час, і побачите підказки:



Наочанок розповімо про ще один важливий елемент керування – **RichEditBox**. Завдяки йому можна редагувати документи формату **text** та **rtf**, де можуть бути параграфи, різні шрифти, зображення тощо.



У поле цього елемента можна легко вставити форматований текст із Word чи завантажити rtf-документ.

Цей елемент керування не містить жодних кнопок, що могли б допомогти форматувати і редагувати текст (тільки контекстне меню). Якщо ви хочете розробити професійний редактор, потрібно реалізувати свій інтерфейс, що дасть змогу розширити функціональність **RichEditBox**. Після додавання кнопок редагування можна буде скористатися інтерфейсами виділення та форматування за допомогою властивості **Document**.

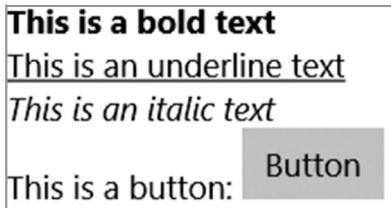
Universal Windows Platform містить декілька користувачьких елементів керування, за допомогою яких текст відображається без можливості редагування. Усі ці елементи успадковані напряму від **FrameworkElement** і не мають шаблонів, але часто самі є базовими будівельними блоками для інших елементів керування.

Перший з таких елементів – це **TextBlock**, що дає змогу відображати неформатований текст. За допомогою властивості **Text** цьому елементу керування можна присвоїти будь-який текст, а завдяки властивостям **FontFamily**, **TextTrimming** чи **TextWrapping** – застосувати базове форматування тексту. Ми використовуватимемо цей елемент керування дуже часто, тому зараз немає сенсу детально зупинятися на ньому.

Нарешті розгляньмо користувацький елемент керування **RichTextBlock**, що відображає форматований текст. По суті він є контейнером, що може містити спеціальні елементи, наприклад **Paragraph**, **Run**, **Span**, **Bold**, **LineBreak** тощо. Подивімось, як використовувати ці елементи усередині **RichTextBlock**:

```
<RichTextBlock HorizontalAlignment="Left"
    Name="rArea" VerticalAlignment="Top"
    Height="300" Width="400" >
    <Paragraph>
        <Bold>This is a bold text</Bold>
        <LineBreak></LineBreak>
        <Underline>This is an underline text</Underline>
        <LineBreak></LineBreak>
        <Italic>This is an italic text</Italic>
        <LineBreak></LineBreak>
        This is a button:
        <InlineUIContainer>
            <Button Content="Button"></Button>
        </InlineUIContainer>
    </Paragraph>
</RichTextBlock>
```

Після запуску коду відобразиться таке:



Зауважте, що **RichTextBlock** може містити й елементи XAML.

## Елементи керування RangeBase

У цій темі ми розглянемо декілька користувацьких елементів керування – елементів прокручування, успадкованих від класу **RangeBase**: **Slider**, **ScrollBar** і **ProgressBar**. Оскільки **RangeBase** успадковано від класу **Control**, він має всі основні властивості й можливості, зокрема шаблони. Найважливішими властивостями цих елементів керування є:

- **Minimum** – визначає мінімальне значення елемента прокручування;
- **Maximum** – визначає максимальне значення елемента прокручування;
- **Value** – дає змогу отримати чи встановити поточне положення елемента прокручування.

Різниця між всіма цими елементами керування полягає в тому, що за допомогою **Slider** і **ScrollBar** можна змінити поточне значення, а **ProgressBar** лише відображає стан.

Наведений нижче код дає можливість показати всі три елементи:

```
<StackPanel>
    <ProgressBar Minimum="0" Maximum="100" Value="50"
        Margin="10"></ProgressBar>
    <ScrollBar Minimum="0" Maximum="100" Value="50"
        Margin="10" IndicatorMode="MouseIndicator"
        Orientation="Horizontal"></ScrollBar>
    <Slider Minimum="0" Maximum="100" Value="50" Margin="10">
    </Slider>
</StackPanel>
```

Після запуску коду відобразиться такий інтерфейс:



Зважайте, що стандартний **ScrollBar** має вертикальну орієнтацію і не містить жодних індикаторів прокручування. Отже, потрібно змінити ці властивості.

## ProgressRing

У разі, якщо **ProgressBar** не дає змоги відобразити поточний стан виконання, можна скористатися елементом керування **ProgressRing**. Він показує, що програма виконує певні дії і не відображає кількість виконаної роботи. Для активації цього елемента потрібна властивість **IsActive**, що за замовчування має значення **false**.

```
<ProgressRing IsActive="True" Margin="10" Height="100"
    Width="100"></ProgressRing>
```

Завдяки властивостям **Width** і **Height** можна визначити розмір рухомого кільця.

## Елемент керування **ToolTip**

Цікавим елементом керування є **ToolTip**. Він дає змогу відобразжати підказки і може бути присвоєний будь-якому елементу керування та містити будь-що. Розглянемо код:

```
<Button Content="Detach" MinWidth="100" Margin="10">
    <ToolTipService.ToolTip>
        <ToolTip Placement="Right">
            <ToolTip.Content>
                <TextBlock Text="Any content here"></TextBlock>
            </ToolTip.Content>
        </ToolTip>
    </ToolTipService.ToolTip>
</Button>
```

Як бачимо, **ToolTip** має властивість **вмісту**, яка може містити будь-який контейнер і елементи керування. Крім того, властивість **Placement** допомагає обрати найкраще місце для підказки.

## Робота з колекціями

UWP містить декілька елементів керування, що дають змогу відобразжати колекцію елементів. Усі класи успадковано від **ItemsControl**, вони мають подібні можливості, але забезпечують різні подання внутрішніх елементів:

```

Windows.UI.Xaml.Controls.ItemsControl
  ▷ Base Types
  ▷ Derived Types
    ▷ AutoSuggestBox
    ▷ CommandBarOverflowPresenter
    ▷ MenuFlyoutPresenter
    ▷ Pivot
    ▷ Selector
      ▷ ComboBox
      ▷ FlipView
      ▷ ListBox
    ▷ ListViewBase
      ▷ GridView
      ▷ ListView
  
```

Завдяки **ItemsControl** усі класи мають принаймні такі властивості:

- **Items** – містить колекцію елементів.
- **ItemsTemplate** – допомагає визначити **DataTemplate**, що відображає певний елемент.
- **ItemsSource** – завдяки цій властивості можна пов'язати колекцію з користувачким елементом керування. Ми обговоримо цю властивість у наступних розділах.
- **ItemsPanel** – дає змогу визначити тип розмітки, використаної для розміщення елементів всередині елемента керування.

Можна використовувати **ItemsControl** для відображення колекції даних. Визначмо такий інтерфейс:

```
<ItemsControl Name="itemsControl" Margin="50">
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <StackPanel></StackPanel>
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                </Grid.RowDefinitions>
                <TextBlock Text="{Binding NewsTitle}"
                           Style="{StaticResource
                           TitleTextBlockStyle}"></TextBlock>
                <TextBlock Grid.Row="1"
                           Text="{Binding NewsTitle}"
                           Style="{StaticResource
                           BodyTextBlockStyle}"></TextBlock>
            </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Як видно з цього прикладу, для подання наших даних (за замовчуванням – вертикально) застосовано **StackPanel**. Було створено спеціальний шаблон даних для певного елемента списку, де використано декілька текстових полів і сітку як контейнер. Усередині шаблону даних ми використовуємо розширення розмітки для зв’язування даних – поки що не намагайтесь зрозуміти, що це означає.

Далі потрібно визначити клас, що міститиме дані, й використати його для створення колекції, яку ми пов’яжемо з елементом керування.

Ви можете створити клас будь-де у власному проекті:

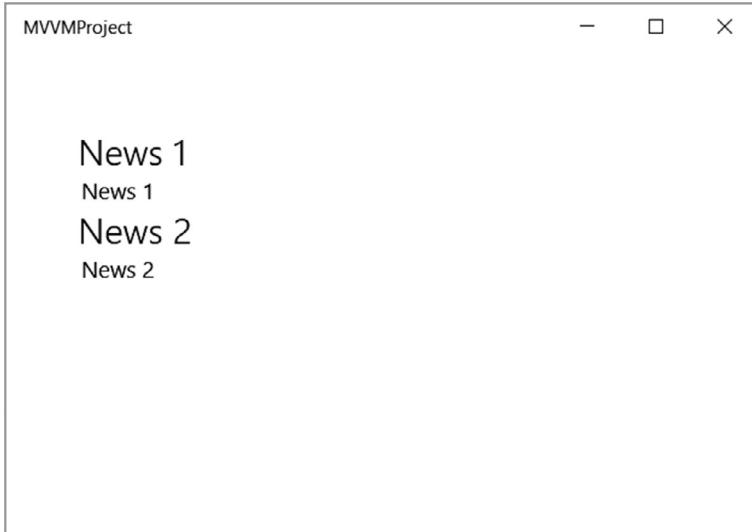
```
public class NewsItem
{
    public string NewsTitle { get; set; }

    public string NewsDescription { get; set; }
}
```

Аби цей код почав працювати, реалізуйте такий метод, що належить до класу **MainPage**:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    ObservableCollection<NewsItem> list =
        new ObservableCollection<NewsItem>();
    list.Add(new NewsItem() { NewsTitle = "News 1",
        NewsDescription = "News description 1" });
    list.Add(new NewsItem() { NewsTitle = "News 2",
        NewsDescription = "News description 2" });
    itemsControl.ItemsSource = list;
}
```

Після запуску коду відобразиться такий інтерфейс:



Звичайно, це базовий інтерфейс, що не дає змоги вибирати елементи, не підтримує анімацію тощо. Тому ми використовуватимемо його для свого елементу керування як базовий клас. Корпорація Microsoft скористалася цим інтерфейсом для створення кількох готових до використання елементів керування, які ми розглянемо далі.

Змінімо наш файл XAML, використовуючи елемент керування **ListView**:

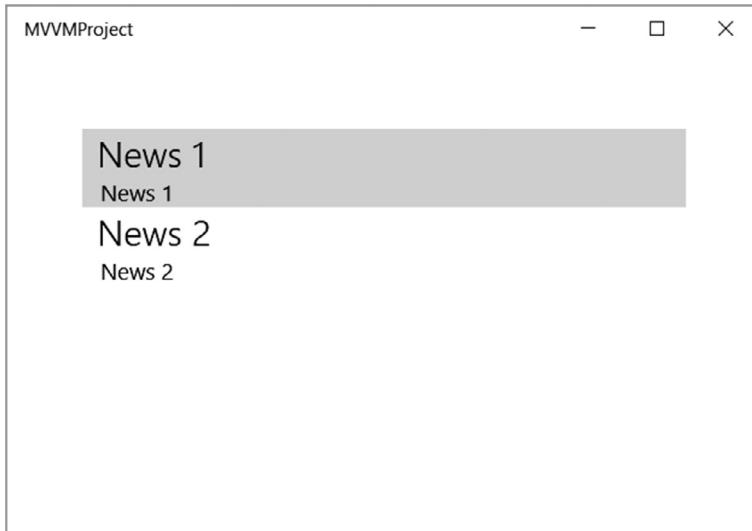
```
<ListView Name="itemsControl" Margin="50">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                </Grid.RowDefinitions>
                <TextBlock Text="{Binding NewsTitle}"
                           Style="{StaticResource
                           TitleTextBlockStyle}"/>
                <TextBlock Grid.Row="1" Text="{Binding
                           NewsDescription}" Style="{StaticResource
                           BodyTextBlockStyle}"/>
            </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ListView>
```

```

        </Grid>
    </DataTemplate>
</ItemsControl.ItemTemplate>
</ListView>

```

Ми видалили **ItemsPanelTemplate** і змінили **ItemsControl** на **ListView**. У результаті відобразиться той самий список даних на екрані, проте можна буде вибрати елемент зі списку, а також підтримуватиметься анімація:



Це стало можливим завдяки класу **Selector**, що підтримує вибір. Клас **ListViewBase** дає змогу обробляти подію вибору елемента, змінювати порядок елементів, застосовувати семантичне масштабування, визначати заголовок і нижній колонтитул тощо. Отже, якщо потрібно розташувати дані вертикально, скористайтеся класом **ListView**.

Змінімо елемент **ListView** на **GridView**:

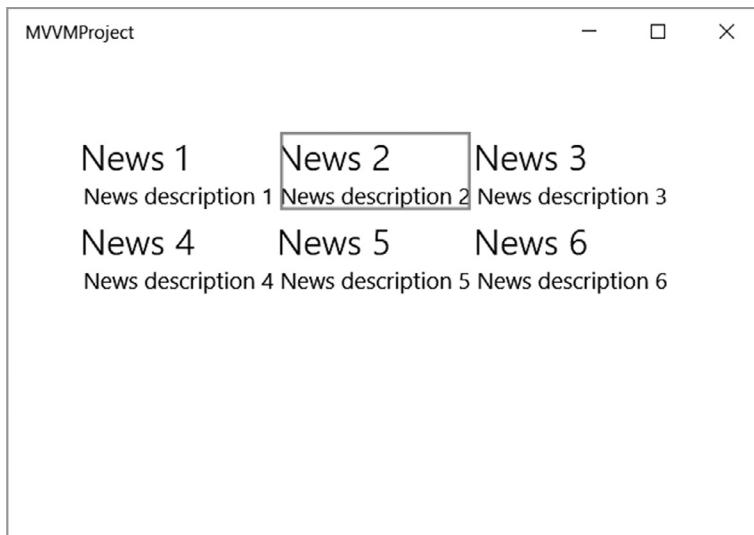
```

<GridView>
    <GridView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto">
                </RowDefinition>

```

```
<RowDefinition Height="Auto">
</RowDefinition>
</Grid.RowDefinitions>
<TextBlock Text="{Binding NewsTitle}"
Style="{StaticResource TitleTextBlockStyle}">
</TextBlock>
<TextBlock Grid.Row="1" Text="{Binding
NewsDescription}" Style="{StaticResource
BodyTextBlockStyle}"></TextBlock>
</Grid>
</DataTemplate>
</GridView.ItemTemplate>
</GridView>
```

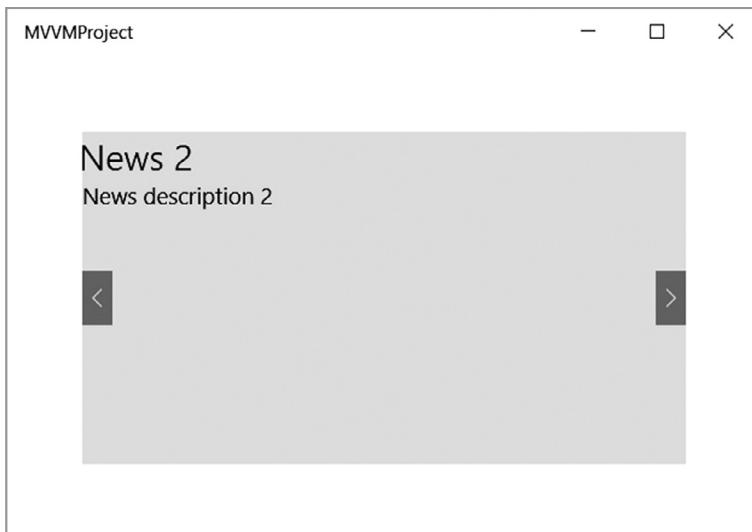
У такому разі дані буде відображені в рядках і стовпцях, які можна прокручувати горизонтально:



І ще один елемент керування, який допомагає працювати з колекціями, – це **FlipView**. На відміну від **ListView** і **GridView**, він не відображає всіх елементів водночас, а показує їх послідовно і забезпечує навігацію. Зазвичай цей елемент розміщується на веб-сайті й слугує для відображення колекції зображень, але загалом він може відображати будь-що, зокрема наші дані:

```
<FlipView Name="itemsControl" Margin="50">
    <FlipView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                </Grid.RowDefinitions>
                <TextBlock Text="{Binding NewsTitle}"
                           Style="{StaticResource TitleTextBlockStyle}" >
                </TextBlock>
                <TextBlock Grid.Row="1" Text="{Binding
                           NewsDescription}" Style="{StaticResource
                           BodyTextBlockStyle}"></TextBlock>
            </Grid>
        </DataTemplate>
    </FlipView.ItemTemplate>
</FlipView>
```

Цей код відображає такий інтерфейс користувача:



## SplitView

Цей елемент керування дає змогу створювати адаптивний інтерфейс. Зазвичай ви користуватиметеся ним для створення меню, але загалом за допомогою **SplitView** можна задати дві панелі із будь-яким контентом, **Pane** і **Content**. Зокрема, панель **Pane** підтримує адаптивні можливості, оскільки підтримує різні режими відображення. Основний синтаксис **SplitView** такий:

```
<SplitView IsPaneOpen="False"
           DisplayMode="CompactInline"
           PaneBackground="Beige"
           OpenPaneLength="200"
           CompactPaneLength="30">
    <SplitView.Pane>
        </SplitView.Pane>
    <SplitView.Content>
        </SplitView.Content>
    </SplitView>
</SplitView>
```

Властивості **DisplayMode** може бути надано такі значення:

- **CompactInline** – панель **Pane** підтримує компактний режим. Коли її розгорнуто, весь вміст зміститься так, щоб залишалося достатньо місця для панелі.
- **CompactOverlay** – схоже на **CompactInline**, але розгортання панелі не впливатиме на решту елементів керування, оскільки панель розміщується поверх контенту.
- **Inline** – підтримується лише в розгорнутому режимі. У разі відображення панелі весь контент зміститься так, щоб для неї залишалося достатньо місця.
- **Overlay** – підтримується лише в розгорнутому режимі. Не впливає на решту вмісту, оскільки панель розміщується поверх контенту.

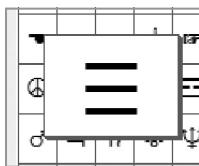
**IsPaneOpen** допомагає визначити, відображено панель у стандартному режимі чи в розгорнутому (або компактному). Отже, якщо встановлено хибне значення властивості **IsPaneOpen** і ввімкнuto компактний режим відображення, панель відображатиметься в компактному режимі. За істинного значення **IsPaneOpen** панель буде відображено в розгорнутому режимі.

Попри те що **SplitView** ніяк не пов'язано з меню, у ньому можна розмістити щось на кшталт **ListView** і оголосити елементи меню. Радимо створювати таке

меню, щоб користувачі могли бачити лише піктограми в компактному режимі, а решту вмісту – в розгорнутому. Реалізуйте інтерфейс **VisualStateManager**, що змінює властивості **DisplayMode** і **IsPaneOpen** під час виконання.

Декілька порад щодо створення меню:

- Слід передбачити три стани відображення меню: розширений (**Expanded**), компактний (**Compact**) і надкомпактний (**UltraCompact**) для різних розмірів екрана. Якщо місця вистачає, можна без жодних проблем показати меню в розширеному режимі, але за браку простору краще, щоб відображалися лише піктограми в компактному режимі. Зрештою, якщо місця немає взагалі (екран телефона в портретній орієнтації), радимо активувати надкомпактний стан і приховати панель **Pane**.
- За компактного і надкомпактного станів вторі меню додайте кнопку маркованого списку, що відкриває панель **Pane**, використовуючи режими **Overlay**. Так ви матимете змогу відкрити меню навіть у надкомпактному режимі, коли воно приховане.



- Використовуйте для меню елемент керування **ListView**.

**VisualStateManager** може мати такий вигляд (**CompactInline** за замовчуванням і значення **true** для **IsPaneOpen**):

```
<VisualState x:Name="Expanded">
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="900">
        </AdaptiveTrigger>
    </VisualState.StateTriggers>
</VisualState>
<VisualState x:Name="Compact">
    <VisualState.Setters>
        <Setter Value="False" Target="splitView.IsPaneOpen">
        </Setter>
        <Setter Value="CompactOverlay"
            Target="splitView.DisplayMode"></Setter>
    </VisualState.Setters>
    <VisualState.StateTriggers>
```

```
<AdaptiveTrigger MinWindowWidth="500">
</AdaptiveTrigger>
</VisualState.StateTriggers>
</VisualState>
<VisualState x:Name="UltraCompact">
<VisualState.Setters>
<Setter Value="False" Target="splitView.IsPaneOpen">
</Setter>
<Setter Value="Overlay" Target="splitView.DisplayMode">
</Setter>
</VisualState.Setters>
<VisualState.StateTriggers>
<AdaptiveTrigger MinWindowWidth="0">
</AdaptiveTrigger>
</VisualState.StateTriggers>
</VisualState>
```

Якщо ви додасте кнопку маркованого списку, вам необхідно буде додати логіку, що дасть змогу розширювати й меню. Можна реалізувати всю логіку в коді або створити ще два стани і власні тригери. Пізніше ми поговоримо про тригери, адаптивний дизайн і **VisualStateManager** докладніше.

У цьому розділі йшлося про основні елементи керування. Однак UWP має їх набагато більше, також існує величезна кількість сторонніх елементів. Отже, у вас не має виникнути труднощів із розробкою інтерфейсу.

У наступному розділі ми поговоримо про те, як реалізувати взаємодію з користувачами.

Розділ 5.

## **Взаємодія з користувачем**

ОС Windows 10 чудово працює з різними способами й засобами введення інформації, такими як миша/клавіатура, перо і жести. Завдяки цьому користувачі можуть використовувати різні пристрої, включно з телефонами та планшетами. Але нові можливості вимагають зміни моделі роботи з подіями. Ці зміни були представлені в Windows 8, коли було анонсовано Windows Runtime. Якщо у вас є певний досвід роботи з програмами Windows Forms чи Win32 API, можливо ви будете трішки спантеличені. Наприклад, ви не зможете знайти події, пов'язані з мишею, оскільки не має значення, за допомогою чого користувач взаємодіє з пристроям – миши, пера чи пальців. Інакше розробнику довелося б реалізовувати різну логіку й обробку подій для різних методів введення.

Система Windows 10 підтримує три групи подій, що стосуються взаємодії з користувачами:

- події, пов'язані з натисканням;
- події, що відповідають жестам;
- події, що відповідають руху вказівника (пальця, пера чи курсора миши).

Кожна із цих груп дає можливість використовувати деякі загальні сценарії. Наприклад, користувач може клацанням активувати контекстне меню чи надіслати подію клацання елементу керування, скажімо кнопці. За допомогою жестів можна масштабувати інтерфейс, перетягувати об'єкти та рухати їх по екрану. Події, пов'язані з вказівниками, дають можливість їх відстежувати й малювати об'єкти з використанням пера, пальців чи миши. Вагомим є й те, що всі ці події визначені в класі **UIElement**, а тому доступні для всіх візуальних елементів. Звичайно, не завжди потрібно працювати з цими подіями безпосередньо. Наприклад, клас **Button** містить подію **Click**, яку потрібно використовувати, тому що користувачі можуть натиснути кнопку, навіть коли вона відключена.

Ознайомимося детальніше зожною із цих груп подій.

Universal Windows Platform визначає чотири події, що пов'язані з натисканням:

- **Tapped** – така подія виконується після торкання. Якщо використовується миша, торкання генерується в результаті клацання лівою кнопкою миши. Але якщо користувач працює пером чи пальцями, йому потрібно натиснути на екран і відпустити його.
- **DoubleTapped** – ця подія подібна до попередньої, однак генерується після подвійного торкання.
- **Holding** – подія вказує, чи користувач притискає палець до екрана або утримує натиснутою праву кнопку миши.

- **RightTapped** – подія виникає, коли користувач відпускає праву кнопку миші. Якщо використовується перо або пальці, подія настає відразу після події **Holding**, коли перо чи палець відтискається від екрана.

Потрібно враховувати, що ці події можуть бути вимкнуті за замовчуванням. Для оброблення цих подій вам необхідно активувати властивості **IsTapEnabled**, **IsDoubleTapEnabled**, **IsHoldingEnabled**, **IsRightTapEnabled**, встановивши для них значення **true**.

Незважаючи на універсальність, ці події дають змогу визначити тип вказівника, що використовується. У параметрах обробників цих подій є властивість **PointerDeviceType**, що може мати такі значення: **Mouse**, **Pen** і **Touch**. Таким чином, можна цілком напевно знати, що користувач застосував для введення даних.

Крім того, події **Holding** можуть бути властиві декілька станів утримування. Завдяки ним завжди легко визначити, коли користувач почав і коли завершив утримувати вказівник.

```
private void Grid_Holding(object sender,
    HoldingRoutedEventArgs e)
{
    switch (e.HoldingState)
    {
        case Windows.UI.Input.HoldingState.Started:
            . . . .
            break;
        case Windows.UI.Input.HoldingState.Completed:
            . . . .
            break;
        case Windows.UI.Input.HoldingState.Canceled:
            . . . .
            break;
    }
}
```

Стан скасування генерується, коли користувач переміщує вказівник за межі елемента керування, продовжуючи утримувати палець або кнопку миші.

Важливо пам'ятати, що всі ці події є переспрямованими. Якщо користувач торкнувся елемента, а той не обробляє подію **Tapped**, подію буде спрямовано до батьківського елемента тощо. Навіть якщо є обробники для всіх цих подій, їх

можна переспрямувати до контейнерів, встановивши для властивості **Handled** другого параметра обробника події значення **false**.

Друга група подій дає можливість обробляти жести. Вона налічує п'ять подій:

- **ManipulationStarting** – дає можливість підготуватися до початку маніпуляції. Зазвичай з її допомогою можна ініціалізувати певні змінні, які можуть бути використані пізніше.
- **ManipulationStarted** – сигналізує, що користувач почав взаємодію з об'єктом. Отже, почато маніпуляцію.
- **ManipulationDelta** – подія, яку розробник зазвичай обробляє. Вона дає змогу визначити, яка маніпуляція відбулася. Отже, можна перевірити всі потрібні параметри й застосувати їх до об'єкта.
- **ManipulationCompleted** – подія сигналізує про завершення маніпуляції.
- **ManipulationInertiaStarting** – подія використовується, коли потрібно додати ефект інерції. Скористайтеся параметрами обробника цієї події, щоб розрахувати та застосувати анімації, які відображатимуть інерцію.

Звичайно, для визначення типів маніпуляцій вам потрібно звернутися до параметрів обробників цих подій. Система зробить усі необхідні обчислення й надасть інформацію про те, яка саме взаємодія з пристроєм відбулась (див. нижче). Але результати всіх обчислень, пов'язаних із маніпуляціями, залежать від продуктивності. Тому всі обчислення, що стосуються маніпуляцій, за замовчуванням вимкнуті. Щоб отримувати інформацію про маніпуляції для вибраних елементів керування, необхідно ініціалізувати властивість **ManipulationMode**. Після цього можна починати роботу з маніпуляціями з використанням обробника події **ManipulationDelta**.

Використовуючи властивість **Delta** параметра обробника події **ManipulationDelta**, ви легко визначите, яка маніпуляція відбулася:

- **Rotation** – містить кут повороту в радіанах.
- **Expansion** – містить зміни в пікселях між двома точками дотику, коли дві точки рухаються. Цю властивість можна використати для масштабування.
- **Translation** – містить інформацію про зміни за осями *x* і *y*.
- **Scale** – містить значення зміни у відсотках між двома точками дотику, коли ці точки рухаються. Цю властивість можна використовувати для масштабування.

Визначивши тип маніпуляції, можна застосувати до елементів керування деякі анімації та трансформації.

Погляньмо на код нижче:

```
<Rectangle x:Name="rect" Width="300" Height="150"
    Fill="Green"
    ManipulationMode="All" ManipulationDelta=
    "rect_ManipulationDelta">
    <Rectangle.RenderTransform>
        <CompositeTransform x:Name="transform">
            </CompositeTransform>
    </Rectangle.RenderTransform>
</Rectangle>
```

А також файл програмної частини:

```
private void rect_ManipulationDelta(object sender,
    ManipulationDeltaRoutedEventArgs e)
{
    transform.TranslateX = transform.TranslateX + e.Delta.
        Translation.X;
    transform.TranslateY = transform.TranslateY + e.Delta.
        Translation.Y;
}
```

Цей код демонструє, як використовувати маніпуляції для пересування простого прямокутника.

Остання група подій – це події вказівника. Існує вісім подій:

- **PointerPressed** – активується, коли вказівник натиснули;
- **PointerReleased** – активується, коли вказівник відпустили;
- **PointerWheelChanged** – подія, що обробляє повороти коліщатка миші;
- **PointerMoved** – активується, коли вказівник перемістили;
- **PointerEntered** – активується, коли вказівник перемістили всередину об'єкта;
- **PointerExited** – активується, коли вказівник вийшов за межі об'єкта;
- **PointerCaptureLost** – активується, коли вказівник було переміщено на інший елемент (чи програму);
- **PointerCanceled** – активується в разі раптово втраченого контакту із вказівником.

Ці події не дуже складні. Розглянемо короткий приклад.

Просто визначимо елемент керування **Canvas** на сторінці:

```
<Canvas Name="LayoutRoot" Background="White"
    PointerPressed=" LayoutRoot_PointerPressed"
    PointerMoved=" LayoutRoot_PointerMoved"
    PointerReleased="LayoutRoot_PointerReleased">
</Canvas>
```

І реалізуємо такий код:

```
private void LayoutRoot_PointerPressed(object sender,
    PointerRoutedEventArgs e)
{
    startPoint = e.GetCurrentPoint(LayoutRoot).Position;
}

private void LayoutRoot_PointerMoved(object sender,
    PointerRoutedEventArgs e)
{
    if (startPoint != null)

    {
        Line l = new Line();
        l.X1 = ((Point)startPoint).X;
        l.Y1 = ((Point)startPoint).Y;
        startPoint = e.GetCurrentPoint(LayoutRoot).Position;
        l.X2 = ((Point)startPoint).X;
        l.Y2 = ((Point)startPoint).Y;
        l.Stroke = new SolidColorBrush(Colors.Red);
        LayoutRoot.Children.Add(l);
    }
}

private void LayoutRoot_PointerReleased(object sender,
    PointerRoutedEventArgs e)
{
    startPoint = null;
}
```

Завдяки цьому коду можна малювати всередині **Canvas** за допомогою миші, пера чи пальців.



За потреби до цієї програми можна додати інструменти для змінення кольору, ширини ліній тощо.



Розділ 6.

## **Стилі, ресурси і теми**

## Основні відомості про стилі

**Style** – це спеціальний підхід, що дає змогу застосовувати спільні властивості відразу до цілих груп елементів керування. Відкривши будь-яку програму, ви побачите, що навіть з урахуванням того, що вона має особливу тему, усі звичні елементи керування виглядають однаково. Наприклад, кнопки мають однаковий колір, той самий шрифт і форму. Щоб отримати краще уявлення про стилі, розглянемо такий код:

```
<StackPanel>
    <Button Width="100" Height="50"
        Background="Green"
        Content="Button 1" Margin="5" FontFamily="Arial"
        FontSize="12" FontWeight="Bold"
        Foreground="Blue" BorderThickness="3">
    </Button>
    <Button Width="100" Height="50"
        Background="Green"
        Content="Button 2" Margin="5" FontFamily="Arial"
        FontSize="12" FontWeight="Bold"
        Foreground="Blue" BorderThickness="3">
    </Button>
    <Button Width="100" Height="50"
        Background="Green"
        Content="Button 3" Margin="5" FontFamily="Arial"
        FontSize="12" FontWeight="Bold"
        Foreground="Blue" BorderThickness="3">
    </Button>
    <Button Width="100" Height="50"
        Background="Green"
        Content="Button 4" Margin="5" FontFamily="Arial"
        FontSize="12" FontWeight="Bold"
        Foreground="Blue" BorderThickness="3">
    </Button>
</StackPanel>
```

У цьому визначено чотири кнопки зі спільними властивостями, такими як **Background**, **Margin**, **FontSize** та ін. Існує дві проблеми: перша – величезна кількість коду лише для чотирьох кнопок; друга – у разі внесення змін їх потрібно буде застосовувати до атрибутів у кожній кнопці. Саме тому XAML підтримує спеціальний елемент, що називається **Style**.

Клас **Style** визначений у просторі імен **Windows.UI.Xaml** і містить чотири важливі властивості: **BasedOn**, **Setters**, **TargetType** і **IsSealed**. Властивість **TargetType** визначає тип, до якого можна застосувати стиль, а **Setters** – це посилання на колекцію об'єктів **Setter**. Відкривши клас **Setter** в Object Browser, ви виявите, що **Setter** зберігає інформацію про властивість та її значення. Але важливіше те, що властивість повинна бути **DependencyProperty**, інші властивості використовувати не можна. Тому в разі створення власного елемента керування важливо успадковувати клас **DependencyObject** і реєструвати усі загально доступні властивості, що встановлюються візуальними засобами, наприклад **DependencyProperty**. Тому коли ви створюєте об'єкт класу **Style**, він повинен містити хоча б колекції **TargetType** і **Setters**. Достатньо перемістити всю спільну візуальну інформацію з елементів керування **TargetType** до стилю. Властивості **IsSealed** і **BasedOn** класу **Style** буде розглянуто далі в цьому розділі.

Отже, нам відомо, як зібрати інформацію про певні властивості та їхні значення всередині одного об'єкта, а зараз з'ясуємо, як застосувати цей об'єкт до елементів керування. Відкривши **FrameworkElement** в Object Browser, ви знайдете там властивість **Style**, що може приймати об'єкт типу **Style**. Це дає змогу записати такий код:

```
Style st = new Style(typeof(Button));
st.Setters.Add(new Setter(Button.BackgroundProperty,
    Colors.Red));
mbutton.Style = st;
```

Але такий підхід не часто застосовується до елементів керування XAML. Натомість в XAML можна оголосити об'єкти **Style**. Оскільки цей елемент не є частиною візуального дерева, його слід включити до ресурсів (детальніше – далі в цьому розділі).

Ось так елемент **Style** визначається для спільних властивостей кнопок у нашому прикладі:

```
<Page.Resources>
    <Style x:Key="buttonStyle" TargetType="Button">
        <Setter Property="Background" Value="Green"></Setter>
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="FontFamily" Value="Arial"></Setter>
        <Setter Property="FontSize" Value="12"></Setter>
        <Setter Property="FontWeight" Value="Bold"></Setter>
        <Setter Property="Foreground" Value="Blue"></Setter>
        <Setter Property="BorderThickness" Value="3"></Setter>
    </Style>
</Page.Resources>
```

Стиль визначено в ресурсах об'єкта **Page**, оскільки елемент **Style** розміщено як внутрішній елемент **Page**.

У прикладі використано тег **Style** із двома властивостями: **TargetType** і **x:Key**. Щойно було згадано першу, але з огляду на те, що Visual Studio визначає типи, для редагування елемента **Style** можна застосовувати систему IntelliSense. Друга властивість є ключем для визначення стилів і використовується для застосування стилів до обраних елементів керування.

Покажемо, як застосовувати стилі до кнопок:

```
<StackPanel>
    <Button Width="100" Height="50"
        Style="{StaticResource buttonStyle}" Content="Button 1">
    </Button>
    <Button Width="100" Height="50"
        Style="{StaticResource buttonStyle}" Content="Button 2">
    </Button>
    <Button Width="100" Height="50"
        Style="{StaticResource buttonStyle}" Content="Button 3">
    </Button>
    <Button Width="100" Height="50"
        Style="{StaticResource buttonStyle}" Content="Button 4">
    </Button>
</StackPanel>
```

Для застосування стилів ми використовуємо розширення розмітки **StaticResource**, завдяки якому можна робити посилання на локальні ресурси. Ім'я стилю було передано як параметр, і цю конструкцію присвоєно властивості **Style** елемента **Button**.

Після додавання стилю до елемента XAML його не можна змінити динамічно. Властивість **IsSealed** визначає, чи можна вносити зміни до стилю. Якщо **IsSealed** має значення **true** і ви намагатиметесь внести зміни всередині стилю, система буде генерувати виняток.

Якщо стиль задається без імені чи неявно, він може не мати ключової властивості. Тоді його буде застосовано до всіх елементів керування зазначеного типу, якщо явно не застосовано інший стиль. Тому з наведеного коду можна видалити навіть розширення розмітки:

```
<Page.Resources>
    <Style TargetType="Button">
```

```

<Setter Property="Background" Value="Green"></Setter>
<Setter Property="Margin" Value="5"></Setter>
<Setter Property="FontFamily" Value="Arial"></Setter>
<Setter Property="FontSize" Value="12"></Setter>
<Setter Property="FontWeight" Value="Bold"></Setter>
<Setter Property="Foreground" Value="Blue"></Setter>
<Setter Property="BorderThickness" Value="3"></Setter>
</Style>
</Page.Resources>
<StackPanel>
    <Button Width="100" Height="50"
        Content="Button 1">
    </Button>
    <Button Width="100" Height="50"
        Content="Button 2">
    </Button>
    <Button Width="100" Height="50"
        Content="Button 3">
    </Button>
    <Button Width="100" Height="50"
        Content="Button 4">
    </Button>
</StackPanel>
</Grid>

```

## Розширення стилів

Стилі XAML підтримують спеціальний атрибут **BasedOn**. За його допомогою можна розширити наявні стилі елементів керування:

```

<Style x:Key="baseStyle" TargetType="Button" >
    <Setter Property="Width" Value="100"></Setter>
</Style>
<Style x:Key="btnStyle"
    BasedOn="{StaticResource baseStyle}" TargetType="Button">
    <Setter Property="Foreground" Value="Green"></Setter>
</Style>

```

Цей приклад не має практичної цінності, однак вдало демонструє ідею.

Уже присвоєне значення можна змінити, якщо оголосити стиль **BasedOn**:

```
<Style x:Key="baseStyle" TargetType="Button" >
    <Setter Property="Foreground" Value="Red"></Setter>
</Style>
<Style x:Key="btnStyle"
    BasedOn="{StaticResource baseStyle}" TargetType="Button">
    <Setter Property="Foreground" Value="Green"></Setter>
</Style>
```

Після застосування **btnStyle** до кнопки її колір зміниться на зелений.

## Ресурси

Говорячи про стилі, ми використовували властивість **Resource** об'єкта **Page**. Ця властивість доступна для всіх візуальних об'єктів, оскільки визначена в класі **FrameworkElement**. Властивість **Resource** визначено також у класі **Application**. Завдяки властивості **Resource** можна визначати стилі, шаблони, відокремлювати об'єкти всередині візуальних елементів і робити посилання на них. Якщо відкрити **FrameworkElement** в Object Browser, буде видно, що властивість **Resource** має тип **ResourceDictionary**. Останній містить властивість **ThemeDictionaries**, що посилається на ресурси, які стосуються тем, а також властивість **MergedDictionaries**, що містить посилання на список усіх інших словників ресурсів. Завдяки **MergedDictionaries** можна визначити ресурси не тільки всередині елемента керування чи об'єкта **Application**, але й в окремому файлі, і зібрати в ньому всі ці ресурси з різних джерел.

Як згадано вище, ресурси використовуються для зберігання не тільки стилів, а й інших об'єктів. Погляньте на код нижче:

```
<Page.Resources>
    <LinearGradientBrush x:Key="myBrush">
        <GradientStop Color="Red" Offset="0"></GradientStop>
        <GradientStop Color="Green" Offset="1"></GradientStop>
    </LinearGradientBrush>
</Page.Resources>
<StackPanel>
    <Button Width="100" Height="50"
        Background="{StaticResource myBrush}"
        Content="Button 1" Margin="5">
    </Button>
</StackPanel>
```

У цьому коді описано об'єкт типу **LinearGradientBrush**, який використовується як пензель для фону кнопки. Ми використовуємо розширення розмітки **StaticResource** як універсальну можливість отримання доступу до ресурсів.

Якщо оголосити ресурси всередині елемента, вони будуть доступні як для цього елемента, так і для будь-якого внутрішнього елемента. Потрібно лише знати ключ. Ресурси, що оголошені всередині елемента **Application**, доступні скрізь у програмі.

```
<StackPanel>
    <StackPanel.Resources>
        <LinearGradientBrush x:Key="myBrush">
            <GradientStop Color="Red" Offset="0">
                </GradientStop>
            <GradientStop Color="Green" Offset="1">
                </GradientStop>
        </LinearGradientBrush>
    </StackPanel.Resources>
    <Button Width="100" Height="50"
        Background="{StaticResource myBrush}"
        Content="Button 1" Margin="5">
    </Button>
</StackPanel>
```

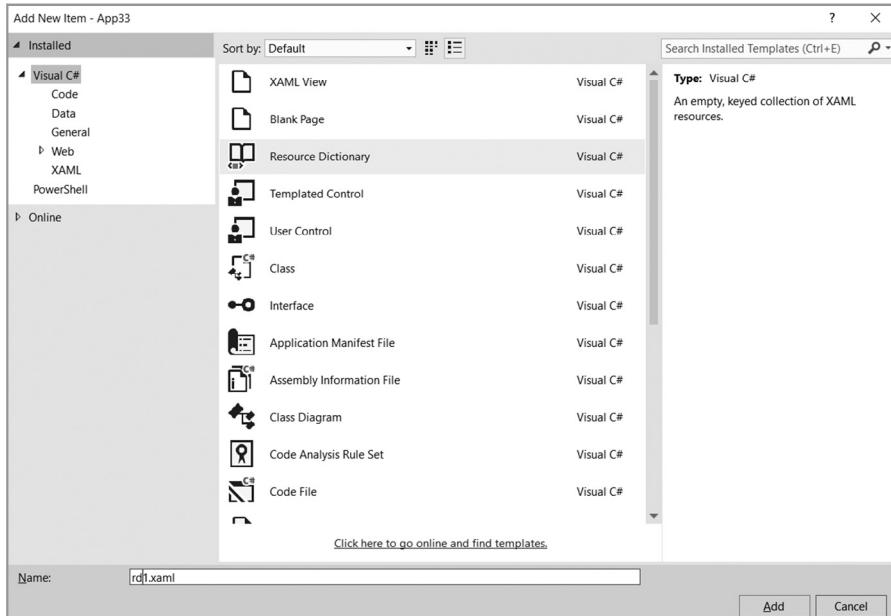
У цьому коді використано властивість **Resource** зі **StackPanel** замість **Page**; той самий підхід застосовано для доступу до об'єкта **LinearGradientBrush**. Тому місце оголошення лише задає область застосування.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Application4">
    <Application.Resources>
        <LinearGradientBrush x:Key="myBrush">
            <GradientStop Color="Red" Offset="0">
                </GradientStop>
            <GradientStop Color="Green" Offset="1">
                </GradientStop>
        </LinearGradientBrush>
    </Application.Resources>
</Application>
```

У цьому прикладі той самий об'єкт оголошено глобально. Об'єкт буде доступний для всіх елементів керування на кожній сторінці.

## Словники ресурсів

Ресурси можна визначати в окремих файлах. У Visual Studio є спеціальний шаблон **Resource Dictionary** для роботи із зовнішніми ресурсами.



Оскільки ресурси в окремих файлах поки не зіставлено з жодними елементами керування, слід використовувати кореневий елемент **ResourceDictionary**:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="myBrush">
        <GradientStop Color="Red" Offset="0"/></GradientStop>
        <GradientStop Color="Green" Offset="1"/></GradientStop>
    </LinearGradientBrush>
</ResourceDictionary>
```

Після оголошення всіх необхідних словників ресурсів їх можна поєднувати все-редині будь-яких елементів керування, включно з об'єктом **Application**. Використовуйте поданий нижче код, щоб включити файли ресурсів до програми:

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Application4">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="RD1.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

У цьому коді застосовано властивість **MergedDictionaries**, що містить список усіх включених словників. Завдяки властивостям **Source** щойно було зроблено посилання на зовнішній файл.

Звичайно, можна додавати необмежену кількість файлів, а також поєднувати їх із явним визначенням об'єктів:

```
<Page.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="rd1.xaml" />
            <ResourceDictionary Source="rd1.xaml" />
        </ResourceDictionary.MergedDictionaries>
        <Style x:Key="mbuttonGreen" TargetType="Button">
            <Setter Property="Background" Value="Green" />
        </Style>
    </ResourceDictionary>
</Page.Resources>
```

У цьому коді додано декілька словників до ресурсів **Page**. Однак зазвичай ви будете працювати з об'єктом **Application**.

## Теми

Створюючи нову сторінку чи програму від початку, скористайтеся стандартним елементом керування **Grid**, що має такий вигляд:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"></Grid>
```

Властивість фону вказує на елемент, який не було визначено в нашому коді – **ApplicationPageBackgroundThemeBrush**. Для створення цього посилання замість **StaticResource** було використано розширення розмітки **ThemeResource**. Приділимо більше уваги цій конструкції.

Радимо почати з **ApplicationPageBackgroundThemeBrush**. Просто вкажіть це ім'я й викличте Peek Definition за допомогою контекстного меню Visual Studio:



Елемент **ApplicationPageBackgroundThemeBrush** було визначено як об'єкт **SolidColorBrush** всередині файлу ресурсів **generic.xaml**. Насправді файл **generic.xaml** – це лише довідка для розробників, а всі ці об'єкти, стилі та шаблони є частиною платформи Universal Windows Platform. Тому **generic.xaml** у жодному разі не можна змінювати.

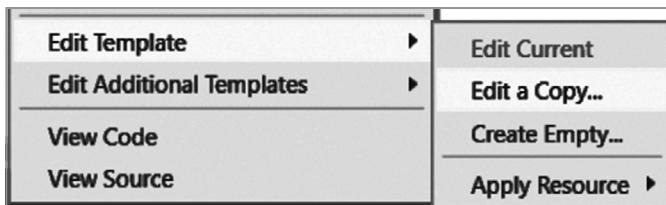
Прокрутіть **generic.xaml**, і побачите стиль для елементів керування **Button**, **CheckBox**, **TextBox** і всіх інших елементів керування, що успадковані від класу **Control**.



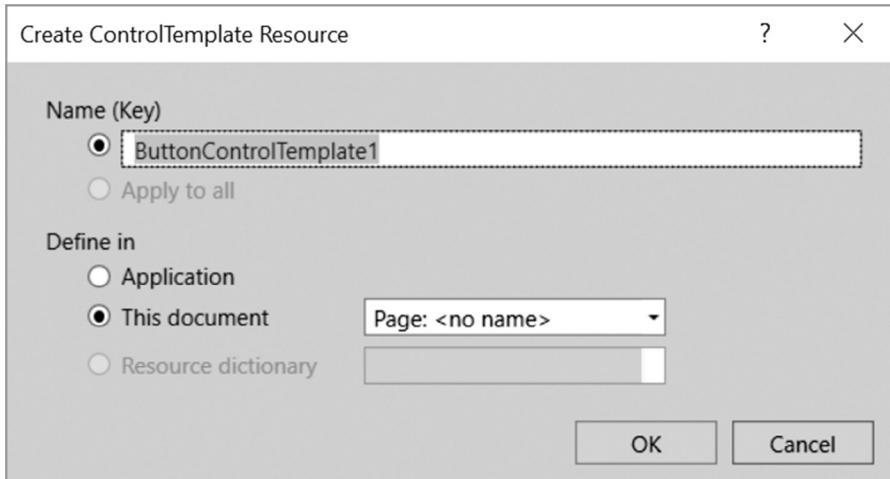
Таким чином, Universal Windows Platform визначає стилі і шаблони для однокових елементів керування чи для спільних значень кольору, шрифту, пензлів тощо. Звичайно, можна використовувати стандартні стилі та шаблони як приклади для створення свого власного стилю або просто переписати деякі об'єкти.

```
<Page.Resources>
    <SolidColorBrush
        x:Key="ApplicationPageBackgroundThemeBrush" Color="Red" />
</Page.Resources>
```

Якщо потрібно внести багато змін до наявних стилів, краще скопіювати їх із **generic.xaml** і змінити те, що потрібно. Visual Studio допомагає створювати копії лише шаблонів, тому прокручувати **generic.xaml** не потрібно. Просто виберіть елемент керування в режимі **Design** і використайте контекстне меню для створення копій:



Visual Studio запропонує вибрати контейнер для шаблону. Тому виберіть елемент керування, об'єкт **Application** чи словник ресурсів (для цього потрібно створити порожній файл):



Отже, ми знаємо про **ApplicationPageBackgroundThemeBrush**, але досі незрозуміло, що таке **ThemeResource**. Давайте відкриємо файл **App.xaml** і переглянемо кореневий елемент:

```
<Application  
    x:Class="App35.App"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/  
        presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="using:App35"  
    RequestedTheme="Light">
```

У коді є властивість **RequestedTheme**, яка за замовчуванням має значення **Light**. Атрибут **RequestedTheme** може бути застосований до будь-якого елемента керування і дає можливість вибирати тему. Завдяки темам можна створювати набори об'єктів, що мають певні параметри. Темна тема може оголошувати шаблон кнопки з білим кольором тексту, але для світлої теми цей колір може бути визначений як чорний. Подивіться на початок файлу **generic.xaml**, де розміщено такий код:

```
<ResourceDictionary.ThemeDictionaries>  
    <ResourceDictionary x:Key="Default">
```

Для оголошення декількох тем використовується властивість **ThemeDictionaries** замість властивості **MergedDictionaries**. Ці теми переважно містять кольори, пензлі та прості об'єкти, але всі шаблони елементів керування, визначені ззовні тем, просто використовують об'єкти тем для своїх властивостей.

Розглянемо такий код:

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">  
    <Grid RequestedTheme="Dark" Background="  
        {ThemeResource ApplicationPageBackgroundThemeBrush}">  
        <Button Content="Hello" Margin="10"></Button>  
    </Grid>  
    <Grid RequestedTheme="Light" Background="  
        {ThemeResource ApplicationPageBackgroundThemeBrush}">  
        <Button Content="Hello" Margin="10"></Button>  
    </Grid>  
</StackPanel>
```

Ми застосували різні теми до двох сіток всередині сторінки, і ці теми були застосовані автоматично до всіх елементів керування всередині. Важливо також, що сітки мають інший фон під час виконання, залежно від теми.



Отже, **ThemeResource** – це спеціальне розширення розмітки, що дає можливість вибирати правильний об'єкт із ресурсів з урахуванням поточної теми. Це розширення розмітки працює як **StaticResource**, але містить певну додаткову логіку, що дає змогу елементам динамічно застосовувати об'єкти з ресурсів. Зокрема, якщо не виправити тему для своєї програми і користувач змінить тему за допомогою системних настройок, ці зміни відразу буде застосовано.

Обговорення тем і шаблонів продовжимо в розділі про настроювані елементи керування.

## Як локалізувати програму

У цьому розділі ми обговорили основні ресурси XAML, які містять різні об'єкти, такі як стилі та шаблони; програми ж, як правило, містять різні типи ресурсів – не тільки ресурси XAML. Наприклад, зображення, піктограми та відеофайли, що включені в пакет програми, також є ресурсами.

Нижче приділимо увагу спеціальним ресурсам, зокрема окремим файлам, які можуть містити текстові рядки. Цей тип ресурсів використовується для локалізації програм.

Розглянемо просту кнопку на сторінці:

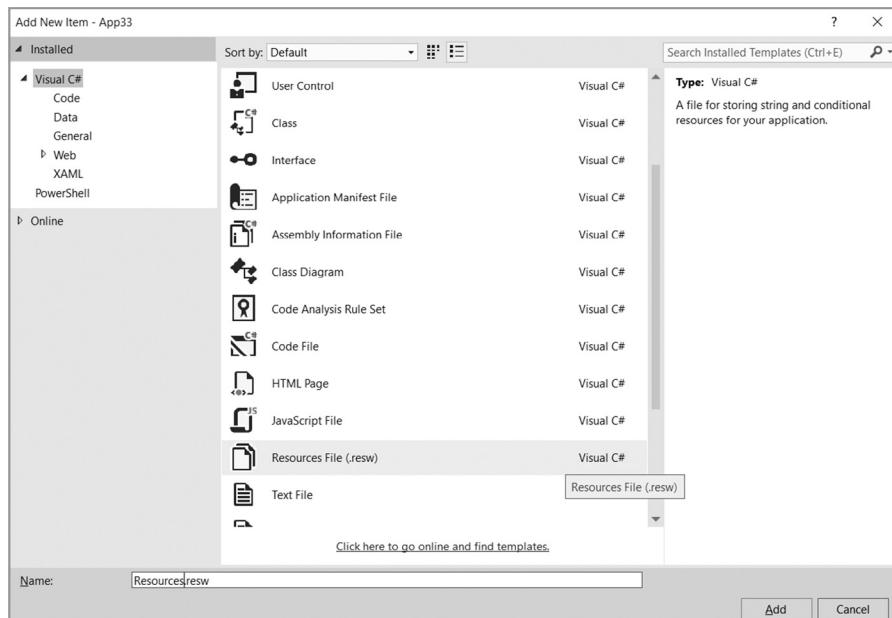
```
<Button Content="Click me!"></Button>
```

Кнопка має властивість **Content**, яка містить текст англійською мовою. А що робити, коли потрібна підтримка кількох мов? Наприклад, локалізація українською мовою? У цьому разі слід подбати про ресурси.

Щоб додати ресурси до програми, слід почати з порожніх файлів ресурсів. Але спершу необхідно визначити, як вказати системі, якої мови стосується кожний файл ресурсів.

Для цього доведеться дотримуватися правила: створювати всередині проекту папки з іменами мовних кодів; у кожній папці потрібно буде розмістити файл **Resources.resw**.

## Windows 10 для C# розробників



Для англійської та української мов буде сформована така структура:



Тепер можна переходити до редагування цих файлів і локалізації вмісту кнопки. Відкрийте обидва файли та додайте однакові імена з нелокалізованим і локалізованим рядками:

Name	Value	Comment
myButton.Content	Click me!	

Name	Value	Comment
myButton.Content	Тисни тут!	

Зверніть увагу, що можна використовувати будь-яке ім'я, але імена властивостей у файлах обох мов мають збігатися.

Змініть код таким чином:

```
<Button x:Uid="myButton"></Button>
```

Ми видалили атрибут контенту з кнопки і додали атрибут **x:Uid**, який є унікальним ідентифікатором нашого рядка ресурсів. Система знайде всі рядки ресурсів із префіксом **myButton** і присвоїть їм оголошені властивості.

На наступному етапі ми повинні змінити маніфест нашої програми, щоб показати, які мови підтримуються. Потрібно знайти елемент ресурсів і змінити його так:

```
<Resources>
    <Resource Language="en-US"/>
    <Resource Language="uk-UA"/>
</Resources>
```

Ось і все. Відтепер програма підтримує дві мови (принаймні для кнопки).

Перевірити різні локалізації програми можна за допомогою сторінки **Language** у вікні **Control Panel**.

Перша мова в списку – це стандартна мова вашої системи, яка буде використовуватись у програмі.

Додатково у файлах ресурсів можна визначити прості рядки, які використовуються в коді C#. Наведений код поверне значення рядка з ім'ям **myString** з поточного файлу ресурсів:

```
ResourceLoader rl = new ResourceLoader();
string s = rl.GetString("myString");
```

Звичайно, якщо є багато рядків для локалізації та багато мов, які потрібно підтримувати, переклад файлу ресурсів і його підтримка можуть бути ускладнені. Тому доцільно завантажити спеціальний Multilingual App Toolkit – розширення для Visual Studio, що дає можливість працювати з файлами **resw** з використанням удосконаленого інтерфейсу. Відвідайте блог <http://blogs.msdn.com/b/matdev/>, де публікуються останні новини щодо інструментарію розробника та посилання для завантаження.



Розділ 7.

## **Графіка, трансформації та анімації**

## Графічні примітиви

Елементи керування у формі ліній, прямокутників, еліпсів є базовими для побудови інших складніших елементів керування. Саме тому важливо знати всі графічні примітиви, адже ви, можливо, будете використовувати ці знання для створення абсолютно нових форм користувацьких елементів керування.

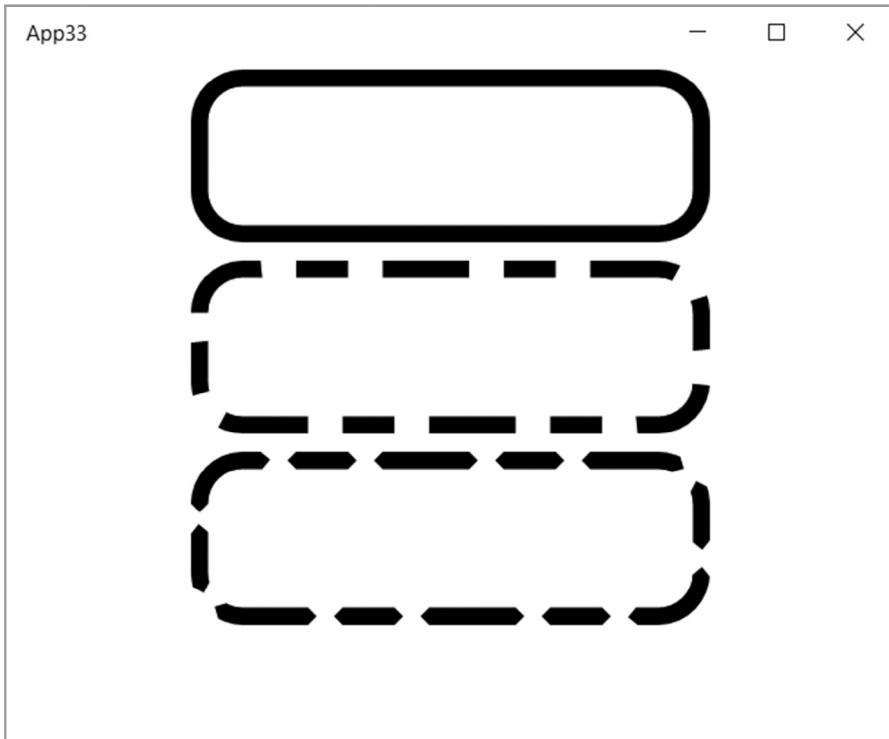
Universal Windows Platform дає змогу використовувати такі графічні примітиви: **Rectangle**, **Ellipse**, **Line**, **Polygon**, **Polyline** і **Path**. Усі ці примітиви успадковані від класу **Shape**, що є нащадком **FrameworkElement**. Завдяки цьому всі графічні примітиви можуть бути розташовані всередині будь-якого контейнера і мати багато властивостей, спільних для всіх елементів керування. Звичайно, клас **Shape** має певні особливі властивості:

- **Stroke** – дає змогу визначити пензель для контуру фігури;
- **Fill** – визначає пензель, що зафарбовує простір всередині фігури;
- **Stretch** – дає можливість визначити, як заповнювати доступний простір всередині контейнера.

Для контуру фігури клас **Shape** містить вісім властивостей, зокрема **StrokeDashArray**, **StrokeDashCap** тощо. Завдяки цим властивостям ви можете визначати контур фігури багатьма різними способами. Подивімось на такий код:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Rectangle Width="300" Height="100" Stroke="Black"
        StrokeThickness="10"
        RadiusX="25" RadiusY="25" Margin="5"></Rectangle>
    <Rectangle Width="300" Height="100" Stroke="Black"
        StrokeThickness="10" RadiusX="25" RadiusY="25"
        StrokeDashArray="5,2,3,2" Margin="5"></Rectangle>
    <Rectangle Width="300" Height="100" Stroke="Black"
        StrokeThickness="10" RadiusX="25" RadiusY="25"
        StrokeDashArray="5,2,3,2" StrokeDashCap="Triangle"
        Margin="5"></Rectangle>
</StackPanel>
```

Виконуючи цей код, ми бачимо три прямокутники з різними контурами:

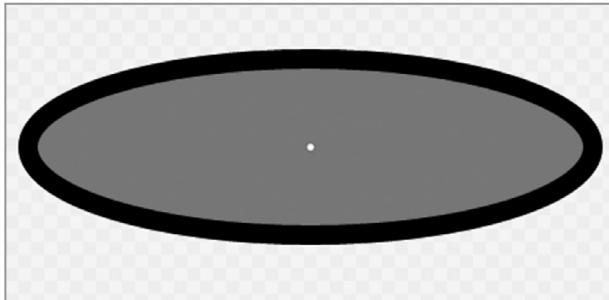


Як бачите, щоб намалювати прямокутник, достатньо визначити властивості **Width** і **Height**. Але адаптувати прямокутник до різної розмітки можна також за допомогою властивостей **MaxWidth**, **MaxHeight**, **MinWidth**, **MinHeight** і **Stretch**. Крім того, можна знайти властивості **RadiusX** і **RadiusY**, які визначені безпосередньо в класі **Rectangle**. За допомогою цих властивостей можна визначити еліпс, що заокруглює кути прямокутника.

За допомогою класу **Ellipse** можна малювати еліпс і коло, задаючи властивості розміру еліпса **Width** і **Height**:

```
<Ellipse Width="300" Height="100" StrokeThickness="10"
         Stroke="Black" Fill="Red">
</Ellipse>
```

У цьому прикладі ми використовуємо властивість **Fill** для надання еліпсу червоного кольору і **StrokeThickness** для визначення товщини контуру:



Наступним графічним примітивом є **Line**, який дає змогу малювати звичайну лінію, задаючи початкову та кінцеву точки:

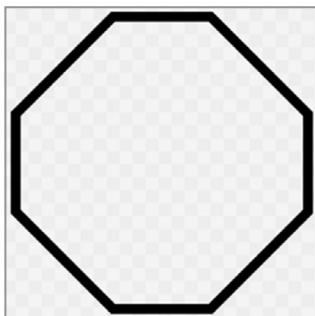
```
<Line X1="0" Y1="0" X2="100" Y2="100" Stroke="Black"></Line>
```

Найважливіше питання тут: де початок координат? Він розташований у верхньому лівому куті контейнера. Якщо ви використовуєте **StackPanel** чи **Grid**, ці контейнери можуть розмістити лінію без будь-яких проблем, порівняно з попередніми версіями Windows Runtime.

Наступний приклад демонструє, як використовувати елемент **Polygon**, який дає змогу створювати замкнений контур:

```
<Polygon Points="0,50,50,0,100,0,150,50,150,100,100,150,50,
150,0,100"
        Stroke="Black" StrokeThickness="5">
</Polygon>
```

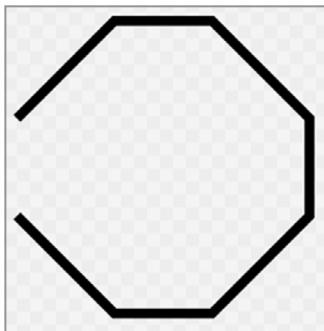
Ми користуємося властивістю **Points** для визначення всіх вершин фігури. Зверніть особливу увагу, що важливим є порядок зазначення вершин.



Ламана – елемент, схожий на попередній, але вона не замикає контур автоматично. Давайте подивимося на ламану з тим самим набором точок:

```
<Polyline Points="0,50,50,0,100,0,150,50,150,100,100,150,50,
150,0,100"
    Stroke="Black" StrokeThickness="5">
</Polyline>
```

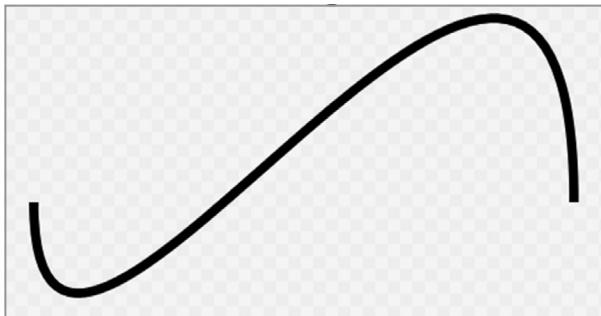
Ви побачите таку форму:



Нарешті розгляньмо графічний примітив **Path**. Завдяки цьому елементу можна малювати лінії будь-якої складності. Він має тільки одну важливу властивість – **Data**, у значенні якої можуть використовуватися спеціальні команди. Подивіться на такий приклад:

```
<Path Stroke="Black" StrokeThickness="5"
    Data="M 10,100 C 10,300 300,-200 300,100">
</Path>
```

Після виконання цього коду бачимо таку фігуру:



Як бачите, ми використовували кілька команд, наприклад **M** і **C**, для створення фігури. Кожна команда має певні параметри, які потрібно розмістити відразу після неї. Елемент **Path** підтримує такі команди:

- **M** – дає змогу перейти до обраної точки.
- **L** – малює лінію від поточної точки до заданої.
- **H** – малює горизонтальну лінію певної довжини від поточної точки.
- **V** – малює вертикальну лінію певної довжини від поточної точки.
- **C** – використовує три точки як параметри і дає змогу будувати криву Безье на основі поточної і третьої точки. Всі інші точки є допоміжними.
- **Q** – використовує дві точки як параметри і дає змогу будувати квадратичну криву Безье на основі поточної і другої точки.
- **S** – ви можете передати дві точки як параметри, щоб відобразити згладжену кубічну криву Безье.
- **T** – дає змогу будувати згладжену квадратичну криву Безье.
- **A** – дає змогу будувати еліптичну криву, використовуючи п'ять параметрів: радіус, кут, тип сегмента (0 або 1), напрямок для кута (0 або 1) і кінцеву точку.
- **Z** – замикає контур, проводячи лінію між першою і поточною точками.

От і всі графічні примітиви. Давайте поговоримо про пензлі, які ви можете використовувати, щоб намалювати контур або заповнити простір всередині форми.

## Пензлі

### **SolidColorBrush**

У цьому і в попередніх розділах ми використовували деякі властивості на кшталт **Fill**, **Background**, **Foreground** тощо. Всі вони визначені як властивості типу **Windows.UI.Xaml.Media.Brush**. Цей тип є базовим для декількох класів, і ми не зираємося використовувати його безпосередньо, а візьмемо похідні типи, зокрема **SolidColorBrush** чи **ImageBrush**. Почнемо огляд з найпростішого пензля – **SolidColorBrush**.

**SolidColorBrush** – це найпростіший пензель. Він дає змогу визначити чистий колір за допомогою властивості **Color**:

```
<Rectangle Width="300" Height="100" Fill="Red"></Rectangle>
<Rectangle Width="300" Height="100">
    <Rectangle.Fill>
        <SolidColorBrush Color="Red"></SolidColorBrush>
    </Rectangle.Fill>
```

```
</Rectangle>
<Rectangle Width="300" Height="100">
    <Rectangle.Fill>
        <SolidColorBrush Color="#FFFF0000"></SolidColorBrush>
    </Rectangle.Fill>
</Rectangle>
```

Всі ці три прямокутники мають одинаковий колір. Але завдяки вбудованим конвертерам можна визначити **SolidColorBrush** як рядок за допомогою одного з попередньо визначених рядків. Те саме можна зробити з властивістю **Color** і, нарешті, можна визначити колір у форматі RGB.

Якщо ви хочете побачити всі попередньо визначені кольори, просто скористайтеся Visual Studio IntelliSense або перевірте клас **Windows.UI.Colors**.

Звичайно, можна створювати об'єкти **SolidColorBrush** за допомогою коду C# і присвоювати значення їхнім властивостям під час виконання:

```
rect.Fill = new SolidColorBrush(Colors.Red);
```

## LinearGradientBrush

**LinearGradientBrush** дає змогу створити пензель, який базується на градієнті. За замовчуванням градієнт розраховується від верхнього лівого кута до правого нижнього та дає змогу показати плавний перехід від одного кольору до іншого.

```
<Rectangle Width="300" Height="300">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Red" Offset="0">
            </GradientStop>
            <GradientStop Color="Green" Offset="1">
            </GradientStop>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Напрямок градієнта можна легко змінити за допомогою **StartPoint** і **EndPoint**, але слід пам'ятати, що необхідно використовувати нормалізований прямокутник:

```
<Rectangle Name="rect" Width="300" Height="300">
    <Rectangle.Fill>
        <LinearGradientBrush StartPoint="0,1" EndPoint="1,0">
            <GradientStop Color="Red" Offset="0">
            </GradientStop>
            <GradientStop Color="White" Offset="0.5">
            </GradientStop>
            <GradientStop Color="Green" Offset="1">
            </GradientStop>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

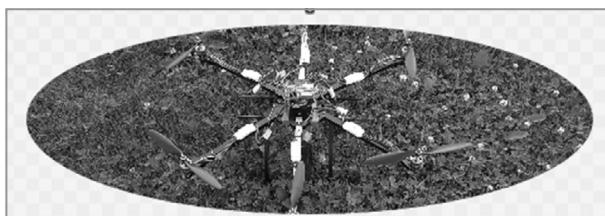
Цей код не тільки змінює напрямок градієнта, але й додає ще кілька кольорів. Отже, наш градієнт змінюється з червоного на білий і з білого на зелений кольори.

## ImageBrush

Для заповнення об'єктів можна використовувати не лише колір, але й зображення. Це можна зробити за допомогою елемента **ImageBrush**, який містить дві основні властивості: **ImageSource** і **Stretch**. Перша використовується, щоб вказати зображення, а друга – для визначення способу заповнення об'єкта.

```
<Ellipse Width="300" Height="100">
    <Ellipse.Fill>
        <ImageBrush ImageSource="Assets/drone.jpg"
                    Stretch="UniformToFill"></ImageBrush>
    </Ellipse.Fill>
</Ellipse>
```

У цьому прикладі для властивості **Stretch** ми використовуємо значення **UniformToFill**. Це показує, що ми хочемо, по-перше, зберегти пропорції між сторонами зображення, а по-друге – заповнити весь простір всередині об'єкта. Ви побачите цю картинку навіть у конструкторі:



## WebViewBrush

Наступний пензель – **WebViewBrush**. Він може використовувати елемент керування **WebView** як джерело зображення. Сьогодні цей елемент керування непопулярний, але оскільки він є, ми вирішили також його оглянути. Давайте розглянемо такий код:

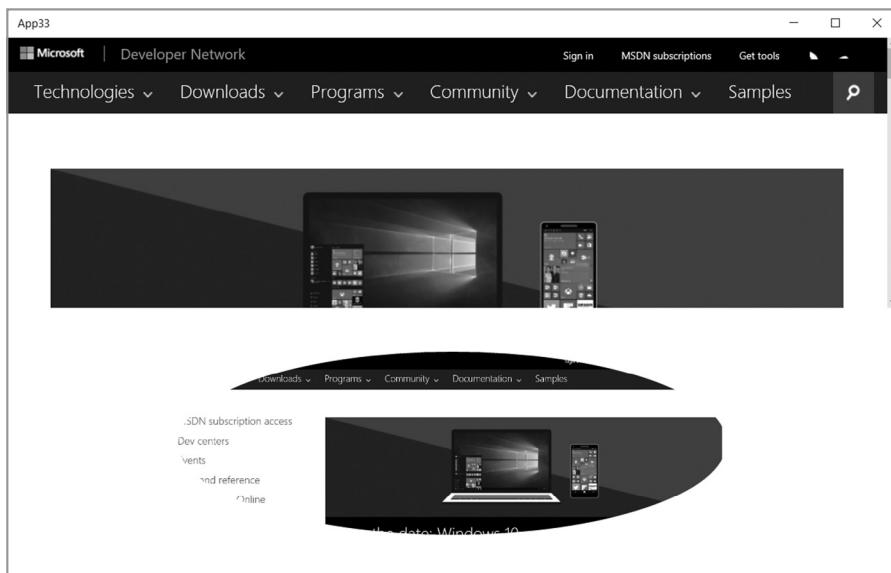
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <WebView Source="http://dev.microsoft.com" Name="webView"
        NavigationCompleted="webView_NavigationCompleted">
    </WebView>

    <Ellipse Width="600" Height="200" Margin="10" Grid.Row="1">
        <Ellipse.Fill>
            <WebViewBrush x:Name="webBrush" SourceName="webView"
                Stretch="UniformToFill">
            </WebViewBrush>
        </Ellipse.Fill>
    </Ellipse>
</Grid>
```

У першому рядку сітки ми створили елемент керування **WebView** і використали його як джерело для **WebViewBrush**. Але існує проблема, **WebView** потрібен час на завантаження вмісту: і в цей самий час слід застосувати **WebViewBrush**. Таким чином, ми повинні перемалювати **WebViewBrush**. У цьому випадку краще буде реалізувати обробник події **NavigationCompleted**:

```
private void webView_NavigationCompleted(WebView sender,
    WebViewNavigationCompletedEventArgs args)
{
    webBrush.Redraw();
}
```

Після виконання цього коду можна побачити таку картину:



## Геометричні фігури

Universal Windows Platform визначає групу класів, які можна використовувати замість складних команд для присвоєння значень властивості **Data** елемента **Path**. Найпростішими з цих класів є:

- **LineGeometry** – використовує дві точки, щоб намалювати лінію.
- **EllipseGeometry** – визначає еліпс. Параметрами є довжини півосей та координати центра еліпса.
- **RectangleGeometry** – дає можливість намалювати прямокутник.

Подивіться на такий код:

```
<Path Stroke="Black" StrokeThickness="2" >
    <Path.Data>
        <EllipseGeometry RadiusX="100" RadiusY="100"
            Center="100,100"/>
    </Path.Data>
</Path>
```

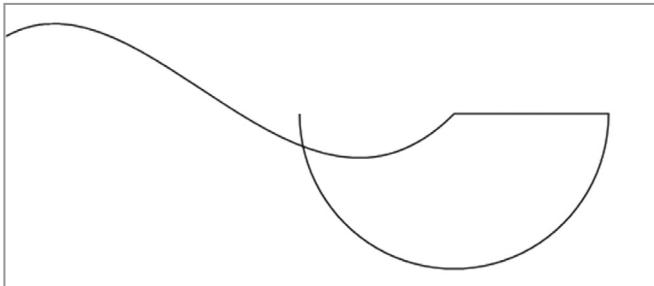
Після виконання цього коду ви побачите просте коло.

Для створення більш складних об'єктів можна використовувати елемент **PathGeometry**. Цей елемент дає змогу визначати групи простих об'єктів – відрізків, які можуть утворювати групи за допомогою елементу **PathFigure**. Сегменти представлені такими класами: **ArcSegment**, **LineSegment**, **BezierSegment**, **PolyBezierSegment**, **QuadraticBezierSegment** і **PolyQuadraticBezierSegment**.

Подивімося, як використовувати ці об'єкти, на прикладі коду:

```
<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>
```

Після виконання цього коду бачимо таку криву:



Крім того, якщо потрібно згрупувати кілька геометричних об'єктів, можна використати клас **GeometryGroup**.

## Трансформації

Трансформації та анімація – це найцікавіші теми, вони дають можливість застосовувати багато різних ефектів в користувацьких елементах керування і допоможуть зробити ваш інтерфейс більш чутливим до дій користувача. Почнемо цю тему з огляду основних трансформацій і завершимо вбудованими анімаціями в наявних елементах керування.

## Основні типи трансформацій

Якщо ви відкриєте Object Browser і подивитесь на клас **UIElement**, то знайдете властивість **RenderTransform**. Вона описана як властивість типу **Transform**. Відкривши цей тип, ви побачите декілька похідних класів, за допомогою яких можна застосовувати основні трансформації до користувацьких елементів керування. Ось ці класи:

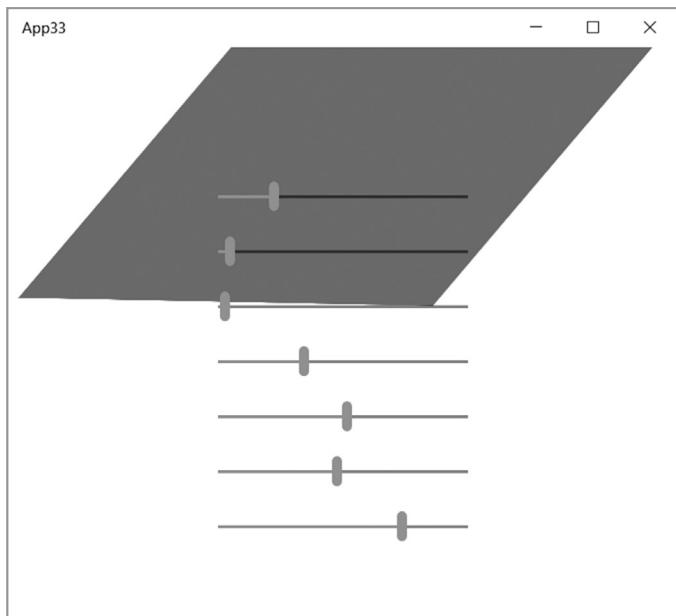
- **RoateTransform**
- **ScaleTransform**
- **SkewTransform**
- **TranslateTransform**
- **MatrixTransform**

Щоб застосувати будь-яке з цих перетворень, потрібно просто створити об'єкт обраного типу і присвоїти його властивості **RenderTransform**. Якщо необхідно присвоїти декілька трансформацій водночас, їх можна поєднати за допомогою класу **TransformGroup**. Розгляньмо простий приклад:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Rectangle Width="200" Height="100" Fill="Red">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <RotateTransform Angle=
                    "{Binding Value, ElementName=rotateSlider,
                    Mode=OneWay}"
                    CenterX="100" CenterY="50">
                </RotateTransform>
                <ScaleTransform CenterX="100" CenterY="50"
                    ScaleX="{Binding Value,
                    ElementName=scaleXSlider, Mode=OneWay}"
                    ScaleY="{Binding Value,
                    ElementName=scaleYSlider, Mode=OneWay}">
                </ScaleTransform>
                <SkewTransform CenterX="100" CenterY="50">
                </SkewTransform>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
</StackPanel>
```

```
        AngleX="{Binding Value,
            ElementName=skewXSlider, Mode=OneWay}"
        AngleY="{Binding Value,
            ElementName=skewYSlider, Mode=OneWay}">
    </SkewTransform>
    <TranslateTransform
        X="{Binding Value, ElementName=translateXSlider,
            Mode=OneWay}"
        Y="{Binding Value, ElementName=translateYSlider,
            Mode=OneWay}">
    </TranslateTransform>
</TransformGroup>
</Rectangle.RenderTransform>
</Rectangle>
<Slider Width="200" Name="rotateSlider" Minimum="0"
    Maximum="360">
</Slider>
<Slider Width="200" Name="scaleXSlider" Minimum="0"
    Maximum="100" Value="1">
</Slider>
<Slider Width="200" Name="scaleYSlider" Minimum="0"
    Maximum="100" Value="1">
</Slider>
<Slider Width="200" Name="skewXSlider" Minimum="-180"
    Maximum="180" Value="0">
</Slider>
<Slider Width="200" Name="skewYSlider" Minimum="-180"
    Maximum="180" Value="0">
</Slider>
<Slider Width="200" Name="translateXSlider"
    Minimum="-100" Maximum="100" Value="0">
</Slider>
<Slider Width="200" Name="translateYSlider"
    Minimum="-100" Maximum="100" Value="0">
</Slider>
</StackPanel>
```

Після виконання цього коду ви побачите прямокутник і декілька повзунків. За допомогою цих повзунків до прямокутника можна застосувати різні види трансформацій:



Зверніть особливу увагу, що трансформації не впливають на розмітку і всі контейнери розраховують необхідну площину на основі вихідних значень без застосованих трансформацій.

## CompositeTransform

Ще один вид трансформації, **CompositeTransform**, поєднує в собі всі види трансформацій. Завдяки цьому типу можна спростити роботу з кількома трансформаціями. І, звичайно, всі вони повинні виконуватися навколо одного центру. Просто перепишемо попередній елемент **Rectangle**, використовуючи цей тип:

```
<Rectangle Width="200" Height="100" Fill="Red">
    <Rectangle.RenderTransform>
        <CompositeTransform CenterX="100" CenterY="50"
            ScaleX="{Binding Value, ElementName=scaleXSlider,
            Mode=OneWay}"
            ScaleY="{Binding Value, ElementName=scaleYSlider,
            Mode=OneWay}"
            Rotation="{Binding Value, ElementName=rotateSlider,
            Mode=OneWay}"
            SkewX="{Binding Value, ElementName=skewXSlider,
            Mode=OneWay}">
```

```

        SkewY="{Binding Value, ElementName=skewYSlider,
        Mode=OneWay}"
        TranslateX="{Binding Value,
        ElementName=translateXSlider, Mode=OneWay}"
        TranslateY="{Binding Value,
        ElementName=translateYSlider, Mode=OneWay}">
    </CompositeTransform>
</Rectangle.RenderTransform>
</Rectangle>

```

## 3D-трансформація

Якщо ви розробляєте програми для Windows Phone 8.1 або Windows 8.1, можна використовувати клас **PlaneProjection** (досі доступний), який дає змогу застосувати 3D-трансформацію до будь-яких елементів інтерфейсу користувача. Але **PlaneProjection** дуже обмежений – за допомогою нього можна зробити лише поворот. Якщо потрібно застосувати більш складні 3D-перетворення, необхідно використовувати **MatrixTransform** і реалізовувати власні математичні алгоритми.

Але, починаючи з Windows 10, з'явився спосіб виконати більш складні 3D-перетворення, використовуючи нові класи, зокрема **PerspectiveTransform3D** і **CompositeTransform3D**. Розглянемо такий код:

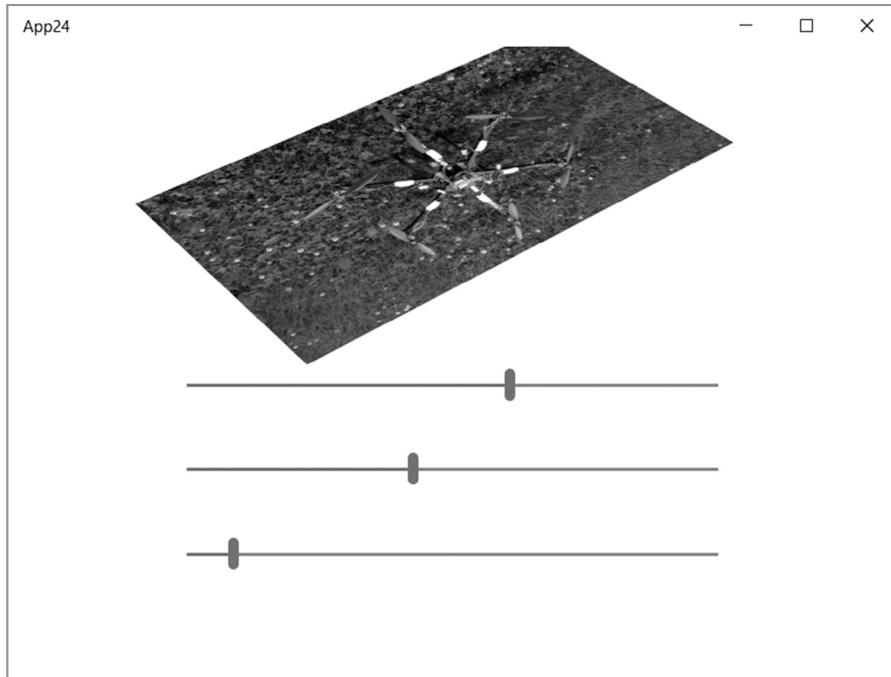
```

<RelativePanel HorizontalAlignment="Center">
    <RelativePanel.Transform3D>
        <PerspectiveTransform3D></PerspectiveTransform3D>
    </RelativePanel.Transform3D>
    <Image Source="Assets\drone.jpg" Width="400" Name="image">
        <Image.Transform3D>
            <CompositeTransform3D CenterX="200" CenterY="100"
                RotationX="{x:Bind sliderX.Value,Mode=OneWay}"
                RotationY="{x:Bind sliderY.Value,Mode=OneWay}"
                RotationZ="{x:Bind sliderZ.Value,Mode=OneWay}">
            </CompositeTransform3D>
        </Image.Transform3D>
    </Image>
    <Slider Maximum="360" RelativePanel.Below="image"
        Name="sliderX" Width="400" Margin="0,10,0,10"></Slider>
    <Slider Maximum="360" RelativePanel.Below="sliderX"
        Name="sliderY" Width="400" Margin="0,10,0,10"></Slider>
    <Slider Maximum="360" RelativePanel.Below="sliderY">

```

```
Name="sliderZ" Width="400" Margin="0,10,0,10">></Slider>  
</RelativePanel>
```

Після виконання коду бачимо екран, наведений нижче, де можна обертати зображення, використовуючи три повзунки:



Відверто кажучи, щось подібне можна було б зробити і за допомогою **PlaneProjection**, але з цього прикладу ми можемо зрозуміти синтаксис нових елементів.

Перш за все, необхідно використати елемент **PerspectiveTransform3D** і присвоїти його властивості **Transform3D**, що доступна для всіх елементів інтерфейсу користувача. **PerspectiveTransform3D** дає змогу оголосити порт загального перегляду для всіх дочірніх елементів. Отже, це слід уявляти, як камеру з перспективою. Можна переміщати камеру за допомогою властивостей **OffsetX** і **OffsetY**, але цей рух не вплине на позицію елементу керування всередині контейнера – він впливає лише на трансформацію, тому що у вас є той самий порт перегляду і будь-який рух камери додасть кут, який впливатиме тільки на проекцію. **PerspectiveTransform3D** також має властивість **Depth**, яка визначає

відстань між камерою і площею  $Z=0$ . За замовчуванням **Depth** дорівнює 1000, але ви можете змінити її значення, і це теж вплине лише на перетворення. Відверто кажучи, ми виявили, що, якщо змінити **Depth**, можна отримати непередбачувані результати.

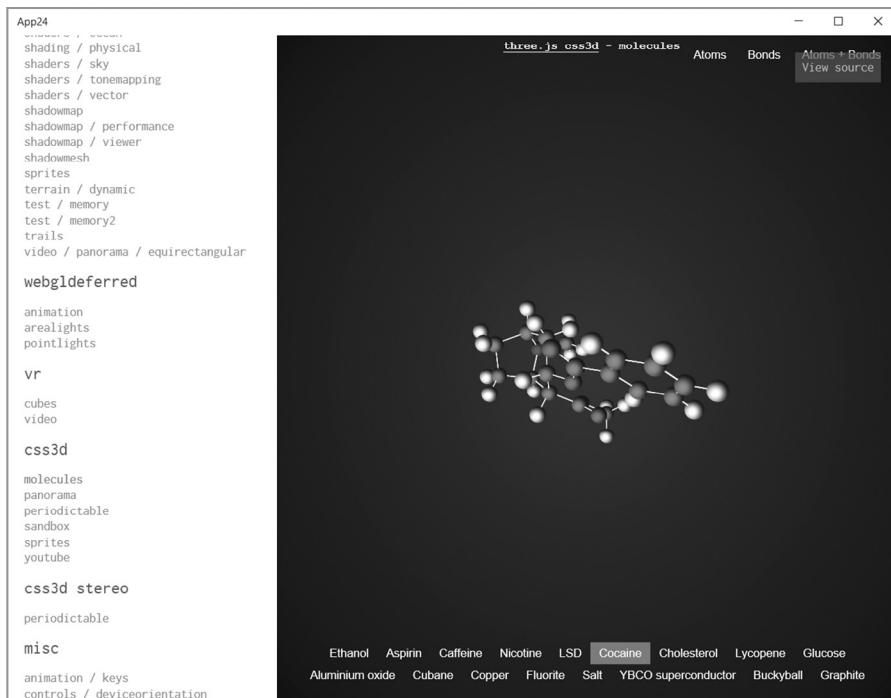
Після застосування **PerspectiveTransform3D** до будь-якого контейнера можна використовувати **CompositeTransform3D**, щоб застосувати будь-які перетворення до елементів всередині порту перегляду. Користувач має змогу масштабувати, обертати і перекручувати елементи інтерфейсу.

Водночас ми виявили два недоліки.

Перш за все, **CompositeTransform3D** дає змогу задати центр трансформації, але для цього необхідно вираховувати пікселі. Тобто, якщо ми хочемо виконати перетворення навколо центру елементу керування, потрібно обчислити його фактичний розмір. Це дивно, тому що навіть **PlaneProjection** дає змогу встановлювати центр у відносних координатах ((0.5, 0.5) – центр).

Другий недолік – це проблема з правильним розміщенням елементів керування в просторі – елементи оброблюються один за одним відповідно до порядку зазначення в XAML. Тож елементи, розташовані найближче один до одного, можуть перекриватися елементами, які розміщені не так близько. MSDN рекомендує використовувати обхідний шлях (**Canvas.ZIndex**), але це вимагає застосування складного коду, який має динамічно змінювати **ZIndex**. Крім того, є багато завдань, у виконанні яких **ZIndex** не допомагає. Так, в поточній версії ви можете використовувати ці класи для більш складних сценаріїв.

Якщо потрібно побудувати складні 3D-моделі, рекомендуємо використовувати **WebView** і реалізувати модель у ньому за допомогою CSS 3D. Починаючи з Windows 10, **WebView** базується на Microsoft Edge і підтримує збереження 3D-значень для 3D-трансформацій.



## Основи анімації

### Основні типи анімації

За допомогою анімації можна створювати гарні візуальні ефекти, анімуючи будь-які властивості залежностей користувачьких елементів керування. І зробити це можна практично без кодування на C#.

Зробимо огляд основних типів анімації:

- **DoubleAnimation** – можна використовувати для анімації властивостей типу **double**. Найбільш важливі властивості – це **From** і **To**, за допомогою яких оголошують початкове і кінцеве значення властивості анімації.
- **ColorAnimation** – застосовується для анімації властивостей типу **Color**, зазвичай для пензлів.
- **PointAnimation** – дає змогу анімувати точку. Можна використовувати властивості **From** і **To**, щоб призначити початкову та кінцеву точки руху.

Незалежно від типу анімації, всі об'єкти анімації мають такі властивості:

- **Duration** – ця властивість дає змогу ініціалізувати тривалість анімації у форматі hh:mm:ss (наприклад, 00:00:03 – 3 секунди). Всі анімовані властивості змінюються рівномірно протягом заданого періоду часу.
- **BeginTime** – якщо ви не збираєтесь починати певну анімацію в групі відразу, можна задати цю властивість, що дає змогу визначити час від термінування анімації. Для неї використовується той самий формат часу.
- **SpeedRatio** – використовуючи цю властивість, можна підвищити (знизити) швидкість анімації в **n** разів, де **n** – це значення типу **double**. За замовчуванням **SpeedRatio** дорівнює 1.
- **AutoReverse** – за допомогою цієї властивості анімації можна відтворювати в обох напрямках. Якщо **AutoReverse** присвоїти істинне значення, процес анімації складатиметься з двох частин: усі властивості будуть анімовані від початкового значення (**From**) до фінального (**To**), а потім – від фінального (**To**) до початкового (**From**).
- **RepeatBehavior** – за допомогою цієї властивості можна встановити спосіб повторення анімації, вказавши одне з таких значень: час в секундах, протягом якого анімація має повторюватися; **Forever** – нескінчена кількість повторень; **Count** – кількість циклів (можна задати як **Nx**, де **N** – це кількість циклів).

Звичайно, для того, щоб створити вдалий ефект, як правило, необхідно використовувати кілька типів анімації одночасно, або принаймні використати одну анімацію для різних властивостей. Ось чому анімації не містять методів, що дають змогу запускати їх або зупиняти, крім того, немає можливості розмістити анімації як окремі об'єкти. Усі анімації повинні бути упаковані в спеціальний об'єкт, який можна створити завдяки класу **Storyboard**. Також анімації використовують **Storyboard.TargetName** і **Storyboard.TargetProperty** як додаткові властивості для визначення імені об'єкту і властивості, що анімуватиметься. Перегляньмо, як створити просту анімацію:

```
<Canvas >
    <Button Name="myButton" Content="Hello">
        <Button.Resources>
            <Storyboard x:Key="buttonAnimation"
                x:Name="buttonAnimation">
                <DoubleAnimation
                    Storyboard.TargetName="myButton"
                    Storyboard.TargetProperty="(Canvas.Left)"
                    To="200" Duration="0:0:5" AutoReverse="True" />
            </Storyboard>
        </Button.Resources>
    </Button>
</Canvas>
```

```
</Button>  
</Canvas>
```

Як бачите, ми використали ресурси для зберігання об'єкту **Storyboard**. Оскільки **Storyboard** – невізуальний об'єкт, немає жодного способу включити його до візуального дерева, але ресурси підходять для будь-якого невізуального об'єкта.

## Як запускати розкадрування

Є два способи запуску розкадрування **Storyboard**:

- запуск анімації з коду C#;
- запуск анімації із XAML, коли об'єкт завантажений.

У разі запуску із C# можна просто викликати метод розкадрування **Begin**. Це дуже універсальний підхід, тому що дає змогу зберігати анімацію як ресурси. Викликати **Begin** ви можете, коли забажаєте.

```
protected override void OnNavigatedTo(NavigationEventArgs e)  
{  
    buttonAnimation.Begin();  
    base.OnNavigatedTo(e);  
}
```

Крім того, можна використовувати методи **Stop** і **Pause**, що актуально для тривалих анімацій. Якщо потрібно запустити код, коли розкадровка завершує виконання, можна створити обробник події **Completed**.

Існує ще один підхід: можна розмістити анімацію в спеціальному об'єкті класу **EventTrigger**. Насправді цей об'єкт реалізує своєрідний тригер, який буде спрацьовувати під час кожного завантаження користувачького елемента керування. Використовуючи **EventTrigger**, можна переписати код таким чином:

```
<Canvas >  
    <Button Name="myButton" Content="Hello">  
        <Button.Triggers>  
            <EventTrigger RoutedEvent="UserControl.Loaded">  
                <BeginStoryboard>  
                    <Storyboard x:Name="buttonAnimation">  
                        <DoubleAnimation  
                            Storyboard.TargetName="myButton"  
                            Storyboard.TargetProperty="(Canvas.Left)"
```

```

        To="200" Duration="0:0:5" AutoReverse="True" />
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Button.Triggers>
</Button>
</Canvas>

```

## Анімація за допомогою ключових кадрів

Всі стандартні типи анімації змінюють властивості користувачьких елементів керування рівномірно, але якщо ви хочете робити це за складнішим алгоритмом, краще використовувати такі анімації:

- **DoubleAnimationUsingKeyFrames**
- **PointAnimationUsingKeyFrames**
- **ColorAnimationUsingKeyFrames**

Як бачите, імена цих класів подібні до стандартних імен анімацій, але з суфіксом **UsingKeyFrame**. Ідея цих анімацій полягає в тому, що ви можете поділити період часу на кілька інтервалів і застосовувати до кожного інтервалу власні параметри й навіть власний алгоритм анімації. Подивімося на такий приклад:

```

<Canvas >
    <Button Name="myButton" Content="Hello">
        <Button.Resources>
            <Storyboard x:Key="buttonAnimation"
                       x:Name="buttonAnimation">
                <DoubleAnimationUsingKeyFrames Duration="0:0:5"
                    AutoReverse="True"
                    Storyboard.TargetName="myButton"
                    Storyboard.TargetProperty="(Canvas.Left)">
                    <LinearDoubleKeyFrame KeyTime="0:0:2"
                        Value="70"></LinearDoubleKeyFrame>
                    <LinearDoubleKeyFrame KeyTime="0:0:5"
                        Value="100"></LinearDoubleKeyFrame>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </Button.Resources>
    </Button>
</Canvas>

```

З цього прикладу видно, що ми ділимо період часу на два інтервали (2 і 3 секунди) і використовуємо об'єкт **LinearDoubleKeyFrame** для визначення алгоритму анімації. Крім того, можна використовувати параметри **DiscreteDoubleKeyFrame** і **SplineDoubleKeyFrame**. Перший дає змогу визначити дискретну анімацію, а другий – анімацію за допомогою кривої Безье.

Ще один тип анімації в цій групі – **ObjectAnimationUsingKeyFrame**. Цей тип дає змогу змінювати будь-які властивості користувачьких елементів керування. Оскільки тип властивості невідомий, можливо використовувати лише дискретні кадри.

## Функції спрощення

Навіть для анімації з ключовими кадрами необхідно написати багато коду, щоб створити щось складне. Наприклад, якщо я хочу намалювати м'яч, який впав на землю, мені потрібно використати величезну кількість об'єктів **LinearDoubleKeyFrame**, оскільки м'яч у реальному житті не буде зупинятися на землі відразу, а відіб'ється кілька разів. Ось чому універсальна платформа Windows підтримує декілька класів, що реалізують «спрощення» анімації. Подивіться на код нижче – він показує, як створити анімацію м'яча, що стрибає:

```
<Canvas x:Name="LayoutRoot" Background="White">
    <Canvas.Resources>
        <Storyboard x:Name="sb1">
            <DoubleAnimation From="0" To="250"
                Storyboard.TargetName="ell1"
                Storyboard.TargetProperty="(Canvas.Top)"
                Duration="0:0:5">
                <DoubleAnimation.EasingFunction>
                    <BounceEase Bounces="10"
                        EasingMode="EaseOut" Bounciness="2">
                    </BounceEase>
                </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
        </Storyboard>
    </Canvas.Resources>
    <Ellipse Fill="Blue" Width="50" Height="50" x:Name="ell1">
    </Ellipse>
</Canvas>
```

У цьому випадку ми використовували елемент **BounceEase**, який дає змогу реалізувати функцію «стрибка». Універсальна платформа Windows підтримує 11 типів функцій спрощення: **ExponentialEase**, **PowerEase**, **QuadraticEase**, **BackEase**,

**BounceEase**, **CircleEase**, **CubicEase**, **ElasticEase**, **QuarticEase**, **QuinticEase**, **SineEase**. Звичайно, кожна з них має власні параметри і найкращий спосіб зрозуміти їх усі – спробувати скористатися функціями.

## Вбудовані анімації

Отже, основні анімації ми розглянули, однак Universal Windows Platform містить ще спеціальну групу вбудованих анімацій, які можуть бути застосовані до користувальських елементів керування відносно інших елементів керування в колекції. Ці анімації можна використовувати з будь-якими контейнерами і користувальськими елементами керування, що працюють з колекціями.

### Переходи

Перша група цих анімацій пов'язана з переходами і дає змогу визначати механізм появи елемента керування, коли його додають до колекції або переміщують всередині неї. Є вісім класів:

- **AddDeleteThemeTransition** – дає змогу налаштувати зовнішній вигляд користувальського елемента керування в разі додавання його до колекції або видалення з колекції.
- **ContentThemeTransition** – ця анімація буде працювати, якщо нове значення присвоюється властивості **Content**. Вона буде працювати для першої появи елементу керування, тому що тоді **Content** ініціалізується.
- **EnteranceThemeTransition** – працює, коли користувальський елемент керування відображається вперше. Можна застосувати цей перехід до окремих об'єктів або контейнерів. У випадку контейнерів (панелей) всі дочірні елементи керування будуть анімовані по одному.
- **ReorderThemeTransition** – дає змогу визначити анімацію, яка запускатиметься, якщо елемент змінює положення в колекції. Як правило, це відбувається через операції перетягування.
- **RepositionThemeTransition** – дає змогу визначити анімацію, якщо переміщення відбулося, а елементів керування в контексті не було.
- **EdgeUIThemeTransition** – дає змогу визначити поведінку під час переходу від краю вікна.
- **PaneThemeTransition** – дає змогу пересувати панель.
- **PopupThemeTransition** – дає змогу визначити переходи для спливаючих компонентів елементів керування.

Розглянемо кілька прикладів використання цих анімацій. Почнемо з **ContentThemeTransition**. Оскільки цей тип належить до руху контенту, він має властивості **HorizontalOffset** і **VerticalOffset**. Ці атрибути дають змогу почати

рух елемента керування до кінцевого положення за допомогою зсуву. Погляньте на такий код:

```
<Grid>
    <Button Content="Hello">
        <Button.Transitions>
            <TransitionCollection>
                <ContentThemeTransition
                    HorizontalOffset="100" VerticalOffset="100"/>
            </TransitionCollection>
        </Button.Transitions>
    </Button>
</Grid>
```

Цей код дає змогу анімувати кнопку, коли вона з'являється на екрані (або змінює **Content**). Як бачите, ми застосували анімацію за допомогою властивості **Transitions** цієї кнопки. Фактично ця властивість є колекцією, і в неї можна додати скільки завгодно елементів. Завдяки цьому дуже легко комбінувати декілька типів анімації.

Якщо у вас є контейнер, який містить користувацькі елементи керування, і ви хочете додати ту саму анімацію для всіх, можна використати властивість **ChildrenTransition**. Наступний приклад працює за тим самим принципом, що й попередній:

```
<Grid>
    <Grid.ChildrenTransitions>
        <TransitionCollection>
            <ContentThemeTransition
                HorizontalOffset="100" VerticalOffset="100"/>
        </TransitionCollection>
    </Grid.ChildrenTransitions>
    <Button Content="Hello">
    </Button>
</Grid>
```

Переходи працюватимуть коректно і для графічних примітивів. Наведений далі код дає змогу побачити «зростаючий» прямокутник:

```
<Grid>
    <Rectangle Height="100" Width="100" Fill="Red">
        <Rectangle.Transitions>
            <TransitionCollection>
```

```

        <AddDeleteThemeTransition/>
    </TransitionCollection>
</Rectangle.Transitions>
</Rectangle>
</Grid>

```

Цей перехід не має жодних спеціальних властивостей, елемент керування просто з'являється з «нічого».

Наступний приклад показує, як працювати з **EntranceThemeTransition**:

```

<Grid>
    <Rectangle Height="100" Width="100" Fill="Red">
        <Rectangle.Transitions>
            <TransitionCollection>
                <EntranceThemeTransition
                    FromHorizontalOffset="300"
                    FromVerticalOffset="300" />
            </TransitionCollection>
        </Rectangle.Transitions>
    </Rectangle>
</Grid>

```

Тепер попробуємо з деякими типами переходів динамічно. Створіть просту сторінку:

```

<Grid>
    <StackPanel Name="stk">
        <Rectangle Width="100" Height="100" Fill="Red"></Rectangle>
        <Button Click="Button_Click">Click Me</Button>
    </StackPanel>
</Grid>

```

І реалізуйте такий обробник події:

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    Rectangle r = new Rectangle()
    {
        Height = 100,
        Width = 100,
        Fill = new SolidColorBrush(Colors.Red)
    };

```

```
r.Transitions = new TransitionCollection();
r.Transitions.Add(new ReorderThemeTransition());
stk.Children.Insert(1, r);
}
```

Кожного разу під час натискання кнопки в робочій програмі вона буде створювати прямокутник і призначати йому перехід **ReorderThemeTransition**.

Звичайно, можна визначати переходи як стилі, використовуючи ресурси:

```
<Page.Resources>
    <Style x:Key="DefaultButtonStyle" TargetType="Button">
        <Setter Property="Transitions">
            <Setter.Value>
                <TransitionCollection>
                    <EntranceThemeTransition/>
                </TransitionCollection>
            </Setter.Value>
        </Setter>
    </Style>
</Page.Resources>
```

## Анімації тем

Попередня група вбудованих анімацій дає змогу реалізувати базові ефекти з мінімальними змінами з боку розробників, а наступна – керувати набагато більшою кількістю параметрів, зокрема визначати час початку. Подивімося, які класи у нас є:

- **DragItemThemeAnimation** – дає змогу застосовувати ефекти до елементу керування, готового до перетягування;
- **DragOverThemeAnimation** – дає можливість застосовувати деякі ефекти до елементів керування під час перетягування;
- **DropTargetItemThemeAnimation** – дає змогу показати, що елементи керування можуть бути джерелом перетягування;
- **FadeInThemeAnimation** – дає змогу налаштувати прозорість, коли елемент керування з'являється;
- **FadeOutThemeAnimation** – дає змогу налаштувати прозорість, коли елемент керування зникає;
- **PopInThemeAnimation** і **PopOutThemeAnimation** – ці типи анімації можна використовувати з розширюваними елементами керування;

- **RepositionThemeAnimation** – дає змогу застосовувати ефекти для елементів керування, які змінюють позиції;
- **SplitCloseThemeAnimation** і **SplitOpenThemeAnimation** – як правило, ці анімації можна побачити в складних сценаріях, коли користувач «розділює» вміст за допомогою двох пальців;
- **TapDownThemeAnimation** і **TapUpThemeAnimation** – дає змогу застосовувати ефекти у разі настання події **Tapped**;
- **SwipeHintThemeAnimation** – як правило, цей ефект можна побачити під час проведення по елементу керування;
- **SwipeBackThemeAnimation** – дає змогу визначити поведінку елементу керування, коли користувач завершив проводити по елементу.

Щоб запустити всі ці анімації, необхідно створити розкадрування. З одного боку, це недолік, тому що необхідно написати додатковий код, але з іншого боку, це добре, тому що можна визначати час анімації і використовувати її для власних сценаріїв. Розглянемо короткий приклад:

```
<StackPanel>
    <StackPanel.Resources>
        <Storyboard x:Name="myStoryboard">
            <FadeOutThemeAnimation
                Storyboard.TargetName="myRectangle" />
        </Storyboard>
    </StackPanel.Resources>
    <Rectangle PointerPressed="Rectangle_Tapped"
        x:Name="myRectangle"
        Fill="Red" Width="100" Height="100" />
</StackPanel>
```

Цей код оголошує розкадрування, але для того, щоб його запустити, ми повинні реалізувати такий обробник події:

```
private void Rectangle_Tapped(object sender,
    PointerRoutedEventArgs e)
{
    myStoryboard.Begin();
}
```

Схоже, що розмову про анімації, перетворення і графічні примітиви завершено. Це був найдовший розділ у книзі.



Розділ 8.

## **Зв'язування даних**

Ми вже знаємо, як створити базові інтерфейси за допомогою основних елементів керування, тому зараз можемо розглянути, як їх використовувати для відображення реальних даних. Звичайно, «відображення» – не зовсім правильне слово, тому що, як правило, ми повинні виконувати набагато складніші завдання, ніж відображення даних із пам'яті. Ось загальні задачі:

- Відобразити поточний стан об'єктів у пам'яті. У такому разі ми не збираємося відстежувати, що відбудеться з об'єктами потім.
- Відобразити поточні значення, що збережені в об'єктах пам'яті. Тоді, якщо деякі об'єкти будуть оновлюватися, потрібно буде оновити також інтерфейс.
- Дозволити користувачам вносити зміни, використовуючи інтерфейс. Вони повинні впливати на об'єкти в пам'яті, а також змінювати їхні стани і поля.
- Змінити стан деяких користувацьких елементів керування в інтерфейсі на основі інших елементів на тій же сторінці.
- Змінити обробники подій і шаблони, що базуються на стані об'єктів в пам'яті.
- Змінити властивості об'єкта на зручніші для зчитування, коли ми відображаємо їх на сторінці. Якщо користувач змінює дані за допомогою інтерфейсу, ми повинні гарантувати зворотне перетворення.

Звичайно, щоб виконати ці завдання, нам потрібно розглянути розширення розмітки. Universal Windows Platform підтримує такі розширення, як **Binding** і **x:Bind**. Обидва вони мають схожий синтаксис і майже однаковий набір функцій. Але ми повинні враховувати, що перше розширення здійснює зв'язування під час виконання за допомогою відображення для того, щоб знайти всі необхідні поля і властивості, а друге розширення працює для строгих типів як статичне зв'язування, яке дає змогу застосовувати багато правил під час компіляції. Очевидно, що **x:Bind** працює краще, ніж **Binding**, тому, якщо у вас є вибір, краще користуватися саме ним. У будь-якому разі ми будемо розглядати обидва розширення.

## Зв'язування двох елементів

Спочатку давайте розглянемо основи зв'язування, що дасть змогу змінювати властивості одного елемента керування на основі властивостей іншого. Для виконання цього завдання ми можемо використовувати обидва розширення розмітки. Давайте подивимося, як це зробити, на наступному прикладі:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VerticalAlignment="Center">
        <StackPanel x:Name="LayoutRoot">
            <Image Source="Assets/drone.jpg" Width="400">
                <Image.Projection>
```

```

        <PlaneProjection RotationY=
            "{Binding Value, ElementName=slider}">
        </PlaneProjection>
    </Image.Projection>
</Image>
<Slider Minimum="0" Maximum="360" Name="slider"
        Width="400" Margin="10"></Slider>
</StackPanel>
</Grid>

```

Запустивши цей код, ви можете побачити зображення і повзунок, за допомогою якого можна повернати зображення навколо осі Y. Зверніть особливу увагу, що ми не створювали ніякого C#-коду. Уся динаміка стала доступна тільки завдяки розширенню розмітки **Binding**.

Ви можете побачити, що **Binding** має два параметри:

- **Path** – дає можливість визначити властивість джерела, яке ми використовуємо для зв'язування. Оскільки шлях є властивістю за замовчуванням, можна використовувати неявний спосіб його оголошення.
- **ElementName** – дає можливість визначити ім'я елемента, який є джерелом зв'язування.

Давайте дещо змінимо наш код:

```

<Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}"
    VerticalAlignment="Center">
    <StackPanel x:Name="LayoutRoot">
        <Image Source="Assets/drone.jpg" Width="400">
            <Image.Projection>
                <PlaneProjection x:Name="plainPojection">
                </PlaneProjection>
            </Image.Projection>
        </Image>
        <Slider Minimum="0" Maximum="360" Name="slider"
                Width="400" Margin="10"
                Value="{Binding RotationY,
                    ElementName=plainPojection, Mode=TwoWay}">
        </Slider>
    </StackPanel>
</Grid>

```

Якщо до цього ми використовували **Binding**, щоб отримати дані від повзунка, тепер ми застосовуємо **Binding** для передавання даних до об'єкта плоскої проекції. Це можливо завдяки тому, що **Binding** може працювати у двох напрямках. За замовчуванням **Binding** використовує односторонній режим, але ми легко можемо це змінити за допомогою властивості **Mode**, якій можна присвоїти одне з таких значень:

- **OneTime** – всі властивості ініціалізуються відразу після того, як об'єкт **Binding** був створений. Цей режим корисний тільки тоді, коли ми хочемо подати поточний стан пам'яті.
- **OneWay** – у цьому режимі цільове значення оновлюватиметься відразу після оновлення джерела. Таким чином, ми матимемо постійно оновлений інтерфейс, навіть якщо деякі зміни до об'єкта застосовуються в коді.
- **TwoWay** – завдяки цьому режиму можна синхронізувати цільове значення із джерелом в обох напрямках.

У першому прикладі ми використовували зв'язування **OneWay**, яке чудово працює, оскільки цільовим значенням є проекція, що оновлюється разом із джерелом (повзунком). Але в другому прикладі цільовий елемент – це повзунок, і нам потрібно оновити джерело (проекцію), як тільки буде оновлено цільовий елемент. У цьому випадку **OneWay** працювати не буде. Зверніть особливу увагу, що коли потрібно змінити властивість **Value** або **RotationY** з коду, слід робити як у другому прикладі. Перший приклад не застосовний у разі зміни **RotationY**, тому що на **Value** через зв'язування **OneWay** вплинути неможливо.

Давайте розглянемо такий приклад:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VerticalAlignment="Center">
        <StackPanel x:Name="LayoutRoot">
            <Image Source="Assets/drone.jpg" Width="400">
                <Image.Projection>
                    <PlaneProjection x:Name="plainPojection">
                    </PlaneProjection>
                </Image.Projection>
            </Image>
            <Slider Minimum="0" Maximum="360" Name="slider"
                Width="400" Margin="10"
                Value="{Binding RotationY,
                    ElementName=plainPojection, Mode=TwoWay}">
            </Slider>
            <TextBox Width="200">
```

```

        Text="{Binding RotationY,
        ElementName=plainPojection,Mode=TwoWay}">
    </TextBox>
</StackPanel>
</Grid>
```

У цьому випадку ми додали ще один елемент керування – **TextBox** і ви можете побачити, що текстове поле і повзунок синхронізовані, тому що вони використовують одне джерело – проекцію та двостороннє зв'язування.

Ви також можете використовувати **x:Bind**, але синтаксис дещо відрізнятиметься:

```

<Grid Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}"
VerticalAlignment="Center">
<StackPanel x:Name="LayoutRoot">
<Image Source="Assets/drone.jpg" Width="400">
<Image.Projection>
<PlaneProjection x:Name="plainPojection">
</PlaneProjection>
</Image.Projection>
</Image>
<Slider Minimum="0" Maximum="360" Name="slider"
Width="400" Margin="10"
Value="{x:Bind plainPojection.RotationY,
Mode=TwoWay}">
</Slider>
<TextBox Width="200"
Text="{x:Bind
plainPojection.RotationY,Mode=TwoWay}">
</TextBox>
</StackPanel>
</Grid>
```

Розширення розмітки **x:Bind** використовує той самий режим, але **OneTime** вибрано за замовчуванням. Крім того, **x:Bind** не містить властивості **ElementName**. Однак у ній немає потреби, тому що компілятор знає, де знайти об'єкт. Ви просто повинні використовувати ім'я об'єкта і його властивість.

## Зв'язування в коді

Оскільки **Binding** працює під час виконання, ви можете легко зв'язувати елементи керування з об'єктами в пам'яті. Або навіть різні користувачькі елементи керування. Щоб це зробити, можна використовувати параметр **Binding** і задати такі ж властивості, як у XAML за допомогою розширення розмітки **Binding**.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Binding binding = new Binding();
    binding.ElementName = "slider";
    binding.Path = new PropertyPath("Value");
    binding.Mode = BindingMode.TwoWay;
    BindingOperations.SetBinding(plainProjection,
        PlaneProjection.RotationYProperty, binding);

    base.OnNavigatedTo(e);
}
```

Ми використовували метод класу **BindingOperations**, що приймає цільовий об'єкт, властивість і об'єкт **Binding** як параметри. Зверніть увагу, що ви можете використовувати як цільовий об'єкт тільки **DependencyProperty**. Це важливо знати, якщо ви хочете створити свій власний елемент керування.

## Зв'язування елемента з об'єктом

Найчастіше доводиться розглядати питання, що стосується зв'язування елемента керування з об'єктом, який створюється в коді. Давайте створимо простий клас, що зберігатиме інформацію про співробітника:

```
public class Employee
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string EMail { get; set; }

    public int Age { get; set; }
}
```

Цей клас містить чотири властивості з модифікатором доступу **public**. Очевидно, що об'єкт **Binding** не може отримати доступ до приватних чи захищених властивостей.

Використовуючи розширення розмітки **Binding**, ми визначили такий XAML, що має подавати об'єкт класу **Employee**:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
      VerticalAlignment="Center" HorizontalAlignment="Center">
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <TextBlock Text="First Name:" Grid.Row="0"
               Grid.Column="0" Margin="5" />
    <TextBox Text="{Binding FirstName, Mode=TwoWay}"
             Grid.Row="0" Grid.Column="1" />

    <TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0"
               Margin="5"/>
    <TextBox Text="{Binding LastName, Mode=TwoWay}"
             Grid.Row="1" Grid.Column="1" />

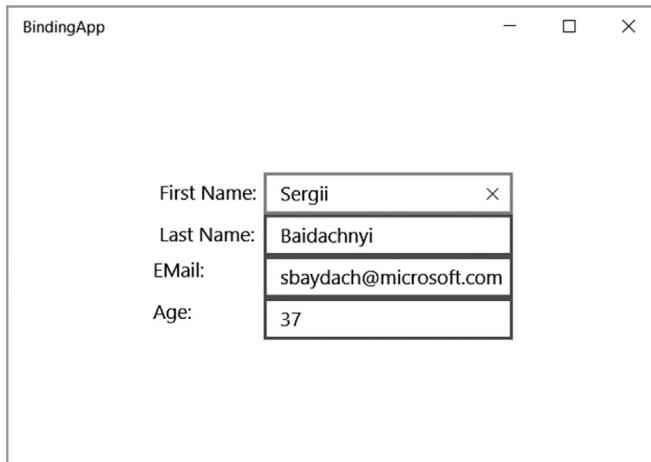
    <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
    <TextBox Text="{Binding EMail, Mode=TwoWay}" Grid.Row="2"
             Grid.Column="1" />

    <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
    <TextBox Text="{Binding Age, Mode=TwoWay}" Grid.Row="3"
             Grid.Column="1" />
</Grid>
```

Наочанок ми повинні створити об'єкт класу **Employee** і активувати його для контексту нашої форми. Давайте розглянемо такий код:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    Employee emp = new Employee()
    {
        FirstName = "Sergii",
        LastName = "Baidachnyi",
        Age = 37,
        EMail = "sbaydach@microsoft.com"
    };
    this.DataContext = emp;
    base.OnNavigatedTo(e);
}
```

Якщо ви запустите цей код, то побачите таку форму:



Як ви бачите, всі дані доступні для редагування завдяки цьому рядку:

```
this.DataContext = emp;
```

**DataContext** – це властивість, що визначена в класі **FrameworkElement** і може зберігати посилання на будь-який об'єкт. Сама властивість не являє собою нічого особливого, але важливо, що об'єкт **Binding** буде намагатися перевірити її для того, щоб знайти об'єкт з полем, що було задано як параметр для **Binding**. Для цього **Binding** перевірить властивість **DataContext** в цільовому елементі і, якщо вона має значення **null**, **Binding** перевірить **DataContext** контейнерів. У нашому коді ми ініціалізували властивість сторінки **DataContext**, що привело

до автоматичної ініціалізації **DataContext** усіх внутрішніх елементів. Звичайно, якщо ви хочете змінити **DataContext** внутрішніх елементів, можна просто присвоїти йому значення, використовуючи інші об'єкти.

Цей код буде працювати так само:

```
<Page
    x:Class="BindingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BindingApp"
    xmlns:d="http://schemas.microsoft.com/expression/
        blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
        markup-compatibility/2006"
    xmlns:code="using:BindingApp.Code"
    mc:Ignorable="d">
    <Page.Resources>
        <code:Employee x:Key="emp" FirstName="Sergii"
            LastName="Baidachnyi" Age="37"
            EMail="sbaydach@microsoft.com"></code:Employee>
    </Page.Resources>
    <Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}"
        DataContext="{StaticResource emp}"
        VerticalAlignment="Center"
        HorizontalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock Text="First Name:" Grid.Row="0"
            Grid.Column="0" Margin="5" />
        <TextBox Text="{Binding FirstName, Mode=TwoWay}"
            Grid.Row="0" Grid.Column="1" />
    
```

```
<TextBlock Text="Last Name:" Grid.Row="1"
    Grid.Column="0" Margin="5"/>
<TextBox Text="{Binding LastName, Mode=TwoWay}"
    Grid.Row="1" Grid.Column="1" />

<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
<TextBox Text="{Binding EMail, Mode=TwoWay}"
    Grid.Row="2" Grid.Column="1" />

<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
<TextBox Text="{Binding Age, Mode=TwoWay}"
    Grid.Row="3" Grid.Column="1" />
</Grid>
</Page>
```

Ви можете бачити, що ми оголосили об'єкт Employee прямо в XAML і просто присвоїли його властивості **DataContext** контейнера **Grid**. У цьому випадку ми використовували ресурси, але XAML дає змогу використовувати елемент **DataContext** безпосередньо, щоб оголосити об'єкт всередині:

```
<Page.DataContext>
    <code:Employee x:Name="emp" FirstName="Sergii"
        LastName="Baidachnyi" Age="37"
        EMail="sbaydach@microsoft.com"></code:Employee>
</Page.DataContext>
```

У цьому випадку ми створили об'єкт і присвоюємо його безпосередньо властивості **DataContext** сторінки. Зазвичай оголошувати такі об'єкти, як об'єкти класу Employee, в XAML не має ніякого сенсу, але цей підхід дуже популярний, якщо ви застосовуєте модель MVVM. У разі використання MVVM ви можете просто створити модель подання і присвоїти її контексту сторінки без C#-коду.

Тепер давайте подивимося на такий код, що буде показувати ту саму форму:

```
<Page
    x:Class="BindingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BindingApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
```

```
markup-compatibility/2006"
xmlns:code="using:BindingApp.Code"
mc:Ignorable="d">
<Page.Resources>
    <code:Employee x:Key="emp" FirstName="Sergii"
        LastName="Baidachnyi" Age="37"
        EMail="sbaydach@microsoft.com"></code:Employee>
</Page.Resources>
<Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">
    <VerticalAlignment="Center"
    HorizontalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
            <ColumnDefinition Width="Auto"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <TextBlock Text="First Name:" Grid.Row="0"
            Grid.Column="0" Margin="5" />
        <TextBox Text="{Binding FirstName, Mode=TwoWay,
            Source={StaticResource emp}}"
            Grid.Row="0" Grid.Column="1" />

        <TextBlock Text="Last Name:" Grid.Row="1"
            Grid.Column="0" Margin="5"/>
        <TextBox Text="{Binding LastName, Mode=TwoWay,
            Source={StaticResource emp}}"
            Grid.Row="1" Grid.Column="1" />

        <TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
        <TextBox Text="{Binding EMail, Mode=TwoWay,
            Source={StaticResource emp}}"
            Grid.Row="2" Grid.Column="1" />

        <TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
        <TextBox Text="{Binding Age, Mode=TwoWay,
            Source={StaticResource emp}}"
            Grid.Row="3" Grid.Column="1" />
    
```

```
        Grid.Row="3" Grid.Column="1" />
    </Grid>
</Page>
```

Як ви бачите, в цьому випадку ми використали параметр **Source** для об'єкта **Binding**, що дає змогу безпосередньо присвоїти контекст. Зазвичай ви не будете робити цього, хоча це можливо.

Давайте приділимо трохи часу, щоб розглянути розширення розмітки **x:Bind**. Воно має той самий синтаксис для зв'язування, але не підтримує властивість **DataContext**. Це можливо через те, що **DataContext** можна присвоїти будь-якому посиланню під час виконання, а для **x:Bind** потрібно точно зазначити тип під час компіляції. Властивість **Source** не підтримується з тієї ж причини. Щоб переконатися, що це не працює, спробуйте скомпілювати такий код:

```
<Page
    x:Class="BindingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BindingApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
    markup-compatibility/2006"
    xmlns:code="using:BindingApp.Code"
    mc:Ignorable="d">
    <Page.Resources>
        <code:Employee x:Key="emp" x:Name="emp"
            FirstName="Sergii" LastName="Baidachnyi"
            Age="37" EMail="sbaydach@microsoft.com">
        </code:Employee>
    </Page.Resources>
    <Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}"
        DataContext="{StaticResource emp}"
        VerticalAlignment="Center"
        HorizontalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
```

```

</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<TextBlock Text="First Name:" Grid.Row="0"
    Grid.Column="0" Margin="5" />
<TextBox Text="{x:Bind FirstName, Mode=TwoWay}"
    Grid.Row="0" Grid.Column="1" />

<TextBlock Text="Last Name:" Grid.Row="1"
    Grid.Column="0" Margin="5"/>
<TextBox Text="{x:Bind LastName, Mode=TwoWay}"
    Grid.Row="1" Grid.Column="1" />

<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
<TextBox Text="{x:Bind EMail, Mode=TwoWay}"
    Grid.Row="2" Grid.Column="1" />

<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
<TextBox Text="{x:Bind Age, Mode=TwoWay}"
    Grid.Row="3" Grid.Column="1" />
</Grid>
</Page>

```

Ви навіть не зможете запустити його, оскільки компілятор буде генерувати таку помилку: **Invalid binding path 'FirstName' : Property 'FirstName' can't be found on type ' MainPage'**. Розглядаючи це повідомлення, можна припустити, що **x:Bind** шукає всі властивості сторінки замість **DataContext**. Тому, якщо ви хочете використовувати **x:Bind**, то маєте оголошувати властивості всередині сторінки або задавати явний шлях. Давайте змінимо наш код так, щоб він використовував явний шлях:

```

<TextBlock Text="First Name:" Grid.Row="0" Grid.Column="0"
    Margin="5" />
<TextBox Text="{x:Bind emp.FirstName, Mode=TwoWay}"
    Grid.Row="0" Grid.Column="1" />

<TextBlock Text="Last Name:" Grid.Row="1" Grid.Column="0"
    Margin="5"/>
<TextBox Text="{x:Bind emp.LastName, Mode=TwoWay}"
    Grid.Row="1" Grid.Column="1" />

```

```
<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
<TextBox Text="{x:Bind emp.EMail, Mode=TwoWay}"
         Grid.Row="2" Grid.Column="1" />

<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
<TextBox Text="{x:Bind emp.Age, Mode=TwoWay}"
         Grid.Row="3" Grid.Column="1" />
```

Зараз все працює чудово. Доповнимо наш приклад і додамо ще один користувацький елемент керування – кнопку:

```
<Page
    x:Class="BindingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BindingApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
        markup-compatibility/2006"
    xmlns:code="using:BindingApp.Code"
    mc:Ignorable="d">
    <Page.Resources>
        <code:Employee x:Key="emp" x:Name="emp"
            FirstName="Sergii" LastName="Baidachnyi"
            Age="37" EMail="sbaydach@microsoft.com">
        </code:Employee>
    </Page.Resources>
    <Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}"
        DataContext="{StaticResource emp}"
        VerticalAlignment="Center"
        HorizontalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
```

```

<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
<TextBlock Text="First Name:" Grid.Row="0"
    Grid.Column="0" Margin="5" />
<TextBox Text="{Binding FirstName, Mode=TwoWay}"
    Grid.Row="0" Grid.Column="1" />

<TextBlock Text="Last Name:" Grid.Row="1"
    Grid.Column="0" Margin="5"/>
<TextBox Text="{Binding LastName, Mode=TwoWay}"
    Grid.Row="1" Grid.Column="1" />

<TextBlock Text="EMail:" Grid.Row="2" Grid.Column="0" />
<TextBox Text="{Binding EMail, Mode=TwoWay}"
    Grid.Row="2" Grid.Column="1" />

<TextBlock Text="Age:" Grid.Row="3" Grid.Column="0" />
<TextBox Text="{Binding Age, Mode=TwoWay}"
    Grid.Row="3" Grid.Column="1" />
<Button Content="Regenerate email" Grid.Row="4"
    Grid.ColumnSpan="2"
    HorizontalAlignment="Center" Margin="5"
    Click="Button_Click" />      </Grid>
</Page>
```

За допомогою кнопки можна змінити адресу електронної пошти, використовуючи обробник події **Click**:

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    emp.EMail = "Sergiy.Baydachnyy@microsoft.com";
}
```

Але якщо ви запустите цей код і натиснете кнопку, нічого не трапиться. Незважаючи на використання режиму зв'язування **TwoWay**, форма нічого не знає про зміни, які ми вносимо в код.

Щоб усунути цю проблему, ми можемо застосувати один з таких підходів:

- Успадкувати наш клас **Employee** від класу **DependencyObject** і зареєструвати всі властивості як властивості залежностей – цей підхід підійде

для користувачьких елементів керування, тому що всі з них уже успадковуються від **DependencyObject**. Однак цей підхід не є загальним. У будь-якому разі ми будемо розглядати це питання в розділі про користувачькі елементи керування.

- Ви можете реалізувати інтерфейс **INotifyPropertyChanged** для того, щоб повідомити об'єкт **Binding** про зміни.

Звичайно, в нашому випадку ми будемо використовувати другий підхід. Давайте його розглянемо:

```
public class Employee : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    private string firstName;
    private string lastName;
    private int age;
    private string email;
    public string FirstName
    {
        get { return firstName; }
        set
        {
            firstName = value;
            OnPropertyChanged(new PropertyChangedEventArgs
                ("FirstName"));
        }
    }
    public string LastName
    {
        get { return lastName; }
        set
        {
            lastName = value;
            OnPropertyChanged(new PropertyChangedEventArgs
                ("LastName"));
        }
    }
}
```

```
public string EMail
{
    get { return email; }
    set
    {
        email = value;
        OnPropertyChanged(new PropertyChangedEventArgs
            ("EMail"));
    }
}
public int Age
{
    get { return age; }
    set
    {
        age = value;
        OnPropertyChanged(new PropertyChangedEventArgs
            ("Age"));
    }
}
```

Як ви бачите, **INotifyPropertyChanged** просто оголошує подію **PropertyChanged** і об'єкти класу **Binding** можуть призначати власні обробники для цієї події. А як тільки подія відбулася, об'єкт **Binding** оновить дані на сторінці. Головне завдання полягає в тому, щоб знайти всі місця, де можливо змінити відстежувані властивості і активувати подію. Зазвичай ми можемо зробити це в сеттерах, як було показано у наведеному вище прикладі.

## Конвертери

Давайте розширимо наш клас **Employee** такою властивістю:

```
private double salary;
public double Salary
{
    get { return salary; }
    set
    {
        salary = value;
        OnPropertyChanged(new PropertyChangedEventArgs("Salary"));
    }
}
```

Якщо ми хочемо відображати цю властивість, то можемо додати ще один рядок до сітки і пов'язати властивість із текстовим полем. Оскільки об'єкт **Binding** перетворює **Salary** на рядок, ми бачимо подвійне подання. Але в якій валюті працівник отримує зарплату, нам не зрозуміло. Тому потрібно знайти спосіб перетворення наших даних на рядки, що міститимуть інформацію про валюту. Крім того, оскільки ми використовуємо **TextBox** і дозволяємо редагувати поля **Employee**, нам потрібен функціонал для того, щоб конвертувати рядок назад до дійсного типу. Для того, щоб зробити це, потрібно створити спеціальний клас-конвертер, який повинен задовольняти тільки одну вимогу – реалізовувати інтерфейс **IValueConverter**.

```
class MoneyConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, string language)
    {
        return ((double)value).ToString("C",
            new CultureInfo("uk-UA"));
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, string language)
    {
        double result;
        try
        {
            result = double.Parse((string)value,
                NumberStyles.AllowThousands |
                NumberStyles.AllowDecimalPoint |
                NumberStyles.AllowLeadingWhite |
                NumberStyles.AllowTrailingWhite);
        }
        catch { }
        return result;
    }
}
```

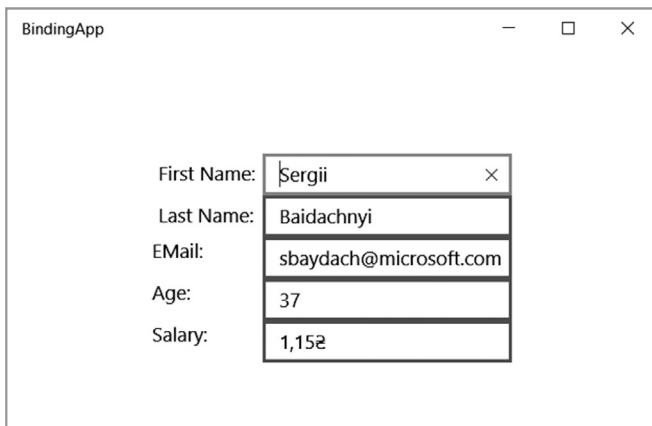
```
        NumberStyles.AllowCurrencySymbol);
    }
    catch
    {
        return DependencyProperty.UnsetValue;
    }
    return result;
}
```

Подивіться на код, щоб побачити реалізацію конвертера. **IValueConverter** вимагає реалізувати тільки два методи. Перший метод отримує початкове значення і перетворює його на значення типу, який приймає цільова властивість. У нашому випадку це рядок, але він може бути будь-якого типу, який має сенс у XAML, наприклад **LinearGradientBrush** або колекція об'єктів. Уявіть собі, що я отримую зарплату в українській валюті. Отже, я використав метод **ToString** для перетворення дійсного значення на валютний формат із застосуванням українських регіональних стандартів. Метод **ConvertBack** дає змогу отримати рядок і перетворити його на значення типу **double**. Якщо щось відбувається неправильно, просто повертаємо **DependencyProperty.UnsetValue**, що дасть змогу запобігти присвоєнню неправильного значення типу **double**.

Для того, щоб застосувати конвертер, потрібно створити відповідний об'єкт і передати його як параметр до об'єкта **Binding**:

```
<Page.Resources>
    <code:Employee x:Key="emp" x:Name="emp"
        FirstName="Sergii" LastName="Baidachnyi"
        Age="37" EMail="sbaydach@microsoft.com"
        Salary="1.15"></code:Employee>
    <code:MoneyConverter x:Key="money"></code:MoneyConverter>
</Page.Resources>
. . . .
<TextBox Text="{Binding Salary, Mode=TwoWay,
    Converter={StaticResource money}}"
    Grid.Row="4" Grid.Column="1" />
```

Якщо хочете, можете передати ще два параметри: **ConverterParameter** і **ConverterLanguage**. Ви можете бачити всі ці параметри, подивившись на прототип методу **Convert**. Але ми не будемо їх використовувати. Нижче ви можете побачити результат (використовується символ гривні):



Оскільки ми говоримо про конвертери, варто згадати ще дві властивості для **Binding** – **TargetNullValue** і **FallbackValue**. Перша дає змогу присвоїти визначене значення, якщо джерелом властивості є значення **Null**. А друга дає можливість присвоїти визначене значення, якщо щось не так зі звязуванням (наприклад, з конвертацією) і об'єкт **Binding** отримав замість значення виняток.

Давайте змінимо текстове поле, яке ми створили для зарплати:

```
<TextBox Text="{Binding Salary, Mode=TwoWay,
    Converter={StaticResource money},
    FallbackValue= {StaticResource exValue},
    TargetNullValue= {StaticResource nullValue} }"
    Grid.Row="4" Grid.Column="1" />
```

Тут **exValue** та **nullValue** – рядки ресурсів:

```
<x:String x:Key="exValue">-1</x:String>
<x:String x:Key="nullValue">0</x:String>
```

Для того, щоб побачити, як це працює, можна змінити метод **Convert** класу **MoneyConverter**, щоб згенерувати виняток:

```
public object Convert(object value, Type targetType,
    object parameter, string language)
{
    throw new Exception();
    return ((double)value).ToString("C", new CultureInfo("uk-UA"));
}
```

Щоб перевірити значення **null**, не можна використовувати конвертер. Тому видаліть конвертер, змініть значення властивості **Salary** на **null** і просто її не ініціалізуйте.

## Зв'язування з колекціями

Ми вже знаємо, як зв'язати властивості об'єкта з простими властивостями елементів керування, але й досі не зрозуміло, як працювати з колекціями, що містять багато елементів. Наприклад **ListView**, **GridView**, **Pivot**, **FlipView** тощо, які представляють відразу багато об'єктів. Насправді, коли ми говоримо про елементи керування для роботи з колекціями, ми маємо на увазі елементи керування, які успадковуються від класу **ItemsControl** і підтримують декілька властивостей, що важливі для зв'язування з колекціями:

- **ItemsSource** – дає змогу визначити посилання на колекцію об'єктів. Щоб присвоїти **ItemsSource**, ми можемо використовувати класичне зв'язування і розширення розмітки **x:Bind** або присвоїти цю властивість у коді.
- **ItemTemplate** – завдяки цій властивості можна використовувати власний шаблон для подання даних із поточного об'єкта в колекції.

Щоб перевірити ці властивості, давайте додамо ще один клас до нашої програми:

```
class EmployeeViewModel
{
    public List<Employee> Items { get; set; }

    public EmployeeViewModel()
    {
        Items = new List<Employee>();
        Items.Add(new Employee()
        {
            FirstName="Sergii",
            LastName="Baidachnyi",
            Age=37,
            EMail="sbaydach@microsoft.com"
        });
        Items.Add(new Employee()
        {
            FirstName = "Viktor",
            LastName = "Baidachnyi",
            Age = 37,
        });
    }
}
```

```
        Items.Add(new Employee()
{
    FirstName = "Tommy",
    LastName = "Lewis",
    Age = 25,
});
}
```

Завдяки цьому класу ми можемо підготувати всю необхідну інформацію для структури. У нашому випадку це просто колекція з трьома елементами всередині. Ми використовуємо узагальнений клас **List** для збереження даних про співробітників. Але потім ми покажемо, чому це не дуже хороший спосіб.

Давайте використаємо **ListView** для подання елементів нашої колекції:

```
<Page
    x:Class="BindingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:BindingApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
    markup-compatibility/2006"
    xmlns:code="using:BindingApp.Code"
    mc:Ignorable="d">
    <Page.Resources>
        <code:EmployeeViewModel x:Key="viewModel">
        </code:EmployeeViewModel>
    </Page.Resources>
    <Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}"
        DataContext="{StaticResource viewModel}"
        VerticalAlignment="Center"
        HorizontalAlignment="Center">
        <ListView ItemsSource="{Binding Items}"></ListView>
    </Grid>
</Page>
```

Ви бачите, що ми зв'язали **ItemsSource** із властивістю **Items** у поточному **DataContext**. Ми оголосили об'єкт **EmployeeViewModel** у файлах ресурсів для

того, щоб мати можливість використовувати **x:Bind** пізніше і не обмежуватись **Binding**.

Використавши цей код, ви побачите такі дані **ListView**:

```
BindingApp.Code.Employee
BindingApp.Code.Employee
BindingApp.Code.Employee
```

Це сталося тому, що ми не визначали ніяких шаблонів, і **ListView** використовує метод **ToString** для отримання даних з об'єкта. Але **ToString** повертає назву класу за замовчуванням. Тож можна почати з перевизначення методу **ToString** у класі **Employee**:

```
public override string ToString()
{
    return $"{FirstName} {LastName}";
}
```

Виконуючи код знову, ми зможемо побачити імена співробітників.

Звичайно, метод **ToString** не є дуже хорошим, оскільки він не забезпечує гнучкість у способах відображення даних. Але ми можемо використовувати **ItemTemplate** для оголошення власного подання:

```
<ListView ItemsSource="{Binding Items}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto">
                    </ColumnDefinition>
                    <ColumnDefinition Width="Auto">
                    </ColumnDefinition>
                </Grid.ColumnDefinitions>
                <TextBlock Text="{Binding FirstName}"
                           Margin="5"></TextBlock>
```

```
<TextBlock Text="{Binding LastName}"  
Margin="5" Grid.Column="1"></TextBlock>  
</Grid>  
</DataTemplate>  
</ListView.ItemTemplate>  
</ListView>
```

**ListView** виглядає так само, але в цьому випадку ми все контролюємо, і можемо легко змінювати вигляд елементів списку.

Також для роботи з колекціями можна використовувати **x:Bind**. Давайте розглянемо такий код:

```
<Page  
x:Class="BindingApp.MainPage"  
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/  
presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
xmlns:local="using:BindingApp"  
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  
xmlns:mc="http://schemas.openxmlformats.org/  
markup-compatibility/2006"  
xmlns:code="using:BindingApp.Code"  
mc:Ignorable="d">  
<Page.Resources>  
    <code:EmployeeViewModel x:Key="viewModel"  
        x:Name="viewModel"></code:EmployeeViewModel>  
</Page.Resources>  
<Grid Background="{ThemeResource  
ApplicationPageBackgroundThemeBrush}"  
    DataContext="{StaticResource viewModel}"  
    VerticalAlignment="Center"  
    HorizontalAlignment="Center">  
    <ListView ItemsSource="{x:Bind viewModel.Items}">  
        <ListView.ItemTemplate>  
            <DataTemplate x:DataType="code:Employee">  
                <Grid>  
                    <Grid.ColumnDefinitions>  
                        <ColumnDefinition Width="Auto">  
                        </ColumnDefinition>  
                        <ColumnDefinition Width="Auto">  
                        </ColumnDefinition>
```

```
</Grid.ColumnDefinitions>
<TextBlock Text="{x:Bind FirstName}"
    Margin="5"></TextBlock>
<TextBlock Text="{x:Bind LastName}"
    Margin="5" Grid.Column="1">
</TextBlock>
</Grid>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</Grid>
</Page>
```

Це майже те саме, але **x:Bind** повинен знати ім'я типу, який ми використовуємо для елементів колекції. Тож потрібно користуватися атрибутом **x:DataType** у **DataTemplate**, який це забезпечить.

Якщо ви використовуєте колекцію **List**, виникне та сама проблема, що і з прости об'єктами, коли ми намагались змінити деякі властивості в коді. Зв'язування не відслідковує зміни в колекції **List**. Тому при роботі з об'єктами ми реалізуємо інтерфейс **INotifyPropertyChanged** для забезпечення можливості відстежувати зміни властивостей. Під час роботи з колекціями необхідно використовувати інтерфейс **INotifyCollectionChanged**.

Але навіть описані вище методи не допоможуть нам вирішити всі проблеми, оскільки реалізувати власну колекцію досі не легко. Ось чому .NET Framework надає спеціальний узагальнений клас **ObservableCollection**, який ви можете використовувати також і для програм Universal Windows Platform. Просто змініть **List** на **ObservableCollection**, і об'єкт **Binding** буде відслідковувати всі зміни динамічно:

```
public ObservableCollection<Employee> Items { get; set; }
```



Розділ 9.

## **Навігація й керування вікнами**

## Сторінки та навігація

Як відомо, під час розробки сторінки використовується об'єкт класу **Page**. Але клас **Page** представляє контент, а не власне вікно. Якщо відкрити Object Browser, можна побачити, що клас **Page** розширює клас **UserControl**. Це користувальський елемент керування, який має стандартну властивість **Content**. Поміж іншого, клас **Page** містить кілька власних методів і властивостей. Деякі з цих властивостей (наприклад, **TopAppBar** і **BottomAppBar**) не надто цікаві – вони містять лише посилання на рядки програм, але інші властивості й методи допомагають зрозуміти, що відбувається з об'єктами класу **Page**.

Почнемо наше дослідження з властивості **Frame**, яка містить посилання на об'єкт класу **Frame**. Саме клас **Frame** відповідає за навігацію між сторінками в програмі. Таким чином, клас **Page** допомагає створювати контент, а клас **Frame** підтримує інфраструктуру для роботи з великою кількістю різних сторінок. Очевидно, що об'єкт класу **Frame** в певний момент часу може представляти тільки одну сторінку. Якщо потрібно активувати нову сторінку, можна просто викликати метод **Navigate** класу **Frame**. Це перевизначений метод, який дає змогу передавати дані щодо типу сторінки як обов'язковий параметр, а також об'єкт як параметр, який може бути використаний для нової сторінки. Третій параметр (необов'язковий) дає змогу визначити перехід між старими й новими сторінками за потреби анимувати процес. Отже, щоб перенести фрейм на нову сторінку, можна просто викликати метод **Navigate**:

```
rootFrame.Navigate(typeof(MainPage), e.Arguments);
```

Зauważте, що ми не створюємо об'єкт класу **Page** – ми лише передаємо тип нової сторінки.

Але об'єкт класу **Frame** – це не вікно. Вікно програми є об'єктом класу **Window**, який створюється об'єктом **Application**. Доступ до об'єкта класу **Window** можна отримати за допомогою статичної властивості **Current**:

```
var window=Window.Current
```

Так, на початку у нас є об'єкти класів **Application** і **Window**. Своєю чергою, об'єкт класу **Window** також містить властивість **Content**. І якщо ви бажаєте продемонструвати певні дані, потрібно ініціалізувати цю властивість. Насправді можна безпосередньо відобразити сторінку. Але клас **Page** не підтримує навігацію, і саме тому об'єкт класу **Frame** – найкращий варіант для подання вмісту вікна.

Розглянемо **App.xaml.cs** і знайдемо метод **OnLaunched**:

```

Frame rootFrame = Window.Current.Content as Frame;

if (rootFrame == null)
{
    rootFrame = new Frame();

    rootFrame.NavigationFailed += OnNavigationFailed;

    if (e.PreviousExecutionState ==
        ApplicationExecutionState.Terminated)
    {
    }
    Window.Current.Content = rootFrame;
}

if (rootFrame.Content == null)
{
    rootFrame.Navigate(typeof(MainPage), e.Arguments);
}
Window.Current.Activate();

```

Як видно, шаблон цього методу створює фрейм і сторінку за сценарієм, який описано вище. Метод **OnLaunched** викликається, коли користувач запускає програму в стандартний спосіб (плитка, значок на панелі завдань тощо). Наведений вище код перевіряє наявність фрейму та за його відсутності створює новий і присвоює його властивості **Content** вікна. Щойно інфраструктура буде готова, фрейм перейде на **MainPage**, створюючи й відображуючи об'єкт відповідного класу. Нарешті, для відображення поточного вікна використовується метод **Activate**.

Кожна сторінка містить посилання на свій фрейм. Скориставшись цією властивістю, можна організувати навігацію між сторінками відповідно до певної логіки.

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(DocumentPage));
}

```

Як зазначалося раніше, під час виклику методу **Navigate** об'єкт **Frame** створює екземпляр сторінки. Але за замовчуванням **Frame** не зберігає об'єкти попередніх сторінок. Отже, щойно ви перейдете до нової сторінки, попередню буде видалено. Цю поведінку можна змінити й увімкнути для конкретних сторінок режим кешування. Повернімося до класу **Page** і звернімо увагу на властивість

**NavigationCacheMode**. Щоб активувати режим кешування, необхідно призначити цю властивість в конструкторі сторінки або всередині файлу XAML:

```
<Page
    x:Class="WindowingApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WindowingApp"
    NavigationCacheMode="Enabled">
```

За замовчуванням режим кешування вимкнуто, але ви можете змінити настройки, встановивши значення **Enabled** чи **Required**. Якщо для режиму кешування встановлено значення **Enabled**, фрейм намагатиметься зберігати його у власному буфері якомога довше, але без жодного спеціального пріоритету. Щойно в кеш-пам'ять знадобиться помістити нову сторінку, фрейм видалятиме найстарішу. Якщо ж установлено значення **Required**, сторінка матиме пріоритет і не буде видалена, доки будуть наявні сторінки з нижчим пріоритетом (навіть ті, що з'явилися пізніше). Використовуючи **CacheSize** із **Frame**, ви можете визначити розмір буфера для сторінок.

Окрім методу **Navigate**, клас **Frame** підтримує метод, який дає змогу використовувати історію переходів, щоб забезпечити навігацію вперед і назад, не визначаючи тип сторінки. Можна використовувати методи **GoBack** і **GoForward**, але спершу варто перевірити, чи доступна навігація вперед або назад. Ви можете виконати перевірку за допомогою властивостей **CanGoBack** і **CanGoForward**.

```
private void MainPage_BackRequested(object sender,
    BackRequestedEventArgs e)
{
    if (this.Frame.CanGoBack) this.Frame.GoBack();
}
```

Можна знайти кілька подій, оголошених у класі **Frame**. За їх допомогою можна визначити, чи відбулася навігація, щоправда, вам не доведеться робити це надто часто. Набагато важливіше скласти уявлення про стан навігації всередині певної сторінки, адже фрейм не має жодних даних про сторінку. Сторінка має сама визначати, які дані завантажити, як ініціалізувати поля, які дані повинні бути збережені, щойно користувач перейде на наступну сторінку, тощо. Саме тому важливіше працювати з методами всередині класу **Page**, який вже має всю необхідну інфраструктуру. Не потрібно створювати нові обробники подій і думати про те, чому їх призначити – просто перевизначте наявні методи **OnNavigatedTo**, **OnNavigatedFrom** і **OnNavigatingFrom**. Останній можна використовувати для

припинення навігації, якщо користувачу потрібно зберегти певні дані або завершити поточну роботу. Метод **OnNavigatedFrom** дуже корисний, якщо певні дані повинні зберегтися, перш ніж сторінку буде видалено. За допомогою цього методу можна зберегти стан сторінки. Нарешті, **OnNavigatedTo** є найпопулярнішим методом, оскільки багато ініціалізацій відбувається саме в ньому. Крім того, за допомогою параметрів цього методу можна отримати інформацію щодо передавання параметра й режиму навігації:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    var par = e.Parameter;
    var state = e.NavigationMode;

    if (state==NavigationMode.New)
    {

    }

    base.OnNavigatedTo(e);
}
```

Інформація щодо режиму навігації допоможе дізнатися, що відбувалося зі сторінкою раніше і за потреби відновити її стан.

## Кнопка «Назад» є всюди

Гаразд. Ми знаємо, як забезпечити навігацію між сторінками. Перейти на наступну сторінку можна за допомогою контенту або кнопок, але як повернутися до попередньої? Як ми знаємо, усі телефони Windows Phone мають апаратну або програмну кнопку **Назад**. Під час розробки програм для Windows Phone 8.x ви можете використовувати готову кнопку, але для програм Windows 8.x доведеться створити власну кнопку **Назад** «з нуля». Звичайно, корпорація Microsoft опублікувала інструкцію з розробки та підтримки кнопки **Назад**, а також роботи з нею в програмах Windows 8.x. Але підхід дуже відрізняється від застосованого для Windows Phone.

Починаючи з Windows 10 розробники можуть застосовувати цей підхід усюди. Кнопку **Назад** інтегровано у Windows 10, і всі програми можуть активувати її та використовувати для власних потреб.

Щоб користуватися кнопкою **Назад** із будь-якої сторінки, необхідно реалізувати такий код:

```
protected async override void OnNavigatedTo  
(NavigationEventArgs e)  
{  
    SystemNavigationManager.GetForCurrentView().AppViewBack  
        ButtonVisibility = AppViewBackButtonVisibility.Visible;  
    SystemNavigationManager.GetForCurrentView().BackRequested  
        += MainPage_BackRequested;  
    base.OnNavigatedTo(e);  
}
```

Завдяки цим двом рядкам коду можна активувати кнопку **Назад**, якщо вона недоступна (наприклад, на настільному комп'ютері), і застосувати обробник події **BackRequested**. З обробником події **BackRequested** ви легко додасте апаратну чи програмну кнопку **Назад**. Найпростіша реалізація може мати такий вигляд:

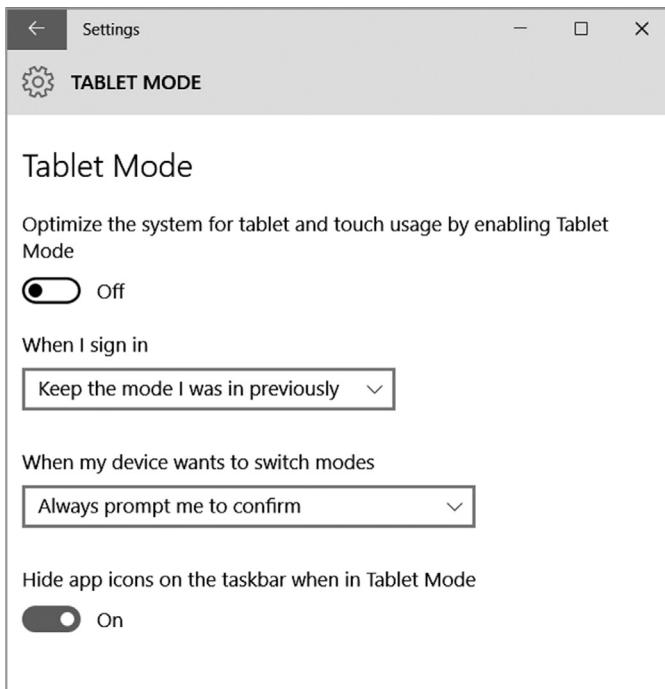
```
private void MainPage_BackRequested(object sender,  
    BackRequestedEventArgs e)  
{  
    if (this.Frame.CanGoBack) this.Frame.GoBack();  
}
```

Давайте подивимося, що відбудеться в режимі настільного комп'ютера, якщо запустити код, наведений вище.



Як бачимо, кнопка **Назад** відображається в рядку назви програми. Користувачі можуть натискати її, як на телефоні.

Щоб подивитися, як кнопка **Назад** працює в режимі планшета (якщо у вас немає планшета), відкрийте вікно налаштувань, переведіть Windows у режим планшета (або натисніть кнопку **Notification Hub** і знайдіть відповідний ярлик):



Якщо зменшити вікно налаштувань до мінімального розміру, видно, що кнопка **Назад** реалізується там сама.

Після переходу Windows у режим планшета кнопка **Назад** відображається на панелі завдань (за межами інтерфейсу), але працює так само, як і раніше.



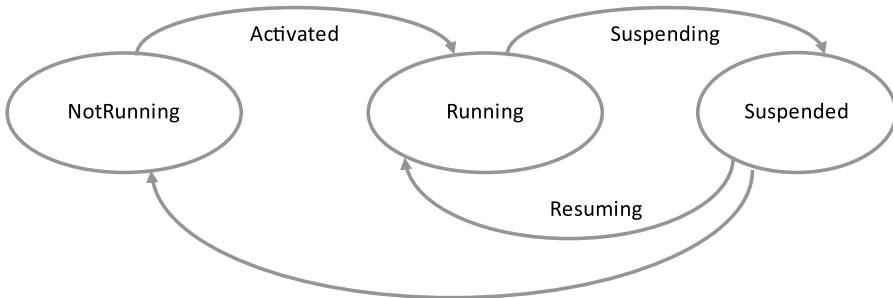
Отже, кнопка **Назад** є скрізь, і за допомогою Universal Windows Platform розробники можуть використовувати єдиний підхід для її реалізації.

## Життєвий цикл програм

У цьому розділі ми зупинимося на життєвому циклі програм (тих, що виконуються не у фоновому режимі). Ми вже знаємо, як реалізувати навігацію між

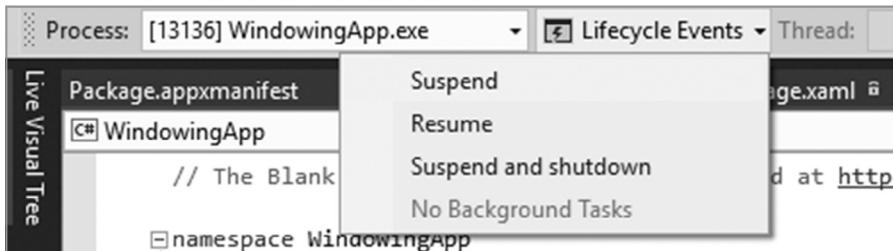
сторінками та функціональність кнопки повернення. Тепер час дізнатися, що відбувається із програмою під час її запуску або закриття користувачем.

Програми Universal Windows Platform підтримують три стани: **NotRunning**, **Running** і **Suspended**. Коли ваша програма не працює, вона перебуває в стані **NotRunning**. Звичайно, цей стан не є значущим і фізично не існує, коли програму не запущено. Екземпляра програми просто немає. ☺ Але коли користувач запускає програму, можна дізнатися, в якому вона була стані – **NotRunning** або **Suspended**. Після запуску програма набуде стану **Running**. У цьому стані програма активна, але щойно користувач згорне програму або переключиться на іншу, програма перейде в стан призупинення **Suspended**. Оскільки сучасні пристрої мають досить пам'яті для одночасної роботи з великою кількістю програм, система не закриває програму, а просто переводить в стан призупинення й підтримує в ньому, доки вистачає пам'яті. Це дає змогу активувати програму за мить, якщо користувач вирішить повернутися до неї. Таким чином, у стані **Suspended** програми ніби «заморожено».



Отже, є три стани програм. Але як саме вони використовуються? У UWP передбачено події, які дають змогу зрозуміти, що відбувається з програмою. Усього цих подій три: **Activated**, **Resuming** і **Suspending**. За яких умов виникає кожна з них, допоможе зрозуміти схема.

Звичайно, ви не можете відстежувати жодні події між станами **Suspended** і **NotRunning**, тому що ваша програма між цими двома станами не була активною. Але ви можете працювати з усіма іншими подіями. Почнімо із **Suspending** і **Resuming**, але перш ніж тестувати ці події, будь ласка, переконайтесь, що панель інструментів **Debug Location** доступна. За допомогою цієї панелі можна передавати з Visual Studio в програму різні події, зокрема **Suspending**.



В іншому разі ви не зможете згенерувати подію **Suspending**, тому що Visual Studio блокує її в режимі налагодження.

**Suspending** є дуже популярною подією – вона надзвичайно корисна, якщо потрібно зберегти дані. Обробник цій події можна присвоїти в конструкторі класу **Application**. Типова реалізація має такий вигляд:

```
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();

    Frame rootFrame = Window.Current.Content as Frame;
    var str=rootFrame.GetNavigationState();
    ApplicationData.Current.LocalSettings.
        Values["navigation"] = str;

    deferral.Complete();
}
```

Оскільки програма не має жодних відомостей про дані на поточній сторінці, щоб зберегти дані програми, пов'язані з історією навігації, краще використовувати обробник події **Suspending**. Щойно буде викликано метод **GetNavigationState**, щоб отримати рядок історії навігації, на поточній сторінці буде ініційовано **OnNavigatedFrom**. Подію **OnNavigatedFrom** можна використовувати для збереження даних, що пов'язані зі сторінкою.

Обробку події **Suspending** слід виконати якомога швидше. Зазвичай у вас менше п'яти секунд, щоб завершити всі необхідні операції. Але в більшості випадків операційна система може виділити достатньо ресурсів для обробника події **Suspending**. Збільшивши період обробки цієї події у Windows 10 можна за допомогою класу **ExtendedExecutionSession**.

```
private async void OnSuspending(object sender,
    SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();

    ExtendedExecutionSession ext = new
        ExtendedExecutionSession()
    {
        Reason = ExtendedExecutionReason.SavingData
    };
    ext.Revoked += Ext_Revoked;
    var result=await ext.RequestExtensionAsync();
    if (result==ExtendedExecutionResult.Allowed)
    {
        //doing something
    }

    deferral.Complete();
}
```

Якщо надається розширений сеанс, можна продовжувати виконання коду. Для нього немає жодних обмежень, але якщо система вирішить, що бракує ресурсів, буде надіслано подію **Revoke**. Після цього у вас залишиться менше однієї секунди для завершення роботи.

Кілька слів про **GetNavigationState**. Цей метод повинен повернати рядок з історією та параметрами навігації, які було передано на сторінки раніше. Ось чому дуже важливо передавати об'єкти в метод **Navigate**, що підтримує серіалізацію.

Для відновлення стану навігації потрібно викликати метод **SetNavigationState**. Щойно ви це зробите, на поточній сторінці буде викликано метод **OnNavigatedTo**.

Обробник події **Resuming** використовується дуже рідко. Зазвичай його застосовують для оновлення даних, якщо програма довго перебувала в стані **Suspended**.

Найцікавіша подія – це **Activated**. Але знайти таку подію з модифікатором **public** не вдасться. Замість події **Activated** клас **Application** підтримує кілька методів, які можна перевизначити. Якщо користувач запускає програму за допомогою плитки або ярлика, для ініціалізації програми можна використовувати метод **OnLaunched**. Стандартний шаблон уже містить реалізацію методу **OnLaunched**, але її можна розширити. Наприклад, використавши параметр **LaunchActivatedEventArgs**, щоб довідатися про попередній стан програми.

І якщо програма перебуvalа в стані **NotRunning** або **ClosedByUser**, можна продовжувати ініціалізувати її у звичний спосіб. Якщо ж програму призупинено системою, можна відновити останній стан, який було збережено за допомогою обробника події **Suspending**. Ось чому шаблон містить такий код:

```
if (e.PreviousExecutionState == ApplicationExecutionState.)
{
    //TODO: Load state from previously suspended application
}
```

Є безліч способів активувати програму Windows 10. Наприклад, ви можете пов'язати її з пошуковим запитом до Кортани або оголосити здатність працювати з файлами, які мають певні розширення, тощо. У такому разі метод **OnLaunched** не працюватиме. Для активації програм за допомогою контрактів потрібно перевизначити метод **OnActivated**.

```
protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind==ActivationKind.VoiceCommand)
    {
        //need to create a frame etc...
    }
    base.OnActivated(args);
}
```

Параметр **IActivatedEventArgs** допоможе визначити, який контракт використано для активації програми й вирішити, як її ініціалізувати.

Крім **OnActivated**, клас **Application** містить кілька методів для конкретних контрактів: **OnCachedFileUpdaterActivated**, **OnFileActivated**, **OnFileOpenPickerActivated**, **OnFileSavePickerActivated**, **OnSearchActivated**, **OnShareTargetActivated**. Під час реалізації контрактів, зазначених вище, не використовуйте метод **OnActivated**. Слід застосовувати тільки спеціальні методи.

## Керування вікнами

У Windows 10 вікна повернулися. У Windows 8 можна використовувати Windows Runtime тільки для розробки повноекранних програм. У такому разі не потрібно думати про заголовки вікна, про те, що станеться, якщо користувач змінить розмір вікна, тощо. Але у Windows 10 Universal Application Platform дає змогу виконувати сучасні програми в такий самий спосіб, як і класичні. У програмах передбачено плитки, піктограми, можливість задати мінімальний розмір і багато

інших функцій, підтримуваних віконними програмами. Давайте подивимося, якими функціями ми можемо керувати за допомогою коду і як це зробити.

## Клас ApplicationView

Перший клас, який ми розглянемо – це **ApplicationView**. Об'єкт цього класу пов'язаний із вікном і містить багато важливої інформації щодо нього, а також дає змогу змінювати певні параметри, наприклад задавати бажаний чи мінімальний розмір або працювати в повноекранному режимі. Щоб використовувати цей клас, не знадобиться створювати власне об'єкт, потрібно викликати статичний метод **GetForCurrentView** і використати посилання, яке повертає цей метод.

Розглянемо методи, які можна використовувати через посилання на об'єкт **ApplicationView**:

- **TryResizeView** – цей метод намагатиметься змінити розмір вікна під час виконання.
- **TryEnterFullScreenMode** – дуже корисний метод для медіапрограм. Ви можете застосовувати його для переведення програм в повноекранний режим.
- **ExitFullScreenMode** – дає змогу вийти з повноекранного режиму.
- **SetPreferredMinSize** – дає змогу встановити мінімальний рекомендований розмір вікна.

Ось як можна викликати один із таких методів:

```
ApplicationView.GetForCurrentView().SetPreferredMinSize  
(new Size(100, 100));
```

В іменах цих методів використано такі слова, як «спроба» та «рекомендований». Це пояснюється тим, що Universal Windows Platform працює на багатьох різних пристроях, і деякі з них можуть не підтримувати вікна. Наприклад, Windows Phone продовжує запускати всі програми в повноекранному режимі, або настільний комп'ютер переведено в режим планшета, і сучасна програма використовується в повноекранному режимі. Таким чином, якщо пристрой не підтримують вікно через режим планшета/телефона, ці методи просто не працюватимуть. Звичайно, деякі з них будуть повернати логічне значення, але **SetPreferredMinSize** не виконуватиме жодних дій.

## Рядок заголовка

Клас **ApplicationView** має також інші властивості, і дві з них можуть бути дуже корисними для змінення рядка заголовка вікна – це властивості **Title** і **TitleBar**. Завдяки першій можна використовувати будь-який текст в рядку заголовка, а друга властивість надає доступ до об'єкта типу **ApplicationViewTitleBar**, який містить інформацію про колір і шрифт власне назви, а також стандартних кнопок для закриття чи розгортання.

Давайте реалізуємо такий код:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    ApplicationView.GetForCurrentView().Title = "Custom text";
    ApplicationView.GetForCurrentView().TitleBar.
        BackgroundColor = Colors.Yellow;
    ApplicationView.GetForCurrentView().TitleBar.
        ForegroundColor = Colors.Green;
    ApplicationView.GetForCurrentView().TitleBar.
        ButtonBackgroundColor = Colors.Yellow;
    base.OnNavigatedTo(e);
}
```

Як бачимо, текст додається перед ім'ям програми, яке залишається там само.



Але всі кольори застосовано, і за допомогою цих властивостей можна узгодити стиль заголовка програми зі стилем контенту.

Звичайно, якщо ви бажаєте реалізувати щось особливe, **Title** і **TitleBar** не допоможуть, але Universal Windows Platform дає змогу охопити також область заголовка. Реалізуємо такий код:

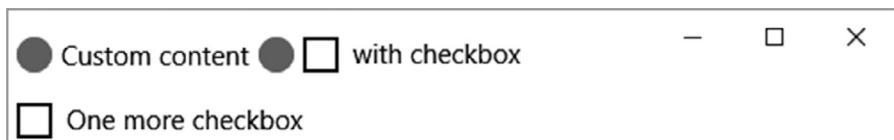
```
<Grid x:Name="myGrid" VerticalAlignment="Top" Margin="5">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" >
        <Ellipse Height="20" Width="20" Fill="Red"></Ellipse>
```

```
<TextBlock Text="Custom content"
    VerticalAlignment="Center"
    Margin="5" Name="txtBox"></TextBlock>
<Ellipse Height="20" Width="20" Fill="Red"></Ellipse>
<CheckBox Content="with checkbox" Name="chkBox"
    Margin="5"></CheckBox>
</StackPanel>
<Grid Grid.Row="1">
    <CheckBox Content="One more checkbox"></CheckBox>
</Grid>
</Grid>
```

Цей XAML-код містить кілька користувацьких елементів керування та графічних примітивів – загалом нічого особливого. Давайте подивимося, як додати цей XAML-код у заголовок вікна. Потрібно написати тільки один рядок коду:

```
CoreApplication.GetCurrentView().TitleBar.
ExtendViewIntoTitleBar = true;
```

Цей код використовує властивість **ExtendViewIntoTitleBar** для розширення нашого контенту на рядок заголовка. Запустивши цей код, ви побачите, що назва зникала, але в області заголовка з'явилися текстовий блок, поле для прапорця та два еліпси:



Але є проблема: якщо ви поставите прапорець в області рядка заголовка, він не працюватиме. Проте другий прапорець буде працювати нормально. Причина в тому, що назву розташовано поверх контенту. Це гарантує, що користувачі зможуть переміщувати вікно. Але ми можемо змінити цю поведінку, якщо пов'яжемо події рядка заголовка з подіями користувацьких елементів керування, які можуть бути розташовані за рештою елементів керування. Розглянемо такий код:

```
<Grid x:Name="myGrid" VerticalAlignment="Top">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
```

```

</Grid.RowDefinitions>
<Rectangle x:Name="titlebar"></Rectangle>
<StackPanel Orientation="Horizontal" >
    <Ellipse Height="20" Width="20" Fill="Red"></Ellipse>
    <TextBlock Text="Custom content"
        VerticalAlignment="Center"
        Margin="5" Name="txtBox"></TextBlock>
    <Ellipse Height="20" Width="20" Fill="Red"></Ellipse>
    <CheckBox Content="with checkbox" Name="chkBox"
        Margin="5"></CheckBox>
</StackPanel>
<Grid Grid.Row="1">
    <CheckBox Content="One more checkbox"></CheckBox>
</Grid>
</Grid>

```

Ми додали тільки один елемент керування – прямокутник, який заповнить перший рядок сітки, і розмістили стек-панель над прямокутником. Пов'язати прямокутник із заголовком можна таким чином:

```
Window.Current.SetTitleBar(titlebar);
```

Запустивши цей код, ви побачите, що тепер прапорець працює, і ви можете переміщати вікно, натиснувши рядок заголовка за межами стек-панелі. Оскільки в області рядка заголовка у нас є також текстовий елемент, ліпше замість попереднього використовувати такий код:

```
Window.Current.SetTitleBar(textBox);
```

Тепер можна перемістити вікно, натиснувши рядок заголовка в будь-якому місці за межами прапорця.

Нарешті, ви повинні пам'ятати дві речі: по-перше, ваші елементи керування ділять простір у рядку заголовка зі стандартними кнопками (закрити, розгорнути/згорнути), а по-друге, у повноекранному режимі заголовок недоступний. Тому важливо відстежувати реальний розмір заголовка та його видимість. Ви можете робити це за допомогою подій **IsVisibleChanged** і **LayoutMetricChanged**, які доступні через **CoreApplication.GetCurrentView().TitleBar**.

## Керування зміненням розміру вікна

У Universal Windows Platform передбачено тільки одну подію, що дає змогу відстежувати всі змінення вікна програми – **SizeChanged**.

Усе, що пов'язано зі зміненням стану вікна, генерує подію **SizeChanged**: змінення орієнтації, перехід у повноекранний режим тощо. Якщо ви признаєте обробник для цієї події, можна використовувати другий параметр – **SizeChangedEventArgs**. Він містить дві властивості: **NewSize** і **PreviousSize**. Ці властивості дають змогу отримати інформацію щодо ширини й висоти вікна до і після змінення.

Властивості **NewSize** і **PreviousSize** не містять жодної додаткової інформації, якщо програму запущено в повноекранному режимі, але за допомогою **ApplicationView** можна отримати доступ до цих даних.

Зазвичай **SizeChanged** використовується, щоб змінити стан інтерфейсу за допомогою **VisualStateManager**. Ми будемо вивчати цей підхід у наступному розділі.

## Робота з додатковими поданнями

Наприкінці розділу приділимо увагу створенню кількох вікон в одній програмі та керуванню ними. Це дуже корисна функція, якщо ви відкриваєте кілька документів одночасно. Наприклад, можна відкрити кілька зображень, щоб вибрати найкраще, або кілька текстових документів, щоб порівняти їх. Нарешті, вам просто може знадобитися закрити поточний документ, коли ви відкриваєте новий. Windows 10 дає змогу реалізувати всі ці сценарії. Для цього необхідно виконати такі дії:

- створити новий об'єкт **CoreApplicationView**, що представляє нове подання;
- використовуючи ланцюжок, пов'язаний із поданням, виконати його ініціалізацію та активувати підготовлене вікно;
- відобразити вже підготовлене вікно на екрані як нове самостійне вікно або переключити на нього наявне подання.

Перший крок найпростіший. Ви можете створити нове подання програми, використовуючи метод **CreateNewView** класу **CoreApplication**:

```
var view = CoreApplication.CreateNewView();
```

Формально нове подання готове, але воно не містить жодного контенту і ще не активоване. Зазвичай, щоб призначити новий контент поточному поданню,

використовується метод **Window.Current**, що повертає посилання на поточне вікно. Але за даних умов ми не можемо це зробити, бо отримаємо посилання на поточне вікно, а не на нове. Тому потрібно реалізувати метод, який викликатиметься в ланцюжку, пов'язаному з новим вікном. Зробити це можна за допомогою властивості **Dispatcher** і методу **RunAsync**:

```
var id=0;
await view.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
() =>
{
    var frame = new Frame();
    frame.Navigate(typeof(DocumentPage));
    Window.Current.Content = frame;
    Window.Current.Activate();
    id = ApplicationView.GetApplicationViewIdForWindow
    (view.CoreWindow);
});
```

Як бачимо, **Dispatcher** використовується, щоб отримати доступ до потрібного нам об'єкта **Window** у новому ланцюжку вікна. Ми застосували цей підхід для створення нового об'єкта **Frame**, присвоїли його властивості **Content** об'єкта **Window** і активували вікно. Зауважте, що використовувалася змінна для зберігання ідентифікатора подання. Нам знадобиться цей ідентифікатор, щоб відкрити вікно, але до нього неможливо отримати доступ з іншого ланцюжка.

Нарешті, ми можемо вивести на екран нове подання як окреме вікно:

```
var viewShown = await ApplicationViewSwitcher.
TryShowAsStandaloneAsync(id);
```

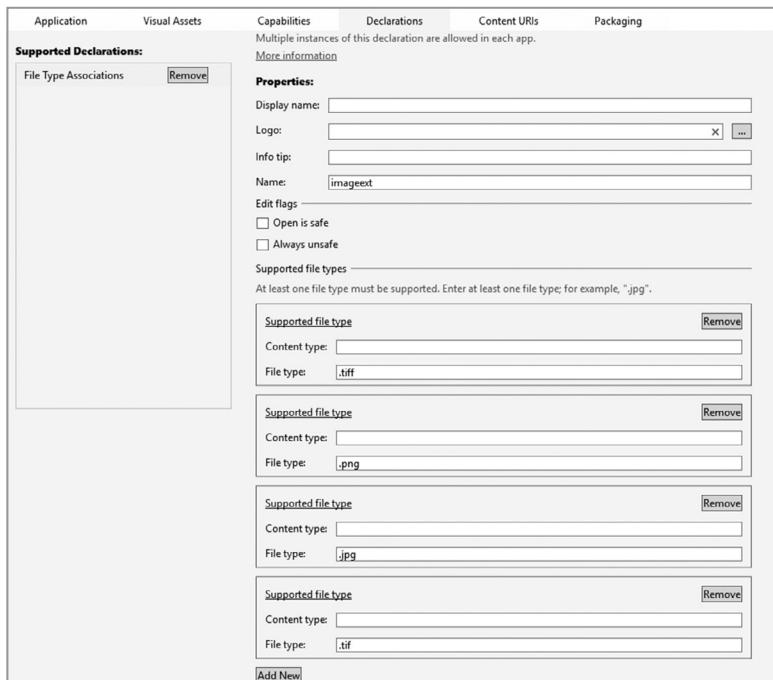
Якщо все гаразд, метод виведе істинне значення й нове вікно відобразиться поверх поточного. Реалізуйте простий інтерфейс із кнопкою і скопіюйте весь код в обробник події натискання кнопки.

Якщо потрібно переключитися з поточного подання на нове, можна просто виконати такий код:

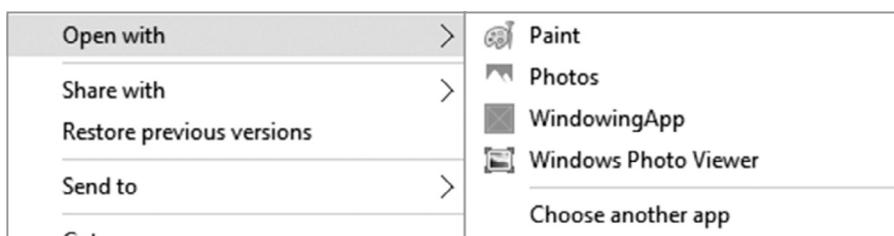
```
await ApplicationViewSwitcher.SwitchAsync(id);
```

Що ж, ми можемо створити нове вікно, але це не вирішить проблему, коли користувач активує програму, натиснувши файл, але при цьому в ній уже відкрито інший документ. У такому разі наявний код із незначними змінами можна використовувати в обробнику будь-якої «активованої» події.

Уявімо, що наша програма працює із зображеннями різних форматів, і ми можемо пов'язати її з найпоширенішими розширеннями зображень та інтегрувати з Провідником Windows. Для цього просто відкрийте файл маніфесту й додайте асоціації типів файлів (**File Type Associations**):



Ми додали сюди кілька поширених типів. Розгорнувши програму просто зараз, ви побачите, що Провідник Windows використовує нашу програму в пункті меню «Відкрити за допомогою» для файлів із заданими розширеннями.



Але для того, щоб працювати з файлами, необхідно реалізувати метод **OnFileActivated**:

```

protected async override void OnFileActivated
(FileActivatedEventArgs args)
{
    if (Window.Current.Content != null)
    {
        var view = CoreApplication.CreateNewView();
        var id = 0;
        await view.Dispatcher.RunAsync
        (CoreDispatcherPriority.Normal, () =>
        {
            var frame = new Frame();
            frame.Navigate(typeof(DocumentPage));
            Window.Current.Content = frame;
            Window.Current.Activate();
            id = ApplicationView.GetApplicationViewIdFor
            Window(view.CoreWindow);
        });
        await ApplicationViewSwitcher.
        TryShowAsStandaloneAsync(id);
    }
    else
    {
        Frame rootFrame = new Frame();
        rootFrame.NavigationFailed += OnNavigationFailed;
        Window.Current.Content = rootFrame;
        rootFrame.Navigate(typeof(DocumentPage));
        Window.Current.Activate();
    }
    base.OnFileActivated(args);
}

```

Метод складається з двох частин. У першій ми створюємо нове вікно й показуємо в ньому наш документ. Це можливо, якщо ви вже відкрили програму й вона перебуває в активному стані. Якщо програма не працює, для відкриття файлу слід використовувати поточне вікно. Звичайно, ми не використовували в цій програмі власне файл(и), але ви можете отримати доступ до всіх файлів, використовуючи параметр **args** і властивість **Files**.

Якщо під час виконання цього коду програму вже відкрито, її буде розташовано поверх усіх вікон, а після цього відобразиться нове вікно. Це досить дивна поведінка, і ліпше її виправити. Щоб залишити початкове вікно у вихідному стані, необхідно реалізувати дві речі:

- Додати такий рядок коду на початку методу **OnLaunched**:

```
ApplicationViewSwitcher.DisableSystemViewActivationPolicy();
```

Цей рядок відключить поточну поведінку.

- Замінити **TryShowAsStandaloneAsync** на такий рядок:

```
await args.ViewSwitcher.ShowAsStandaloneAsync(id);
```

Ви можете знайти властивість **ViewSwitcher** у всіх подіях активації. Її спеціально додано, щоб розширити функціональність Windows 10.

Запустіть програму, згорніть головне вікно й відкрийте нове, вибравши та відкривши файл. Як бачимо, головне вікно досі згорнуто, а поява нового жодним чином не вплинула на його стан.

Розділ 10.

## **Як створити адаптивні інтерфейси**

## Що ми вже знаємо про адаптивні інтерфейси?

Нам уже відомо, що Universal Windows Platform дає змогу створювати програми для всіх пристройів Windows 10. Крім того, якщо ви працюєте з програмами Windows 10 на ПК, то маєте можливість змінювати розмір вікна будь-якої програми або принаймні зафіксувати програми по лівому або правому краю. Тому, щоб створювати програми для Windows 10, потрібний адаптивний інтерфейс, що пристосовується до різного розміру екрана, різної його орієнтації та працюватиме без збоїв і в тому разі, якщо користувач вирішить змінити розмір вікна програми. Очевидно, що адаптивний інтерфейс змінюється під час виконання, і ви маєте бути готові до цих змін, щоб використовувати нові макети, змінювати розташування користувацьких елементів керування і подавати дані у вигляді, що задовольнятиме вимоги користувачів.

У попередньому розділі ми вже обговорювали подію **SizeChanged**, яку генеруватимуть будь-які зміни вікна програми. Навіть якщо ви вирішите змінити розмір вікна за допомогою програми, подію буде згенеровано. Очевидно, що **SizeChanged** – найкращий спосіб відстеження будь-яких змін і відновлення вигляду програми їх основі.

Отже, із **SizeChanged** легко впоратися, але як змінити вигляд програм? У Windows 8 часто створювали кілька подібних панелей для різних екранів і змінювали властивості **Visibility** на основі поточного розміру екрана. Це давало змогу реалізувати **SizeChanged** дуже швидко, але такий підхід має багато недоліків, оскільки потрібно конструювати різні інтерфейси для різних екранів. Таким чином, замість однієї форми ми мали б створити принаймні 2–3 форми, що потребує додаткових ресурсів. Водночас цей підхід вимагає багато пам'яті й дуже завантажує процесор, оскільки потрібно створювати багато елементів керування та реалізовувати зв'язування з тим самим набором даних декілька разів. Зрештою, цей код не так просто підтримувати. Але була причина застосовувати цей підхід – якщо у вас є складний макет, що базується на одній чи декількох панелях **Grid** і **StackPanel**, то майже неможливо адаптувати його до нових розмірів екрана. Ось чому найперша і найважливіша рекомендація полягає у використанні нового користувацького елемента керування для створення адаптивних інтерфейсів. Ми вже обговорювали ці елементи керування в попередніх розділах: **RelativePanel**, **SplitView** і **CommandBar**.

Перший із цих елементів керування допомагає побудувати дуже складні макети, але він не використовує глибоку внутрішню ієархічну структуру. Можна розмістити всі елементи керування на тому самому рівні та просто визначити для них місця, використовуючи вкладені властивості, наприклад **RelativePanel.LeftOf** чи **RelativePanel.RightOf**. У такий спосіб легко змінити положення будь-якого

елемента керування відносно іншого. Другий елемент керування, **SplitView**, дає змогу будувати навігаційні панелі. Завдяки підтримці різних режимів та можливості змінювати режим під час виконання цей елемент можна використовувати для телефонів і комп'ютерів.

У Windows 10 **CommandBar** – той самий для різних пристрій. Він дає можливість розмістити будь-які команди і відтак можете використовувати навіть якщо програмі бракує місця для контенту.

Тому за допомогою оновленої Universal Windows Platform можна сконструювати той самий інтерфейс для всіх пристрій, що використовують Windows 10.

Насправді, якщо потрібно адаптувати інтерфейс програми, ви просто реалізуєте обробник події **SizeChanged** і змінюєте його, використовуючи нові можливості UWP. Але як щодо XAML? Він дає змогу реалізувати будь-що, тож, напевно, можна задати всі необхідні зміни безпосередньо в коді XAML?

Саме так. Можна уникнути об'ємного коду всередині обробника події **SizeChanged** та реалізувати фактично всі необхідні зміни за допомогою XAML – завдяки **VisualStateManager**.

## VisualStateManager

Якщо обговорювати користувацькі елементи керування, особливу увагу слід приділити **VisualStateManager**. Насправді знайти цей компонент можна, переглядаючи будь-який шаблон для стандартних користувацьких елементів керування, наприклад, шаблон кнопки:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="CommonStates">
        <VisualState x:Name="Normal">
            <Storyboard>
                . . .
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Pressed">
            <Storyboard>
                . . .
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Disabled">
            <Storyboard>
```

```
    . . .
    </Storyboard>
  </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

Ідея дуже проста: будь-який користувацький елемент керування може використовувати компонент **VisualStateManager** для оголошення стану чи групи станів. Кожен стан має набір анімацій і сеттерів, які допомагають елементам керування переходити з одного стану до іншого. Зверніть особливу увагу, що можна зробити це відразу або ж використати «довготривалу» анімацію. Кнопка має три стани; якщо її відключено або натиснуто, запускається анімація. Всі стани кнопок належать до однієї групи. Роль груп полягає в тому, що елементи керування не можуть перебувати в різних станах однієї групи, хоча можна легко визначити кілька груп.

Звичайно, не досить оголосити різні стани. Повинен бути код, за допомогою якого елементи керування переходять з одного стану до іншого. Де розташовувати цей код, залежить від елементів керування і від того, які сценарії реалізуються. Наприклад, для подій, пов'язаних із вказівником, можуть існувати різні обробники.

Отже, елементи керування можуть мати деякі стани, але що це нам дає? Те, що можна присвоїти власний **VisualStateManager** майже будь-якому елементу керування, наприклад, головному елементу керування **Grid** в інтерфейсі програми, і визначити за його допомогою всі необхідні стани та анімації. Ці анімації допомагають приховувати або відображати деякі елементи керування, змінювати властивості **ItemTemplate**, інші властивості елементів керування тощо. Для однієї групи можна визначити кілька станів. Група становитиме набір станів для різного розміру вікон. Кожен стан міститиме анімації і сеттери для певної роздільної здатності. Наприклад, для основної сітки можна використовувати такий шаблон:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="DefaultLayout">
      . . .
    </VisualState>
    <VisualState x:Name="Layout500">
      . . .
    </VisualState>
    <VisualState x:Name="Layout1024">
      . . .
    </VisualState>
```

```

    </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>

```

Усередині кожного стану є змога розмістити елемент керування **Storyboard** і визначити всі потрібні анімації. У наведеному нижче коді ми використали анімацію **ObjectAnimationUsingKeyFrame** для того, щоб змінити властивість **Visibility**, присвоїти нове значення **ItemTemplate** тощо.

```

<ObjectAnimationUsingKeyFrames Storyboard.
TargetName="itemListView"
    Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
</ObjectAnimationUsingKeyFrames>
<ObjectAnimationUsingKeyFrames Storyboard.
TargetName="itemListView"
    Storyboard.TargetProperty="ItemTemplate">
    <DiscreteObjectKeyFrame KeyTime="0"
        Value="{StaticResource smallItemTemplate}"/>
</ObjectAnimationUsingKeyFrames>

```

Після визначення всіх необхідних візуальних станів можна реалізувати код, який дає змогу змінювати візуальний стан залежно від розміру вікна. Це можна зробити в обробнику події **SizeChanged** одним рядком коду:

```
VisualStateManager.GoToState(myGrid, "Layout500", true);
```

## Сеттери

Загалом підхід, який дає змогу змінювати властивості елементів керування за допомогою анімацій, що працюють усередині візуальних станів, не є новим. Його можна застосувати для Windows 8.x, Windows Phone і навіть Silverlight. Однак у багатьох випадках анімації не потрібні. Наприклад, якщо потрібно змінити макет через те, що користувач змінив орієнтацію екрана, властивості елементів керування слід змінювати дуже швидко. Так, розробники зазвичай використовують **ObjectAnimationUsingKeyFrame** для того, щоб усі необхідні зміни відбулися миттєво:

```

<Storyboard>
    <ObjectAnimationUsingKeyFrames
        Storyboard.TargetName="itemListView"

```

```
        Storyboard.TargetProperty="Visibility">
    <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
</ObjectAnimationUsingKeyFrames>
</Storyboard>
```

Видно, що цей підхід не є ідеальним, оскільки для виконання простої дії потребує використання кількох складних об'єктів із багатьма параметрами. Ось чому новий елемент XAML, що називається **Setter**, бувавельмии корисним. Наприклад, він дає змогу переписати зазначений вище код так:

```
<VisualState.Setters>
    <Setter Target="comboBox.Visibility" Value="Collapsed">
    </Setter>
</VisualState.Setters>
```

Так набагато зрозуміліше. Розробники мають визначити ім'я властивості й нове значення в обраному стані.

Якщо потрібно, можна поєднати різні сеттери та **Storyboard** всередині того самого стану.

## Адаптивні тригери

Як уже зазначалося, недостатньо оголосити всі можливі стани – розробники мають реалізувати код, який дає змогу змінювати стани в динаміці. Наприклад, якщо ви плануєте змінювати стан залежно від розміру екрана, необхідно реалізувати обробник подій для події **SizeChanged** і використати метод **GoToState** класу **VisualStateManager**. Іноді незрозуміло, коли слід застосовувати стан. Крім того, якщо у вас є кілька груп станів і треба об'єднати декілька станів, можна легко припуститися помилки. Ось чому корпорація Microsoft реалізувала інфраструктуру для тригерів станів. Щоб зрозуміти, який стан слід застосовувати, можна оголосити один тригер або набір тригерів у XAML. У такий спосіб можна задати всі необхідні правила без кодування.

У поточній версії Universal Windows Platform Microsoft представила лише один тригер – **AdaptiveTrigger**. Але завжди можна розробити свої тригери.

Як використовувати **AdaptiveTrigger**, видно з цього коду:

```
<VisualState x:Name="Normal">
    <VisualState.Setters>
```

```

<Setter Target="comboBox.Visibility"
        Value="Visible"></Setter>
</VisualState.Setters>
<VisualState.StateTriggers>
    <AdaptiveTrigger MinWindowWidth="700">
    </AdaptiveTrigger>
</VisualState.StateTriggers>
</VisualState>
<VisualState x:Name="Mobile">
    <VisualState.Setters>
        <Setter Target="comboBox.Visibility"
                Value="Collapsed"></Setter>
    </VisualState.Setters>
    <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0">
        </AdaptiveTrigger>
    </VisualState.StateTriggers>
</VisualState>

```

Як видно, **AdaptiveTrigger** має тільки два параметри: **MinWindowWidth** і **MinWindowHeight**. Вони допомагають змінювати стан вікна залежно від розміру. У нашому прикладі, якщо ширина вікна менше за 700 пікселів, ми згортаемо елемент **ComboBox**.

Погляньмо, як створити свій тригер.

Фактично всі програми передбачають підключення до Інтернету. Але отримання даних із мережі забирає деякий час. **ProgressRing** показує, що отримання даних триває, а **VisualStateManager** допомагає переглянути або приховати елементи залежно від наявності цих даних. Звичай потрібно розробити три стани – **Loading**, **Loaded** і **Error**. Звичайно, потрібно реалізувати певний код, який змінить стан програми на основі стану «моделі подання». Розгляньмо, як можна реалізувати свій тригер, що допомагає уникнути кодування повністю.

Насамперед потрібно перевірити, чи клас моделі подання готовий до використання тригерів. Найкраще реалізувати властивість, що показує поточний стан нашої моделі, а також подію, яка відбувається щоразу, коли модель змінює свій стан. Звичайно, було б доцільніше реалізувати базовий клас для всіх моделей подання в нашій програмі. У наступній частині ми поговоримо про шаблони MVVM і детальніше розглянемо ці моменти. А зараз навчимося реалізовувати простий клас, який не міститиме жодних даних, а тільки базову інфраструктуру для наших тригерів:

```

public enum StateEnum
{
    Loading,
    Loaded,
    Error
}
public class StateChangeEventArgs:EventArgs
{
    public StateEnum State { get; set; }
}

public delegate void StateChangedDelegate(object model,
    StateChangeEventArgs args);

public class PageViewModel
{
    public event StateChangedDelegate StateChanged;

    public void InitModel()
    {
        if (StateChanged != null) StateChanged.Invoke(this,
            new StateChangeEventArgs()
            { State = StateEnum.Loading });
        //load data
        if (StateChanged != null)
            StateChanged.Invoke(this,
                new StateChangeEventArgs()
                { State = StateEnum.Loaded });
    }
}

```

Ми реалізували метод **InitModel** як приклад коду, де потрібно ініціювати подію **StateChanged**. Здебільшого ви реалізовуватимете цей метод в успадкованих класах у власний спосіб.

Після оновлення моделі подання можна створити об'єкт у файлі сторінки XAML:

```

<Page.Resources>
    <st:PageViewModel x:Name="model"></st:PageViewModel>
</Page.Resources>

```

Тепер можна створити свій тригер. Для цього слід створити новий клас, що успадковує клас **StateTriggerBase**. Усередині класу можна оголосити будь-які методи і властивості, однак необхідно визначити, де ви звертатиметеся до методу **SetActive**. За його допомогою можна активувати або деактивувати тригер. Наприклад, ми реалізували такий клас:

```
public class DataTrigger: StateTriggerBase
{
    private PageViewModel model;

    public PageViewModel Model
    {
        get
        {
            return model;
        }
        set
        {
            model = value;
            model.StateChanged += Model_StateChanged;
        }
    }

    public string StateOfModel { get; set; }

    private void Model_StateChanged(object model,
        StateChangeEventArgs args)
    {
        SetActive(args.State.ToString().Equals(StateOfModel));
    }
}
```

Як бачимо, є дві властивості в класі, які дають змогу задати посилання на поточну модель подання й визначити стан, який буде застосовано для активації тригера. Після того, як модель подання ініціалізовано, ми активуємо або деактивуємо тригер за допомогою обробника події **StateChanged**.

Нарешті визначмо такі стани:

```
<VisualState x:Name="Loading">
    <VisualState.Setters>
        <Setter Target="gridView.Visibility" Value="Collapsed">
        </Setter>
```

```
<Setter Target="progress.Visibility" Value="Visible">
</Setter>
</VisualState.Setters>
<VisualState.StateTriggers>
    <st:DataTrigger Model="{StaticResource model}"
        StateOfModel="Loading"></st:DataTrigger>
</VisualState.StateTriggers>
</VisualState>
<VisualState x:Name="Loaded">
    <VisualState.Setters>
        <Setter Target="gridView.Visibility" Value="Visible">
        </Setter>
        <Setter Target="progress.Visibility" Value="Collapsed">
        </Setter>
    </VisualState.Setters>
    <VisualState.StateTriggers>
        <st:DataTrigger Model="{StaticResource model}"
            StateOfModel="Loaded"></st:DataTrigger>
    </VisualState.StateTriggers>
</VisualState>
```

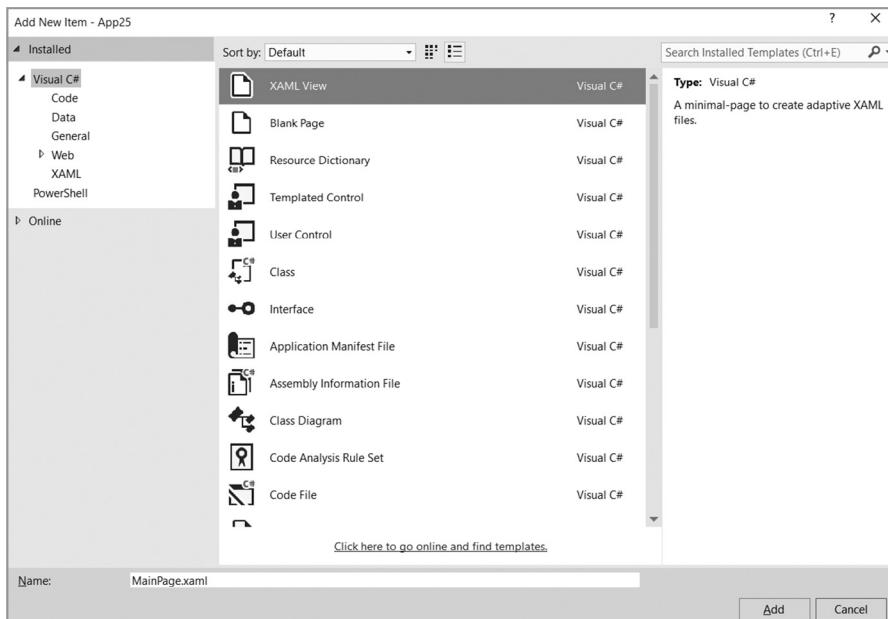
Це справді цікаво й дає змогу ефективніше реалізувати шаблон MVVM.

## Як створити різні подання

Ми вже обговорили, як створити адаптивний інтерфейс усередині подання. Проте створити універсальну сторінку іноді справді непросто. Можна, скажімо, адаптувати всі елементи керування, але слід пам'ятати, що користувачі телефонів зазвичай працюють із програмами, послуговуючись лише однією рукою. Таким чином, слід орієнтуватися не тільки на макет, а й на спосіб використання також.

Саме тому часом буває потрібно створити різні подання для різних пристроїв. Розгляньмо, як це зробити за допомогою Visual Studio 2015.

Звичайно, щоб почати працювати з додатковими поданнями, потрібно мати початкові сторінки. Наприклад, потрібно створити нове подання для наявної сторінки **MainPage.xaml**. Для цього необхідно додати до проекту нове подання XAML.



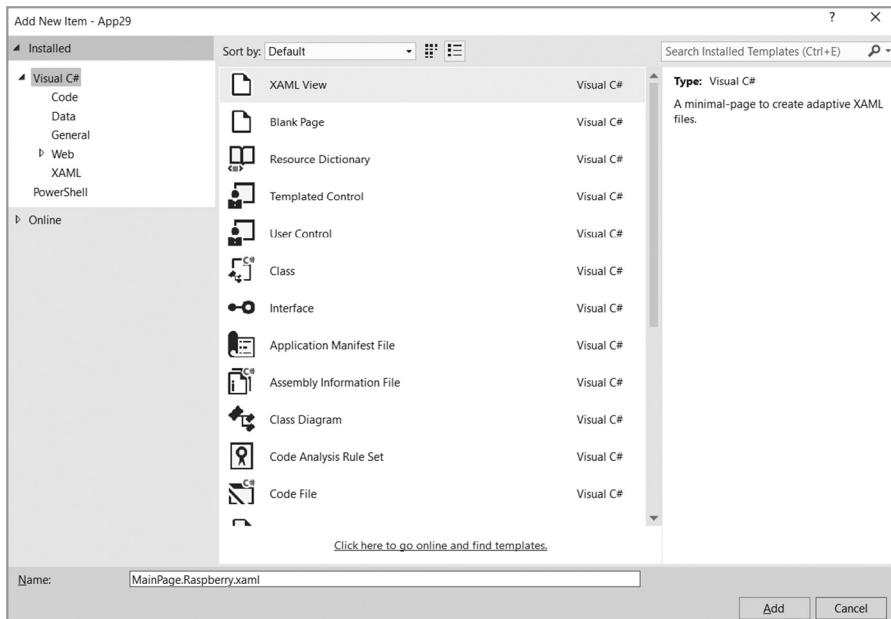
З цього прикладу зрозуміло, що шаблон подання **XAML View** створить тільки сторінку XAML, без файлу коду. Звичайно, цього недостатньо. Щоб зв'язати нове подання зі стандартною сторінкою, потрібно обрати один із двох підходів:

- Можна помістити подання в ту саму папку, що й початкову сторінку. Тоді слід дати поданню назву, використовуючи ім'я початкової сторінки та префікс. Два префікса можна назвати відразу – це **DeviceFamily-Mobile** та **DeviceFamily-Desktop**. Ми не мали змоги протестувати Xbox.
- Для подання можна вжити те саме ім'я, яке має сторінка за замовчуванням, але помістити подання слід до спеціальної папки – **DeviceFamily-Mobile** чи **DeviceFamily-Desktop**.

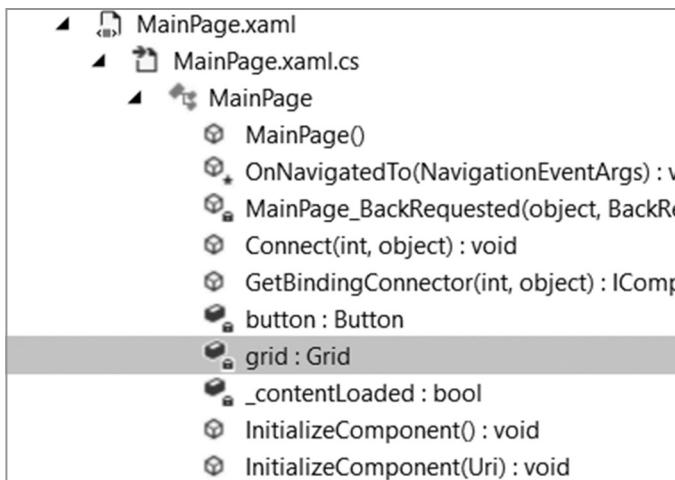


От і все. Запустивши цю програму на телефоні, ви матимете подання, що відповідає правилу **DeviceFamily-Mobile**. Це подання й буде застосовано. Проте якщо нове подання для **DeviceFamily-Mobile** не реалізувати, буде застосовано початкове подання XAML. Той самий принцип діє для ПК.

Для зв'язування нового подання з наявним файлом коду насамперед слід дотримуватися правила іменування: **<початкова назва сторінки>.будь-яке розширення.xaml**. Не використовуйте **DeviceFamily**- для власних розширень – це слово, ймовірно, зарезервовано Visual Studio, і ви отримаєте повідомлення про помилку під час виконання, якщо спробуєте скористатися ним. У разі створення нового подання **MainPage.xaml** можна використати, наприклад, ім'я **MainPage.Raspberry.xaml**.



Після того як до проекту додано нове подання, Visual Studio згенерує ще один метод **InitializeComponent**.



На цьому етапі Visual Studio нічого не перевіряє – просто додає новий метод, який застосовуватиметься для різних типів пристройів. Звичайно, якщо ви використовуєте тільки перед задані пристрої, метод **InitializeComponent** із параметром буде викликано автоматично. Однак за умови написання власної логіки можна робити виклик безпосередньо зі свого коду. Звичайно, слід змінити наявний конструктор сторінки і викликати **InitializeComponent** у такий спосіб:

```
this.InitializeComponent(new System.Uri(
"ms-appx:///MainPage.Raspberry.xaml"));
```

Цей підхід дає змогу реалізувати будь-яку логіку в конструкторі й застосовувати різні подання для різних пристройів, параметрів тощо.



Розділ 11.

## **Плитки та повідомлення**

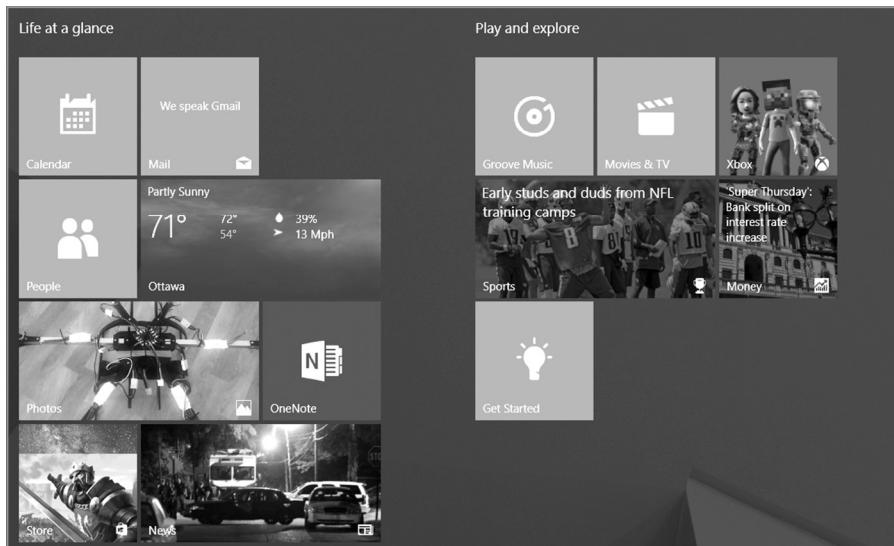
Перш ніж обговорювати, як публікувати програми в Магазині, слід розглянути ще одну тему, без якої важко починати роботу з Магазином програм. Це інформація про плитки, які представляють вашу програму. Крім того, ми обговоримо в одному контексті плитки та спливаючі повідомлення, завдяки яким забезпечується взаємодія з користувачами, коли програми не запущено.

У деяких розділах цього посібника вміщено відомості про сповіщення у фоновому режимі і сповіщення, які можна надсилати за допомогою служби сповіщень Windows.

Почнемо огляд з основних плиток.

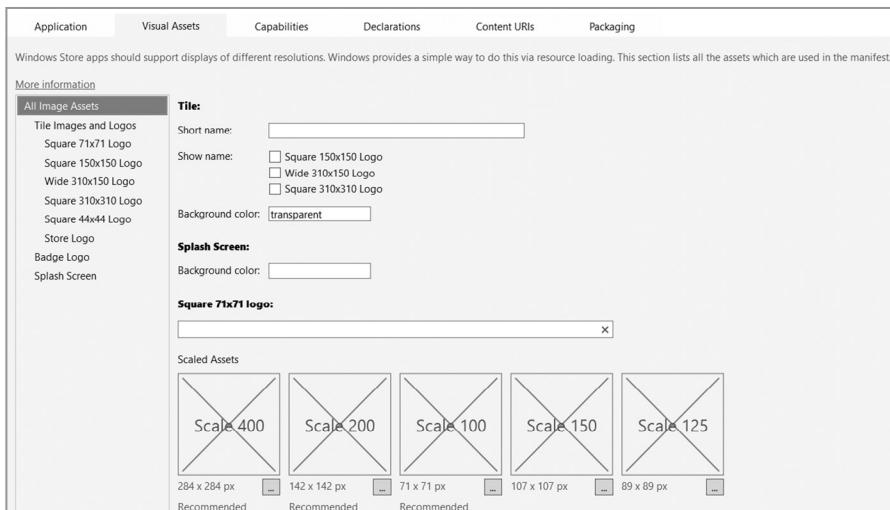
## Динамічні плитки

Усі програми мають плитки. Звичайно, функції плитки видаються неістотними, коли ви працюєте в режимі настільного ПК, але після переходу до режиму планшета або в разі використання планшета чи мобільного телефону плитка дає можливість настроїти початковий екран і персоналізувати інформацію на ньому: місцеві новини, погоду, зображення, інформацію про непрочитані повідомлення тощо.



Таким чином, з точки зору розробника, плитки – це засіб, що допомагає спілкуватися з користувачами, навіть коли програму закрито. І, зрештою, плитка

мотивує користувачів запускати програму знову і знову. Робота над плиткою для розробників починається з файлу маніфесту. У маніфесті зазначається, які типи плиток буде підтримувати ваша програма, а також стандартні параметри для плитки – фон, назва програми та зображення.



Програми підтримують до чотирьох видів плиток: малі, середні, широкі та велиki (тільки для настільних ПК); редагування маніфесту дає змогу використовувати зображення для кожного із цих видів. Зазвичай слід надіслати запит дизайнера на створення всіх необхідних піктограм для вашої програми. У маніфесті можна знайти всі рекомендовані розміри (в пікселях), які потрібно надати дизайнерові. Звісно, можна і не звертатися до дизайнерів, але наслідки можуть бути негативними: плитка презентує програму в Магазині, і зазвичай користувачі рідко встановлюють програми з погано оформленіх плиток. Тому варто витратити час, щоб створити справді якісні плитки. Зверніть увагу, що Windows 10 підтримує величезну кількість пристроїв із різними форм-факторами, роздільною здатністю та розмірами екрана. Ось чому краще подбати про створення піктограм усіх можливих масштабів. Звичайно, якщо ви використовуєте лише масштаб «100 %», Windows також буде змінювати розмір піктограм, але через те, що зображення растрое, його якість може знизитися.

У будь-якому разі ви повинні надати базовий набір зображень для стартового екрана, і це легко зробити за допомогою конструктора маніфестів. Він дає змогу створити зображення для заставки та емблеми. Зображення для заставки використовується під час завантаження програми, а емблема відображається на екрані блокування, якщо у вас є підтримка цієї можливості.

Якщо ви редагуєте код прямо в маніфесті, то за допомогою тегу **VisualElements** зможете задати тільки плитки малого та середнього розміру. Це обов'язкові два типи плиток, а за потреби створити також широкі й велики плитки потрібно використовувати внутрішній елемент **DefaultTile**:

```
<Applications>
  <Application Id="App" Executable="$targetnametoken$.exe"
    EntryPoint="TileApplication.App">
    <uap:VisualElements DisplayName="TileApplication"
      Square150x150Logo="Assets\Square150x150Logo.png"
      Square44x44Logo="Assets\Square44x44Logo.png"
      Description="TileApplication"
      BackgroundColor="transparent">
      <uap:DefaultTile
        Wide310x150Logo="Assets\Wide310x150Logo.png">
      </uap:DefaultTile>
      <uap:SplashScreen Image="Assets\SplashScreen.png" />
    </uap:VisualElements>
  </Application>
</Applications>
```

Звичайно, усі ці налаштування не дуже цікаві для розробників, але більшої уваги варто те, як зробити ці плитки динамічними. Давайте перейдемо від конструктора маніфесту до практики кодування, що пов'язана з плитками.

Після того, як ваша програма запрацює або буде запущена у фоновому режимі, можна буде оновити плитку програми. У динамічній плитці налічується кілька шарів: контент, коротке ім'я, піктограма програми (44 на 44 пікселі) і емблема (номер або символ). Звичайно, для невеликих плиток не має жодного сенсу змінювати стандартну поведінку, але для всіх інших типів можна легко оновити будь-який із цих шарів.

Коли ми викладали аналогічний матеріал для розробників Windows 8.x, зазвичай починали з простору імен **Windows.UI.Notifications**, який містив усі класи, що дають можливість працювати з плиткою та сповіщеннями. Але у Windows 10 краще почати з шаблонів, оскільки в них відбулися значні зміни.

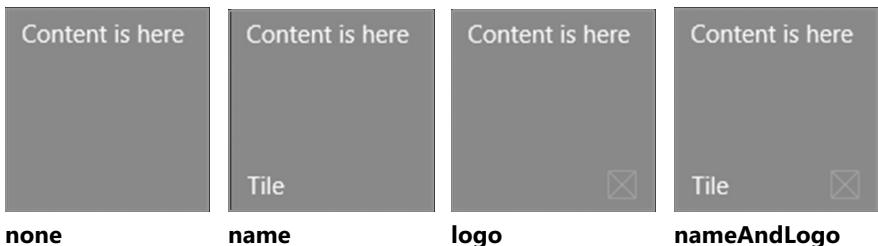
Шаблони визначають, як буде представлений контент, надаючи поля для заповнення текстом і зображеннями. Windows 8.x підтримує понад 40 шаблонів, і їх досі можна використовувати. Але Windows 10 надає розробникам можливість скористатися адаптивними інтерфейсами, що забезпечують багато способів подання контенту для різних пристройів, роздільної здатності та розмірів екрана, стандартних відстаней між користувачами та пристроями. Але в разі

використання адаптивного інтерфейсу, можливо, логічно буде застосовувати адаптивні плитки, щоб мати змогу подавати інформацію залежно від наявного простору. Ось чому у Windows 10 корпорація Microsoft запровадила спеціальний адаптивний шаблон, який забезпечує більшу гнучкість, ефективно працює в адаптивному середовищі та підтримує всі функції, які були реалізовані в понад 40 шаблонах попередніх версій. Тому якщо ви розробляєте нову програму для Windows 10, ми рекомендуємо використовувати новий адаптивний шаблон.

Як і раніше, перш ніж починати працювати із шаблоном, ви маєте потренуватися з XML. Так, на першому етапі вам необхідно згенерувати документ XML, який буде містити всі необхідні оновлення. На вищому рівні документ XML має містити такі елементи:

```
<tile>
  <visual>
    <binding template="TileSmall">
      . . .
    </binding>
    <binding template="TileMedium">
      . . .
    </binding>
    <binding template="TileWide">
      . . .
    </binding>
    <binding template="TileLarge">
      . . .
    </binding>
  </visual>
</tile>
```

Усі теги **binding** дають можливість визначити конкретну плитку, і ви можете оголосити їх усі або принаймні одну. Завдяки шаблону атрибутів можна визначити, які плитки потрібно буде оновлювати і розмістити весь контент усередині **binding**. Крім того, елементи **binding** і **visual** можуть містити атрибут **displayName**. У разі вибору елемента **visual** атрибут буде застосовуватися до всіх елементів **binding**, але якщо ви розміщуєте його для **binding**, він перепишє коротке ім'я для конкретної плитки. Елемент **visual** підтримує ще один важливий атрибут – **branding**. Завдяки цьому атрибуту можна вказати, чи бажаєте ви відображати невелику емблему і коротку назву для плитки. Ось зображення, яке показує подання різних значень **branding**:



Щоб почати тестування різних значень і параметрів, слід запустити такий код:

```
 XmlDocument xml = new XmlDocument();
xml.LoadXml(
    "<tile>" +
    "<visual branding=\"none\">" +
    "<binding template = \"TileMedium\""
    "displayName = \"Tile\">" +
    "<text>Content is here</text>" +
    "</binding>" +
    "</visual>" +
    "</tile>" +
);
var updater=Windows.UI.Notifications.TileUpdateManager.
CreateTileUpdaterForApplication();
TileNotification tile = new TileNotification(xml);
updater.Update(tile);
```

Це той самий код, який працює на Windows 8.x, але в цьому разі використано адаптивний шаблон. Так чи інакше, якщо вам потрібно оновити плитку, запакуйте всі оновлення в об'єкт  **XmlDocument**. Клас  **XmlDocument** розташовано в просторі імен  **Windows.Data.Xml.Dom**. Після цього слід завантажити туди XML із використанням методу  **LoadXml**. Коли плитка готова, для її оновлення можна використовувати два класи:  **TileUpdater** і  **TileNotification**.

Другий із цих класів використовується для створення об'єкта, що безпосередньо базується на  **XmlDocument**. Ви можете додати також час завершення дії, а якщо цього не зробите, то в разі незапуску програми протягом 3 днів Windows автоматично видалить повідомлення з плитки.

Елемент  **TileUpdater** неможливо створити безпосередньо, однак можна використовувати  **CreateTileUpdaterForApplication** класу  **TileUpdateManager**. І, нарешті, можна просто використати об'єкти  **TileUpdater** і  **TileNotification** для оновлення плитки.

Повернімось до адаптивного шаблона й обговоримо, який контент може відображати плитка.

Для подання і впорядкування контенту на плитці використовується два типи елементів. Щоб відобразити контент, застосовуються **text** і **image**, а для впорядкування – **group** і **subgroup**.

Почнемо з елемента **text**. Він може відображати на плитці будь-який текст з використанням таких атрибутів:

- **hint-style** – завдяки стилям можна визначати розмір шрифту, колір і значущість елементів тексту. Функція визначення власних стилів недоступна, але можна скористатися одним із багатьох попередньо визначених стилів, таких як **caption**, **body**, **base**, **subtitle**, **title**, **subheader**, **header**, **titleNumeral**, **subheaderNumeral**, **headerNumeral**, **captionSubtle**, **bodySubtle**, **baseSubtle**, **subtitleSubtle**, **titleSubtle**, **titleNumeralSubtle**, **subheaderSubtle**, **subheaderNumeralSubtle**, **headerSubtle**, **headerNumeralSubtle**.
- **hint-wrap** – за замовчуванням перенесення за словами не активовано, але якщо присвоїти цьому атрибуту значення **true**, текст переноситься на нові рядки.
- **hint-align** – цей атрибут дає можливість вирівняти текст по горизонталі. Можна використовувати його значення **right**, **left** або **center**.
- **hint-maxLines** і **hint-minLines** – дають змогу застосовувати мінімальну та максимальну кількість рядків.

Змініть наш документ XML, додавши ще один текстовий елемент і деякі атрибути:

```
xml.LoadXml(
    "<tile>" +
    "<visual branding=\"none\">" +
    "<binding template = \"TileMedium\""
        displayName = \"Title\">" +
    "<text hint-style=\"title\">Content is here</text>" +
    "<text hint-style=\"body\" hint-wrap=\"true\">
        Content is here</text>" +
    "</binding>" +
    "</visual>" +
    "</tile>" +
);
```

Як бачите, перший рядок уривається, а для другого було використано атрибут перенесення і це дало ефект.



Ще одним атрибутом, який допоможе вирівняти текст, є **hint-textStacking**. Він дає змогу вирівнювати текст по вертикалі; його також можна застосувати до елемента **binding** або **subgroup**.

Наступний елемент, який може відображати контент, – це **image**. Основними його атрибутами є такі.

- **placement** – ця властивість підтримує три значення: **inline**, **background** і **peek**. Значення **inline** застосовне, коли зображення і текст розміщаються разом та текст вирівнюється відповідно до положення зображення. Завдяки значенню **background** зображення можна використати як тло, поверх якого розміщується текст. У разі вибору значення **peek** зображення буде використовувати анімацію руху вгору і вниз.
- **src** – джерело зображення.
- **hint-crop** – дає можливість обітнути зображення за допомогою кола – цьому атрибуту потрібно присвоїти значення **circle**.
- **hint-align** – цей атрибут підтримує такі значення: **stretch** (розтягнуто), **left** (зліва), **center** (по центру) і **right** (справа). Завдяки йому можна вказати, як розташувати зображення на плитці.
- **hint-removeMargin** – стандартне зображення має 8-піксельні поля, але їх можна видалити, присвоївши цьому атрибуту значення **true**.

Продовжимо попередній приклад, додавши зображення як тло плитки:

```
xml.LoadXml(
    "<tile>" +
    "<visual branding=\"none\">" +
    "<binding template = \"TileMedium\""
        displayName = \"Tile\\"" +
    "<image src=\"Assets/drone.jpg\""
        placement=\"background\""+
        " hint-align=\"stretch\"/>" +
    "<text hint-style=\"title\">Content is here</text>" +
    "<text hint-style=\"body\" hint-wrap=\"true\">
        Content is here</text>" +
```

```

    "</binding>" +
    "</visual>" +
    "</tile>" +
);

```

Результат відображеного нижче:



У цьому прикладі для заповнення простору плитки використано атрибут **placement**. Оскільки для тла плитки використовується зображення, атрибут **hint-removeMargin** використовувати не потрібно.

Наступні два елементи, які будуть корисні для створення адаптивних плиток – це **group** і **subgroup**.

Перший елемент дає можливість розташувати весь вміст, що має бути відображенний на плитці, або не відображати його зовсім. Це дуже важливий елемент для створення адаптивних плиток: ви не можете передбачити всі можливі розміри плитки, але натомість можна розташувати декілька груп всередині, і плитка буде відображати стільки груп, скільки можливо. Отже, якщо користувач буде використовувати великі плитки на великому екрані, плитка відображатиме декілька груп, але на меншому екрані та сама плитка відображатиме меншу кількість груп.

Кожний елемент **group** містить принаймні один елемент **subgroup**, який дає можливість розташовувати контент плитки у стовпцях. Розгляньмо код нижче:

```

xml.LoadXml(
    "<tile>" +
    "<visual branding=\"none\">" +
    "<binding template = \"TileWide\""
        displayName = \"Title\">" +
    "<group>" +
    "<subgroup hint-weight=\"1\">" +
    "<text hint-style=\"body\" hint-align =\"center\">
        Team A</text>" +

```

```
"<text hint-style=\"headerNumeral\"  
hint-align =\"center\">>0</text>" +  
"</subgroup>" +  
"<subgroup hint-weight=\"1\>" +  
"<text hint-style=\"body\> hint-align =\"center\>  
Team B</text>" +  
"<text hint-style=\"headerNumeral\>  
hint-align =\"center\>2</text>" +  
"</subgroup>" +  
"</group>" +  
"</binding>" +  
"</visual>" +  
"</tile>"  
) ;
```

Завдяки цьому коду можна підготувати для плитки інформацію про поточний рахунок у футбольному матчі.



Ширина стовпця була визначена як для стовпців сітки. Ми поділили плитку на декілька частин і вказали, скільки частин використовує кожен стовпець (у цьому прикладі – загалом 2 частини, по 1 частині для кожного стовпця).

Отже, ми вже знаємо, як працювати із вмістом, плитками, піктограмами, але є ще один елемент, який можна розміщувати на плитці, – емблема. Це номер чи гліф, що розміщується в правому нижньому куті плитки. Зазвичай цей елемент розміщують на плитках програм для передачі повідомлень для позначення кількості непрочитаних повідомлень, але його можна використовувати й інакше.

На жаль, неможливо визначити емблему за допомогою шаблону адаптивної плитки. Для того, щоб змінити або створити емблему, вам потрібно використати різні класи: **BadgeUpdateManager**, **BadgeNotification**, **BadgeUpdater** і **BadgeTemplateType**. Усі ці класи розташовані в просторі імен **Windows.UI.Notifications**; додатково для роботи з документом XML вам знадобиться простір імен **Windows.Data.Xml.Dom**.

Спочатку вам буде потрібний шаблон для емблеми. Як і шаблон для плитки, це документ XML, але його не потрібно створювати «з нуля». Замість цього для отримання шаблону скористайтеся **BadgeUpdateManager**:

```
var template=BadgeUpdateManager.GetTemplateContent
(BadgeTemplateType.BadgeNumber);
```

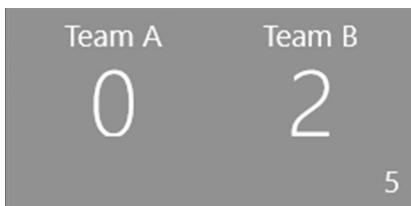
Метод **GetTemplateContent** приймає **BadgeTemplateType** як параметр і може повернути шаблон для числа чи гліфів. У будь-якому разі метод поверне об'єкт **XmlDocument**, і число для емблеми можна додати тут:

```
XmlElement element = (XmlElement)template.SelectSingleNode
("/badge");
element.SetAttribute("value", "5");
```

Врешті-решт, можна створити об'єкт класу **BadgeNotification** й оновити плитку, застосувавши атрибут **BadgeUpdater**:

```
BadgeNotification badge = new BadgeNotification(template);
BadgeUpdateManager.CreateBadgeUpdaterForApplication().
Update(badge);
```

Після запуску цього коду в нижньому правому куті плитки з'явиться невелика цифра:



Можливо, ця цифра для нашої програми не означає нічого, але метод працює ☺.

## Спливаючі сповіщення

Фактично ми вже розпочали роботу зі сповіщеннями. Оновлюючи плитку чи емблему, ми надсилаємо сповіщення системі за допомогою шаблону, і Windows виконує оновлення. Очевидно, такі сповіщення можна надсилати не тільки з локально розташованої програми. Windows застосовує декілька методів доставки сповіщень:

- **Local** – для надсилання сповіщення для плитки з самої програми.
- **Scheduled** – дає можливість запланувати сповіщення, навіть коли ваша програма неактивна; користувачі можуть переглядати заплановані сповіщення.
- **Periodic** – дає змогу надіслати запит системі для отримання оновленої інформації з хмари та оновити плитку з її врахуванням.
- **Push** – дає можливість налаштувати інтеграцію зі службою Windows Notification Service, розташованою в хмарі. Завдяки цій службі надходять сповіщення від сервера до вашої програми.

Спливаючі сповіщення розглянемо пізніше, зараз зосередьмося на огляді перших трьох типів.

Ми вже використовували локальні сповіщення **Badge** і **Tile**; ОС Windows підтримує ще два типи сповіщень – **Toast** і **Raw**. Сповіщення **Toast** містяться в Action Center; вони вільно переміщаються по екрану, і користувачу потрібно клацнути таке сповіщення, щоб перейти до програми. Отже, сповіщення **Toast** – це ще один спосіб запустити вашу програму та привернути до неї додаткову увагу. Для сповіщень **Toast** застосовуються методи **Local**, **Scheduled** і **Push**. Сповіщення **Raw** працюють лише з методом **Push**, про це детальніше йтиметься в наступних розділах.

Перш ніж дізнатися, як використовувати сповіщення **Scheduled** і **Periodic**, слід ознайомитися з функціональністю сповіщень **Toast**.

У Windows 10 з'явився адаптивний шаблон не лише для сповіщень плиток, а й для спливаючих сповіщень. Спливаючі сповіщення підтримують декілька типів внутрішніх елементів: текст, зображення, дії та аудіо. Отже, для них немає груп чи підгруп елементів, але вони й не потрібні для спливаючих сповіщень. Як і в випадку з плиткою, визначити спливаюче сповіщення можна за допомогою документа XML. Зазвичай можна використовувати поданий нижче шаблон:

```
<toast>
  <visual>
    <binding template="ToastGeneric">
      . .
      </binding>
    </visual>
    <actions>
      . .
    </actions>
    <audio src="ms-appx:///..." />
  </toast>
```

Переважно використовується елемент **binding**, що може містити текст чи зображення. Як і у випадку зі сповіщеннями плитки, текстові блоки та зображення визначаються з використанням елементів **text** і **image**, але ці елементи майже не містять жодних атрибутів, хоча в елемента **image** все-таки є атрибути **src** і **hint-crop**. Крім того, для зображення, що підтримує тільки два значення, **inline** і **appLogoOverride**, можна використовувати й атрибут **placement**. Значення **inline** дає можливість поєднувати текст і зображення всередині повідомлення, а **appLogoOverride** – використовувати власне зображення для емблеми замість одного зі стандартних.

Розгляньмо приклад, що дає можливість запускати локальні спливаючі сповіщення:

```
 XmlDocument xml = new XmlDocument();
xml.LoadXml(
    "<toast>" +
    "<visual>" +
    "<binding template=\"ToastGeneric\">" +
    "<text>Football application</text>" +
    "<text>Team A - Team B - 2:0</text>" +
    "<image src=\"Assets/drone.jpg\""
        placement=\"appLogoOverride\" />" +
    "</binding >" +
    "</visual>" +
    "</toast>");

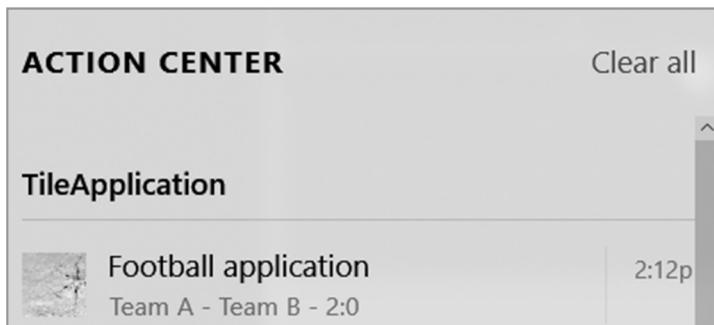
var toastNot = ToastNotificationManager.CreateToastNotifier();
ToastNotification toast = new ToastNotification(xml);
toastNot.Show(toast);
```

Як бачите, ми використовуємо майже ті самі класи для надсилання сповіщень, однак слово **Tile** замінено на **Toast**. Після запуску цього коду виникає таке сповіщення:



**Football application**  
Team A - Team B - 2:0

Таке саме сповіщення з'являється і в Action Center. Тобто якщо користувач пропустив повідомлення, він зможе переглянути його пізніше.



Якщо закрити програму й клацнути сповіщення, програма запуститься знову. Отже, спливаючі сповіщення можна використовувати як ще один механізм активації програм. Для стеження за активаціями, що виникають після клацання на сповіщеннях, потрібно реалізувати метод **OnActivated**. Його вже згадували в попередніх розділах, а зараз продемонструємо його шаблон:

```
protected override void OnActivated(IActivatedEventArgs args)
{
    if (args.Kind == ActivationKind.ToastNotification)
    {
        //якісь дії
    }
}
```

Спливаючі сповіщення можна також застосовувати для активації фонових задач, але про це йтиметься в розділі про фонові задачі.

Наступним елементом шаблону є аудіо. Цей елемент дає вам можливість задати власний звук отримання сповіщення.

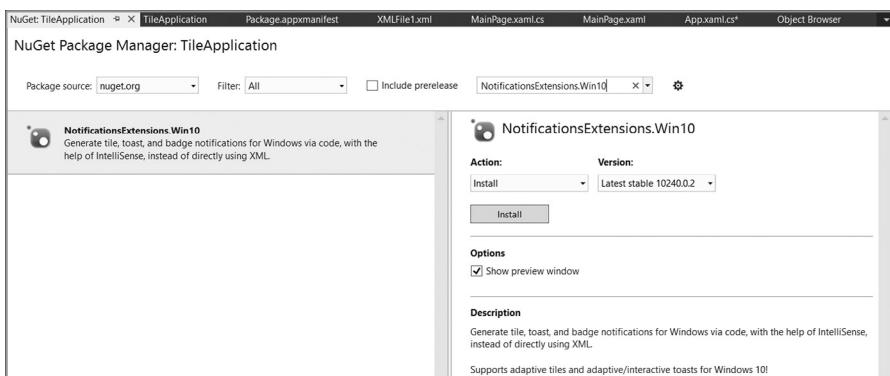
Останнім і найцікавішим елементом є **actions**. Він підтримує взаємодію з користувачем навіть без запуску програми. Цей елемент може містити три різні типи елементів керування:

- **input** – елемент може відображати текстовий блок або поле зі списком, залежно від значення атрибута **type**. Якщо атрибут **type** має значення **text**, відображається текстовий блок, а якщо значення **selection** – поле зі списком.
- **selection** – елемент дає можливість задавати елементи для поля зі списком. Тобто ці елементи розташовані всередині елемента **input** і підтримують атрибути **id** і **content**.

- **action** – елемент дає можливість оголошувати кнопки. Атрибути **content**, **arguments** і **imageUri** використовуються для розміщення тексту на кнопці або зображенні, а також для передачі тексту до самої програми. Цей елемент має атрибут **activationType**, що дає можливість запускати певний код в активному чи фоновому режимі.

Ми вже знаємо, як використовувати адаптивні шаблони для плитки та спливаючих сповіщень. Але якщо виникає потреба обйтися без використання шаблона XML, це можна зробити за допомогою розширення сповіщень.

Відкрийте менеджер пакетів NuGet і додайте пакет **NotificationsExtensions.Win10**.



У пакеті міститься декілька просторів імен (**NotificationsExtensions.Badges**, **NotificationsExtensions.Tiles**, **NotificationsExtensions.Toasts**), кожен із яких містить багато класів. Але завжди слід починати з класів **ToastContent**, **TileBindingContentAdaptive**, **BadgeNumericNotificationContent** чи **BadgeGlyphNotificationContent** і продовжувати створювати сповіщення, залучаючи й інші класи. Наприклад, у коді, наведеному нижче, створено спливаюче сповіщення, аналогічне щойно розглянутому, але зараз буде використано елемент **ToastContent** і присвоєно деякі властивості із застосуванням об'єктів класів **ToastVisual**, **ToastText**, **ToastAppLogo**:

```
ToastContent toastDoc = new ToastContent()
{
    Visual = new ToastVisual()
    {
        TitleText = new ToastText()
        {
            Text = "Football application"
        }
    }
}
```

```
        },
        BodyTextLine1 = new ToastText()
        {
            Text = "Team A - Team B - 2:0"
        },
        AppLogoOverride = new ToastAppLogo()
        {
            Source = new ToastImageSource("Assets/drone.jpg")
        }
    }
};

var toastNot = ToastNotificationManager.CreateToastNotifier();
ToastNotification toast = new ToastNotification
(toastDoc.GetXml());
toastNot.Show(toast);
```

Отже, ми вже знаємо, як працювати із шаблоном для спливаючих сповіщень і як активувати локальні сповіщення, а тепер приділимо увагу сповіщенням **Scheduled** і **Periodic**.

## Заплановані сповіщення

Завдяки запланованим сповіщенням оновлювати плитку програми або надсилати повідомення користувачу можна навіть тоді, коли сповіщення не активоване. Однак, коли програму запущено, фактично будь-які зміни плитки або спливаючі сповіщення можна налаштувати завчасно. Звичайно, у цьому разі користувачі не можуть отримувати актуальній контент, тому що вам потрібно заздалегідь формувати XML-документ, але це хороший спосіб привернути увагу користувача.

Створення запланованих сповіщень не становить труднощів. Вам лише потрібно використовувати класи **ScheduledTileNotification** і **ScheduledToastNotification** замість **TileNotification** і **ToastNotification** і застосовувати метод **AddToSchedule** замість **Show** і **Update**.

Наприклад, код нижче приведе до появи спливаючих сповіщень п'ять разів на хвилину; їх запуск почнеться через хвилину після виконання коду:

```
var toastNot = ToastNotificationManager.CreateToastNotifier();
ScheduledToastNotification toast = new ScheduledToastNotification(
    toastDoc.GetXml(), DateTime.Now.AddMinutes(1),
```

```
TimeSpan.FromMinutes(1), 5);  
toastNot.AddToSchedule(toast);
```

Звичайно, у деяких випадках вам буде потрібно видалити наявні заплановані сповіщення. Для цього скористайтеся методами **GetScheduledToastNotifications** і **RemoveFromSchedule** класу **ToastNotifier** для спливаючих сповіщень чи **GetScheduledTileNotifications** і **RemoveFromSchedule** класу **TileUpdater** для плиток.

## Періодичні сповіщення

Періодичні сповіщення цікаві тим, що дають можливість отримувати оновлену інформацію із сервера. Але, застосовуючи періодичні сповіщення, ви можете оновити тільки плитки чи емблеми. Для роботи зі спливаючими повідомленнями вам потрібно використовувати службу Windows Notification Service.

Для формування запиту й отримання інформації із сервера служба Windows Runtime надає метод **StartPeriodicUpdate**, який використовується в класах **TileUpdater** і **BadgeUpdater**. Цей метод вимагає принаймні двох параметрів для URI, за яким можна знайти оновлені документи XML, і для проміжку часу, що дає можливість визначити частоту оновлень. Загалом, щоб почати оновлення плитки програми, можна скористатися поданим нижче блоком коду:

```
var updater = TileUpdateManager.CreateTileUpdaterForApplication();  
updater.StartPeriodicUpdate(new Uri("http://toastsbaydachtest.  
azurewebsites.net/ToastHandler.ashx"),  
PeriodicUpdateRecurrence.Hour);
```

У цьому разі Windows буде щогодини запитувати новий XML-документ із вказаного URI. Майте на увазі, що плитку буде оновлено негайно, але якщо потрібно ініціювати процес за деякий час, доведеться використати ще один параметр для методу **StartPeriodicUpdate**. Якщо необхідно припинити запити до сервера, скористайтеся методом **StopPeriodicUpdate**.

Крім того, клас **TileUpdater** містить метод **StartPeriodicUpdateBatch**, завдяки якому можна передати до 5 URI з різними XML-документами. Застосувавши цей метод, ви зможете сформувати чергу з плиток, які будуть відображатися до наступного оновлення. Цю функцію активують за допомогою методу **EnableNotificationQueue**.

Для того, щоб перевірити використання періодичних сповіщень, слід розгорнути веб-службу, яка повертатиме XML. Це той самий XML-код, який раніше використовувався локально. Рекомендуємо скористатися Azure Web Sites і розгорнути простий проект зі звичайним обробником. Тоді можна буде зосередитися на XML-коді, а не на веб-службі.

Наприклад, один із варіантів реалізації виглядає так:

```
public void ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/plain";
    context.Response.Write(
        "<tile>" +
        "<visual branding=\"none\">" +
        "<binding template = \"TileWide\" " +
        "displayName = \"Tile\">" +
        "<group>" +
        "<subgroup hint-weight=\"1\">" +
        "<text hint-style=\"body\" " +
        "hint-align =\"center\">>Team A</text>" +
        "<text hint-style=\"headerNumeral\" " +
        "hint-align =\"center\">>0</text>" +
        "</subgroup>" +
        "<subgroup hint-weight=\"1\">" +
        "<text hint-style=\"body\" " +
        "hint-align =\"center\">>Team B</text>" +
        "<text hint-style=\"headerNumeral\" " +
        "hint-align =\"center\">>2</text>" +
        "</subgroup>" +
        "</group>" +
        "</binding>" +
        "</visual>" +
        "</tile>"
    );
}
```

На інсталяцію й розгортання було витрачено близько 5 хвилин.

У цьому розділі ми тільки почали знайомитися з плитками та сповіщеннями. Далі буде розглянуто, як працювати зі сповіщеннями у фоновому режимі, а також особливості роботи зі службою Windows Notification Service і неформатованими сповіщеннями.

Розділ 12.

## **Як публікувати програми в Магазині**

Наприкінці першої частини книги поговорімо про те, як публікувати програми в Магазині. У цьому розділі ми розповімо, як створити обліковий запис розробника, що потрібно знати, перш ніж публікувати програми, і як це зробити. До цієї теми ми повернемося в розділі 26, де обговоримо платні програми, рекламу та інші функції, пов'язані з отриманням доходу від Магазину.

## Створення облікового запису для публікації

Передусім потрібно створити обліковий запис розробника, який дасть змогу публікувати програми в Магазині. Найкращий спосіб зареєструватися в Магазині надає сайт <http://dev.microsoft.com>. На цьому сайті легко знайти посилання на документацію, статті, інструменти тощо. Клацнувши посилання на панель керування (**Dashboard**), можна зареєструватися в Магазині або ввійти у свій обліковий запис.

Насамперед зауважте, що для доступу до панелі керування потрібно створити ідентифікатор Live ID. Він дає змогу пов'язати програму з Магазином (завантажити сертифікати та підпис), публікувати програми, перевіряти звіти тощо. Два або більше ідентифікаторів Live ID неможливо пов'язати з одним тим самим обліковим записом, як і надати певні дозволи кожному з них. Отож переконайтесь, що обрали найкращий з ідентифікаторів.

Обравши Live ID та ввійшовши до системи, розпочинайте реєстрацію. На першому етапі необхідно вибрати тип облікового запису:

## Розділ 12. Як публікувати програми в Магазині

The screenshot shows the 'Registration - Account info' page of the Windows Dev Center. At the top, there's a navigation bar with links for Home, Explore, Docs, Downloads, Samples, Community, Programs, and Dashboard. On the left, there's a sidebar with 'Account info' selected, followed by Payment and Review. The main content area has two sections: 'Account country/region' and 'Account type'. In the 'Account country/region' section, 'Canada' is selected from a dropdown menu. A note below says: 'Select the country/region where you live or where your business is located. Once you complete your account info, you can't change your account country/region.' In the 'Account type' section, there are two options: 'Individual' (selected) and 'Company'. The 'Individual' option costs 20.00 CAD and allows developing and selling apps as an individual, student, or unincorporated group. The 'Company' option costs 99.00 CAD and allows developing and selling apps using a regionally recognized and registered business name, plus access to advanced analytics and additional app capabilities. At the bottom, there are links for English (United States), Privacy and cookies, Terms of use, Trademarks, © 2015 Microsoft, and Feedback.

Є два типи облікових записів: персональний і корпоративний. Другий належатиме вашій компанії, а перший – особисто вам. Очевидно, що вони різняться за ціною. Персональний коштує 20 канадських доларів, корпоративний – 99. Це разовий платіж, та перш ніж віддати цю суму, не полінуйтеся дізнатися, як можна уникнути оплати взагалі.

Наприклад, якщо ви студент, то маєте доступ до веб-сайту DreamSpark.com, звідки всі студенти можуть завантажити програмне забезпечення Microsoft безкоштовно. Там також надається можливість створити безкоштовний обліковий запис для Магазину. Те саме стосується передплатників MSDN. На сторінці переваг передплати MSDN є промо-код, за допомогою якого можна зареєструватися безкоштовно.

Якщо ви оберете персональний обліковий запис, то, ймовірно, дістанете змогу публікувати програми впродовж 10 хвилин. Власникам корпоративного облікового запису доведеться зчекати якийсь час, доки триватиме перевірка компанії.

Поле для введення країни є дуже важливим. Слід указати країну ведення бізнесу й отримання прибутку. Змінювати країну або тип облікового запису не можна.

На наступному етапі необхідно вибрати псевдонім видавця й надати контактну інформацію. Для публікації можна обрати будь-яке ім'я, якщо воно, звичайно, ще не зайняте:

The screenshot shows a web browser window for the Windows Dev Center at <https://dev.windows.com/en>. The page is titled 'Publisher display name'. It contains fields for entering a first name, last name, email address, phone number, website, address, city, state/province, postal code, and preferred email language. A note below the publisher display name field states: 'Customers will see your apps listed in the Store under your unique publisher display name. You must have permission to use the name you select here.' A 'Contact info' section follows, with a note: 'We use this info for account verification and to contact you about your account.' A 'Next' button is at the bottom.

У разі реєстрації корпоративного облікового запису потрібно буде надати підтвердження:

The screenshot shows a 'Company approver' section of the registration form. It asks for company approval information, stating: 'Please provide the following info so we can verify that you are authorized to create and manage this account on behalf of your organization.' It includes fields for first name, last name, email address, and phone number, mirroring the structure of the previous page's contact information.

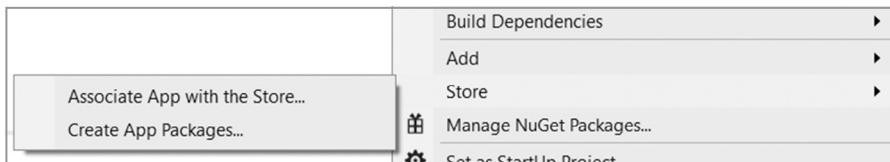
Ця інформація необхідна незалежній агенції, що перевірятиме, чи є у вас права на створення облікового запису. Зверніть особливу увагу, що в разі створення корпоративного облікового запису ви маєте використовувати корпоративні адреси електронної пошти. По завершенню реєстрації агенція зв'яжеться із затверджувачем і попросить його надіслати певний перелік документів. Зазвичай цей процес триває 3–5 днів, тож майте терпіння. Звичайно, власникам персональних облікових записів не потрібно проходити будь-які перевірки.

На наступному етапі потрібно ввести промо-код, якщо ви його маєте як користувач dreamspark.com або MSDN, чи надати платіжну інформацію. На цьому все. Після оплати вас буде пересправлено на панель керування, де ви зможете розпочати публікацію першої програми.

Описувати панель керування тут недоречно, але пізніше ми поговоримо про деякі компоненти, пов'язані зі звітністю та платежами. Зараз – лише кілька слів про оплату. Перед публікацією першої програми, яка не є безкоштовною, необхідно налаштувати відповідний обліковий запис. Для цього треба буде заповнити дві форми: форму з даними про платіжний рахунок – можна надати інформацію про банківський рахунок або PayPal, а також форму з податковою інформацією.

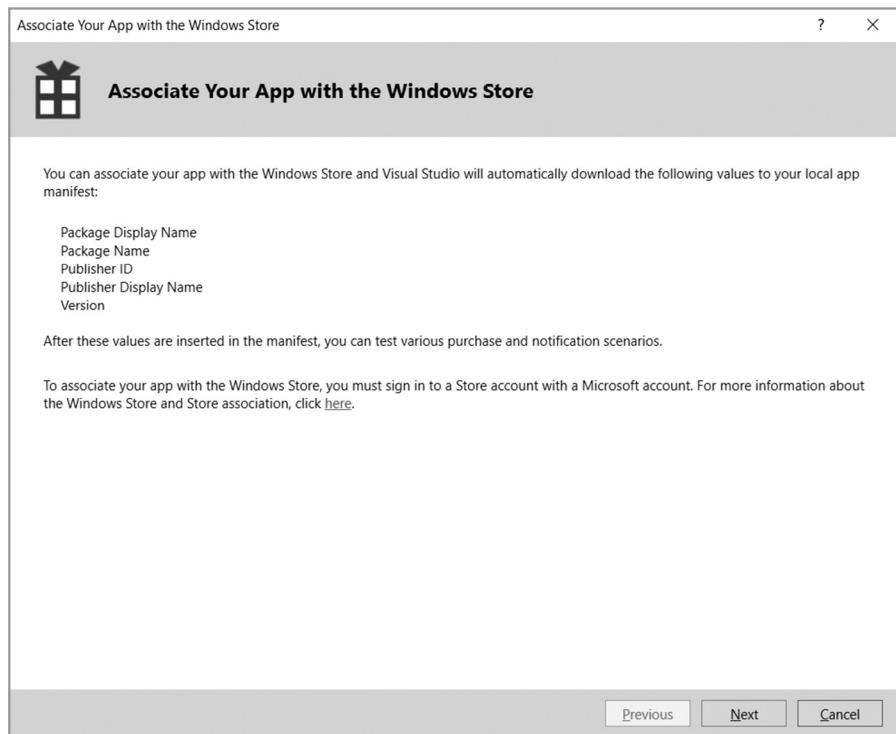
## Підготовка до публікації програми

Після створення облікового запису можна розпочинати підготовку пакету програми до публікації. Клацнувши назву проекту та активувавши контекстне меню, знайдіть пункт **Store**:

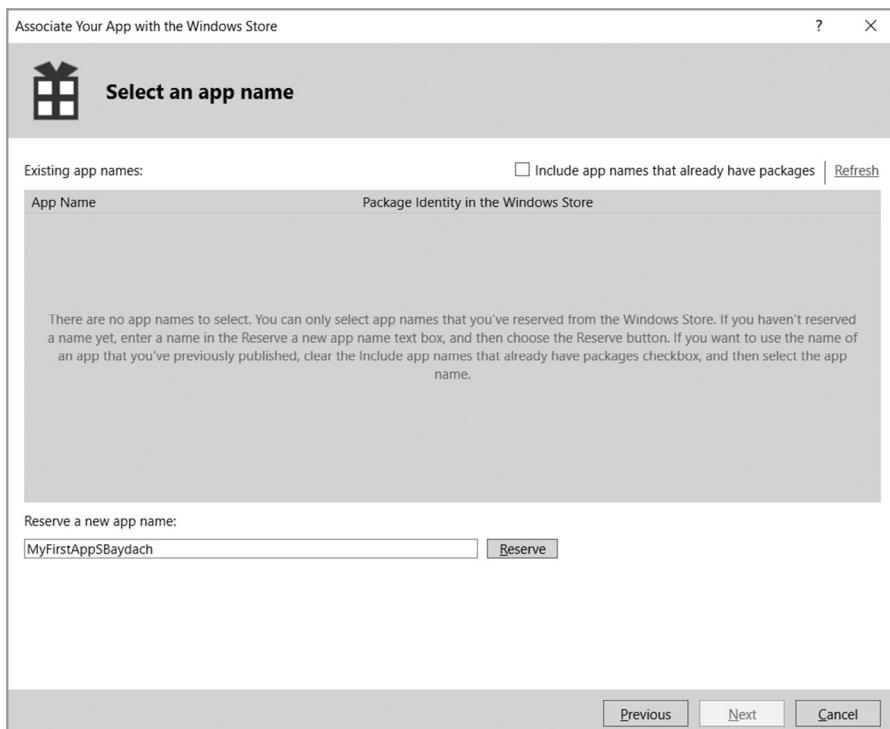


За допомогою команд **Associate App with the Store** і **Create App Packages** можна зарезервувати ім'я для програми і підписати пакет сертифікатом, пов'язаним із обліковим записом. Є два пункти меню, оскільки деякі функції, наприклад push-сповіщення і карти, не працюють належним чином із цифрового локального сертифікатом з власним підписом. Тож якщо ви хочете перевірити ці функції, але не готові публікувати програму, виберіть перший пункт меню. Для публікації – другий.

Якщо вибрали пункт меню **Associate App with the Store**, з'явиться помічник, що допоможе вам завершити процес:



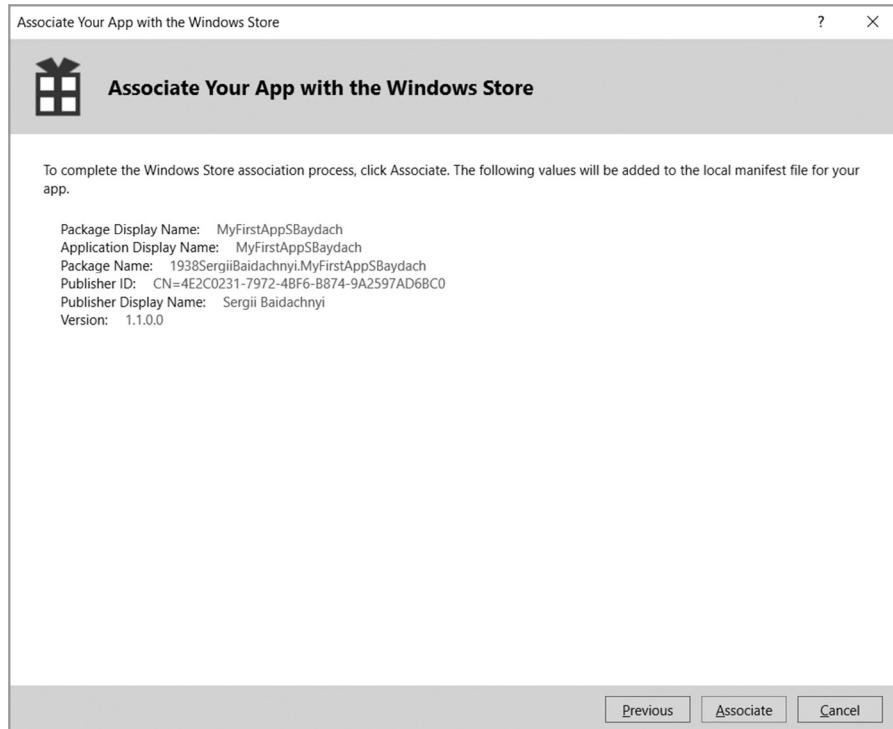
Тепер увійдіть до Магазину. Відразу після цього виконайте наступний етап – виберіть або зарезервуйте назву програми.



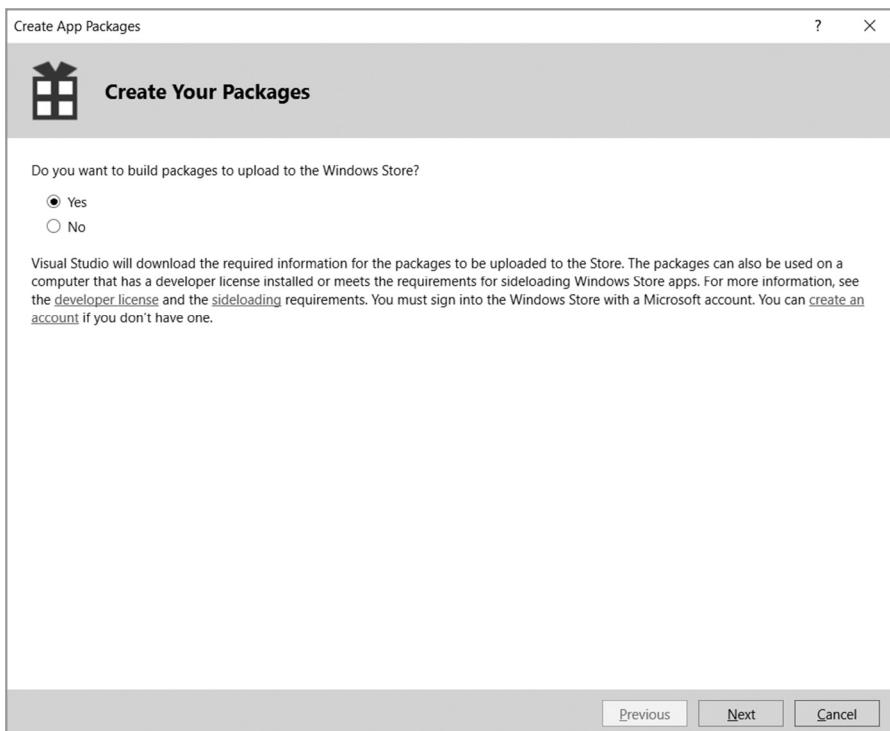
Якщо ви вже зарезервували назву програми на панелі керування або опублікували попередню версію програми, можете вибрати назву зі списку. Зважайте, що всі назви, для яких уже є пакети, за замовчуванням приховано. Перш ніж публікувати нову програму, можна замовити нову назву. Її бачитимуть користувачі, отже, підходьте до цього питання виважено: якщо назву вже використовують, ви не зможете зарезервувати її і мусите вибрати нову.

Коли назву вибрано, майстер надасть вам інформацію про зв'язування та сертифікати, а також про застосування змін до маніфесту програми.

## Windows 10 для C# розробників



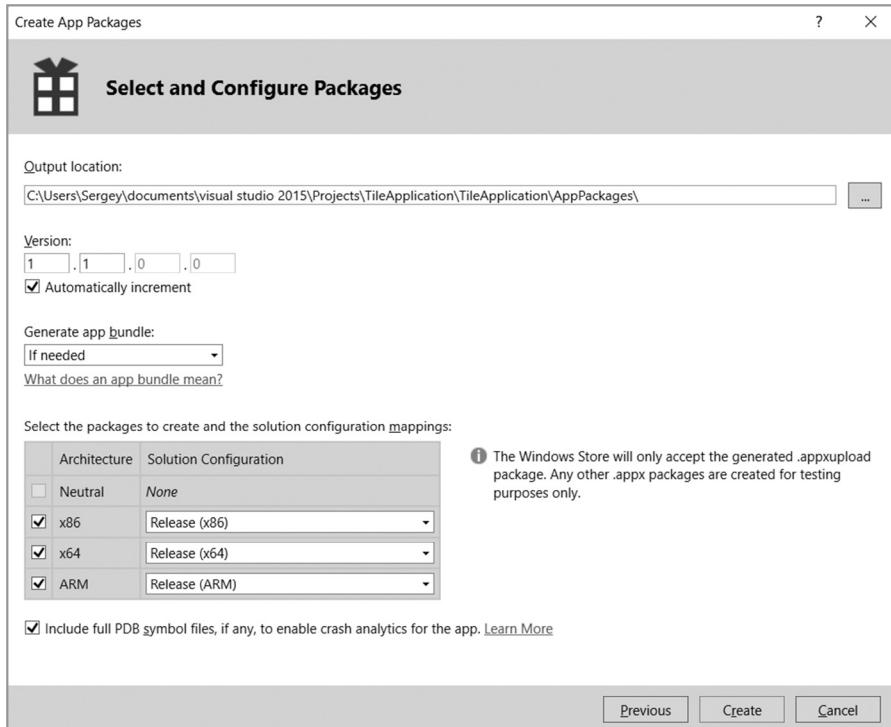
Коли ви будете готові опублікувати свою програму, виберіть пункт меню **Create App Packages**. Погляньте на перше вікно майстра:



На цьому етапі слід вирішити, який пакет вам потрібний, оскільки є спосіб створити пакет, що не буде опублікований у Магазині, але поширюватиметься серед тестерів для інсталяції на своїх комп'ютерах. Якщо ви оберете **No**, Visual Studio створить папку зі сценарієм Power Shell, пакет і всі необхідні складові. Тож якщо необхідно розгорнути програму на іншому комп'ютері, просто скопіюйте цю папку та запустіть сценарій Power Shell. Звичайно, комп'ютер має бути активовано для розробки.

На наступному етапі з'явиться той самий екран вибору назви, що й на попередньому. Тому цей етап ми просто пропустимо.

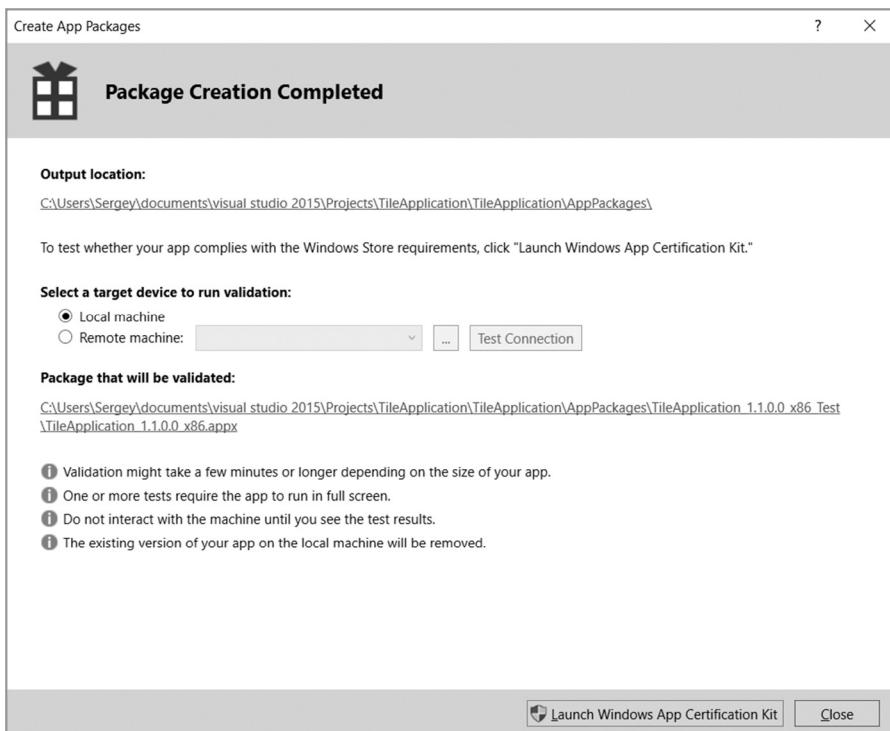
Далі слід вибрати цільовий каталог для пакета, версію та підтримувані платформи. Оскільки Windows 10 працює на різних апаратних платформах, можна створити пакет для x86, x64 і навіть для ARM:



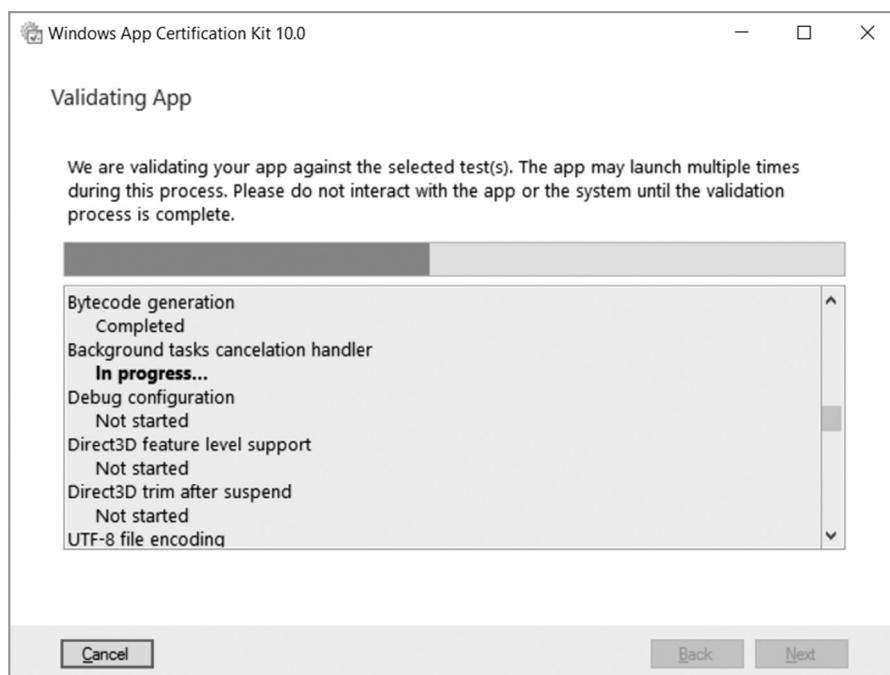
Після натискання кнопки **Create** пакет готовий. Visual Studio підготує файл **.appxupload**, який потрібно завантажити до Магазину.

Варте уваги те, що для фінальних версій програми Visual Studio використовує нативну компіляцію .NET. Це новий метод, що його Visual Studio 2015 застосовує для попередньої компіляції програм у нативний код. За компіляції програм в режимі налагодження буде застосовано традиційний метод.

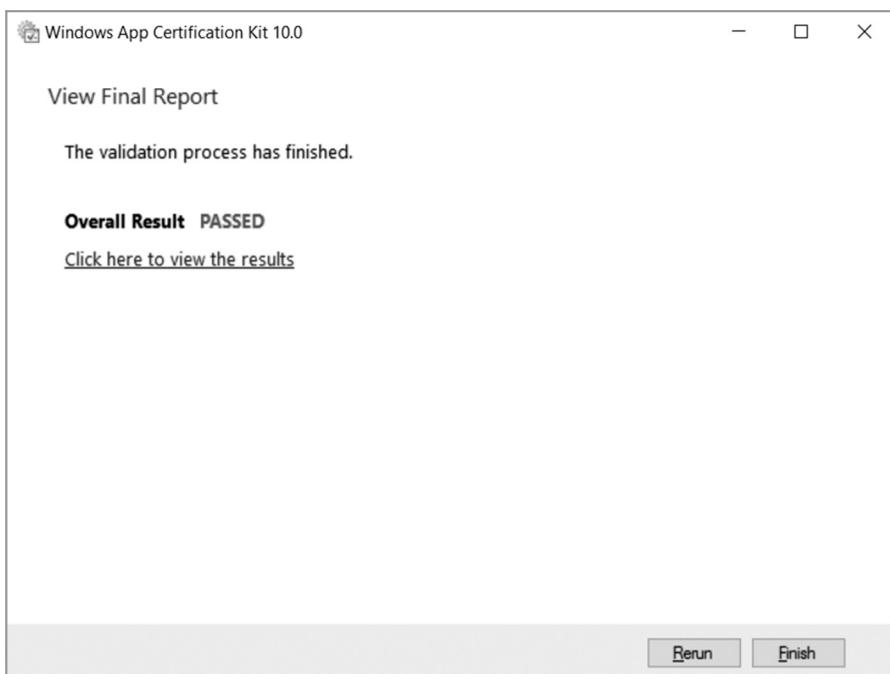
Останній крок перед публікацією – перевірка пакета. Ви маєте пересвідчитися, що програма не використовує жодних заборонених інтерфейсів API, має всі логотипи та піктограми й оголошує всі можливості як належить. Після того як ви опублікуєте програму, корпорація Microsoft попервах застосовуватиме цей засіб для її тестування, але якщо ви знайдете деякі проблеми ще в локальному варіанті, то заощадите час. Можна розпочати перевірку на останньому кроці майстра або запустити програму App Certification Kit вручну:



Не чіпайте комп'ютер, доки майстер App Certification Wizard не завершить роботу. Він запустить вашу програму кілька разів, відкриє консоль, тож ви все одно не матимете змоги виконувати на ПК жодних дій:

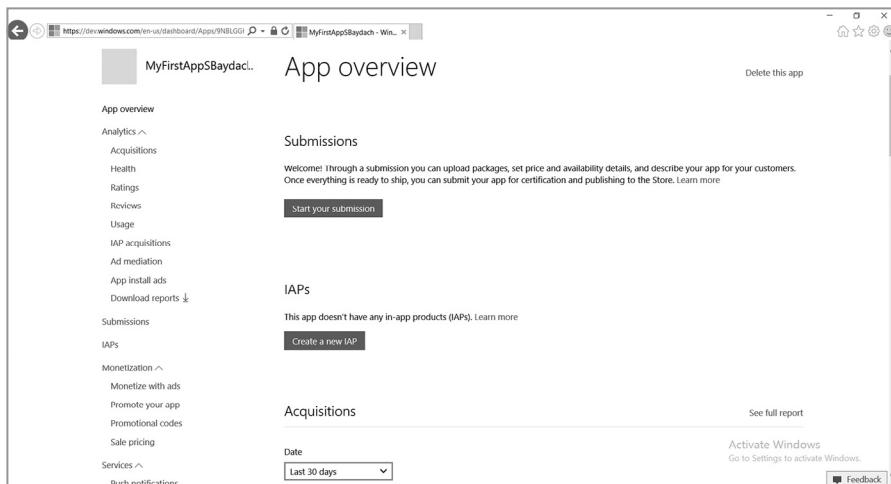


Якщо все гаразд, з'явиться повідомлення, що перевірку завершено успішно і ви можете опубліковати свою програму в Магазині. Якщо виникне якась проблема, її спершу потрібно буде вирішити.



## Публікація

Після того як пакет підготовлено і перевірено, його можна надсилати до Магазину. Для цього відкрийте панель керування на сторінці [dev.windows.com](https://dev.windows.com) і виберіть програму зі списку.



Натисніть кнопку **Start your submission**, і вас буде переспрямовано на таку сторінку

Submission 1

Delete

Pricing and availability	Not started	<input type="radio"/>
App properties	Not started	<input type="radio"/>
Packages	Not started	<input type="radio"/>
Descriptions	Not started	<input type="radio"/>
You'll be able to edit your descriptions after you upload packages.		
Notes for certification	Optional	<input type="radio"/>
Submit to the Store	Activate Window Go to Settings to activate	

Немає сенсу показувати тут усі знімки екрана, тому що їх може бути багато. Розгляньмо лише деякі параметри.

Передусім необхідно надати інформацію про ціни та доступ. Можна опублікувати програму як безкоштовну або обрати для неї ціну. У Магазині є безліч безкоштовних програм. Наприклад, ніхто не платитиме за ту, що дає змогу замовити таксі, але завдяки такій програмі служба таксі може збільшити кількість клієнтів. Деякі програми можуть містити рекламу і в такий спосіб приносити прибуток.

Визначивши ціну, можете додати ознайомлювальну версію своєї програми. Звичайно, код має передбачати підтримку такої версії. Про це мова піде в наступному розділі, присвяченому Магазину.

Ви можете опублікувати програму на 242 ринках. За замовчуванням вибрано всі ринки, але за посиланням **Show Options** можна вибрати пріоритетні, а для деяких із них указати іншу ціну. Звичайно, якщо ви публікуєте програму, приміром, для канадського ринку, недоцільно робити її доступною для всіх ринків.

Далі поговорімо про продаж програм – питання, що цікавить розробників найбільше:

### Sale pricing

Set limited-time price reductions for your app. Learn more

Note: Sales will only be visible to customers on Windows 10 devices.

New sale

Sale price tier	Start	End	Markets
Start date*	Start hour*	End date*	End hour*
8/27/2015	18:00 UTC	8/27/2015	18:00 UTC

Sale price tier\*

Pick a price tier

Show custom market pricing options

Done Delete

Можна налаштовувати параметри продажу програми. Це допоможе розрекламувати її серед ширшого кола клієнтів. Інформацію про новий розпродаж можна викласти будь-коли, при цьому немає потреби повторно надсилати всі пакети.

У розділі «Розповсюдження і видимість» можна вибрати варіанти подальших дій: провести бета-тестування або поширити програму серед колег.

### Distribution and visibility

Hide options

Anyone can find your app in the Store

Hide this app in the Store. Customers with a direct link to the app's listing can still download it, except on Windows 8 and Windows 8.1. Learn more

Hide this app in the Store. Only customers with the email addresses you enter below can download it, via a direct link on Windows Phone 8.1 and earlier. Learn more

Hide this app and stop selling Learn more

Є можливість опублікувати програму для загального користування, для обраного списку користувачів або приховати її від пошуку в Магазині та поширювати, надаючи пряме посилання.

На наступному етапі слід вирішити питання доступу до програми. Ви вже знаєте, що Windows 10 підтримують найрізноманітніші пристрої, зокрема планшети, ноутбуки, ПК, телефони, Xbox тощо. Наразі доступ до Магазину Windows 10 можливий лише з ПК і мобільних пристройів. Проте щойно корпорація Microsoft відкриє Магазин Xbox Windows 10 чи HoloLens, вашу програму можна буде завантажити й на нові пристрої.

Windows 10 device families Hide options

Note that customers using a given Windows 10 device can see your app's Store listing only if you have packages which are able to run on that type of device. We recommend leaving all boxes checked unless you have a specific reason to exclude a certain Windows 10 device family.

If you remove a previously-used device family here, customers who already have your app will still be able to use it, and will get any updates you submit. However, no new customers will be able to download it on that type of Windows 10 device. [Learn more](#)

Desktop

Mobile

Let Microsoft decide whether to make this app available to any future device families

Require your permission before making this app available to any future device families

У цьому ж вікні вам пропонують вирішити, включати програму до нових Магазинів автоматично, чи ви зробите це самі згодом.

Наступний параметр – вибір способу інсталяції програми: за корпоративними онлайновими ліцензіями чи в автономному режимі. Цьому типу інсталяції найбільше віддають перевагу організації, що планують купувати пакет різних ліцензій або мають багато комп'ютерів без доступу до Інтернету.

Тепер необхідно вказати дату публікації. Програму може бути опубліковано автоматично, відразу по завершенню сертифікації, у зазначений день або вручну.

На наступному етапі потрібно вказати деякі властивості програми. Спершу виберіть категорію, у якій програму буде розміщено в Магазині.

Окремо слід сказати про рейтинги в Магазині. Детально ознайомтеся з усіма можливими рейтингами. Майте на увазі, що для деяких програм, особливо для ігор, може знадобитися сертифікат з інформацією про рейтинг. Далі перейдіть до списку всіх наявних сертифікатів і списку країн, де вони підтримуються.

Тепер укажіть, які засоби потрібні для користування вашою програмою:

### Hardware preferences

Indicate which hardware features are required in order for your app to run properly. Customers on hardware that doesn't meet your app's requirements will see a warning before they download your app. Learn more

- |                                       |                                   |                                       |                                    |
|---------------------------------------|-----------------------------------|---------------------------------------|------------------------------------|
| <input type="checkbox"/> Touch screen | <input type="checkbox"/> Keyboard | <input type="checkbox"/> Mouse        | <input type="checkbox"/> Camera    |
| <input type="checkbox"/> NFC HCE      | <input type="checkbox"/> NFC      | <input type="checkbox"/> Bluetooth LE | <input type="checkbox"/> Telephony |

Наприклад, є ігри, що не були опубліковані раніше, оскільки для них потрібний сенсорний екран. Користувачі Windows 10, перш ніж завантажити або придбати програму, отримуватимуть застереження, якщо їхні пристрої не мають відповідних функцій.

В останньому розділі на цьому кроці зібрано все, що не було включено до інших розділів: можливість інсталювати програму на SD-карту, використовувати пла-тіжні системи сторонніх розробників, можливості доступу і резервного копію-вання до OneDrive.

Далі необхідно завантажити пакети для всіх підтримуваних платформ. Це най-простіший етап. Ваші пакети буде проаналізовано, і залежно від кількості підтри-муваних мов вам потрібно буде надати опис програми для кожної з них.

Після цього треба буде завантажити набір зображень, надати опис, повідомити електронну пошту для підтримки тощо. Зважайте: програма, що не містить промозображенень, не відображатиметься на головній сторінці Магазину, навіть якщо вона вельми популярна та якісна.

На останньому етапі можна додати примітки для тестерів. Це все, що допоможе успішно пройти сертифікацію, зокрема посилання на угоди з іншими компані-ями, використання логотипів, дані для тестового входу і т. ін.

По завершенню всіх етапів натисніть кнопку **Надіслати**. Надані відомості буде надіслано корпорації Microsoft.

Сьогодні процес сертифікації не забирає багато часу. Зазвичай результат відо-мий уже за кілька годин. Зауважимо: навіть коли програма пройшла сертифи-кацію, вона не завжди з'являється у Магазині відразу. Іноді серверам Microsoft необхідний певний час, щоб завершити синхронізацію процесів.

Отже, тепер ви можете опублікувати свою програму.



# Про авторів



## **Сергій Байдачний**

працює у сфері розробки програмного забезпечення більше 14 років і останні 9 років є співробітником компанії Microsoft на позиції технічного євангеліста. В активі Сергія декілька книг, присвячених розробці програм на платформі .NET.

Блог Сергія доступний за адресою  
<http://en.baydachnyy.com>



## **Маргарита Остапчук,**

фахівець з інформаційних технологій у відділі стратегічних досліджень компанії Microsoft Ukraine. Основними сферами спеціалізації є розробка для платформи Windows та Microsoft Azure.

Блог Маргарити доступний за адресою  
<http://in4margaret.azurewebsites.net>



# Зміст

Розділ 13.

<b>Знайомство із шаблоном проектування MVVM .....</b>	<b>1</b>
---	----------

Розділ 14.

<b>Робота з файлами і налаштуваннями .....</b>	<b>9</b>
--	----------

Робота з папками та файлами .....	10
Робота з налаштуваннями та тимчасовими даними .....	13
Вікно вибору файлів .....	16
Відомі папки .....	18
Асоціювання з певним типом файлів .....	19

Розділ 15.

<b>Обмін даними між програмами .....</b>	<b>21</b>
--	-----------

Як реалізувати функціонал перетягування .....	22
Буфер обміну .....	25
Обмін даними .....	27
Як запустити зовнішні програми за протоколом URL .....	30
Кеш видавця .....	37

Розділ 16.

<b>Служби програм і фонові завдання.....</b>	<b>39</b>
--	-----------

Фонові завдання .....	40
Служби програм .....	46

Розділ 17.

**Робота в мережі..... 53**

Інформація про мережу.....	54
Робота з даними .....	56
Як працювати з RSS feeds .....	57
Використання класу WebAuthenticationBroker .....	58

Розділ 18.

**Аудіо і відео .....** 63

Елементи керування медіаданими .....	64
Media casting .....	77
Media Editing and Transcoding .....	81

Розділ 19.

**API камери .....** 83

Використання діалогового вікна CameraCaptureUI .....	84
Media Capture API.....	90

Розділ 20.

**Розпізнавання мови і Cortana .....** 95

Озвучення тексту.....	96
Розпізнавання мовлення .....	99
Cortana .....	102

Розділ 21.

**Карти .....** 111

Розділ 22.

<b>Використання пера.....</b>	<b>121</b>
-------------------------------	------------

Розділ 23.

<b>Інтерфейс API для датчиків .....</b>	<b>129</b>
---	------------

Розділ 24.

<b>Розширення платформи.....</b>	<b>135</b>
----------------------------------	------------

Розділ 25.

<b>Як публікувати веб-програми в Магазині.....</b>	<b>141</b>
--	------------

WebView .....	142
---------------	-----

Проект Westminster .....	151
--------------------------	-----

Розділ 26.

<b>Як заробити гроші в Магазині.....</b>	<b>155</b>
--	------------

Платні програми .....	156
-----------------------	-----

Безкоштовні програми.....	162
---------------------------	-----

Розділ 27.

<b>Служби Windows Notification Services.....</b>	<b>169</b>
--	------------

Розділ 28.

**Надсилання повідомлень за допомогою служб  
Microsoft Azure ..... 174**

Вихідний сценарій розробки .....	175
Основні відомості про Notification Hub.....	177
Створення простого сценарію з Notification Hub.....	179
Mobile Service API та Notification Hub API .....	186
Служба сповіщень Windows Notification Service і середовище виконання Windows Runtime .....	190
Серверний код .NET .....	201
Інші можливості Mobile Services.....	206

Розділ 29.

**Як створити власні елементи керування ..... 212**

Шаблони.....	213
Клас UserControl.....	217
Шаблон Templated control.....	218

Розділ 30.

**Тестування і налагодження ..... 222**

Емулятори та симулятори .....	223
Live Visual Tree у Visual Studio .....	227
Засоби профілювання і налагодження у Visual Studio 2015 .....	229
Інструменти XAML у Visual Studio 2015 .....	233

Розділ 31.

**Win2D: Як використовувати графіку,  
не знаючи DirectX..... 236**

Розділ 32.

**API Composition ..... 250**

---

Розділ 33.**Як використовувати Blend ..... 260**

Огляд Blend .....	261
Як створювати анімації в Blend .....	262
Стани й тригери станів у Blend.....	264

## Розділ 34.

**Інтернет речей..... 270**

Огляд мікроконтролерів .....	271
Windows 10 IoT Core .....	281
Розширення IoT.....	282
Аналогові сигнали і PWM на Raspberry.....	289

## Розділ 35.

**Як розробляти ігри для Windows 10 в Unity3D..... 294**

Інструменти Visual Studio для Unity.....	295
Публікація гри Unity в Магазині Windows.....	303
Як створювати плагіни для Unity .....	306



Розділ 13.

## **Знайомство із шаблоном проектування MVVM**

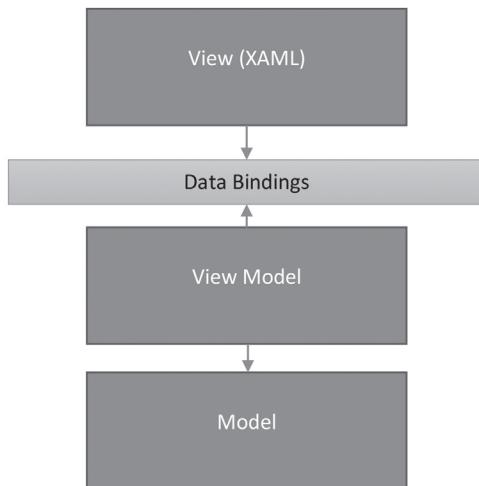
У книзі ми не розглядаємо складних проектів, але в практичному програмуванні, найімовірніше, вам доведеться працювати з різноманітними формами, складною бізнес-логікою, великою кількістю обробників подій тощо. Тож украй важливо вибрати правильний підхід до створення програми, починаючи із шаблону проектування. Для вибору шаблону слід врахувати такі аспекти.

- Співпраця з дизайнером. У цій книзі ми не будемо зупинятися на принципах дизайну програм для Windows 10, однак їх чимало. Ми знаємо, що розробники не хотіли б витрачати багато часу на перевірку зручності та пошук найкращого розташування для конкретної кнопки, хоча це дуже важливо для сенсорних інтерфейсів. Тому варто постійно взаємодіяти з дизайнером, але робити це потрібно таким чином, щоб у вас обох була змога працювати над проектом.
- Створення юніт-тестів. Багато розробників відчувають потребу в написанні юніт-тестів. Але це непросте завдання, якщо бізнес-логіка змішується з кодом інтерфейсу / XAML-кодом. Отже, потрібно знайти спосіб їх розмежувати.
- Повторне використання логіки програми в різних формах і навіть проектах.

Для досягнення цих цілей радимо використовувати шаблон проектування MVVM (Model – View – ViewModel). Цей шаблон застосовується з часу появи Windows Presentation Foundation і Silverlight. Він дає змогу створювати чіткий та надійний код програми.

Ідея дуже проста. Щоб застосувати MVVM, необхідно розділити код на три шари, які описано нижче.

- Модель (Model) – містить дані «як є». Зазвичай створюються класи, які відображають таблиці бази даних або файлову структуру XML/JSON, але в кожному разі там містяться тільки дані.
- Подання (View) – це просто файл XAML. Зазвичай подання не потребує створення жодного допоміжного коду. Цей підхід дає змогу працювати з дизайнером, оскільки вам не доведеться редагувати XAML-код.
- Подання-модель (ViewModel) – вам потрібно буде багато працювати з ViewModel, щоб підготувати дані для подання. Код у ViewModel повинен знати все і про модель, і про подання. Тож якщо необхідно перетворити певні дані, об'єднати кілька полів, виконати обчислення, обробити події, пов'язані з користувачким інтерфейсом тощо – усе це потрібно робити всередині ViewModel.



Оскільки ViewModel має всю інформацію щодо моделі та може створювати її ініціалізувати власне модель, між ViewModel і моделлю не потрібні жодні додаткові прошарки. Але за потреби ви можете створити додаткові класи, які допоможуть під час ініціалізації та оновлення даних.

Щодо подання жодної конкретної інформації не потрібно. ViewModel призначено для підготовки даних для подання, але для їх ініціалізації відомості про елементи керування та їхні імена не знадобляться. Натомість подання та ViewModel для обміну даними використовують зв'язування. Отже, потрібно створити тільки об'єкт ViewModel та ініціалізувати об'єкти зв'язування.

Давайте створимо просту програму, що дає змогу працювати з новинами. Почати потрібно з моделі та створити клас, який міститиме вихідні дані з бази даних або будь-якого іншого джерела. Завжди можна починати роботу з класом даних, тому що зазвичай їх структура відома й чітко визначена.

```

public class NewsItem
{
    public DateTime NewsDate { get; set; }

    public string NewsTitle { get; set; }

    public string NewsDescription { get; set; }

    public string NewsBody { get; set; }
}
  
```

Цей клас стандартний, він містить лише властивості зі сховища даних.

Щоб створити клас ViewModel, потрібно знати, які поля будуть представлені в поданні. На даному етапі вам знадобиться лише загальна інформація щодо подання. Наразі не важливо, як дизайнер розмістить поля або інші елементи.

У нашому випадку потрібно показати дату й заголовок для новин, а також задати відображення позначки «Сьогодні» для свіжих новин. Для цього слід реалізувати такий шаблон ViewModel:

```
public class NewsItemViewModel
{
    NewsItem Item { get; set; }

    public NewsItemViewModel()
    {
        Item = Helper.GetNewsItem();
    }

    public string Date
    {
        get
        {
            if (DateTime.Today.Day == Item.NewsDate.Day)
                return "Today";
            return String.Format($" {Item.NewsDate:dd.MM.yyyy}");
        }
    }

    public string Title
    {
        get
        {
            return Item.NewsTitle;
        }
        set
        {
            Item.NewsTitle = value;
        }
    }
}
```

```

}

public class Helper
{
    public static NewsItem GetNewsItem()
    {
        return new NewsItem()
        {
            NewsDate = DateTime.Now.AddDays(-2),
            NewsTitle = "News Item 1",
            NewsBody = "News Body",
            NewsDescription = "News Description"
        };
    }
}

```

Як бачимо, у цьому коді відформатовано дату й задано обтікання для полів заголовків, які використовуватимуться в поданні. Крім того, створено клас `Helper`, який ініціалізує елементи (самі дані ви можете читувати з файлу, бази даних або іншого джерела).

Цей код не є ідеальним, адже для подання даних планується використовувати зв'язування, але воно не працюватиме, якщо ми змінимо дані у `ViewModel` або навіть якщо завантаження даних триватиме певний час. Тому необхідно реалізувати інтерфейс **INotifyPropertyChanged**, на чому ми зупинялися в розділі 8:

```

public class NewsItemViewModel : INotifyPropertyChanged
{
    NewsItem Item { get; set; }

    public NewsItemViewModel()
    {
        Item = Helper.GetNewsItem();
    }

    public string Date
    {
        get
        {
            if (DateTime.Today.Day == Item.NewsDate.Day)
                return "Today";
            return String.Format

```

```
        ($" {Item.NewsDate:dd.MM.yyyy} ") ;
    }
    private set
    {
        Item.NewsDate = Convert.ToDateTime(value);
        RaisePropertyChanged(Date);
    }
}

public string Title
{
    get
    {
        return Item.NewsTitle;
    }
    set
    {
        Item.NewsTitle = value;
        RaisePropertyChanged>Title);
    }
}

public event PropertyChangedEventHandler PropertyChanged;

private void RaisePropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
        handler(this, new PropertyChangedEventArgs
        (propertyName));
    }
}
}
```

Звичайно, якщо потрібно реалізувати певні обробники подій, їх також слід включити у ViewModel. Якщо у вас є досвід роботи з Windows 8.1 або Windows Phone, ви знаєте, що для реалізації обробників подій слід використовувати інтерфейс **ICommand**. Але зараз він не потрібен, тому що **x:Bind** також підтримує зв'язування для обробників подій.

Нарешті ми можемо створити просте подання:

```

<Page
    x:Class="MVVMProject.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MVVMProject"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/
        markup-compatibility/2006"
    xmlns:code="using:MVVMProject.Code"
    mc:Ignorable="d">
    <Page.Resources>
        <code:NewsItemViewModel x:Name="viewModel">
        </code:NewsItemViewModel>
    </Page.Resources>

    <Grid Background="{ThemeResource
        ApplicationPageBackgroundThemeBrush}">
        <StackPanel Orientation="Vertical">
            <TextBlock Text="{x:Bind viewModel.Title}">
            </TextBlock>
            <TextBlock Text="{x:Bind viewModel.Date}">
            </TextBlock>
        </StackPanel>
    </Grid>
</Page>

```

Як бачимо, безпосередньо в документі XAML створено об'єкт **NewsItemViewModel** без жодного допоміжного коду. Але інколи, особливо якщо завантаження даних триває досить довго й використовується асинхронний підхід, можна ініціалізувати зв'язування також у коді.

Ви можете просто використовувати ті самі класи, якщо потрібно створити клас ViewModel для подання списку елементів. У такому разі можна використовувати **ObservableCollection** і наявний клас **NewsItemViewModel**:

```

public class NewsListViewModel
{
    public ObservableCollection<NewsItemViewModel> Items =
        new ObservableCollection<NewsItemViewModel>();
    . . .
}

```



Розділ 14.

## **Робота з файлами і налаштуваннями**

## Робота з папками та файлами

У сучасній моделі програм Windows не передбачено використання класів Win32 API чи навіть .NET Framework для отримання доступу до всіх файлів і папок на диску. Проте Universal Windows Platform підтримує кілька сценаріїв доступу до файлів і папок у безпечному режимі. У цьому розділі ми розглянемо всі можливі сценарії роботи з файлами, зокрема з відомими та тимчасовими папками, елементами керування, що використовуються для відкривання файлів і папок, тощо.

Зрозуміло, перш ніж перейти до сценаріїв, варто ознайомитися з базовим набором класів. У цьому розділі ми будемо використовувати три простори імен. Вони перелічені нижче.

- **Windows.Storage** містить класи для налаштувань програм, файлів і папок.
- **Windows.Storage.Streams** надає класи для читання й записування даних у файли.
- **Windows.Storage.Pickers** містить кілька елементів керування, за допомогою яких користувач може вибирати файли та папки, а також шлях для їх збереження.

Почнемо огляд із простору імен **Windows.Storage**. У ньому є багато класів, найважливіші з яких – **StorageFolder** і **StorageFile**. Вони застосовуються скрізь, де потрібно працювати з папками та файлами.

Роботу, певно, краще починати з файлів, що входять до пакета програми. Можна відкрити потрібний файл за іменем або скористатися переліком усіх файлів, отримавши доступ до папки, де розгорнуто програму.

Реалізуємо код із переліком усіх папок у папці програми. Для доступу до папки можна використати клас **Windows.ApplicationModel.Package**.

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    var package= Package.Current;
    var folder = package.InstalledLocation;
    var files = await folder.GetFoldersAsync();
    listView.ItemsSource = files;
}
```

У першому рядку ми отримуємо доступ до об'єкта з класу **Package**. За допомогою цього об'єкта можна отримати відомості про дату інсталяції, розташування програми, відображувану назву тощо. Зараз нам потрібна лише властивість **InstalledLocation**, що повертає об'єкт **StorageFolder**. Тепер можна скористатися

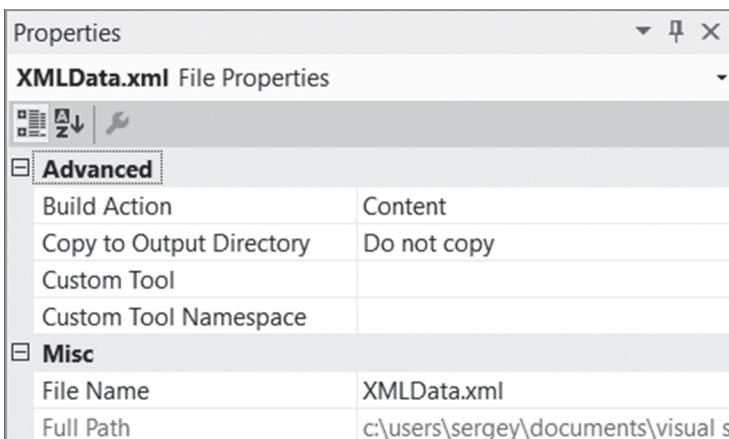
об'єктом папки, щоб отримати доступ до файлів, підпапок тощо. Змінити папку **InstalledLocation** неможливо, проте в ній можна відкривати будь-які файли та папки. Нижче наведено інтерфейсну частину програми:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <ListView Name="listView" Margin="50">
        <ListView.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding DisplayName}">
                    </TextBlock>
                </StackPanel>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

Тепер навчимося отримувати доступ до файлу з пакета за іменем. Скажімо, в папці міститься документ XML із даними, які необхідно прочитати в програмі. Можна створити простий XML-файл на кшталт цього:

```
<Items>
    <Item>Item 1</Item>
    <Item>Item 2</Item>
    <Item>Item 3</Item>
</Items>
```

Пам'ятайте, що файли мають бути позначені в проекті як файли контенту. Відкрийте вікно властивостей файлу у Visual Studio і переконайтесь, що властивості **Build Action** надано значення **Content**:



Замість попереднього коду можна реалізувати такий:

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
    StorageFile file =
        await StorageFile.GetFileFromApplicationUriAsync(
            new Uri("ms-appx:///XMLData.xml"));
    var xml = await XmlDocument.LoadFromFileAsync(file);
    listView.ItemsSource = xml.ChildNodes[0].ChildNodes;
}
```

Тут ми використовуємо метод **GetFileFromApplicationAsync** і для створення **URI** застосували префікс **ms-appx://**. Завдяки цьому префіксу можна отримати доступ до будь-якого файлу в пакеті.

Звичайно, доведеться трохи змінити код інтерфейсу:

```
<ListView Name="listView" Margin="50">
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding InnerText}">
                </TextBlock>
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

У попередньому прикладі ми створювали XML-документ із використанням об'єкта, що реалізує інтерфейс **IStorageFile** (нагадую, що це був об'єкт **StorageFile**). Однак клас **StorageFile** забезпечує можливість зчитувати дані просто з файлу. Наприклад, можна скористатися методом **OpenReadAsync**, щоб відкрити потік і прочитати дані з нього. У нижченнаведеному коді створимо файл і, використовуючи потоки, запишемо в нього «Hello»:

```
public async void WriteFile()
{
    StorageFolder current = ApplicationData.Current.LocalFolder;
    StorageFile file = await current.CreateFileAsync(
        "hello.txt", CreationCollisionOption.ReplaceExisting);
    IRandomAccessStream writeStream =
        await file.OpenAsync(FileAccessMode.ReadWrite);
    IOutputStream outputStream =
        writeStream.GetOutputStreamAt(0);
    DataWriter dataWriter = new DataWriter(outputStream);
    dataWriter.WriteString("Hello");
    await dataWriter.StoreAsync();
    await outputStream.FlushAsync();
}
```

У цьому коді ми використали інтерфейс **IRandomAccessStream**. Це базовий інтерфейс для всіх операцій з контентом у файлах, і за його допомогою легко отримати вхідний чи вихідний потік.

Зверніть особливу увагу на цей рядок:

```
StorageFolder current = ApplicationData.Current.LocalFolder;
```

У цьому використано клас **ApplicationData**, що дає змогу отримати посилання на певну папку через властивість **LocalFolder**. У цій пов'язаній із програмою папці можна зберігати локальні файли. З'ясуємо, які папки взагалі доступні для будь-якої програми.

## Робота з налаштуваннями та тимчасовими даними

Платформа Universal Windows Platform не дає змоги отримувати доступ до всіх папок і файлів на диску. Однак програмі потрібен простір для зберігання власних файлів і даних. Особливо, якщо вона, наприклад, підтримує кешування даних із

сервера або збереження настроєних користувачем параметрів. Із цією метою платформа UWP підтримує три типи сховищ.

- **Local storage.** Користуючись локальним сховищем, ви розміщуєте файли у спеціальній папці, пов'язаній із програмою. Таке сховище схоже на особистий диск для програми.
- **Roaming storage.** Це онлайнове сховище подібне до попереднього із тією відмінністю, що всі дані зберігаються в службі Microsoft Cloud. У цьому сховищі діють обмеження на обсяг даних, але файли будуть доступні для вашої програми на будь-якому пристрої.
- **Temporary storage.** Це своєрідна тимчасова папка, з якої система може будь-якої миті видалити дані. Використовуйте її, якщо вам потрібен простір для роботи та не хочеться перейматися його очищеннем.

Розглянемо кожне з цих сховищ.

## Локальне сховище

Для локального зберігання даних можна використовувати файли та папки або звичайний словник. Для простих налаштувань цілком досить словника, а для складніших краще підійдуть файли.

Вище було показано, як отримати доступ до локальної папки і створити файл.

Давайте подивимося, як читати дані з файлу:

```
public async void ReadFile()
{
    StorageFolder current = ApplicationData.Current.LocalFolder;
    StorageFile sampleFile =
        await current.GetFileAsync("hello.txt");
    IRandomAccessStream readStream =
        await sampleFile.OpenAsync(FileAccessMode.Read);
    IInputStream inputStream = readStream.GetInputStreamAt(0);
    DataReader dataReader = new DataReader(inputStream);
    string myString = dataReader.ReadString((uint)readStream.
        Size);
}
```

Доступ до наявного файлу можна також отримати за допомогою URI. У нашому прикладі префіксом цього URI має бути **ms-appdata:///local/**.

Нижче перелічено класи, які використовуються для роботи зі словниками.

- **ApplicationDataContainer** – це контейнер для зберігання простих типів даних і об'єктів класу **ApplicationDataCompositeValue**. Насправді цей контейнер апріорі є у кожної програми, але ви за необхідності можете створювати додаткові.
- **ApplicationDataCompositeValue** допомагає формувати складені типи даних із простих типів, а також полегшує групування даних.

Ці два класи містяться в просторі імен **Windows.Storage** і дуже зручні у використанні. Наприклад, для доступу до стандартного контейнера використовується такий код:

```
ApplicationDataContainer cur =
ApplicationData.Current.LocalSettings;
```

Якщо в стандартному контейнері потрібно створити підконтейнер, додайте до проекту такий код:

```
ApplicationDataContainer cur =
ApplicationData.Current.LocalSettings;
ApplicationDataContainer named =
cur.CreateContainer("myContainer",
ApplicationDataCreateDisposition.Always);
```

Для роботи з контейнерами можна просто використовувати колекцію **Values** та індексатор, щоб отримувати доступ до даних за значенням ключа:

```
cur.Values["myValue"] = 5;
cur.Containers["myContainer"].Values["mySecondValue"] = "Hello";
```

А для створення складеного значення використовуйте як шаблон такий код:

```
ApplicationDataCompositeValue composite =
new ApplicationDataCompositeValue();
composite["firstVal"] = 1;
composite["secondVal"] = "Hello";
cur.Values["compValue"] = composite;
```

## Онлайнове сховище

Якщо програмі бажано отримувати доступ до налаштувань із різних пристройів, використовуйте онлайнове сховище. З ним можна працювати так само, як і з локальним. Наприклад, доступ до словника та папки в хмарі можна отримати за допомогою таких рядків:

```
ApplicationDataContainer current =  
    ApplicationData.Current.RoamingSettings;  
StorageFolder file = ApplicationData.Current.RoamingFolder;
```

Звичайно, онлайнове сховище невигідно використовувати для значних обсягів інформації. Пам'ятайте про інтернет-трафік! Дізнатися про доступний простір допоможуть такі властивості класу **ApplicationData**:

- **RoamingStorageQuota** показує обсяг простору, який доступний для програми;
- **RoamingStorageUsage** показує обсяг простору, який використовується програмою.

Зазвичай дані в хмарі доступні для перегляду лише після завантаження програми, однак вказаний далі код дає змогу відстежувати зміни під час виконання.

```
void InitHandlers()  
{  
    ApplicationData.Current.DataChanged += DataChangeHandler;  
}  
void DataChangeHandler(ApplicationData appData, object o)  
{  
    //update  
}
```

## Тимчасове сховище

Для роботи з тимчасовим сховищем застосуйте такий код:

```
StorageFolder folder = ApplicationData.Current.TemporaryFolder;
```

Ви отримаєте посилання на папку **StorageFolder**, яку можете використовувати як завгодно.

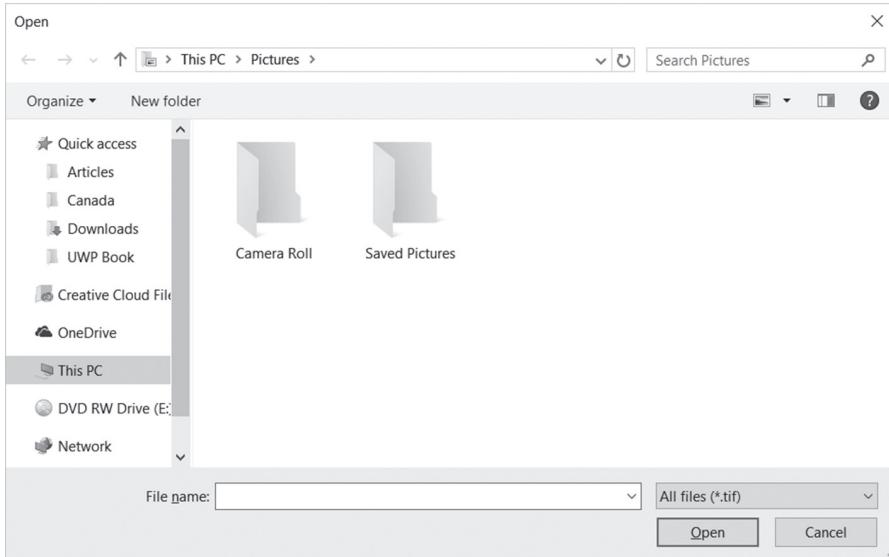
## Вікно вибору файлів

Сучасні програми не бачать на диску жодних файлів, окрім власних, однак натомість можуть запитувати користувача про доступ до потрібного файлу чи папки. Universal Windows Platform надає цю можливість за допомогою кількох класів, що активують стандартні діалоги для вибору файлів і папок та збереження файлів. Розглянемо такий метод:

```
public async void SelectFile()
{
    FileOpenPicker openPicker = new FileOpenPicker();
    openPicker.ViewMode = PickerViewMode.Thumbnail;
    openPicker.SuggestedStartLocation =
        PickerLocationId.PicturesLibrary;
    openPicker.FileTypeFilter.Add(".tif");
    StorageFile file = await openPicker.PickSingleFileAsync();
}
```

За допомогою цього методу можна активувати вікно для вибору одного файла на диску. Для цього було застосовано клас **FileOpenPicker** та ще кілька властивостей. Наприклад, ми вирішили зробити стандартним розширення **TIF** і пересправлювати засіб вибору файлів до бібліотеки зображень.

Якщо запустити цей код на комп'ютері, відобразиться таке вікно:



Якщо користувач має вибрати кілька файлів, реалізуйте той самий підхід, але вже з методом **PickMultipleFilesAsync**.

Для вибору папки або шляху збереження можна використати класи **FolderPicker** і **SaveFilePicker** і той самий підхід, що й для **FileOpenPicker**.

## Відомі папки

Тепер ми знаємо, що програма може працювати з файлами та папками, якщо вони зберігаються в спеціальних папках або якщо користувач вказав до них шлях у вікні вибору файлів. У цьому підрозділі ми розглянемо ще один спосіб: доступ до файлів зі стандартних бібліотек на кшталт Зображення, Музика та Відео.

Користувач, певна річ, має знати, що програма використовуватиме вміст його бібліотек, однак запитувати дозволи та стежити за наданням доступу не потрібно. Натомість у Windows 10 дозвіл можна запитати непрямо за допомогою файлу маніфесту. Просто оголосіть відповідні можливості програми в маніфесті, щоб користувач міг переглянути всі потрібні дозволи, перш ніж її інсталює. Якщо користувача не влаштовує, що програма матиме доступ, скажімо, до бібліотеки Зображення, він просто скасуює інсталяцію.

Щоб запитати доступ до бібліотек користувача, слід додати одну або кілька з таких можливостей: **Pictures Library**, **Videos Library**, **Music Library**. Наприклад, якщо програмі потрібна тільки бібліотека Зображення, додайте можливість **Pictures Library**.

The screenshot shows the Windows App Manifest configuration page. The top navigation bar includes tabs for Application, Visual Assets, Capabilities (which is currently selected and highlighted in grey), Declarations, Content URIs, and Packaging. Below the tabs, there is a note: "Use this page to specify system features or devices that your app can use." The main content area is divided into two columns: "Capabilities:" and "Description:". The "Capabilities:" column lists various system features with checkboxes. The "Pictures Library" checkbox is checked and highlighted with a grey border. Other checked checkboxes include "Internet (Client)" and "Music Library". The "Description:" column provides a brief explanation for each capability, such as "Provides the capability to add, change, or delete files in the Pictures Library for the local PC and HomeGroup PCs." and a link to "More information".

Capabilities:	Description:
<input type="checkbox"/> All Joyn	Provides the capability to add, change, or delete files in the Pictures Library for the local PC and HomeGroup PCs. <a href="#">More information</a>
<input type="checkbox"/> Blocked Chat Messages	
<input type="checkbox"/> Chat Message Access	
<input type="checkbox"/> Code Generation	
<input type="checkbox"/> Enterprise Authentication	
<input checked="" type="checkbox"/> Internet (Client)	
<input type="checkbox"/> Internet (Client & Server)	
<input type="checkbox"/> Location	
<input type="checkbox"/> Microphone	
<input checked="" type="checkbox"/> Music Library	
<input type="checkbox"/> Objects 3D	
<input type="checkbox"/> Phone Call	
<input checked="" type="checkbox"/> Pictures Library	
<input type="checkbox"/> Private Networks (Client & Server)	
<input type="checkbox"/> Proximity	
<input type="checkbox"/> Removable Storage	
<input type="checkbox"/> Shared User Certificates	
<input type="checkbox"/> User Account Information	
<input checked="" type="checkbox"/> Videos Library	
<input type="checkbox"/> VOIP Calling	
<input type="checkbox"/> Webcam	

Universal Windows Platform також підтримує доступ до бібліотеки документів, але для публікації такої програми потрібно отримати спеціальні дозволи. Власне кажучи, така функція реалізується лише в корпоративних програмах.

Щойно ви запитали дозволи в маніфесті, можна починати роботу з бібліотеками. За допомогою вказаного нижче коду запитується доступ до бібліотеки Зображення та папок у ній:

```
var library = await StorageLibrary.  
    GetLibraryAsync(KnownLibraryId.Pictures);  
var folders = library.Folders;
```

Зауважте, що в коді використовується клас **StorageLibrary** та ідентифікатор **KnownLibraryId** для повернення об'єкта **StorageLibrary**. Завдяки цьому класу можна отримувати посилання на всі папки в бібліотеці, навіть якщо вони пов'язані зі скрипцем OneDrive. Є ще клас **KnownFolders** із низкою властивостей, що повертає об'єкт **StorageFolder** безпосередньо для бібліотек Зображення, Відео та Музика (**Pictures**, **Videos**, **Music**). Але цей клас повертає посилання лише на локальні папки. Тому якщо потрібно отримати всі можливі папки, використовуйте клас **StorageLibrary**.

Звичайно, під час роботи з бібліотеками система Windows перевірятиме, чи використовуються файли лише відомих типів. Зокрема, в бібліотеці Зображення будуть доступні файли JPEG, JPG, GIF, BMP тощо.

## Асоціювання з певним типом файлів

Насамкінець розглянемо спосіб відкривання файлів за допомогою нашої програми просто в провіднику Windows Explorer. Для цього програму потрібно асоціювати з певними розширеннями файлів. Звичайно, вибирати слід лише підтримувані програмою розширення. Нижче ми асоціювали програму з розширенням **JPG** в редакторі маніфестів.

## Windows 10 для C# розробників

Use this page to add declarations and specify their properties.

### Available Declarations:

Select one...

### Supported Declarations:

File Type Associations	Remove

### Description:

Registers file type associations, such as jpeg, on behalf of the app.

Multiple instances of this declaration are allowed in each app.

[More information](#)

### Properties:

Display name:

Logo:

Info tip:

Name:

#### Edit flags

Open is safe

Always unsafe

#### Supported file types

At least one file type must be supported. Enter at least one file type; for example, \*.jpg".

Supported file type	Remove
Content type: <input type="text"/>	
File type: <input type="text" value=".jpg"/>	

[Add New](#)

Якщо запустити програму, користувачі бачитимуть її в контекстному меню для будь-якого JPG-файлу серед інших програм, асоційованих із форматом JPG.



Цього, звісно, недостатньо, адже потрібно ще реалізувати код для обробки файлу. Зазвичай доводиться перевизначати метод **OnActivated**, але для асоціювання з файлами клас **Application** підтримує метод **OnFileActivated**:

```
protected override void OnFileActivated
(FileActivatedEventArgs args)
{
    var files=args.Files;
    //create the frame and process the file
}
```

Пам'ятайте, що користувач може вибрати кілька файлів. Саме тому клас **FileActivatedEventArgs** містить не один об'єкт **StorageFile**, а цілий набір.

Розділ 15.

## **Обмін даними між програмами**

Програми Windows 10 працюють у власних «пісочницях», мають обмежений доступ до файлової системи та жодного зв'язку з іншими програмами. Проте Universal Windows Platform підтримує кілька способів обміну даними між програмами. У цьому розділі ми поговоримо про ці способи – від простого перетягування до кешу видавця.

## Як реалізувати функціонал перетягування

У Windows 10 уже можна здійснювати перетягування між частинами інтерфейсу користувача програми або використовувати зовнішні джерела чи цільові об'єкти, зокрема програми Win 32.

Розглянемо власне механізм перетягування. Для цього додамо зображення з пакету програми до головної сторінки.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Image Source="Assets\drone.jpg" Name="image"
        CanDrag="True" DragStarting="Image_DragStarting"
        Margin="100" VerticalAlignment="Top"
        HorizontalAlignment="Left"></Image>
</Grid>
```

За активацію перетягування відповідають два важливі атрибути: **CanDrag** і **DragStarting**. Атрибут **CanDrag** – це пропорець, який включає зазначену функцію для всіх елементів керування **UIElement**, а от атрибут **DragStarting** містить ім'я обробника події, завдяки якому можна визначити контент, що перетягуватиметься. У нашому випадку реалізуємо такий обробник:

```
private async void Image_DragStarting(UIElement sender,
    DragStartingEventArgs args)
{
    List<IStorageItem> files = new List<IStorageItem>();
    StorageFile file = await StorageFile.
        GetFileFromApplicationUriAsync
        (new Uri("ms-appx:///Assets/drone.jpg"));
    files.Add(file);

    args.DragUI.SetContentFromDataPackage();
    args.Data.RequestedOperation = DataPackageOperation.Copy;
    args.Data.SetStorageItems(files);
}
```

У цьому обробнику події клас **StorageFile** передає зображення як файл, а завдяки властивості **Data** параметра **DragStartingEventArgs** ми запаковуємо файл в об'єкт класу **DataPackage**. Це дуже популярний клас в Universal Windows Platform. Зазвичай його потрібно передати операційній системі, яка дає змогу вибирати цільову програму. Але у разі перетягування користувач сам вибирає цільовий об'єкт. Отже, нам достатньо підготувати **DataPackage**.

Крім того, ми використали дві важливі властивості: **DragUI** та **RequestedOperation**. Завдяки **RequestedOperation** можна вибрати ту чи іншу операцію з перетягування, при цьому користувач не матиме змоги вибирати що-небудь із системного меню – він зможе просто перетягувати об'єкт. **DragUI** дає можливість застосувати вміст, що відображатиметься під час перетягування. Якщо властивість **DragUI** не застосовано, користувач бачитиме таке саме зображення тих самих розмірів, як у вашій програмі. Перетягувати величезне зображення не дуже зручно, особливо якщо **RequestedOperation** не використовується, адже системне меню буде приховано за зображенням. Ось чому за допомогою **DragUI** можна присвоїти будь-який інший вміст або ж, скориставшись методом **SetContentFromDataPackage**, попросити API підготувати для вас відповідну піктограму залежно від вмісту **DataPackage**.

Запустіть програму і перетягніть зображення до провідника файлів. Зображення буде скопійовано до обраної папки.

Подивімось, як виконати протилежне завдання – скидання. Скажімо, потрібно отримати декілька зображень. Щоб всі вони відобразилися, скористаймося **ListView**.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
      AllowDrop="True" Drop="Grid_Drop"
      DragEnter="Grid_DragEnter">
    <ListView Margin="50" Name="listView">
        <ListView.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Image Source="{Binding Source}" Width="200" Margin="10"/>
                </Grid>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

Як бачимо, властивість **AllowDrop** використано для активації скидання, **DragEnter** – для налаштування дозволених операцій (копіювання) і **Drop** – отримання вмісту від **DataPackage** і відображення його за допомогою **ListView**.

Для створення джерел зображень реалізуємо клас **BitmapItem**:

```
class BitmapItem
{
    public ImageSource Source { get; set; }
}
```

На наступному етапі для того, щоб сповістити систему про підтримувані операції, реалізуємо обробник події **DragEnter**.

```
private void Grid_DragEnter(object sender, DragEventArgs e)
{
    e.AcceptedOperation = DataPackageOperation.Copy;
}
```

Зрештою, **DataPackageView** дає нам посилання на вміст. **DataPackageView** може містити будь-що, але ми хочемо працювати тільки з файлами. Тож викличмо **GetStorageItemsAsync**, щоб отримати посилання на файли, і за допомогою **BitmapImage** підготуймо файли зображень для об'єктів **Image**.

```
private async void Grid_Drop(object sender, DragEventArgs e)
{
    var files=await e.DataPackageView.GetStorageItemsAsync();
    List<BitmapItem> items = new List<BitmapItem>();
    foreach(StorageFile file in files)
    {
        try
        {
            BitmapImage bi = new BitmapImage();
            bi.SetSource(await file.OpenAsync
                (FileAccessMode.Read));
            items.Add(new BitmapItem() { Source = bi });
        }
        catch { }
    }
    listView.ItemsSource = items;
}
```

Наразі ми вирішили не перейматися перевіркою і використовуємо порожній блок захоплення на випадок, якщо користувач передає файл чи файли без зображення.

Отівсе. Бачите, як можна реалізувати функцію перетягування. Поекспериментуйте з різними типами вмісту або реалізуйте функціональність перетягування всередині тієї самої програми (перетягування контенту з однієї частини програми до іншої).

## Буфер обміну

У Windows 10 можна реалізовувати операції, пов'язані з буфером обміну, не тільки для ПК, а й для всіх пристройів Windows 10.

У попередньому розділі ми використовували клас **DataPackage** для того, щоб підготувати дані до надсилання до зовнішніх програм, і клас **DataPackageView** для отримання даних, що надійшли із зовнішнього джерела. До буфера обміну слід застосовувати той самий підхід, але замість обробників подій потрібно реалізувати меню контенту зі стандартними командами.

Подивімось, як реалізувати функцію вставки. Скористаймося тією самою програмою, що й для перетягування, бо тут знадобиться той самий код. Функцію вставки буде застосовано до зображень, які відображатимуться в **ListView**. Тож нам потрібно реалізувати простий **MenuFlyout**:

```
<ListView Margin="50" Name="listView" RightTapped="listView_RightTapped" IsRightTapEnabled="True">
    <ListView.Resources>
        <MenuFlyout x:Name="menuFlyout">
            <MenuFlyout.Items>
                <MenuFlyoutItem Name="pasteItem" Text="Paste" Click="MenuFlyoutItem_Click"></MenuFlyoutItem>
            </MenuFlyout.Items>
        </MenuFlyout>
    </ListView.Resources>
    <ListView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Image Source="{Binding Source}" Width="200" Margin="10"></Image>
            </Grid>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

```
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
```

Отже, ми оголосили **MenuFlyout** як ресурс для **ListView**. Клас **MenuFlyout** не дає змоги показати меню автоматично. Тому ми дозволили клацнути правою кнопкою миші **ListView** і реалізували обробник події **RightTapped** у такий спосіб:

```
private async void listView_RightTapped(object sender,
    RightTappedRoutedEventArgs e)
{
    var format = Clipboard.GetContent().Contains("FileDrop");
    pasteItem.IsEnabled = format;
    menuFlyout.ShowAt(listView, eGetPosition(null));
}
```

Щоб вдосконалити реакцію програми, перевірятимемо, чи доступні якісь файли, і вмикатимемо або вимикатимемо пункт меню.

Якщо користувач вибирає пункт меню **Paste** (Вставка), то для отримання всіх доступних файлів і підготовки їх до демонстрації в **ListView** потрібний клас **Clipboard**:

```
private async void MenuFlyoutItem_Click(object sender,
    RoutedEventArgs e)
{
    var files = await Clipboard.GetContent() .
        GetStorageItemsAsync();
    List<BitmapItem> items = new List<BitmapItem>();
    foreach (StorageFile file in files)
    {
        try
        {
            BitmapImage bi = new BitmapImage();
            bi.SetSource(await file.OpenAsync
                (FileAccessMode.Read));
            items.Add(new BitmapItem() { Source = bi });
        }
        catch { }
    }
    listView.ItemsSource = items;
}
```

Як бачимо, ми застосували той самий код, що і для перетягування, і змінили тільки перший рядок коду – використали клас **Clipboard** для отримання **DataPackageView**.

Отже, перетягування та роботу з буфером обміну краще реалізовувати разом, оскільки для них можна застосувати той самий підхід, і тепер ці функції є універсальними.

## Обмін даними

Припустимо, ваша програма має публікувати дані в соціальних мережах. Звичайно, ви можете інтегрувати її з Facebook чи Twitter, але ж хтозна, в якій соцмережі зареєстрований користувач вашої програми. До того ж інтеграція з усіма можливими мережами – завдання фактично нездійснене, адже для інтеграції хоча б із Facebook потрібний якийсь час.

Windows 10 надає привабливу можливість унезалежнити програму від способу використання її даних. Потрібно просто надати всі необхідні дані системі, і Windows сама перевірить, чи якісь інші програми можуть їх використовувати. Наприклад, якщо ваша програма надає посилання, Windows перевіряє всі зареєстровані в системі програми щодо того, чи здатні вони працювати з веб-посиланнями. Якщо такі програми доступні, Windows дає можливість користувачеві вибрати будь-яку з них, після чого запустить її та передасть до неї всі дані. Тож вам потрібно лише реалізувати код для передачі Windows даних, до яких слід надати спільній доступ.

За допомогою класу **DataTransferManager** можна обмінюватися даними з Windows. Зазвичай для отримання посилання на об'єкт цього класу слід використовувати метод **OnNavigatedTo**:

```
dataTransferManager = DataTransferManager.GetForCurrentView();
dataTransferManager.DataRequested += DataTransferManager_
    DataRequested;
```

Ці два рядки коду дають змогу отримати посилання і присвоїти події **DataRequested** обробник. Подія відбувається, коли програма починає обмін даними. В обробнику події можна готувати дані і надсилюти їх Windows.

```
private void DataTransferManager_DataRequested
(DataTransferManager sender, DataRequestedEventArgs args)
{
    var request = args.Request;
```

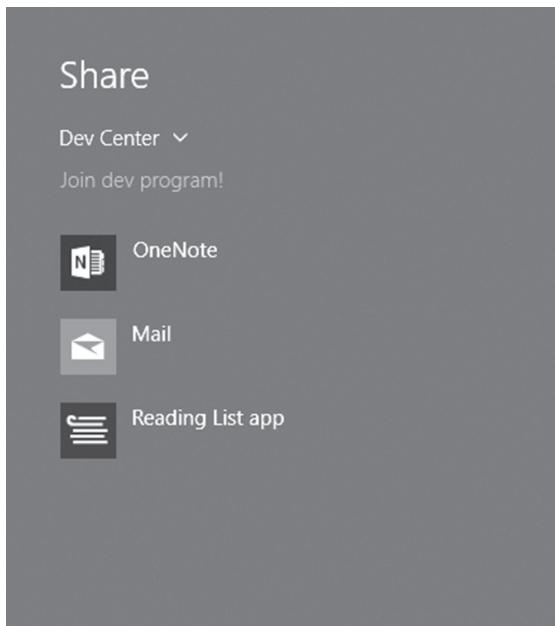
```
    request.Data.SetWebLink(new Uri("http://dev.windows.com")) ;  
    request.Data.Properties.Title = "Dev Center";  
    request.Data.Properties.Description = "Join dev program!";  
}
```

У наведеному вище коді використовується параметр **DataRequestEventArgs**, що дає можливість отримати доступ до об'єкта, який програма надсилатиме Windows. Завдяки цьому об'єкту можна надсилати фактично будь-що: зображення, посилання, HTML, тексти тощо. У своєму коді ми надішлемо посилання на Dev Center.

Після того як обробник події **DataRequested** реалізовано, функції надання спільногодоступу можна активувати де завгодно – наприклад, додати до інтерфейсу кнопку **Share**. В обробнику події натискання цієї кнопки потрібно реалізувати лише один рядок коду:

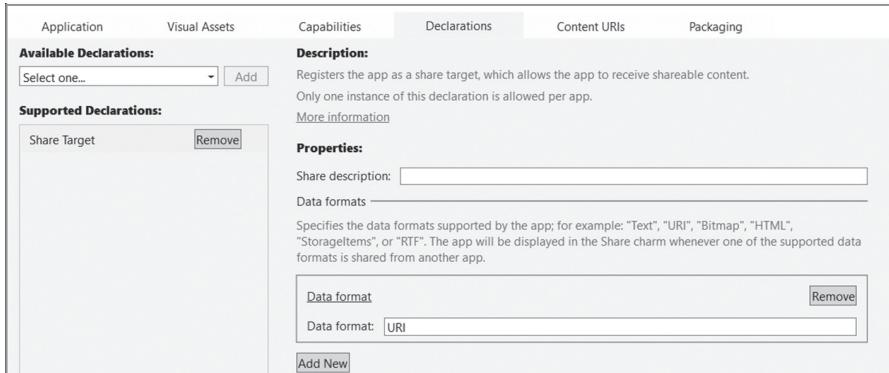
```
DataTransferManager.ShowShareUI();
```

Цей код даватиме Windows вказівку активувати стандартне вікно **Share**. Завдяки наведеному вище обробнику події за допомогою вікна обміну ми зможемо поділитися посиланням:



На нашому комп'ютері є лише три програми, що можуть обробляти посилання (цей ПК почали використовувати нещодавно).

Звичайно, ви можете приймати дані, доступ до яких надають інші програми. Для цього слід відредактувати маніфест програми й додати оголошення **Share Target**:



Потрібно надати Windows усі типи даних, які ви хочете використовувати. У маніфесті це виглядатиме так само, як і для розширення:

```
<uap:Extension Category="windows.shareTarget">
    <uap:ShareTarget>
        <uap:DataFormat>URI</uap:DataFormat>
    </uap:ShareTarget>
</uap:Extension>
```

Після оголошення розширення необхідно реалізувати код, який використовуватиметься для активації програми за допомогою контракту **Share**. Під час обміну даних не слід застосовувати метод **OnActivated**. Натомість клас Application містить метод **OnShareTargetActivated**, і потрібно просто перевизначити його:

```
protected override async void OnShareTargetActivated
(ShareTargetActivatedEventArgs args)
{
    ShareOperation shareOperation = args.ShareOperation;
    if (shareOperation.Data.Contains
        (StandardDataFormats.WebLink))
    {
        var link = await shareOperation.Data.GetWebLinkAsync();
```

```
    }  
}
```

## Як запустити зовнішні програми за протоколом URL

У попередньому розділі ми вже обговорювали зв'язки з типами файлів і те, як викликати зовнішні програми залежно від типу файлу. Ми використовували клас **Launcher**, і якщо перевірити всі методи всередині цього класу, можна знайти деякі, що не стосуються файлів, але дають змогу виконувати певні дії з URI. Наприклад, за потреби відкрити браузер і перейти до веб-сайту можна запустити такий код:

```
await Launcher.LaunchUriAsync(new Uri("http://dev.windows.com"));
```

У ньому було застосовано http-протокол. Проте клас **Launcher** підтримує універсальний підхід і дає змогу використовувати будь-які інші протоколи. Наприклад, у Windows 10, щоб викликати вікно настроек, можна скористатися протоколом **ms-settings** і на його основі створити Uri для класу **Launcher**:

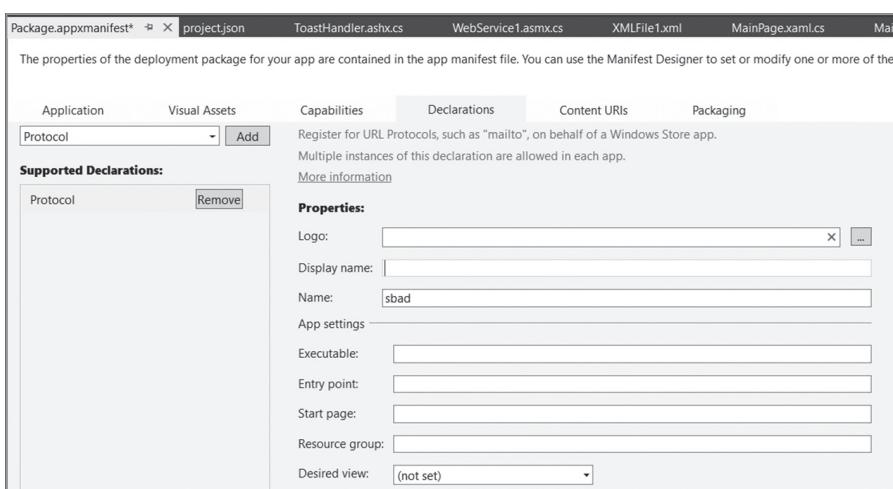
```
await Launcher.LaunchUriAsync(new Uri("ms-settings:"));
```

Крім того, є вбудовані протоколи для різних завдань:

- **ms-store**: дає змогу активувати Магазин програм і здійснювати навігацію в ньому;
- **mailto**: запускає стандартний поштовий клієнт;
- **bingmaps**: запускає програму Windows Map.

Звичайно, ви можете оголосити власний протокол для своєї програми й надати дозвіл на її виклик з інших програм, що використовують цей протокол. Якщо потрібно оголосити підтримку одного або декількох протоколів, слід відредактувати файл маніфесту програми.

Якщо ви використовуєте конструктор маніфестів, виберіть оголошення **Protocol**:



Заповніть принаймні поле **Name** – введіть ім'я протоколу, яке використовуватимуть інші програми для виклику програми.

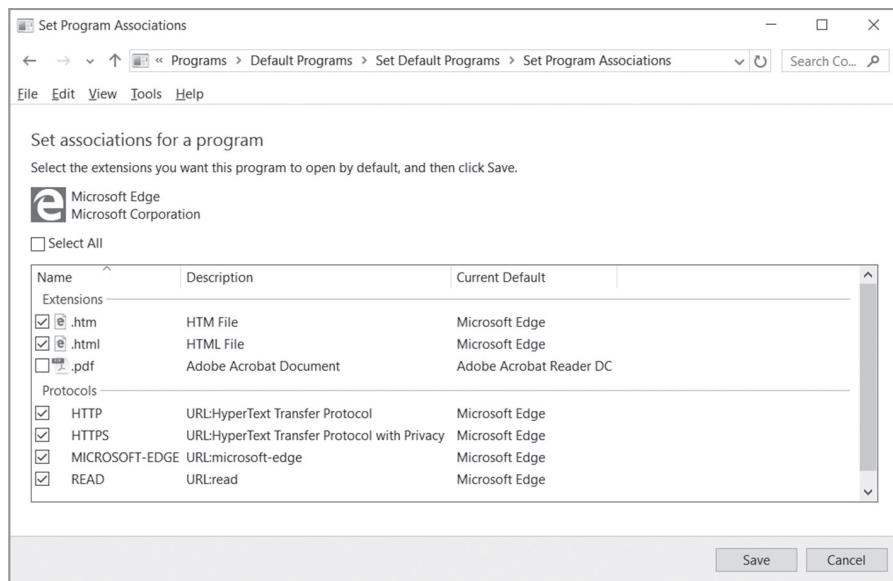
Після цього Visual Studio додасть елемент **Extension** до файлу маніфесту:

```
<Extensions>
  <uap:Extension Category="windows.protocol">
    <uap:Protocol Name="sbad" />
  </uap:Extension>
</Extensions>
```

Цей елемент має атрибут **Category**, якому було надано значення **windows.protocol**. Коли користувач інсталює програму, Windows читає цю інформацію і зв'язує програму з протоколом.

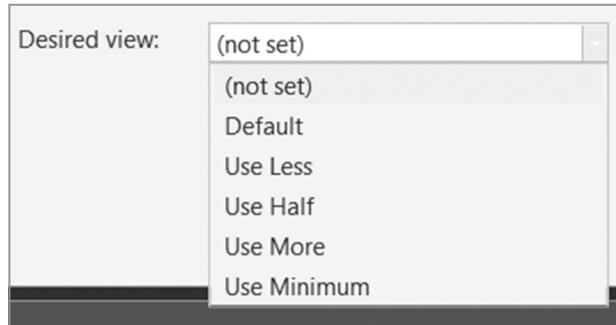
На панелі керування у розділі **Set Default Programs** наведено й інші зв'язки. Нижче показано знімок екрана, де зазначено стандартні файли та протоколи для браузера Microsoft Edge:

## Windows 10 для C# розробників



Крім того, можна заповнити поля логотипа та короткого імені, які є важливими для вікна **Set Default Programs**, але загалом не впливають на протокол.

Оберіть варіант **Desired view**, що стосуватиметься тільки програм для настільних ПК.



Після того як протокол оголошено, потрібно реалізувати логіку, що підтримуватиме активацію протоколу за методом **OnActivated**, про який ішлося в попередніх розділах.

```
public partial class App
{
```

```

protected override void OnActivated(IActivatedEventArgs
args)
{
    if (args.Kind == ActivationKind.Protocol)
    {
        // doing something
    }
}
}

```

Тепер поговорімо про клас **Launcher** і його нові функції у Windows 10.

У Windows 8.x метод **LaunchUriAsync** не давав змоги передати що-небудь, крім Uri. Якщо потрібно було передати декілька параметрів до зовнішньої програми, ви могли використовувати той самий підхід, що й для протоколу http: надіслати параметри за допомогою URI. Але якщо необхідно було передати файл як параметр, цього неможливо було зробити за допомогою **LaunchUriAsync**. Насправді клас **Launcher** підтримує метод **LaunchFileAsync**, а цей метод підтримує **StorageFile** як параметр. Однак передати декілька файлів у такий спосіб не вдається, як і поєднати обидва методи (запустити програму за допомогою Uri і водночас передати файл). Навіть щоб передати файл до сторонньої програми, потрібно було зареєструвати відповідні розширення імен файлів. Окрім того, **LaunchUriAsync** і **LaunchFileAsync** не дають можливості визначати, які програми мають бути запущені. Якщо кілька програм зареєстрували одне та саме розширення, користувачеві необхідно були вибрати програму зі списку. Також неможливо було зрозуміти, чи запущено вже програму та як отримати з неї якийсь результат.

Підсумуймо всі недоліки класу **Launcher** у Windows 8.x:

- надає два різні методи передачі Uri та файлу без можливості поєднати їх;
- не дає змоги передати декілька файлів;
- система може попросити користувача вибрати програму зі списку;
- неможливо дізнатися, чи запущено зовнішню програму;
- неможливо отримати відгук від зовнішньої програми.

Але у Universal Windows Platform корпорація Майкрософт внесла до класу **Launcher** суттєві зміни, що позбавили його всіх згаданих недоліків. Які ж це зміни? Подивімося на такий код:

```

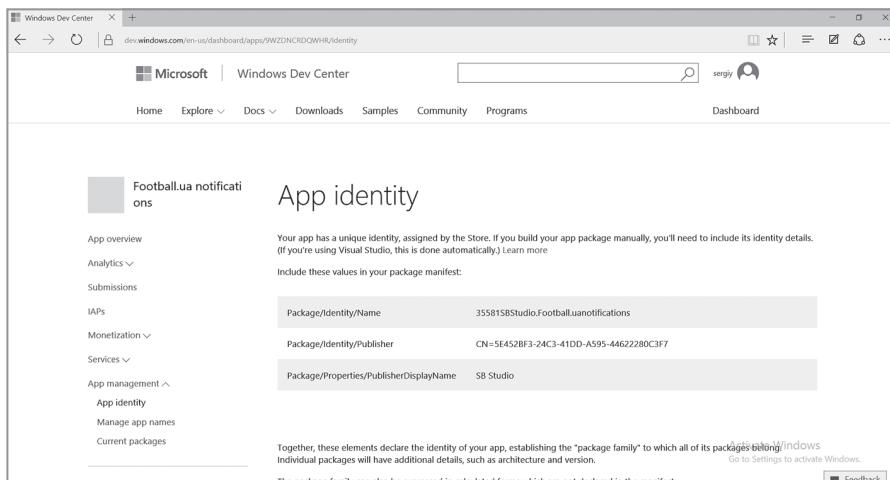
LauncherOptions options = new LauncherOptions()
{

```

```
TargetApplicationPackageFamilyName =
    "Microsoft.MicrosoftEdge_8wekyb3d8bbwe"
};

await Launcher.LaunchUriAsync(new Uri
("http://www.microsoft.com"), options);
```

Під час запуску цього коду до класу **Launcher** було надіслано запит на запуск програми, що має визначене ім'я пакета програм. Цей підхід дуже корисний для корпоративних систем, коли для клієнтів розроблено кілька програм. **LauncherOptions** гарантує, що система запустить ту програму з групи, яку потрібно. Звичайно, слід знати назву програми, але її можна легко знайти в Магазині:



Крім того, за допомогою методу **FindUriSchemeHandlersAsync** класу **Launcher** можна отримати інформацію про всі пакети, які приймають обрану схему:

```
var res = await Launcher.FindUriSchemeHandlersAsync("http");
```

Цей метод повертає масив, що містить усю необхідну інформацію, зокрема ім'я пакета:

Watch 1	
Name	Value
res.Count	1
res[0]	{Windows.ApplicationModel.AppInfo}
AppUserModelId	"Microsoft.MicrosoftEdge_8wekyb3d8bbwe!MicrosoftEdge"
DisplayInfo	{Windows.ApplicationModel.AppDisplayInfo}
Id	"MicrosoftEdge"
PackageFamilyName	"Microsoft.MicrosoftEdge_8wekyb3d8bbwe"
Native View	To inspect the native object, enable native code debugging.

Звичайно, якщо пакета не існує в системі, метод **LaunchUriAsync** поверне значення **false**, і потрібно буде просити користувачів інсталювати додаткову програму.

Клас **Launcher** містить іще один метод, **QueryUriSupportAsync**, який дає змогу отримати інформацію за умови, що система підтримує вибраний Uri:

```
var res = await Launcher.QueryUriSupportAsync(new Uri("http://www.microsoft.com"), LaunchQuerySupportType.Uri);
```

За цим методом не вдасться повернути жодної інформації про зовнішні програми, але можна перевірити, чи вийде запустити програму, використовуючи переданий Uri. Ще важливіше, що цей метод дає змогу перевірити результат, використовуючи параметр **PackageFamilyName**, і з'ясувати, чи зовнішня програма може повернути відгук.

Зазначимо, що для відкриття файлів із зовнішніх програм можна використовувати ті самі методи: **LaunchFileAsync**, **FindFileHandlersAsync** і **QueryFileSupportAsync**. Звичайно, вони не усувають проблеми передачі кількох файлів, проте в UWP метод **LaunchUriAsync** можна використовувати для передачі декількох файлів (посилань) як параметрів. Погляньмо, як це реалізувати.

Ідея полягає у використанні класу **SharedStorageAccessManager**. Завдяки йому можна обмінюватися файлами між програмами за допомогою маркерів.

```
var token=SharedStorageAccessManager.AddFile(myfile);
```

Оскільки маркер є рядком, він може слугувати параметром в **Uri**. Отже, не потрібно передавати якісь об'єкти **IStorageFile** – лише той самий **Uri**. Можна створити стільки маркерів, скільки потрібно.

Після надходження маркерів зовнішня програма отримує доступ до об'єктів **IStorageFile**:

```
string myFileToken = queryStrings.  
GetFirstValueByName ("GpxFile");  
  
if (!string.IsNullOrEmpty (myFileToken))  
{  
    StorageFile file=await SharedStorageAccessManager.RedeemTo  
kenForFileAsync (myFileToken));  
}
```

Якщо маркер одного разу було активовано, цього не можна буде зробити знову. Маркер є дійсним упродовж 14 днів. Тож якщо було виявлено проблему із запуском зовнішньої програми, можна видалити маркер зі списку, використовуючи метод **RemoveFile**.

Звичайно, коли маркери стосуються файлу, їх легко включити до Uri, проте за потреби можна передавати будь-які серіалізовані об'єкти. Для цього призначено клас **ValueSet**, який є словником серіалізованих об'єктів. Розробники можуть використовувати його для передачі як маркерів, так і будь-чого взагалі:

```
ValueSet v = new ValueSet();  
v.Add("token1", token);  
var f = await Launcher.LaunchUriAsync (myUri, options, v);
```

Окрім того, клас **Launcher** дає змогу запускати Uri для отримання результатів. Візьмімо для прикладу програму зі здійснення платежів. З її допомогою можна здійснювати платежі всередині вашої програми, але для цього слід отримати із зовнішнього джерела інформацію про те, що оплата відбулася і, можливо, деякі відомості для перевірки, чи кошти отримано. У Windows таку можливість дає метод **LaunchUriForResultsAsync**. Він має той самий список параметрів, що й **LaunchUriAsync**, проте повертає об'єкт **LaunchUriResult** замість **bool**. Цей об'єкт потрібний для відображення стану та отримання відомостей про те, чи був запуск успішний.

```
var result = await Windows.System.Launcher.  
LaunchUriForResultsAsync (myUri, options, inputData);  
if (result.Status == LaunchUriStatus.Success)  
{  
    ValueSet theValues = result.Result;  
    //do something here  
}
```

Отже, всі проблеми, що їх мала Windows 8.x із класом **Launcher**, вирішено, і знайдено зручний спосіб установити зв'язок між різними програмами.

## Кеш видавця

Іще один спосіб установити зв'язок між програмами того самого видавця – це папка кешу видавця. Ідея полягає в тому, щоб створити спеціальну папку або папки, призначенні не для програм, а для видавця. Тож якщо видавець створює групу програм і хоче зробити деякі дані спільними для них, можна використовувати папки кешу.

Щоб створити папку чи папки кешу видавця, розробники мають використовувати файл маніфесту на кшталт такого:

```
<Extensions>
  <Extension Category ="windows.publisherCacheFolders">
    <PublisherCacheFolders>
      <Folder Name="myFolder"/>
    </PublisherCacheFolders>
  </Extension>
</Extensions>
```

Тут ми створили тільки одну папку. Саме створили – це правильне формулювання, оскільки папку буде створено автоматично на основі маніфесту програми. Зауважте, що цю папку розміщено в спеціальному каталозі, пов'язаному з видавцем, а не з програмою. Наприклад, у **C:\Users\Sergey\AppData\Local\Publishers\kj4a6z6kv5v3p\myFolder**.

Для роботи з цією папкою чи папками потрібно лише отримати посилання на об'єкт **StorageFolder** за допомогою класу **ApplicationData**:

```
var f = ApplicationData.Current.GetPublisherCacheFolder("myFolder");
```

Вам одразу буде відкрито доступ до папки, де ви зможете створювати вкладені папки, перевіряти файли в папці, здійснювати будь-які доступні операції. Усе це – завдяки простому об'єкту **StorageFolder**.

Щоб очистити папку кешу видавця, скористайтеся методом **ClearPublisherCacheFolderAsync**:

```
ApplicationData.Current.ClearPublisherCacheFolderAsync("myFolder");
```

Отже, ви бачите, як легко і просто реалізувати обмін даним між корпоративними програмами, якщо є набір різних програм, але зі спільними налаштуваннями, тимчасовими файлами тощо.

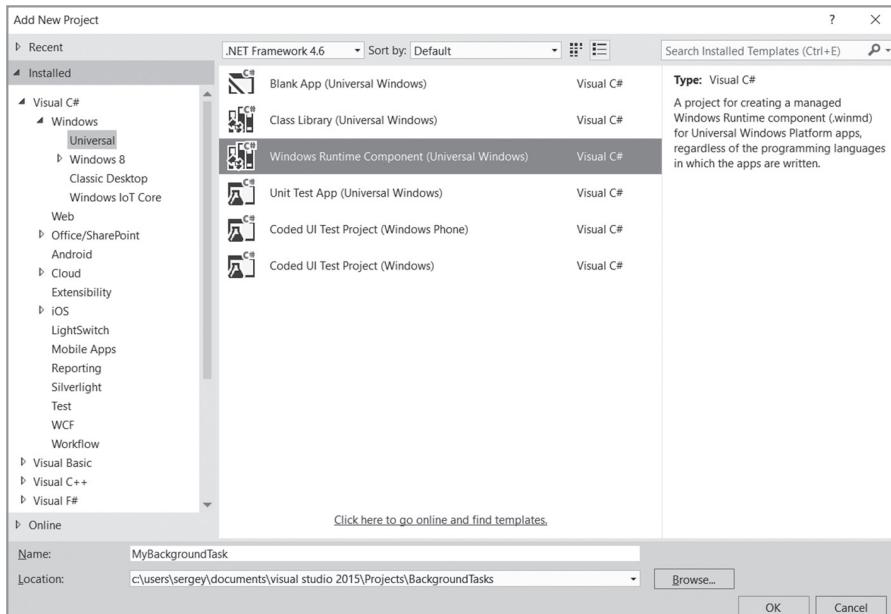
Розділ 16.

## **Служби програм і фонові завдання**

## Фонові завдання

У попередніх розділах ми обговорювали роботу програм лише в активному режимі. Але Universal Windows Platform уможливлює виконання певних операцій також у фоновому режимі, коли програма не активна. Поговоримо про те, як створити просте фонове завдання й активувати його.

Насамперед майте на увазі, що для реалізації фонового завдання потрібно створити окремий проект. Адже суть такого завдання полягає у виконанні певної дії, поки програма не активна, тому використовувати її власні методи та класи не вийде. Натомість створюється окремий компонент. У Visual Studio потрібно просто додати до рішення ще один проект і вибрати шаблон Windows Runtime Component:



За допомогою цього шаблону можна створити клас, що розширює середовище Windows Runtime. За замовчуванням шаблон містить клас, який можна перейменувати і використовувати на власний розсуд.

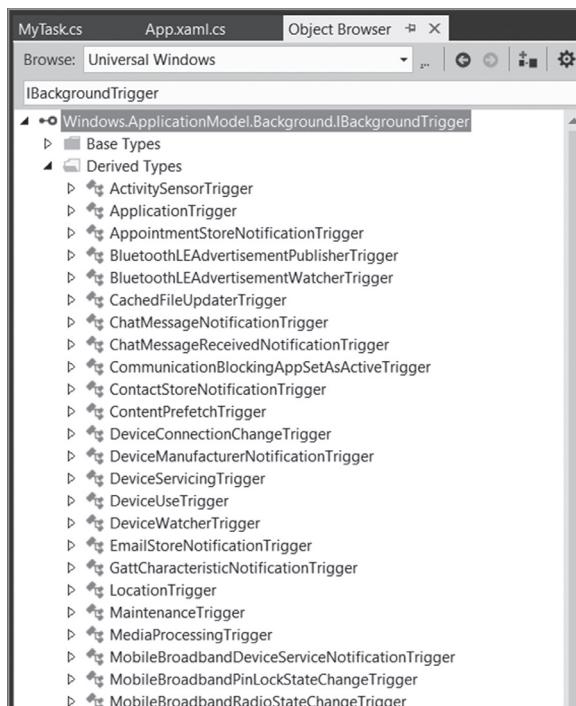
Звичайно, не кожен компонент Windows Runtime система Windows може використовувати як фонове завдання. Щоб Windows зрозуміла, у який спосіб запускати вашу задачу, слід реалізувати інтерфейс **IBackgroundTask**. Цей інтерфейс містить тільки один метод – **Run**, і система використовуватиме його для виконання коду, щойно буде створено екземпляр завдання.

Далі ми змінюємо наявний клас таким чином:

```
namespace MyBackgroundTask
{
    public sealed class MyTask : IBackgroundTask
    {
        public void Run(IBackgroundTaskInstance taskInstance)
        {
            throw new NotImplementedException();
        }
    }
}
```

Цей крок обов'язковий для створення будь-якого фонового завдання. Однак перш ніж реалізувати вміст методу **Run**, розглянемо, коли і як Windows активує фонові завдання.

Universal Windows Platform дає змогу активувати фонові завдання за допомогою тригерів. Щоб переглянути повний перелік доступних тригерів, відкрийте у вікні Object Browser всі класи з підтримкою інтерфейсу **IBackgroundTrigger**:



Загалом доступно 39 класів, що дають змогу прив'язати фонові завдання різноманітних подій. Наприклад, код може виконуватися, щойно надійде push-сповіщення чи зміниться стан підключення, або ж через певні проміжки часу.

Далі потрібно визначити, який тригер відповідає вашим вимогам.

Вибрали потрібний тригер, налаштуйте відповідним чином програму й повідомте систему про наявність фонових завдань у програмі. Для цього оголосіть їх у маніфесті, додавши пункт **Background Tasks**:

The screenshot shows the 'Available Declarations' section with 'Background Tasks' selected. An 'Add' button is visible. In the 'Supported Declarations' section, 'Background Tasks' is listed with a 'Remove' button. The 'Description' section explains that it enables specifying the class name of an in-proc server DLL that runs the app code in the background in response to external trigger events. It notes that multiple instances are allowed and provides a link to 'More information'. The 'Properties' section lists supported task types, with 'Timer' checked and other options like Audio, Bluetooth, Chat message notification, etc., available but unchecked.

Available Declarations:	Description:
Background Tasks	Enables the app to specify the class name of an in-proc server DLL that runs the app code in the background in response to external trigger events. The class hosted in the in-proc server DLL is activated for background activation, and its Run method is invoked. Multiple instances of this declaration are allowed in each app. <a href="#">More information</a>
Supported Declarations:	Properties:
Background Tasks	Supported task types <input type="checkbox"/> Audio <input type="checkbox"/> Bluetooth <input type="checkbox"/> Bluetooth device connection <input type="checkbox"/> Chat message notification <input type="checkbox"/> Control channel <input type="checkbox"/> Device use trigger <input type="checkbox"/> General <input type="checkbox"/> Location <input type="checkbox"/> Media processing <input type="checkbox"/> Phone call <input type="checkbox"/> Push notification <input type="checkbox"/> System event <input checked="" type="checkbox"/> Timer <input type="checkbox"/> VPN client

Якщо програма може запускати кілька фонових завдань, кожне з них слід оголосити окремо. Виберіть в параметрах тип завдання і вкажіть точку входу – повне ім'я класу фонового завдання включно з простором імен:

The screenshot shows the 'App settings' configuration screen with the following fields:

- Executable: (empty)
- Entry point: MyBackgroundTask.MyTask
- Start page: (empty)
- Resource group: (empty)

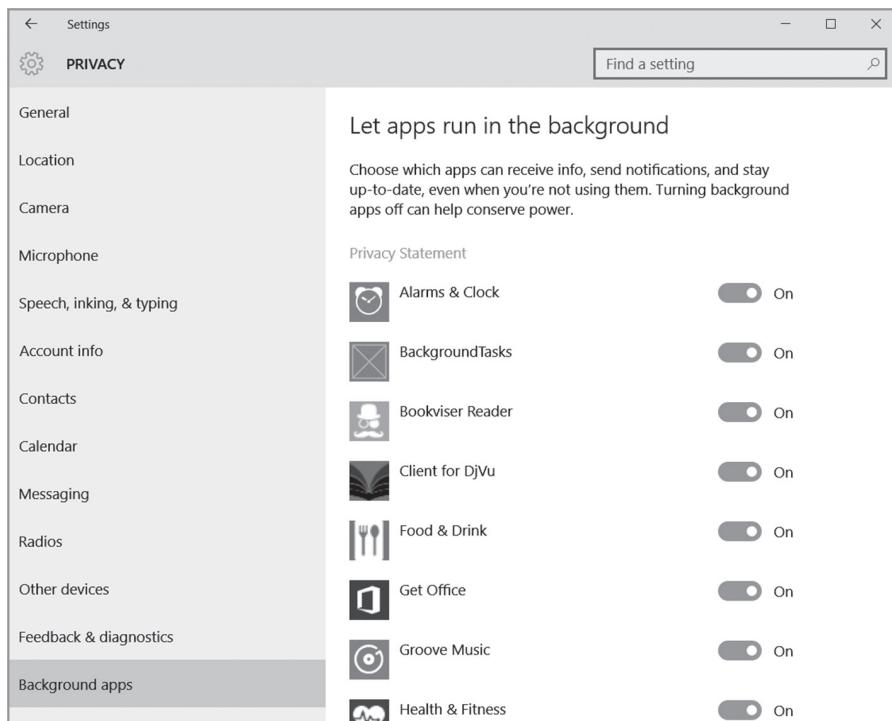
Для прикладу ми створимо фонове завдання, яке запускатиметься за тригером часу.

Тепер нам потрібен код для реєстрації завдання в системі. Для цього ми реалізуємо метод **RegisterBackgroundTasks** і викличемо його в належний час.

Перед реєстрацією необхідно запитати дозволи і з'ясувати, чи можливо створювати фонові завдання. Це можна зробити за допомогою класу **BackgroundExecutionManager** і методу **RequestAccessAsync**.

```
var status = await BackgroundExecutionManager.  
RequestAccessAsync();
```

Зазвичай у відповідь на цей виклик система не створює жодних повідомлень, а просто надає доступ. Однак користувач може будь-коли скасувати право доступу на вкладці **Privacy у вікні налаштувань**:



Реєстрація починається з перевірки доступності фонових завдань. Це дуже важливий етап. Якщо завдання доступні, їх можна видалити або повернути з

методу й зареєструвати – залежно від сценарію. У нашому прикладі ми видалимо наявні завдання і створимо нові. Universal Windows Platform містить клас **BackgroundTaskRegistration**, в якому можна легко перелічити всі зареєстровані програмою завдання:

```
foreach (var task in BackgroundTaskRegistration.AllTasks)
{
    task.Value.Unregister(true);
}
```

Тепер зареєструємо фонове завдання. Додамо проект із цим завданням до проекту програми у вікні **Add Reference i продовжимо** реалізацію коду:

```
var timeTrigger = new TimeTrigger(15, true);
var backgroundBuilder = new BackgroundTaskBuilder();
backgroundBuilder.Name = "timerTask";
backgroundBuilder.TaskEntryPoint = typeof(MyTask).FullName;
backgroundBuilder.SetTrigger(timeTrigger);
backgroundBuilder.Register();
```

У коді вище ми використали клас **TimeTrigger** для створення тригера, що активує фонове завдання через 15 хвилин. Ви не можете визначити інтервал, що менше 15 хвилин для часових тригерів. Другий параметр конструктора часового тригера дає змогу визначити, буде завдання запущено один раз чи періодично. У нашому випадку ми активуємо завдання лише раз.

Зареєструвати завдання можна за допомогою класу **BackgroundTaskBuilder**. Використовуючи екземпляр класу, ви можете визначити ім'я завдання, ім'я компонента Windows Runtime і тригера. Викликавши метод **Register**, ми завершимо процес реєстрації. Нижче ви можете знайти повний опис методу:

```
public async void RegisterBackgroundTasks()
{
    var status = await BackgroundExecutionManager.
        RequestAccessAsync();

    if (status == BackgroundAccessStatus.Denied)
        throw new Exception("Access is denied");

    foreach (var task in BackgroundTaskRegistration.AllTasks)
    {
        task.Value.Unregister(true);
    }
}
```

```

var timeTrigger = new TimeTrigger(15, true);
var backgroundBuilder = new BackgroundTaskBuilder();
backgroundBuilder.Name = "timerTask";
backgroundBuilder.TaskEntryPoint = typeof(MyTask).FullName;
backgroundBuilder.SetTrigger(timeTrigger);
backgroundBuilder.Register();
}

```

Клас **BackgroundBuilder** має ще один важливий метод, **AddCondition**. Завдяки цьому методу ви можете надіслати запит на запуск завдання за виконання певних умов. Наприклад, можна запустити завдання, якщо користувач досяжний:

```

backgroundBuilder.AddCondition(new SystemCondition
(SystemConditionType.UserPresent));

```

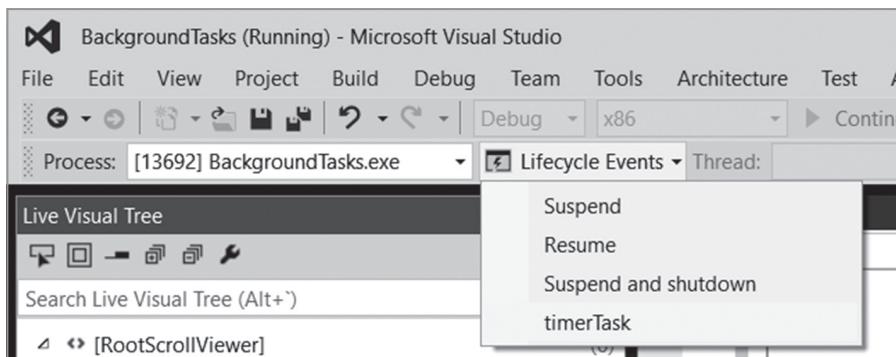
Ми досі не реалізували код для фонового завдання. У ньому ми плануємо відображати лише toast-повідомлення. Тож можна реалізувати такий код:

```

public void Run(IBackgroundTaskInstance taskInstance)
{
    XmlDocument xml = new XmlDocument();
    xml.LoadXml(
        "<toast>" +
        "<visual>" +
        "<binding template=\"ToastGeneric\">" +
        "<text>Football application</text>" +
        "<text>The second period will start shortly</text>" +
        "</binding>" +
        "</visual>" +
        "</toast>");
    var toastNot = ToastNotificationManager.CreateToastNotifier();
    ToastNotification toast = new ToastNotification(xml);
    toastNot.Show(toast);
}

```

А тепер ви можете побудувати рішення і запустити програму. Зауважте, що вам не потрібно буде очікувати 15 хвилин для того, щоб відтестувати завдання. Якщо ви запустили програму в режимі налагодження, то можете обрати будь-яке доступне завдання на панелі інструментів налагодження (**Debug Location**):



У нашому прикладі ми використовували лише синхронні виклики в реалізації завдання, але якщо вам потрібно також викликати з методу **Run** деякі асинхронні методи, ви повинні застосувати відтермінування (**deferral**), для чого рекомендуємо використовувати такий шаблон:

```
BackgroundTaskDeferral _deferral;
public async void Run(IBackgroundTaskInstance taskInstance)
{
    _deferral = taskInstance.GetDeferral();

    //doing async tasks

    _deferral.Complete();
}
```

Якщо ви не використовуєте **deferral** для **async**-викликів, система може завершити ваше завдання до того, як буде завершено роботу методу.

## Служби програм

Далі в цьому розділі мова йтиме про ще один спосіб встановлення зв'язків між різними програмами, а саме про служби програм (Application Services). В Application Services Microsoft поєднує дві такі ідеї, як фонові завдання і веб-сервіси, що дає змогу використовувати фонові завдання як служби для інших програм на тому самому ПК.

Цю функцію призначено насамперед для корпоративних компаній, які використовують багато різних програм для виконання різних завдань. Відомо багато сценаріїв, коли спільні завдання можуть бути винесені в окремі фонові

завдання і опубліковані для інших програм. Наприклад, IT-відділ банку може розробити програму для автономної роботи, яка резервуватиме деякі контактні дані заздалегідь, ділитиметься цими даними з іншими програмами на тому самому комп'ютері, збиратиме інформацію про їх використання та синхронізуватиме дані з власного інтерфейсу. Водночас банк може попросити постачальників розробити інші програми, які не будуть зв'язуватися із сервером банку безпосередньо, а використовуватимуть програму банку, як спосіб отримувати контактні дані та надсилати дані для синхронізації. Крім того, служби програм можуть бути використані для стандартних програм Windows, але загальнодоступні програми із вбудованими службами програм – не найнадійніше рішення. У будь-якому разі подивімося, як реалізувати служби програм, а потім – як розробники їх використовують.

Спробуємо реалізувати дві програми. У першій фонове завдання буде службою, вона отримуватиме інформацію про користувача з іншої програми і повертає новий ідентифікатор контракту. Друга програма працюватиме як користувач служби. Ми не зираємося реалізовувати жоден інтерфейс – тільки код, який демонструє використання функції Application Services.

Зауважте, що служби програм можуть бути реалізовані всередині стандартної програми на Universal Windows Platform. Таким чином, якщо вам навіть не потрібен користувацький інтерфейс, а ви тільки хочете створити деякі служби, необхідно розробити стандартну UWP-програму. Щоб активувати службу програми, користувачі не повинні запускати програму, але про всякий випадок вам все-таки потрібно буде реалізувати основний екран програми. Таким чином, щоб почати реалізацію і тестування служб програм, потрібно створити два порожніх UWP-проекта.

Щойно ви створите дві програми, можете починати з проекту, який міститиме службу. Розпочнемо з фонового завдання і опишемо такий клас:

```
class ContractIDService : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        throw new NotImplementedException();
    }
}
```

Це типовий шаблон для фонового завдання, але для того, щоб зробити його службою, нам потрібно внести певні зміни у файл маніфесту в програмі:

```
<Application Id="App"
. . .
<Extensions>
    <uap:Extension Category="windows.appService"
        EntryPoint="AppService.ContractIDService">
        <uap:AppService Name="GetContractIDService"/>
    </uap:Extension>
</Extensions>
</Application>
```

Ми використали ім'я класу (включно з простором імен) як вхідну точку для служби та за допомогою елемента **AppService** оголосили ім'я служби, що має використовуватися в клієнтській програмі (можна обрати будь-яке ім'я).

Реалізуємо клас **ContractIDService**. Перш за все необхідно реалізувати метод **Run**:

```
private static BackgroundTaskDeferral taskDeferal;

public void Run(IBackgroundTaskInstance taskInstance)
{
    taskDeferal = taskInstance.GetDeferral();

    var appService = taskInstance.TriggerDetails as
        AppServiceTriggerDetails;
    if (appService.Name== "GetContractIDService")
    {
        appService.AppServiceConnection.RequestReceived +=
            AppServiceConnection_RequestReceived;
    }
    else
    {
        taskDeferal.Complete();
    }
}
```

Цей код створює відтермінований об'єкт і перевіряє ім'я служби. Якщо ви не створите відтермінований об'єкт у цьому методі, він завершить роботу і службу буде видалено. Крім того, може бути кілька служб з тією самою точною входу, тому ім'я краще перевірити. Для того, щоб отримати дані, нам потрібно присвоїти обробник події **RequestReceived**, як це було зроблено в коді.

Розгляньмо приклад реалізації обробника події **RequestReceived**:

```
private async void AppServiceConnection_RequestReceived(
    AppServiceConnection sender,
    AppServiceRequestReceivedEventArgs args)
{
    var messageDeferral = args.GetDeferral();
    var message = args.Request.Message;
    string name = message["Name"].ToString();

    ValueSet returnMessage = new ValueSet();
    returnMessage.Add("contractID", $"{Guid.NewGuid().
        ToString()}{name}");
    var responseStatus=await args.Request.
        SendResponseAsync(returnMessage);

    messageDeferral.Complete();
    taskDeferral.Complete();
}
```

Звичайно, це суто демонстраційна реалізація, і ваша реальна програма відкриємо ховище даних, отримуватиме новий ідентифікатор і зберігатиме вхідні дані. До того ж ви можете перевіряти дозволи і реалізувати ще багато чого, однак у будь-якому разі можна вважати цей шаблон базовим.

Як бачимо, наш обробник події був описаний із ключовим словом **async** і ми маємо ще один відтермінований об'єкт і можемо гарантувати, що всі await-вилики будуть завершенні, а вхідне повідомлення не загубиться. Використовуючи властивості **Message**, ми можемо легко отримати доступ до об'єкта класу **ValueSet**. Ми будемо використовувати цей клас для запакування даних. Завдяки цьому словнику ви можете запаковувати будь-які серіалізовані об'єкти і **ValueSet** використовується багатьма класами UWP, такими як **Launcher**. Крім того, ми використовуємо той самий клас для створення ховища для вихідної інформації, а метод **SendResponseAsync** – для повернення даних клієнту.

Зауважте, що ми зверталися до відтермінованого об'єкта з методу **Run**, щоб залишити процес. Це означає, що наступного разу, коли клієнт надішле дані до тієї самої служби, система створить його знову і викличе метод **Run**. Отже, якщо ви знаєте, що будете використовувати службу час від часу, можете залишити відтермінований об'єкт «живим» і просто реалізувати для нього команду «звільнення» (**dispose**):

```
private async void AppServiceConnection_RequestReceived(
    AppServiceConnection sender,
    AppServiceRequestReceivedEventArgs args)
{
    var messageDeferral = args.GetDeferral();
    var message = args.Request.Message;
    string command = message["Command"].ToString();

    switch (command)
    {
        case "getID":
            string name = message["Name"].ToString();

            ValueSet returnMessage = new ValueSet();
            returnMessage.Add("contractID",
                $"'{Guid.NewGuid().ToString()}{name}'");
            var responseStatus =
                await args.Request.SendResponseAsync
                (returnMessage);

            messageDeferral.Complete();
            break;

        case "exit":
            taskDeferral.Complete();
            break;
    }
}
```

Якщо ви це зробите, розробники зможуть оптимізувати продуктивність клієнтських програм. Ось власне приклад реалізації клієнтської програми:

```
AppServiceConnection app = new AppServiceConnection();
app.AppServiceName = "GetContractIDService";
app.PackageFamilyName =
"ad5ff53a-7ccc-4f70-b0c2-6b909bba77a0_kj4a6z6kv5v3p";

AppServiceConnectionStatus status = await app.OpenAsync();

if (status == AppServiceConnectionStatus.Success)
{
    ValueSet message = new ValueSet();
```

```

message.Add("command", "getID");
message.Add("Name", "Sergii");
AppServiceResponse response =
    await app.SendMessageAsync(message);
if (response.Status == AppServiceResponseStatus.Success)
{
    //doing something
}
}

```

Ми просто створили об'єкт **AppServiceConnection** і підключилися до служби, використовуючи метод **OpenAsync**. Зверніть особливу увагу, що вам потрібно знати ім'я пакету. Ви можете отримати його з Магазину, якщо вже опублікували програму, а для тестування можна скористатися одним із таких двох способів:

- Коли розгортаєте службу, Visual Studio відображає у вікні результатів щось схоже на такий рядок: **Deployment complete (113ms). Full package name: "ad5ff53a-7ccc-4f70-b0c2-6b909bba77a0\_1.0.0.0\_x86\_kj4a6z6kv5v3p"**. Просто видаліть версію і платформу з цього повідомлення (**\_1.0.0.0\_x86\_**) і ви отримаєте назву пакету.
- Щоб отримати назву напряму з вашої програми, можете використовувати такий код:

```
string name = Package.Current.Id.FamilyName;
```

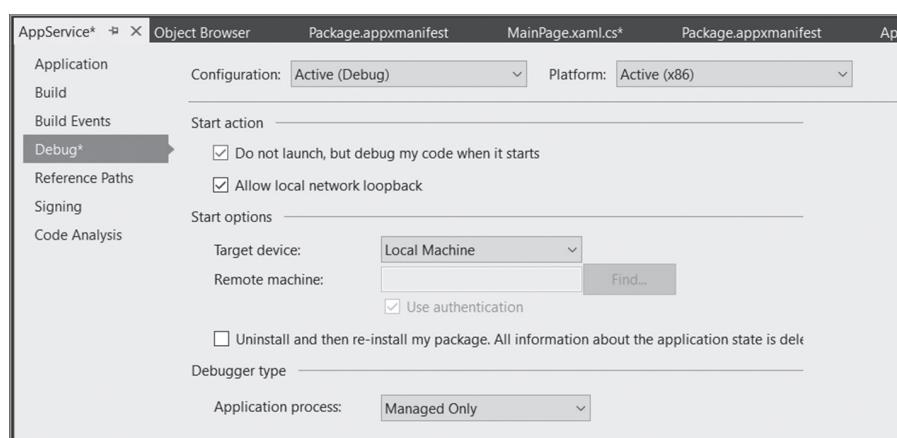
Просто скористайтесь точкою зупину і дізнайтеся значення цієї змінної в режимі налагодження.

Як тільки ви створите підключення, зможете використовувати його для того, щоб надсилати повідомлення за допомогою методу **SendMessageAsync**.

Клас **AppServiceConnection** має подію **RequestReceived**. Отже, ваша програма може отримувати запити зі служб також. Використовуючи ці класи, ви можете встановити двосторонній зв'язок.

Врешті решт, якщо вам потрібно налагоджувати службу, ми б рекомендували використовувати функцію **Do not launch but debug my code when it starts** (Запускати код у режимі налагодження). Вона дає змогу бачити, що трапляється, коли клієнт активує вашу службу.

## Windows 10 для C# розробників



Розділ 17.

## **Робота в мережі**

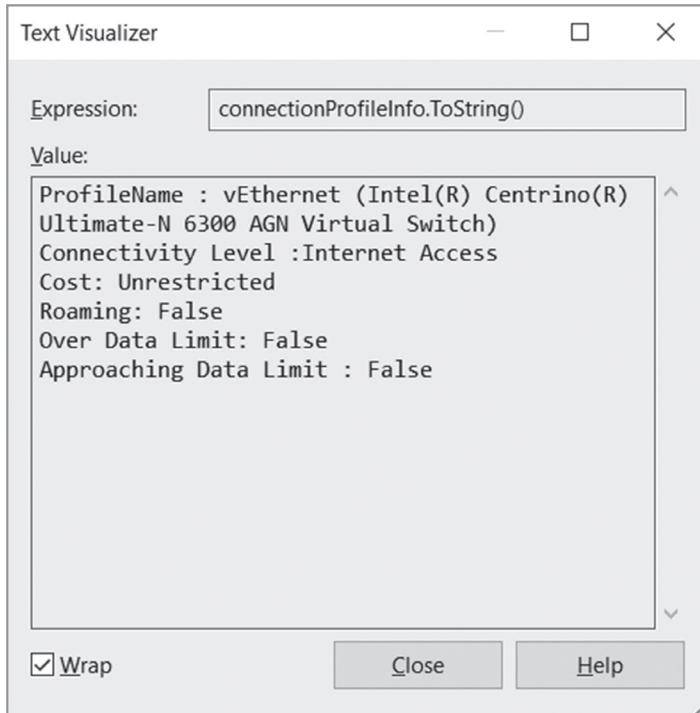
## Інформація про мережу

Розпочнімо цей розділ із розгляду класів, що дають змогу отримати інформацію про статус мережі. Усі ці більш ніж 20 класів розташовані в просторі імен **Windows.Networking.Connectivity**, а найважливіші з них – **NetworkInformation** і **ConnectionProfile**. Використовуючи ці класи, ви легко можете дізнатися поточний статус мережі. Для цього вам потрібно використати статичний метод **GetInternetConnectionProfile** із класу **NetworkInformation**, який поверне посилання на об'єкт **ConnectionProfile**:

```
var connectionProfile =
    NetworkInformation.GetInternetConnectionProfile();
var connectionProfileInfo = new StringBuilder($"ProfileName :
{connectionProfile(ProfileName)}\n");
switch (connectionProfile.GetNetworkConnectivityLevel())
{
    case NetworkConnectivityLevel.None:
        connectionProfileInfo.AppendLine("Connectivity Level :
None");
        break;
    case NetworkConnectivityLevel.LocalAccess:
        connectionProfileInfo.AppendLine("Connectivity Level :
Local Access");
        break;
    case NetworkConnectivityLevel.ConstrainedInternetAccess:
        connectionProfileInfo.AppendLine("Connectivity Level :
Constrained Internet Access");
        break;
    case NetworkConnectivityLevel.InternetAccess:
        connectionProfileInfo.AppendLine("Connectivity Level :
Internet Access");
        break;
}
var connectionCost = connectionProfile.GetConnectionCost();
switch (connectionCost.NetworkCostType)
{
    case NetworkCostType.Unrestricted:
        connectionProfileInfo.AppendLine("Cost: Unrestricted");
        break;
    case NetworkCostType.Fixed:
        connectionProfileInfo.AppendLine("Cost: Fixed");
        break;
    case NetworkCostType.Variable:
```

```
        connectionProfileInfo.AppendLine("Cost: Variable");
        break;
    case NetworkCostType.Unknown:
        connectionProfileInfo.AppendLine("Cost: Unknown");
        break;
    default:
        connectionProfileInfo.AppendLine("Cost: Error");
        break;
}
connectionProfileInfo.AppendLine($"Roaming:
{connectionCost.Roaming}");
connectionProfileInfo.AppendLine($"Over Data Limit:
{connectionCost.OverDataLimit}");
connectionProfileInfo.AppendLine($"Approaching Data Limit :
{connectionCost.ApproachingDataLimit}");
```

Ви можете помістити цей код у метод **OnNavigatedTo** і використовувати точки зупину для перегляду інформації в **connectionProfileInfo**:



Крім того, завдяки **ConnectionProfile** можна дізнатися обсяги отриманих і надісланих даних. Для цього ви можете використати метод **GetLocalUsage** і прочитати всю інформацію за допомогою класу **DataUsage**.

Таким чином, ви можете бачити, що все, що вам потрібно, є у вашій програмі.

## Робота з даними

Як тільки ви дізнаєтесь статус підключення до мережі, можете починати працювати з даними в Інтернеті. Universal Windows Platform надає для цього кілька класів і найпростіший із них – це **HttpClient**, що розміщується в просторі імен **Windows.Web.Http**. Порівняно з **HttpWebRequest/HttpWebResponse** з .NET Framework цей клас базується на новому підході **async/await** і вам не потрібно використовувати специфічних делегатів, обробників подій чи подібних засобів. Надіслати запит на отримання контенту через Uri тепер можна за допомогою такого коду:

```
HttpClient client = new HttpClient();
string s=await client.GetStringAsync(new Uri
("http://www.microsoft.com"));
```

Отже, у більшості випадків, коли потрібно прочитати дані у форматі JSON чи XML, ви будете використовувати цей простий код і працювати з рядками, як тільки метод поверне дані. Але якщо ви хочете повернути більш складні дані, зокрема заголовки, можете використовувати методи **PostAsync**, **PutAsync** чи **SendRequestAsync**. Усі ці методи дають змогу передати об'єкт, що реалізує інтерфейс **IHttpContent**. Отже, ви можете обирати між різним типом контенту.

Звичайно, **HttpClient** працює добре, якщо користувачі не переходятять на іншу сторінку і тільки очікують даних. Але в деяких випадках, якщо вам потрібно завантажити з Інтернету чи в Інтернет великі обсяги даних, клієнт **HttpClient** не працюватиме належним чином. Не думаю, що варто змушувати користувача чекати кілька хвилин, не закриваючи сторінку, поки файл завантажується. Фактично користувач може навіть перейти до іншої програми. Тож якщо ви хочете завантажити чи передати великі обсяги даних, краще скористатися класами **BackgroundDownloader** та **BackgroundUploader**, які можуть працювати у фоновому режимі. Ці класи розташовано в просторі імен **Windows.Networking**. **BackgroundTransfer** і ви можете використовувати їх прямо в активній частині програми. Таким чином, вам не потрібно створювати жодних спеціальних завдань, бібліотек тощо. Розгляньмо такий приклад:

```

DownloadOperation download = null;
try
{
    Uri source = new Uri(serverAddressFile);
    StorageFile destinationFile =
        await KnownFolders.PicturesLibrary.
            CreateFileAsync("Image.jpg",
            CreationCollisionOption.GenerateUniqueName);
    BackgroundDownloader downloader = new BackgroundDownloader();
    download = downloader.CreateDownload(source, destinationFile);
    Progress<DownloadOperation> progressCallback =
        new Progress<DownloadOperation>(DownloadProgress);
    await download.StartAsync().AsTask(progressCallback);
}
catch (Exception ex)
{
}

```

Ми почали завантажувати файл з сервера для збереження на локальному диску. Як видно, для того, щоб почати завантажувати файл, вам просто потрібно створити об'єкт **BackgroundDownloader** та викликати його метод **CreateDownload**. Як тільки об'єкт **DownloadOperation** створено, ви можете розпочинати процес і передати посилання на обробник події, щоб отримати оновлену інформацію про статус.

Зверніть особливу увагу, що цей код працює добре, але на той випадок, коли користувач може закрити сторінку чи програму, нам потрібно писати додатковий код, що допоможе завантажити файл і оновити зміни в інтерфейсі відповідно до отриманої інформації. Для того, щоб це зробити, ми просто зможемо використати статичний метод **GetCurrentDownloadsAsync** класу **BackgroundDownloader**.

## Як працювати з RSS feeds

Ще один корисний простір імен – це **Windows.Web.Syndication**. Завдяки цьому простору імен ви можете спростити свою роботу з RSS feeds. Цей простір імен підтримує RSS 2.0 і Atom 1.0, але у багатьох випадках цього достатньо.

Щоб почати працювати з веб-каналом (feed), вам потрібно використовувати клас **SyndicationClient**, який дає змогу завантажити веб-канал і отримати посилання на об'єкт **SyndicationFeed**, що містить feed-дані:

```
SyndicationClient client = new SyndicationClient();
SyndicationFeed currentFeed =
    await client.RetrieveFeedAsync(uri);
```

Крім того, ви можете використовувати властивість **BypassCacheOnRetrieve**, щоб уникнути проблем з кешованими даними.

Як тільки дані завантажені, ми можемо використовувати властивість **Items** для отримання доступу до кожного запису:

```
foreach (var item in currentFeed.Items)
{
    string title = item.Title != null ? item.Title.Text :
        "(no title)";
    string link = string.Empty;
    if (item.Links.Count > 0)
    {
        link = item.Links[0].Uri.AbsoluteUri;
    }
    string content = "(no content)";
    if (item.Content != null)
    {
        content = item.Content.Text;
    }
    else if (item.Summary != null)
    {
        content = item.Summary.Text;
    }
}
```

Отже, як бачите, використовувати ці класи зовсім нескладно.

## Використання класу **WebAuthenticationBroker**

Сучасні програми часто не використовують власну систему автентифікації, але дають змогу входити за допомогою облікового запису Facebook, Live ID тощо. Зазвичай процес автентифікації базується на механізмі OAuth і вимагає використовувати WebView, щоб відкрити зовнішню сторінку для автентифікації, обробляти навігацію для WebView і створити власну логіку, виходячи зі статусу і поточної сторінки. Ви можете здійснювати обробку з нуля, але UWP підтримує

спеціальний класу **WebAuthenticationBroker**, що спрощує вашу роботу. Завдяки цьому класу ви можете використовувати OAuth чи Open ID і все що вам потрібно – викликати статичний метод **AuthenticateAsync** і обробити результат.

Наведений далі код демонструє, як використовувати Facebook для автентифікації користувачів у програмі. Спочатку ми маємо підготувати URI для Facebook. Ви можете знайти ці URI, відвідавши сторінку Facebook для розробників:

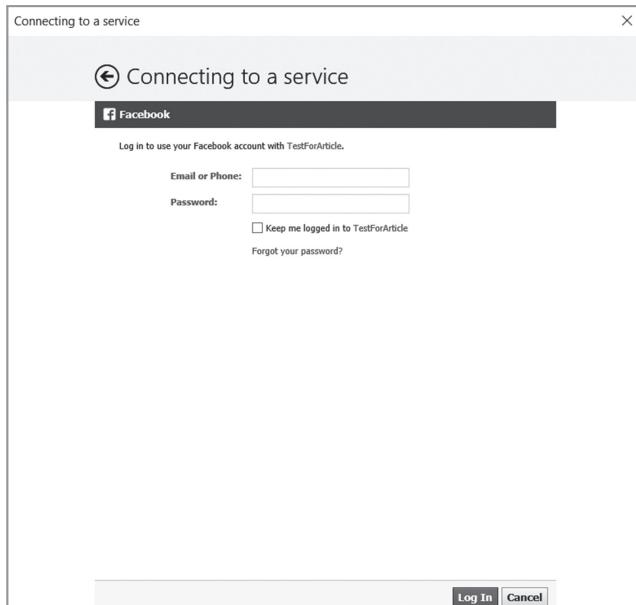
```
private static string FACEBOOK_DATA_LINK =
    "https://graph.facebook.com/me?access_token=";
private static string FACEBOOK_URI =
    "https://www.facebook.com/dialog/oauth";
private static string FACEBOOK_REDIRECT_URI =
    "https://www.facebook.com/connect/login_success.html";
private static string CLIENT_ID = "...";
private static string CLIENT_SECRET_KEY = "...";
private static string RESPONSE_TYPE = "token";
private static string TOKEN_PATTERN =
    string.Format("{0}#access_token={1}&expires_in={2}",
FACEBOOK_REDIRECT_URI, "(?<access_token>.+)",
"(?<expires_in>.+)");
```

Для отримання **client\_id** і секретного ключа вам потрібно зареєструвати вашу програму на сторінці Facebook для розробників і скопіювати значення звідти. Як тільки всі дані отримані, ви можете реалізовувати код автентифікації:

```
try
{
    var requestUri =
        new Uri(string.Format("{0}?client_id={1}&redirect_uri=
{2}&response_type={3}&display=popup&scope=
publish_stream&client_secret={4}&scope=
publish_stream,user_photos,user_location,
offline_access",
        FACEBOOK_URI, CLIENT_ID, FACEBOOK_REDIRECT_URI,
        RESPONSE_TYPE, CLIENT_SECRET_KEY),
        UriKind.RelativeOrAbsolute);
    var callbackUri = new Uri(FACEBOOK_REDIRECT_URI,
        UriKind.RelativeOrAbsolute);
    var auth = await WebAuthenticationBroker.AuthenticateAsync
        (WebAuthenticationOptions.None, requestUri, callbackUri);
    switch (auth.ResponseStatus)
    {
```

```
        case WebAuthenticationStatus.ErrorHttp:
            break;
        case WebAuthenticationStatus.Success:
            var match = Regex.Match(auth.ResponseData,
                TOKEN_PATTERN);
            var access_token = match.Groups["access_token"].Value;
            var expires_in = match.Groups["expires_in"].Value;
            break;
        case WebAuthenticationStatus.UserCancel:
            break;
        default:
            break;
    }
}
catch (Exception ex)
{
    throw ex;
}
```

Після запуску цього коду відобразиться вікно, що дає змогу пройти автентифікацію за допомогою облікового запису Facebook:



Звичайно, ваша програма не матиме змоги отримати доступ до логіну і паролю, але якщо їх введено правильно, ви отримаєте унікальний токен, який можна асоціювати з користувачем. Використовуючи токен, ви можете отримати доступ до потрібних вам даних користувача, наприклад таких як фото.

Зверніть особливу увагу, що в разі використання облікового запису Facebook вам потрібно вказати тип програми. Facebook підтримує Windows-програми, але якщо ідентифікатор програми з Магазину вам не відомий, можна змінити налаштування Client OAuth, щоб уникнути проблем із безпекою під час тестування:

The screenshot shows the 'Client OAuth Settings' page with the following configuration:

- Client OAuth Login:** Enabled (YES). Description: Enables the standard OAuth client token flow. Secure your application and prevent abuse by locking down which token redirect URLs are allowed with the options below. Disable globally if not used. [?]
- Web OAuth Login:** Enabled (YES). Description: Enables web based OAuth client login for building custom login flows. [?]
- Force Web OAuth Reauthentication:** Disabled (NO). Description: When on, prompts people to enter their Facebook password in order to log in on the web. [?]
- Embedded Browser OAuth Login:** Enabled (YES). Description: Enables browser control redirect uri for OAuth client login. [?]
- Valid OAuth redirect URLs:** A text input field containing "Valid OAuth redirect URLs."
- Login from Devices:** Enabled (YES). Description: Enables the OAuth client login flow for devices like a smart TV [?]



Розділ 18.

## **Аудіо і відео**

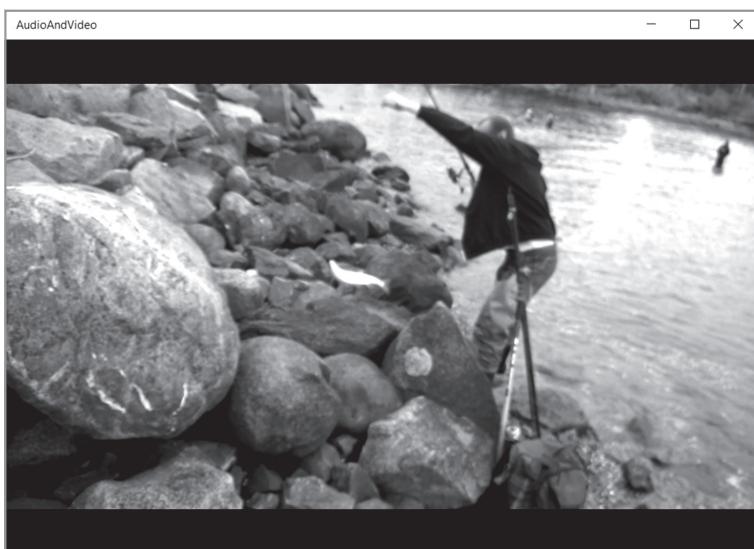
## Елементи керування медіаданими

### MediaElement

Найпростіший спосіб відобразити відео на екрані – це розмістити на сторінці елемент керування **MediaElement**. Він потрібен для всіх інших плеєрів, що можуть відтворювати аудіо і відео. Для відтворення вам просто необхідно розмістити елемент керування **MediaElement** на вашій сторінці і задати шлях до відео- чи аудіофайлу. Наприклад, відео вилову лосося можна відтворити за допомогою такого коду:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <MediaElement Source="Assets/salmon.mp4"></MediaElement>
</Grid>
```

Отже, лише один рядок коду – і ми можемо переглядати відео, використовуючи весь доступний простір у вікні програми:



Звичайно, це дуже простий код і насправді **MediaElement** містить багато методів, подій і властивостей. У цій темі ми поговоримо про деякі з них. Певно, почати варто з підтримуваних форматів і джерел відео.

Коли йшлося про Windows 8, ще був шанс описати в книжці всі доступні формати, а зараз це зовсім непросте завдання, адже тут треба враховувати ще

залежність від контейнеру і версії Windows (Desktop, IoT, Mobile, Xbox). Тому вам потрібно знати, що Windows підтримує Windows Media Video, зокрема Windows Media Video з Advanced profile і форматом MPEG, включно з H.265. Крім того, у вас не виникне проблем з адаптивною потоковою трансляцією. Детальніші відомості, звичайно, можна знайти в статтях MSDN з цієї тематики.

Говорячи про можливі джерела, слід зауважити, що в нашому прикладі ми використовували вбудований у програму файл, але ви можете звернутися до **MediaElement** для доступу до будь-яких файлів у локальному сховищі програми чи доступу за допомогою URL із протоколом **http**, **https** чи **mms**. І ви можете використовувати властивість **Source** для роботи з джерелом у XAML чи C#, але **MediaElement** має також метод **SetSource**, що дає змогу передавати посилання **IRandomAccessStream**. Цей підхід є пріоритетним, якщо ви працюєте з об'єктами **StorageFile**.

Давайте поговоримо про основні властивості, події і методи **MediaElement**. Крім властивості **Source**, **MediaElement** має такі основні властивості:

- **AutoPlay** – дає змогу відтворювати відео/аудіо автоматично, як тільки надано **Source**. Ця властивість за замовчуванням має істинне значення.
- **IsMuted** – завдяки цій властивості можна вимикати звук. Зазвичай ви будете використовувати цю властивість, коли реалізовуватимете функціональність для кнопки вимкнення звуку.
- **Volume** – дає можливість встановити чи отримати рівень гучності.
- **Balance** – якщо користувач має стереодинаміки, ця властивість дає змогу налаштувати співвідношення звуку в лівому та правому каналі.
- **Position** – дає змогу отримати або встановити поточну позицію відтворення фрагмента. Іноді, наприклад у разі роботи з потоковим відео, ця властивість недоступна. Завдяки наступній властивості ви можете зрозуміти, чи можна працювати з **Position** у поточному контексті.
- **CanSeek** – дає змогу зрозуміти, чи можна працювати з властивістю **Position**.
- **CanPause** – зазвичай, якщо ви не можете встановлювати позицію відтворення, призупиняти відтворення ви не можете також. Але інколи це все-таки можливо (наприклад, у разі відтворення реклами). Саме тому **MediaElement** містить окрім властивості **CanPause**.
- **CurrentState** – відображає поточний стан **MediaElement** і може містити значення з перелічуваного типу **MediaElementState: Buffering, Opening, Playing, Closed, Paused, Stopped**.
- **DownloadProgress** – повертає відсоток завантаження контенту.
- **DownloadProgressOffset** – містить зсув, що є початковою позицією для завантаження фрагмента. Зазвичай ця властивість не дорівнює нулю,

якщо користувач пересунув позицію відтворення всередині медіаелемента до контенту, який ще не завантажений.

- **Markers** – колекція маркерів. Детально ми будемо обговорювати маркери далі в цьому розділі.
- **NaturalDuration** – повертає загальну тривалість медіазапису.
- **NaturalVideoWidth** – повертає ширину відео.
- **NaturalVideoHeight** – повертає висоту відео.

Отже, властивостей **MediaElement** дуже багато. Але серед методів найпопулярнішими є **Play** і **Pause**.

Взагалі **MediaElement** підтримує багато подій, що виникають у таких випадках:

- **BufferingProgressChanged** – коли новий медіафрагмент буферизується;
- **CurrentStateChanged** – коли властивість **Position** змінюється;
- **DownloadProgressChanged** – коли новий фрагмент завантажується із сервера;
- **MarkerReached** – коли досягнуто маркера;
- **MediaEnded** – коли **MediaElement** завершує відтворення медіа;
- **MediaFailed** – коли щось трапилося із джерелом контенту;
- **MediaOpened** – коли джерело контенту було знайдено і пройшло валідацію.

Давайте реалізуємо простий інтерфейс, що буде використовувати деякі властивості і події. Звичайно, ми не будемо приділяти аж надто уваги зовнішній красі інтерфейсу. Зрозуміло також, що на практиці створювати власний інтерфейс доводиться не часто. Однак наведений нижче код дасть вам зможу зрозуміти **MediaElement** краще:

```
<StackPanel HorizontalAlignment="Left">
    <MediaElement Name="myMedia" Height="300"
        Source="Assets/salmon.mp4" Margin="5"
        AutoPlay="False"
        MediaOpened="myMedia_MediaOpened"
        MediaEnded="myMedia_MediaEnded" />
    <StackPanel Orientation="Horizontal">
        <TextBlock Name="durationText"
            Text="Duration: " Margin="5" />
        <TextBlock Text="{Binding Position.TotalSeconds,
            ElementName=myMedia, Mode=OneWay}" Margin="5" />
        <TextBlock Text="/" Margin="5" />
        <TextBlock Text="" Name="secondsText" Margin="5" />
    </StackPanel>
```

```

<StackPanel Orientation="Horizontal">
    <Button Content="Play" Name="playButton"
        Margin="5" Width="100" IsEnabled="False"
        Click="playButton_Click" />
    <Button Content="Pause" Name="pauseButton"
        IsEnabled="False" Margin="5" Width="100"
        Click="pauseButton_Click" />
    <Button Content="Stop" Name="stopButton"
        Margin="5" Width="100" IsEnabled="False"
        Click="stopButton_Click" />
    <CheckBox Content="Mute" Margin="5"
        IsEnabled="False" Name="muteBox"
        IsChecked="{Binding IsMuted,
            ElementName=myMedia, Mode=TwoWay}" />
</StackPanel>
<StackPanel Orientation="Horizontal" >
    <TextBlock Text="Volume:" Margin="5" />
    <Slider Name="volumeSlider" Minimum="0"
        Maximum="1" Width="200" IsEnabled="False"
        StepFrequency="0.1" Value="{Binding Volume,
            ElementName=myMedia, Mode=TwoWay}" />
</StackPanel>
</StackPanel>

```

Звичайно, вам потрібно реалізувати певний обробник події:

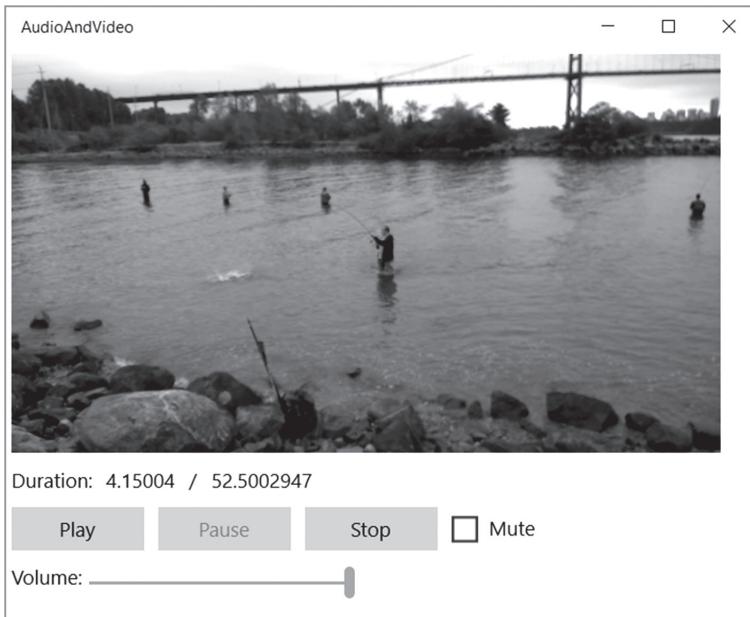
```

private void myMedia_MediaOpened(object sender, RoutedEventArgs e)
{
    volumeSlider.IsEnabled = true;
    playButton.IsEnabled = true;
    stopButton.IsEnabled = true;
    muteBox.IsEnabled = true;
    secondsText.Text = myMedia.NaturalDuration.TimeSpan.
        TotalSeconds.ToString();
}
private void playButton_Click(object sender, RoutedEventArgs e)
{
    playButton.IsEnabled = false;
    pauseButton.IsEnabled = true;
    myMedia.Play();
}
private void pauseButton_Click(object sender, RoutedEventArgs e)

```

```
{  
    pauseButton.IsEnabled = false;  
    playButton.IsEnabled = true;  
    myMedia.Pause();  
}  
private void stopButton_Click(object sender, RoutedEventArgs e)  
{  
    pauseButton.IsEnabled = false;  
    playButton.IsEnabled = true;  
    myMedia.Stop();  
}  
private void myMedia_MediaEnded(object sender, RoutedEventArgs e)  
{  
    pauseButton.IsEnabled = false;  
    playButton.IsEnabled = true;  
}
```

Врешті решт, запустивши цей код, ви побачите щось схоже на таке:



Ви зрозуміли, що реалізувати медіаплеєр із базовою функціональністю не дуже важко. Більше часу доведеться витратити на створення його дизайну та інтерфейсу.

Давайте обговоримо ще одну можливість **MediaElement** – маркери.

Маркери дають змогу вказувати певний момент часу в медіаконтенті і ви можете використовувати їх у багатьох сценаріях відтворення медіаданих. Наприклад, ви можете використовувати маркери для створення навчального контенту, де в разі досягнення маркера користувачі пройдуть тест чи побачать анімацію, або для відображення субтитрів.

У наведеному нижче прикладі ми використаємо маркери для демонстрації тексту, що пов'язаний із відео. Ось XAML-код:

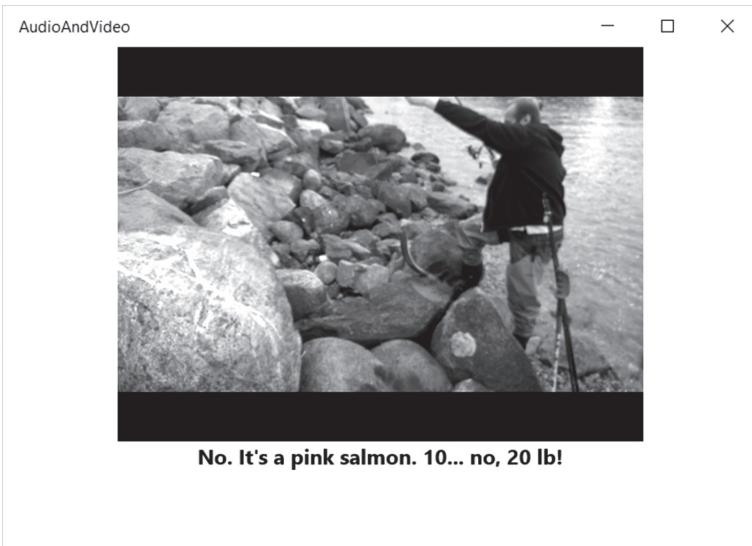
```
<StackPanel>
    <MediaElement Width="400" Height="300" Name="myMedia"
        Source="Assets/salmon.mp4"
        MediaOpened="myMedia_MediaOpened"
        MarkerReached="myMedia_MarkerReached" />
    <TextBlock Text="" HorizontalAlignment="Center"
        Name="subtitleText" FontSize="16" FontWeight="Bold" />
</StackPanel>
```

А ось C#-код:

```
public sealed partial class MainPage : Page
{
    Subtitle[] subtitles = new Subtitle[3];
    public MainPage()
    {
        InitializeComponent();
        Subtitle s = new Subtitle();
        s.text = "Lots of people in the water";
        s.time = new TimeSpan(0, 0, 0);
        subtitles[0] = s;
        s = new Subtitle();
        s.text = "Something strange. Is it a shark?";
        s.time = new TimeSpan(0, 0, 15);
        subtitles[1] = s;
        s = new Subtitle();
        s.text = "No. It's a pink salmon. 10... no, 20 lb!";
        s.time = new TimeSpan(0, 0, 20);
        subtitles[2] = s;
    }
    private void myMedia_MediaOpened(object sender,
        RoutedEventArgs e)
```

```
{  
    TimelineMarker t;  
    foreach (Subtitle s in subtitles)  
    {  
        t = new TimelineMarker();  
        t.Time = s.time;  
        t.Text = s.text;  
        myMedia.Markers.Add(t);  
    }  
}  
  
private void myMedia_MarkerReached(object sender,  
    TimelineMarkerRoutedEventArgs e)  
{  
    subtitleText.Text = e.Marker.Text;  
}  
}  
  
class Subtitle  
{  
    public TimeSpan time;  
    public String text;  
}
```

Запустивши цю програму, ви побачите відео і субтитри:



Звичайно, використовуючи елемент керування **MediaElement**, не обов'язково створювати заголовки «з нуля». Нижче ми продемонструємо, як використовувати наявні властивості і формати.

Елемент керування **MediaElement** не новий, але у Windows 10 до нього додалося декілька нових можливостей. На основі досвіду розробки попередніх медіаплеєрів можна назвати певні недоліки цього елемента керування у Windows 8.x:

- Проблема з адаптивним потоковим відео – для того, щоб почати з ним працювати, вам потрібно використовувати зовнішні бібліотеки, такі як Player Framework і Smooth Streaming SDK, тому що Windows 8.x SDK не підтримує вбудовані властивості для адаптивного потокового відео. Навіть якщо ви використовуєте ці бібліотеки, ви отримаєте лише підтримку Smooth Streaming.
- Немає підтримки існуючих форматів із закритими субтитрами – Windows 8.x SDK не підтримує закриті субтитри взагалі. SMPTE-TT і TTML були введені лише в Player Framework SDK.
- Немає змоги змінити шаблон для існуючого елемента керування медіаплеєра – Windows 8.x підтримував **SystemMediaTransportControls**, але можливостей щось змінити було небагато. Тож вам потрібно було створювати власний плеєр «з нуля» чи використовувати Player Framework з меншими можливостями для внесення змін.

Давайте подивимось, як Microsoft виправив ці недоліки в Universal Windows Platform.

## Адаптивне потокове відео

HTTP Live Streaming (HLS), Dynamic Adaptive Streaming поверх HTTP (DASH) і Microsoft Smooth Streaming – найбільш популярні технології для адаптивного потокового відео. Новий елемент керування **MediaElement** підтримує їх усі.

Якщо ви хочете випробувати, як це працює, краще скористатися обліковим записом Azure (є пробна безкоштовна версія) і створити під ним екземпляр Media Service. Media Service підтримує динамічне пакування, отже, ви можете завантажити в нього ваше відео і виконати кодування, використовуючи одну з доступних опцій для адаптивного потокового відео.

## Process

At least one streaming unit is required for dynamic packaging to Smooth, HLS, and MPEG-DASH. Without a streaming unit, only progressive download and static packaging are possible. Learn more about dynamic packaging.

### PROCESSOR

Azure Media Encoder



#### Presets for Adaptive Streaming (dynamic packaging)

- H264 Adaptive Bitrate MP4 Set 1080p
- H264 Adaptive Bitrate MP4 Set 720p
- H264 Adaptive Bitrate MP4 Set SD 16x9
- H264 Adaptive Bitrate MP4 Set SD 4x3
- H264 Adaptive Bitrate MP4 Set 1080p for iOS Cellular Only
- H264 Adaptive Bitrate MP4 Set 720p for iOS Cellular Only
- H264 Adaptive Bitrate MP4 Set SD 16x9 for iOS Cellular Only
- H264 Adaptive Bitrate MP4 Set SD 4x3 for iOS Cellular Only

#### Presets for Progressive Download

- H264 Broadband 1080p
- H264 Broadband 720p
- H264 Broadband SD 16x9
- H264 Broadband SD 4x3



#### Legacy Presets for Adaptive Streaming

- H264 Smooth Streaming 1080p
- H264 Smooth Streaming 720p
- H264 Smooth Streaming SD 16x9
- H264 Smooth Streaming SD 4x3
- H264 Smooth Streaming Windows Phone 7 Series

#### Other

- Encode with PlayReady content protection
- Playback on PC/Mac (via Flash/Silverlight)
- Playback via HTML5 (IE/Chrome/Safari)
- Playback on iOS devices and PC/Mac

Зверніть особливу увагу, що вам потрібно створити принаймні один блок для потокового відео. Як тільки відео буде закодовано, завдяки динамічному пакуванню ви зможете відразу відтворити його в будь-якому з попередньо вказанчих форматів (HLS, Dash, Smooth).

Давайте протестуємо, як працює DASH. Для цього вам потрібно лише додати **MediaElement** до сторінки XAML:

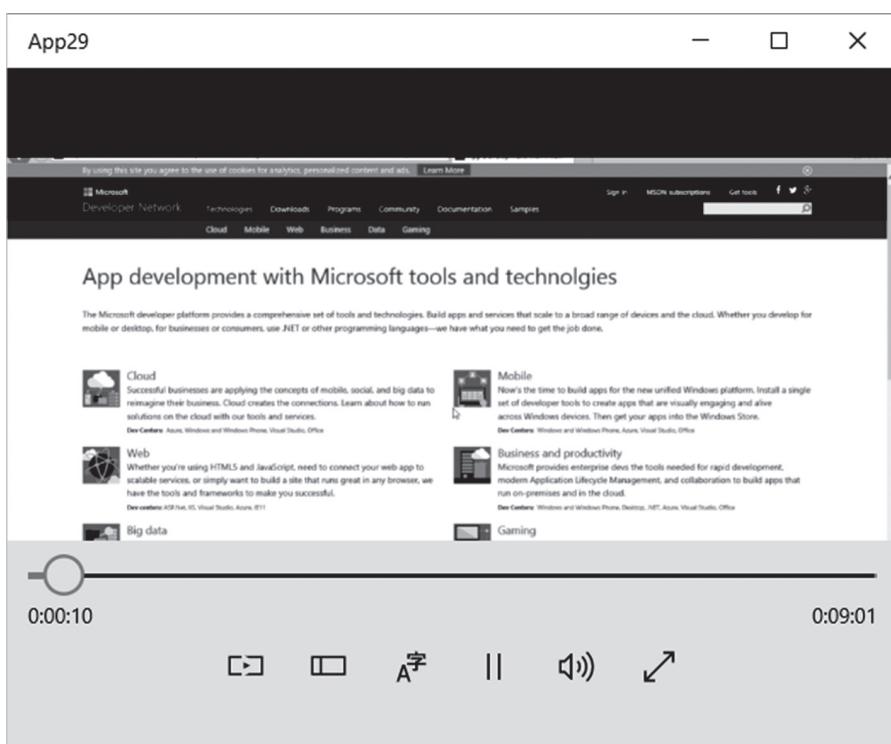
```
<MediaElement Name="media" AreTransportControlsEnabled="True" />
```

Щоб створити джерело адаптивного медіаконтенту, ви можете використати клас **AdaptiveMediaSource** таким чином:

```
AdaptiveMediaSourceCreationResult result = await  
AdaptiveMediaSource.CreateFromUriAsync(new Uri  
("http://testadaptive.streaming.mediaservices.windows.  
net/e1f03724-c228-4f86-9570-7321f1767fc5/Module%202.1_  
H264_4500kbps_AAC_und_ch2_128kbps.ism/Manifest(format=  
mpd-time-csf)", UriKind.Absolute));  
if (result.Status == AdaptiveMediaSourceCreationStatus.Success)  
{  
    var astream = result.MediaSource;  
    media.SetMediaStreamSource(astream);  
}
```

Відео відтворюється справно, а завдяки **MediaTransportControls** ми отримали чудовий інтерфейс для медіаплеєра:

## Windows 10 для C# розробників



## Captions

Завдяки новим елементам керування також легко додавати субтитри. Просто використайте нижчезаведений код для пов'язування TTM-файлу з наявним адаптивним потоком:

```
AdaptiveMediaSourceCreationResult result = await
AdaptiveMediaSource.CreateFromUriAsync(new Uri
("http://testadaptive.streaming.mediaservices.windows.
net/e1f03724-c228-4f86-9570-7321f1767fc5/Module%202.1_"
H264_4500kbps_AAC_und_ch2_128kbps.ism/Manifest
(format=mpd-time-csf)", UriKind.Absolute));
if (result.Status == AdaptiveMediaSourceCreationStatus.Success)
{
    var astream = result.MediaSource;
    var ttmSource = TimedTextSource.CreateFromUri(new
        Uri("ms-appx:///assets/captions.ttm"));
    var mediaSource=MediaSource.CreateFromAdaptiveMediaSource
```

```
(astream);  
mediaSource.ExternalTimedTextSources.Add(ttmSource);  
var mediaElement = new MediaPlaybackItem(mediaSource);  
media.SetPlaybackSource(mediaElement);  
}
```

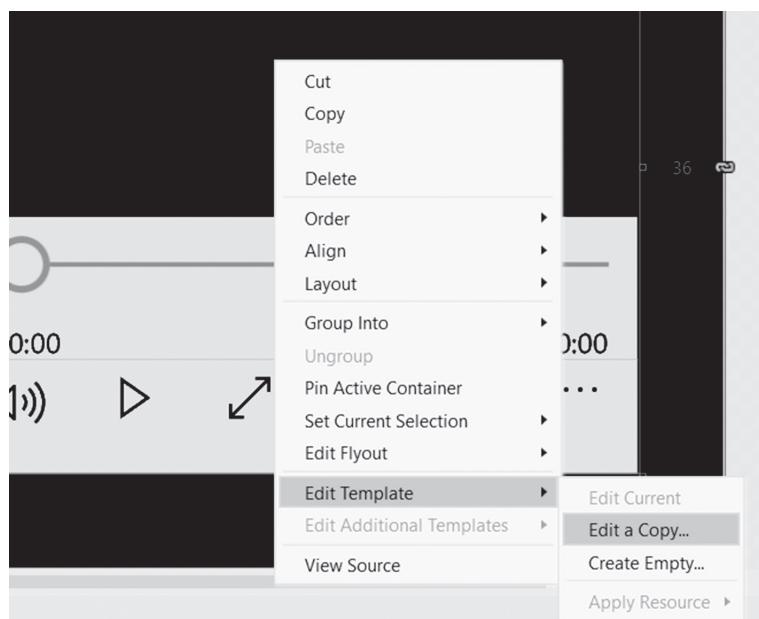
Як бачите, ми використовуємо клас **TimedTextSource** для створення джерела контенту на основі файлу з субтитрами. Щоб пов'язати субтитри із джерелом медіаконтенту, ми використовуємо клас **MediaSource**, а клас **MediaPlaybackItem** – для підготовки джерела для **MediaElement**.

## Шаблон для Media

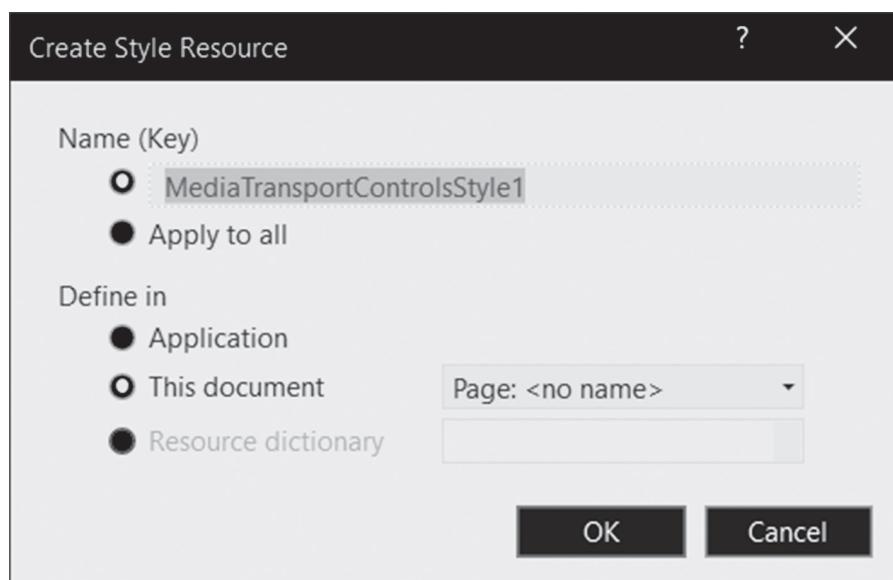
Починаючи з Windows 10, клас **MediaTransportControls** має власні стани і шаблон відповідно до нового дизайну. Це дає змогу змінювати будь-що, модифікуючи наявний шаблон XAML. Отже, вам не потрібно створювати кнопки «з нуля», реалізувати логіку та ін. Є три способи отримати наявний шаблон для **MediaTransportControl**: відвідати MSDN, знайти файл **generic.xaml** на комп'ютері чи використати Blend.

Файл **generic.xaml** ви можете знайти в папці **C:\Program Files (x86)\Windows Kits\10\DesignTime\CommonConfiguration\Neutral\UAP\10.0.10069.0\Generic**. Просто відкрийте **generic.xaml** і знайдіть у ньому шаблон для **MediaTransportControls**.

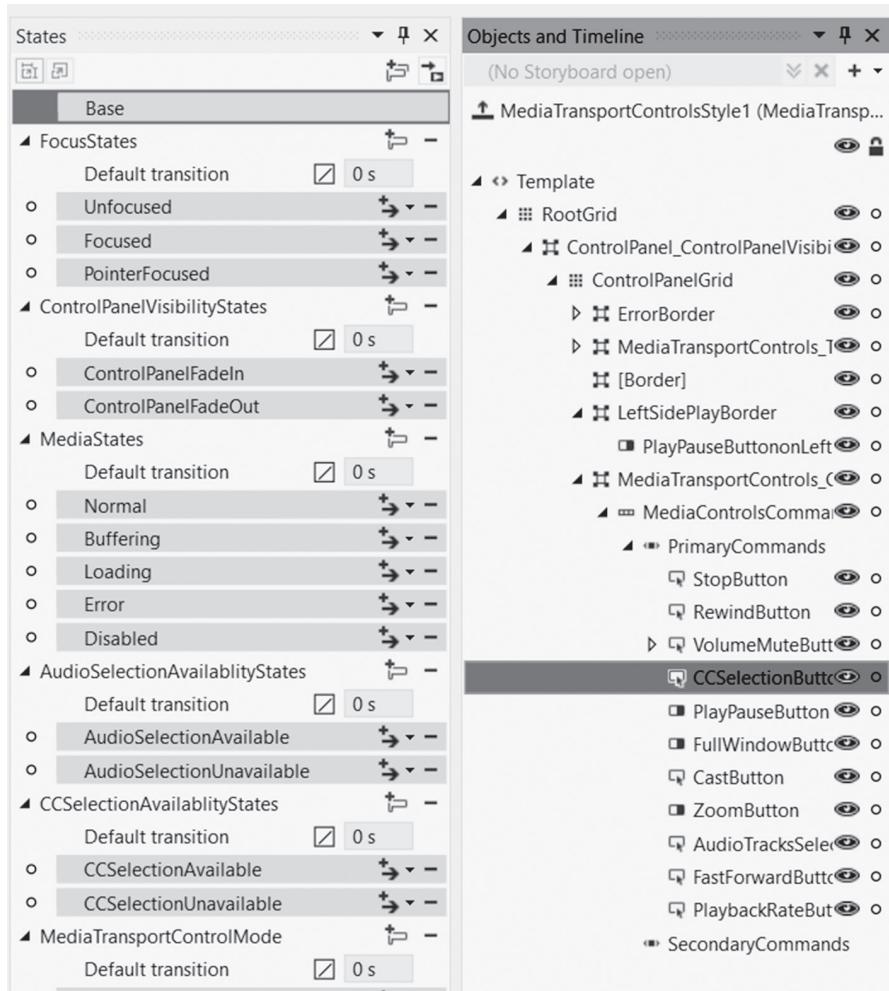
Врешті-решт, ви можете вибрати шаблон, використовуючи інструмент Blend. Рекомендуємо цей варіант, тому що в такий спосіб легко не лише створити копію шаблону, але й модифікувати його. Щоб вибрати шаблон для **MediaTransportControls**, вам потрібно створити новий проект (чи відкрити наявний) в Blend і додати **MediaTransportControls** до сторінки. У контекстному меню виберіть пункти **Edit Template -> Edit a Copy**.



Blend запропонує вибрати місце розташування та ім'я для нового стилю. Отже, вкажіть ці дані та натисніть **OK**.



Відразу після цього ви можете відкрити XAML і модифікувати шаблон чи використати для перегляду та редагування потужний редактор в Blend.

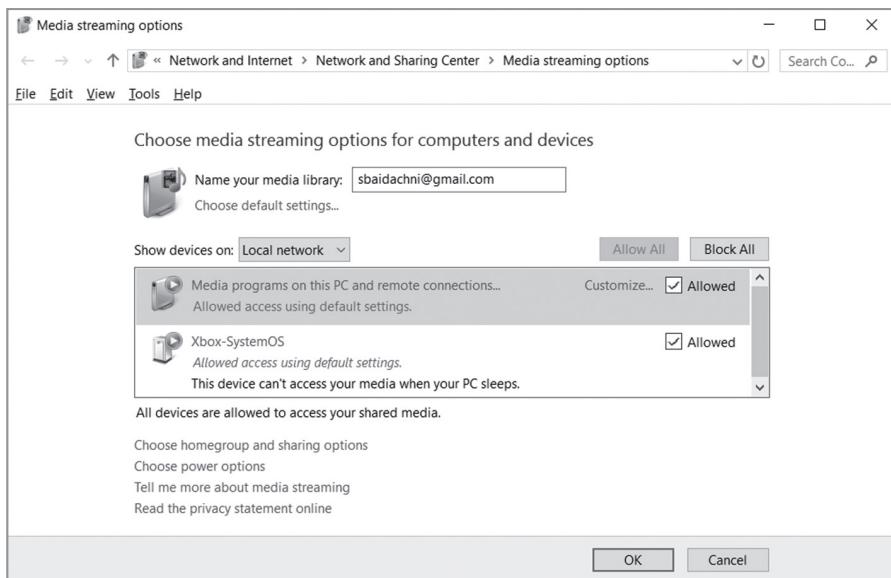


## Media casting

Media casting – це дуже важлива функціональність, яку дуже легко реалізувати. Однак багато розробників забивають про неї, коли створюють власний медіаплеєр. Media casting – це спеціальна можливість Windows 10, що дає змогу відтворювати медіаконтент потоком на зовнішніх пристроях. Майже всі сучасні

телевізори, консолі, медіаплеєри підтримують такі протоколи, як UPnP і дають вам змогу надсилати медіа з комп'ютера. Наприклад, нас задовольняє якість відтворення певного технічного контенту на комп'ютері, але якби була можливість переглянути його на Xbox на великому екрані, ми б цим неодмінноскористалися. Тож як реалізувати media casting у ваших програмах?

Перш за все вам потрібно переконатися, що ваш комп'ютер дає змогу надсилати медіафайли на віддалений пристрій. Наприклад, ця опція на Windows 10 може бути вимкнута й відразу буде незрозуміло, чому неможливо знайти жоден пристрій. У такому разі вам потрібно відкрити розділ **Media streaming options** і перевірити, чи все гаразд.



Щоб почати відтворювати медіаконтент, у багатьох випадках вам не потрібно робити нічого. Наприклад, якщо ви вирішили використовувати елемент керування для транспортування медіа, який ми обговорювали раніше, media casting буде доступним за замовчуванням.



Натиснувши кнопку **Cast to Device**, ви зможете побачити діалогове вікно зі списком усіх доступних пристрій. Просто виберіть пристрій і почніть трансляцію вашого медіаконтенту:



У нашому випадку є Xbox One, де потрібно запустити Xbox Video чи Media Player – й усе інше Xbox зробить автоматично.

Звичайно, якщо ви вирішите створити власний інтерфейс, це також можна зробити. Universal Windows Platform містить простір імен **Windows.Media.Casting**, у якому є численні класи, такі як **CastingDevice**, **CastingConnection**, **CastingSource**, **CastingDevicePicker** тощо. Ви можете використовувати всі ці класи для реалізації власного медіаінтерфейсу і навіть власного діалогового вікна для вибору пристрою.

Ми не будемо створювати власне діалогове вікно, але давайте поглянемо, як використовувати наявне вікно для вибору – **CastingDevicePicker**. Лише додамо кнопку на сторінку, де розташовано **MediaElement**, і реалізуємо обробник для події **Click**:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    CastingDevicePicker picker = new CastingDevicePicker();
    picker.CastingDeviceSelected += Picker_CastingDeviceSelected;
    picker.Show(new Rect(x,y,100,100));
}
```

Ми просто створили об'єкт класу **CastingDevicePicker** і використали метод **Show**. Завдяки цьому методу ви можете відображати недалеко від краю заданого прямокутника стандартне діалогове вікно для вибору пристрою. Цей варіант достатньо вдалий і підтримує дві події, що виникають, коли вибрано пристрій чи знято фокус із діалогового вікна. У нашому випадку ми вирішили реалізувати обробник події **CastingDeviceSelected**, що стається, коли користувач обирає пристрій:

```
private async void Picker_CastingDeviceSelected(CastingDevice
Picker sender, CastingDeviceSelectedEventArgs args)
{
    await Dispatcher.RunAsync(Windows.UI.Core.
CoreDispatcherPriority.Normal, async () =>
{
    CastingConnection connection =
args.SelectedCastingDevice.CreateCastingConnection();

    await connection.RequestStartCastingAsync
(myMedia.GetAsCastingSource());

    myMedia.Position = TimeSpan.FromSeconds(0);
});
}
```

Як видно, це не дуже складний обробник події. Вам просто потрібно запустити код в потоці інтерфейсу. Використовуючи обрані пристрої, слід створити з'єднання і надіслати запит на перегляд. Крім того, ми змінили позицію відео на 0, щоб переглядати його з початку.

## Media Editing and Transcoding

Universal Windows Platform підтримує два найкорисніші простори імен для редагування і перекодування медіа: **Windows.Media.Editing** і **Windows.Media.Transcoding**, де ви можете знайти всі потрібні класи.

У разі перекодування найбільш важливі класи – це **MediaTranscoder**, що підтримує перекодування медіафайлів з одного формату в інший, і **MediaEncodingProfile** з простору імен **Windows.Media.MediaProperties**, що підтримує декілька наперед визначених профілів для кодування. Тому, якщо ви хочете закодувати файл чи дані з потоку, вам потрібно створити профіль і використати **MediaTranscoder** для кодування.

Щоб створити профіль, ми можемо використати такий код:

```
var profile = MediaEncodingProfile.  
CreateMp4(VideoEncodingQuality.HD1080p);
```

Ви також можете створити новий профіль за допомогою методу **CreateFromFileAsync** чи **CreateFromStreamAsync** або використати конструктор **MediaEncodingProfile** та ініціалізувати всі необхідні властивості для створення власного профілю «з нуля».

Як тільки профіль створено, вам потрібно створити об'єкт класу **MediaTranscoder** і викликати метод **PrepareFileTrascodeAsync** (чи **PrepareStreamTranscodeAsync**), передавши вхідний файл, результатуючий файл і профіль. Якщо цей метод поверне **CanTranscode**, ви можете просто викликати метод **TranscodeAsync**.

Завдяки подіям **Progress** і **Completed** ви можете визначити поточний стан виконання процесу, а також чи завершено його.

Для редагування можна використати класи **MediaClip** і **MediaComposition** з простору імен **Windows.Media.Editing**. Перший клас відображає медіакліп і підтримує властивості обтинання, а другий містить колекцію кліпів, з яких може бути згенеровано фінальний медіафайл.



Розділ 19.

## **API камери**

## Використання діалогового вікна CameraCaptureUI

Universal Windows Platform підтримує декілька способів інтеграції вашої програми з камерою. Ви можете використовувати API для отримання доступу до відеотрансляції, створити свій інтерфейс для роботи з камерою або використати вбудовані елементи керування для того, щоб спростити свій код. У цій частині ми обговоримо всі доступні способи. Тож почнемо з найпростішого способу – використання діалогового вікна **CameraCaptureUI**.

Давайте реалізуємо простий інтерфейс, що містить кнопку і елемент керування **Image**. Ми будемо використовувати кнопку для відкриття діалогового вікна **CameraCaptureUI**. Завдяки цьому вікну за допомогою нашої програми можна буде зробити фото і відобразити його, використовуючи елемент керування **Image**.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Vertical">
        <Button Content="Take a Picture" Click="Button_Click" />
        <Image Name="photo" Height="400" Width="400"/>
    </StackPanel>
</Grid>
```

В коді нижче ви можете бачити, що як тільки користувач натисне кнопку, діалогове вікно **CameraCaptureUI** буде відкрито.

```
private async void TakePicture()
{
    CameraCaptureUI camera = new CameraCaptureUI();
    camera.PhotoSettings.Format = CameraCaptureUIPhotoFormat.Png;
    camera.PhotoSettings.MaxResolution =
        CameraCaptureUIMaxPhotoResolution.HighestAvailable;
    StorageFile file = await camera.CaptureFileAsync
        (CameraCaptureUIMode.Photo);
    if (file != null)
    {
        BitmapImage btn = new BitmapImage();
        btn.SetSource(await file.OpenAsync(FileAccessMode.Read));
        photo.Source = btn;
    }
}
private void Button_Click(object sender, RoutedEventArgs e)
```

```
{
    TakePicture();
}
```

**CameraCaptureUI** підтримує не лише фото, але й відео. І, залежно від того, обираєте ви фото чи відео, можете модифікувати властивість **PhotoSettings** чи **VideoSettings**. У нашому випадку потрібно використовувати **PhotoSettings** і найуживаніші параметри – **Format** і **MaxResolution**.

Як тільки ми завершимо налаштування, можна буде викликати **CaptureFileAsync**, що активує стандартне діалогове вікно Windows. Зауважте, що вам не потрібно нічого спеціально задавати в маніфесті, тому що у вас немає безпосереднього доступу до камери і користувачі будь-коли можуть скасувати задані в діалоговому вікні налаштування. Саме тому вам потрібно перевірити, чи повертає діалогове вікно певне зображення. Якщо об'єкт класу **StorageFile** – не **null**, ви можете зберегти його в папці програми, продемонструвати на екрані чи отримати доступ до масиву пікселів.

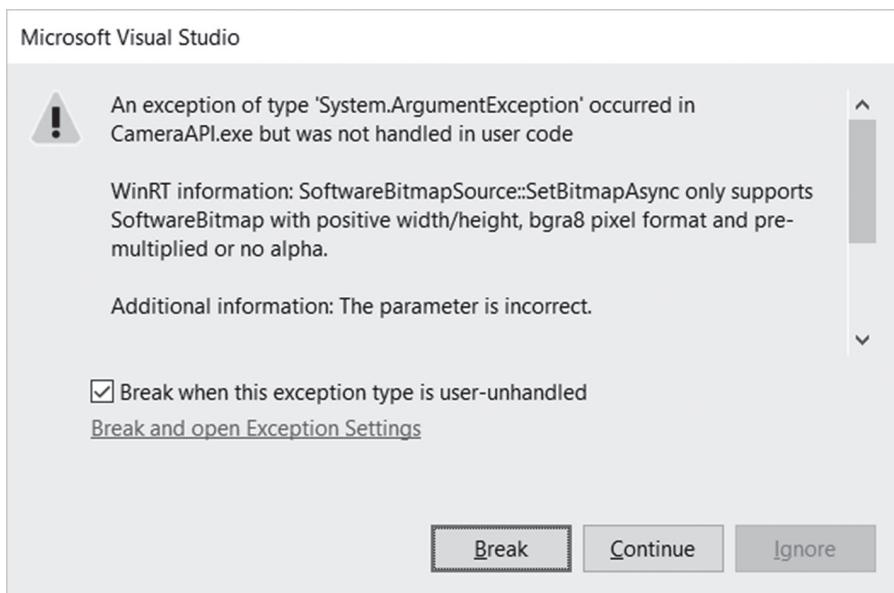
Для подання зображення використаємо клас **BitmapImage**, але якщо ви використовуєте **SoftwareBitmap**, можна змінити код у такий спосіб:

```
if (file != null)
{
    IRandomAccessStream stream =
        await file.OpenAsync(FileAccessMode.Read);
    BitmapDecoder decoder =
        await BitmapDecoder.CreateAsync(stream);
    SoftwareBitmap sBitmap =
        await decoder.GetSoftwareBitmapAsync();

    SoftwareBitmapSource bitmapSource = new SoftwareBitmapSource();
    await bitmapSource.SetBitmapAsync(sBitmap);

    photo.Source = bitmapSource;
}
```

Але цей код не працює, і якщо ви будете його налагоджувати, ви побачите таке повідомлення:



Отже, проблема у форматі й нам потрібно додати декілька рядків коду для перетворення наявного об'єкта класу **SoftwareBitmap** до типу, що задовольняє вимоги. Код набуде такого вигляду:

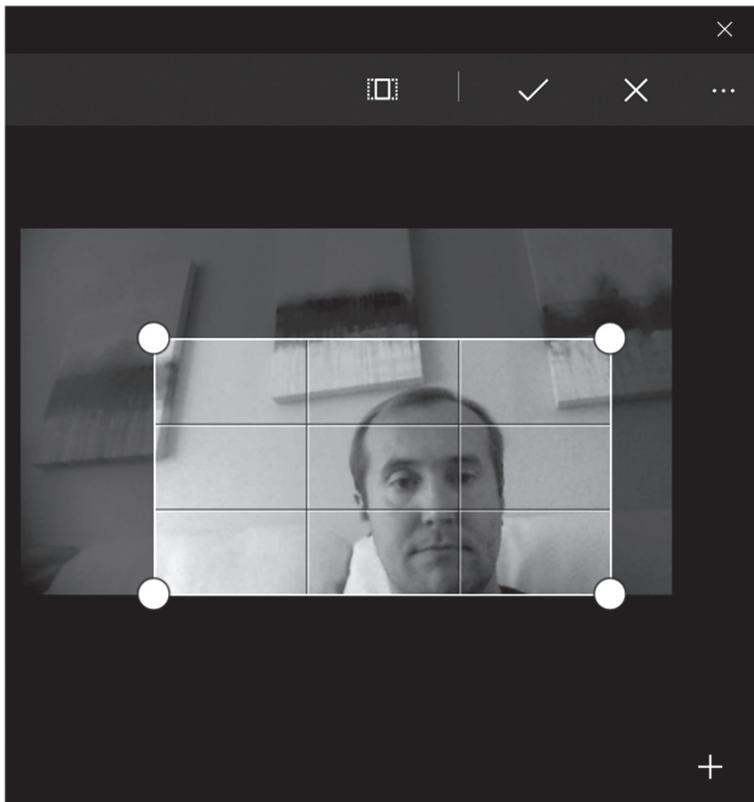
```
if (file != null)
{
    IRandomAccessStream stream =
        await file.OpenAsync(FileAccessMode.Read);
    BitmapDecoder decoder =
        await BitmapDecoder.CreateAsync(stream);
    SoftwareBitmap sBitmap =
        await decoder.GetSoftwareBitmapAsync();

    var sBitmapConverted = SoftwareBitmap.Convert(sBitmap,
        BitmapPixelFormat.Bgra8,
        BitmapAlphaMode.Premultiplied);

    SoftwareBitmapSource bitmapSource = new SoftwareBitmapSource();
    await bitmapSource.SetBitmapAsync(sBitmapConverted);

    photo.Source = bitmapSource;
}
```

Якщо ви запустите цей код і натиснете кнопку **Take a Picture**, то побачите діалогове вікно, однак відразу після цього відобразиться рамка для обтиснання зображення:



Це вікно добре використовувати тоді, коли ви хочете попросити користувача замінити фото в профілі чи в подібних ситуаціях. Але інколи ця стандартна поведінка не бажаною. Її легко змінити, використовуючи властивості **PhotoSettings** та **AllowCropping** – просто надайте їм значення **false**, щоб скасувати функціональність обтиснання.

Давайте модифікуємо інтерфейс, щоб отримувати відео замість зображення:

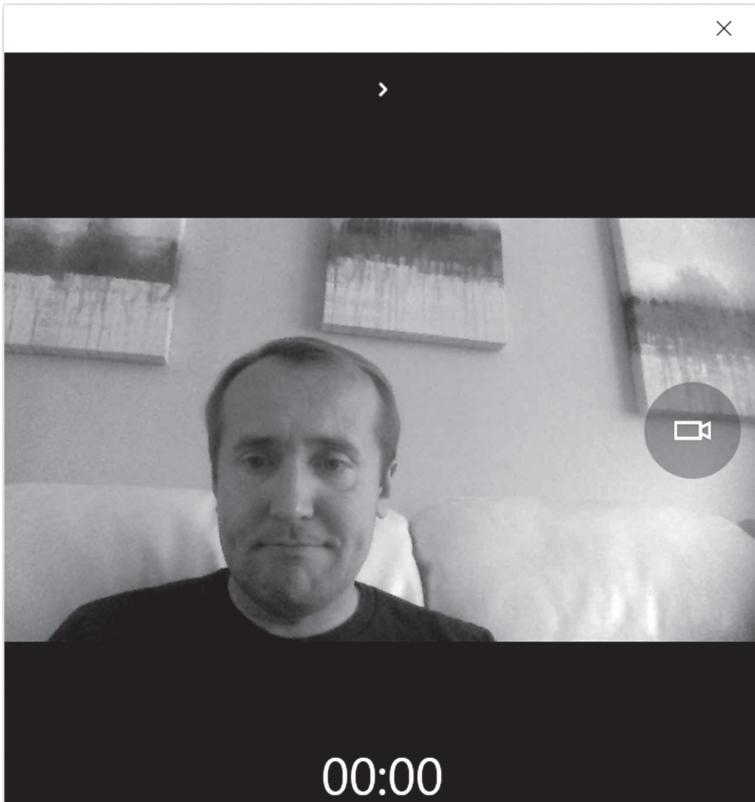
```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Vertical">
        <Button Content="Take a Video" Click="Button_Click" />
        <MediaElement x:Name="video" Width="320" Height="240" />
```

```
        AreTransportControlsEnabled="True"/>
    </StackPanel>
</Grid>
```

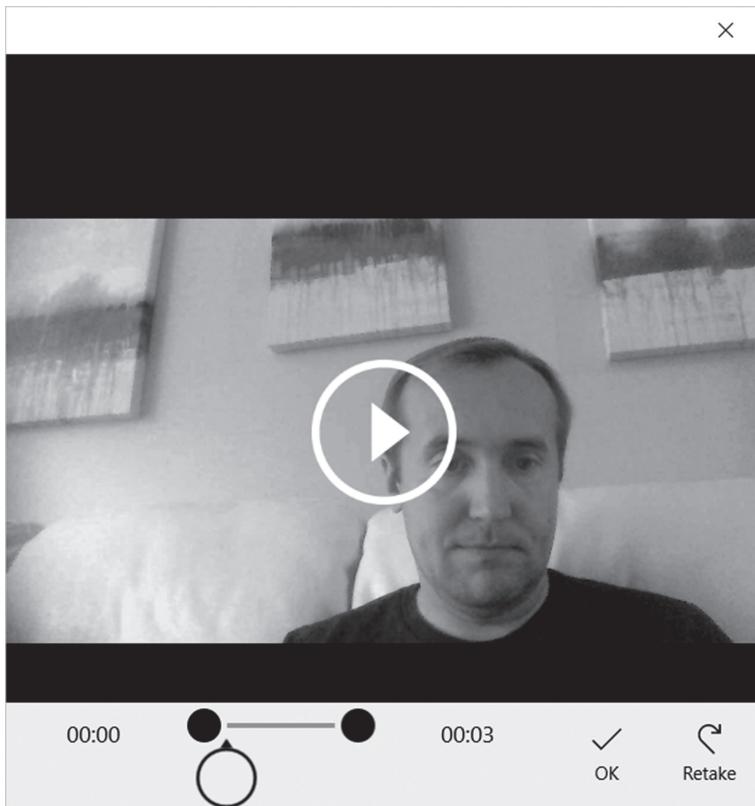
Ми змінили елемент керування **Image** на **MediaElement**, щоб відображати відео. Отже, це вимагає небагатьох зміг в інтерфейсі. Але нам потрібно ще трохи переписати логіку:

```
private async void TakePicture()
{
    CameraCaptureUI camera = new CameraCaptureUI();
    camera.VideoSettings.Format = CameraCaptureUIVideoFormat.Wmv;
    camera.VideoSettings.MaxResolution =
        CameraCaptureUIMaxVideoResolution.HighestAvailable;
    StorageFile file = await camera.CaptureFileAsync
        (CameraCaptureUIMode.Video);
    if (file != null)
    {
        IRandomAccessStream stream =
            await file.OpenAsync(FileAccessMode.Read);
        video.SetSource(stream, "");
    }
}
private void Button_Click(object sender, RoutedEventArgs e)
{
    TakePicture();
}
```

Як бачите, ідея та сама, і якщо ви запустите Windows-програму, з'явиться діалогове вікно, що дасть вам змогу записувати відео:



І переглядати його, перш ніж надсилати до програми:



## Media Capture API

Зараз ми вже знаємо, як отримувати фото і відео, використовуючи вбудований діалог для камери, але якщо ви бажаєте створити власний API, вам потрібно використовувати інший підхід – Media Capture API. Давайте поглянемо на відповідні класи, і дізнаємося, як їх використовувати.

Оскільки ми хочемо отримати прямий доступ до камери, першим кроком буде змінення маніфесту. Ви можете відкрити маніфест у режимі дизайну та активувати можливості **Microphone** і **Webcam**:

<b>Capabilities:</b>	<b>Description:</b>
<input type="checkbox"/> All Joyn <input type="checkbox"/> Blocked Chat Messages <input type="checkbox"/> Chat Message Access <input type="checkbox"/> Code Generation <input type="checkbox"/> Enterprise Authentication <input checked="" type="checkbox"/> Internet (Client) <input type="checkbox"/> Internet (Client & Server) <input type="checkbox"/> Location <input checked="" type="checkbox"/> Microphone <input type="checkbox"/> Music Library <input type="checkbox"/> Objects 3D <input type="checkbox"/> Phone Call <input type="checkbox"/> Pictures Library <input type="checkbox"/> Private Networks (Client & Server) <input type="checkbox"/> Proximity <input type="checkbox"/> Removable Storage <input type="checkbox"/> Shared User Certificates <input type="checkbox"/> User Account Information <input type="checkbox"/> Videos Library <input type="checkbox"/> VOIP Calling <input checked="" type="checkbox"/> Webcam	Provides access to the microphone's audio feed, which allows the app to record audio from connected microphones. <a href="#">More information</a>

API Media Capture містить багато різних класів, але найважливіший з них – клас **MediaCapture**. Завдяки цьому класу можна створити об'єкт, що може отримати доступ до відео- і аудіопотоків. Отже, найпершим завданням є підготовка об'єкта **MediaCapture**. Ми можемо це зробити, використовуючи такий код:

```
private async void VideoPreview()
{
    var devices = await DeviceInformation.
        FindAllAsync(DeviceClass.VideoCapture);
    var device = devices.FirstOrDefault();
    if (device==null)
    {
        throw new Exception("No camera device");
    }

    var media = new MediaCapture();

    var settings = new MediaCaptureInitializationSettings();
    settings.StreamingCaptureMode = StreamingCaptureMode.Video;
    settings.VideoDeviceId = devices[0].Id;

    try
    {
        await media.InitializeAsync(settings);
    }
```

```
        catch (Exception ex)
    {
        throw new Exception("Access denied", ex);
    }
    capturePreview.Source = media;
    await media.StartPreviewAsync();
}
```

Перш ніж починати запис відео чи аудіо, потрібно перевірити, чи доступний призначений для цього пристрій. Це можна зробити за допомогою класу **DeviceInformation**. Він дає змогу знайти багато різновидів пристроїв, але нам потрібні лише пристрої **VideoCapture** та/або **AudioCapture**. Звичайно, користувач може мати декілька камер чи мікрофонів, тому зазвичай вам потрібно буде створити інтерфейс, що дає користувачу можливість вибрати потрібний пристрій. Крім того, ви можете вибрати пристрій, властивості **IsDefault** якого надано значення **true**. Але в нашому випадку ми просто обираємо перший пристрій у списку. Звичайно, потрібно перевірити, чи доступні якісь пристрої взагалі.

До того ж, крім властивості **IsDefault**, клас **DeviceInformation** має властивість **EnclosureLocation**, що може допомогти вибрати відеопристрій залежно від його розташування. Наприклад, ви можете перевірити, розміщено камеру на задній, чи на передній панелі пристрою.

```
if (device.EnclosureLocation.Panel ==
Windows.Devices.Enumeration.Panel.Back)
{
    //doing something
}
```

Як тільки ми виберемо камеру, можемо почати працювати з класом **MediaCapture**. Є три окремих завдання: створити об'єкт класу, створити об'єкт, що містить налаштування, та ініціалізувати об'єкт **MediaCapture**, використовуючи створені налаштування. Звичайно, перше завдання дуже легке. Для підготовки налаштувань UWP підтримує клас **MediaCaptureInitializationSettings**, що дає змогу ініціалізувати режим запису та ідентифікатор пристрою. Останнім кроком буде виклик методу **InitializeAsync** класу **MediaCapture** і це дуже важливий крок, оскільки якщо користувач не надасть дозвіл вашій програмі, це приведе до помилки. Отже, нам потрібно викликати цей метод в блоці **try** і бути готовими відразу обробити помилку.

Ми вже готові починати запис за допомогою **StartRecordToStreamAsync** та **StartRecordToFileAsync**, але зазвичай користувачі бажають спочатку

побачити попередній перегляд відео. Звичайно, попередній перегляд не повинен використовувати багато ресурсів GPU (особливо для HD-камери). Щоб отримати попередній перегляд, UWP підтримує спеціальний клас **CaptureElement**. Ви можете розмістити цей елемент керування будь-де та використовувати його для попереднього перегляду:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <CaptureElement Name="capturePreview"
        Stretch="UniformToFill"></CaptureElement>
</Grid>
```

Як тільки елемент керування розміщено, ми можемо ініціалізувати його і почати попередній перегляд, використовуючи два рядки коду:

```
capturePreview.Source = media;
await media.StartPreviewAsync();
```

Але як тільки користувач натисне кнопку для запису, ви можете починати запис.

Як уже зазначалося, об'єкт **MediaCapture** створити легко. Ви витратите більше часу на створення інтерфейсу користувача і вам потрібно буде здійснювати обробку багатьох різних ситуацій. Наприклад, відразу після створення об'єкту класу **MediaCapture** варто обробляти подію **RecordLimitationExceeded**, завдяки якій можна уникати проблем із пам'яттю і часом відтворення відео: щойно буде досягнуто граничних значень, записування потрібно зупинити і повідомити про це користувача.



Розділ 20.

## **Розпізнавання мови i Cortana**

Якщо відверто, ми не впевнені, чи це вдала ідея розмовляти зі своїм комп’ютером, але ситуації бувають різні. Якщо ви збираєтесь рухати курсор миші голосом, мабуть, це не дуже зручно, але, наприклад, є багато варіантів використання голосових механізмів, якщо йдеться про Інтернет речей. У деяких випадках ви вже використовували голос для керування пристроями, наприклад, GPS-навігатором. У GPS голос використовується в обох напрямках, адже сучасні GPS-системи повідомляють вам напрямок руху і стан доріг, використовуючи технології озвучення тексту. Надрукувати адресу місця призначення під час руху ще важче, ніж стежити за самим рухом, однак зовсім неважко назвати її вголос. А для цього вже знадобиться технологія розпізнавання мовлення. Тому представники Microsoft часто починають презентації з технології Cortana, яка дає змогу використовувати пристрой по-новому і відкриває нові горизонти для розробників.

Давайте поглянемо, як використовувати розпізнавання мовлення для UAP-програм. Ми будемо обговорювати класи, призначені для озвучення тексту, класи для розпізнавання мовлення, а також нові можливості Cortana.

## Озвучення тексту

Для того, щоб перетворити текст на мову, можна скористатися класом **SpeechSynthesizer** з простору імен **Windows.Media.SpeechSynthesis**. У найпростішому випадку код може бути таким:

```
private async void TalkSomething(string text)
{
    SpeechSynthesizer synthesizer = new SpeechSynthesizer();

    SpeechSynthesisStream synthesisStream =
        await synthesizer.SynthesizeTextToStreamAsync(text);

    media.AutoPlay = true;
    media.SetSource(synthesisStream, synthesisStream.ContentType);
    media.Play();
}
```

У перших двох рядках ми створили об’єкт **SpeechSynthesizer** і використали метод **SynthesizeTextToStreamAsync**, щоб отримати посилання на потік, який повинен містити аудіо. Щоб відтворити аудіо в програмі, можна використати об’єкт **MediaElement**, який ви можете розмістити у будь-якому місці сторінки. Цей код дуже простий і вам не потрібно запитувати ніяких дозволів, щоб його виконати. Але якщо ви збираєтесь використовувати службу озвучення тек-

сту у більш складних випадках, знадобиться більше часу. Наприклад, цей код не дає змогу дізнатися, якою мовою відбувається озвучення чи як вимовляти складний текст.

Почнімо з мови. Оскільки ваша програма може використовувати лише встановлені мови, не варто припускати, що комп'ютер користувача розмовлятиме, скажімо, російською чи іспанською. Тому якщо ви плануєте використовувати ще якісь мови, окрім англійської, краще перевірити, чи ця мова доступна на пристрої. Якщо ви використовуєте англійську, варто перевірити, чи вибрано її за замовчуванням, оскільки у деяких випадках це може бути і не так. Для цього можна використати такі статичні властивості класу **SpeechSynthesizer**:

- **DefaultVoice** – надає інформацію про стандартний голос для вашої програми, якщо ви не встановили інший;
- **AllVoices** – дає доступ до списку всіх голосів у системі. Ви можете використовувати LINQ чи індексатор, щоб знайти голос, який підходить.

Зауважте, що якщо ви шукаєте певну мову, то можете знайти декілька голосів у списку, оскільки голос – це не лише мова. Наприклад, у вашій системі встановлено два голоси за замовчуванням і обидва англійською, але перший – це чоловічий голос, а другий – жіночий. Властивості **DefaultVoices** і **AllVoices** дають змогу працювати з об'єктами **VoiceInformation**, які містять всю потрібну інформацію як, наприклад, **Language**, **DisplayName** і **Gender**.

Якщо ви хочете перевірити мову, можете використати такий код:

```
var list = from a in SpeechSynthesizer.AllVoices
           where a.Language.Contains("en")
           select a;

if (list.Count() > 0)
{
    synthesizer.Voice = list.Last();
}
```

У класі **SpeechSynthesizer** можна знайти два методи, **SynthesizeTextToStreamAsync** і **SynthesizeSsmlToStreamAsync**. Завдяки другому методу можна реалізувати складніший сценарій озвучення тексту. Він підтримує мову Speech Synthesis Markup Language (SSML), що дає змогу конструювати складніші речення і визначати, як їх правильно вимовляти. Звичайно, SSML – це текстовий формат, що базується на XML, тож ви легко можете створити свій чи модифікувати наявний SSML-код.

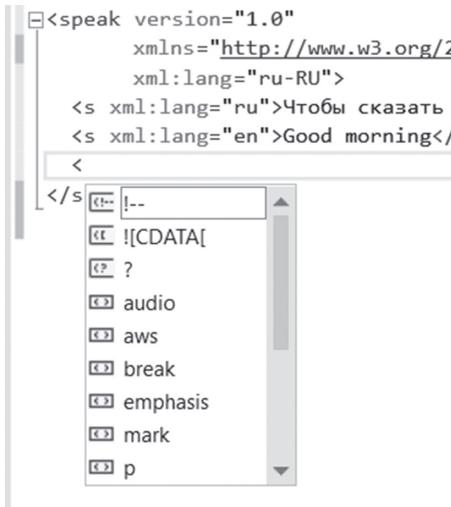
Розглянемо деякі елементи SSML.

- **speak** – кореневий елемент для SSML, що також дає змогу встановити мову за замовчуванням.
- **audio** – дає змогу включити в промову аудіофайл, наприклад, ефекти, музику тощо.
- **break** – ви можете задати паузи, використовуючи цей елемент. Він має певні атрибути, такі як **duration** і **strength**.
- **p** чи **s** – дає змогу визначити абзаци, що мають іншу мову. Завдяки цьому елементу ви можете озвучувати текст на різних мовах, що підтримуються.
- **prosody** – дає змогу встановити гучність.
- **voice** – дає можливість встановити наперед визначені мови чи атрибути, такі як стать.

Ось короткий приклад SSML-файлу, що поєднує речення двома мовами:

```
<speak version="1.0"
      xmlns="http://www.w3.org/2001/10/synthesis"
      xml:lang="ru-RU">
  <s xml:lang="ru">Чтобы сказать добрый день по-английски,
    произнесите</s>
  <s xml:lang="en">Good morning</s>
</speak>
```

Якщо ви плануєте створювати SSML в Visual Studio, вам потрібно використати шаблон XML, визначити елемент **speak** і Visual Studio надаватиме підтримку Intellisense і для SSML.



Для того, щоб використовувати SSML-файл, вам потрібно завантажити його як рядок і передати методу **SynthesizeSsmIAsStreamAsync**.

## Розпізнавання мовлення

Ми уже знаємо, як перетворювати текст на мову, і настав час обговорити пропоновану задачу.

Universal Application Platform підтримує простір імен **Windows.Media.SpeechRecognition** і декілька способів розпізнавання мовлення. Ви можете визначити вашу власну граматику, використовувати наявну граматику чи граматику для веб-пошуку. У багатьох випадках ви будете використовувати клас **SpeechRecognizer**. Давайте поглянемо, як використовувати цей клас у різних сценаріях.

Як і клас **SpeechSynthesizer**, **SpeechRecognizer** має деякі статичні властивості, що дають можливість зрозуміти доступні для розпізнавання мови. Перша властивість – це **SystemSpeechLanguage**, що показує, яка мова є стандартною системною. Наступні властивості – **SupportedTopicLanguages** і **SupportedGrammarLanguages** – не дуже зрозумілі з першого погляду, тому що для класів озвучення тексту була лише одна властивість для всіх підтримуваних мов. Однак **SpeechRecognizer** дає змогу розпізнавати ваш голос локально або використовувати онлайнові словники. Саме тому **SpeechRecognizer** має дві властивості: **SupportedGrammarLanguages** – для офлайнових завдань і **SupportedTopicLanguages** – для онлайнових граматик.

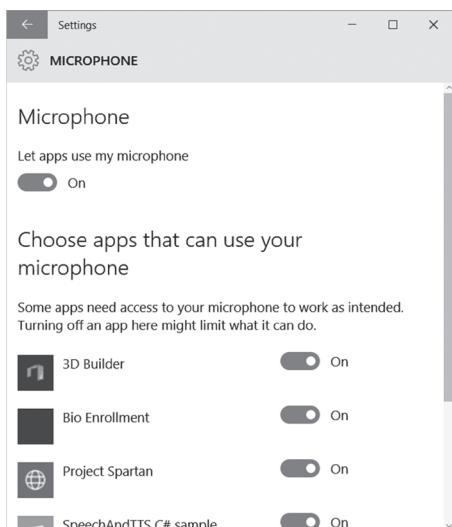
Почнімо з того, що покажемо, як використовувати об'єкти **SpeechRecognizer** кількома способами. Але перш за все вам необхідно задати цю властивість в маніфесті вашої програми, що дасть вам змогу використовувати механізм розпізнавання. UAP не містить спеціальних можливостей, як Windows Phone 8.1, тому вам необхідно описати лише можливість використання мікрофону. Зазвичай ваш маніфест виглядає так:

```
<Capabilities>
    <Capability Name="internetClient" />
    <DeviceCapability Name="microphone" />
</Capabilities>
```

Звичайно, цього недостатньо, і вам варто реалізувати додаткові можливості, що гарантуватимуть надання користувачем дозволу для вашої програми. Для цього варто реалізувати такий код:

```
bool permissionGained = await AudioCapturePermissions.
RequestMicrophonePermission();
if (!permissionGained)
{
    //ask user to modify settings
}
```

У Windows 10 користувачі можуть вимкнути мікрофон для обраних програм чи для всіх відразу. Ви можете легко знайти вікно, що дає змогу це зробити (**Settings -> Privacy -> Microphone**).



Якщо все гаразд із дозволами, можна починати виконувати певні методи, що реалізують логіку розпізнавання.

Залежно від сценарію, ви можете реалізувати такі варіанти розпізнавання мовлення:

- **Наперед визначені граматики** – у цьому випадку механізм розпізнавання буде використовувати онлайнові граматики. Вам не потрібно створювати вашу власну граматику і у вас є два варіанти: використати загальну граматику чи граматику, що базується на найбільш популярних пошукових запитах в Інтернеті. Таким чином, використовуючи загальну граматику, ви будете мати змогу розпізнати будь-який текст, але другий варіант оптимізований для пошуку.
- **Список програмних обмежень** – цей підхід дає можливість створити список рядків з певними словами чи фразами, які користувачі можуть вживати в розмові. Краще використовувати цей підхід, тоді ваша програма буде мати наперед визначений набір команд. Крім того, ви можете керувати списком під час виконання програми залежно від контексту.
- **SRGS-граматика** – завдяки мові SRGS ви можете створити документ XML із вбудованою граматикою. Це дає змогу створювати більш гнучкі програми без зафікованої в коді граматики.

За будь-якого підходу вам потрібно виконати такі дії.

- Створити об'єкт класу **SpeechRecognizer**. Це найлегший крок, який не вимагає спеціальних знань.
- Підготувати словник. Для цього вам потрібний об'єкт класу, що реалізує інтерфейс **ISpeechRecognitionConstraint**. Існує чотири класи обмежень, але ми обговоримо три з них: **SpeechRecognitionGrammarFileConstraint**, **SpeechRecognitionListConstraint** і **SpeechRecognitionTopicConstraint**. Перший дає змогу створити граматику, базуючись на файлі. Ви можете просто створити об'єкт **StorageFile** і передати його як параметр. Другий дає змогу використовувати програмний список як граматику, а останній підтримує заздалегідь визначені граматики.
- Як тільки ви створите обмеження, їх можна буде додати до колекції **Constraints** об'єкта **SpeechRecognizer** і викликати метод **CompileConstraintsAsync**. На цьому всі приготування буде завершено. Якщо ви не зробили ніяких помилок в обмеженнях, метод поверне статус **Success** і можна буде рухатись далі.
- Тепер можна розпочинати розпізнавання команд і тут також є декілька опцій: використати метод **RecognizeAsync** класу **SpeechRecognizer** чи властивість **ContinuousRecognitionSession** із викликом методу **StartAsync**. Перший спосіб дає змогу розпізнавати короткі команди,

застосовуючи наперед задані налаштування, але другий адаптований до тривалого розпізнавання тексту, що надиктовується. Звичайно, використовуючи **RecognizeAsync**, ви можете отримати потрібний результат, а застосувавши метод **StartAsync**, вам необхідно буде використовувати обробники подій **ContinuousRecognitionSession.Completed** і **ContinuousRecognitionSession.ResultGenerated**.

Також є методи, що задіюють вбудовані діалогові панелі для розпізнавання мовлення – просто використайте метод **RecognizeWithUIAsync**.

Якщо ви хочете знайти певні приклади розпізнавання мовлення, ми б порекомендували вам перейти за посиланням <https://github.com/Microsoft/Windows-universal-samples>.

Далі ми обговоримо більш цікаву тему – Cortana.

## Cortana

Windows 10 підтримує різні сценарії використання Cortana у вашій програмі. Як і раніше, ви можете інтегрувати деякі команди в Cortana, що допоможе користувачам запустити вашу програму і передати до неї певні дані. Але на сьогодні Cortana не лише розпізнає мовлення, але й допомагає, приймає рішення і є системою, що підтримує діалог з користувачем. Тому в UWP додано нові можливості, що дають змогу використовувати вашу програму як базу знань, що виконується у фоновому режимі і надає відповіді на питання користувача. Спочатку ми обговоримо, як реалізувати за допомогою Cortana прості команди, а потім – як використовувати програму у фоновому режимі, щоб вона могла працювати з Cortana.

Перш ніж починати щось робити з Cortana, потрібно перевірити, чи активована ця система на вашому пристрої. Для цього знайдіть вікно **Search** і перейдіть до розділу **Settings**:

 **Settings**

 Cortana can give you suggestions, ideas, reminders, alerts and more.

  Off

 Turning Cortana off clears what Cortana knows on this device, but won't delete anything from the Notebook. After Cortana is off, you can decide what you'd like to do with anything still stored in the cloud.

Manage what Cortana knows about me in the cloud

**Search online and include web results**

  On

Use my previous Bing searches, and location (if available) to improve search suggestions and provide more relevant web results.

Bing SafeSearch settings

Change how Bing filters adult content from your search results.

 Other privacy settings

 **Search the web and Windows**

Cortana досі недоступна в багатьох регіонах. Якщо ви живете в такому регіоні, то можете просто змінити місце розташування вашого комп'ютера на Сполучені Штати – і відразу після перезавантаження пристрою Cortana має стати доступною. Після того як Cortana з'явиться на вашому пристрой, можна починати розробляти програми із функціями, що стосуються цієї системи. Щоб додати нові команди в базу даних Cortana, необхідно створити XML-файл, що містить команди, які базуються на Voice Command Definition (VCD). VCD-файл є простим XML-файлом, який повинен містити стандартний XML-заголовок

```
<?xml version="1.0" encoding="utf-8"?>
```

і може містити такі основні елементи.

- **VoiceCommands** – кореневий елемент.
- **CommandSet** – цей елемент описує набір команд для конкретної мови і може містити команди.
- **Command** – описує саму команду. Ви будете використовувати атрибут **Name**, щоб визначити ім'я команди, але саму команду опишете завдяки наступним елементам.
- **Example** – приклади фраз, які користувач може сказати.
- **ListenFor** – слова або фрази, які Cortana має розпізнавати.
- **Feedback** – текст, який буде відображатися і озвучуватися, якщо команду розпізнано.
- **PhraseList** – завдяки цьому елементу можна визначити список слів або фраз, які можуть бути пов'язані з командою. Вимовивши будь-яку з них, користувач активує команду.

Переглянемо такий код:

```
<?xml version="1.0" encoding="utf-8" ?>
<VoiceCommands xmlns="http://schemas.microsoft.com/
voicecommands/1.2">
    <CommandSet xml:lang="en-us">
        <CommandPrefix> Cinema Application, </CommandPrefix>
        <Example> Find show times to Terminator </Example>

        <Command Name="showtimesCommand">
            <Example> find show times to Terminator </Example>
            <ListenFor> find show times to {film} </ListenFor>
            <Feedback> Finding show times to {film} </Feedback>
            <Navigate/>
        </Command>

        <PhraseList Label="film">
            <Item> Mission Impossible </Item>
            <Item> No escape </Item>
            <Item> Terminator </Item>
        </PhraseList>
    </CommandSet>
</VoiceCommands>
```

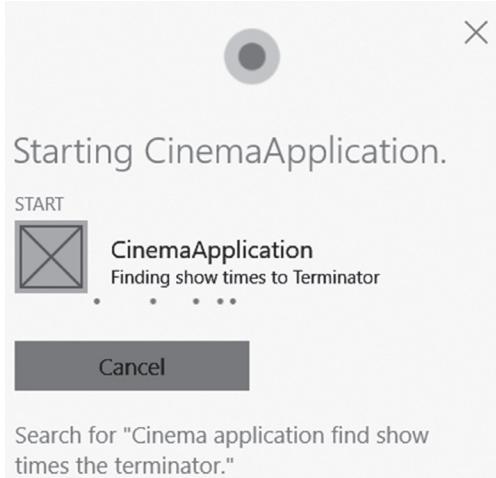
Цей XML-код містить тільки одну команду для американського варіанту англійської мови і дає змогу активувати програму, використовуючи ім'я **Cinema** і команду **Find show times to**.

Щоб активувати цю команду, програму слід запустити хоча б один раз, щоб виконався такий код:

```
var storageFile = await StorageFile.GetFileFromApplicationUriAsync
(new Uri("ms-appx:///VoiceCommands.xml"));
await VoiceCommandDefinitionManager.InstallCommandDefinitions
FromStorageFileAsync(storageFile);
```

У цьому прикладі ми створили об'єкт **StorageFile**, базуючись на нашому XML-файлі і використали клас **VoiceCommandDefinitionManager** для реєстрації.

Якщо все гаразд, ви можете запустити програму, щоб зареєструвати VCD-файл і відразу після цього використати Cortana для запуску програми:



Звичайно, програма ще не працюватиме, тому що ми повинні обробляти подію активації голосовою командою. У цьому випадку ви можете використати метод **OnActivated** і перевірити тип активації:

```
protected override void OnActivated(IActivatedEventArgs e)
{
    if (e.Kind == ActivationKind.VoiceCommand)
    {
        var vArgs = e as VoiceCommandActivatedEventArgs;
        var command=vArgs.Result.Text;
        //activate the application
    }
}
```

Отже, вам потрібно створити фрейм, розібрати параметри і перемістити фрейм до необхідної сторінки. Звичайно, список фраз час від часу потрібно оновлювати. Наприклад, у випадку кінотеатрів необхідно щотижня оновлювати список фільмів. Для цього ви можете використати такий код:

```
VoiceCommandDefinition commandSet;

if (VoiceCommandDefinitionManager.InstalledCommandDefinitions.
    TryGetValue(
        "CinemaApp_en-us", out commandSet))
{
    await commandSet.SetPhraseListAsync(
        "film", new string[] {"film 1", "film 2", "film 3"});
}
```

**CinemaApp\_en-us** – це ім'я **CommandSet**. Ми не вказали його у вихідному файлі, але якщо ви хочете змінювати фрази динамічно, необхідно використовувати атрибут **Name**.

Гаразд. Ми знаємо, як використовувати Cortana для запуску наших програм, але Cortana – це особистий помічник, і якщо користувач хоче реалізувати просте завдання в програмі, немає жодного сенсу запускати програму у фоновому режимі – просто дайте Cortana завдання зробити всі необхідні речі. Для цього UWP дає змогу вашим програмам взаємодіяти з Cortana, використовуючи фонові завдання.

Ідея проста: ви можете скористатися службами програм як джерелом даних для Cortana. Ми не будемо описувати, як додати службу програм, тому що відповідну інформацію надано в розділі 16. Однак я хочу показати, що вам потрібно зробити для того, щоб активувати цю функцію. Перш за все, необхідно змінити VCD-файл, додавши елемент **VoiceCommandService**:

```
<Command Name="showTripToDestination">
    <Example> find show times to Terminator </Example>
    <ListenFor> find show times to {film} </ListenFor>
    <Feedback> Finding show times to {film} </Feedback>
    <VoiceCommandService Target="CinemaCommandService"/>
</Command>
```

Зауважте, що елемент **VoiceCommandService** був включений у специфікацію VCD версії 1.2, а багато прикладів із MSDN базуються на специфікації 1.1, де немає цього елемента.

У цьому прикладі **CinemaCommandService** – це ім'я служби, яку ви використовували в маніфесті, коли оголошували там служби програм.

Реалізувати службу програм можна в такий спосіб:

```
public sealed class CinemaCommandService : IBackgroundTask
{
    private BackgroundTaskDeferral deferral;
    VoiceCommandServiceConnection voiceConnection;

    public async void Run(IBackgroundTaskInstance taskInstance)
    {
        deferral = taskInstance.GetDeferral();

        var triggerDetails =
            taskInstance.TriggerDetails as
        AppServiceTriggerDetails;

        try
        {
            voiceConnection =
                VoiceCommandServiceConnection.
                FromAppServiceTriggerDetails(
                    triggerDetails);

            voiceConnection.VoiceCommandCompleted +=
                (obj, arg) => { if (deferral != null)
                deferral.Complete(); };

            VoiceCommand voiceCommand =
                await voiceConnection.GetVoiceCommandAsync();

            switch (voiceCommand.CommandName)
            {
                case "showtimesCommand":
                {
                    var film =
                        voiceCommand.Properties["film"][0];
                    SendResponse(film);
                    break;
                }
                default:
                    var userMessage =

```

```
        new VoiceCommandUserMessage();
        userMessage.SpokenMessage =
            "Launching Cinema Application";

        var response = VoiceCommandResponse.
CreateResponse(userMessage);

        await voiceConnection.
RequestAppLaunchAsync(response);
        break;
    }
}
finally
{
    if (deferral != null)
    {
        deferral.Complete();
    }
}
}

private async void SendResponse(string film)
{
    var userMessage = new VoiceCommandUserMessage();
    userMessage.DisplayMessage = "Show times for "+film;
    userMessage.SpokenMessage = "Show times for today";

    var tileList = new List<VoiceCommandContentTile>();

    var tile = new VoiceCommandContentTile();
    tile.ContentTileType =
        VoiceCommandContentTileType.TitleWithText;

    tile.AppLaunchArgument =
        string.Format("film={0},type=IMAX", film);
    tile.Title = "IMax";
    tile.TextLine1 = "03:00pm, 05:00pm, 09:30pm";
    tileList.Add(tile);

    var tile2 = new VoiceCommandContentTile();
    tile2.ContentTileType =
        VoiceCommandContentTileType.TitleWithText;
```

```
tile2.AppLaunchArgument = string.Format  
    ("film={0},type=DBOX", film);  
tile2.Title = "DBox";  
tile2.TextLine1 = "03:10pm, 05:20pm, 09:30pm";  
tileList.Add(tile2);  
  
var response =  
    VoiceCommandResponse.CreateResponse(  
        userMessage, tileList);  
  
response.AppLaunchArgument =  
    string.Format("film={0}", film);  
  
await voiceConnection.ReportSuccessAsync(response);  
}  
}  
}
```

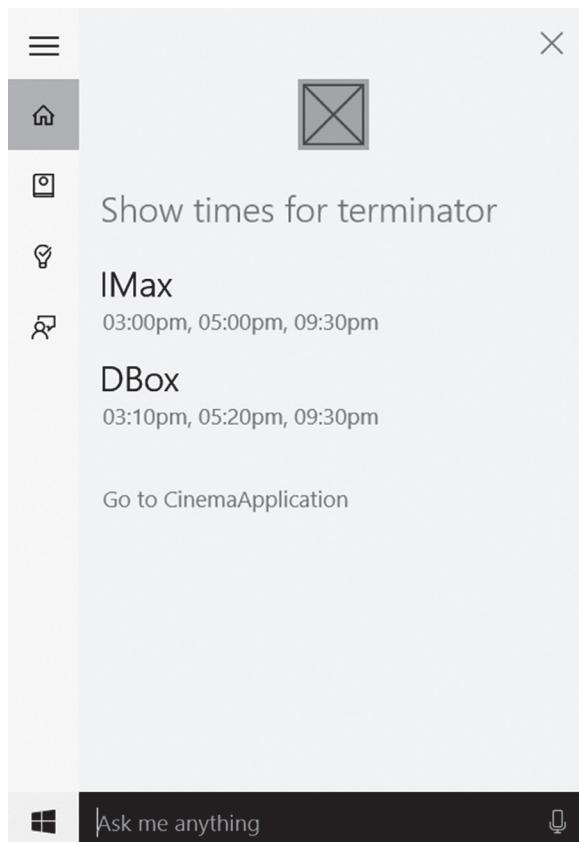
Цей код довгий, але насправді не надто складний. На першому етапі ми повинні отримати посилання на об'єкт **VoiceCommandServiceConnection**, щоб дістати інформацію про команду і повернути результати.

Відразу після цього ми повинні перевірити ім'я команди, яку використовували у VCD-файлі та, якщо його буде знайдено, можемо підготувати відповідь, що відповідає імені.

Щоб згенерувати відповіді, ми використали класи **VoiceCommandUserMessage** і **VoiceCommandContentTile** для створення назви і списку повідомлень всередині відповіді. Ці класи мають дуже прості властивості, і, як бачите, ви можете відображати повідомлення, що містять текст та зображення. Клацнувши будь-яке повідомлення, користувач може запустити програму, і завдяки параметрам, перейти на сторінку, яка найкраще підходить для повідомлення.

Щоб активувати наш компонент, потрібно запустити програму.

Запитавши Cortana про фільм, можна побачити таке повідомлення:

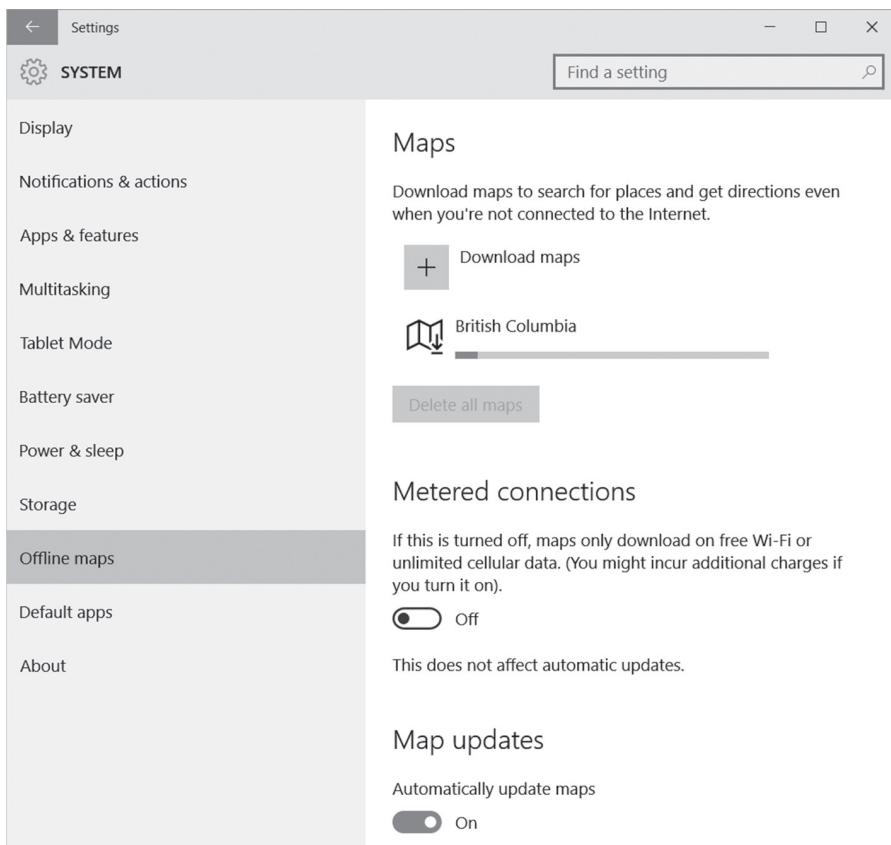


Розділ 21.

## **Карти**

У цьому розділи ми поговоримо про те, як використовувати карти в програмах Universal Windows Platform. Варто почати з відповіді на питання, яке часто ставлять розробники: коли оффлайнові карти стануть доступними для Windows-програм?

Починаючи з Windows 10, карти в режимі офлайн доступні не тільки для телефонів, а й для настільних комп'ютерів, ноутбуків, планшетів тощо. Користувач може відкрити вікно Налаштування, і завантажити в ньому всі необхідні карти. Після того, як карти завантажаться, наявний елемент керування буде використовувати їх за замовчуванням.



Це дійсно чудово, оскільки ми можемо використовувати свої пристрой, що працюють на Windows 10, у будь-якому місці, щоб знайти адресу, напрямок руху тощо. Як можна отримати доступ до цієї інформації? Безпосередньо з вашого пристрою.

Але офлайнові карти – це не єдина особливість карт на Windows 10. Нижче наведено й інші:

- **Unified platform** – починаючи з Windows 10 вам більше не потрібно користуватись такими програмами, як Here або Bing Maps для виконання різних завдань (або для отримання послуг на різних пристроях). Всі послуги доступні відразу й можуть бути використані для будь-яких UWP-програм.
- **Adaptive interface** – наявна програма Maps чудово працює на всіх пристроях. Ви можете використовувати сенсор, стилус або миш, а також змінювати розміри екрана. Те саме можна сказати про клас **MapControl**, що також доступний для розробників.
- **2D view with business and traffic information** – наявні карти підтримують всі загальні можливості 2D-карт, такі як різні вигляди, інформація про трафік (ви можете ввімкнути або вимкнути її), надання інформації про бізнес-об'єкти і транзит.
- **Location** – програма Maps дає змогу знайти та відобразити місце розташування користувача. Використовуючи API **Windows.Devices.Geolocation**, ви можете визначити розташування користувача за допомогою вашої програми, що покращить її функції і дасть змогу реалізувати багато цікавих сценаріїв.
- **External elements** – розробники можуть розміщувати на картах власні значки, прямокутники, багатокутники і навіть елементи керування XAML. Це дасть змогу налаштувати наявні карти і розширити кількість можливих сценаріїв.
- **Geofence** – API Geofencing дає змогу повідомити програму, якщо користувач перебуває в певній місцевості.
- **Routing** – розробники можуть користуватися службами Maps, щоб розрахувати маршрут до обраного пункту, використовуючи різні способи (наприклад, за кермом, пішки).
- **StreetSide** – новий функціонал, який дає змогу отримувати зображення вибраної області, щоб відображати їх у певному інтерфейсі. Це дуже корисно, якщо користувач хоче зрозуміти, як виглядає дана область.
- **3D views** – ще один новий функціонал, доступний для розробників, починаючи з Windows 10: прямо зараз ви можете переглядати карти не тільки в 2D-, а й у 3D-режимі. Якщо 3D-режим доступний для вибраної області, ви можете реалізувати той самий набір функцій, що і в стандартній програмі Maps.

Настав час обговорити, як використовувати карти у ваших програмах. Є два варіанти: перенаправляти деякі запити користувача до програми Maps або інтегрувати карти безпосередньо у вашу програму.

Якщо ви збираєтесь використовувати наявну програму Maps, то можете перевідправити на неї користувача за допомогою класу **Launcher**. Цей клас містить статичний метод **LaunchUriAsync**, який дає змогу відкрити зовнішню програму, використовуючи інформацію з URI. Наприклад, якщо ви використовуєте "http://...." у вашому Uri, то **Launcher** відкриє веб-браузер. Але якщо ви працюєте з Maps, URI має починатися з префіксу **bingmaps**: Звичайно, цього не достатньо, і вам необхідно передати деякі параметри за допомогою URI. Повний список можливих параметрів ви можете знайти за посиланням: [aka.ms/mapsapps/](https://aka.ms/mapsapps/).

Наприклад, цей код відкриє програму Maps, перемістить центр до Північного Ванкувера, і наблизить карту до 14 рівня:

```
await Launcher.LaunchUriAsync(new Uri  
("bingmaps:?lvl=14&rpt=adr.North%20Vancouver%20BC"));
```

Для того, щоб почати працювати з картами у вашій програмі, потрібно отримати «ключ доступу» на Maps Dev Center (<https://www.bingmapsportal.com/>).

Ви можете обрати пробний ключ, або ключ Basic, відповідно до ваших потреб. Відразу після випуску Windows 10 ви зможете знайти універсальну програму в списку доступних, але на момент написання книги такої інформації щодо Windows 10 не було. У будь-якому випадку, якщо не вдається опубліковувати програму для Windows 10 зараз, можна використати карти без ключа. Вони чудово працюють, але ви побачите повідомлення про те, що **MapServiceToken** не вказано, і ви не можете публіковувати програму без **MapServiceToken**.

Для Windows 10 ви можете створити ключ Basic для універсальної програми за допомогою порталу:

## Create key

**Application name \***

MyFirstW10Application

**Application URL**

Enter application URL

**Key type \***

What's This

Trial



**Application type \***

Public Windows App (8.x and earlier)



**Enter the characters you see \***



jD%3{



Create

Для того, щоб почати працювати з картами у вашій програмі, потрібно використовувати три простори імен – **Windows.UI.Xaml.Controls.Maps**, **Windows.Devices.Geolocation** і **Windows.Services.Maps** – та один елемент керування – **MapControl**. Перший простір імен містить **MapControl** і кілька класів, які дають можливість розміщувати дані на картах, налаштовувати камеру і стилі, настроювати вигляд з вікна. Другий простір імен дає змогу отримати розташування користувача, а також містить **Geofencing**. Третій простір підтримує декілька утиліт, які дають змогу знайти місце за адресою, розрахувати маршрут тощо.

Оскільки **MapControl** не є стандартним XAML-простором імен, вам потрібно додати новий простір імен до XAML-файлу:

```
xmlns:maps="using:Windows.UI.Xaml.Controls.Maps"
```

Тепер можна використовувати **MapControl**:

```
<maps:MapControl MapServiceToken="..."/>
```

Зраз ми знаємо, як додати карти до вашої програми. Почнімо розбиратися з базовим функціоналом, таким як центр карти, її збільшення тощо.

Подивимося на такий код:

```
maps.ZoomLevel = 14;  
maps.Center = (await MapLocationFinder.FindLocationsAsync  
("North Vancouver, BC", null)).Locations[0].Point;
```

У цьому коді ми використовуємо посилання на **MapControl** для того, щоб присвоїти значення двом властивостям: **ZoomLevel** і **Center**. Звичайно, у випадку **ZoomLevel** ви не повинні робити нічого особливого, але якщо бажаєте відцентрувати карту на основі адреси, необхідно використовувати такі картографічні сервіси, як **MapLocationFinder**. У прикладі вище ми використали метод **FindLocationsAsync**, і надали перше місце в списку властивості **Center**.

Щоб застосувати до карт 3D-подання, вам просто необхідно використати атрибут **Style**. Це дещо заплутує, оскільки атрибут стилю існує для всіх елементів керування користувачького інтерфейсу і його можна тлумачити по-різному.

```
<maps:MapControl Name="maps" Style="Aerial3D"/>
```

Звичайно, у разі 3D-подання краще змінити кут за замовчуванням для камери карти. Ви можете зробити це, використовуючи статичний метод **CreateFromLocationAndRadius** класу **MapScene**. Завдяки цьому методу можна створити сцену, використовуючи радіус (в метрах), рівень масштабування і радіус (в градусах). Щойно ви отримаєте об'єкт **MapScene**, потрібно буде застосувати метод **TrySetSceneAsync** до сцени:

```
await maps.TrySetSceneAsync(MapScene.CreateFromLocationAndRadius  
((await MapLocationFinder.FindLocationsAsync("Vancouver Downtown,  
BC", null)).Locations[0].Point, 500, 90, 60));
```

Якщо ви запустите цей код, то побачите центр Ванкувера в 3D:



Warning: MapServiceToken not specified.

Отже, застосовувати 3D-подання легко. Те саме стосується і функціоналу StreetView. Якщо вам потрібен перегляд вулиці, запишіть всього два рядки коду:

```
StreetsidePanorama panorama=await StreetsidePanorama.  
FindNearbyAsync((await MapLocationFinder.FindLocationsAsync  
("Vancouver Downtown, BC", null)).Locations[0].Point);  
maps.CustomExperience = new StreetsideExperience(panorama);
```

Якщо ви запустите цей код, то зможете побачити щось на зразок цього:



Звичайно, наявні карти відображають багато різної інформації, але якщо ви інтегруєте карти до вашої програми, то, можливо, забажаєте додати свої власні об'єкти.

Ось найпростіші об'єкти, які ви можете додати на карту: **MapIcon**, **MapPolygon** і **MapPolyline**. Наприклад, якщо ви хочете додати значок до карт, можна запустити такий код:

```
MapIcon mapIcon = new MapIcon();
mapIcon.Location = maps.Center;
mapIcon.NormalizedAnchorPoint = new Point(0.5, 1.0);
mapIcon.Image = RandomAccessStreamReference.CreateFromUri
(new Uri("ms-appx:///Assets/weather.png"));
mapIcon.ZIndex = 0;
maps.MapElements.Add(mapIcon);
```

Ви можете використовувати цей код, щоб розмістити значки погоди в даній місцевості, позначити визначні пам'ятки тощо. Крім того, ви можете додати до карт елементи керування XAML. У цьому разі вам потрібно використовувати колекцію **Children** замість **MapElements**, як було раніше. Але цього не достатньо. Щоб

додати елемент керування XAML UI, вам необхідно розташувати його в потрібному місці на карті. Для цього ви можете використовувати метод **SetLocation** класу **MapControl**. Цей метод має два параметри: посилання на додатковий елемент керування і розташування.

Таким чином, як ви бачите, поточна версія **MapControl** дуже потужна, і підтримує безліч функцій. Можна навіть написати окрему книгу тільки про цей елемент керування. Якщо ви хочете дізнатися більше, краще за все почати з Порталу Bing для розробників (<http://www.microsoft.com/maps/choose-your-bing-maps-API.aspx>). На цьому порталі є численні приклади та інформація про те, як використовувати Bing Maps з JavaScript або в програмах для альтернативних платформ.



Розділ 22.

## **Використання пера**

Ще один потужний елемент керування в Universal Windows Platform – це **InkCanvas**. Завдяки цьому елементу керування можна малювати на екрані у вашій програмі, використовуючи не тільки стілус, але також миш чи пальці. Тож завдяки цій функції можна додати можливість використання пера на будь-якому пристрої з ОС Windows 10.

Щоб активувати можливість використання пера, необхідно розмістити елемент **InkCanvas** усередині контейнера на штатл **StackPanel**, **Grid** тощо. Як і решта елементів керування, що базуються на **FrameworkElement**, **InkCanvas** містить низку властивостей, але у найпростіших випадках вам не потрібно нічого додатково оголошувати, щоб мати змогу їх використати:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <InkCanvas Name="ink"></InkCanvas>
</Grid>
```

У цьому випадку **InkCanvas** заповнить увесь простір всередині контейнера, і ви зможете почати робити нотатки або малювати щось. Зверніть особливу увагу, що за замовчуванням **InkCanvas** сприймає лише стілус (перо). Тому, якщо ви хочете використати пальці чи миш, вам потрібно реалізувати такий код:

```
ink.InkPresenter.InputDeviceTypes =
CoreInputDeviceTypes.Mouse | CoreInputDeviceTypes.Touch;
```

У цьому коді ми використовуємо інший важливий об'єкт – **InkPresenter**. **InkCanvas** – це просто елемент керування, який містить лише одну властивість, пов'язану з використанням пера, – посилання на **InkPresenter**, але **InkPresenter** містить всю інформацію про методи введення даних та багато інших настроїв, зокрема сукупність штрихів. Об'єкт **InkPresenter** не можна створити безпосередньо, але можна отримати посилання на нього за допомогою властивості **InkPresenter** з **InkCanvas**. Як уже згадувалося, **InkCanvas** не містить інших властивостей чи методів, що стосуються використання пера. Але у нас є про що говорити стосовно **InkCanvas**, і найбільш важливе питання – як додати можливість використання пера всюди, тому що зазвичай потрібно застосовувати перо до зображенень, відеороликів і тексту, а не для малювання всередині пустого контейнера. Давайте додамо до контейнера кілька елементів керування і подивимось, що з ним відбудеться.

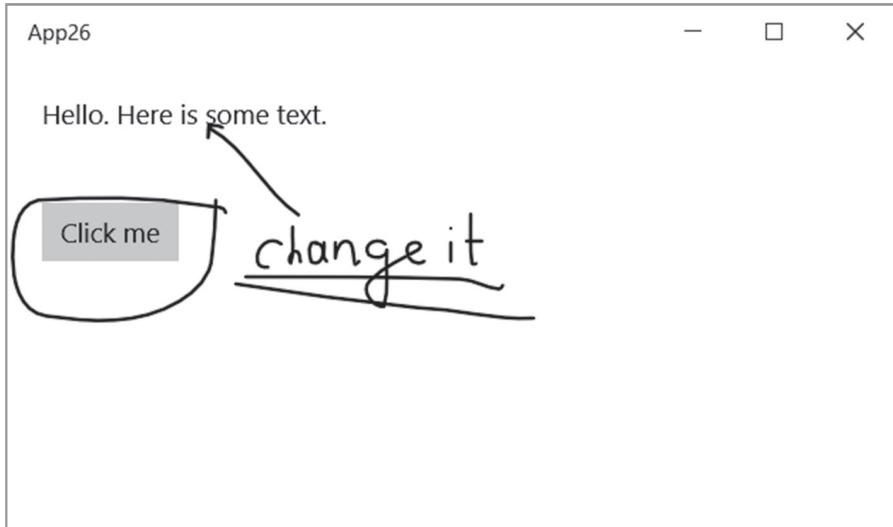
```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <InkCanvas Name="ink"></InkCanvas>
    <StackPanel>
```

```

<TextBlock Text="Hello. Here is some text." Margin="20">
</TextBlock>
<Button Content="Click me" Margin="20"></Button>
</StackPanel>
</Grid>

```

Якщо ми запустимо цей код, то побачимо, що всі елементи керування працюють правильно і можна робити будь-які нотатки.



У нашому прикладі ми розташували **InkCanvas** за іншими елементами керування, але якщо поміняти місцями **StackPanel** та **InkCanvas**, побачимо, що **InkCanvas** розташований над **StackPanel** і елементи керування на кшталт **Button** не працюють взагалі. Цю проблему можна вирішити за допомогою властивості **Canvas.ZIndex**.

```

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Canvas.ZIndex="1">
        <TextBlock Text="Hello. Here is some text." Margin="20">
        </TextBlock>
        <Button Content="Click me" Margin="20"></Button>
    </StackPanel>
    <InkCanvas Name="ink" ></InkCanvas>
</Grid>

```

Давайте поговоримо про налаштування об'єкта **InkPresenter** і про те, як додавати до нього різні ефекти. Мова піде про клас **InkDrawingAttributes**. Вам потрібно лише створити об'єкт цього класу і встановити різні властивості, наприклад **Color**, **PenTip**, **Size** тощо.

```
InkDrawingAttributes attr = new InkDrawingAttributes();
attr.Color = Colors.Red;
attr.IgnorePressure = true;
attr.PenTip = PenTipShape.Circle;
attr.Size = new Size(4, 10);
attr.PenTipTransform = Matrix3x2.CreateRotation((float)
(70 * Math.PI / 180));
ink.InkPresenter.UpdateDefaultDrawingAttributes(attr);
```

Код нижче показує, як використовувати **InkDrawingAttributes**. Щоб оновити атрибути, потрібно викликати метод **UpdateDefaultDrawingAttributes** і передати в нього об'єкт **InkDrawingAttributes**. У коді нижче ми також використали властивість **PenTipTransform**. Це дуже цікава властивість, яка дає змогу застосувати перетворення для форми і отримати більш природний вигляд штрихів завдяки їх різній висоті, що залежить від напрямку пера (напрямок переміщення / кут).



Ви можете поглянути на зображення, де ми намагалися намалювати символ "f", і помітити, що воно містить штрихи різної висоти. Фактично ми використовували ті самі налаштування, але цей ефект було застосовано за допомогою **PenTipTransform**.

Врешті-решт, можна намалювати будь-що, але як стерти штрихи, якщо ви помілсія? Для цього необхідно змінити режим **InkPresenter**:

```
ink.InkPresenter.InputProcessingConfiguration.Mode =
InkInputProcessingMode.Erasing;
```

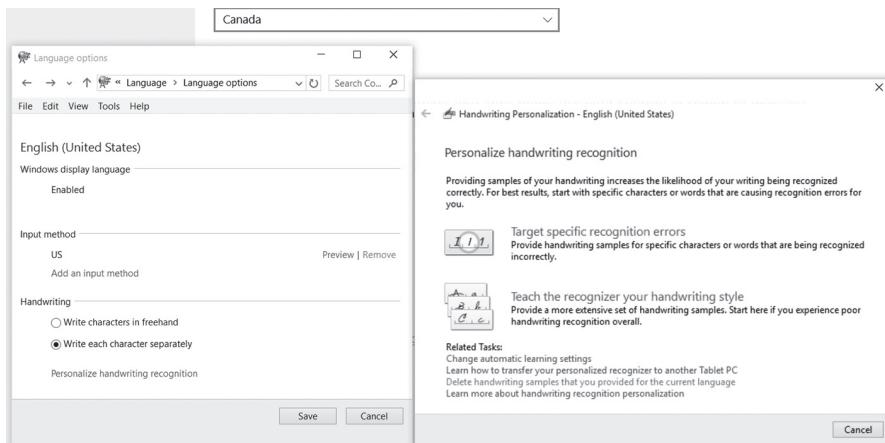
Наступним важливим класом, пов'язаним із використанням пера, є **InkStroke**. Все, що ви малюєте на **InkCanvas**, складається з **InkStroke**. Щоб отримати доступ до функціональності об'єкта **InkStroke**, можна використовувати властивість **StrokeContainer** об'єкта **InkPresenter**. Завдяки цій властивості можна зберегти наявні штрихи у файлі, додати нові штрихи, отримати доступ до вибраних штрихів тощо. Таким чином, якщо ви хочете реалізувати будь-яку функціональність, що стосується штрихів, для цього вам нагодиться клас **StrokeContainer**. Звичайно, найбільш популярними є методи **LoadAsync** і **SaveAsync**.

Багато програм також потребують функціональність розпізнавання. Щоб розпізнати рукописний текст, можна використовувати клас **InkRecognizerContainer**. Найпростіший спосіб використання виглядає так:

```
InkRecognizerContainer container =
new InkRecognizerContainer();
var result=await container.RecognizeAsync(ink.InkPresenter.
StrokeContainer, InkRecognitionTarget.All);
string s=result[0].GetTextCandidates()[0];
```

Цей код буде використовувати розпізнавач за замовчуванням і перетворить першого текстового кандидата зі списку на текст. Звичайно, цього недостатньо для більш складних сценаріїв. Наприклад, упродовж 10 років ви вчилися у школі писати російською та українською, але не англійською мовою. Тож коли ви пробуватимете написати щось англійською, то, як правило, отримуватимете правильний результат на другому або третьому місці в списку текстів-кандидатів. Крім того, в мене на комп'ютері може бути інсталювано кілька розпізнавачів.

Таким чином, краще обрати правильний механізм для розпізнавання. Для того, щоб це зробити, можна просто використовувати **SetDefaultRecognizer** і **GetRecognizers** об'єкта **InkRecognizerContainer**. Якщо англійська не є вашою рідною мовою, мабуть, не варто відображати повний перелік можливих результатів. Краще попросити користувачів налаштувати розпізнавач, застосувавши відповідні системні настройки. Наприклад, я можу настроїти стандартний розпізнавач так, щоб він адаптувався до моого стилю письма. Для цього слід вибрати варіант «писати кожний символ окремо» і зробити деякі «навчальні» налаштування:



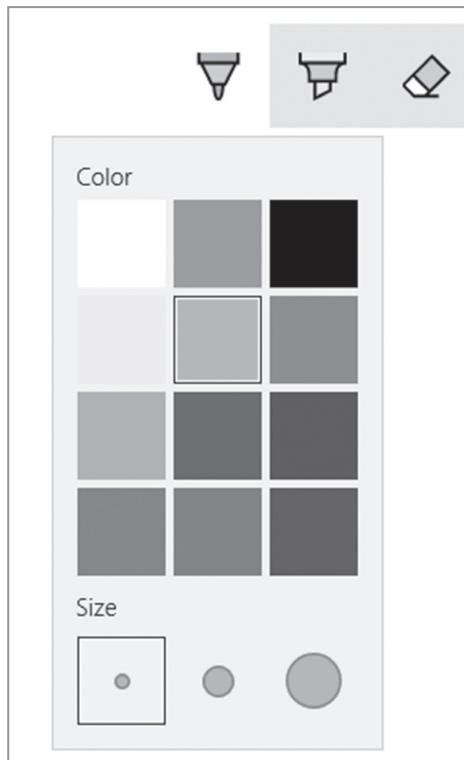
Наприкінці я хочу звернути вашу увагу на елемент керування [елемент Ink Toolbar](#). Це не стандартний елемент керування UWP, і його потрібно інсталювати окремо. Після інсталяції елемент керування буде доступний і готовий до використання на вкладці «Extension» у діалоговому вікні «Add Reference» (закрійте та відкрийте Visual Studio після інсталяції). Потрібно лише додати посилання на бібліотеку:

```
xmlns:ink="using:Microsoft.Labs.InkToolbarControl"
```

Знайдіть місце для елемента керування і зробіть посилання на існуючий **InkCanvas**:

```
<InkCanvas Name="ink" ></InkCanvas>
<ink:InkToolbar TargetInkCanvas="{x:Bind ink}"
VerticalAlignment="Top" HorizontalAlignment="Right">
</ink:InkToolbar>
```

Нижче можна побачити сам елемент керування, який генерує наш код:





Розділ 23.

## **Інтерфейс API для датчиків**

Сучасні пристрої мають безліч різних датчиків, що надають відомості про орієнтацію пристроя, його переміщення, наявність предметів попереду, рівень освітленості тощо. У деяких пристройових датчиків менше, у деяких більше, і поділити їх на категорії досить важко, оскільки зараз випускається чимало гібридних типів. Наприклад, будь-який телефон, планшет або планшетний ноутбук відстежує орієнтацію – це величезна категорія пристройових. Невеликі пристрої, зокрема телефони, оснащені датчиками відстані, однак такий датчик є і в найбільшого пристроя під керуванням Windows 10 – інтерактивної дошки Surface Hub. Тому у програмі, що працює з датчиками, насамперед слід переконатися, що потрібний датчик є на пристрой і справно повертає дані, а відтак передбачити в інтерфейсі програми можливість працювати навіть за його відсутності.

Нижче перелічено датчики, які підтримуються у Windows 10.

- **Light sensor.** Датчик світла визначає рівень освітленості довкола. За його показниками можна змінювати налаштування інтерфейсу програми. Наприклад, програма для читання книжок регулюватиме контрастність сторінки відповідно до рівня освітленості.
- **Accelerometer.** Датчик прискорення визначає силу інерції за осями x, y і z. Це досить популярний датчик, який часто використовується в іграх – змінюючи нахил пристроя, користувач може керувати швидкістю об'єктів.
- **Proximity sensor.** Датчик відстані фіксує об'єкти попереду і враховує відстань до них. У телефонах такі відомості використовують, щоб визначити, чи розмовляє користувач цієї міті по телефону. Крім того, датчик дає змогу взаємодіяти з інтерфейсом, навіть не торкаючись пристроя.
- **Activity sensor.** Датчик активності дає змогу збирати дані про поточний стан пристроя, визначаючи, що саме зараз робить користувач: біжить, їде на велосипеді, їде в машині тощо. Цей прилад широко використовується в програмах для фітнесу. У Windows 10 дані такого датчика можна зберігати протягом 30 днів.
- **Magnetometer.** Магнітометр вимірює напругу магнітного поля. Навряд чи ви будете безпосередньо використовувати його дані, однак вони потрібні для роботи таких приладів, як компас, вимірювач нахилу і датчик орієнтації.
- **Compass.** Windows 10 підтримує функцію визначення сторін світу на основі даних магнітометра і гіроскопа. Компас можна переключати в режим географічного або магнітного Північного полюса.
- **Orientation sensor.** Windows 10 підтримує два типи датчиків орієнтації: **SimpleOrientationSensor** і **OrientationSensor**. Перший надає основні відомості про те, як розташовано пристрой: у вертикальній чи горизонтальній орієнтації та екраном униз чи догори. Другий точно та з усіма подробицями визначає положення пристроя у тривимірному просторі.

- **Pedometer.** Крокомір – це ще один український популярний датчик для фітнес-програм. Він рахує крохи користувача, який має цей пристрій при собі.
- **Barometer.** Барометр аналізує динаміку атмосферного тиску й уможливлює прогнозування погоди. Було б чудово розробити власну програму з прогнозування, оскільки на місцевих сайтах з відомостями про погоду вже вкотре обіцяють дощ, а його немає та й немає.
- **Altimeter.** Висотомір визначає висоту над рівнем моря, на якій зараз перебуває пристрій. Згадайте про нього, коли розробляти програми для фітнесу або майструвати літак під керуванням Windows 10.
- **Gyrometer.** Гіроскоп вимірює кутові швидкості за осями x, y і z. Він зазвичай застосовується в іграх, а також у різноманітних апаратних рішеннях.
- **Inclinometer.** Інклінометр визначає кут нахилу відносно гравітаційного поля Землі. Він стане вам у пригоді під час розробки безпілотника під керуванням ОС Windows 10 або реалізації функцій, що використовують дані про орієнтацію пристрою.

Як бачимо, за допомогою різних наборів датчиків можна реалізовувати безліч цікавих завдань. І для створення справді хорошої програми необхідно вміти з ними працювати. Про це ми й поговоримо далі.

Насамперед розглянемо простір імен **Windows.Devices.Sensors**, де міститься все, що вам потрібно. Тут можна знайти будь-який клас і з легкістю визначити, із яким датчиком він пов'язаний. Зверніть увагу на відсутність базового класу (за винятком **Object**), що пояснюється браком спільних властивостей. Однак у багатьох датчиків є схожі властивості, події та методи, зокрема **MinimumReportInterval**, **ReportInterval**, **ReadingChanged**, **GetDefault** і **GetCurrentReading**. Перші дві властивості дають змогу визначити мінімальний підтримуваний датчиком інтервал між звітами та встановити бажаний інтервал, щоб вимірювати дані з потрібною частотою. Подія **ReadingChanged** стається в разі виявлення нових доступних даних або множини даних. Застосуйте її за потреби отримувати інформацію, щойно вона стане доступною. Завдяки методу **GetDefault** можна отримувати посилання на поточний датчик обраного типу. А якщо дані треба прочитати лише раз, використуйте метод **GetCurrentReading**, який повертає останні дані.

Наприклад, для барометра використовується такий код:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    barometer = Barometer.GetDefault();
    if (barometer==null)
    {
        Debug.WriteLine("Barometer is not found");
    }
}
```

```

    }
    double curData = barometer.GetCurrentReading();
    StationPressureInHectopascals;
    Debug.WriteLine($"The current pressure is: {curData}");
    barometer.RadingChanged += Barometer_ReadingChanged;
}

private void Barometer_ReadingChanged(Barometer sender,
    BarometerReadingChangedEventArgs args)
{
    Debug.WriteLine($"The new pressure is:
    {args.Rading.StationPressureInHectopascals}");
}

```

Зверніть особливу увагу на той факт, що показники, зокрема **BarometerReading**, можуть набувати значення **null**, якщо датчик пошкоджено (це особливо стосується зовнішніх датчиків), тож перевірку на рівність значенню **null** теж варто виконувати.

Для зовнішніх датчиків (крокомірів, датчиків відстані й активності тощо) краще реалізувати код, який буде перевіряти наявність датчика в середовищі виконання. З цією метою можна застосувати клас **DeviceWatcher**.

```

DeviceWatcher watcher;
Pedometer pedometer;
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    watcher = DeviceInformation.CreateWatcher(
        Pedometer.GetDeviceSelector());
    watcher.Added += Watcher_Added;
    watcher.Start();
}

private async void Watcher_Added(DeviceWatcher sender,
    DeviceInformation args)
{
    pedometer = await Pedometer.FromIdAsync(args.Id);
}

```

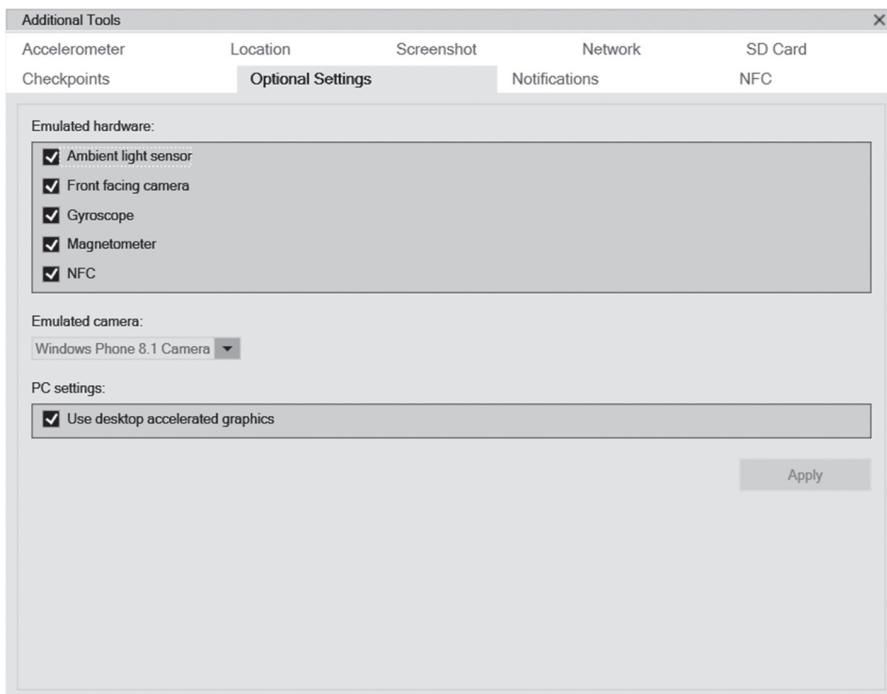
Зважайте, що хоча більшість датчиків можна використовувати без попередніх приготувань, для деяких із них (зокрема, крокоміра та датчика активності) необхідно оголосити спеціальні можливості пристрою в маніфесті:

```
<Capabilities>
    <DeviceCapability Name="activity" />
</Capabilities>
```

Це не вельми конфіденційні дані, однак користувач має знати, що ви збираєте певні відомості.

Якщо ваша програма використовує датчики, необхідно протестувати відповідні її функції на пристроях. Мусимо відмінити, що це дуже поширенна проблема, оскільки на комп'ютері розробника зазвичай менше датчиків, ніж на пристроях, де програма використовуватиметься.

Тож, найімовірніше, вам доведеться придбати пристрой з потрібними датчиками для тестування програми за допомогою інструментів віддаленого налагодження. Для тестування датчиків світла або прискорення, а також магнітометра можна скористатися емулятором Windows Phone. Остання версія містить спеціальний інструмент для емуляції датчика прискорення та підтримує можливість отримувати дані від датчика світла, гіроскопа і магнітометра:



У просторі імен **Windows.ApplicationModel.Background** міститься спеціальний тригер для виконання коду в разі повернення датчиком нових даних:

```
ActivitySensorTrigger tr = new ActivitySensorTrigger(5000);  
tr.SubscribedActivities.Add(ActivityType.Biking);  
tr.SubscribedActivities.Add(ActivityType.Running);
```

Він застосовується в програмах для фітнесу зі спливаючими сповіщеннями.

Окрім стандартних, можна використовувати й спеціальні датчики, для яких є універсальні драйвери. У цьому разі слід оголосити таку можливість пристрою і використовувати ідентифікатор інтерфейсу вашого датчика:

```
<Capabilities>  
    <DeviceCapability Name="sensors.custom">  
        <Device Id="any">  
            <Function Type="interfaceId:4025A865-638C-43AA-A688-  
                98580961EEAE"/>  
        </Device>  
    </DeviceCapability>  
</Capabilities>
```

Відразу після цього можна використати клас **CustomSensor**, щоб отримати посилання на датчик і розпочати читання даних:

```
customSensor = await CustomSensor.  
FromIdAsync(customSensorDevice.Id);  
if (customSensor != null)  
{  
    CustomSensorReading reading = customSensor.GetCurrentReading();  
    if (reading.Properties.ContainsKey(myKey))  
    {  
        //doing something  
    }  
}
```

Розділ 24.

## **Розширення платформи**

Універсальна платформа Windows дає змогу створювати універсальні програми для будь-яких пристроїв під керуванням Windows 10. Досі йшлося переважно про класи, які працюють на настільних комп'ютерах, ноутбуках, планшетах, телефонах тощо. А як щодо функцій, притаманних лише певному пристрою? Наприклад, Raspberry Pi підтримує можливості, яких немає на телефонах чи планшетах. На ПК чи ноутбуці, на відміну від телефона, можна відразу відправляти документи на друк, натомість смартфони підтримують функцію вібрації, яку рідко зустрінеш у ноутбуках.

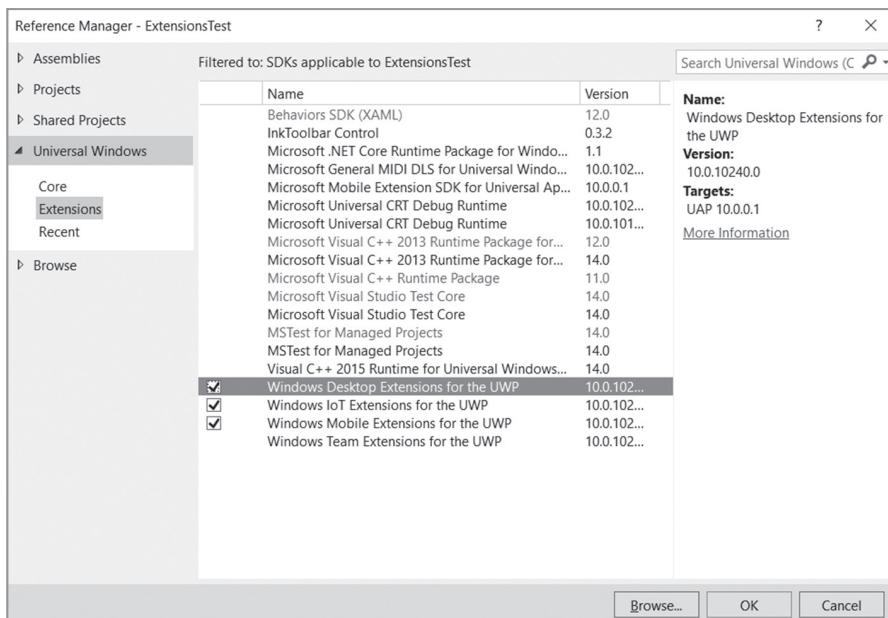
Середовища виконання для ОС Windows 8.1 і Windows Phone 8.1 суттєво відрізняються, тож розробнику зазвичай доводилося створювати різні інтерфейси програми для комп'ютерів і телефонів, усуваючи помилки логіки за допомогою директив препроцесора. Це досить незручно, адже навіть за використання шаблона Universal Application необхідно дублювати чималі сегменти коду та підтримувати відразу кілька проектів для різних пристройів.

Щоб уникнути цих проблем і створити дійсно універсальне рішення, корпорація Microsoft змінила підхід і впровадила розширення платформи. Це, по суті, набір контрактів, які з тих чи інших причин досі не реалізовані для всіх платформ. Наявні розширення виглядають, як зовнішні бібліотеки, але мають одну суттєву перевагу: щоб додати розширення, не потрібно створювати новий проект або ще раз компілювати програму для іншої платформи. Перевірка доступних контрактів здійснюється у тому самому середовищі виконання, і один і той самий код функціонуватиме на будь-якому пристрої під керуванням Windows 10.

Звісно, спроба використати контракти, які не знайдено, генерує винятки. Тому можна виявити всі ділянки коду, де застосовуються нетипові контракти, і обробити можливі винятки. Цей метод теж не вельми зручний через брак можливості налаштувати інтерфейс до того, як станеться виняток. Однак платформа UWP підтримує безліч інтерфейсів, у яких можна заздалегідь перевірити наявність потрібних контрактів.

Розглянемо наявні розширення та інтерфейси API.

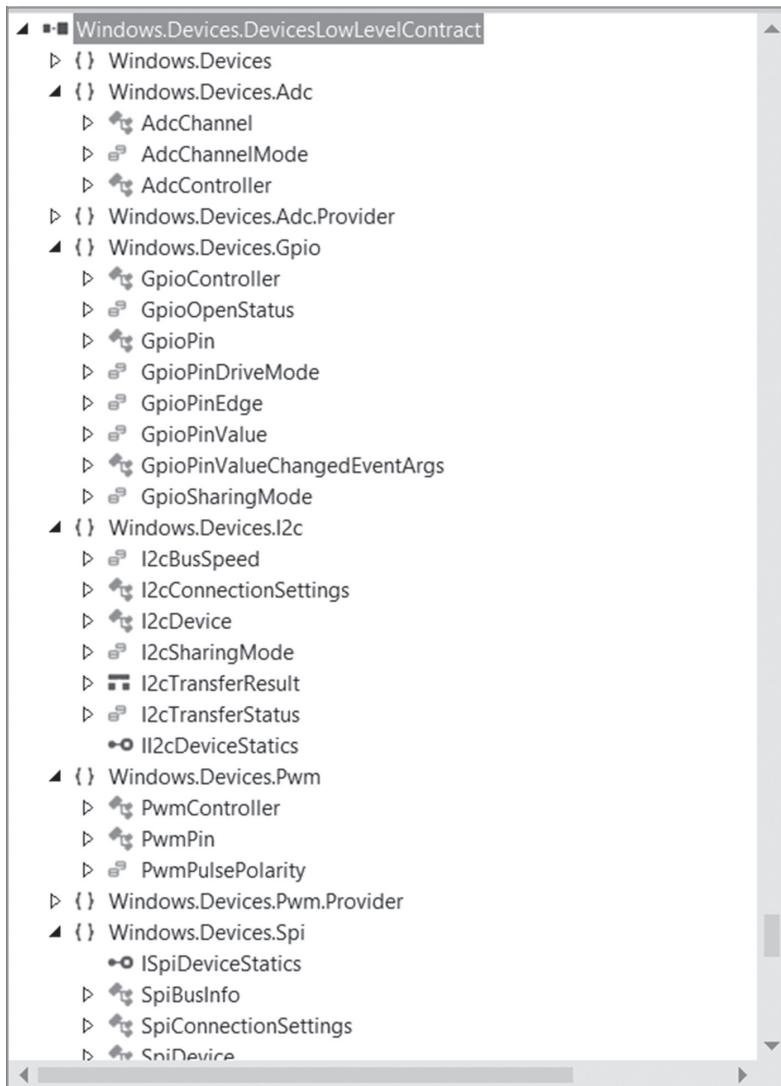
Насамперед потрібне розширення слід додати до проекту, відкривши вкладку Extensions у діалоговому вікні Add Reference. Зараз на ній доступні три найважливіші розширення: Mobile Extension, Desktop Extension і IoT Extension.



Незабаром їх буде доповнено іншими. Зокрема, корпорація Microsoft планує додати розширення для Xbox і Hololens. Повний перелік контрактів розширення можна переглянути у файлі маніфесту для цього розширення. Наприклад, маніфест IoT, що міститься в папці **C:\Program Files (x86)\Windows Kits\10\Extension SDKs\WindowsIoT\10.0.10240.0**, має такий вигляд:

```
<?xml version="1.0" encoding="utf-8"?>
<FileList TargetPlatform="UAP" TargetPlatformMinVersion="10.0.0.1"
TargetPlatformVersion="10.0.10240.0" SDKType="Platform"
DisplayName="Windows IoT Extensions for the UWP"
AppliesTo="WindowsAppContainer" MinVSVersion="14.0"
ProductFamilyName="Windows.IoT" SupportsMultipleVersion=
"Error" TargetFramework=".NETCore, version=v4.5.3;" 
SupportPrefer32Bit="True" MoreInfo="http://www.microsoft.com/
en-us/server-cloud/internet-of-things.aspx">
    <ContainedApiContracts>
        <ApiContract name="Windows.Devices.
        DevicesLowLevelContract" version="1.0.0.0"/>
        <ApiContract name="Windows.System.
        SystemManagementContract" version="1.0.0.0"/>
    </ContainedApiContracts>
</FileList>
```

Зараз там лише два контракти: **Windows.Devices.DevicesLowLevelContract** і **Windows.System.SystemManagementContract**. Щоб знайти для них класи, використовуйте вікно Object Browser у Visual Studio.



Для розширення IoT (Internet of Things – Інтернет речей) є кілька класів, що уможливлюють роботу з GPIO, I2C, SPI, а також надають інфраструктуру для PWM- і аналогового вводу. Якщо ви не працювали з мікроконтролерами, то

навряд чи знайомі з цими термінами, адже для телефонів, консолей Xbox і настільних комп'ютерів такі рішення не застосовуються. А от розробники програм для Raspberry, Arduino та інших плат знайдуть у цьому розширенні багато корисних класів.

Якщо вас цікавить розширення IoT, ласкаво просимо до розділу 35, де розглядається питання, пов'язані з Інтернетом речей.

У маніфесті розширення Mobile Extension міститься дещо більше контрактів, зокрема для роботи з віртуальними смарт-картами та телефонними дзвінками.

Поділ нетипових контрактів за розширеннями дуже умовний. Адже, скажімо, з деяких планшетів можна телефонувати, а певні моделі смартфонів підтримують підключення до док-станцій із монітором і принтером, завдяки чому приналідно слугують мініатюрними комп'ютерами. Врешті, Microsoft може реалізувати деякі нетипові контракти для всіх пристройів. Тому це суто логічний поділ, якого не обов'язково дотримуватися. Просто переконайтесь, що вибраний контракт доступний на пристройі. Далі його слід буде перевірити в середовищі виконання.

Найкраще зробити це за допомогою класу **ApiInformation**.

Клас **ApiInformation** міститься в просторі імен **Windows.Foundation.Metadata**. До його складу входять кілька статичних методів, зокрема **IsApiContractPresent**, **IsEventPresent**, **IsMethodPresent** тощо.

Щоб перевірити доступність GPIO, використовуйте наведений нижче код.

```
if (ApiInformation.IsApiContractPresent("Windows.Devices.  
DevicesLowLevelContract", 1))  
{  
    //doing something with GPIO  
}
```

Зверніть особливу увагу, що для методу **IsApiContractPresent** необхідно вказати два параметри. Другим буде основна версія (можна також зазначити проміжну версію, що вважатиметься третім параметром). Таким чином ви зможете посилатися на конкретну версію контракту, якщо корпорація Microsoft його оновлюватиме.

Звісно, наведений вище приклад не означає, що для застосування одного чи кількох контрактів необхідно створити кількасот блоків **if**. Краще створіть за цими даними настроюваний тригер і реалізуйте кілька візуальних станів.



Розділ 25.

## **Як публікувати веб-програми в Магазині**

У цьому розділі ми поговоримо про те, як зробити ваш веб-сайт/програму доступними в Магазині Windows для всіх користувачів. Звичайно, Visual Studio 2015 підтримує проекти на JavaScript/HTML 5 і представники Microsoft на презентаціях часто демонструють, як копіювати і вставляти наявний HTML/JavaScript-код і перекомпільовувати його в програму для Windows 10. Але насправді з цим пов'язані певні проблеми.

- Не всі веб-програми побудовані з використанням лише JavaScript/HTML. Наприклад, у Visual Studio не існує способу скопіювати/вставити ASP.NET-програму у проект для Windows 10. Крім того, багато розробників надають перевагу PHP, JSP чи навіть Node.js.
- Навіть якщо ми імпортуємо веб-програму у Windows 10, отримаємо два окремих проекти: веб-програму та програму для Windows 10. Якщо ми продовжимо витрачати час та ресурси на розробку веб-програми, потрібно буде стільки ж зусиль витратити і на розробку програми для Windows 10. І якщо часу чи коштів не вистачить, то, скоріш за все, до публікації програми для Windows 10 в Магазині справа так і не дійде.

Беручи до уваги цю проблему, давайте обговоримо, як публікувати наявну веб-програму в Магазині взагалі без імпортування. Почнемо огляд з простого елемента керування **WebView** і завершимо проектом Westminster.

## WebView

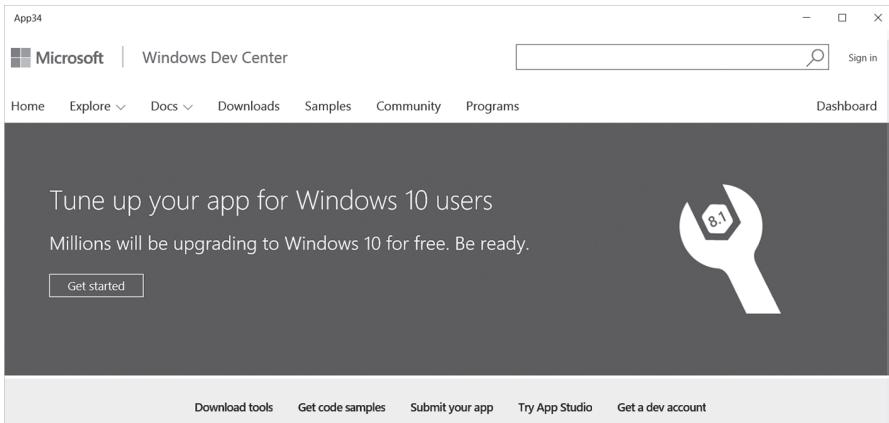
Починаючи з Windows 8, Microsoft представила елемент керування **WebView** для розробників, що дає змогу отримати зовнішній веб-контент через URI. Раніше цей елемент керування навіть не працював як стандартний елемент керування XAML і розробникам потрібно було «танцювати з бубоном», щоб змусити його виконувати належні функції. Але у Windows 8.1 елемент керування **WebView** було оновлено і зараз він успадковується з **FrameworkElement** і підтримує багато можливостей XAML, таких як прозорість, трансформації та ін.

Звичайно, у Windows 8.1 елемент керування **WebView** базувався на Internet Explorer і підтримував майже всі можливості цього браузера, крім плагінів, але, починаючи з Windows 10, цей елемент розробниками Microsoft було переписано, і зараз він базується на новому браузері Microsoft Edge. Це хороша новина, тому що Microsoft Edge підтримує багато нових можливостей HTML 5 і забезпечує вищу продуктивність JavaScript для сучасних веб-програм. Також корпорація Microsoft пообіцяла оновлювати Microsoft Edge частіше, ніж Internet Explorer, і водночас надавати ці оновлення для елемента керування **WebView**. Тепер розглянемо, як використовувати **WebView** у вашій програмі.

Ви можете додати елемент керування **WebView** до сторінки, як і будь-який інший елемент керування XAML. Наприклад, цей код додає **WebView** і переміщає вас на сайт **dev.windows.com**:

```
<Grid>
    <WebView Source="http://dev.windows.com"></WebView>
</Grid>
```

Відкривши цю сторінку, ви побачите повністю функціональну веб-програму:



Отже, в найгіршому випадку ви можете опубліковувати програму в Магазині, що буде перенаправляти вас до вашої веб-програми. Але в такому разі ваша програма не матиме жодних можливостей Windows 10 і буде доступно лише в онлайні. Отже, у випадку **WebView** краще вбудувати онлайн-контент до справжньої програми з Магазину.

Щоб виконати перехід у **WebView** за допомогою C#-коду, можна застосовувати метод **Navigate**.

```
webView.Navigate(new Uri("http://www.microsoft.com"));
```

Зауважте, що, починаючи з Windows 8.1, елемент керування **WebView** може переміщати вас до зовнішнього контенту так само, як і до контенту, що міститься в пакеті. Для цього вам потрібно використовувати спеціальні префікси для URI і ці префікси будуть додані автоматично до всіх відповідних URI всередині вашого документа.

```
<WebView Source="ms-appx-web:///html/index.html"
Name="webView"></WebView>
```

Цей код пересправлює елемент керування **WebView** на сторінку **index.html** в локальному пакеті. Просто створіть підпапку **html** у вашому проекті, використовуючи Visual Studio, і розмістіть там файл **index.html** із будь-яким вмістом.

На практиці не часто доводиться переходити за допомогою **WebView** до локальної сторінки в пакеті, але префікс **ms-appdata://local** ми кілька разів використовували.

```
<WebView Source="ms-appdata:///local/html/index.html"  
Name="webView"></WebView>
```

Цей код переводить **WebView** до тієї ж сторінки в папці **html**, але сторінка і папка розміщені у локальному сховищі програми. Тому ви можете легко використовувати цей підхід для офлайнових сценаріїв. Наприклад, деякі партнери публікують свої журнали в **html**-форматі і поширяють як **zip**-архіви. Архів містить усі файли сценаріїв, зображення, **html**-файли, і все, що вам потрібно, – це відобразити його у **WebView**. Наприклад, ваша програма може надати користувачеві можливість завантажувати будь-яку інформацію та читати її в офлайновому режимі. Для контенту в локальному сховищі діють ті самі правила: префікс буде застосовано до всього відповідного контенту. Тому вам не потрібно робити нічого особливого, а просто розпакувати архів в локальне сховище і перемістити **WebView** на головну сторінку.

Крім методу **Navigate**, **WebView** містить ще три важливих методи, які допоможуть продемонструвати HTML-контент.

- **NavigateToString** – дає змогу використовувати значення рядкової змінної як **html**-контент. Цей метод дуже популярний для програм, що висвітлюють новини, коли зовнішній контент з RSS надходить у **html**-форматі.
- **NavigateWithHttpRequestMessage** – дає можливість надсилати на сервер POST-запити. Цей метод важливий, якщо сервер запитує дані для автентифікації.
- **NavigateToLocalStreamUri** – дає змогу використовувати власний **URI**-розвізнавач для локального контенту.

Розгляньмо метод **NavigateToLocalStreamUri** детальніше. Ми говорили про програму для журналу, у якій користувачеві надсилається **zip**-архів з новими випусками. Звичайно, ви можете розпакувати випуски в локальну папку, але в деяких випадках це небажано. Наприклад, розробник може попросити вас зберігати лише зашифрований контент, щоб уникнути копіювання інформації з локального диску. У цьому випадку ви можете залишити контент в архіві і реалізувати спеціальний **URI**-розвізнавач. Як тільки ви передасте **URI** в метод **NavigateToLocalStreamUri**, він перешле **URI** вашому розпізнавачу й очікува-

тиме на зворотний потік з контентом для демонстрації. Якщо у вас є посилання на певний залежний контент на вашій веб-сторінці, будуть застосовані ті самі правила. Для наших партнерів ми реалізували розпізнавач, що базується на такому шаблоні:

```
public sealed class StreamUriWinRTResolver : IUriToStreamResolver
{
    private string magazineID;

    public StreamUriWinRTResolver(string magazineID)
    {
        this.magazineID = magazineID;
    }

    public IAsyncOperation<IInputStream> UriToStreamAsync(Uri uri)
    {
        lock (this)
        {
            if (uri == null)
            {
                throw new Exception();
            }
            string path = uri.AbsolutePath;

            return GetContent(path).AsAsyncOperation();
        }
    }

    private async Task<IInputStream> GetContent(string path)
    {
        //get content from archive
    }
}
```

Як ви бачите, розпізнавач реалізує інтерфейс **IUriToStreamResolver**, у якому є метод **UriToStreamAsync**, що отримує URI як параметр і повертає потік із контентом. Щоб використовувати цей розпізнавач, вам потрібно реалізувати такий код:

```
Uri url = webView.BuildLocalStreamUri(id, "index.html");
StreamUriWinRTResolver myResolver =
new StreamUriWinRTResolver(id);
webView.NavigateToLocalStreamUri(url, myResolver);
```

У першому рядку коду ми використали метод **BuildLocalStreamUri**, що допомагає створити URI у правильному форматі та направити його на локальний контент. Це краще, ніж згадувати правильні префікси. Відразу після цього ми створимо розпізнавач і передамо URI та розпізнавач до методу **NavigateToLocalStreamUri**.

На додачу до методів навігації, **WebView** має декілька методів і властивостей, що можуть допомогти покращити інтерфейс для навігації між сторінками. Наприклад, ви можете використовувати властивості **CanGoForward** та **CanGoBack**, щоб зрозуміти, чи можливо реалізувати в інтерфейсі кнопки переходу вперед і назад. Щоб реалізувати саму навігацію, ви можете використовувати такі методи, як **GoBack**, **GoForward**, **Refresh** і **Stop**.

Елемент керування **WebView** також містить деякі важливі події.

Є багато сайтів, де будь-яке посилання відкривається в новому вікні. Наприклад, якщо ви розробляєте пошукову систему, краще відкривати нове вікно/вкладку, коли користувач вибирає результат пошуку, оскільки він може забажати відкрити декілька посилань. Проте **WebView** не підтримує кількох вкладок. Як тільки користувач клацне посилання, що відкривається в новому вікні, **WebView** перенаправить його до Microsoft Edge і відкриє посилання в цьому браузері. Це не дуже хороша практика – втрачати взаємодію користувача з програмою і перенаправляти його до іншої програми. Саме тому у Windows 8.1 типовою методикою був перегляд усього HTML-контенту з метою видалення всіх атрибутів **target**, які приводять до відкриття нового вікна. Але у Windows 10 **WebView** містить цікаву подію, що називається **NewWindowRequested**. Якщо ви хочете уникнути відкриття контенту в новому вікні, можна реалізувати такий обробник подій:

```
private void webView_NewWindowRequested(WebView sender,
WebViewNewWindowRequestedEventArgs args)
{
    webView.Navigate(args.Uri);
    args.Handled = true;
}
```

У цьому прикладі ми просто здійснюємо навігацію у **WebView** до нового контенту замість того, щоб відкривати нове вікно в зовнішній програмі. Звичайно, на практиці вам потрібно буде реалізовувати складніші алгоритми, але їхня ідея буде тією самою.

Три наступні події – це **NavigationStarting**, **NavigationFailed** та **NavigationCompleted**. Завдяки цим подіям ви можете робити що завгодно з навігацією у вашій програмі.

Починаючи з Windows 10, **WebView** підтримує спеціальний конструктор, що дає змогу запускати **WebView** не в UI-потоці. Зараз ви можете спробувати скористатися ним, створивши **WebView** динамічно.

```
WebView wv = new WebView(WebViewExecutionMode.SeparateThread);  
grid.Children.Add(wv);  
wv.Navigate(new Uri("http://www.microsoft.com"));
```

Чудово, нам уже відомо, як демонструвати веб-контент, використовуючи елемент керування **WebView**. Розгляньмо можливості **WebView** щодо інтеграції зовнішнього веб-контенту і XAML-програм.

Перш за все, зовнішній веб-контент може визначати спеціальний інтерфейс для вашої XAML-програми. Це може бути будь-який набір методів JavaScript, які ви можете викликати з C#-коду, використовуючи метод **InvokeScriptAsync**. Давайте протестуємо цей підхід, створимо пусту веб-програму у Visual Studio і додамо просту сторінку:

```
<!DOCTYPE html>  
<html>  
<head>  
    <title></title>  
    <meta charset="utf-8" />  
    <script>  
        function f()  
        {  
            document.body.innerText = "Hello from Windows Runtime";  
        }  
    </script>  
</head>  
<body></body>  
</html>
```

Ця сторінка містить одну функцію, що не викликається за замовчуванням. Просто запустіть цю програму локально і скопіюйте URL-адресу з браузера. Використовуйте цю URL-адресу для створення такого **WebView**:

```
<WebView Name="wv" NavigationCompleted="Wv_NavigationCompleted"  
    Source="http://localhost:10289/">  
</WebView>
```

Щойно **WebView** завантажить контент, він активує метод **NavigationCompleted**, що викличе наш JavaScript-код:

```
private async void Wv_NavigationCompleted(WebView sender,
    WebViewNavigationCompletedEventArgs args)
{
    await wv.InvokeScriptAsync("f", null);
}
```

Якщо у вас немає ніякого спеціального JavaScript API, ви можете вставити JavaScript-код, використовуючи функцію **eval** із JavaScript і метод **InvokeScriptAsync**. Це дає змогу розширити функціональність сторінки, використовуючи сценарії, що динамічно генеруються із C#-коду.

Ви можете викликати C#-код із JavaScript також. Universal Windows Platform надає для цього дві можливості.

Перш за все, ви можете використати метод **window.external.notify** для надсилання повідомлень до **WebView** і обробник події **ScriptNotify**, щоб обробити переданий рядок і вирішити, що ви будете робити з ним далі у C#-коді. Давайте відредагуємо сторінку в нашій веб-програмі в такий спосіб:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script>
        function f()
        {
            window.external.notify("Hello from JavaScript");
        }
    </script>
</head>
<body onload="javascript:f()">
    Вміст веб-сторінки
</body>
</html>
```

Цей код надсилає повідомлення до зовнішньої програми (до **WebView**), щойно тіло сторінки завантажено. Щоб перевірити, чи отримав **WebView** повідомлення, можна реалізувати такий XAML-код:

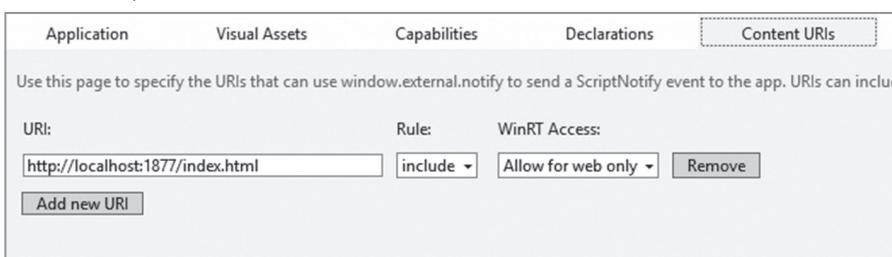
```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Text="" Name="myText" Margin="10"></TextBlock>
    <WebView x:Name="webView" Grid.Row="1"
        ScriptNotify="webView_ScriptNotify"></WebView>
</Grid>
```

Ми готові використовувати **TextBlock**, щоб відобразити отримане повідомлення, і задали для події обробник **ScriptNotify**. Реалізація в коді дуже проста:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    webView.Navigate(new Uri("http://localhost:1877/index.html"));
    base.OnNavigatedTo(e);
}

private void webView_ScriptNotify(object sender, NotifyEventArgs e)
{
    myText.Text = e.Value;
}
```

Але якщо ви запустите цей код, то не побачите ніякого тексту. Проблема в стандартних налаштуваннях безпеки, адже зовнішні веб-сторінки не можуть надсилати жодних повідомлень до **WebView** чи використовувати які-небудь об'єкти Windows Runtime. Отже, вам потрібно додати всі сторінки, яким ви довіряєте, до файлу маніфесту програми. Ви можете це зробити, використовуючи редактор маніфесту у Visual Studio 2015 – просто відкрийте вкладку Content URI та додайте URI веб-сторінки.



Вам потрібно застосувати до сторінок правило **include** і достатньо задати налаштування **Allow for web only**. Якщо ви бажаєте відредагувати маніфест у xml-редакторі, вам потрібно додати такі рядки коду в елемент **Application**:

```
<uap:ApplicationContentUriRules>
    <uap:Rule Match="http://localhost:1877/index.html"
        Type="include" WindowsRuntimeAccess="allowForWebOnly" />
</uap:ApplicationContentUriRules>
```

Якщо ви запустите Windows-програму ще раз, у текстовому блоці має з'явитися повідомлення.

Зверніть особливу увагу, що вам не потрібно звертатися до яких-небудь налаштувань безпеки для JavaScript із пакету програми чи з локального або тимчасового сховища.

Починаючи з Windows 10, **WebView** підтримує передачу об'єктів Windows Runtime до JavaScript-коду. Отже, ви можете підготувати спеціальний об'єкт із багатьма службовими методами. Тоді JavaScript буде мати можливість активувати деякі функції Windows 10, використовуючи ці методи. Щоб виконати таку активацію, вам потрібно створити окремий об'єкт Windows Runtime, адже якщо ви створите клас всередині програми напряму, то не зможете його використати. Отже, використайте Visual Studio 2015 для реалізації компоненту Windows Runtime і додайте посилання на нього з програми. Об'єкт дуже простий, але достатній для того, щоб продемонструвати цю можливість:

```
[AllowForWeb()]
public sealed class W10Util
{
    public string getData()
    {
        return "Hello from W10";
    }
}
```

Щоб використати об'єкт класу, вам потрібно застосувати до класу атрибут **AllowForWeb**, як показано вище. Клас готовий, тож реалізуємо в нашій програмі такий код:

```
W10Util w10obj=new W10Util();
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    webView.AddWebAllowedObject("w10obj", w10obj);
```

```
        webView.Navigate(new Uri("http://localhost:1877/index.html"));
        base.OnNavigatedTo(e);
    }
```

У вищеприведеному коді ми створили об'єкт класу **W10Util** і передали його до **WebView**, використавши метод **AddWebAllowedObject**. Він має два параметри: ім'я, яке ви можете використовувати в JavaScript, і посилання на об'єкт.

Врешті-решт, у JavaScript ми можемо використати об'єкт, який передали, як і будь-який інший об'єкт:

```
function f()
{
    if (window.w10obj !==undefined)
        document.body.innerText=w10obj.getData();
    else
        document.body.innerText="Object is not found";
}
```

Отже, ви можете застосовувати **WebView** не лише для відображення веб-контенту, але і для взаємодії між C# і JavaScript. Звичайно, цей підхід вдалий, якщо ви розробляєте програму на C# і хочете додати певний веб-контент. Але починаючи з Windows 10, ви можете гарантувати веб-програмі повний доступ до Windows Runtime. Це можливо завдяки проекту Westminster. Розглянемо цей підхід детальніше.

## Проект Westminster

Елемент керування **WebView** дає можливість використовувати компоненти Windows Runtime на зовнішніх веб-сторінках, але також вимагає створити деякі обертки і, відверто кажучи, **WebView** – необов'язковий компонент, якщо ви хочете розмістити веб-програму в Магазині, не впливаючи на наявний процес публікації. Тому під час конференції Build 2015 корпорація Microsoft представила декілька «мостів», що дають можливість переносити нативні програми з інших платформ у Магазин Windows 10. Одним із таких мостів є проект Westminster. Завдяки ньому ви можете публікувати наявні веб-програми у Магазині, не переписуючи їх заново. Звичайно, це можна було робити і раніше, наприклад, розмістивши звичайний елемент **WebView** у програмі і виконуючи переспраямування в ньому на головну сторінку вашого сайту. Але такі програми не підтримують усіх можливостей Windows через обмежений доступ до Windows Runtime, у той час як проект Westminster дає змогу отримати повний доступ до Windows Runtime.

без жодних обгорток чи проксі-елементів керування, таких як **WebView**. Таким чином, ви можете і далі працювати з веб-сайтом на сервері, реалізовуючи весь необхідний код. У той же час користувач постійно матиме доступ до останньої версії веб-програми.

На жаль, зараз проект Westminster працює тільки для Windows-програм на JavaScript . Але в деяких випадках ви можете редагувати лише маніфест.

Давайте створимо нову Windows-програму на JavaScript. Як тільки це буде зроблено, вам потрібно буде внести певні зміни у файл маніфесту.

Перш за все, вам потрібно зробити головну сторінку веб-програми стартовою:

Application	Visual Assets	Capabilities	Declarations	Content URIs	Packaging
Use this page to set the properties that identify and describe your app.					
Display name:	<input type="text" value="JavaSApplication"/>				
Start page:	<input type="text" value="http://localhost:1877/index.html"/>				
Default language:	<input type="text" value="en-US"/>	<a href="#">More information</a>			
Description:	<input type="text" value="JavaSApplication"/>				

Зараз програма буде запускатися, використовуючи віддалену сторінку на сайті. Але цього недостатньо, і ми повинні вирішити ті самі питання безпеки, що й для **WebView**. Відкрийте вкладку Content URI і додайте URL веб-сторінки:

Application	Visual Assets	Capabilities	Declarations	Content URIs	Packaging
Use this page to specify which URLs can be navigated to by an iframe in your app and which URLs, when loaded in a WebView, can use wildcards in subdomain names (for example https://*.microsoft.com or https:///*.*.microsoft.com).					
URI:	Rule:	WinRT Access:			
<input type="text" value="http://localhost:1877/index.html"/>	<input type="button" value="include"/>	<input type="button" value="All"/>	<input type="button" value="Remove"/>		
<input type="button" value="Add new URI"/>					

У цьому випадку ми обрали налаштування **All** для доступу до WinRT, щоб мати змогу працювати з компонентами Windows Runtime напряму без обгорток. Якщо обрано варіант **Allow for web only**, ви можете працювати лише з доданими об'єктами.

Щойно ми додамо всі сторінки до файлу маніфесту (ми можемо використовувати символ «\*» також і для обраного домену), з'явиться можливість публікувати програму (не забудьте додати значки). Усі можливості Windows 10 будуть реалізовані на сервері за допомогою JavaScript. Наприклад, наведена нижче веб-сторінка буде генерувати спливаюче повідомлення, якщо її запущено як

частину програми Windows 10. Однак якщо її запустити з браузера, поведінка буде іншою.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta charset="utf-8" />
    <script>
        function f()
        {
            if (typeof Windows !== 'undefined' &&
                typeof Windows.UI !== 'undefined' &&
                typeof Windows.UI.Notifications !==
                'undefined') {
                var notifications = Windows.UI.Notifications;
                toast = notifications.ToastTemplateType.
                    toastText01;
                toastContent = notifications.
                    ToastNotificationManager.
                    getTemplateContent(toast);
                toastText = toastContent.
                    getElementsByTagName('text');

                toastText[0].appendChild(toastContent.
                    createTextNode('Message from JavaScript'));

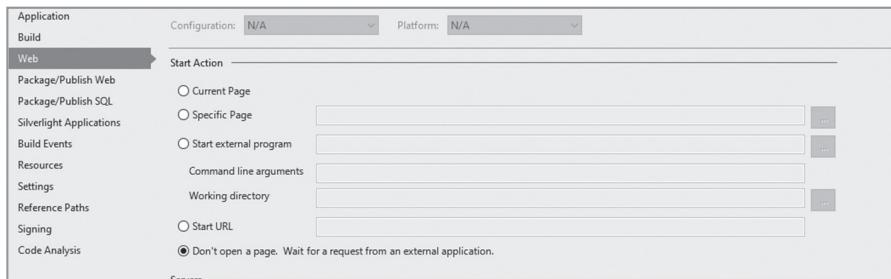
                var toastNotification = new notifications.
                    ToastNotification(toastContent);

                notifications.ToastNotificationManager.
                    createToastNotifier().show(toastNotification);
            }
            else {
                document.body.innerText =
                    "I am running in a browser";
            }
        }
    </script>
</head>
<body onload="javascript:f()">
    Веб page content

```

```
</body>  
</html>
```

Щойно ви почнете розробляти веб-програму, вам потрібно буде налагоджувати код JavaScript, щоб перевірити, чи все працює справно. Але якщо ви захочете налагодити код, що стосується Windows 10, то не зможете запустити веб-програму безпосередньо в режимі налагодження, оскільки вона має бути запущена з клієнта Windows 10. Але Visual Studio 2015 може вам допомогти. Просто відкрийте/створіть розробку, що містить проект для Windows 10 і веб-програму, та встановіть усі необхідні точки зупину в JavaScript. Використовуючи властивості проекту, вам потрібно відкрити вкладку **Web** і обрати варіант **Don't open a page. Wait for a request from an external application** (Не відкривати сторінку. Чекати на запит від зовнішньої програми). Ця опція дає змогу ініціювати налагодження, як тільки сторінка буде завантажена всередині програми для Windows 10:



На наступному кроці створіть програму для Windows 10 як **Startup project** і почніть налагодження. Visual Studio перенаправить вас до веб-програми, як тільки точка зупину буде досягнута.

Звичайно, якщо ви вирішите створити локальну JavaScript-сторінку у вашій програмі для Windows 10, ніщо не завадить зробити це. Отже, проект Westminster дає змогу публікувати не лише хост-програми, але й гібридні також.

Таким чином, як ви бачите, Windows 10 надає багато можливостей для веб-розробників. Ви можете легко інтегрувати веб-програму з програмою для Windows 10 і навіть опублікувати її як хост-програму без змін у коді клієнта.

Розділ 26.

## **Як заробити гроші в Магазині**

В останньому розділі першої книги ми обговорювали, як опублікувати свою першу програму в Магазині. Але як тільки ви навчитеся писати програми для Windows 10, то почнете думати про те, як заробити через Магазин гроші. Оскільки Магазин надає щодо цього безліч різних можливостей, ми поділимо всі програми на дві групи – платні та безкоштовні – і для кожної групи опишемо свої методи монетизації.

## Платні програми

Найпростіший спосіб спробувати отримати гроші з Магазину – опублікувати кілька платних програм. Але цей шлях не є ефективним, а в деяких випадках ви можете опублікувати хоч сто платних програм і нічого не отримати. Це тому, що люди не люблять купувати «кота в мішку», і якщо ви не продали попередню версію програми і не маєте величезної бази даних клієнтів, ніхто на вашу пропозицію не спокуситься. Ось чому ви повинні думати про додаткові заходи з вашого боку, щоб залучити якомога більше клієнтів. Ми не зираємося обговорювати рекламні кампанії або як працювати з компаніями, які генерують певний трафік. У цій галузі дуже важко рекомендувати щось конкретне, тому ми обговоримо можливості Магазину тільки з технічної точки зору.

## Близнюки

Можливо, багато людей в Microsoft не поділяє цей підхід, але в деяких випадках він чудово спрацьовує. Ідея проста: ви можете опублікувати платну і безкоштовну версію програми, як окремі пропозиції. Просто введіть ключове слово Lite в Магазині, і ви знайдете безліч програм, які опубліковані за допомогою цієї моделі. Як правило, розробники публікують безкоштовну версію з обмеженою функціональністю, що демонструє основні функції та сприяє продажу платної. Звичайно, ви можете об'єднати декілька підходів. Наприклад, опублікувати повнофункциональну безкоштовну версію, але із вбудованою реклами чи функцією продажу певних бонусів.

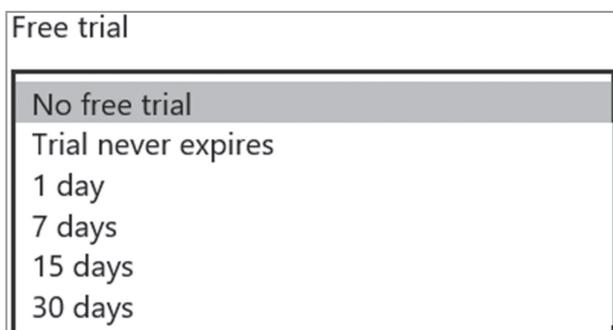
Основна причина для публікації двох окремих програм – упевненість, що користувачі не бажають встановлювати платні програми, навіть у пробному режимі. Крім того, дехто вважає, що користувачі будуть переглядати лише категорію безкоштовних програм і можуть просто не знайти платну. Проте це не зовсім так. Магазин містить наразі більше ста тисяч програм. Як правило, люди будуть використовувати пошук і ваша програма буде включена до списку, навіть якщо вона платна. На цьому етапі важливоскористатися всіма можливостями й показати найкращі картинки, графіку, чіткий опис і надати підтримку пробної версії. Але якщо ви вирішили опублікувати дві окремі версії, то можете зробити це, і тут немає, що обговорювати з технічного погляду. Все, що вам потрібно – реалі-

зувати код, який рекламируватиме платну версію програми. Для цього ви можете відображати різноманітні повідомлення, спливаючі сповіщення тощо. Але врешті-решт ви повинні надати можливість перейти до платної версії і купити її. Це можна зробити за допомогою класу **Launcher**:

```
Launcher.LaunchUriAsync(new Uri("ms-windows-store://pdp/?ProductId=<your id>"));
```

## Пробний режим

Можливо, це найкращий спосіб заохочити людей до встановлення платної програми. На Панелі керування ви можете знайти два типи пробних режимів: без терміну завершення та на певну кількість днів.



У першому випадку користувач може використовувати вашу програму стільки, скільки забажає. Зазвичай у цьому режимі користувачеві надається обмежений функціонал, але щойно він придбає програму, вам потрібно обробити цю подію та надати повний набір функцій. У програмах для бізнесу ви можете відключити деякі пункти меню, а в іграх – надати доступ лише до певних рівнів.

Інший варіант – надати повну функціональність, але на обмежену кількість днів.

Зверніть особливу увагу, що активація пробного режиму на певну кількість днів потребує мінімальних зусиль із вашого боку. Загалом вам не потрібно робити щось особливе; достатньо просто вибрати кількість днів. Windows буде стежити за вашою програмою і відразу по завершенні пробного терміну користувач не зможе її запустити. Натомість Windows відобразить повідомлення, що заохочує користувачів встановити програму. У разі пробного режиму з обмеженим функціоналом ви повинні будете реалізувати весь код, який перевіряє поточну модель ліцензування. Отже, насправді ви повинні заздалегідь вирішити, якою буде модель.

Щойно модель обрано, ви можете реалізовувати код, який додаватиме певний функціонал та відключатиме рекламу, коли користувач купить програму. Для цього ви можете використати клас **LicenseInformation**, що містить інформацію про режим використання програми. Клас **LicenseInformation** не містить жодних загальнодоступних конструкторів, але ви можете отримати об'єкт класу, використовуючи клас **CurrentApp** із простору імен **Windows.ApplicationModel.Store**:

```
var license = CurrentApp.LicenseInformation;
```

Використовуючи отримане посилання, ви можете перевірити, чи перебуває програма в пробному режимі, чи він ще активний (якщо задано пробний режим на певний термін), і дізнатися про дату його завершення. Загалом, якщо ви бажаєте відключити деякі функції або показати певні рекламні повідомлення, можна використовувати такий код:

```
if (license.IsTrial)
{
    //відключити певні функції або захотити до придбання програми
}
```

Код виглядає тривіально, але є проблема: клас **CurrentApp** не працюватиме належним чином, якщо ваша програма не опублікована в Магазині. Отже, вам не вдасться протестувати програму як слід. Щоб виправити цю проблему, ви можете використовувати спеціальний клас **CurrentAppSimulator**, який надає ту саму інформацію, що й клас **CurrentApp**, але дає також змогу задати налаштування для класу власноруч. Отже, поки ваша програма в розробці, можна використовувати клас **CurrentAppSimulator**, але щойно ви захочете опублікувати програму в Магазині, слово **Simulator** потрібно буде видалити.

Звичайно, клас нікому не потрібний, якщо не задати його налаштувань. Це можна зробити у файлі **WindowsStoreProxy.xml**, який потрібно створити. Однак завдяки зусиллям спеціалістів Microsoft вам не потрібно створювати цей файл «з нуля». Щойно ви використаєте клас **CurrentAppSimulator**, файл буде створено автоматично. Зазвичай його можна знайти у такій папці:

**C:\Users\<username>\AppData\Local\Packages\<appid>\LocalStorage\Microsoft\Windows Store\ApiData**

Якщо ви відкриєте цей файл, то побачите щось на кшталт такого XML-коду:

```
<?xml version="1.0" encoding="utf-16" ?>
<CurrentApp>
```

```
<ListingInformation>
    <App>
        <AppId>00000000-0000-0000-0000-000000000000</AppId>
        <LinkUri>http://apps.microsoft.com/webpdp/app/
            00000000-0000-0000-0000-000000000000</LinkUri>
        <CurrentMarket>en-US</CurrentMarket>
        <AgeRating>3</AgeRating>
        <MarketData xml:lang="en-us">
            <Name>AppName</Name>
            <Description>AppDescription</Description>
            <Price>1.00</Price>
            <CurrencySymbol>$</CurrencySymbol>
            <CurrencyCode>USD</CurrencyCode>
        </MarketData>
    </App>
    <Product ProductId="1" LicenseDuration="0"
        ProductType="Durable">
        <MarketData xml:lang="en-us">
            <Name>Product1Name</Name>
            <Price>1.00</Price>
            <CurrencySymbol>$</CurrencySymbol>
            <CurrencyCode>USD</CurrencyCode>
        </MarketData>
    </Product>
    <Product ProductId="2" LicenseDuration="0"
        ProductType="Consumable">
        <MarketData xml:lang="en-us">
            <Name>Product2Name</Name>
            <Price>1.00</Price>
            <CurrencySymbol>$</CurrencySymbol>
            <CurrencyCode>USD</CurrencyCode>
        </MarketData>
    </Product>
</ListingInformation>
<LicenseInformation>
    <App>
        <IsActive>true</IsActive>
        <IsTrial>true</IsTrial>
    </App>
    <Product ProductId="1">
        <IsActive>true</IsActive>
    </Product>
```

```
</LicenseInformation>
<ConsumableInformation>
    <Product ProductId="2" TransactionId="00000000-
        0000-0000-0000-000000000000" Status="Active" />
</ConsumableInformation>
</CurrentApp>
```

Як бачите, усю потрібну інформацію містить елемент **LicenseInformation** і з даними в ньому ви можете поекспериментувати. Усі інші частини потрібні для реалізації функціоналу покупок у програмі.

У деяких випадках розробникам потрібно дізнатися, чи змінився статус ліцензування. Інколи вам необхідно змінити конфігурацію чи ввімкнути всі можливості відряду. Для цього ви можете перехоплювати подію **LicenseChanged** й оновлювати інформацію про ліцензування та реалізовувати власну логіку в її обробнику. Проте, чесно кажучи, не надто багато програм використовують цей обробник подій.

Нарешті, якщо користувач вирішить купити вашу програму, можете використати клас **CurrentApp** замість **Launcher**. Лише викличте метод **RequestAppPurchaseAsync** і Windows зробить все за вас.

## Продажі!

Починаючи з Windows 10, Магазин підтримує декілька важливих можливостей із просуванням програм. Одна з них – це продажі, що дає змогу змінити ціну програми на певний період часу. Наприклад, ви можете зробити спеціальну акцію до свята або запустити промо-кампанію.

Для цього вам слід надіслати програму заново, але не потрібно додавати пакети чи щось іще – просто відкрийте вкладку **Pricing and availability** (Ціни та доступ) і задайте нову ціну:

New sale

Name	Sale price tier	Start	End	Markets
Name	Sale price tier*			
<input type="text"/>	<input type="button" value="Pick a price tier"/>			
Start date*	Start hour*	End date*	End hour*	
10/7/2015 <input type="button"/>	15:00 <input type="button"/> UTC	10/7/2015 <input type="button"/>	15:00 <input type="button"/> UTC	
Show custom market pricing options				
<input type="button" value="Done"/> <input type="button" value="Delete"/>				

Як бачите, можна обрати ціну, початкову й кінцеву дату для ціни, чи навіть різні ціни для різних країн. Звичайно, з точки зору програміста не потрібно нічого робити, але це важлива функція, яку не можна оминути.

## Промо-коди

Починаючи з Windows 10, Microsoft додав можливість надсилати запит на отримання промо-кодів для вашої програми. Це дуже важливо, якщо ви хочете надати посилання на вашу програму пресі чи подарувати вашу програму в якості призу тощо. Для замовлення промо-кодів вам потрібно лише відкрити вкладку **Monetization** (Монетизація) і на Панелі керування вибрати розділ **Promotional codes** (Промо-коди):

# New promotional codes order

App or in-app product (IAP)

Football.ua notifications



Order name

Quantity (up to 250)

Order codes

Cancel

Promotional codes are usually available for download within 60 minutes of placing the order, although some orders may take longer to process.

## Безкоштовні програми

Звичайно, ви можете використовувати безкоштовні програми як спосіб просування платної версії, як ми обговорювали раніше. Але є можливості заробити гроші навіть із безкоштовних програм. Основні методи отримання прибутку – це реклама та покупки в програмі. Розглянемо ці можливості детальніше.

### Реклама

Багато безкоштовних програм містять рекламу. Користувачі можуть завантажити програму безкоштовно, але ви можете отримати певні гроші завдяки розміщенню реклами.

Але перш ніж яким-небудь чином інтегрувати в програму рекламу, ви повинні вибрати постачальника, який підтримує Universal Windows Applications, надає можливість публікувати на цій платформі оголошення та отримувати певний дохід. Традиційно можна співпрацювати з Microsoft Ads, але на ринку є і багато інших постачальників. Починаючи з Windows 8.x і Windows Phone, ви можете

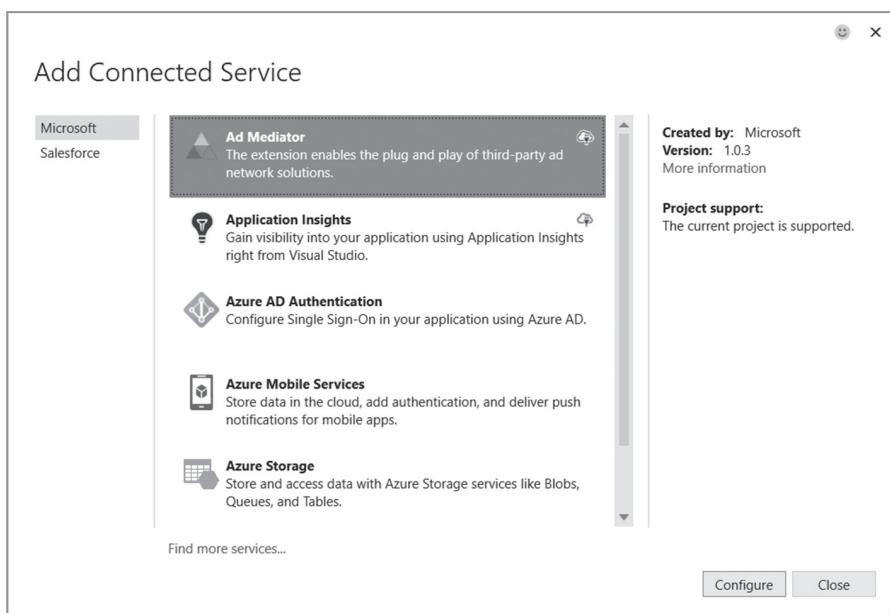
знайти SDK-комплекти для AdDuplex, Smaato і т. д. Не всі ці компанії оновили свої SDK до Windows 10, але принаймні AdDuplex оновила. У всяком разі, щойно ви отримаєте доступ до SDK, можете почати додавати до програми рекламні елементи керування. Але існує проблема. Різні постачальники працюють в різних країнах і, крім того, дуже важко передбачити, чиї оголошення працюватимуть краще. Ось чому Microsoft рекомендує користуватися послугами рекламного посередника, який дає можливість вибрати всі підтримувані мережі рекламних оголошень і налаштувати використання кожної мережі на Панелі керування.

Щоб почати працювати з рекламним посередником, необхідно встановити спеціальне розширення для Visual Studio. Ви можете знайти його за посиланням <https://visualstudiogallery.msdn.microsoft.com/401703a0-263e-4949-8f0f-738305d6ef4b>. Після встановлення розширення на панелі інструментів з'явиться новий елемент керування **AdMediatorControl**. Краще перетягнути його на форму, щоб Visual Studio заповнила всі необхідні властивості й додала посилання на бібліотеки.

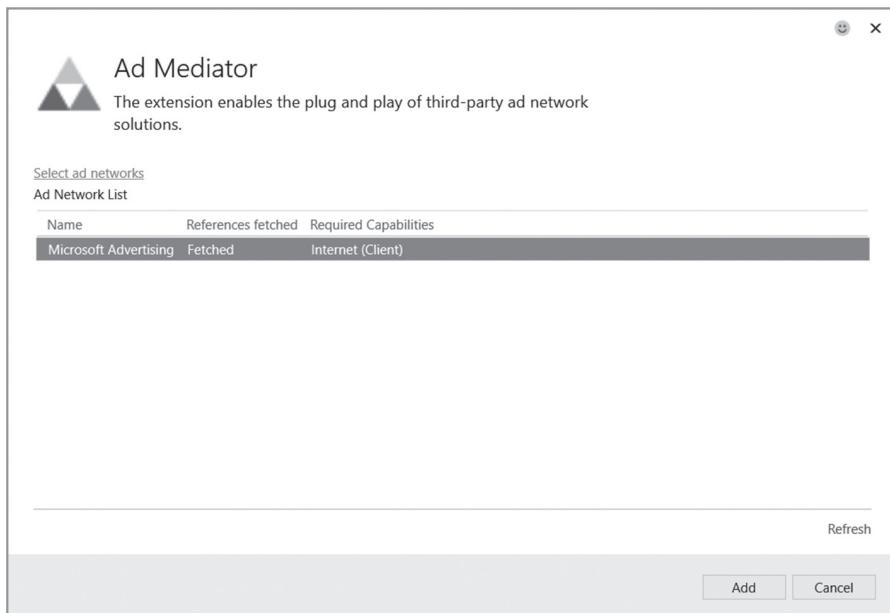
Якщо ви все правильно зробили, Visual Studio додасть рядок, подібний до цього:

```
<Universal:AdMediatorControl x:Name="AdMediator_3699BC"
Height="250" Id="AdMediator-Id-61C6D7FC-A97C-421B-9FAE-
9388E7A75DD8" Width="300"/>
```

Настав час налаштовувати посередника й створити для нього файл конфігурації. Ви можете зробити це у вікні **Connected Service** (Підключена служба). Просто відкрийте контекстне меню в Solution Explorer і виберіть пункт **Add > Connected Service**. Visual Studio запустить майстер, у якому вам потрібно вибрати варіант **Ad Mediator** (Рекламний посередник):



На наступному кроці ви маєте вибрати всіх рекламних провайдерів, яких забажаєте, і натиснути кнопку **Add**:



Щойно ви натиснете кнопку **Add**, Visual Studio створить файл конфігурації.

На зображенні ви бачите лише одного провайдера – Microsoft Advertising. Дійсно, на момент написання книги Ad Mediator підтримував тільки Microsoft Ads для UWP, але набір підтримуваних провайдерів постійно розширюється. Наприклад, для UWP існує елемент керування AdDuplex, але потрібно дивитися, чи є він серед розширень, оскільки для оновлення розширень розробникам потрібен певний час.

Для елементів керування існують події, що дають змогу обробляти будь-які пов'язані з ними помилки. Крім того, можна налаштувати ширину і висоту елемента керування відповідно до обраного типу банера. Але загалом ви можете спробувати завантажити програму в Магазин. Компанія Microsoft реалізувала взаємодію посередника реклами із Магазином. Отже, коли ви завантажите пакет, то зможете знайти всі налаштування.

Зауважте, що для змінення вартості реклами від певних провайдерів і для певних країн ви не повинні надсилати програму в Магазин заново – можна просто відкрити Панель керування і налаштувати всі потрібні параметри на вкладці Монетизація.

## Покупки в програмі

Останнє, що ми обговоримо в цьому розділі – це покупки в програмі. Завдяки цій функції ви можете продати додаткові інструменти або цифровий контент безпосередньо з програми. Ви можете знайти багато ігор, які продають нові рівні, мають внутрішні гроші для придбання оновлень тощо. Ось чому покупки в програмі слід поділити на дві групи: стандартні та витратні. Стандартні – це покупки незмінних речей, таких як нові рівні або доступ до деяких додаткових функцій, а витратні – це покупки золота, монет та інших речей, які ви можете витрачати в програмі, здійснюючи покупки знову і знову.

Windows 10 підтримує обидва типи покупок і в обох випадках вам необхідно пройти два етапи.

На першому етапі слід створити всі продукти, які ви збираєтесь продавати в програмі. Для цього ви можете відкрити вкладку IAPs і створити новий продукт:

The screenshot shows the Windows Store submission interface for a product named "Football.ua notifications". The left sidebar includes links for App overview, Analytics, Submissions, IAPs (10coins), Submission 1, Monetization, and Services. The main content area displays "Submission 1" with tabs for Properties (Not started), Pricing and availability (Not started), Descriptions (Not started), and Add languages.

Після того як ви створите продукт, необхідно заповнити всі параметри. Найважливішою є вкладка властивостей, що містить основну інформацію про тип покупки, типу вмісту і ключові слова:

The screenshot shows the "Product type\*" section where "Consumable" is selected from a dropdown. A note indicates that changes will take effect after the IAP is published. Below it is the "Content type\*" section with a dropdown menu. The "Keywords" section contains four input fields for entering up to ten keywords, with a note specifying a 30-character limit per keyword and noting that it's not available for Windows 8 and 8.1 packages.

Як ми вже згадували, ви повинні вибрати між покупками в програмі, що надається на постійній основі і тими, які можна витрачати. Крім того, Магазин підтримує тимчасові товари тривалого користування.

Створивши в Магазині всі продукти, можна починати розробку коду, що реалізує функціональність покупок у програмі.

Для реалізації функціональності товарів тривалого користування придатний той самий клас **LicenseInformation**, але в цьому випадку вам потрібно буде скористатися колекцією **ProductLicenses**:

```
var license = CurrentApp.LicenseInformation;
if (!license.ProductLicenses["coins10"].IsActive)
{
    try
    {
        await CurrentApp.RequestProductPurchaseAsync
            ("coins10", false);
```

```

    }
    catch (Exception)
    {
        }

    }
}
else
{
}
}

```

Звичайно, щоб перевірити програму, можете скористатися класом **CurrentAppSimulator**, як і раніше. Всередині **ListingInformation** у файлі **WindowsStoreProxy.xml** ви можете визначити список товарів, використовуючи елемент **Product**:

```

<Product ProductId="2" LicenseDuration="0"
ProductType="Consumable">
    <MarketData xml:lang="en-us">
        <Name>Product2Name</Name>
        <Price>1.00</Price>
        <CurrencySymbol>$</CurrencySymbol>
        <CurrencyCode>USD</CurrencyCode>
    </MarketData>
</Product>

```

Для емуляції продуктів, які користувач уже купив, можна скористатися елементом **LicensesInformation** із елементом **Product** всередині:

```

<Product ProductId="1">
    <IsActive>true</IsActive>
</Product>

```

Для покупок, що витрачаються, ви можете використовувати той самий код, але користувач не буде в змозі купити певну річ знову, поки ваша програма не повідомить Магазин про завершення покупки:

```

var result = await CurrentApp.ReportConsumableFulfillmentAsync
("coins10", transactionId);

```

Щоб отримати ідентифікатор транзакції **transactionId**, використайте результат, який повертає метод **RequestProductPurchaseAsync**.

Нарешті, для емуляції транзакцій і покупок, які можна витрачати, у файлі **WindowsStoreProxy.xml** є ще один елемент – **ConsumableInformation**:

```
<ConsumableInformation>
    <Product ProductId="2" TransactionId="00000000-0000-
        0000-0000-000000000000" Status="Active" />
</ConsumableInformation>
```

Розділ 27.

## **Служби Windows Notification Services**

У розділі 11 першої книги ми обговорювали локальні сповіщення, зокрема Toast-сповіщення та сповіщення плиток. Є багато завдань, які можна вирішити завдяки локальним сповіщенням, але інколи нам потрібно дещо більше. Наприклад, ви хочете повідомити користувача про статус футбольного матчу. Зробити це за допомогою локальних сповіщень не вдастся, тому що ваша програма не може зазирати у майбутнє, щоб завантажити інформацію про перебіг матчу, тільки-но користувач запустить її. Звичайно, якщо програма активна, то можна перевіряти служби час від часу, шукаючи оновлені дані, але якщо користувач закриє програму, то вже нічого не вдієш. Ось чому ОС Windows дає змогу отримувати зовнішні сповіщення та надає для цього інфраструктуру.

Звичайно, говорячи про зовнішні сповіщення для пристрою, слід припускати, що він підключений до мережі Інтернет. На будь-який пристрій сповіщення повинні бути доставлені в безпечному режимі, в однаковому форматі, і без будь-якого перенавантаження трафіку (це особливо важливо для мобільних пристройів). Ось чому не можна реалізувати службу, яка буде доставляти сповіщення на конкретний пристрій. Натомість слід використовувати спеціальну службу, яка є частиною інфраструктури операційної системи – Windows Notification Services (WNS).

Насправді WNS виглядає як шлюз, що підтримує зв'язок між вашою службою та екземплярами вашої програми.

Якщо ви хочете реалізувати функціональність, яка використовує служби Windows Notification Services, краще почати з самої програми для Windows 10. На першому етапі екземпляр програми повинен почати процес реєстрації (кроки 1 і 2 на рисунку). Це нескладний процес: програмі просто потрібно відправити WNS-повідомлення про те, що вона хоче отримувати сповіщення. Реалізувати це можна за допомогою пари рядків коду:

```
PushNotificationChannel channel = null;

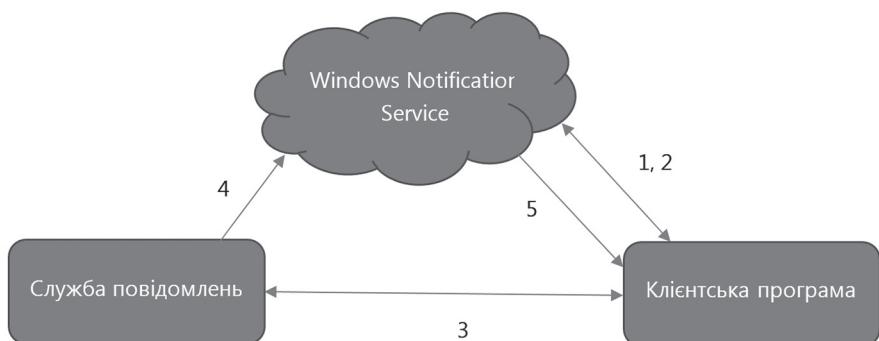
try
{
    channel = await PushNotificationChannelManager.
        CreatePushNotificationChannelForApplicationAsync();
}

catch (Exception ex)
{
    // Якщо неможливо створити канал.
}
```

Як бачите, ми використали клас **PushNotificationChannelManager**, який дає змогу зареєструвати екземпляр і повернути спеціальний об'єкт – канал для push-сповіщень. Останній є спеціальним URI, що містить всю необхідну інформацію про те, як надіслати сповіщення програмі. Оскільки об'єкт (URI) повертається самій програмі, вона може вирішувати, як надавати спільній доступ до цього URI.

Отже, на наступному кроці необхідно надати URI зовнішній службі (3). Зазвичай всім активним клієнтам сповіщення надсилає ваша власна служба. У службі можна реалізувати будь-що, оскільки вона не повинна застосовувати якусь особливу технологію. Можна використовувати будь-яку базу даних для зберігання URI та пов'язаної інформації, і, коли знадобиться надіслати повідомлення, можна вибрати тільки ті URI, які задовольняють вашим вимогам. У деяких випадках потрібно буде надіслати повідомлення лише на конкретний пристрій, а в інших – всім підписникам служби. Але за будь-яких умов, якщо хочете надіслати повідомлення всім користувачам з вашої бази даних, доведеться робити це по черзі. Звичайно, це може вплинути на трафік сервера, і якщо абонентів багато, ви повинні бути в змозі керувати великою кількістю серверів і підтримувати чергу повідомлень. Але всі ці проблеми можна вирішити завдяки службі Microsoft Azure, яку ми обговоримо в наступному розділі.

Коли вашій службі знадобиться надіслати повідомлення, вона буде використовувати URI, наданий екземпляром програми. Звичайно, цей URI повинен бути використаний для пересилання повідомлення за допомогою WNS, а не самої програми (4). Ваша служба підключиться до WNS, надасть URI і повідомлення, а WNS завершить процес (5).



Найбільш складним завданням є надсилання повідомлення від служби до WNS. Щоб гарантувати, що URI не будуть перехоплені, WNS потребує додаткової автентифікації від служби. Автентифікація базується на протоколі OAuth 2.0 і вимагає спеціальний ідентифікатор програми та секретний ключ. Таким чином,

для реалізації служби необхідно почати з процесу автентифікації, але спочатку ви повинні отримати ідентифікатор і ключ.

Ідентифікатор і ключ можна отримати лише в процесі публікації. Отже, щоб перевірити надсилання сповіщень, необхідно запитати ім'я для програми в Магазині та асоціювати саму програму з Магазином. До встановлення такої асоціації WNS належного URI програмі не надасть.

Установивши таку асоціацію, відкрийте на Панелі керування розділ **Services > Push notifications:**

## Push notifications

### Windows Push Notification Services (WNS) and Microsoft Azure Mobile Services

The Windows Push Notification Services (WNS) enables you to send toast, tile, badge, and raw updates from your own cloud service. Learn more

If you have an existing WNS solution or need to update your current client secret, visit the Live Services site

You can also use Microsoft Azure Mobile Services to send push notifications, authenticate and manage app users, and store app data in the cloud. Sign in to your Microsoft Azure account or sign up now to add services to up to ten apps for free.

Clicking Live Services site you will be navigated to a page there you can find the id and key:

## Football.ua notifications

The screenshot shows the Windows Store App Settings page. On the left, a sidebar lists 'Settings', 'Basic Information', 'API Settings', 'App Settings' (which is selected), and 'Localization'. The main content area displays the following information:

- Package SID:** ms-app://s-1-15-2-225 4688822-3077577576-1314037951-305615383-2770 01521-20 7243510-2058990373  
Link to different app
- Application identity:** <Identity>  
Name=35581SBStudio.Football.uanotifications  
Publisher=CN=5E452BF3-24C3-41DD-A595-44622280C3F7</Identity>  
This is the unique identifier for your Windows Store app.
- Client ID:** 0000000040131D26  
This is a unique identifier for your application.
- Client secret:** QIRj5H wQLOM IS97MUfE twbln6j7Xw  
For security purposes, don't share your client secret with anyone.

Наступний крок трохи складніший. Коли ви отримали ідентифікатор і ключ, потрібно скористатися ними на сайті <https://login.live.com/accesstoken.srf> для автентифікації за протоколом OAuth 2.0. Звичайно, код залежить від обраної вами технології, але знайти приклад не дуже важко.

Використовуючи OAuth 2.0, ви отримаєте маркер автентифікації. Якщо ви отримали маркер, процес автентифікації завершено, і можна надсилати повідомлення. Для цього можна скористатися URI, який було надано клієнту на початку, і згенерувати POST-повідомлення за допомогою маркера автентифікації, щоб сформувати заголовок у потрібному форматі:

```
Authorization: Bearer  
EgAaAQMAAAAEgAAAACoAAPzCGedIbQb9vRfPF2Lxy3K//QZB79mLTgK  
X-WNS-RequestForStatus: true  
X-WNS-Type: wns/toast  
Content-Type: text/xml  
Host: db3.notify.windows.com  
Content-Length: 196
```

Для вмісту можна використати XML-документ у тому самому форматі, що й для локальних сповіщень.

На нашу думку, краще не реалізовувати власний сервер, який буде використовувати WNS для надсилання сповіщень, адже це може зробити для вас Microsoft Azure, і коштувати це буде не так дорого, як у разі використання локальних служб, трафіку тощо. Ось чому в цьому розділі ми обговорювали лише ідею, а інформацію про те, як реалізувати готову до використання систему сповіщень, ви знайдете в наступному розділі.



Розділ 28.

# **Надсилання повідомлень за допомогою служб Microsoft Azure**

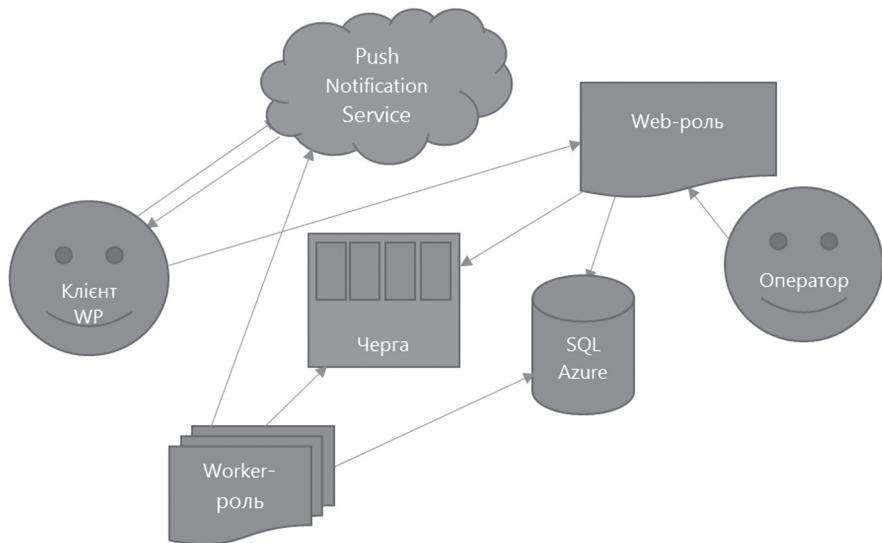
Microsoft Azure надає доступ до багатьох різних служб, більшість із яких допомагають підвищити продуктивність ваших програм. Оскільки ця книга не про Azure, ми не розповідатимемо про всі можливості пакета Azure SDK і всі пропоновані служби. Однак докладно розглянути принаймні одну службу буде цілком доречно, продовжуючи тему, яку ми почали в попередньому розділі.

Додаткові відомості про Azure та опис усіх служб можна знайти на сайті <http://azure.microsoft.com>. Там же можна створити тимчасовий пробний обліковий запис. Майте на увазі: для застосування на практиці відомостей цього розділу також потрібно буде створити ознайомчий обліковий запис Azure.

## Вихідний сценарій розробки

Ця історія почалася 2010 року. Власникам найбільшого футбольного порталу України потрібна була служба, яка б у режимі реального часу сповіщала абонентів із різних країн світу (у першу чергу користувачів Windows Phone і пристройів із iOS) про ключові моменти чергового матчу. Розробити простеньку програму для Windows Phone, яка б отримувала новини й давала змогу відображати їх у вигляді спливаючих сповіщень, було нескладно, а от із серверною частиною програми виникли проблеми. Передусім треба було знайти достатню кількість апаратних ресурсів і забезпечити потужний канал зв'язку з Інтернетом. Під час матчів портал відвідують сотні тисяч уболівальників, тож, аби надіслати їм усім одночасно сповіщення через Microsoft Push Notification Server, потрібні додаткові сервери та інтернет-канал із надзвичайно високою пропускною спроможністю. Саме тому ми вирішили скористатися службами Azure.

У 2010 році корпорація Microsoft уже представила свої хмарні служби з ролями Web (взаємодія з користувачами) і Worker (обробка даних), онлайнові сховища (таблиці, BLOB-об'єкти і черги) і SQL Azure. Нам знадобилися всі ці служби. Схему рішення проілюстровано нижче.



Перш за все ми створили веб-службу (Web-роль) для зберігання унікальних ідентифікаторів каналів сповіщень (Notification Channel URL), присвоєних телефонам Windows Phone. Унікальний URI створюється службою Microsoft Push Notification Service для кожного пристрію і може використовуватися для надсилання повідомень на пристрій. Тобто, щоб передати сповіщення на 100 пристріїв, необхідно мати 100 URL. Для зберігання цих ідентифікаторів і відомостей про користувачів ми використовували базу даних SQL Azure.

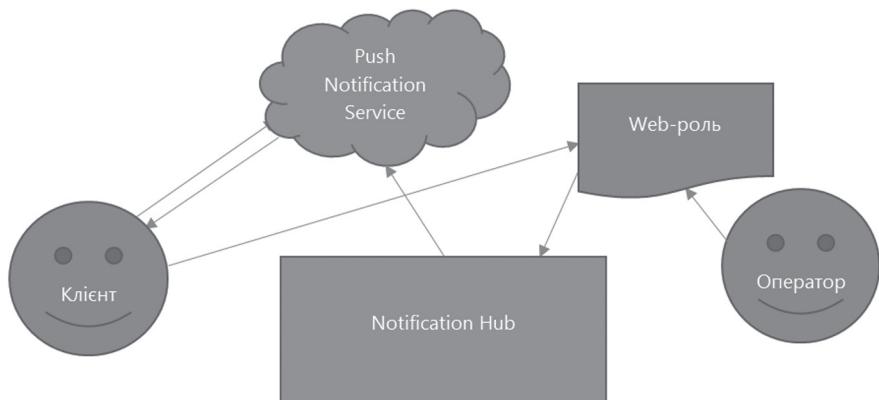
На другому етапі ми створили ще одну службу (також Web-роль, тож можна було скористатися службою, описаною в попередньому абзаці), цього разу для отримання повідомень від оператора на боці клієнта (це може бути працівник або програмний засіб). Ці повідомлення містили інформацію про попередні результати матчу і мали надсилатися абонентам порталу. Застосувати для розсилання новостворену службу ми не могли, адже процес передавання повідомлень великої кількості абонентів досить тривалий і може забрати до кількох годин, а відзвітувати оператору про отримання треба якнайшвидше. Тому служби з Web-роллю зберігають нові повідомлення в черзі сховища Azure. Така структура підтримує одночасний багатоканальний доступ до збережених даних, що досить корисно, якщо для обробки повідомлень використовується кілька екземплярів Worker-ролі.

Насамкінець було створено Worker-роль для надсилання повідомлення абонентам. Ми забезпечили можливість контролювати чергу і змінювати кількість Worker-служб відповідно до кількості матчів, абонентів і повідомлень у черзі.

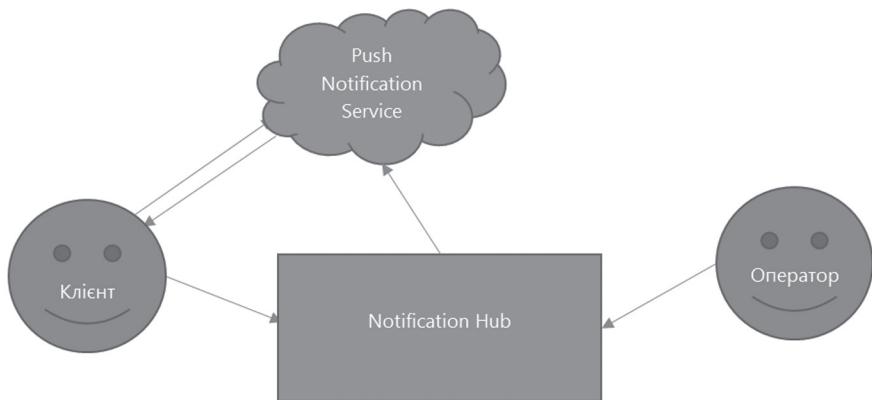
Запропоноване рішення було надійним і гнучким, однак мало свої недоліки. Перше, на що варто звернути увагу, це складність структури і неможливість її реалізувати без певних знань Azure. Далі слід згадати про немалу вартість окремих ресурсів Azure, зокрема хмарної служби, сховища, бази даних SQL Azure тощо. Це досить важомі фактори, які не одного розробника змусять відмовитися від ідеї реалізації простих стандартних рішень за допомогою Azure (зокрема й служб надсилання повідомлень в клієнтські програми). Однак у ті часи напрям розробок з використанням ресурсів цільової платформи на Azure тільки освоювався. Відтоді корпорація Microsoft розробила кілька якісних готових рішень. Це, зокрема, служби Azure Mobile Services і Mobile Apps, які використовують компонент Notification Hub. Розглянемо згаданий компонент докладніше.

## Основні відомості про Notification Hub

Notification Hub дає змогу створювати рішення, не заглиблюючись в технічні деталі. Не витрачаючи час на вивчення баз даних SQL Azure, Worker-ролей, черг і клієнтської платформи, можна застосовувати Web-ролі для автентифікації, форматування повідомлень тощо.



Для найпростішого сценарію потрібне лише Notification Hub, який надає інтерфейси для надсилання повідомлень за допомогою програм оператора та оновлення каналу сповіщень із клієнтських пристрійв.



Необхідно лише створити та налаштувати службу Notification Hub в обліковому записі Azure. До неї можна буде прив'язати скільки завгодно абонентських пристрій і надсилати на них сповіщення. Notification Hub підтримує різні типи клієнтських платформ: Windows, iOS, Android та інші. Можна сказати, що Notification Hub – це така чарівна скринька, що дає змогу пов'язувати клієнтські програми і програми операторів. Це стандартний готовий сценарій від корпорації Microsoft.

Далі ми перелічимо кілька можливостей, які можуть зацікавити розробників.

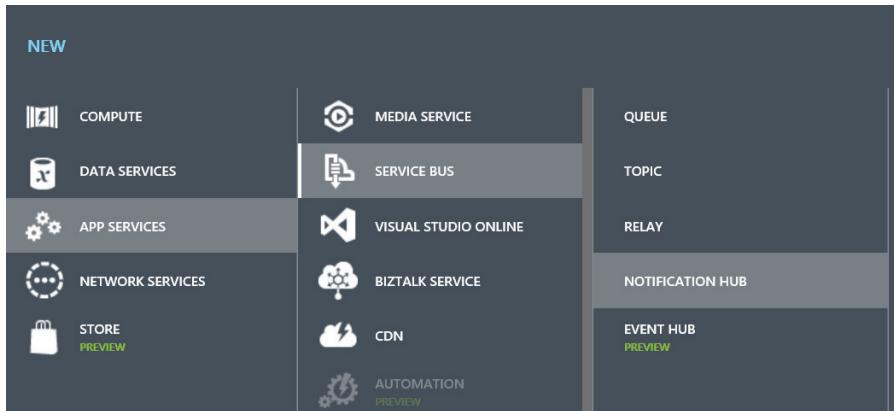
- **Універсальність.** Notification Hub підтримує більшість доступних на ринку платформ. Його можна налаштувати на надсилання повідомлень на пристрой з Windows 8, Windows Phone, iOS, Android, Kindle і т. д. API з підтримкою цих платформ дає змогу зареєструвати пристрой будь-якого типу.
- **Підтримка міток.** Мітки уможливлюють надсилання повідомлень лише вибраний підмножині абонентів. Наприклад, користувач може підписатися лише на сповіщення щодо футбольних матчів чемпіонату країни.
- **Підтримка шаблонів.** Повідомлення можна персоналізувати і локалізувати за допомогою шаблонів. Наприклад, оператор створює лише одне базове повідомлення, а Notification Hub на основі шаблонів надсилає його в різних варіантах на різні пристрой. Скажімо, одні користувачі отримають текст англійською, а інші – українською.
- **Керування реєстрацією пристройів.** Можна забезпечити реєстрацію за допомогою коду, щоб користувач міг попередньо авторизуватися. Це важливо в разі надання платних послуг.
- **Надсилання повідомлень за графіком.** Можна заздалегідь вказати час надсилання повідомлень.

Отже, необхідно розуміти, з якою самою метою застосовується Notification Hub. Далі ми докладно розкажемо про використання його окремих функцій.

## Створення простого сценарію з Notification Hub

Ми вже ознайомилися з основними можливостями Notification Hub, але переважно як з альтернативами для базових сценаріїв. А зараз викладемо покрокові інструкції з його використання і докладніше зупинимося на окремих характеристиках.

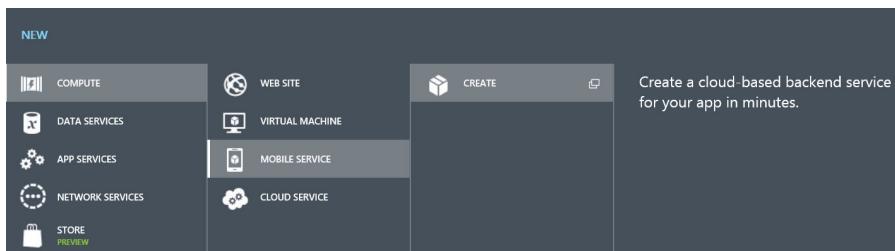
Насамперед зазначимо, що у корпорації Microsoft є кілька готових рішень, що використовуються у стандартних сценаріях для мобільних пристройів. Це служби Azure Mobile Services і Mobile Apps. Notification Hub із цими службами НІЯК не зв'язаний. З ним можна працювати окремо, наприклад використовуючи вкладку **App Services** (Служби програм).



З іншого боку, поєднавши Notification Hub зі службами Mobile Services чи Mobile Apps, ми отримаємо додаткові можливості. Тож пропоную почати зі створення служби Mobile Services, для якої Notification Hub буде створено автоматично.

Основна відмінність в роботі з Mobile Services, порівняно з Mobile Apps, полягає в тому, що для першої служби використовується старий портал керування (<https://manage.windowsazure.com>), а для другої – новий (<https://portal.azure.com>).

Почнемо роботу з Mobile Services.



Цей крок дуже простий. Слід вибрати ім'я (URL-адресу), базу даних SQL і регіон. У полі **Backend** вказується мова коду серверної частини програми. Вибрати можна між варіантами .NET (майже напевно – C#) та JavaScript. Не всі люблять JavaScript, але для написання сценаріїв Notification Hub радимо використовувати саме цю мову, тому що кількість таких сценаріїв, як і обсяг коду для них, зовсім невеликі. Власне кажучи, необхідно буде створити лише один сценарій – для процесу реєстрації нових пристройів. Сценарій на JavaScript буде легше розгорнути, налаштувавши на панелі керування Azure низку спеціальних параметрів і застосувавши редактор JavaScript. В наступному розділі ми розберемо сценарії на JavaScript і C# докладніше, щоб ви могли використовувати будь-яку з цих мов.

### NEW MOBILE SERVICE

## Create a Mobile Service

### URL

.azure-mobile.net

### DATABASE



### REGION



### BACKEND



ADVANCED PUSH NOTIFICATION SETTINGS

Натиснувши кнопку **OK**, ви за 1–2 хвилини матимете готові служби Mobile Service і Notification Hub (у разі використання Mobile Apps необхідно створити Notification Hub на новому порталі). Відкрити панель керування **Hub** можна кіль-

кома способами, але краще це робити на вкладці **Mobile Services**. Тут зручно редагувати сценарій реєстрації на JavaScript і змінювати дозволи для реєстрованих пристрій, налаштовуючи додаткові правила безпеки. Для дозволів можна вибрати один із перелічених далі варіантів.

- **Everyone (Усі). До реєстрації допускаються всі пристрої.** Це основний варіант для багатьох сценаріїв, тому що сама по собі реєстрація не дає змогу отримувати повідомлення, якщо застосовано неправильний сертифікат (або секретний ключ клієнт чи ідентифікатор безпеки пакета для Windows 8).
- **Anybody with application key** (Усі з ключем програми). Реєструються лише пристрої, які надіслали запит із ключем програми. Це не вельми надійний метод, оскільки такі ключі зазвичай зберігаються в самих програмах і викрасти їх дуже легко. Для кращого захисту потрібно включити в програму механізм автентифікації.
- **Only Authenticated Users** (Лише автентифіковані користувачі). Користувачі можуть зареєструватися лише після автентифікації через підтримувану службу: Facebook, Twitter, Google, обліковий запис Microsoft, Azure Active Directory тощо.
- **Only Script and Admins** (Тільки сценарії й адміністратори). Доступ до служби отримують лише сценарії з головним ключем і користувачі з правами адміністратора (через портал керування).

Майте на увазі, що в разі вибору мови .NET розділ дозволів для реєстрації пристрій не відображається. Натомість, якщо користується C#, дозволи можна надати у файлі **WebApiConfig.cs**.

The screenshot shows the Azure Mobile Services interface for the 'test-ms-sbaydach' application. On the left is a sidebar with various icons representing different service components. The main area is titled 'test-ms-sbaydach' and shows the 'notification hub' configuration. It includes sections for 'registration endpoints' (with a dropdown for 'PERMISSION' set to 'Anybody with the Application Key'), 'REGISTRATION SCRIPT' (with an 'EDIT SCRIPT' button), 'windows application credentials' (with fields for 'CLIENT SECRET' and 'PACKAGE SID'), and 'windows phone notification settings (mpns)'. A blue bar at the bottom of the main content area contains the text 'Windows Phone 8.1'.

Обговорюючи параметри безпеки, ми згадували ключ програми і головний ключ. Їх можна знайти на панелі керування **Mobile Service** і швидко змінити в разі небажаного розкриття інформації.

## Manage Access Keys

When you regenerate your access keys, client apps might lose the ability to connect to your mobile service.

### APPLICATION KEY

olRnaomiNKC0tqn1wLDFgSMkVyVKNc18  

### MASTER KEY

MubHNhzYjZFIvZIMyvCjqpPwzHsBsC46  



У разі використання бібліотек Media Services варто докладніше ознайомитися з параметрами безпеки цієї служби, однак із Notification Hub можна працювати й окремо, навіть якщо він інтегрований з Media Services у вашій підписці.

Тим, хто працює безпосередньо з Notification Hub, згадувані ключі не потрібні, однак їм доведеться застосовувати політики Notification Hub. На панелі керування Notification Hub (вкладка конфігурації) вже задано кілька політик доступу. За потреби можна змінити їх або створити нові. Ці політики слід застосовувати для прямого підключення Notification Hub. Додатково для кожної політики передбачено два ключі, які мають використовуватися в рядках підключення. Ці рядки можна знайти на панелі керування Notification Hub і вибрати потрібний з урахуванням наявних дозволів. Скажімо, якщо потрібен рядок підключення для клієнтських пристрій, які будуть лише прослуховувати канал сповіщень, можна скористатися рядком **DefaultListenAccessSignature**. Якщо ж створюється служба на боці сервера, яка ще й надсилає сповіщення, слід вибрати рядок із дозволом на надсилання.

### shared access policies

NAME	PERMISSIONS
DefaultListenSharedAccessSignature	Listen
DefaultFullSharedAccessSignature	Manage, Send, Listen
ZumoManagementSasKey	Manage, Send, Listen
<i>NEW POLICY NAME</i>	

### shared access key generator

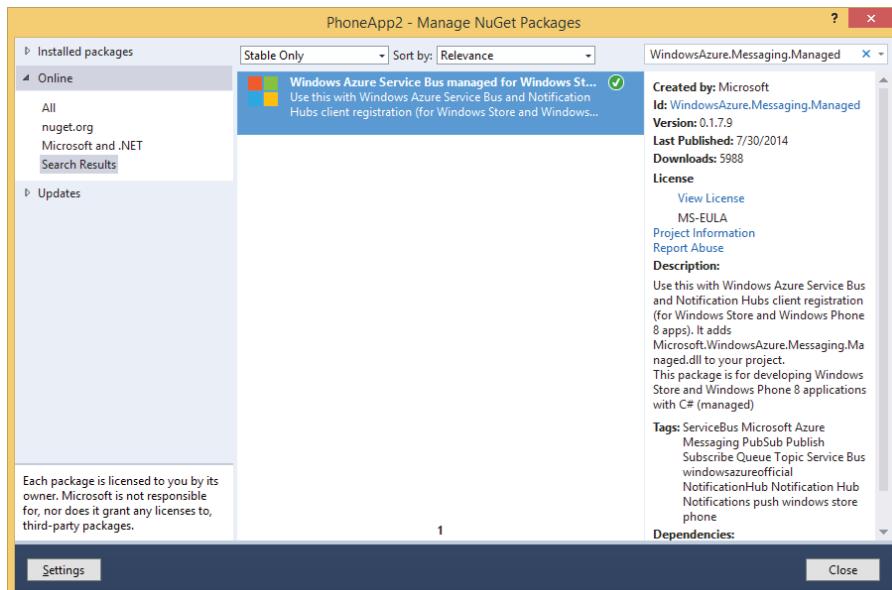
POLICY NAME	DefaultListenSharedAccessSignature
PRIMARY KEY	r8cuE03unwoQNEyj3OS4g5IH/4lqAPdWlyGWaIZGn4=
SECONDARY KEY	o+t4i2MG9gSo/P84S/1W1AKUmYHQqDzhnt+XhZaiOON

**Regenerate** **Regenerate**

Що ж, тепер можна створити невелику програму, яка просто отримуватиме сповіщення. З цією метою виберемо програму для Windows Phone з використанням Silverlight. Такі програми не потребують реєстрації в Магазині, зате для них необхідно встановити пропорець **Enable unauthenticated push notifications** (Увімкнути **неавтентифіковані push-сповіщення**) на вкладці конфігурації. Це дасть змогу надсилати до 500 повідомлень без сертифіката – цілком достатньо для тестування.

Тепер відкриємо Visual Studio і створимо програму для Windows Phone з використанням Silverlight. Вона, на відміну від програм, створених у середовищі виконання Windows, буде використовувати службу старого типу, проте не вимагатиме оформлення сертифіката, про який ішлося вище.

Ми спробуємо налаштувати безпосередній обмін даними з Notification Hub, а також зв'язок через Mobile Service. Почнемо з першого методу. За допомогою інструменту NuGet додамо збірку **WindowsAzure.Messaging.Managed**, що уможливить надсилання даних до Notification Hub без створення коду JSON і залучення класів **WebRequest** і **WebResponse**.



Тепер можна додати код. Відкриємо файл **App.xaml.cs** і змінимо метод **Application\_Launching** таким чином:

```
private async void Application_Launching(object sender,
    LaunchingEventArgs e)
{
    var channel = HttpNotificationChannel.Find("MyPushChannel12");
    if (channel == null)
    {
        channel = new HttpNotificationChannel("MyPushChannel12");
        channel.Open();
        channel.BindToShellToast();
    }

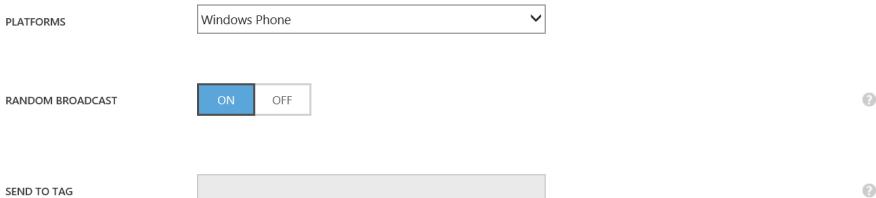
    channel.ChannelUriUpdated +=
        new EventHandler<NotificationChannelUriEventArgs>(async
            (o, args) =>
    {

```

```
        var hub = new NotificationHub("test-ms-sbaydachhub",
            "<connection string>");
        await hub.RegisterNativeAsync(args.ChannelUri.ToString());
    });
}
```

За допомогою цього коду ми отримаємо канал сповіщень (Notification Channel) із MPNS й оновимо канал в нашому Notification Hub. Готово! Запустіть програму на своєму телефоні і спробуйте надіслати повідомлення, скориставшись вкладкою **Debug** на порталі керування Azure.

### settings



### body

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <wp:Notification xmlns:wp="WPNotification">
3 <wp:Toast>
4 <wp:Text1>NotificationHub</wp:Text1>
5 <wp:Text2>Test message</wp:Text2>
6 </wp:Toast>
```

Щоб зареєструвати пристрій для отримання сповіщень за допомогою служби Mobile Services, додайте до проекту пакет Azure Mobile Service і змініть метод **Application\_Launching** таким чином:

```
private async void Application_Launching(object sender,
LaunchingEventArgs e)
{
    var channel = HttpNotificationChannel.Find("MyPushChannel2");
    if (channel == null)
    {
        channel = new HttpNotificationChannel("MyPushChannel2");
        channel.Open();
        channel.BindToShellToast();
    }
}
```

```

channel.ChannelUriUpdated +=
    new EventHandler<NotificationChannelUriEventArgs> (async
(o, args) =>
{
    MobileServiceClient MobileService =
        new MobileServiceClient(
            "https://test-ms-sbaydach.azure-mobile.net/"

        );
    await MobileService.GetPush().
        RegisterNativeAsync(channel.ChannelUri.
            ToString());
});
}

```

Зауважте: у цьому коді не використовується рядок підключення з Notification Hub, однак він не працюватиме, поки не буде надано дозвіл **Everyone** (Усі) для реєстрації або вказано ключ програми як другий параметр конструктора **MobileServiceClient**.

## Mobile Service API та Notification Hub API

У попередній темі ми стисло описали можливості Notification Hub і деякі методи їх використання. Настав час створити програму в середовищі виконання Windows. Це буде простенька клієнтська програма для надсилання сповіщень.

Не будемо залучати віртуальні рішення, а спробуємо відтворити програму, яку описували вище, коли йшлося про Notification Hub та розробку з використанням базових ресурсів платформи. Це буде універсальна програма, за допомогою якої зручно отримувати найсвіжіші новини про футбольні ігри й актуальні дані про перебіг поточного матчу.

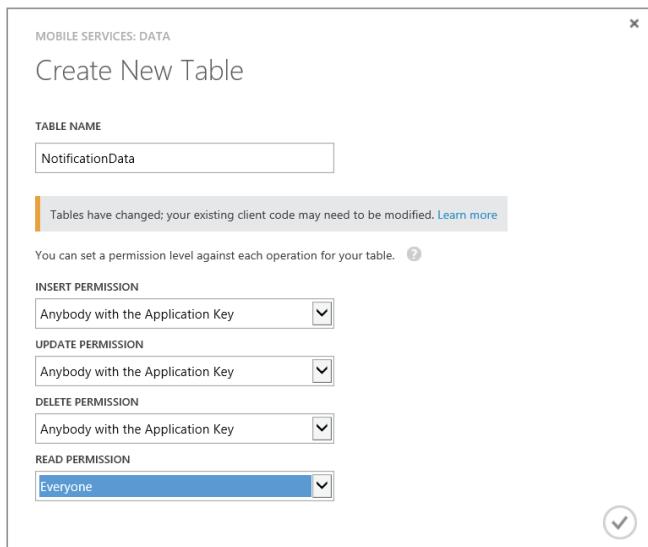
Перш ніж узятися до написання коду, необхідно вибрати спосіб взаємодії з Notification Hub. До нього, як уже було сказано, можна підключатися безпосередньо або через інфраструктуру Mobile Services. Ці способи схожі, але в нашому випадку ми повинні зберігати архів повідомлень в Azure, оскільки програма повинна показувати як нові повідомлення, так і старі. Ось чому нам потрібно створити таблицю для повідомлень на боці Azure. Крім того, ми повинні створити службу, яка допоможе оновлювати таблицю, а також думати про безпеку.

Тому, якщо ми використовуємо Notification Hub безпосередньо, ми все ще повинні реалізувати багато речей, але якщо ми використовуємо Mobile Services, це може допомогти нам, і ми уникнемо додаткової роботи. Найбільш важливою особливістю Mobile Services є інфраструктура для керування даними. Завдяки Mobile Services у нас є простий спосіб створення таблиць, внутрішнього керування даними, написання власної бізнес-логіки, яка забезпечить тригери для таких операцій, як вставляння, видалення і т. д. Крім того, інфраструктура Mobile Services підтримує деякі механізми безпеки.

Зверніть увагу, що інтерфейс API служби Mobile Services не дає зможи надсилати повідомлення за межами інфраструктури Mobile Services. Ось чому таблиці і тригери настільки важливі. Насправді наша серверна програма працюватиме з таблицею, надсилаючи нові дані для її заповнення, а тригер допоможе передавати сповіщення на зареєстровані пристрої.

## Таблиці

Ми повинні почати нашу розробку з таблиць. Зараз нам потрібна лише одна таблиця для зберігання повідомлень. Щоб створити цю таблицю, ви можете використовувати портал керування Azure, де слід вказати ім'я і надати дозволи на такі дії, як вставлення, видалення, оновлення та вибір. Оскільки ми будемо змінювати дані в таблиці тільки з нашої серверної програми і показувати наш архів для всіх клієнтів без спеціальних дозволів, можна налаштувати нашу таблицю, щоб використовувати ключ програми для оновлення, видалення і вставлення, а також дозвіл **Everyone** (Усі) для вибору.



Натисніть кнопку **OK** – і таблиця готова до введення даних, хоча ми ще не створили стовпців. Якщо відкрити вкладку **COLUMNS** (Стовпці), там буде кілька попередньо створених стовпців.

## notificationdata

BROWSE    SCRIPT    COLUMNS    PERMISSIONS

COLUMN NAME	TYPE	INDEX
id	string	✓ Indexed
_createdAt	date	✓ Indexed
_updatedAt	date	
_version	timestamp (MSSQL)	
_deleted	boolean	

Ми можемо створити декілька стовпців, використовуючи кнопку **Add Column** (Додати стовпець), але в цьому немає потреби, оскільки всі таблиці мають за замовчуванням динамічні схеми.



Динамічна схема доступна для Media Services з серверною JavaScript-програмою і дає змогу створювати необхідні стовпці на основі отриманих даних у форматі JSON. Тож якщо ви надсилаєте дані, які вимагають нових стовпців, вони будуть створені «на льоту», а у виробничому середовищі динамічний режим краще відключити. Але оскільки ми використовуємо JavaScript, то годі про таблиці. Звернімо краще увагу на бізнес-логіку.

## Як надсилати сповіщення із сервера?

Бізнес-логіка дуже проста. Відкриємо тригер вставлення для таблиці і змінимо вихідний код таким чином:

```
function insert(item, user, request)
{
    request.execute(
    {
        success: function()
```

```

        {
            push.mpns.sendToast(null,
            {
                text1:item.text
            },
            {
                success: function (pushResponse) {
                    console.log("Sent push:", pushResponse);
                }
            });
        });

        request.respond();
    }
);
}
}

```

У цьому коді ми зберегли метод виконання об'єкта запиту, але додали параметр. Це анонімний метод, який допомагає надсилати сповіщення. З цією метою ми використовуємо **push**-об'єкт, який включає кілька властивостей, зокрема **mpns**, **wns**, **aws**, **gcm**. Ці властивості містять посилання на об'єкти, які допомагають надсилати сповіщення на різні сервери сповіщень від Microsoft, Google і Apple. Оскільки код для Windows Phone (Silverlight) частково готовий, ми вирішили використовувати об'єкт **mpns**, але в наступних темах покажемо ще й **wns**.

Метод `sendToast` дуже простий і дає змогу надсилати повідомлення на конкретний пристрій, що їх транслює. У нашому прикладі використовується трансляція і тому значення першого параметра буде `null`. Якщо метод завершується успішно, ми додамо запис про це в журнал Mobile Services. За використання JavaScript журнал дуже важливий, оскільки дає змогу знайти помилки в коді.

## test-ms-sbaydach

LEVEL	MESSAGE	SOURCE	TIME STAMP
Information	Sent push: { isSuccessfu...	NotificationData/insert	Mon Sep 22 2014, 9:04:36 PM
Error	Error in script '/table/NotificationData.inse...	NotificationData/insert	Mon Sep 22 2014, 9:58:12 PM
Error	Error in callback for table 'NotificationData...	NotificationData/insert	Mon Sep 22 2014, 8:56:48 PM
Error	Error in script '/table/NotificationData.inse...	NotificationData/insert	Mon Sep 22 2014, 8:55:48 PM
Error	Error in callback for table 'NotificationData...	NotificationData/insert	Mon Sep 22 2014, 8:52:23 PM
Error	Error in callback for table 'NotificationData...	NotificationData/insert	Mon Sep 22 2014, 8:51:45 PM

## Програма для оператора

Тепер можна написати код, який буде надсилюти повідомлення клієнту. Звичайно, ми не можемо показувати код партнера на порталі ASP.NET. Тому ми створили просту консольну програму.

```
static void Main(string[] args)
{
    MobileServiceClient MobileService = new MobileServiceClient(
        "https://test-ms-sbaydach.azure-mobile.net/",
        "<application key>");

    IMobileServiceTable<NotificationData> messageTable =
        MobileService.GetTable<NotificationData>();

    messageTable.InsertAsync(new NotificationData()
    {
        Text = "My first notification"
    }).Wait();
}
```

Це все. Тепер можна створити складнішу програму в середовищі виконання Windows.

## Служба сповіщень Windows Notification Service і середовище виконання Windows Runtime

Час розробити реальну програму зі зрозумілим інтерфейсом і підтримкою кількох типів сповіщень, яку можна було б викласти в Магазин. У авторів немає досвіду подібних розробок для платформ Android і iOS, тож спробуємо створити програму UWP для публікації в Магазині Microsoft.

У попередній темі ми розробили програму для Windows Phone (Silverlight) без спеціального інтерфейсу, використовуючи службу Microsoft Push Notification Service (MPNS) в анонімному режимі, що підходить тільки для тестування. Сьогодні будемо використовувати службу сповіщень Windows (WNS), яка підтримується UWP.

Перед власне створенням коду слід налаштовувати облікові дані програми Windows в Mobile Services (вкладка **Push**), а для цього потрібна реєстрація в Магазині Microsoft. Якщо у вас немає облікового запису в Магазині, настав час його створити.

Якщо обліковий запис є, перейдіть на панель керування і спробуйте передати в Магазин нову програму (авжеж, ми в курсі, що її ще не створили ☺). Під час передавання ви резервуєте ім'я програми та вказуєте спосіб продажу. Наступним кроком буде створення клієнтського секретного ключа для нашої програми. Перейдемо за посиланням на сайт служби Live.

## Push notifications

### Windows Push Notification Services (WNS) and Microsoft Azure Mobile Services

The Windows Push Notification Services (WNS) enables you to send toast, tile, badge, and raw updates from your own cloud service. [Learn more](#)

If you have an existing WNS solution or need to update your current client secret, visit the [Live Services site](#)

You can also use [Microsoft Azure Mobile Services](#) to send push notifications, authenticate and manage app users, and store app data in the cloud. [Sign in](#) to your Microsoft Azure account or [sign up](#) now to add services to up to ten apps for free.

Вас буде переспрямовано на сторінку **App Settings** (налаштування програми), де можна знайти ідентифікатор пакета (**Package SID**), клієнтський ідентифікатор (**Client ID**) і клієнтський секретний ключ (**Client secret**).

### Football.ua notifications

To protect your app's security, [Windows Push Notification Services \(WNS\)](#) and [services using Microsoft account](#) use client secrets to authenticate the communications from your server.

Packge SID:  
ms-app://s-1-15-2-225 468822-3077577576-1314037951-305615383-2770 01521-20 7243510-2058990373  
[Link to different app](#)

This is the unique identifier for your Windows Store app.

Application identity:  
<Identity>  
Name=“**355815BStudio.Football.uanotifications**”  
Publisher=“CN=5E452BF3-24C3-41DD-A595-44622280C3F7”>

To set your application's identity values manually, open the AppManifest.xml file in a text editor and set these attributes of the <identity> element using the values shown here.

Client ID:  
00000000040131D26

This is a unique identifier for your application.

Client secret:  
QlRj5H wQLOM IS97MUfE twbln6j7Xw

For security purposes, don't share your client secret with anyone.

Ця інформація необхідна для надсилання сповіщень на зареєстровані пристрой. Для цього скопіюйте ці ключі і вкажіть їх на вкладці **Identity** служби Media Services.

Налаштувавши параметри Mobile Services і створивши ім'я програми в Магазині, можна завантажити всю необхідну інформацію до проекту у Visual Studio. Не

зробивши цього, ви не зможете протестувати програму, оскільки вона не отримуватиме повідомлення через неправильний ідентифікатор пакета. Для того, щоб зробити це, ви можете використовувати контекстне меню для кожного з наших проектів.

## Серверна частина програми, створеної в середовищі виконання Windows

Ми закінчили налаштування Media Services, але щоб уможливити надсилання повідомлень із сервера на зареєстровані пристрой, слід змінити тригер вставки. У попередній темі було реалізовано простий тригер на JavaScript для надсилання сповіщень через MPNS. Тепер замінимо MPNS на WNS. І додамо більше типів сповіщень, зокрема RAW-сповіщення і сповіщення плиток. RAW-сповіщення можна отримувати, коли програма активна. Це дає змогу оновлювати вміст програми в режимі реального часу. Сповіщення плитки використовується для оновлення плитки програми.

Нижче наведено код для надсилання сповіщень **Toast** (сповіщення плитки додамо згодом).

```
function insert(item, user, request)
{
    request.execute(
    {
        success: function()
        {
            push.wns.sendToastText02(null,
            {
                text1:"Football.ua",
                text2:item.text
            },
            {
                success: function (pushResponse)
                {
                    console.log("Sent push:", pushResponse);
                }
            });
        }

        request.respond();
    }
);
}
```

Використовуючи MPNS, ми викликали метод **sendToast**, однак в ОС Windows 8.1 і Windows Phone 8.1 можна налаштувати різні моделі відображення сповіщень **Toast** і сповіщень плиток. Для цієї програми будемо використовувати шаблон **ToastText02**, у якому **text1** відображається як назва сповіщення (рядок), а **text2** – як його вміст (два рядки на екрані).

## Інтерфейс нашої програми

Нарешті перейдемо створення власне клієнтської програми.

Ми почнемо з коду, який буде запитувати канал сповіщень для нашого пристроя і надсилати цей канал у службу Mobile Services. Оскільки це потрібно робити щоразу під час запуску програми, ми будемо використовувати файл **App.xaml.cs**.

Для створення каналу сповіщень використаємо класи **PushNotificationChannel** і **PushNotificationChannelManager**. У MSDN потрібно зберігати канал сповіщення після кожного запиту і порівнювати новий канал зі старим, щоб уникнути відправки дублікатів на сервер. Але ми підемо іншим шляхом. Будемо отримувати канал сповіщень для нашої програми під час кожного запуску і відразу надсилати його на Mobile Services. Звісно, можуть виникати проблеми з підключенням до Інтернету, тому ми будемо перевіряти відповідні виняткові ситуації і в разі відключення продовжимо запускати програму без сповіщень. Такий сценарій виправданий, адже ми будемо завантажувати старі дані на наступному етапі. Тож у нас буде можливість повідомити користувача про проблему мережі. У кожному разі канал сповіщень не можна вважати гарантованим способом доставки сповіщень.

Пропонуємо нижче наведений метод для отримання каналу повідомлень та реєстрації його в Mobile Services:

```
private async void CreateNotificationChannel()
{
    try
    {
        PushNotificationChannel channel;

        channel = await PushNotificationChannelManager.
CreatePushNotificationChannelForApplicationAsync();

        MobileServiceClient mobileService =
new MobileServiceClient("https://test-ms-sbaydach.
azure-mobile.net/");
    }
}
```

```

        await mobileService.GetPush() .
        RegisterNativeAsync(channel.Uri) ;
    }
    catch
    {
        ...
    }
}

```

Зверніть особливу увагу, що ми не використовуємо ключ програми на стороні клієнта. Тож необхідно відповідним чином змінити дозвіл для сценарію реєстрації.

## registration endpoints

PERMISSION

Everyone



Тепер виклик методу можна розмістити на початку обробника події **OnLaunched**, щоб отримувати toast-сповіщення відразу.

Час реалізувати бізнес-логіку нашої програми. Оскільки класи даних не пов'язані з конкретним інтерфейсом, будемо використовувати загальний проект, щоб створити всі необхідні класи. Створимо папку **Code** з файлом **DataClasses.cs**. Там будуть два класи.

Один із цих класів описуватиме нашу таблицю даних в Azure Mobile Services. Слід використовувати імена таблиць і стовпців, щоб не використовувати атрибути та інші типи зв'язування. Наш клас виглядатиме так:

```

public class NotificationData
{
    public string id { get; set; }
    public string text { get; set; }
    public DateTime __createdAt { get; set; }
}

```

Другий клас – допоміжний. Він використовуватиметься для завантаження даних з Azure Mobile. Реалізуємо такий код:

```

public class DataClass
{
}

```

```

public static async Task<ICollection<NotificationData>>
    LoadData()
{
    MobileServiceClient client =
        new MobileServiceClient("https://test-ms-sbaidachni.
            azure-mobile.net/");
    var table=client.GetTable<NotificationData>();
    var query = (from item in table
                  orderby item._createdAt descending
                  select item).Take(100);
    var items = await query.ToCollectionAsync();

    return items;
}
}

```

Ми надали загальний дозвіл на отримання даних з Azure Mobile, тому ключі не використовуємо. Але ми завантажимо всього 100 записів із нашої бази даних. Згодом можна додати ще один метод, який здійснюватиме інкрементне завантаження. Ви можете використовувати метод **Skip(n)**, щоб пропустити вже завантажені записи: **.Skip(n).Take(100)**.

Ми реалізували всю бізнес-логіку і налаштували програму для отримання сповіщень. Роботу з інтерфейсом можна вважати завершеною.

Скористаймося елементом керування **GridView**, щоб показати всі сповіщення. Слід відключити всі функції вибору й активування клацанням та відобразити сповіщення й дати. Розглянемо нижченаведений код.

```

<Page.BottomAppBar>
    <CommandBar>
        <AppBarButton Label="Refresh" Icon="Refresh"
            Click="AppBarButton_Click"></AppBarButton>
    </CommandBar>
</Page.BottomAppBar>

<Grid Background="{ThemeResource
    ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>

```

```
<Image Source="Assets/football.png" Height="36"
   HorizontalAlignment="Left" Margin="120,30,0,10">
</Image>
<GridView Name="myGrid" Grid.Row="1"
   Padding="120,40,100,80" SelectionMode="None"
   Visibility="Collapsed">
    <GridView.ItemTemplate>
        <DataTemplate>
            <Grid Width="450"
               HorizontalAlignment="Left"
               Height="100" Background="Gray" >
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto">
                    </RowDefinition>
                    <RowDefinition Height="*">
                    </RowDefinition>
                </Grid.RowDefinitions>
                <TextBlock Text="{Binding __createdAt}"
                   Style="{StaticResource
                   CaptionTextBlockStyle}"
                   Foreground="Green" Margin="5">
                </TextBlock>
                <TextBlock Text="{Binding text}"
                   Grid.Row="1" Style="{StaticResource
                   BodyTextBlockStyle}"
                   TextWrapping="Wrap" Margin="5">
                </TextBlock>
            </Grid>
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
<ProgressRing Name="progressBox" IsActive="True"
   HorizontalAlignment="Center" VerticalAlignment="Center"
   Width="100" Height="100" Grid.Row="1"
   Visibility="Visible"></ProgressRing>
<TextBlock Name="errorBox" TextWrapping="Wrap"
   Text="Что-то случилось с сетью. Попробуйте загрузить
   данные снова." Visibility="Collapsed"
   HorizontalAlignment="Center"
   VerticalAlignment="Center"
   TextAlignment="Center"
   Grid.Row="1" Style="{StaticResource
```

```
HeaderTextBlockStyle}"></TextBlock>

<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="Common">
        <VisualState x:Name="Loading">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames
                    Storyboard.TargetName="errorBox"
                    Storyboard.TargetProperty=
                    "Visibility">
                    <DiscreteObjectKeyFrame
                        Value="Collapsed" KeyTime="0">
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
                    Storyboard.TargetName="progressBox"
                    Storyboard.TargetProperty=
                    "Visibility">
                    <DiscreteObjectKeyFrame
                        Value="Visible" KeyTime="0">
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
                <ObjectAnimationUsingKeyFrames
                    Storyboard.TargetName="myGrid"
                    Storyboard.TargetProperty=
                    "Visibility">
                    <DiscreteObjectKeyFrame
                        Value="Collapsed" KeyTime="0">
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Loaded">
            <Storyboard>
                <ObjectAnimationUsingKeyFrames
                    Storyboard.TargetName="errorBox"
                    Storyboard.TargetProperty=
                    "Visibility">
                    <DiscreteObjectKeyFrame
                        Value="Collapsed" KeyTime="0">
                    </DiscreteObjectKeyFrame>
                </ObjectAnimationUsingKeyFrames>
```

```
<ObjectAnimationUsingKeyFrames  
Storyboard.TargetName="progressBox"  
Storyboard.TargetProperty=  
"Visibility">  
    <DiscreteObjectKeyFrame  
    Value="Collapsed" KeyTime="0">  
    </DiscreteObjectKeyFrame>  
</ObjectAnimationUsingKeyFrames>  
<ObjectAnimationUsingKeyFrames  
Storyboard.TargetName="myGrid"  
Storyboard.TargetProperty=  
"Visibility">  
    <DiscreteObjectKeyFrame  
    Value="Visible" KeyTime="0">  
    </DiscreteObjectKeyFrame>  
</ObjectAnimationUsingKeyFrames>  
    </Storyboard>  
</VisualState>  
<VisualState x:Name="Error">  
    <Storyboard>  
        <ObjectAnimationUsingKeyFrames  
        Storyboard.TargetName="errorBox"  
        Storyboard.TargetProperty=  
"Visibility">  
            <DiscreteObjectKeyFrame  
            Value="Visible" KeyTime="0">  
            </DiscreteObjectKeyFrame>  
</ObjectAnimationUsingKeyFrames>  
<ObjectAnimationUsingKeyFrames  
Storyboard.TargetName="progressBox"  
Storyboard.TargetProperty=  
"Visibility">  
    <DiscreteObjectKeyFrame  
    Value="Collapsed" KeyTime="0">  
    </DiscreteObjectKeyFrame>  
</ObjectAnimationUsingKeyFrames>  
<ObjectAnimationUsingKeyFrames  
Storyboard.TargetName="myGrid"  
Storyboard.TargetProperty=  
"Visibility">  
    <DiscreteObjectKeyFrame  
    Value="Collapsed" KeyTime="0">  
    </DiscreteObjectKeyFrame>
```

```
        </DiscreteObjectKeyFrame>
    </ObjectAnimationUsingKeyFrames>
</Storyboard>
</VisualState>
</VisualStateGroup>

</VisualStateManager.VisualStateGroups>
</Grid>
```

Так, код трохи завеликий, зате дає змогу продемонструвати різні частини нашого інтерфейсу, для яких використовуються різні події в програмі. Цей код можна умовно поділити на три частини.

У першій частині ми оголосили елемент керування меню з кнопкою оновлення. Ця кнопка використовуватиметься для оновлення інтерфейсу, якщо користувач відкрив програму певний час тому й вона досі перебуває в пам'яті.

У другій частині оголошено розмітку **Grid** і деякі її елементи керування, такі як **GridView**, **Image** й кілька текстових полів із різними повідомленнями. Залежно від ситуації має відображатися **GridView** або повідомлення.

В останній частині ми оголосили елемент керування **VisualStateManager**, який, своєю чергою, дає змогу оголосити кілька користувачьких станів. У нас буде три стани:

- **Loading** – дані завантажуються зі служб Azure Mobile;
- **Loaded** – дані завантажено, і їх можна демонструвати;
- **Error** – виникла проблема з Інтернетом.

Залежно від стану деякі частини інтерфейсу буде показано, а деякі – приховано.

Нарешті, потрібно реалізувати код, який встановлюватиме стан нашого інтерфейсу. Цей код буде застосовувати допоміжний клас, щоб завантажити дані. Пропонуємо використовувати в **MainPage.xaml.cs** такий код:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    LoadData();
}

private void AppBarButton_Click(object sender,
    RoutedEventArgs e)
```

```
{
    LoadData();
}

private async void LoadData()
{
    try
    {
        VisualStateManager.GoToState(this, "Loading", false);
        var items = await DataClass.LoadData();
        myGrid.ItemsSource = items;
        VisualStateManager.GoToState(this, "Loaded", false);
    }
    catch (Exception ex)
    {
        VisualStateManager.GoToState(this, "Error", false);
    }
}
```

Напевно, тут краще використовувати тригери й сеттери, але цей код було реалізовано до випуску Windows 10. Але ви можете змінити його за власним бажанням.

Найважливішим методом є **LoadData**. Він використовується, щоб переводити інтерфейс у належний стан. Це дає змогу завантажувати та зв'язувати дані.

Таким чином, ми розробили програму, що дасть нам змогу отримувати повідомлення про футбольні матчі, для оголошень про які використовується служба Windows Notification Service. Крім того, у програмі відображаються останні повідомлення.

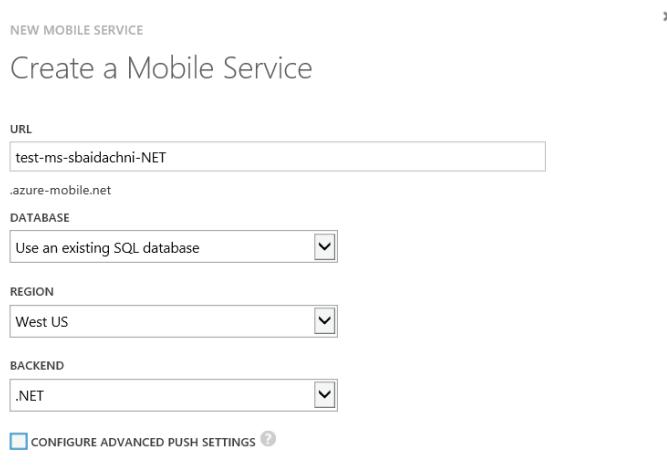
## **FOOTBALL.UA**

28.12.2014 8:54:34 Футбол. Матч Ман Сіти - Бернлі 2:2 завершено	28.12.2014 7:24:10 Футбол. Гол! Давід Сильва, 23; Ман Сіти - Бернлі 1:0	28.12.2014 3:53:22 Футбол. Матч Тоттенхом - Манчестер Юнайтед 0:0
28.12.2014 8:41:43 Футбол. Гол! Ешли Барнс, 82; Ман Сіти - Бернлі 2:2	28.12.2014 6:52:32 Футбол. Гол! Зден Азар, 45+1; Саутгемптон - Челсі 1:1	28.12.2014 3:46:21 Футбол. Матч Тоттенхом - Манчестер Юнайтед 0:0
28.12.2014 8:07:40 Футбол. Гол! Джордж Бойд, 47; Ман Сіти - Бернлі 2:1	28.12.2014 6:42:08 Футбол. Матч Ман Сіти - Бернлі починається	28.12.2014 3:00:35 Футбол. Трансляція сьогодні: АПЛ, 14:00 Тоттенхом 2:0, 16:00 Саутгемптон - Челсі (Футбол 2), 17:00 Манчестер Юнайтед
28.12.2014 7:58:57 Футбол. Матч Саутгемптон - Челсі 1:1 завершено	28.12.2014 6:23:24 Футбол. Гол! Садіо Мане, 17; Саутгемптон - Челсі 1:0	26.12.2014 8:54:01 Футбол. Матч Бромемч - Манчестер Сіти 1:1
28.12.2014 7:34:10 Футбол. Гол! Фернандіно, 34; Ман Сіти - Бернлі 2:0	28.12.2014 5:53:45 Футбол. Матч Саутгемптон - Челсі (Футбол 2) починається	26.12.2014 8:53:41 Футбол. Матч Манчестер Юнайтед - Ньюкасл 3:1

## Серверний код .NET

Для обробки операцій вставлення ми використовували серверний код JavaScript. Це давало змогу надсилати повідомлення всім абонентам. У прикладі використовувався простий код, і застосування JavaScript було виправданим, незважаючи на те, що ми віддаємо перевагу .NET. Але в багатьох проектах код може бути більш складним, і для розробки, можливо, знадобиться використовувати Visual Studio і C#. Тому зупинимося на застосуванні Visual Studio для написання серверного коду.

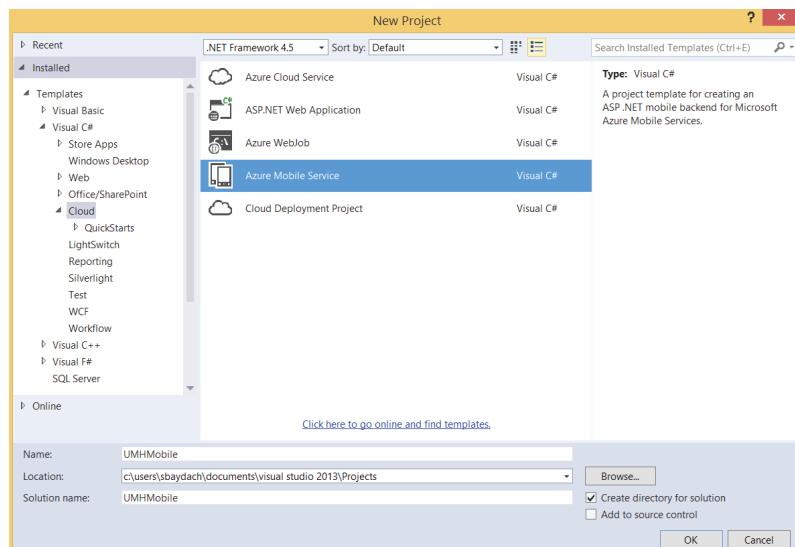
Насамперед, щоб використовувати для серверного коду C#, у діалоговому вікні **Create a Mobile Service** необхідно вибрати як технологію .NET.



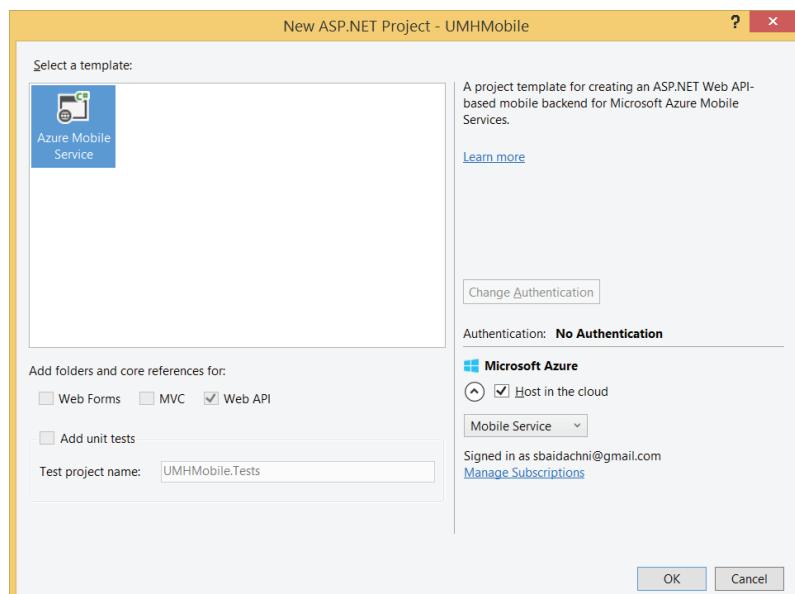
Для створення Mobile Service можна використати і Visual Studio, але в силу традиції ми надаємо перевагу панелі керування.

Отже, ми готові використовувати Visual Studio. На першому етапі створимо проект на основі шаблона **Azure Mobile Service**. Спершу потрібно вказати ім'я проекту.

## Розділ 28. Надсилання повідомень за допомогою служб Microsoft Azure



На другому етапі потрібно вибрати кілька параметрів. Необхідно зняти пра-рець **Host in the cloud**. Якщо не зробити цього, Visual Studio запропонує ство-рити нову службу Mobile Service.



Щойно ви натиснете **OK**, Visual Studio створить проект із кількома готовими класами: **TodoItem**, **TodoItemController** тощо. Можна використовувати цей код, щоб розгорнути таблицю **TodoItem** в Azure Mobile Service.

Ми не будемо видаляти автоматично створений код, а просто задамо інші імена для таблиці та стовпців. Змінимо **TodoItem** на **NotificationData**. Для змінення імені можна використовувати інструменти рефакторингу. Крім того, потрібно видалити властивість **Complete** і у властивостях змінити **Text** на **text**. Решту змін ви можете внести за допомогою атрибута **JsonProperty**.

```
public class NotificationData : EntityData
{
    [JsonProperty("text")]
    public string Text { get; set; }
}
```

Оскільки EntityFramework і Web API вже мають необхідну для нашої служби інфраструктуру, зосередимо увагу на таких питаннях:

- надсилання широкомовних повідомлень з коду;
- встановлення прав доступу для операцій із таблицями;
- створення власного коду, який оброблятиме реєстрацію нових пристройів.

Найпростіше завдання – це надсилання повідомлень. Потрібно відкрити файл **TodoItemController.cs** і змінити метод **PostTodoItem** таким чином:

```
public async Task<IHttpActionResult>
PostTodoItem(NotificationData item)
{
    NotificationData current = await
InsertAsync(item);

    WindowsPushMessage message =
        new WindowsPushMessage();

    message.XmlPayload = @"<?xml version=""1.0"""
encoding=""utf-8""?>" +
        @"<toast><visual><binding
template=""ToastText02"">" +
        @"<text id=""1"">football.ua
</text>" +
        @"<text id=""2"">" + item.Text
        + @"</text>" +
```

```

        @"</binding></visual></toast>";
    try
    {
        var result = await
            Services.Push.SendAsync(message);
        Services.Log.Info(result.State.ToString());
    }
    catch (System.Exception ex)
    {
        Services.Log.Error(ex.Message, null,
            "Push.SendAsync Error");
    }

    return CreatedAtRoute("Tables", new
    {
        id = current.Id }, current);
}

```

Як бачимо, тут використано два класи. **WindowsPushMessages** дає змогу створити повідомлення про відображення спливаючого сповіщення, а **ApiServices** (екземпляр служби) – надсилати повідомлення й додавати інформацію в журнал. Як і з JavaScript, ми використовували шаблон **ToastText02**, який містить назву й повідомлення. Отже, маємо той самий код, але на C#.

Тепер розгляньмо дозволи. Використовуючи JavaScript, ми встановлювали дозволи за допомогою панелі керування Azure, а для C# ми маємо застосовувати атрибути. Щоб встановити права доступу до методів класу **Controller**, необхідно використовувати атрибут **AuthorizeLevel**.

```

[AuthorizeLevel(AuthorizationLevel.Application)]
public async Task<IHttpActionResult>
    PostTodoItem(NotificationData item)
[AuthorizeLevel(AuthorizationLevel.Anonymous)]
public IQueryable<NotificationData> GetAllTodoItems()

```

Звичайно, цей підхід працюватиме тільки для табличних операцій, але з JavaScript він дає змогу встановити рівень авторизації за допомогою панелі керування Azure. Використовуючи C#, його можна встановити у файлі **WebApiConfig.cs**. Щоб зареєструвати метод, просто додайте до коду такий рядок:

```
options.PushAuthorization = AuthorizationLevel.Anonymous;
```

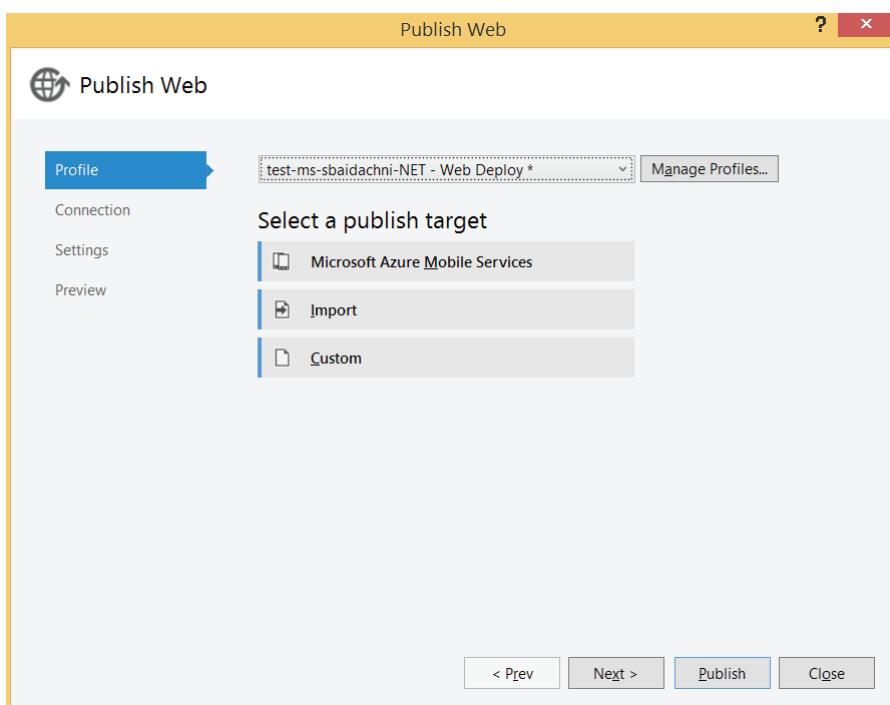
У прикладі ми не змінювали послідовність реєстрації, але за потреби це можна зробити, реалізувавши інтерфейс **NotificationHandler**.

```
public class PushRegistrationHandler : INotificationHandler
{
    public System.Threading.Tasks.Task Register(
        ApiServices services, HttpContext context,
        NotificationRegistration registration)
    {
        throw new NotImplementedException();
    }

    public System.Threading.Tasks.Task Unregister(
        ApiServices services, HttpContext context,
        string deviceId)
    {
        throw new NotImplementedException();
    }
}
```

Реалізація інтерфейсу вибиратиметься автоматично, і ви можете запустити власний код.

Нарешті, щоб перевірити рішення, його потрібно розгорнути. Для цього просто виберіть в контекстному меню пункт **Publish** (Опублікувати) й налаштуйте всі необхідні параметри:



Щоб спростити процес, можна завантажити профіль публікації з панелі керування у вже створеній службі.

mobile service endpoint status [PREVIEW](#)

You have not configured mobile service endpoint monitoring.

CONFIGURE MOBILE SERVICE ENDPOINT MONITORING [\(?\)](#)

quick glance

[View Applicable Applications and services](#)

[Download publish profile](#)

## Інші можливості Mobile Services

Ми описали дуже просту програму, але розглянули лише половину всіх функцій Azure Mobile Services. Тому, щоб мати повну картину щодо даної теми, опишемо решту функцій.

## Теги

Ця функція пов'язана з Push Notification Hub, але в Azure Mobile Services її також реалізовано. Ідея дуже проста – використовувати спеціальні теги з метою виявлення конкретних груп користувачів. Наприклад, якщо ви розробляєте сайт новин, то можете створити тег дляожної групи новин. Це дасть користувачу змогу вибрати тільки ті групи, які йому цікаві. Хочь не любить політику, а комусь іншому, можливо, не цікаві спортивні новини і т.д.

Звичайно, для цього потрібен спосіб вибору тегів у процесі реєстрації користувача. Крім того, необхідна можливість транслювати повідомлення тільки для обраних тегів. Гарна новина – інтерфейс API Mobile Service підтримує теги. Наприклад, для реєстрації із застосуванням тегів можна використовувати такий код:

```
await mobileService.GetPush().RegisterNativeAsync(channel.Uri,  
    new List<string>() {"sport", "movies", "Canada"});
```

У цьому прикладі з реєстрацією користувачів буде пов'язано три теги.

Щоб надіслати всім користувачам повідомлення з тегом «кіно», можна використовувати такий код (C#):

```
await Services.Push.SendAsync(message, "movies");
```

Зауважте, що спеціальні методи надсилання повідомлень користувачам, як правило, дають змогу передавати кілька тегів. Вираз із тегами можна використовувати для передавання виразів на основі тегів, застосовуючи логічні оператори на кшталт **&&**, **||** та **!**: **«(tagA && !tagB)»**

## Шаблони

Шаблони – це ще один спосіб створення персоналізованих повідомлень для користувачів. Наприклад, щоб надіслати повідомлення із C# для користувачів Windows, можна використовувати такий код:

```
WindowsPushMessage message = new WindowsPushMessage();  
  
message.XmlPayload = @"<?xml version=""1.0""  
encoding=""utf-8""?>" +  
    @"<toast><visual><binding  
    template=""ToastText02"">" +  
    @"<text id=""1"">football.ua</text>" +  
    @"<text id=""2"">" + item.Text + @"</text>" +
```

```
@"</binding></visual></toast>";
```

Якщо сформувати повідомлення таким чином, усі користувачі бачитимуть те саме повідомлення. Але інколи вам може знадобитися передати параметри повідомлення на основі користувацьких уподобань. Наприклад, якщо ваша програма підтримує кілька мов, можна надсилати локалізовані повідомлення. Якщо програма містить дані на кшталт відстані або температури, ви можете використовувати систему вимірювання, зрозумілу для користувачів, тощо.

Шаблони дають змогу використовувати параметри в XML або JSON (залежно від платформи), які зазвичай створюються на стороні сервера. У разі використання шаблонів на стороні сервера повідомлення не повинні створюватися «з нуля». Натомість клієнтська програма має зареєструвати шаблон повідомлення, зокрема параметри та ім'я шаблона. Для вибраних шаблонів сервер надсилає тільки параметр.

Наприклад, якщо ви створюєте програму для Windows, щоб зареєструвати шаблон, клієнт може використовувати такий код:

```
string xmlTemplate = @"<?xml version=""1.0"" encoding=""utf-8""?>" +
    @"<toast><visual><binding template=""ToastText02"">" +
    @"<text id=""1"">football.ua</text>" +
    @"<text id=""2"">$({ru_message})</text>" +
    @"</binding></visual></toast>";

await mobileService.GetPush().RegisterTemplateAsync(
    channel.Uri, xmlTemplate, "langTemplate");
```

У такому разі клієнтська програма реєструє шаблон із довгою назвою і з параметром **ru\_message**. Другий пристрій замість **ru\_message** може використовувати параметр **en\_message** і т. д. Це необхідно для того, щоб користувач пристрою міг отримати повідомлення рідною мовою. Таким чином, усі пристрої матимуть той самий шаблон, але з різними параметрами.

На наступному етапі на стороні сервера потрібно сформувати повідомлення, яке міститиме назву шаблона і всі параметри. Notification Hub дає змогу створювати унікальні повідомлення для всіх пристрій на базі зареєстрованих шаблонів і параметрів. Щоб підготувати повідомлення на стороні сервера, можна застосовувати вже готові класи. Наприклад, якщо ви використовуєте C#, то можете реалізувати такий код:

```
TemplatePushMessage mess = new TemplatePushMessage();  
  
mess.Add("ru_message", "Сообщение");  
mess.Add("en_message", "Message");  
  
await Services.Push.SendAsync(mess);
```

Notification Hub переглядатиме кожен зареєстрований шаблон і в разі виявлення одного з цих параметрів надішле відповідне повідомлення.

## Jobs

Деякі програми, такі як Facebook для Windows Phone, надсилають багато повідомлень протягом заданого періоду часу. Наприклад, як правило, у 23:00 ми отримуємо повідомлення про дні народження друзів. Звичайно, надсилення повідомлень за розкладом – непогана ідея. І служби Azure Mobile Services підтримують цю функцію

З Azure Mobile Services ви зможете запланувати виконання будь-якого коду. Можна налаштувати одну подію або створити складний графік, використовувати панель керування Azure для створення коду JavaScript або Visual Studio для створення коду C#.

Використовуючи JavaScript, можна легко створювати елементи за допомогою вкладки **Scheduler**. Зауважте, що передбачено спосіб створення завдання, яке працюватиме залежно від запитів:

The screenshot shows the Azure portal interface for managing a scheduled job. At the top, there's a dark header bar with a progress bar. Below it, the job name 'sss' is displayed. Underneath the name are two tabs: 'CONFIGURE' and 'SCRIPT'. A horizontal line separates this from the main content area. In the content area, there's a red 'disabled' status bar with a warning icon and the text 'This job will not run. To enable it, click the Enable button.' Below this, the word 'status' is underlined. Another horizontal line follows. Under 'LAST RUN', it says 'Never'. Another horizontal line follows. Under 'NEXT RUN', it also says 'Never'. Another horizontal line follows. Below these, the word 'schedule' is underlined. Under 'schedule', there are two options: 'Every 15 minutes' (with a radio button selected) and 'On demand' (with an unselected radio button). The entire screenshot is framed by a blue vertical bar on the left.

Код JavaScript може мати доступ до всіх об'єктів, які ми застосовували раніше, зокрема до об'єкта **push**.

Використовуючи C#, потрібно працювати з Visual Studio, як ми робили в попередньому розділі, і створити клас, який успадковує клас **ScheduledJob**. Метод **ExecuteAsync** дасть змогу створити всередині будь-який код.

## Користувацький інтерфейс API

Ще одна цікава річ – в Azure Mobile Services є можливість створення користувацького API. Можна створити набір функцій JavaScript на стороні сервера, використовуючи панель керування:



```
hhh

SCRIPT PERMISSIONS

1 exports.post = function(request, response) {
2     // Use "request.service" to access features of your mobile service, e.g.:
3     //   var tables = request.service.tables;
4     //   var push = request.service.push;
5
6     response.send(statusCodes.OK, { message : 'Hello World!' });
7 };
8
9 exports.get = function(request, response) {
10    response.send(statusCodes.OK, { message : 'Hello World!' });
11};
```

У разі використання Visual Studio потрібно створити клас, що успадковує **ApiController**.

Оскільки код викликатиметься за допомогою стандартного протоколу HTTP, можна застосувати один із таких методів, як **GET**, **POST**, **PUT**, **PATCH**, **DELETE**.

Розділ 29.

## **Як створити власні елементи керування**

Universal Windows Platforms пропонує безліч різних елементів керування, але за бажання ви можете створити свій власний – щодо цього немає жодних обмежень. Є три способи створення елементів керування:

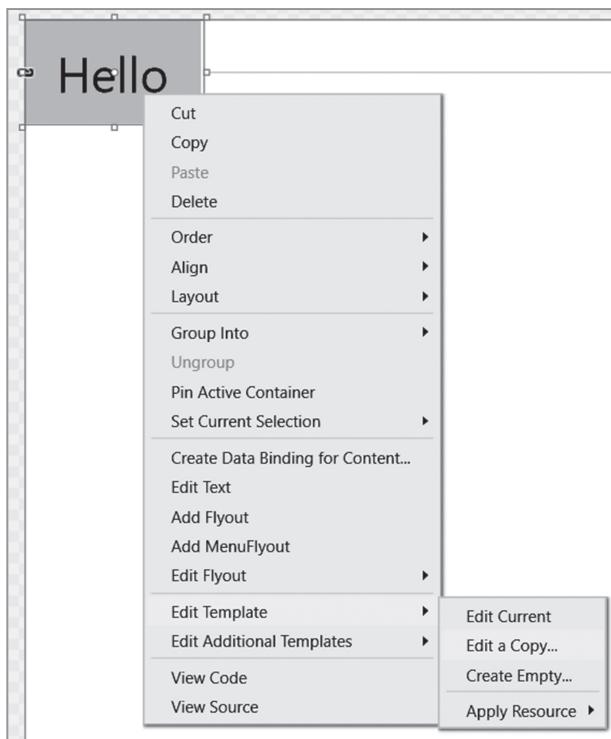
- Можна створити новий елемент керування на основі наявного, дещо змінивши шаблон і поведінку. У шаблоні можна перебудовувати елемент керування й реалізувати власний диспетчер візуального стану.
- Ви можете об'єднати кілька елементів керування, додати певні специфічні методи й поведінку, щоб спростити роботу з однією групою елементів керування для багатьох сторінок. Наприклад, для входу в систему можна створити елемент керування, який міститиме текстові поля, кнопки й текстові блоки.
- Можна створити елемент керування «з нуля».

Цей розділ присвячено створенню кожного з описаних вище типів елементів керування.

## Шаблони

Як зазначено вище, майже всі елементи керування підтримують властивість **Template**. Ця властивість вводиться в класі **Control** і дає змогу зберігати подання елемента керування в XAML. Звичайно, властивість **Template** є загальнодоступною. Будь-якому з елементів керування можна призначити нове подання, використовуючи стилі. Найкращий спосіб – це змінити все необхідне безпосередньо в XAML.

Якщо потрібно змінити зовнішній вигляд елемента керування, варто почати з копії наявного шаблона. Її можна легко створити за допомогою Visual Studio 2015. Просто розмістіть екземпляр елемента керування на сторінці та скористайтеся контекстним меню, щоб створити копію шаблона:



Щойно ви виберете розташування, Visual Studio включить шаблон XAML до вашого проекту. Відтак можна починати його редагування.

Зверніть особливу увагу, що в шаблоні не тільки присвоюються значенням властивості **Template**, але й міститься багато методів сеттерів, що присвоюють елементу керування стандартні властивості. Але властивість **Template** є найцікавішою, оскільки вона містить подання власне елемента керування й дає змогу змінювати не тільки основне подання, але й візуальні стани. Розгляньмо скорочений варіант шаблона для кнопки:

```
<ControlTemplate TargetType="Button">
    <Grid x:Name="RootGrid"
        Background="{TemplateBinding Background}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal">
                    <Storyboard>
                        . . .
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
    </Grid>
</ControlTemplate>
```

```
</VisualState>
<VisualState x:Name="PointerOver">
    <Storyboard>
        . . . .
    </Storyboard>
</VisualState>
<VisualState x:Name="Pressed">
    <Storyboard>
        . . . .
    </Storyboard>
</VisualState>
<VisualState x:Name="Disabled">
    <Storyboard>
        . . . .
    </Storyboard>
</VisualState>
</VisualStateManager.VisualStateGroups>
<ContentPresenter x:Name="ContentPresenter"
    BorderBrush="{TemplateBinding BorderBrush}" . . . . />
</Grid>
</ControlTemplate>
```

Тут видалено всі анімації та деякі властивості, щоб код мав більш наочний вигляд. Як бачимо, у поданні XAML елемент керування «кнопка» – це просто сітка, на якій можна розміщувати контент. Кнопки, звичайно, підтримують багато станів, які можна знайти в шаблоні елемента керування. Але якщо вам потрібно створити власну кнопку, можна просто змінити наявний шаблон або створити новий «із нуля».

За певних умов подання потребуватимуть додаткових властивостей. Розгляньмо, наприклад, шаблон **AppBarButton**:

```
<ControlTemplate TargetType="AppBarButton">
    <Grid x:Name="Root" Background="{TemplateBinding Background}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="ApplicationViewStates">
                <VisualState x:Name="FullSize"/>
                <VisualState x:Name="Compact">
                    <Storyboard>
                        </Storyboard>
                </VisualState>
```

```

<VisualStateManager>
    <VisualStateGroup x:Name="CommonStates">
        <VisualState x:Name="Normal">
            <Storyboard>
                ...
            </Storyboard>
        </VisualState>
        <VisualState x:Name="PointerOver">
            <Storyboard>
                ...
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Pressed">
            <Storyboard>
                ...
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Disabled">
            <Storyboard>
                ...
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager>
<StackPanel x:Name="ContentRoot" >
    <ContentPresenter x:Name="Content" />
    <TextBlock x:Name="TextLabel" />
</StackPanel>
<TextBlock x:Name="OverflowTextLabel" />
</Grid>
</ControlTemplate>

```

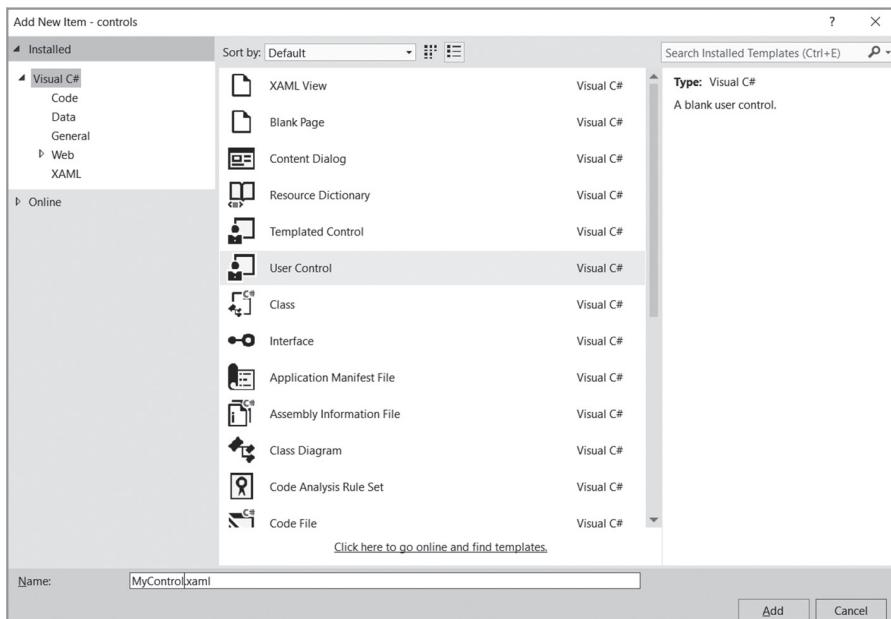
Як бачимо із шаблона, у Microsoft повністю переробили кнопку. Але для нової кнопки необхідні нові властивості (принаймні значок), що відповідають новому шаблону. Ось чому спеціалісти Microsoft реалізували новий клас **AppBarButton**, що розширює клас **Button** і містить додаткову логіку.

Зауважте, що шаблони використовують спеціальне розширення розмітки під назвою **TemplateBinding**. Це так звана прив'язка, що дає змогу посилатися на властивості елемента керування. З її допомогою розробники можуть використовувати стилі або динамічно змінювати властивості елемента керування, щоб вплинути на його зовнішній вигляд.

## Клас UserControl

Другою категорією елементів керування є ті, які можна створити на основі класу **UserControl**. Цей клас успадковується від **Control** і має тільки одне додовнення – властивість **Content**. Завдяки цьому можна використовувати **UserControl** як контейнер із такими елементами, як **Grid**, **StackPanel** тощо. Ідея дуже проста: якщо потрібно об'єднати кілька елементів керування й використовувати їх у вашій програмі, просто розмістіть їх усі в **UserControl**, скориставшись будь-яким зі звичайних контейнерів. Потім потрібно розмістити користувацький елемент керування в окремому файлі й додати бізнес-логіку у відповідний файл коду. Visual Studio надає всі необхідні шаблони, підтримує режим розробки, засоби кодування та інші засоби для користувацьких елементів керування.

Щоб почати розробку складного елемента керування, необхідно додати новий елемент на основі шаблона **User Control**:

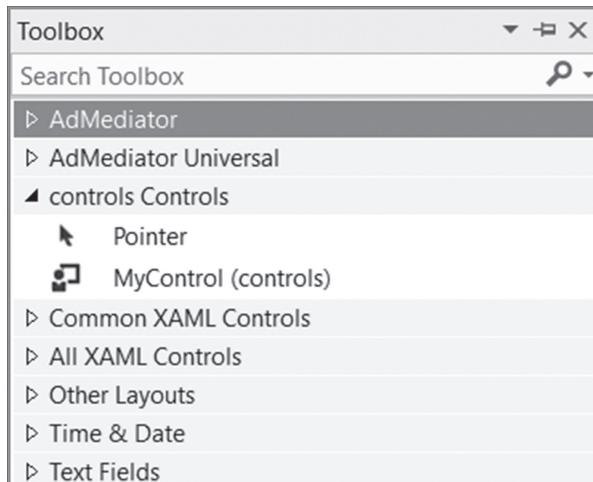


Щойно ви натиснете кнопку **Add**, Visual Studio створить файл XAML і відповідний файл коду:

- ◀ MyControl.xaml
- ▶ MyControl.xaml.cs

Відтак можна працювати з елементом керування як зі сторінкою. Ви можете почати з контейнера, розмістивши в ньому будь-які елементи керування та реалізувавши обробники подій. Є тільки одна відмінність – як правило, у сторінки немає жодних загальнодоступних API. Але зауважте, що складені елементи керування розміщуватимуться на сторінці й потрібно надати базовий набір методів і властивостей, щоб настроїти властивості та поведінку елемента керування, а також за потреби обробляти певні події.

Щойно елемент керування буде готовий, можна його використовувати. Перебу́йте свою програму, і ви знайдете елемент керування на панелі інструментів:

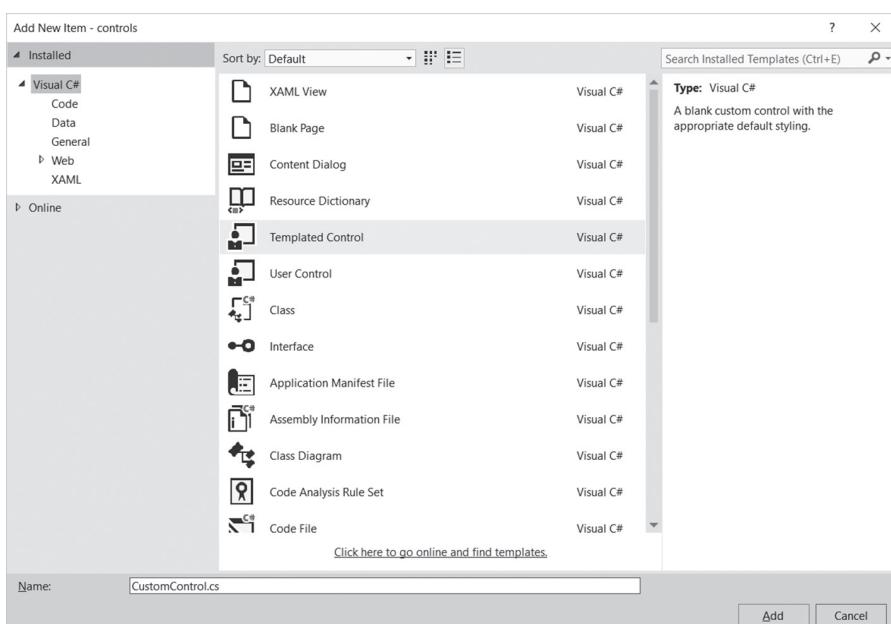


Просто перетягніть його на сторінку, і Visual Studio створить відповідний простір імен.

## Шаблон **Templated control**

Ви можете створити елемент керування «з нуля». Почати краще із шаблона **Templated Control**:

## Windows 10 для C# розробників



Як бачимо, Visual Studio створює клас із кількома рядками коду:

```
public sealed class CustomControl : Control
{
    public CustomControl()
    {
        this.DefaultStyleKey = typeof(CustomControl);
    }
}
```

Але для початку їх достатньо, оскільки цей клас успадковується від **Control** і має властивість **Template**. Це дає змогу сформувати певне подання елемента керування. Завдяки **DefaultStyleKey** ми можемо задати для елемента керування неявно визначений стиль. У Solution Explorer ви знайдете ще один файл – **generic.xaml**. Це простий файл ресурсів, який можна використовувати як відправну точку для створення користувачького інтерфейсу елемента керування:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:controls">
```

```

<Style TargetType="local:CustomControl" >
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="local:
                CustomControl">
                <Border
                    Background="{TemplateBinding
                        Background}"
                    BorderBrush="{TemplateBinding
                        BorderBrush}"
                    BorderThickness="{TemplateBinding
                        BorderThickness}">
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</ResourceDictionary>

```

За замовчуванням цей файл містить тільки елемент **Border**, але навіть за таких умов можна розмістити цей елемент керування на сторінці, присвоїти значення його основним властивостям й переглянути його:

```
<local:CustomControl Height="100" BorderThickness="3"
BorderBrush="Red"/>
```

На подальших етапах потрібно виконати описані нижче дії.

- Створити для елемента керування користувацький інтерфейс у XAML, зокрема стани.
- Реалізувати логіку елемента керування.
- Реалізувати загальнодоступні методи й події.
- Реалізувати властивості.

Щодо властивостей, то краще скористатися властивостями залежностей. Цей підхід дає змогу працювати з властивостями в XAML. При цьому всі вони підтримуватимуть усі відповідні функції. Нижче наведено приклад реалізації властивості залежностей:

```
public string Label
{
    get { return GetValue(LabelProperty).ToString(); }
    set { SetValue(LabelProperty, value); }
}

public static readonly DependencyProperty LabelProperty =
    DependencyProperty.Register(
        "Label",
        typeof(string),
        typeof(CustomControl),
        new PropertyMetadata(
            false,
            new PropertyChangedCallback(OnLabelChanged)
        )
);
```

Звичайно, це не все, що потрібно знати про елементи керування. Наприклад, ми не розглядали атрибути, редактори властивостей і багато іншого. Але цих відомостей має бути достатньо, щоб почати розробку першого користувачького елемента керування.

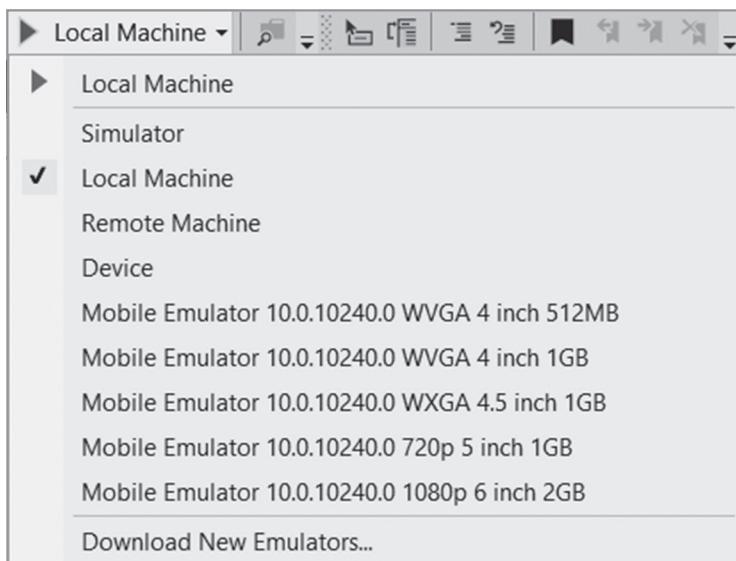
Розділ 30.

## **Тестування і налагодження**

У цьому розділі ми зібрали всю інформацію про Visual Studio 2015 та засоби розробки програм для Windows 10, про які не йшлося раніше. Це емулятори, засоби налагодження та деякі нові корисні можливості XAML.

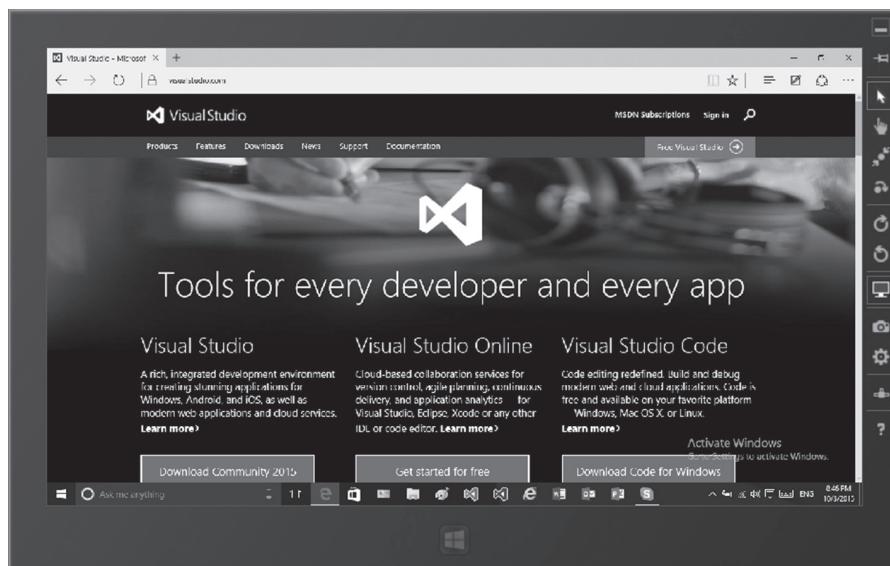
## Емулятори та симулятори

Спершу мова піде про емулятори й симулятори. Це важливі засоби, оскільки деякі розробники звикли працювати на комп'ютері без сенсорного монітора і багатьох інших складових, наявних у сучасних планшетах і телефонах. Водночас інтерфейс розробок зручніше тестувати, якщо є змога змінити розмір екрана, його орієнтацію та роздільну здатність. Звичайно, можна купити кілька різних пристрій і використовувати їх для розгортання та тестування програм, але це задорого, а до того ж незручно користуватися декількома пристроями водночас. Тож найкраще використовувати стандартну панель інструментів, зображену нижче.



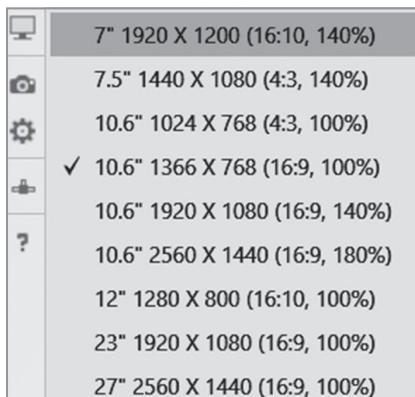
За замовчуванням Visual Studio запускає програми на локальному комп'ютері, але їх можна перемикати на симулятор, потрібний пристрій, віддалений комп'ютер або мобільний емулятор.

Користуючись симулятором, ви, як і раніше, інсталюватимете й запускатимете програму на локальному комп'ютері, однак Visual Studio запустить інструмент, що відкриє можливість взаємодіяти з вашим комп'ютером за допомогою багатьох різних інструментів:

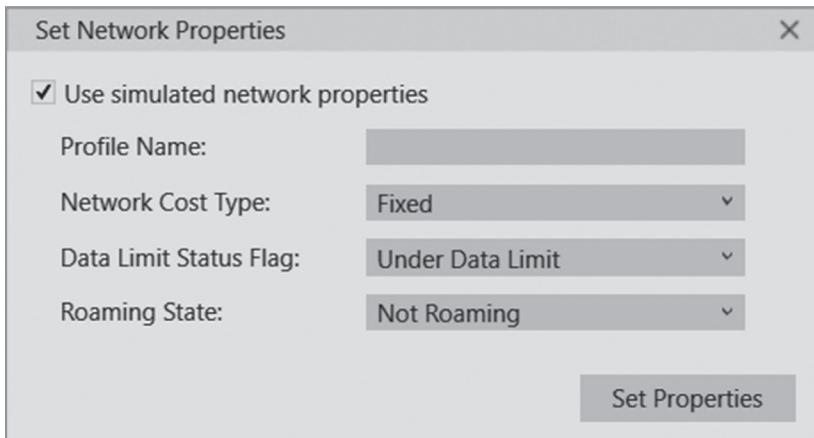


На правій панелі симулятора містяться такі засоби:

- Засоби дотику:** дають змогу імітувати основні дотики або деякі жести, наприклад масштабування й обертання. Звичайно, у розробці програми, для якої потрібно багато складних жестів, ці засоби не допоможуть, здебільшого вони є досить корисними.
- Засоби орієнтації:** змінюють орієнтацію симулятора і дають змогу спостерігати роботу програми в альбомному режимі.
- Вибір роздільної здатності:** допомагає імітувати різні режими роздільної здатності, зокрема за кількістю точок на дюйм. Це важливо для перевірки ресурсів та інтерфейсу.



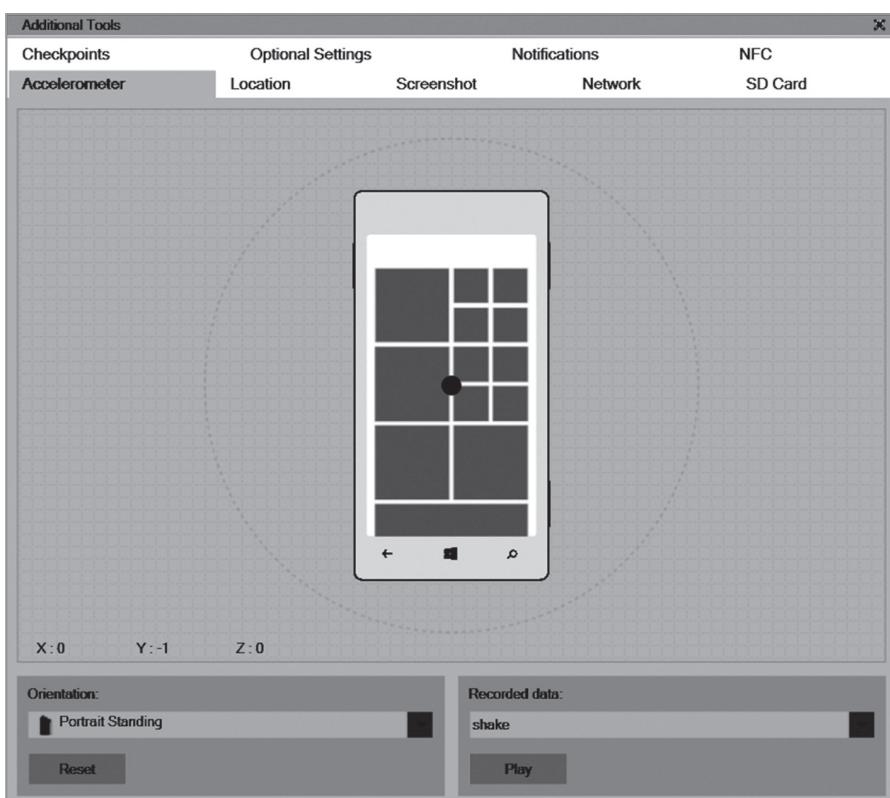
- **Засіб копіювання знімків екрана:** ви послуговуватиметеся ним весь час, відколи ваша програма буде підготовлена для розміщення в Магазині. Щоб скористатися цим засобом, потрібно запустити його та зробити кілька знімків екрана в потрібній роздільній здатності.
- **Мережевий засіб:** слугує для імітації різних умов підключення до мережі. За допомогою цього засобу перевіряють, як працює програма в разі виникнення будь-яких проблем із мережею або за обмеженого доступу до Інтернету.



Зважайте: якщо ви розробляєте програму для Windows 10 на ПК з Windows 8.x, запустити симулятор не вдасться, оскільки власною операційною системою має бути саме Windows 10.

На відміну від симулятора, емулятор є засобом, що працює на віртуальній машині з реальним образом операційної системи. Завдяки цьому емулятори можна запускати й на машині з Windows 8.x. Проте працює емулятор лише з Windows 10 Mobile SKU, тому запустити Windows 10 Desktop SKU не вдасться, але подеколи цілком можна обмежитися ОС Windows Mobile 10 SKU, особливо якщо ви розробляєте програми на машині з Windows 10.

Емулятор застосовує такі самі сенсорні засоби й засоби орієнтації, але він не підтримує миттєву зміну роздільної здатності, натомість дає змогу запускати віртуальні машини, які імітують різні телефони. Крім того, емулятор підтримує кілька додаткових засобів.



До таких належать акселерометр, засіб для визначення розташування, емулятор деяких датчиків, наприклад експонометра і магнітометра, емулятори NFC, SD-карти та сповіщень. Таких засобів чимало, що дає змогу навіть за відсутності телефона перевірити, як працюють ваші програми.

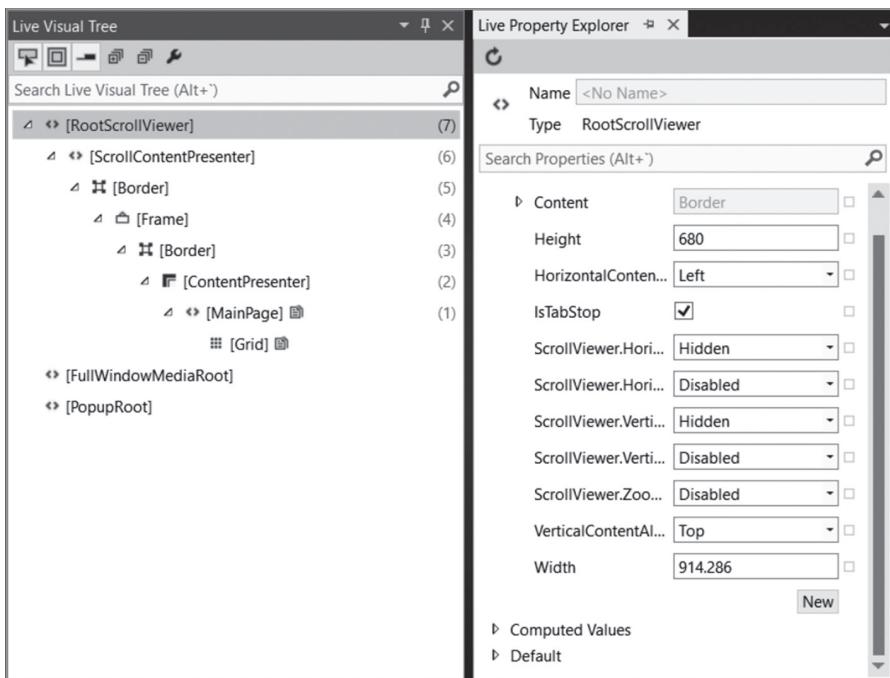
## Live Visual Tree у Visual Studio

Функція Live Visual Tree є доступною в програмах, призначених для Магазину і WPF, зокрема для Магазину Windows 8.x.

Live Visual Tree дає змогу переглядати візуальне дерево XAML, оцінювати властивості елементів і навіть змінювати їх під час виконання. Це дає змогу перевірити, як зміни вплинути на інтерфейс, не перезапускаючи програму. За багатьох сценаріїв цей засіб є корисним, зокрема ним можна скористатися під час виконання наведених нижче завдань.

- Іноді, коли є дуже багато візуальних станів інтерфейсу, важко їх усі перевірити. Live Visual Tree допомагає перевірити властивості всіх елементів керування і, змінюючи їх, зрозуміти, що відбувається – які стани застосовано і яким є результат.
- Візуальний дизайнер у Visual Studio працює належним чином, але за великої кількості зв'язків за допомогою Live Visual Tree можна вносити зміни під час виконання і бачити, як це впливає на дизайн запущеної програми з активними зв'язками і даними.
- Live Visual Tree дає змогу візуалізувати макет із метою показати вирівнювання і простір для розміщення елементів інтерфейсу користувача. Це допомагає знайти помилки, що їх неможливо виявити в режимі конструктора через відсутність реальних даних.
- Перевіряючи кількість елементів у кожному контейнері, можна виявити потенційну проблему в роботі програми.
- Live Visual Tree уможливлює перевірку дерева XAML не тільки для ваших програм – налагоджувач можна легко прикріпити до будь-якої запущеної «XAML-програми», переглянути її візуальне дерево XAML і застосувати будь-які зміни за тим самим алгоритмом.

Для того, щоб у програмі відкрити вікно Live Visual Tree, потрібно запустити її в режимі налагодження і вибрати пункт меню **Debug > Windows**. Інше вікно, пов'язане з Live Visual Tree, – це Live Property Explorer. Live Visual Tree дає змогу переміщатися між елементами XAML, а за допомогою Live Property Explorer можна перевірити, змінювати та створювати властивості.



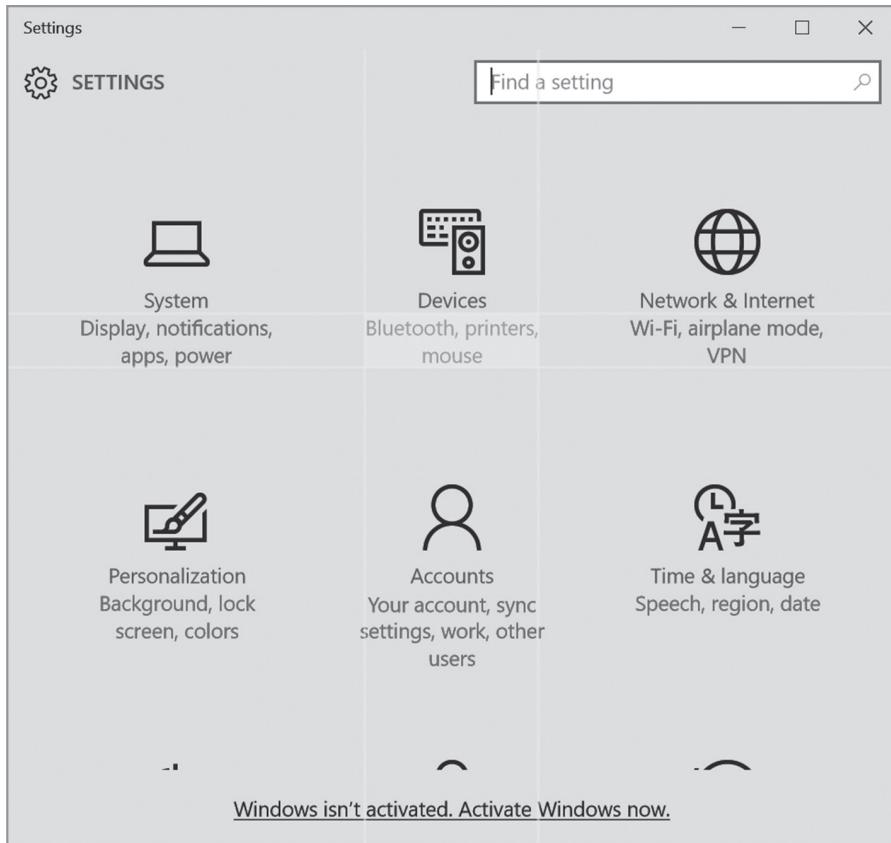
Як видно, Live Visual Tree містить інформацію про кількість елементів XAML усередині кожного контейнера. Зверніть увагу, що відображаються лише елементи візуального дерева XAML. Тож Live Visual Tree містить тільки видимі елементи, і якщо стан інтерфейсу змінюється, Live Visual Tree також зазнає змін у режимі виконання.

Live Property Explorer показує стандартні значення властивостей, значення, успадковані від інших елементів керування, та локальні значення властивостей елементів керування. Звичайно, змінювати можна тільки локальні значення. Можна змінити значення, яке було присвоєно властивості раніше, а також додавати інші доступні властивості та присвоювати їм значення.

У вікні Live Visual Tree містяться дві корисні кнопки. За допомогою першої в запущеній програмі вибирають будь-який елемент, щоб знайти його в дереві XAML. Це особливо потрібно у випадках, коли необхідно знайти кнопку або інший елемент керування, з яким пов'язано обробник події дотику. Друга кнопка візуалізує макет, як тільки ви виберете певний елемент.

Як уже було згадано, Visual Studio можна прикріпити до будь-якого наявного «XAML-вікна». Наприклад, відкрийте вікно настройок і виберіть у Visual Studio

2015 пункт **Attach to Process** (Прикріпити до процесу) в меню **Debug** (Налагодження). У діалоговому вікні **Attach to Process** виберіть процес **SystemSettings** (Системні настройки). Відобразиться структура вікна. Тепер можна змінити його поточні настройки та переглянути макет.

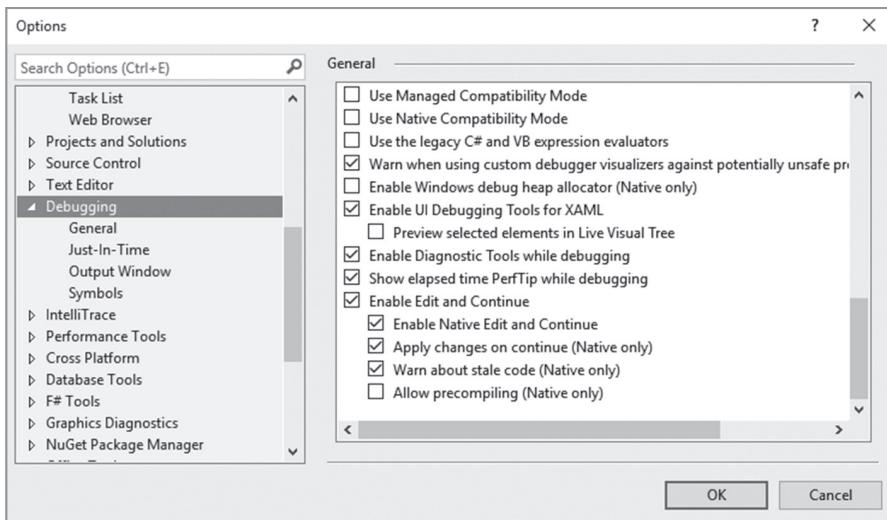


Отже, Live Visual Tree є дуже потужним засобом удосконалення програми.

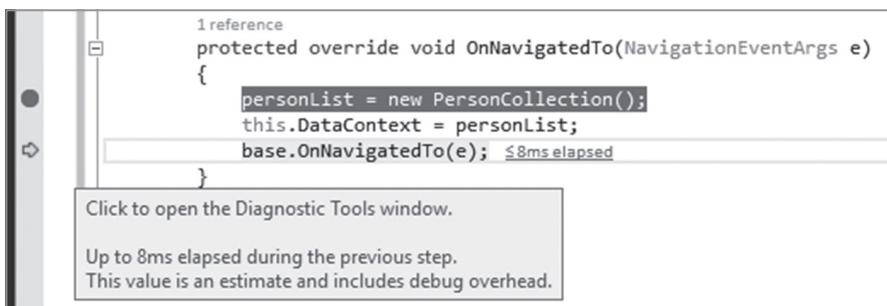
## Засоби профілювання і налагодження у Visual Studio 2015

Поговорімо про інші засоби налаштування програми на базі Universal Windows Platform.

Перш ніж розпочати, переконайтесь, що у властивостях налагодження ввімкнuto засоби **Enable Diagnostic Tools while debugging** (Інструменти діагностики під час налагодження) та **PerfTip while debugging** (Підказки щодо продуктивності під час налагодження). Їх має бути активовано за замовчуванням, але завжди не зайде буде знати, як увімкнути чи вимкнути ці засоби.

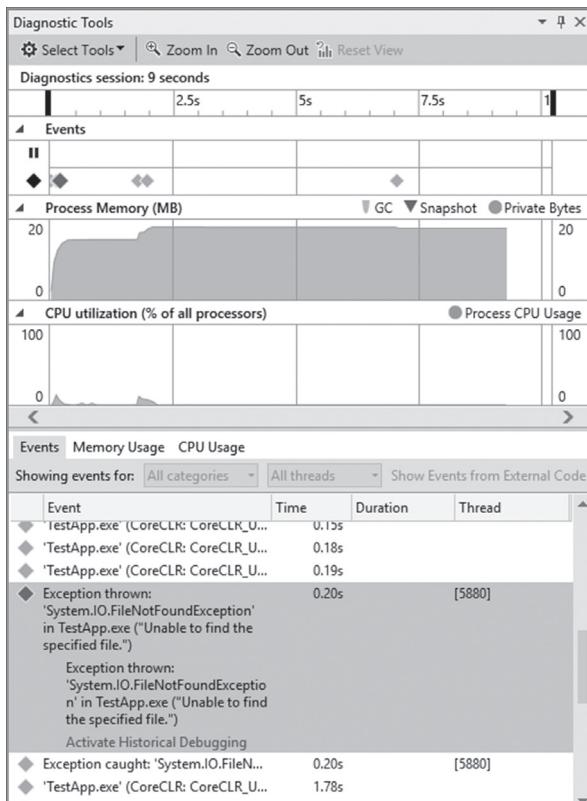


Тепер перейдемо до конкретних функцій. Передусім – до підказок із перевірки продуктивності. Завдяки цій функції в режимі налагодження можна точно дізнатися, скільки часу минуло від проходження останньої точки зупинки:



Щоб отримати такі відомості щодо кожного рядка коду, слід просто пересуваєтися в межах цього коду покроково. Але якщо потрібно дізнатися, скільки часу витрачено на якийсь із блоків, установіть дві точки зупинки: на його початку і в кінці. Ось так цей інструмент допомагає виявити проблеми з продуктивністю.

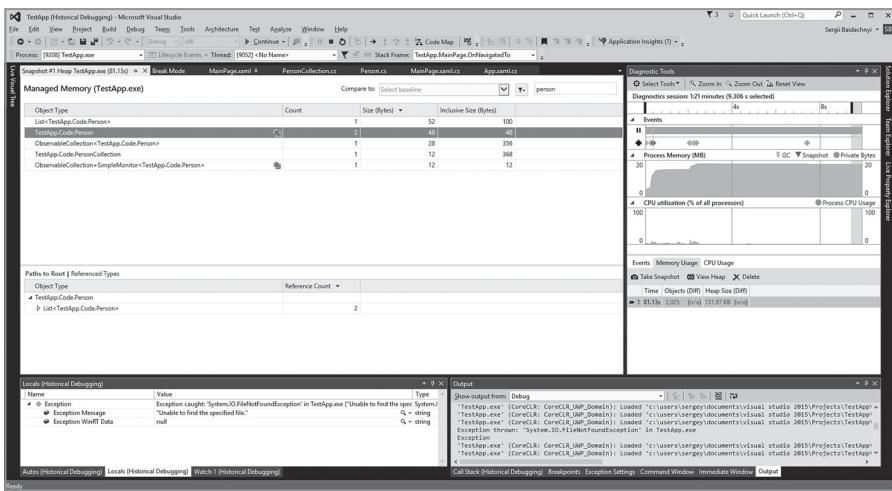
У редакторі Visual Studio такі відомості відображаються у вигляді підказок. Вибравши їх, ви перейдете до вікна **Diagnostic Tools** (Засоби діагностики) — ще одного важливого засобу, що його можна використовувати разом із порадами з перевірки продуктивності.



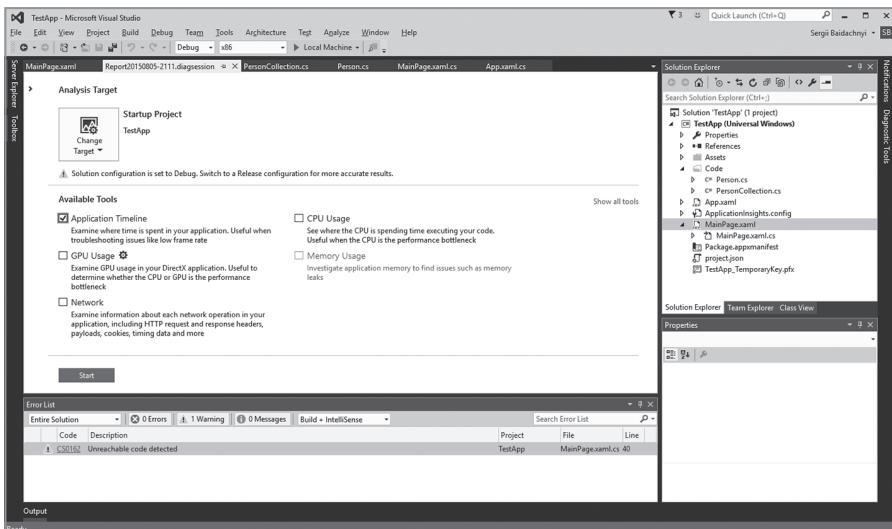
Вікно **Diagnostic Tools** складається з трьох областей. У першій відображено події, наприклад відомості про винятки, навіть коли такий виняток виявлено; повідомлення від вікна виводу, згенеровані класом **Debug**; події **IntelliTrace**, пов'язані з потоками, збірками тощо. У двох інших розділах демонструється навантаження процесора і використання пам'яті під час виконання. Проте цей налагоджувач можна призупинити і, вибравши потрібний інтервал часу, перевірити події і використання пам'яті та процесора.

Крім того, у будь-який момент налагодження програми є можливість зробити знімок пам'яті, а також перевірити кількість посилань для кожного об'єкта, обсяг пам'яті тощо.

## Розділ 30. Тестування і налагодження

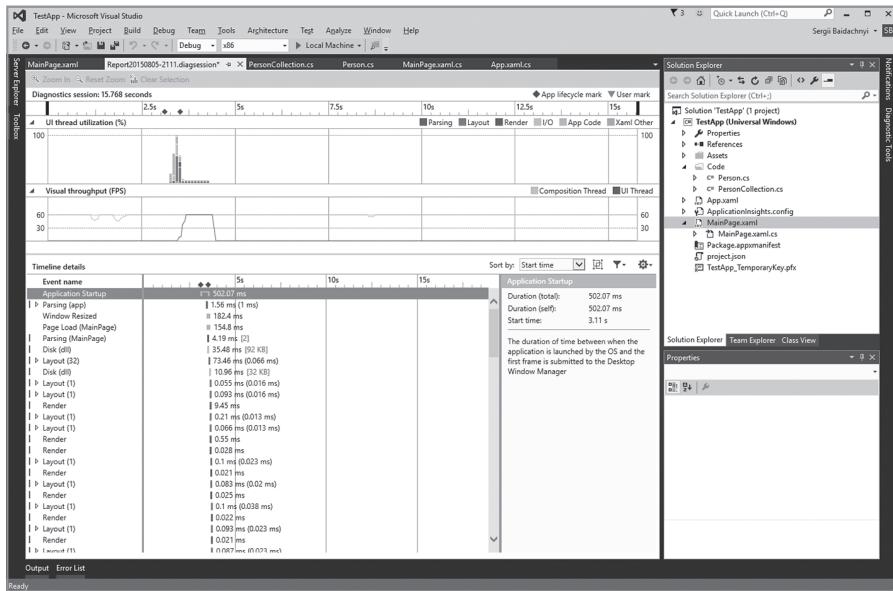


Виконавши команду **Start diagnostic tools without debugging** (Запустити інструменти діагностики без налагодження), ви знайдете зображеній нижче набір засобів.



Це засоби для збирання інформації під час виконання. Оскільки збирання відомостей відбувається не в режимі налагодження, якість даних вища, проте їх неможливо перевірити в режимі реального часу. Засоби, про які йдеться, збирають інформацію про програму, доки ви не зупините їх. Після цього на

підставі зібраних показників буде сформовано звіт, який можна проаналізувати за допомогою зображененої нижче панелі керування.

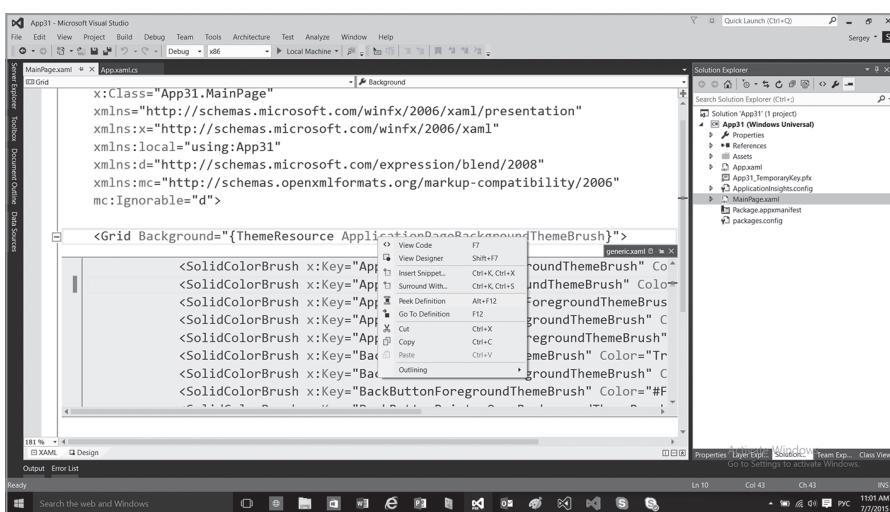


З-поміж усіх засобів найбільше рекомендуємо Application Timeline. Він допомагає виявити проблеми підготовки звітів і переглянути найважливіші частини вашої програми.

## Інструменти XAML у Visual Studio 2015

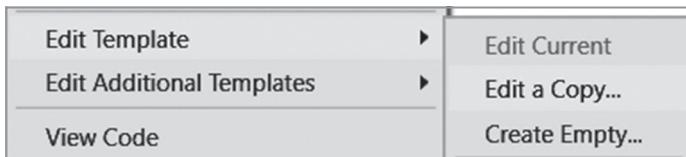
Visual Studio, безумовно, є найкращим з-поміж редакторів. Здивувати чимось тих, хто користується ним, дуже непросто, однак розробникам Visual Studio 2015 це вдалося. Йдеться передусім про функції, пов'язані з редактором XAML.

Насамперед варто згадати про функцію «Вікно перегляду» (peek window). Вона є не тільки в редакторі XAML, але саме у поєднанні з ним набуває особливої цінності, оскільки відкриває багато нових можливостей – наприклад, дає змогу відображати вікна «залежного коду» безпосередньо в поточному вікні.



Це дає перевіряти стилі навіть у **generic.xaml**, переглядати визначення елементів керування, змінювати обробники подій тощо. Слід лише пам'ятати, що не можна закривати основне вікно.

Вікно перегляду можна відкрити як із контекстного меню, так і за допомогою спеціальних засобів Visual Studio 2015, наприклад оновленого редактора шаблонів. Якщо потрібно створити новий шаблон для будь-якого елемента керування, відкрийте контекстне меню і виберіть **Edit Template > Edit a Copy**.

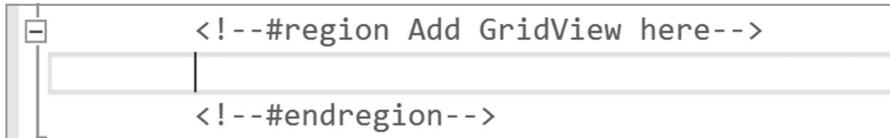


Якщо ви використовуєте засоби Application resources чи Resource Dictionary, Visual Studio спрямує вас до відповідного файлу. Однак перевагою Visual Studio 2015 є те, що цей редактор дає змогу працювати з новим шаблоном у тому самому вікні, редагувати шаблон за допомогою дизайнера та перевіряти чи змінюювати код у вікні перегляду.



У режимі редактора шаблонів Visual Studio 2015 відображає кольорову рамку для візуального редактора.

Ще однією особливістю редактора XAML є його здатність додавати іменовані регіони у коді XAML так само, як у C#. За наведеним нижче синтаксисом можна створити позначеній регіон усередині XAML.



У C# його можна згорнути:



Цю функцію використовують для позначення наявного коду, створення шаблонів для майбутньої роботи тощо.

Отже, ми розглянули три нові функції редактора XAML: «вікно перегляду», оновлений редактор шаблонів і позначені блоки XAML. Усі вони полегшують процес кодування.

Розділ 31.

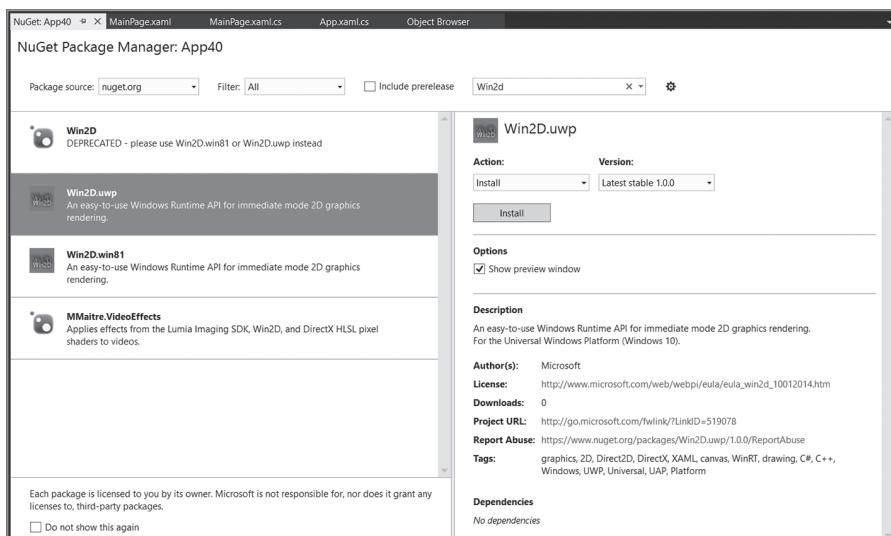
# **Win2D: Як використовувати графіку, не знаючи DirectX**

Direct2D – технологія, використання якої вимагає певної обізнаності з DirectX. На жаль, багато розробників, які пишуть бізнес-програми, ніколи не використовували DirectX у своїй роботі. Ось чому вони вважають функції Direct2D окремими інтерфейсами Windows Runtime API, які призначено для розробників неігрових програм. Win2D є таким інтерфейсом API.

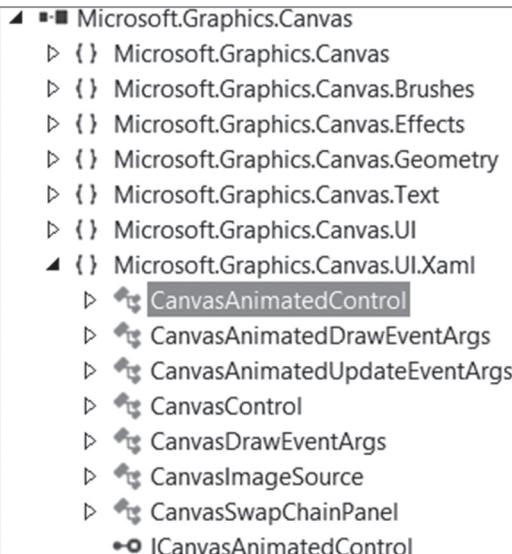
Win2D – це інтерфейс Windows Runtime API, доступний для розробників на C++ і C#. Він дає змогу використовувати 2D-графіку, оптимізовану для GPU, у XAML-програмах для Windows 8x / Windows 10.

Перш ніж починати роботу з Win2D, радимо відвідати блог Win2D Team (<http://blogs.msdn.com/b/win2d/>), де містяться посилання на документацію, зразки коду, вихідний код (Win2D є проектом із відкритим вихідним кодом) тощо. Але щоб скористатися можливостями Win2D повною мірою, треба почати працювати з цим інтерфейсом. Отже, давайте ознайомимося з основами програмування з використанням інтерфейсу Win2D.

Win2D є проектом із відкритим вихідним кодом і не належить за замовчуванням до універсальної платформи Windows. Тому, щоб додати останню версію бібліотеки Win2D ([Win2D.uwp](#)), слід використовувати менеджер пакетів NuGet.



NuGet додасть посилання на збірку **Microsoft.Graphics.Canvas**, що містить усі класи та простори імен Win2D.



Відкрийте простір імен **Microsoft.Graphics.Canvas.UI.Xaml** і перегляньте розміщені там елементи керування XAML. Там міститься два класи, що базуються на **UserControl**: **CanvasAnimatedControl** і **CanvasControl**. Також там є клас **CanvasSwapChainPanel**, що успадковується від класу **Grid**. Давайте розглянемо ці класи детальніше.

Передусім вам необхідно додати простір імен **Microsoft.Graphics.Canvas.UI.Xaml** до файлу XAML:

```
xmlns:canvas="using:Microsoft.Graphics.Canvas.UI.Xaml"
```

Після цього можна буде використовувати в ньому елементи керування. Почнімо з найпростішого, **CanvasControl**.

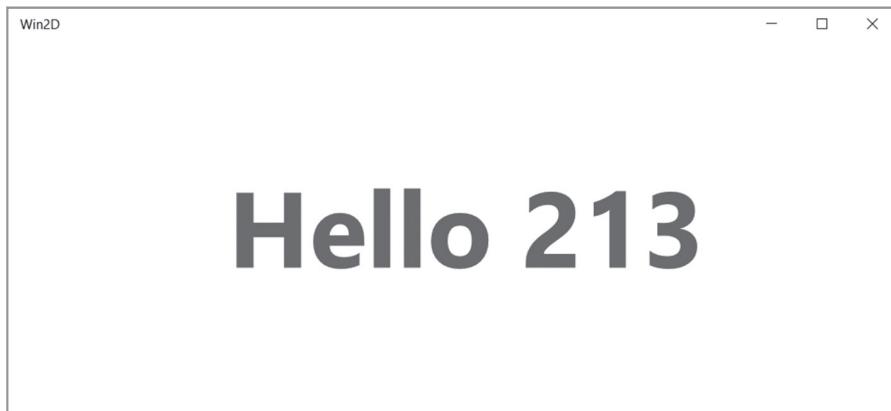
```
<canvas:CanvasControl Draw="CanvasControl_Draw">
</canvas:CanvasControl>
```

**CanvasControl** – це місце, де ми будемо малювати, але десь ще потрібно реалізувати алгоритми малювання. У **CanvasControl** є дві важливі події: **Draw** і **CreateResources**. Давайте почнемо з події **Draw** і розглянемо такий код:

```
int i = 0;
private void CanvasControl_Draw(Microsoft.Graphics.Canvas.
UI.Xaml.CanvasControl sender, Microsoft.Graphics.Canvas.
```

```
UI.Xaml.CanvasEventArgs args)
{
    i++;
    CanvasTextFormat format = new CanvasTextFormat()
    { FontSize = 96, FontWeight = FontWeights.Bold };
    args.DrawingSession.DrawText($"Hello {i}", 200, 100,
        Colors.Green, format);
}
```

У цьому обробнику події ми використовували метод **DrawText** для друку тексту. У класі **CanvasDrawingSession** є безліч різних методів, і він простий у використанні. Зверніть увагу: ми ввели цілочисельну змінну у рядок, що виводиться. Завдяки цій змінній можна побачити, коли працює наш обробник події. Просто запустіть програму та спробуйте працювати з вікном. Ви побачите, що подія **Draw** виникає тоді, коли це необхідно Windows для перемальовування вікна. Таким чином, якщо ви зміните розмір вікна, лічильник буде збільшуватися, а якщо ви не будете звертатися до вікна, лічильник залишиться таким самим.



Саме тому **CanvasControl** зручно використовувати для «статичного» контенту.

Другою важливою подією для **CanvasControl** є подія **CreateResources**. Звернімося до нашого коду. Програма створює об'єкт **CanvasTextFormat**: щоразу, коли викликається обробник події **Draw**. Звичайно, цей об'єкт не дуже складний, але в реальних сценаріях розробники повинні створити багато об'єктів, перш ніж почати малювати. Крім того, розробники мають гарантувати, що всі об'єкти будуть створені до виклику обробника події. Є два способи ініціалізації всіх необхідних об'єктів: призначити обробник події **Draw** динамічно (відразу після ініціалізації методів) або використати обробник події **CreateResources**. Другий спосіб простіший і реалізує деякі елементи для асинхронного програмування.

Давайте змінимо нашу сторінку XAML:

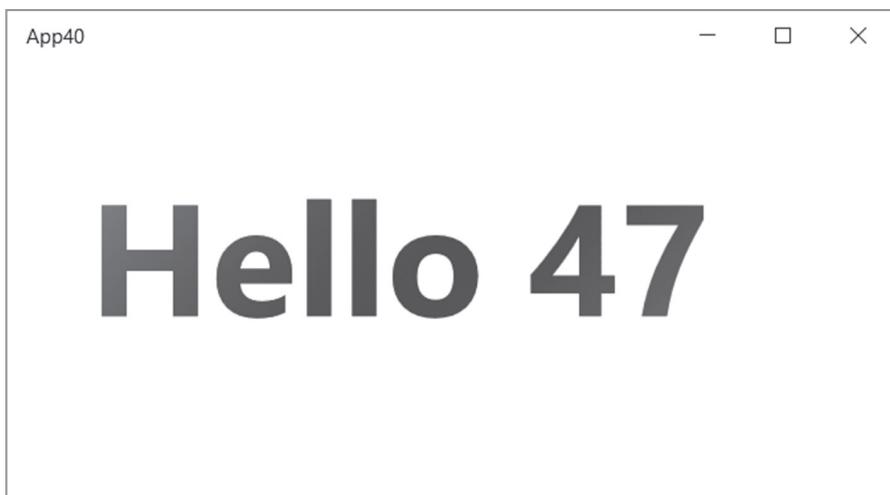
```
<canvas:CanvasControl Draw="CanvasControl_Draw"
CreateResources="CanvasControl_CreateResources">
</canvas:CanvasControl>
```

Ось код:

```
CanvasLinearGradientBrush brush;
CanvasTextFormat format;
private void CanvasControl_CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.CanvasControl sender, Microsoft.Graphics.Canvas.UI.CanvasCreateResourcesEventArgs args)
{
    brush = new CanvasLinearGradientBrush(sender,
    Colors.Red, Colors.Green);
    brush.StartPoint = new Vector2(50, 50);
    brush.EndPoint = new Vector2(300, 300);
    format = new CanvasTextFormat() { FontSize = 96,
    FontWeight = FontWeights.Bold };
}

int i = 0;
private void CanvasControl_Draw(Microsoft.Graphics.Canvas.UI.Xaml.CanvasControl sender, Microsoft.Graphics.Canvas.UI.Xaml.CanvasDrawEventArgs args)
{
    i++;
    args.DrawingSession.DrawText($"Hello {i}", 50, 50,
    brush, format);
}
```

У коді видно, що ми використовували метод **CanvasControl\_CreateResources** для того, щоб створити пензель для тексту.



Використовуючи обробник події **CreateResource**, ви будете певні, що подія **Draw** не виникне до завершення обробника події.

У попередньому прикладі ми використовували обробник події для синхронного методу **CreateResource**, але навіть для простого завантаження зображення необхідно викликати асинхронні інтерфейси API. Зазвичай досвідчені розробники на C# додають ключове слово **async** перед обробником події, але цей підхід не спрацьовує для обробника події **CreateResources**. Замість додавання ключового слова **async** потрібно застосовувати такий підхід:

```
CanvasImageBrush brush;
CanvasTextFormat format;
private void CanvasControl_CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.CanvasControl sender, Microsoft.Graphics.Canvas.UI.CanvasCreateResourcesEventArgs args)
{
    args.TrackAsyncAction(CreateResources(sender).
    AsAsyncAction());
}

async Task CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.CanvasControl sender)
{
    brush = new CanvasImageBrush(sender);
    brush.Image= await CanvasBitmap.LoadAsync(sender,
    "Assets/drone.jpg");
```

```

        format = new CanvasTextFormat() { FontSize = 96,
            FontWeight = FontWeights.Bold };
    }

int i = 0;
private void CanvasControl_Draw(Microsoft.Graphics.Canvas.
    UI.Xaml.CanvasControl sender, Microsoft.Graphics.Canvas.
    UI.Xaml.CanvasDrawEventArgs args)
{
    i++;
    args.DrawingSession.DrawText($"Hello {i}", 50, 50,
        brush, format);
}

```

Як бачите, ми використовували параметр обробника події для виклику методу **TrackAsyncAction**. Цей метод може отримувати **async**-метод як параметр.

Отже, завдяки **CanvasControl** ми можемо зробити «статичний» контент, і це прийнятно для багатьох сценаріїв. Наприклад, цей елемент керування можна використовувати, щоб застосувати ефекти до зображень; можна також створювати діаграми або високопродуктивні програми для візуалізації тексту. Але елемент керування **CanvasControl** не можна застосувати для розробки простих 2D-ігор або аналогічних програм із «динамічним» вмістом. Для цього в інтерфейсі Win2D є інший елемент керування – **CanvasAnimatedControl**.

Можна використати майже той самий код із незначними змінами:

```

<canvas:CanvasAnimatedControl Draw="CanvasAnimatedControl_Draw"
    CreateResources="CanvasAnimatedControl_CreateResources">
</canvas:CanvasAnimatedControl>
CanvasImageBrush brush;
CanvasTextFormat format;
async Task CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.
    CanvasAnimatedControl sender)
{
    brush = new CanvasImageBrush(sender);
    brush.Image= await CanvasBitmap.LoadAsync(sender,
        "Assets/drone.jpg");

    format = new CanvasTextFormat() { FontSize = 96,
        FontWeight = FontWeights.Bold };
}

```

```
int i = 0;
private void CanvasAnimatedControl_Draw(Microsoft.Graphics.
Canvas.UI.Xaml.ICanvasAnimatedControl sender, Microsoft.
Graphics.Canvas.UI.Xaml.CanvasAnimatedDrawEventArgs args)
{
    i++;
    args.DrawingSession.DrawText($"Hello {i}", 50, 50,
        brush, format);
}

private void CanvasAnimatedControl_CreateResources(Microsoft.
Graphics.Canvas.UI.Xaml.CanvasAnimatedControl sender, Microsoft.
Graphics.Canvas.UI.CanvasCreateResourcesEventArgs args)
{
    args.TrackAsyncAction(CreateResources(sender) .
        AsAsyncAction());
}
```

Відобразиться той самий текст, але лічильник буде збільшуватися дуже швидко (через кожні 16,6 мс, або 60 разів за секунду). Тому елемент керування **CanvasAnimatedControl** використовується для створення різноманітних динамічних об'єктів: елементів, які летять, стрибають тощо. Звичайно, застосування одного методу **Draw** недостатньо, тому що для реальних ігор розробник має гарантувати однакову швидкість на всіх пристроях. Тому ігровий цикл складніший, ніж просто виклик методу **Draw**. Але розробники інтерфейсу Win2D знають про це: у класі **CanvasAnimatedControl** містяться додаткові події та корисні **властивості**.

Давайте розглянемо властивості та події, попередньо дещо змінивши обробник події **Draw**:

```
private void CanvasAnimatedControl_Draw(Microsoft.Graphics.
Canvas.UI.Xaml.ICanvasAnimatedControl sender, Microsoft.
Graphics.Canvas.UI.Xaml.CanvasAnimatedDrawEventArgs args)
{
    i++;
    args.DrawingSession.DrawText($"Hello {i}", 50, 50,
        brush, format);
    sender.Paused = true;
}
```

Як бачите, ми використовували властивість **Paused**, щоб зробити паузу в циклі нашої гри. І якщо ви не торкалися вікна програми, видно, що лічильник «заморожений», але відразу після зміни розміру вікна значення лічильника буде швидко зростати. Це відбувається тому, що в режимі паузи метод **Draw** викликається відразу, щойно Windows необхідно перемалювати вікно. Таким чином, обробник події **Draw** краще використовувати тільки для малювання. Якщо вам потрібно змінити будь-які дані в ігровому циклі, краще застосовувати подію **Update**. Обробник події **Update** буде викликатися перед методом **Draw**, й у разі паузи обробник події **Update** буде «заморожено».

```
private void CanvasAnimatedControl_Update(Microsoft.Graphics.Canvas.UI.Xaml.ICanvasAnimatedControl sender, Microsoft.Graphics.Canvas.UI.Xaml.CanvasAnimatedUpdateEventArgs args)
{
    i++;
}
```

Загалом обробники події **Draw+Update** мають викликатися кожні 16,6 мс, але якщо у вас повільний пристрій, можна обйтися без **Update**. Тоді перед викликом **Update** відбуватиметься кілька викликів **Draw**. Це дає можливість забезпечувати ту саму швидкість гри, але можуть виникнути проблеми з тим, як гра промальовується. Якщо це відбуватиметься не дуже часто, то на це можна не зважати, але в деяких випадках можна зменшити час, що витрачається на кожен рух (наприклад, до 30 кадрів на секунду). Це легко зробити за допомогою властивості **TargetElapsedTime**.

Крім того, можна використовувати такі події, як **GameLoopStarting** і **GameLoopStopped**. Вони виникають до і після циклу гри й можуть бути використані для ініціалізації сцени і для знищення всіх об'єктів з пам'яті відповідно.

Зверніть увагу на те, що всі обробники подій виникають в окремому ігровому циклі. Таким чином, жодні дії в них не блокуватимуть потік інтерфейсу. Але розробники повинні зважати на те, як передати дані в ігровий потік з одного інтерфейсу. Наводимо оптимальний спосіб викликати метод **RunOnGameLoopThreadAsync** в потоці інтерфейсу:

```
await myCanvas.RunOnGameLoopThreadAsync(() => /*call something here*/);
```

Останнім елементом керування в просторі імен **Microsoft.Graphics.Canvas.UI.Xaml** є **CanvasSwapChainPanel**. Якщо ви обізнані з розробкою ігор, то маєте знати про ланцюжки підстановки. Основна їх ідея полягає в тому, щоб мати два або більше буферів (сторінки) в грі. Першу сторінку розробник використовує

для подання оновленої сцени на екрані, поки оновлюється інша, попередня сцена. Не використовуйте клас **CanvasSwapChainPanel** або **CanvasSwapChain** із **CanvasAnimatedControl**, тому що останній має власні ланцюжки підстановки. Але якщо ви бажаєте реалізувати власний елемент керування **CanvasAnimatedControl** або подібний до нього, можна використати обидва класи.

```
var swapChain = new CanvasSwapChain(device, width, height, dpi);
swapChainPanel.SwapChain = swapChain;
//draw
swapChain.Present();
```

Отже, зараз ми знаємо, як малювати і в чому полягають відмінності між елементами керування. Тому перейдемо до інших корисних класів, зокрема до ефектів зображення.

Категорія ефектів для зображень в інтерфейсі Win2D зростає найшвидше. Нешодавно було додано ще 10 ефектів, але розробники чекають на появу ще деяких класів, що допомагають створювати власні ефекти. У будь-якому разі інтерфейс Win2D API вже містить понад 50 ефектів, тож кожен розробник зможе знайти корисні для себе фільтри. Ось приклад того, як застосувати два ефекти до одного зображення:

```
GrayscaleEffect effect;
GaussianBlurEffect blurEffect;
async Task CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.
CanvasControl sender)
{
    effect = new GrayscaleEffect();
    var bitmap=await CanvasBitmap.LoadAsync(sender,
        "Assets/drone.jpg");
    effect.Source = bitmap;

    blurEffect = new GaussianBlurEffect();
    blurEffect.BlurAmount = 5;
    blurEffect.Source = effect;
}

private void myCanvas_Draw(Microsoft.Graphics.Canvas.UI.Xaml.
CanvasControl sender, Microsoft.Graphics.Canvas.UI.Xaml.
CanvasDrawEventArgs args)
{
    args.DrawingSession.DrawImage(blurEffect);
```

```

}

private void myCanvas_CreateResources(Microsoft.Graphics.
Canvas.UI.Xaml.CanvasControl sender, Microsoft.Graphics.
Canvas.UI.CanvasCreateResourcesEventArgs args)
{
    args.TrackAsyncAction(CreateResources(sender) .
        AsAsyncAction());
}

```

У цьому прикладі використовувалося розмиття й фільтр градації сірого. Додаткової логіки не було описано, лише використано обробник події **CreateResources**.



Якщо ви використовуєте той самий ефект для одного зображення, його можна застосувати заздалегідь, щоб повернутися до нього пізніше. Але в багатьох випадках вам потрібно побудувати складніші об'єкти, які повинні не просто містити ефект, а й щось малювати та ін. У цьому разі ви можете використовувати клас **CanvasCommandList**, щоб підготувати та зберегти свій об'єкт для подальшого використання. Наводимо дещо модифікований попередній приклад:

```

GrayscaleEffect effect;
GaussianBlurEffect blurEffect;
CanvasCommandList cl;
async Task CreateResources(Microsoft.Graphics.Canvas.UI.Xaml.
CanvasControl sender)
{

```

```
cl = new CanvasCommandList(sender);
using (CanvasDrawingSession clds =
    cl.CreateDrawingSession())
{
    effect = new GrayscaleEffect();
    var bitmap = await CanvasBitmap.LoadAsync(sender,
        "Assets/drone.jpg");
    effect.Source = bitmap;

    blurEffect = new GaussianBlurEffect();
    blurEffect.BlurAmount = 5;
    blurEffect.Source = effect;

    clds.DrawImage(blurEffect);
}
}

private void myCanvas_Draw(Microsoft.Graphics.Canvas.UI.Xaml.
    CanvasControl sender, Microsoft.Graphics.Canvas.UI.Xaml.
    CanvasDrawEventArgs args)
{
    args.DrawingSession.DrawImage(cl);
}

private void myCanvas_CreateResources(Microsoft.Graphics.
    Canvas.UI.Xaml.CanvasControl sender, Microsoft.Graphics.
    Canvas.UI.CanvasCreateResourcesEventArgs args)
{
    args.TrackAsyncAction(CreateResources(sender).
        AsAsyncAction());
}
```

У цьому прикладі ми створили наш власний об'єкт **CanvasDrawingSession** і використали його, щоб намалювати всі необхідні елементи та застосувати фільтри. Відразу після цього ми знищили **CanvasDrawingSession** і зберегли **CanvasCommandList** для майбутнього використання (зробили на нього глобальне посилання). Завдяки такому підходу можна створити всі складні об'єкти заздалегідь, а потім намалювати їх у вигляді зображень.

Коли йшлося про ефекти зображення, не було згадано про геометрію і текст, однак ефекти можна застосувати і до них. Щоб зробити це, вам потрібно використовувати клас **CanvasRenderTarget** для перетворення векторних даних на пікселі:

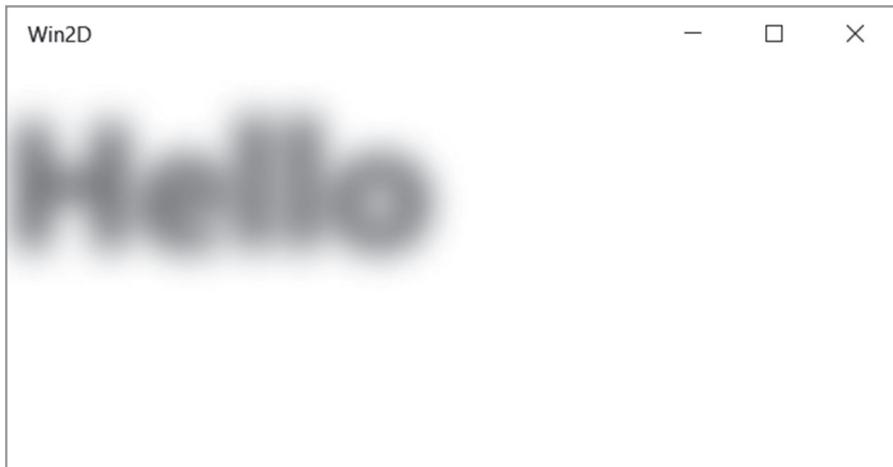
```

private void myCanvas_Draw(Microsoft.Graphics.Canvas.UI.Xaml.
CanvasControl sender, Microsoft.Graphics.Canvas.UI.Xaml.
CanvasDrawEventArgs args)
{
    var myBitmap = new CanvasRenderTarget(sender, 300, 300);
    using (var ds = myBitmap.CreateDrawingSession())
    {
        ds.DrawText("Hello", 0, 0, Colors.Green,
            new CanvasTextFormat() {FontSize=96,
                FontWeight=FontWeights.Bold });
    }

    var blur = new GaussianBlurEffect
    {
        BlurAmount = 10,
        Source = myBitmap
    };
    args.DrawingSession.DrawImage(blur);
}

```

Виконавши цей код, ви побачите таке вікно:



Мабуть, цього достатньо для першого знайомства з інтерфейсом Win2D. Інформацію про багато корисних речей, які допомагають працювати не лише з XAML або дають змогу працювати з зображеннями попіксельно, можна отримати з документації, адже нічого складного у Win2D немає.



Розділ 32.

## **API Composition**

Робота з елементами керування XAML відбувається на рівні Framework. Для сучасних програм під ОС Windows 10 це рівень універсальної платформи Universal Windows Platform. Але перш ніж надіслати всі дані на рівень графічного пристрою для відображення контенту піксель за пікселем, операційні системи необхідно створити візуальний рівень. Саме візуальний рівень відповідає за рендеринг, виконання анімації, пошарове розміщення всіх компонентів тощо. Коли ми кажемо «операційній системі необхідно створити», це означає, що існує спеціальний об'єкт, який можна назвати модулем компонування, – він дає змогу отримати всі компоненти і скомпонувати їх на візуальному рівні.

Раніше це було не дуже важливо, тому що раніше розробники не мали доступу до візуального рівня. У разі застосування XAML існувала можливість працювати лише з елементами керування XAML, але якщо ви потребували чогось особливого, можна було звернутися безпосередньо до DirectX. Проте в UWP вийшла попередня версія інтерфейсу API Composition, і завдяки ній ви можете отримати доступ до візуального рівня.

На момент написання книги API Composition був доступний лише в режимі попереднього ознайомлення. Його не можна було використовувати для публікації програм безпосередньо в Магазині, оськільки розробники Microsoft хотіли насамперед отримати відгуки щодо цього API. Але якщо ви вже почали розмірковувати про зовсім нові стильні користувачькі елементи керування, раджу розглянути API Composition просто зараз.

За допомогою API Composition можна створювати UWP-програми, які не містять XAML-коду взагалі. Але це не дуже цікаво в контексті нашої книги, тому я покажу, як використовувати API Composition разом з елементами керування XAML.

Оскільки API перебуває в режимі попереднього перегляду, виконання навіть на комп'ютері розробника вимагає додаткових можливостей. Отже, щоб розпочати роботу з API Composition, необхідно внести деякі зміни до файлу маніфесту. У кореневий елемент маніфесту (**Package**) дайте такий простір імен:

```
xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities"
```

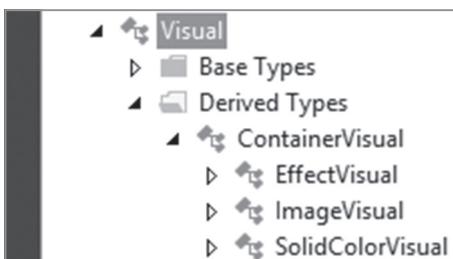
Обмежені можливості не заявлені в стандартних схемах, тому виконання цього кроку є важливим. Коли ви додасте простір імен, можна додавати можливості, які надають доступ до API Composition:

```
<Capabilities>
  <Capability Name="internetClient" />
  <rescap:Capability Name="previewUiComposition"/>
```

```
</Capabilities>
```

Почнімо працювати з найбільш поширеними класами. Усі класи API Composition можна знайти в просторі імен **Windows.UI.Composition**. Непотрібо додавати жодного спеціального пакету, щоб ним користуватись, тому що API Composition включено в ядро Universal Windows Platform. Ось чому весь код буде чудово працювати на будь-якому пристрої під керуванням ОС Windows 10, зокрема на телефонах.

У просторі імен **Windows.UI.Composition** можна знайти багато класів. Але, як згадувалося вище, має бути модуль компонування і класи, які подають об'єкти на візуальному рівні, а також сам цей рівень. Знайти ці класи легко. Першим із них є **Compositor** – він дає змогу створювати різні об'єкти і містить посилання на графічний пристрій. Базовий клас для різних візуальних об'єктів – це **Visual**. У ньому міститься багато властивостей, зокрема **Opacity**, **Scale**, **RotationAngle** тощо. Усі ці властивості дають змогу визначити положення візуального об'єкта і різні типи перетворень. Але безпосередньо створити об'єкт класу **Visual** неможливо. Це просто базовий клас, який визначає загальні властивості, але не може мати жодного об'єкта. Отже, давайте подивимося, чи є які-небудь класи, успадковані від **Visual**:



Перший нащадок – клас **ContainerVisual**. Об'єкти цього класу можна створювати, використовуючи методи з **Compositor**. Насправді клас **ContainerVisual** може об'єднувати будь-які інші візуальні об'єкти. Цей клас сам по собі не містить нічого нового, крім властивості **Children**, що вказує на колекцію. Використовуючи цю властивість, можна додати будь-яку кількість візуальних об'єктів і об'єднати їх разом. Тож перш ніж робити що-небудь, необхідно знайти спосіб отримати посилання на цей об'єкт.

Нарешті, **ContainerVisual** має три похідні класи: **EffectVisual**, **ImageVisual** і **SolidColorBrushVisual**. Усі ці класи представляють найпростіші візуальні об'єкти. Щойно ми отримаємо посилання на **ContainerVisual**, можна створити будь-який із цих об'єктів і додати його до колекції.

Використовуючи клас **ElementCompositionPreview**, можна отримати об'єкт **ContainerVisual** для будь-якого елемента керування. Це передбачувано, тому що кожен елемент XAML представлений об'єктом **Visual** на візуальному рівні. Найкращий спосіб інтегрувати API Composition і XAML – це використовувати наявний об'єкт **Visual**.

Із цієї причини, якщо на вашій сторінці є елемент **Grid** під назвою **myGrid**, можна отримати посилання на **ContainerVisual**, використовуючи такий код:

```
visual = ElementCompositionPreview.GetContainerVisual(myGrid)  
as ContainerVisual;
```

І, оскільки **ContainerVisual** має посилання на **Compositor**, можна легко починати працювати з цим. Подивіться на такий код:

```
visual = ElementCompositionPreview.GetContainerVisual(myGrid)  
as ContainerVisual;  
comp = visual.Compositor;  
  
SolidColorBrush background = comp.CreateSolidColorBrush();  
background.Size = new System.Numerics.Vector2(100, 100);  
background.Color = Colors.Red;  
visual.Children.InsertAtBottom(background);
```

У цьому коді ми створили об'єкт класу **SolidColorBrush** і додали його до **ContainerVisual**, пов'язаного з об'єктом **myGrid**. Запустивши цей код, ви побачите червоний прямокутник розміром 100x100 пікселів.

Давайте спробуємо створити візуальне зображення. Це теж не дуже важко, але вам доведеться використовувати більше властивостей, щоб налаштувати зображення. Код наведено нижче.

```
visual = ElementCompositionPreview.GetContainerVisual(myGrid)  
as ContainerVisual;  
comp = visual.Compositor;  
  
CompositionImage profilePic = comp.DefaultGraphicsDevice.  
CreateImageFromUri(  
    new Uri("ms-appx:///Assets/drone.jpg"));  
  
ImageVisual profilePicVisual = comp.CreateImageVisual();  
profilePicVisual.Image = profilePic;  
profilePicVisual.Stretch = CompositionStretch.UniformToFill;
```

```
profilePicVisual.Size = new System.Numerics.Vector2(100,100);
profilePicVisual.Offset = new System.Numerics.Vector3(100,100,0);

visual.Children.InsertAtTop(profilePicVisual);
```

Оскільки зображення може бути джерелом для різних завдань, об'єкт класу **ImageVisual** не містить самого зображення. Він містить тільки властивості, пов'язані із зображенням (як його подати) і посилання на **CompositionImage** (**ICompositionSurface**), що є джерелом зображення. Щоб створити **CompositionImage**, ми використовували посилання на графічний пристрій за замовчуванням і метод **CreateImageFromUri**.

Зверніть особливу увагу, що ми використовували метод **InsertAtTop**, але в попередньому прикладі був використаний **InsertAtBottom**. У нашому випадку неважливо, який метод використовувати, але за допомогою різних методів можна розмістити новий візуальний елемент у колекції саме там, де він потрібен.

Гаразд. Зараз ми знаємо, як додавати зображення і кольорові прямокутники до візуального контейнера, але основні можливості API Composition – це системи ефектів та анімації. Давайте обговоримо систему ефектів.

Ми вже знаємо, що додати кілька ефектів можна, використовуючи клас **EffectVisual** і створюючи об'єкти цього класу за допомогою **Composer**. Але клас **EffectVisual** – це просто подання візуального елемента, він не містить жодного методу, який би давав змогу розробити і застосувати якісь ефекти. Натомість він містить посилання **Effect**, і ви повинні заздалегідь підготувати ефект та застосовувати його за допомогою цього посилання. Отже, подивімось на клас **Composer** і дізнаємось, як створити сам ефект, тому що клас **CompositionEffect**, об'єкт якого повинен бути присвоєний властивості **Effect**, не містить жодного конструктора. Оскільки будь-який ефект вимагає опису, немає сенсу створювати всі ефекти «з нуля», особливо якщо ви хочете застосувати одинаковий ефект до кількох елементів. Ось чому клас **Composer** містить метод, який дає змогу створити фабрику замість певного ефекту, а використовуючи фабрику, можна відтворити ефект потрібну кількість разів.

Тож процес є досить звичним: створюємо фабрику на основі опису, використовуємо фабрику, щоб створити об'єкт **CompositionEffect** і присвоїти його властивості **Effect EffectVisual**. Але є проблема, де взяти опис ефекту. API Composition не надає власних описів ефектів, тому що безліч ефектів вже реалізовано в інших підсистемах UWP. Натомість API Composition використовує опис ефектів з Win2D. Наприкінці червня 2015 року API Composition підтримував такі ефекти: **ArithmeticComposite**, **Blend**, **Saturation** і **Composite**. Ознайомтеся з останньою

версією API, щоб дізнатися, чи підтримуються інші ефекти. Але навіть у цих чотирьох ефектів є багато параметрів, які дають змогу реалізувати величезну кількість їхніх варіантів.

Таким чином, щоб застосувати ефект, необхідно використовувати менеджер пакетів NuGet та інсталювати пакет Win2D.UWP. У наступному прикладі ми використовуємо ефект **Saturation**. Зверніть особливу увагу, як цей код використовує клас **CompositionEffectSourceParameter**, щоб створити параметр, який ви можете заповнити відразу після генерування фабрикою нового об'єкта **CompositionEffect**.

```
visual = ElementCompositionPreview.GetContainerVisual(myGrid)
    as ContainerVisual;
comp = visual.Compositor;

var rectangle=comp.CreateSolidColorVisual();
rectangle.Color = Colors.Green;
rectangle.Size= new System.Numerics.Vector2(320, 220);
rectangle.Offset= new System.Numerics.Vector3(90, 90, 0);

visual.Children.InsertAtTop(rectangle);

CompositionImage profilePic =comp.DefaultGraphicsDevice.
CreateImageFromUri(new Uri("ms-appx:///Assets/drone.jpg"));

SaturationEffect saturationEffect = new SaturationEffect();
saturationEffect.Saturation = 0;
saturationEffect.Source = new CompositionEffectSourceParameter
("image");
saturationEffect.Name = "saturationW2D";

CompositionEffectFactory effectFactory =
    comp.CreateEffectFactory(saturationEffect);

CompositionEffect eff = effectFactory.CreateEffect();
eff.SetSourceParameter("image", profilePic);

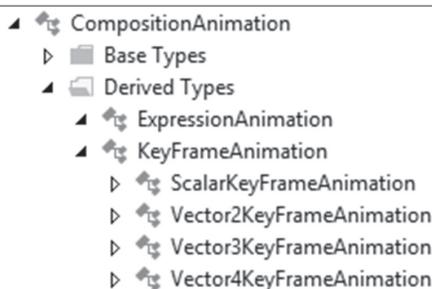
EffectVisual saturationVisual = comp.CreateEffectVisual();
saturationVisual.Effect = eff;
saturationVisual.Size = new System.Numerics.Vector2(300, 200);
saturationVisual.Offset = new System.Numerics.Vector3(10, 10, 0);
rectangle.Children.InsertAtBottom(saturationVisual);
```

У цьому прикладі ми використали об'єкт **SolidColorBrush** як контейнер для **EffectVisual**, щоб створити контур і мати можливість застосувати анімацію до контуру і зображення водночас. Виконавши цей код, ви побачите на екрані чорно-біле зображення.



Нарешті давайте обговоримо систему анімації, яка підтримується API Composition.

API Composition підтримує два типи анімації: **KeyFrameAnimation** та **ExpressionAnimation**, але перший є базовим класом для більш як чотирьох конкретних класів, які працюють зі скалярними значеннями і векторами:



Кожен із цих типів можна створити за допомогою об'єкта **Compositor** і будь-якого з типів анімації ключових кадрів – у цьому немає нічого особливого. Ми вже використовували стандартні анімації ключових кадрів у XAML, але анімація виразів більш цікава, це щось нове. Давайте почнемо з простої анімації ключових кадрів. У коді нижче ви можете побачити, як анімувати кут візуального об'єкта:

```
var keyframe = comp.CreateScalarKeyFrameAnimation();

keyframe.InsertKeyFrame(1.0f, angle);
keyframe.Duration = TimeSpan.FromMilliseconds(1000);

animator=visual.ConnectAnimation("RotationAngle", keyframe);
animator.Start();
```

У цьому коді ми створили простий об'єкт **KeyFrameAnimation** і застосували його до об'єкта класу **Visual**, використовуючи метод **ConnectAnimation**, який повертає об'єкт класу **CompositionPropertyAnimator**. Використовуючи останній, можна запустити анімацію і призначити власний обробник події, який виконуватиметься, тільки-но анімація завершиться.

Подивіться на аналогічний код, який використовує ключовий кадр **Expression** замість стандартного:

```
var keyframe = comp.CreateScalarKeyFrameAnimation();

keyframe.InsertExpressionKeyFrame(1.0f, "visualObj.
RotationAngle+initial");
keyframe.Duration = TimeSpan.FromMilliseconds(2000);
keyframe.SetReferenceParameter("visualObj", rectangle);
keyframe.SetScalarParameter("initial", 1.0f);

animator=visual.ConnectAnimation("RotationAngle", keyframe);
animator.Start();
```

Виконавши цей код, ви побачите, що аніматор весь час використовує **RotationAngle**. Тож прямокутник буде обертатися протягом двох секунд. Звичайно, ви можете створити більш складні вирази, засновані на різних об'єктах.

Нижче ми наводимо повний лістинг коду, який обертає зображення нескінченно.

```
Compositor comp;
ContainerVisual visual;
```

```
SolidColorBrush rectangle;
CompositionPropertyAnimator animator;
float angle = 1;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    visual = ElementCompositionPreview.
        GetContainerVisual(myGrid)
    as ContainerVisual;
    comp = visual.Compositor;

    rectangle = comp.CreateSolidColorBrush();
    rectangle.Color = Colors.Green;
    rectangle.Size = new System.Numerics.Vector2(320, 220);
    rectangle.Offset = new System.Numerics.Vector3(190, 190, 0);
    rectangle.RotationAngle = 45;

    visual.Children.InsertAtTop(rectangle);

    CompositionImage profilePic =
comp.DefaultGraphicsDevice.CreateImageFromUri(
new Uri("ms-appx:///Assets/drone.jpg"));

    SaturationEffect saturationEffect = new SaturationEffect();
    saturationEffect.Saturation = 0;
    saturationEffect.Source = new
        CompositionEffectSourceParameter("image");
    saturationEffect.Name = "saturationW2D";

    CompositionEffectFactory effectFactory =
comp.CreateEffectFactory(saturationEffect);

    CompositionEffect factory = effectFactory.CreateEffect();
    factory.SetSourceParameter("image", profilePic);

    EffectVisual saturationVisual = comp.CreateEffectVisual();
    saturationVisual.Effect = factory;
    saturationVisual.Size = new System.Numerics.Vector2
(300, 200);
    saturationVisual.Offset = new System.Numerics.Vector3
(10, 10, 0);
    rectangle.Children.InsertAtBottom(saturationVisual);
```

```
        ApplyAnimation(rectangle, angle);

        base.OnNavigatedTo(e);
    }

private void ApplyAnimation(ContainerVisual visual, float angle)
{
    if (animator!=null)
    {
        animator.Dispose();
        animator = null;
    }

    var keyframe = comp.CreateScalarKeyFrameAnimation();

    keyframe.InsertExpressionKeyFrame(1.0f, "visualObj.
RotationAngle+initial");
    keyframe.Duration = TimeSpan.FromMilliseconds(4000);
    keyframe.SetReferenceParameter("visualObj", rectangle);
    keyframe.SetScalarParameter("initial", angle);

    animator=visual.ConnectAnimation("RotationAngle", keyframe);
    animator.AnimationEnded += Animator_AnimationEnded;
    animator.Start();
}

private void Animator_AnimationEnded
(CompositionPropertyAnimator sender,
AnimationEndedEventArgs args)
{
    sender.AnimationEnded -= Animator_AnimationEnded;
    angle = -angle;
    ApplyAnimation(rectangle, angle);
}
```

У цьому коді ми використали подію **AnimationEnded**, щоб почати анімацію у протилежному напрямку. Таким чином, прямокутник обертається в протилежних напрямках.

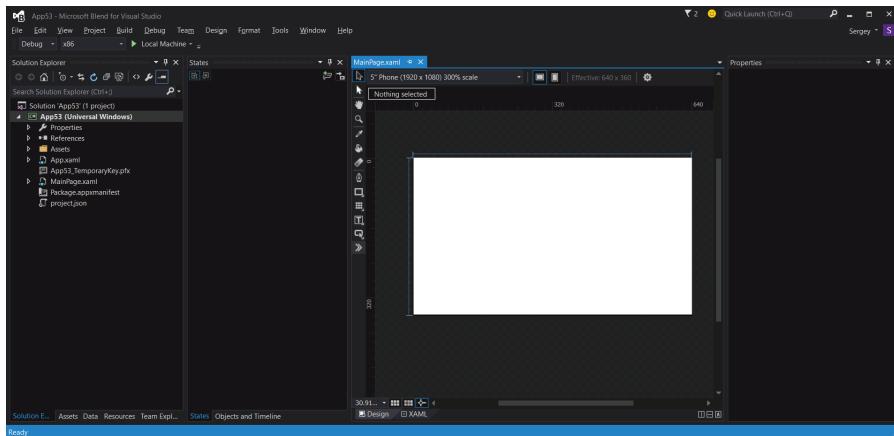
Якщо ви хочете дізнатися більше про API Composition, підписуйтесь на новини команди розробників у Twitter – **@wincomposition**.

Розділ 33.

## **Як використовувати Blend**

## Огляд Blend

Розробник може швидко створити програму для Windows 10 будь-якого рівня складності, але найважливішим завданням є забезпечення зручності для користувача. Адже можна реалізувати подання даних та бізнес-логіку, але не забезпечивши належний дизайн і зручність використання, успіху досягти навряд чи вдасться. Ось чому дуже важливо залучити до роботи дизайнера, який створить привабливий інтерфейс, піктограми, зображення тощо. Завдяки шаблону MVVM ви зможете поділитися кодом XAML із дизайнером і сконцентрувати свою увагу на C#-коді, працюючи разом із дизайнером над одними тими самими формами водночас. Але є одна проблема: зазвичай дизайнерам не надто подобається Visual Studio. Ось чому Microsoft підтримує ще один інструмент, який встановлюється з Visual Studio – Blend для Visual Studio.



Microsoft Blend – окремий інструмент, який дає змогу редагувати інтерфейси XAML, використовуючи безліч дизайнерських функцій, і може стати в пригоді дизайнерам і розробникам, яким потрібно ці інтерфейси змінювати. Завдяки Microsoft Blend ви зможете працювати з одним форматом рішення. Blend можна використовувати навіть для створення нового проекту, щоб згодом відкрити його у Visual Studio. Можливо, ви вже знаєте, що в меню **View** у Visual Studio є пункт **Design in Blend**, а в меню **View** у Blend – пункт **Edit in Visual Studio**. Тож ви зможете легко переключати контекст між двома редакторами. Але Microsoft Blend є досить потужним інструментом, з якого можна запускати та налагоджувати програми безпосередньо.

Нижче описано найпоширеніші способи використання **Blend**.

- Створення сторінок XAML за допомогою графічних примітивів і елементів керування.
- Створення анімацій і перетворень.
- Робота із шаблонами.
- Створення адаптивних тригерів.

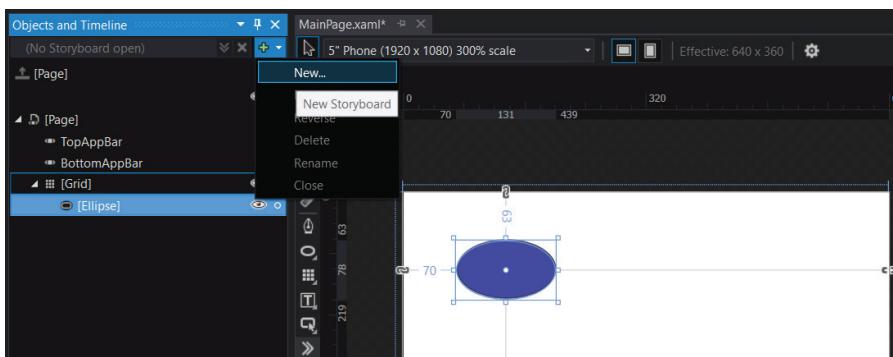
Створюючи новий проект у Blend, ви побачите, що Blend підтримує не тільки UWP-проекти, але й проекти для Silverlight, WPF і Windows 8.x / Windows Phone 8.x. Тобто в Blend можна створити будь-який проект, що використовує XAML, а також редагувати його інтерфейс.

Microsoft Blend має безліч функцій, але ми зупинимося лише на деяких із них. А саме, розглянемо редагування анімації, використання шаблонів і створення тригерів стану в Blend. Ці завдання дуже важливі і, як правило, виконати їх у Visual Studio не надто легко. З Blend зробити це значно простіше.

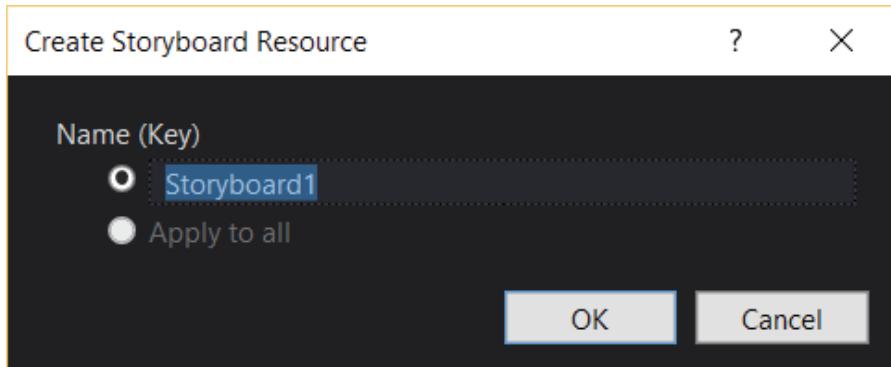
## Як створювати анімації в Blend

Пропоную створити пустий інтерфейс і розмістити там еліпс або будь-який інший елемент. Спробуймо оживити цей елемент у Blend.

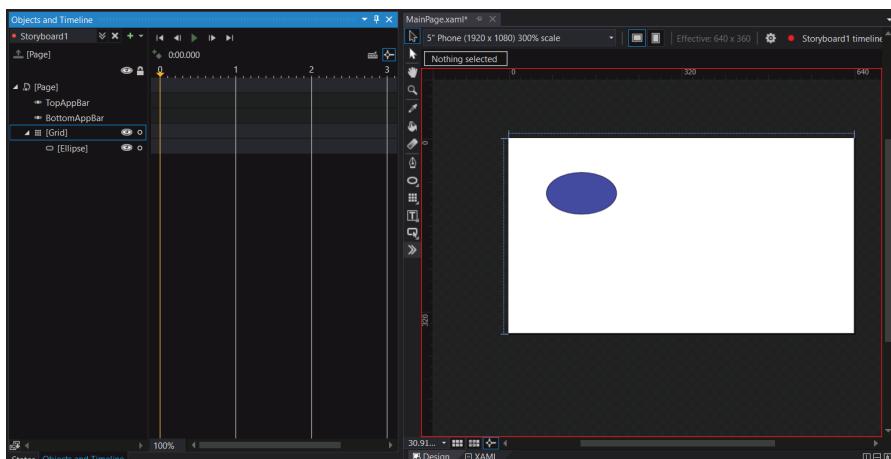
Щоб створити анімацію у Visual Studio, спершу необхідно створити об'єкт **Storyboard** (розкадрування) і розмістити в ньому всі необхідні анімації, передаючи безліч параметрів і редагуючи безпосередньо XAML-код. За допомогою Blend усе це можна зробити в режимі дизайнера. Просто відкрийте вікно **Objects and Timeline** (Об'єкти та часова шкала) і створіть новий об'єкт розкадрування за допомогою меню:



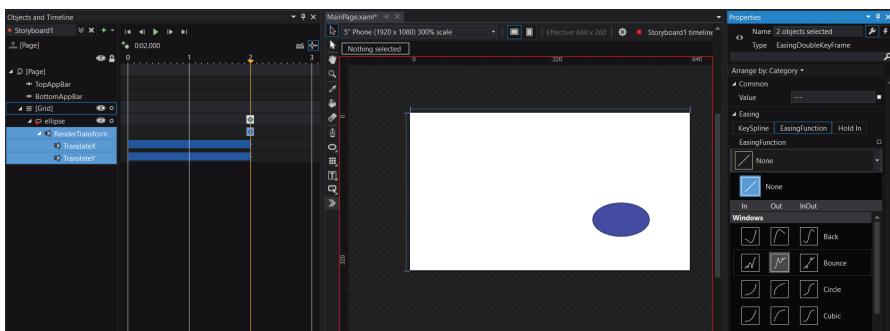
Visual Studio попросить ввести ім'я розкадрування:



Blend переключить розкадрування в режим записування:



У цьому режимі можна вибрати будь-які елементи інтерфейсу й за допомогою перетягування по черзі застосовувати до них анімації. Просто виберіть будь-який елемент керування та властивість, перетягніть покажчик часу на кілька секунд уперед і змініть властивість. Можна змінити положення, колір або будь-який інший елемент, що піддається анімації:



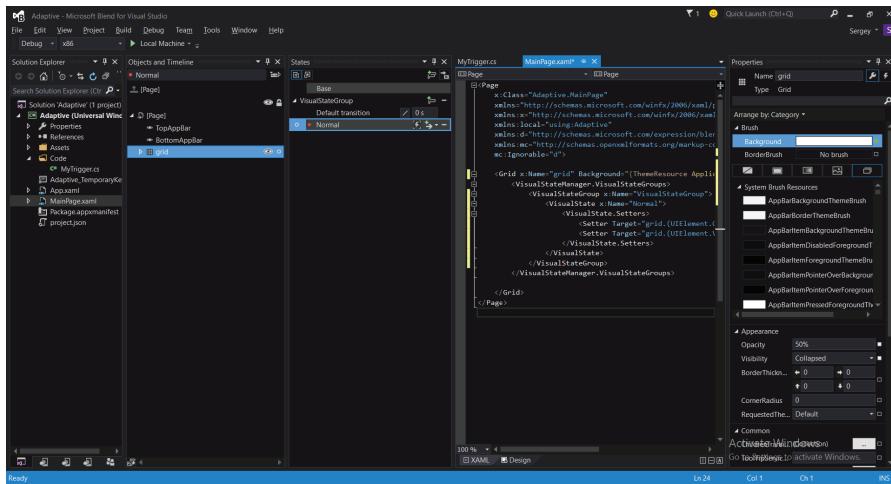
Щойно це буде зроблено, Blend створить анімацію, яку можна редагувати з використанням функцій уповільнення або інших параметрів. Щоб відтворити розкадрування, натисніть кнопку **Play**. Перегляньте результат.

Якщо ви певний час попрацюєте з вікном **Objects and Timeline**, то виявите, що його можливості дуже потужні та прості у використанні. Навіть якщо ви не розболяли інтерфейс у Blend, можна просто скопіювати створені анімації та додати в потрібне місце у своєму проекті.

## Стани й тригери станів у Blend

Щоб скласти уявлення про нові можливості, просто створіть новий проект у Blend і у вікні **Solution Explorer** виберіть файл **MainPage.xaml**. Щойно ви це зробите, у вікні **Objects and Timeline** відобразиться структура вашої сторінки. Зазвичай для створення візуальних станів використовується головний контейнер, такий як **Grid**. Отже, щоб створити кілька візуальних станів, краще вибрати кореневий елемент **Grid** в **Object and Timeline** і вікно **States** для редагування візуальних станів.

Blend дає змогу редагувати візуальні стани взагалі без кодування. За допомогою вікна **States** можна створити потрібну кількість станів. Щойно ви виберете стан, Blend переключить редактор у режим записування (**Ctrl+R**). У цьому режимі можна змінювати властивості будь-яких елементів керування, і всі ці зміни буде автоматично включені до вибраного стану. Пропоную створити тільки один стан за допомогою кнопки **Add State** (Додати стан) у вікні **States** і вибрати основну сітку, щоб змінювати певні властивості. Наприклад, за допомогою вікна **Properties** можна змінити властивості **Opacity** (Прозорість) і **Visibility** (Видимість).



Відкривши документ XAML, ви побачите такий код:

```
<VisualState x:Name="Normal">
    <VisualState.Setters>
        <Setter Target="grid.(UIElement.Opacity)" Value="0.5"/>
        <Setter Target="grid.(UIElement.Visibility)" Value="Collapsed"/>
    </VisualState.Setters>
</VisualState>
```

Як бачимо, редактор Blend повністю інтегровано з новим підходом Universal Windows Platform завдяки сеттерам.

Якщо ви бажаєте використовувати старий підхід або анімувати властивості від одного стану до іншого, можна продовжувати застосовувати часову шкалу, а замість сеттерів Blend використовуватиме анимацію.

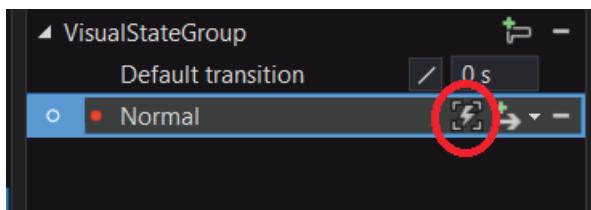
Ще однією особливістю Universal Windows Platform є тригери станів, які дають змогу перевести інтерфейс з одного стану в інший взагалі без кодування. UWP підтримує тільки **AdaptiveTrigger**, але можна створювати власні тригери. Розгляньмо використання тригерів у Blend.

Пропоную додати клас у проект, який матиме тільки одну властивість. Цей клас повинен бути успадкований від **StateTriggerBase**:

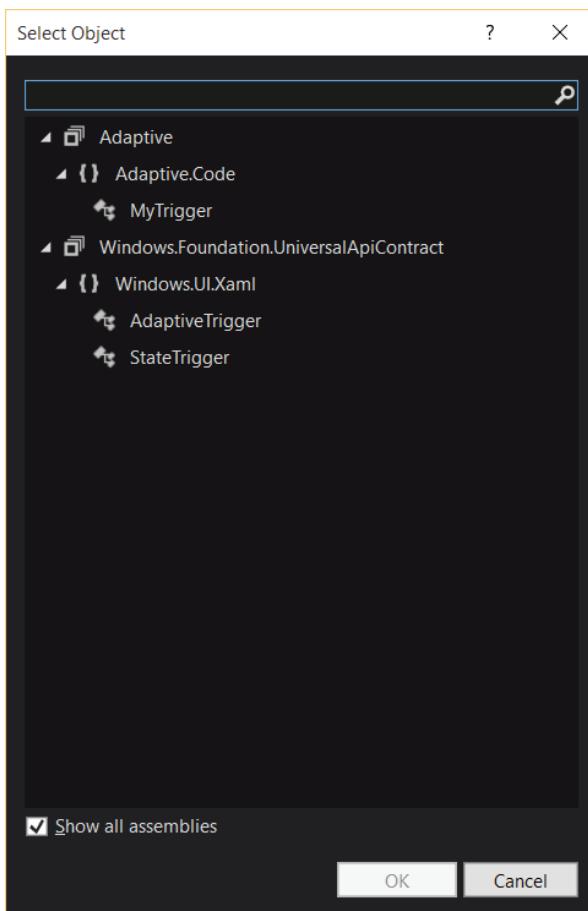
```
class MyTrigger: StateTriggerBase
{
    public int State { get; set; }
}
```

Звичайно, цей клас не має жодного сенсу, оскільки не містить логіки. Але його достатньо для відображення в Blend. Просто потрібно перекомпілювати програму, оскільки Blend виконуватиме пошук збірки проекту.

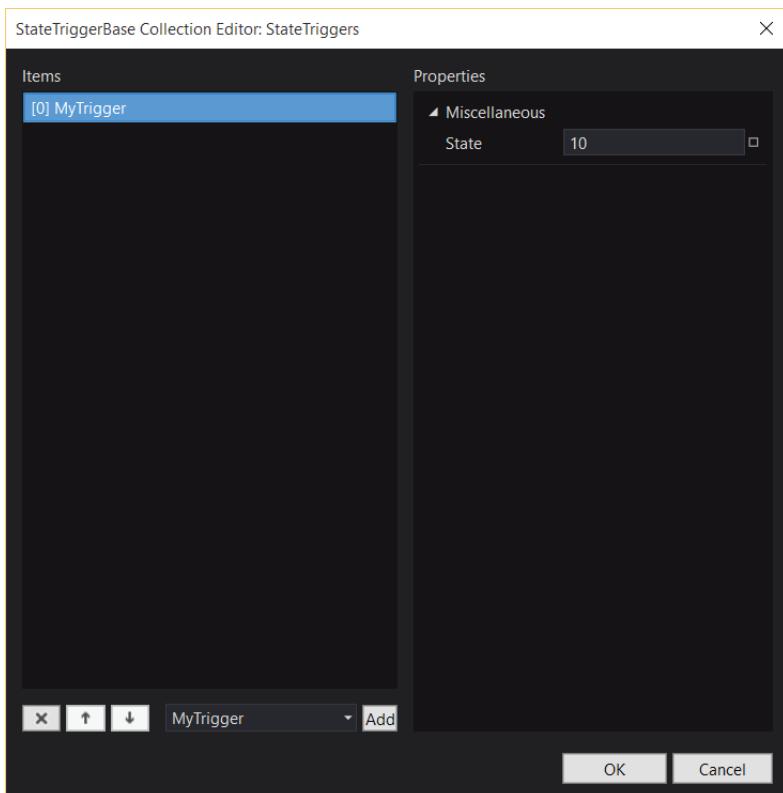
Спробуймо застосувати створений тригер до одного зі станів, використовуючи вікно **States**. Зауважте, що це вікно містить нову кнопку, яка дає змогу редагувати адаптивні тригери:



Просто натисніть кнопку для будь-якого стану, і в Blend відобразиться діалогове вікно для вибору доступних тригерів проекту:



Тепер виберіть потрібний тригер, і Blend дасть змогу ініціалізувати в ньому всі загальнодоступні властивості (у нас є тільки одна):



Введіть будь-яке значення й натисніть кнопку **OK**. Blend створить такий код:

```
<VisualState.StateTriggers>
    <Code:MyTrigger State="10"/>
</VisualState.StateTriggers>
```

Щоб розробити справжній адаптивний інтерфейс, знадобиться створити багато станів і використовувати безліч тригерів. І найкращим інструментом для цього є Microsoft Blend.



Розділ 34.

## **Інтернет речей**

## Огляд мікроконтролерів

Інтернет речей (IoT) – дуже «гаряча» тема сьогодні. Усім подобається створювати власні пристрої, підключати їх до Інтернету та розробляти власне програмне забезпечення для власних сценаріїв поведінки. Це стало можливим завдяки широкій доступності мікроконтролерів, які ви можете знайти всередині телевізорів, холодильників, плеєрів. Вони допомагають нам створювати нові автоматизовані системи та пристрої, які можна носити із собою.

Дуже важко зробити огляд мікроконтролерів, оскільки їх представлено на ринку надто багато. А інвестувавши \$ 15 чи трохи більше, ви можете створити власного робота, систему безпілотного керування або автоматизації домашнього господарства. У кожному разі, ми постараємося розглянути найпопулярніші контролери на ринку.

### Arduino

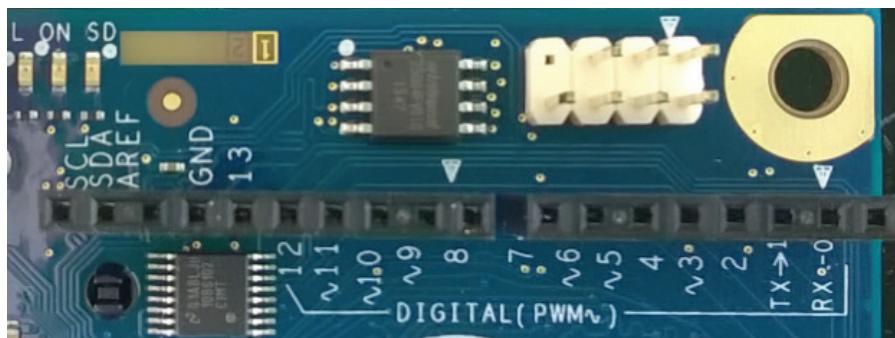
Це дуже популярна платформа з відкритим вихідним кодом для створення прототипів на базі мікросхеми ATMega. Вона підтримує все, що потрібно в більшості випадків: цифровий вхід, аналоговий вхід, I2C, SPI та інших важливі речі. Arduino коштує досить дешево. Плату Arduino Uno ви можете придбати за 30–35 доларів, або створити свою власну плату за 15 \$. Ale Arduino – не тільки плата, яку можна знайти на <http://arduino.cc>, а ще IDE для розробки відповідного програмного забезпечення і велика спільнота.



Тому, відвідавши сайт Arduino, ви можете завантажити і встановити Arduino IDE для Windows, Mac або Linux, а на відкритих форумах – знайти відповідь на будь-яке питання. Працюючи з Arduino, легко зрозуміти, як підключати до плати зовнішні датчики та компоненти і як почати розробку для Інтернету речей.

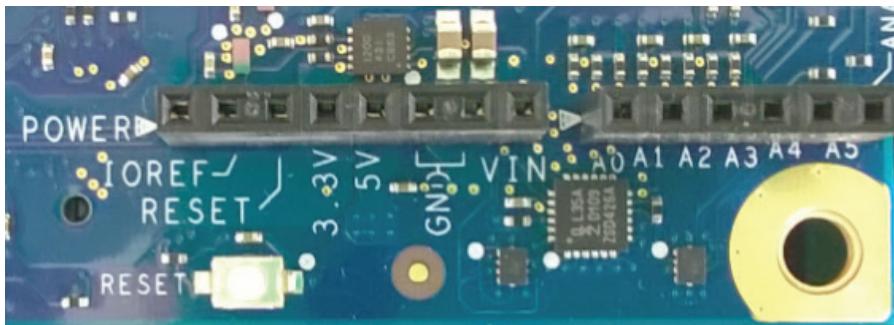
Розгляньмо найважливіші частини плати для розробників. Ми обговорюватимемо Arduino Uno, оскільки цю плату легко використовувати без жодних додаткових компонентів.

Arduino Uno містить два набори контактів, через які ми можемо керувати вхідними та вихідними даними. Перший набір – це цифрові контакти. Ви можете використовувати ці контакти для виведення в режимі «є/немає»: вихідна напруга може становити 0 В або 5 В.



Отже, ми маємо 13 контактів для цифрового вводу/виводу. Але, як ви, можливо, помітили, деякі з цих контактів мають спеціальну позначку «~». Такі контакти ми можемо використовувати для надсилання вихідних сигналів в імпульсному режимі. Це дає нам можливість створювати ефект реостату, коли можна надсилати тільки певний відсоток струму за одиницю часу. Ви можете застосовувати цей ефект, щоб регулювати яскравість освітлення або температуру в квартирі тощо. Як правило, ці контакти називаються контактами PWM (широтно-імпульсної модуляції).

Другий набір контактів (A0–A5) використовується тільки для вводу. Але це має бути аналоговий ввід, як, наприклад, дані з термометрів, потенціометрів, змінних резисторів і т. д.



Цей набір контактів важливий, коли ми отримуємо дані, які описують більше двох станів. Ви можете використовувати ці контакти в більш складних проектах.

Нарешті, ви можете знайти ще кілька контактів, таких як POWER, 5V, GND і так далі. Деякі з них призначені для керування платою, деякі (5V і VIN) використовуються як джерело струму, а контакт GND – для заземлення.

Отже, про контакти ми дещо дізналися, і зараз саме час, щоб розглянути інструменти розробки для Arduino.

Для встановлення Arduino IDE вам просто необхідно відвідати веб-сайт <https://www.arduino.cc/en/Main/Software> і завантажити інсталяційний файл. Відразу після встановлення інструментів Arduino ви можете підключити вашу Arduino Uno до комп’ютера за допомогою USB і запустити Arduino IDE.

Якщо все гаразд, ваша плата Arduino буде знайдена, і ви можете використовувати пункт меню **Tools**, щоб налаштувати правильний COM-порт і тип плати. Загалом ці значення мають бути правильно ініціалізовані за замовчуванням, але в разі виникнення проблем ви можете зробити це вручну.

Найпростіший спосіб перевірити плату Arduino – запустити готовий приклад Blink. Для цього необхідно вибрати меню **File > Examples > Basic – Blink**. Arduino IDE відкриє готовий до розгортання код, який буде використовувати цифровий контакт 13 для надсилання сигналів ввімкнення/вимкнення:

```

Blink | Arduino 1.6.4
File Edit Sketch Tools Help
Blink
/*
Blink
Turns on an LED on for one second, then off for one second, repeatedly.

Most Arduinos have an on-board LED you can control. On the Uno and
Leonardo, it is attached to digital pin 13. If you're unsure what
pin the on-board LED is connected to on your Arduino model, check
the documentation at http://arduino.cc

This example code is in the public domain.

modified 8 May 2014
by Scott Fitzgerald
*/

// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);    // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);     // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}

```

1 Arduino Uno on COM4

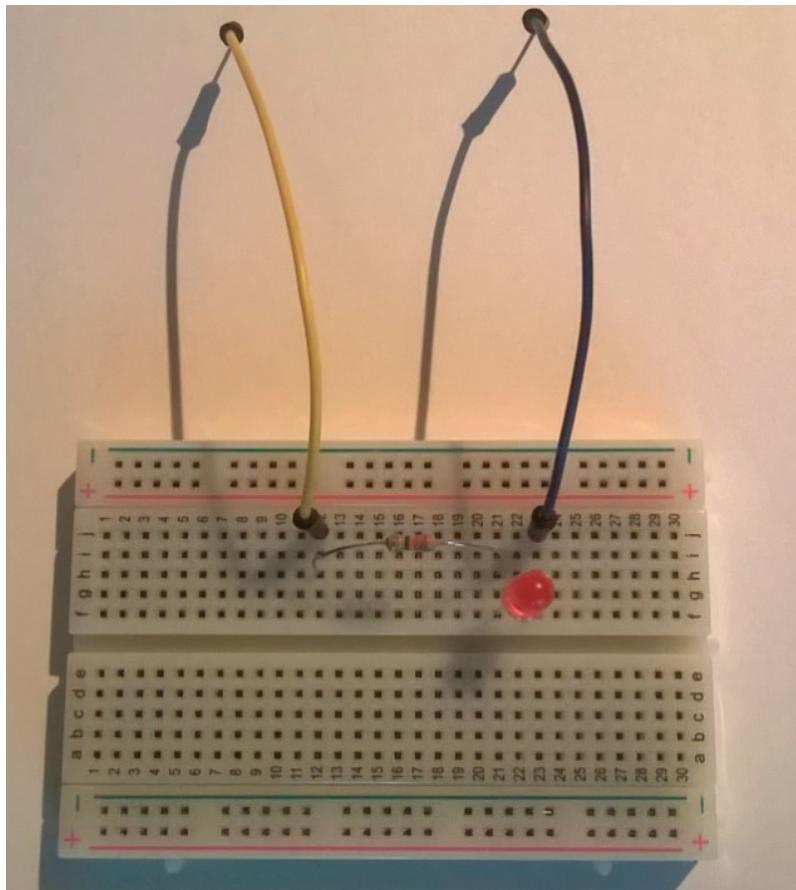
Звичайно, ви не повинні нічого підключати до цифрового контакту 13, але цей контакт пов'язаний зі світлодіодом на платі. Тому, якщо ви запустите цей приклад, то зможете побачити, як блимає на платі світлодіод.

Подивіться на функції **setup** і **loop** у нашому коді. Із функції **setup** викликається лише метод **pinMode**, який інформує плату про те, що ми будемо працювати з контактом під номером 13 в режимі виводу. Метод **loop** містить чотири рядки, найважливіші з яких – виклики функції **digitalWrite**. Ця функція приймає такі два параметри, як номер контакту і стан. Стан **LOW** відповідає напрузі 0 В, а стан **HIGH** – напрузі 5 В.

Ми використаємо наданий у шаблоні код, але для зовнішнього світлодіоду. Для цього нам знадобляться такі компоненти.

- LED – світлодіод, що зазвичай споживає напругу близько 1,7 В. Таким чином, ми маємо зменшити напругу 5 В і для цього скористаємося резистором.
- Резистор – щоб забезпечити таке змінення напруги, потрібен резистор на 330 Ом.
- Макетна дошка – як правило, ви будете використовувати її для того, щоб створити прототип плати з метою тестування та дослідження. Її можна купити окремо або знайти в складі багатьох комплектів.
- 2 дроти.

Ось як виглядає макет:



Щоб зрозуміти, як він працює, потрібно дізнатися про деякі особливості макетної дошки. Вона має кілька рядів контактів, що позначені цифрами від 1 до 30. Контакти в кожному ряді з'єднані між собою. Отже, жовтий провід використано для підключення контакту на платі до ряду 12 на макетній дошці. Тому коли ми подамо на цифровий контакт напругу **HIGH**, струм з'явиться в ряді 12. Резистор підключається до рядів 12 і 21. Тепер підключимо світлодіод – довший провід (анод) під'єднаємо до того самого ряду контактів (21), а коротший (катод) – до ряду 23. До ряду 23 підключимо також синій провід.

Запустіть код ще раз і ви побачите миготіння світлодіоду на макетній дошці.

## Arduino та мова C#

Якщо вам дуже подобається мова C#, можете використовувати її і для Arduino. Відвідайте сайт <http://www.visualmicro.com/> і завантажте плагін Visual Micro для Visual Studio.

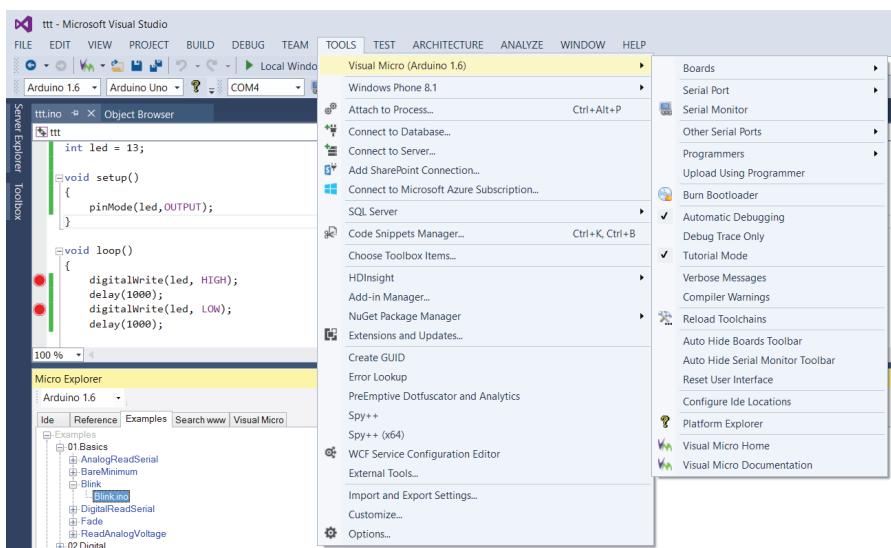
Є дві версії цього плагіну: безкоштовна та платна. Платна версія забезпечує краще налагодження і безліч додаткових можливостей. Але якщо ви тільки почали працювати з Arduino, можете використовувати безкоштовну версію, яка є розширенням Visual Studio і підтримує IntelliSense, основні засоби налагодження, налаштування, зразки тощо. Звичайно, цей плагін не ідеальний, але важливо, що можна використовувати багато цікавих можливостей безкоштовно. А якщо ви зираєтесь розробляти багато програм для Arduino, можете витратити 29 доларів і отримати всі платні функції.

Отже, для того, щоб встановити плагін, ви повинні спершу встановити Arduino IDE. Зараз Visual Micro підтримує Arduino 1.6.3, але все швидко змінюється, тому варто знайти правильну версію Arduino IDE, перш ніж її встановлювати. Після того, як ви встановите Arduino IDE, можете встановити плагін Visual Micro для Visual Studio, але перевірте версію вашого Visual Studio, тому що Visual Micro не підтримує Express-версії, хоча ви можете використовувати версію Community Edition, що є також безкоштовною.

Під час первого запуску Visual Studio після встановлення плагіну буде відображене вікно із пропозицією налаштувати каталог Arduino IDE та придбати плагін. Для каталогу Arduino IDE потрібно вибрати ту саму папку, що й для зберігання ваших незавершених проектів.

Щоб почати новий проект у Visual Studio, тепер необхідно вибрати не **New > Project**, а **New > Sketch Project**. Знадобилося 30 секунд, щоб знайти, де можна створити проект, але це невелика проблема.

Отже, як тільки ви створили проект, можна починати кодування. Visual Studio має кілька спливаючих списків, де можна вибрати версію Arduino SDK, тип вашої плати і порт, який ви використовуєте для її підключення. Крім того, є приклади, вікно властивостей і багато інших інструментів, які ви можете знайти в меню **Tools > Visual Micro**.



Отже, плагін Visual Micro може бути дуже корисним і дійсно допомогти здійснювати розробку у вашому улюбленому середовищі. І якщо ви розробляєте не тільки код для Arduino, але і деякі служби та зовнішні програми, можете робити це в межах однієї розробки.

## Netduino

Наступна мікроплата – це Netduino. Ви можете знайти декілька її моделей на сайті <http://www.netduino.com/hardware/>. Мабуть, варто порекомендувати найсучаснішу плату Netduino 3 з підтримкою Wi-Fi.

У будь-якому разі всі плати Netduino базуються на .NET Micro Framework – платформі з відкритим вихідним кодом, яка розроблена для пристроїв з невеликим обсягом пам'яті та обмеженими ресурсами. Каркас .NET Micro Framework може бути запущений на пристроях з 64 КБ пам'яті, що важливо для дешевих і дійсно невеликих пристроїв. Про випуск .NET Micro Framework було оголошено достатньо давно, близько 9 років тому; ця платформа має дуже цікаву історію. Вона підтримує різні мікроконтролери, найбільш популярними з яких на сьогоднішній

день є Netduino і Gadgeteer, причому плати Netduino є Arduino-сумісними. Це дуже важливо, тому що зараз легко знайти багато різних розробок для Arduino.

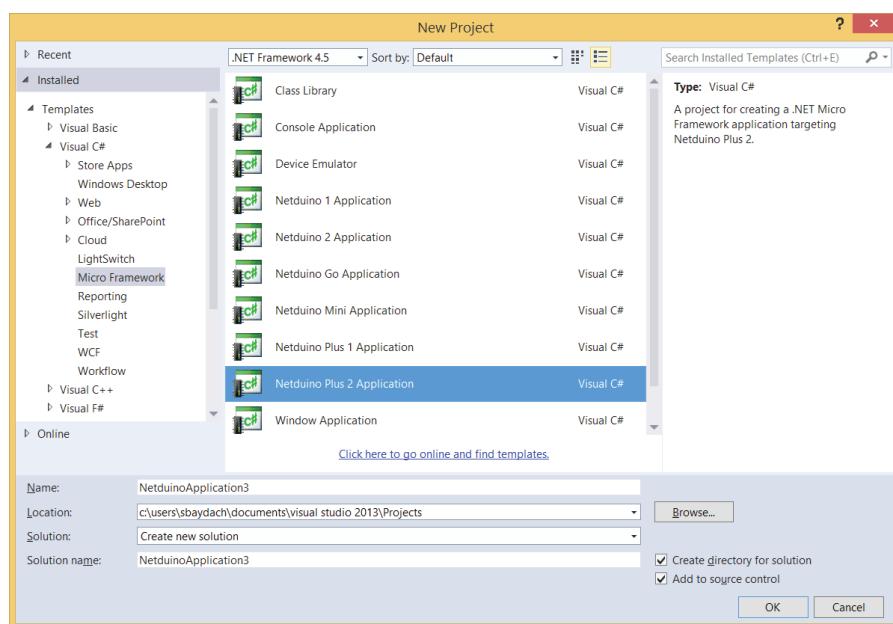
У нас є плата Netduino Plus 2, на якій ми спробуємо зробити кілька невеликих проектів. Звичайно, якщо у вас є певна плата, потрібно буде встановити деяке ПЗ на комп’ютері, наприклад .NET Micro Framework SDK, інструменти для Visual Studio і SDK до вашої плати. Для Netduino Plus 2 ми б порекомендували виконати такі кроки:

- Оновіть прошивку Netduino – коли ви купляєте плату, на ній може бути встановлена стара версія .NET Micro Framework. Якщо версія прошивки на платі старіша за 4.3, ви не зможете використовувати найновішу версію SDK. Інформацію про оновлення прошивки можна знайти на форумі Arduino.
- Встановіть .NET Micro Framework SDK та інструменти для Visual Studio.
- Встановіть Netduino SDK.

У будь-якому випадку, перш ніж щось скачувати, перевірте останній версії SDK і прошивки.

Після того, як встановите все необхідне програмне забезпечення, скористайтеся кабелем USB -> micro USB, щоб підключити плату до комп’ютера. Цей кабель забезпечує живлення Netduino. У Visual Studio ви можете створювати програми на основі спеціальних шаблонів для плат Netduino, використовуючи C# і багато класів з .NET Micro Framework і Arduino. Visual Studio підтримує не тільки розгортання програм для Netduino, але й налагодження та засіб Intellisense. Таким чином, розробка на платформі Windows дуже подібна до розробки для мікроплат. Це дійсно чудово.

## Windows 10 для C# розробників



## Raspberry Pi 2

Решта матеріалу цього розділу буде присвячено платі Raspberry Pi 2. Насамперед зробимо стислий огляд її можливостей.



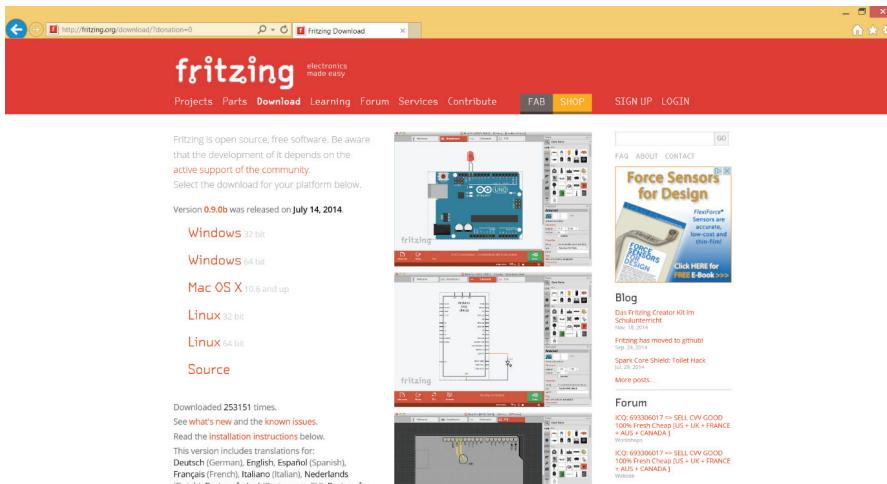
Raspberry Pi 2 – це не Arduino-сумісна плата, вони цілком різні. Зазвичай ви будете використовувати Arduino і Netduino для простих проектів або основних компонентів більш складних проектів, тому що ці плати мають дуже обмежені можливості в порівнянні зі справжніми комп'ютерами. Але Raspberry Pi 2 має всі основні компоненти звичайних комп'ютерів: GPU, порт HDMI, підтримка аудіо, 4 USB-порти, мережевий сокет, потужний процесор тощо. Але, що більш важливо, ви можете встановити на Raspberry OC Windows 10.

Далі ми побачимо, як використовувати знання платформи UWP для розробки на Raspberry програм під Windows 10.

## Як створити схеми

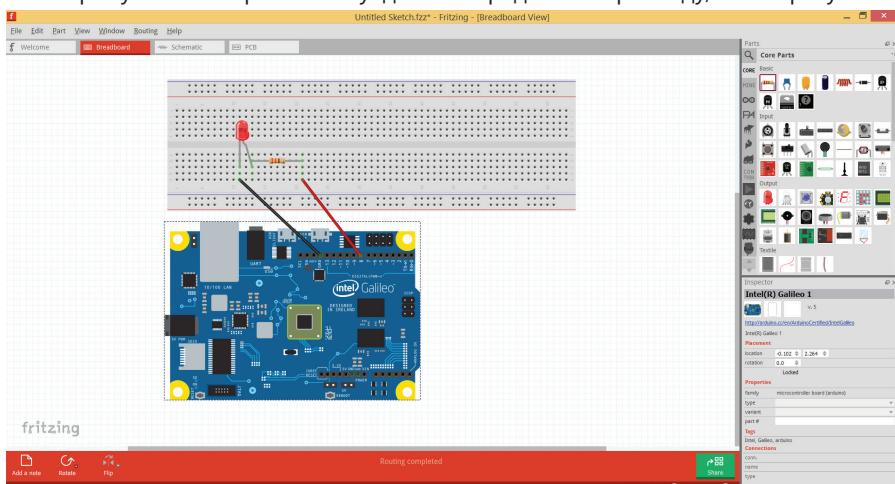
Для створення схем ми рекомендуємо використовувати безкоштовну програму Fritzing з відкритим вихідним кодом. Якщо ви створюєте кілька проектів, вам знадобиться інструмент, який дає змогу розробляти схеми, і ним може стати Fritzing.

Ця програма була розроблена Університетом прикладних наук в Потсдамі, Німеччина. Ви можете завантажити її із сайту <http://fritzing.org>, де містяться посилання на версії для різних операційних систем, таких як Windows, Linux та OS X.



Все дуже просто: перетягніть необхідні компоненти з області **Core parts** (Базові компоненти) на макет і з'єднайте їх, щоб отримати електричне коло.

Ми спробували створити схему для попереднього прикладу, і ось результат:



Якщо деяких компонентів у програмі не пропонується, можна пошукати їх в Інтернеті. Наприклад, на сайті спільноти Intel є чудовий макет плати Galileo. Компоненти можна імпортувати у вигляді fzpz-файлів на вкладці **Mine**.

## Windows 10 IoT Core

Як ми згадували в розділі 1 першої книги, Microsoft підтримує різні SKU для Windows. Один із них – це Windows IoT Core, який можна встановити на IoT-пристрої безкоштовно.

Для початку відвідайте <https://dev.windows.com/en-US/iot> і завантажте збірку та інструкції зі встановлення. Як бачите, всі інсталяційні файли загальнодоступні, і потрібно просто кланцнути посилання, щоб завантажити образ.

Перш ніж встановлювати Windows 10 на Raspberry, зверніть особливу увагу на такі аспекти.

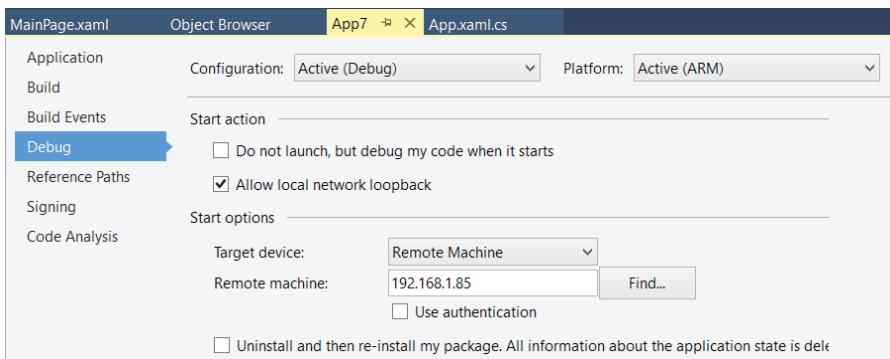
- Ви маєте використовувати карту microSD рівня 10.
- Перший запуск триває довго. Будьте терплячими!
- Ви повинні підключити Raspberry до локальної мережі для розгортання програм із машини Windows 10 . Тому потрібно підготувати мережевий кабель і маршрутизатор або адаптер Wi-Fi. Але зауважте, що образ підтримує тільки офіційний Wi-Fi адаптер Raspberry Pi (білий із зображенням малини).

Якщо все гаразд, ви побачите зображення Raspberry і важливу інформацію, таку як назва та IP-адреса пристроя. Отже, пристрій готовий до використання, і настав час встановити зв'язок між ПК і Raspberry. Для цього ви можете відвідати сайт <http://ms-iot.github.io/content/en-US/win10/samples/PowerShell.htm> і встановити з'єднання за допомогою PowerShell.

Після підключення до плати Raspberry ви можете змінити пароль, ім'я пристрою, виконати деякі команди налаштування тощо. Зауважте, що Raspberry підтримує два режими: **headed** і **headless** (з GUI і без нього).

Щойно буде встановлено з'єднання між комп'ютером і Raspberry, ви можете спробувати щось розробити. Завдяки Visual Studio розгорнати і налагоджувати рішення на Raspberry дуже просто. За замовчуванням Raspberry запускає віддалений налагоджувач, так що ви не повинні робити жодних додаткових налаштувань.

Для початку потрібно вибрати мову. Ви можете вибирати поміж Node.js, Python і стандартними мовами для універсальних програм, такими як C#. Звичайно, ми вирішили використовувати C#, але ви легко можете встановити інструменти для Python або Node.js із сайту Connect. Отже, для розгортання та налагодження спершу потрібно створити просту універсальну програму, змінити платформу на ARM і вибрати цільовий пристрій **Remote Machine** (Віддалена машина).

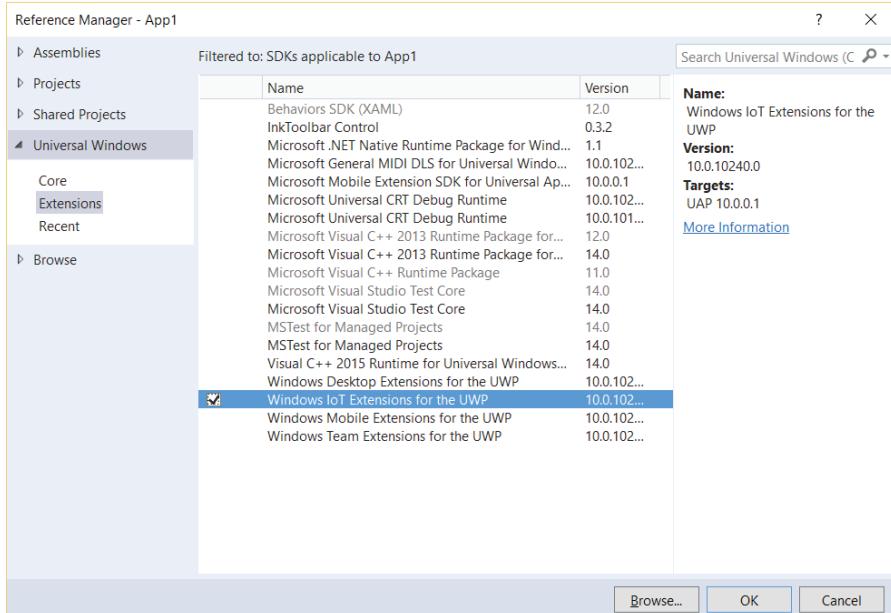


Нарешті ми готові розробляти й розгорнати свою першу програму для Raspberry.

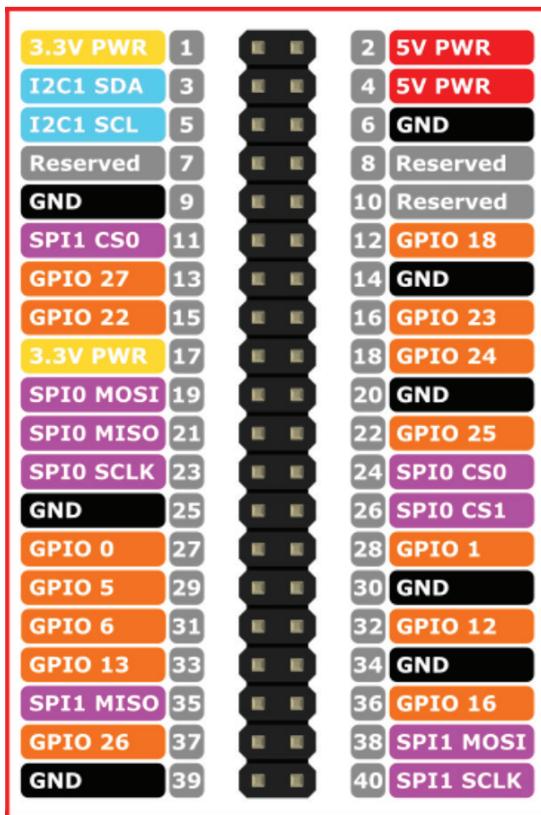
## Розширення IoT

Якщо ви вже встановили Windows 10 на Raspberry, настав час обговорити, як використовувати контакти і шину I2C. Звичайно, якщо ви збираєтесь розгорнути програму, що не працює з інтерфейсом GPIO, то не потрібно робити нічого

особливого – просто створіть за допомогою C# або інших мов таку саму універсальну програму, як для настільного ПК або телефону, і розгорніть її за допомогою ARM-платформи безпосередньо з Visual Studio. Однак інтерфейс GPIO залежить від особливостей пристроя, і немає жодного сенсу включати його в загальну бібліотеку. Так що, якщо ви збираєтесь використовувати деякі специфічні особливості IoT, потрібно додати до проекту розширення IoT.



Це розширення оголошує тільки два контракти: **Windows.Devices.DevicesLowLevelContract** і **Windows.System.SystemManagementContract**. Відповідні оголошення легко знайти в папці **C:\Program Files (x86)\Windows Kits\10\Extension SDKs\WindowsIoT\10.0.10240.0** у файлі **SDKManifest.xaml**. Контракт **SystemManagementContract** не містить нічого особливого – тільки класи, які допомагають вимкнути пристрій і змінити часовий пояс. Але **DevicesLowLevelContract** містить багато важливих класів. Давайте подивимось на Raspberry і побачимо, що ми можемо отримати за допомогою цих класів.



Працюючи з GPIO, можна використовувати жовті контакти для надсилання сигналів на датчики або отримання від них даних. На Raspberry всі контакти генерують напругу 3,3 В. Рожеві контакти підтримують протокол SPI, і ви можете підключити до них до двох пристройів/датчиків (можливо, і три, але ми такий експеримент не проводили). Нарешті, сині контакти дають змогу використовувати концентратор шини I2C, який підтримує більше ста пристройів/датчиків. Звичайно, контакти GND призначенні для заземлення, і є також 4 контакти для живлення (два по 5 В і два по 3,3 В).

Почнімо з GPIO. У просторі імен **Windows.Devices.Gpio** є необхідні класи (і перелічувані типи), що дають розробникам можливість керувати контактами GPIO на платі. Перший клас – це **GpioController**, який надає доступ до контролера і посилання на необхідні контакти. Якщо ви збираєтесь використовувати GPIO, то маєте почати з виклику методу **GetDefault**, який повертає посилання на поточний контролер GPIO (якщо такий є):

```
var controller=GpioController.GetDefault();
```

Якщо ви збираєтесь написати справді універсальний код, то потрібно перевірити, чи не має контролер значення **null**. Якщо це так, на вашому пристрої немає GPIO (наприклад, його немає на настільному ПК, телефоні і т. д.). Це дуже корисно, оскільки, починаючи з Windows 10, у нас є універсальна платформа, і ви можете запустити той самий двійковий файл на будь-якому пристрої.

Якщо у вас є доступ до контролера, можна спробувати отримати посилання на контакти. Метод **OpenPin** дає змогу отримати посилання на контакт, блокувати його для ексклюзивного використання або залишити в режимі спільного доступу. Звичайно, цей метод не допоможе вам надіслати на контакт сигнал і просто поверне посилання на об'єкт **GpioPin**. Так само і з **GpioController** – вам потрібно перевірити, чи повернуте посилання не дорівнює **null**, оскільки інші процеси можуть заблокувати контакт для використання з власною метою.

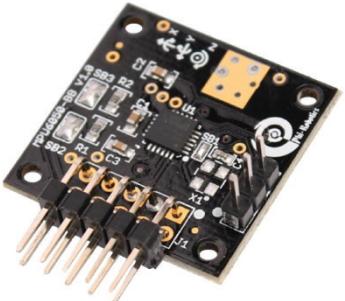
```
var pin = controller.OpenPin(0);

if (pin == null)
{
    return;
}
```

За допомогою методу **OpenPin** ви можете отримати номер контакту, і ви повинні використовувати ті самі номери контактів, що й на зображеннях вище. Нарешті, якщо ви отримали доступ до контакту, то можете спробувати працювати з ним. Є три важливі методи, такі як **SetDriveMode**, **Read** і **Write**. За допомогою першого можна зробити контакт вхідним або вихідним. Методи **Read** і **Write** дають змогу надсилати або отримувати сигнал.

Більш цікаві класи містяться в просторах імен **Windows.Devices.I2C** і **Windows.Devices.SPI**.

Для тих, хто хоче побудувати власний дрон чи гелікоптер, є сенсор MPU6050, який надсилає дані гіроскопа і акселерометра. На ринку є багато різних плат, що містять лише цей чіп 6050. Якщо ви зацікавились, можете придбати, наприклад, плату MPU6050-BB тут: <http://phi-education.com/store/MPU6050-BB> чи тут: <http://robotshop.ca>.



Цей датчик не вимагає пайки і містить перетворювач напруги з 5 В на 3,3 В (чіп 6050 живиться від напруги 3,3 В), що дає можливість використовувати живлення від плати ESC або інших плат із контактами живлення на 5 В.

Датчик MPU 6050 використовує для зв'язку концентратор шини I2C. Таким чином, щоб отримати готовий до роботи датчик, необхідно підключити контакти 5 В і GND на Raspberry до контактів VCC і GND на датчику. Крім того, ви повинні підключити контакти SDA і SCL. Цей датчик не має світлодіодів, і тому не легко зрозуміти, чи все працює справно. Найпростіший спосіб перевірити це – почати яку-небудь розробку.

Всі необхідні класи ви можете знайти в просторі імен **Windows.Devices.I2c** і першим із них є **I2cDevice**. Кожен пристрій, який підключається через концентратор I2C, має бути пов'язаний з об'єктом класу **I2cDevice**, і завдяки цьому об'єкту розробники можуть здійснювати взаємодію з пристроєм. Найуживаніші методи – це **Read** і **Write**, що працюють з масивом байтів для отримання або надсилання даних. Але в багатьох випадках вам потрібно надіслати дані на пристрій, щоб запитати яку-небудь інформацію і зчитати відповідь. Щоб уникнути виклику відразу двох методів, клас **I2CDevice** підтримує метод **ReadWrite**. Цей метод має два параметри, що є масивами байтів. Перший масив містить дані, які ви збираєтеся надіслати на пристрій, а другий – це буфер для отримання даних із пристрою.

Завдяки класу **I2cDevice** легко обмінюватися даними з пристроями, але для того, щоб отримати посилання на об'єкт **I2cDevice**, вам потрібно виконати кілька завдань.

Перш за все, потрібно отримати посилання на пристрій I2C на платі (не на сенсор, а на відповідні I2C-контакти). Microsoft застосовує той самий підхід, що і для всіх інших пристрій, таких як пристрії Bluetooth, Wi-Fi і т. д. Ви повинні використовувати зрозуміле ім'я, щоб створити рядок запиту для пристрію та

спробувати знайти пристрій на платі. Метод **GetDeviceSelector** класу **I2cDevice** дає змогу створити рядок запиту, і ви повинні використовувати в ньому I2C-сумісне ім'я. Щоб знайти інформацію про наявний пристрій, скористайтеся методом **FindAllAsync** класу **DeviceInformation**. Цей метод повертає інформацію про доступні пристрої I2C, і ви можете використовувати її для створення об'єкта класу **I2cDevice**. Наступний крок – створити рядок підключення для вашого датчика. Це легко зробити за допомогою класу **I2cConnectionString**, передавши адресу датчика до конструктора класу. Якщо у вас є інформація про I2C на платі і рядки підключення для зовнішнього пристрою/датчика, можна створити об'єкт **I2cDevice**, використовуючи метод **FromIdAsync**.

Таким чином, для MPU 6050 ми створили такий код:

```
class MPU6050
{
    //I2C address
    private const byte MPUAddress = 0xD2>>1;

    private I2cDevice mpu5060device;

    public async Task BeginAsync()
    {
        string advanced_query_syntax =
            I2cDevice.GetDeviceSelector("I2C1");
        DeviceInformationCollection
            device_information_collection =
                await DeviceInformation.FindAllAsync
                    (advanced_query_syntax);
        string deviceId = device_information_collection[0].Id;

        I2cConnectionSettings mpu_connection =
            new I2cConnectionSettings(MPUAddress);
        mpu_connection.BusSpeed = I2cBusSpeed.FastMode;
        mpu_connection.SharingMode = I2cSharingMode.Shared;

        mpu5060device = await I2cDevice.FromIdAsync
            (deviceId, mpu_connection);

        mpuInit();
    }
}
```

Метод **mpuInit** надсилає датчику початкові значення, і цей процес ми описемо нижче. Відповідно до документації, значення **MPUAddress** повинно дорівнювати 0xD2, але ми маємо прийняти лише 7 бітів цього значення, тому зсунули його на один біт вправо.

Щойно ми отримаємо об'єкт **I2cDevice**, можемо почати працювати з пристроєм. Це не так легко, тому що MPU 6050 має безліч реєстрів, і ви повинні розуміти призначення більшості з них. Крім того, ви маєте ініціалізувати датчик, щоб отримувати належним чином масштабовані значення тощо. Розгляньмо кілька реєстрів:

- 0x6B – керування живленням. Можна встановити різні параметри, які стосуються режиму живлення, але найважливішим бітом є сьомий. Завдяки цьому біту ви можете задати початковий стан датчика.
- 0x3B–0x40 – дані акселерометра. Є 6 байтів, які містять дані для осей x, y і z. Оскільки для подання даних щодо кожної вісі потрібно два байти, то маємо 6 байтів, а не 3. Таким чином, щоб сформувати результат, ви повинні використовувати перший байт як старший байт значення типу **short (int16)**, а другий – як молодший.
- 0x41–0x42 – два байти, які містять значення температури – старший і молодший.
- 0x43–0x48 – 6 байтів для даних гіроскопа (або акселерометра).

Отже, ви можете використовувати метод **mpuInit** для встановлення початкового стану сенсора. А скинути налаштування датчика можна, наприклад, такою командою:

```
mpu5060device.Write(new byte[] { 0x6B, 0x80 });
```

Щоб виміряти що-небудь, скористайтеся методом **WriteRead**. Покажемо для прикладу, як виміряти температуру. Ось відповідний код:

```
byte mpuRegRead(byte regAddr)
{
    byte[] data=new byte[1];

    mpu5060device.WriteRead(new byte[] { regAddr },data);

    return data[0];
}

public double mpuGetTemperature()
{
    double temperature;
```

```

    short calc = 0;
    byte []data = new byte[2];
    data[0] = mpuRegRead(MPU_REG_TEMP_OUT_H); //0x41
    data[1] = mpuRegRead(MPU_REG_TEMP_OUT_L); //0x42
    calc = (short) ((data[0] << 8) | data[1]);

    temperature = (calc / 340.0) + 36.53;
    return temperature;
}

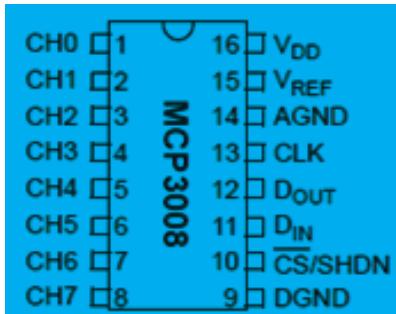
```

## Аналогові сигнали і PWM на Raspberry

На відміну від Arduino, Raspberry Pi 2 не має аналогових контактів і навіть PWM-контактів. В IoT-розширенні для Universal Windows Platform є три набори класів: I2C, SPI і GPIO. Останній дає можливість використовувати Raspberry GPIO для надсилання/приймання тільки високої або низької напруги. Тож, якщо ви бажаєте створити безпілотника або робота на основі Raspberry Pi 2 із Windows 10, слід відповісти на такі питання:

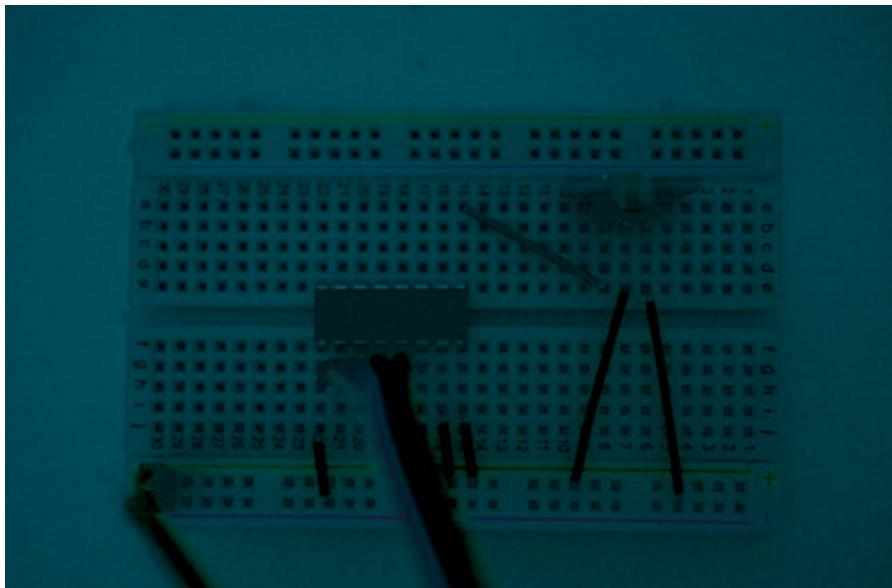
- як зчитувати аналогові сигнали;
- як емулювати PWM (широтно-імпульсна модуляція);
- як зчитувати цифрові сигнали від різних цифрових датчиків (якщо ці сигнали – не лише **LOW** або **HIGH**).

Є кілька способів вирішити ці завдання. Перший – використання зовнішніх конвертерів. Для аналогових входів ми рекомендуємо конвертори від Microchip Technology різноманітних модифікацій: MCP3002, MCP3008, MCP3208 і т. д. Наприклад, MCP3008 підтримує до 8 каналів і подає аналогові дані в 10-бітному форматі. Оскільки Arduino і Netduino використовують той самий формат (10 бітів, значення від 0 до 1024), то цей формат можна вважати найбільш природним.



MCP3008 працює на основі шини SPI, а отже, ми маємо використовувати шину SPI на Raspberry. Для цього потрібно підключити ніжку CLK чіпа до контакту SPI0 CSLK (19), ніжку D (вихід) – до контакту SPI0 MISO (21), ніжку D (вхід) – до SPI0 MOSI (23) і ніжку CS/SHDN – до SPI0 CS0 (24). Ніжки V (dd) і V (ref) підключено до живлення і DGND – до землі.

У нас є тільки аналоговий датчик-фоторезистор. Тому використано тільки канал 0 і підключено сигнальну ніжку датчика до CH0.



Нижче наведено код, який можна скопіювати і вставити у файл **MainPage.xaml.cs** вашої універсальної програми. Ми не створювали ніяких інтерфейсів і просто використали клас **Debug** для відображення даних датчика у вікні виводу:

```
byte[] readBuffer = new byte[3];
byte[] writeBuffer = new byte[3] { 0x06, 0x00, 0x00 };

private SpiDevice spi;

private DispatcherTimer timer;

private void Timer_Tick(object sender, object e)
{
    spi.TransferFullDuplex(writeBuffer, readBuffer);
    int result = readBuffer[1] & 0x07;
```

```
        result <= 8;
        result += readBuffer[2];
        result >= 1;
        Debug.WriteLine(result.ToString());
    }

protected async override void OnNavigatedTo
(NavigationEventArgs e)
{
    await StartSPI();
    this.timer = new DispatcherTimer();
    this.timer.Interval = TimeSpan.FromMilliseconds(500);
    this.timer.Tick += Timer_Tick;
    this.timer.Start();
    base.OnNavigatedTo(e);
}

private async Task StartSPI()
{
    try
    {
        var settings = new SpiConnectionSettings(0);
        settings.ClockFrequency = 5000000;
        settings.Mode = SpiMode.Mode0;

        string spiAqs = SpiDevice.GetDeviceSelector("SPI0");
        var deviceInfo = await DeviceInformation.
            FindAllAsync(spiAqs);
        spi = await SpiDevice.FromIdAsync(deviceInfo[0].Id,
            settings);
    }
    catch (Exception ex)
    {
        throw new Exception("SPI Initialization Failed", ex);
    }
}
```

Другий спосіб набагато більш радикальний – можна створити надбудову Arduino на основі Raspberry Pi 2. Завдяки цьому ви можете отримати PWM-, аналогові і цифрові входи.

Звичайно, плати Raspberry Pi і Arduino мають різні форм-фактори і плата Arduino не може бути приєднана до Raspberry Pi безпосередньо, тому ми вирішили побудувати свою власну плату Arduino «з нуля» як плату для Raspberry.

Відвідавши, наприклад, сайт <http://www.bc-robotics.com/>, можна придбати такі компоненти:

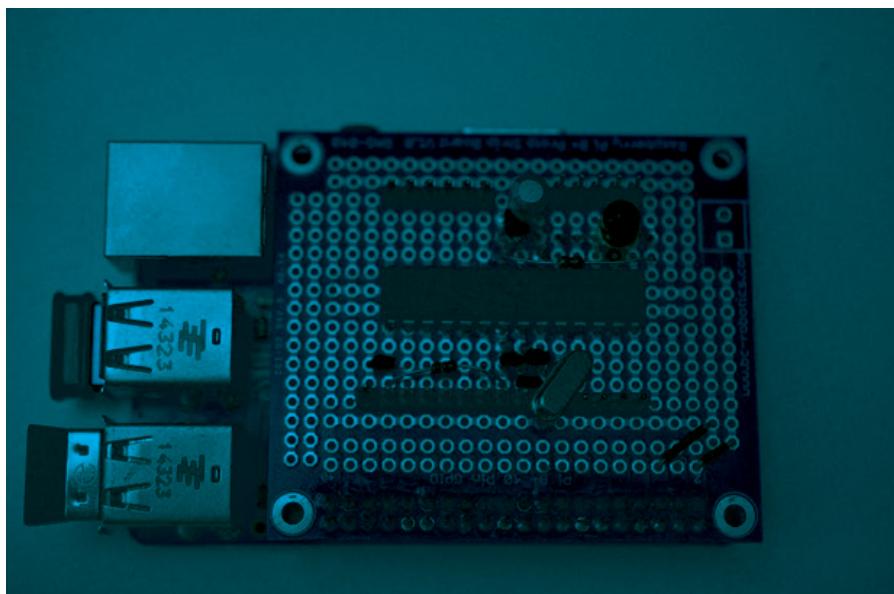
- Raspberry Proto Strip Board;
- Stacking Header;
- ATMega 328P-PU із завантажувачем (можна купити його без завантажувача, але тоді знадобиться ще один Arduino для флеш-завантажувача);
- кристал на 16Mhz;
- декілька конденсаторів ємністю 22 pF і 0.1 uF та один конденсатор на 10uF.

Крім того, необхідно переконатися, що у вас є різні резистори (принаймні, на 10 кОм і 220 Ом), принаймні один світлодіод, кнопка і конвертер з послідовного інтерфейсу на USB. Слід переконатися, що конвертер має DTR-контакт (а не лише RX і TX). Це все коштує не більше 20 доларів (на китайських сайтах – ще дешевше, але доведеться довго чекати).

На наступному кроці вам потрібно знайти схему для побудови власного Arduino «з нуля». Ми взяли схему з цього сайту: <http://shrimping.it/blog/shrimp/>, однак скористалися Protected Shrimp і з метою економії простору прибрали зі схеми кнопку.

Є тільки одна порада: зберіть усе на макеті, оскільки Raspberry Proto Strip Board має аналогічну архітектуру. Щойно ви перевірите схему, можна перемістити всі компоненти на плату.

За 30 хвилин можна отримати такий результат:



Тепер ви можете скористатися перетворювачем з USB на послідовний інтерфейс, щоб розгорнути все необхідне програмне забезпечення на своєму Arduino. Як це зробити, залежить від обраного методу підключення Arduino до Raspberry. Наприклад, можна використовувати ескіз **StandardFirmata**, який встановлюється з Arduino SDK, а також I2C- або послідовні підключення.

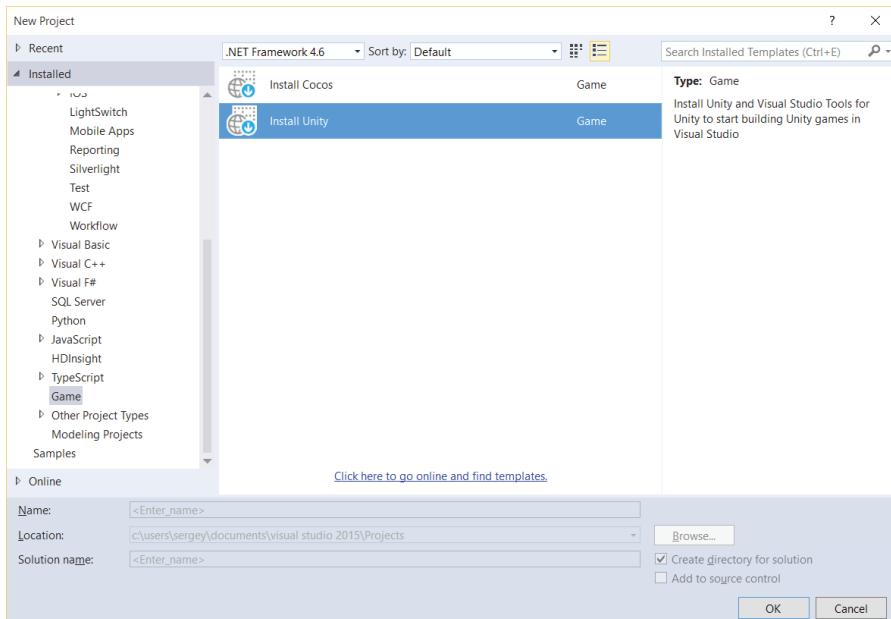
Розділ 35.

# **Як розробляти ігри для Windows 10 в Unity3D**

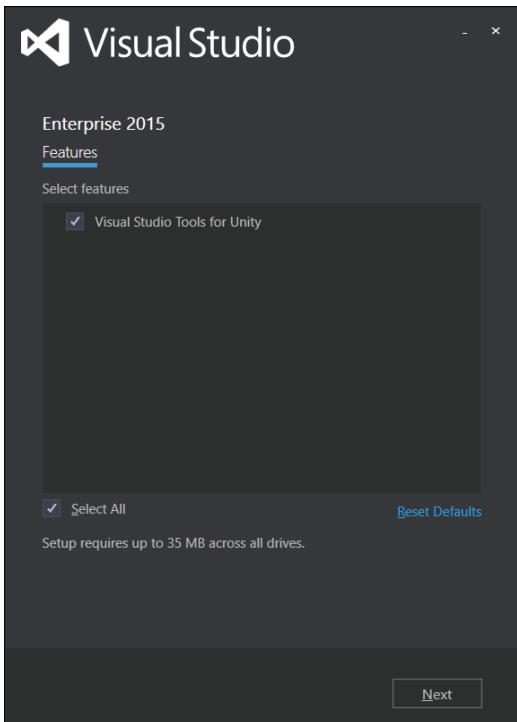
## Інструменти Visual Studio для Unity

Багато розробників використовують Visual Studio як стандартний редактор коду для Unity протягом багатьох років і, схоже, з часу випуску Visual Studio 2015 інтеграція між двома цими продуктами набагато посилилася. Так, із часу випуску Unity3D 5.2 у Visual Studio Community Edition 2015 є редактор коду Unity3D, призначений за замовчуванням для розробників на платформі Windows.

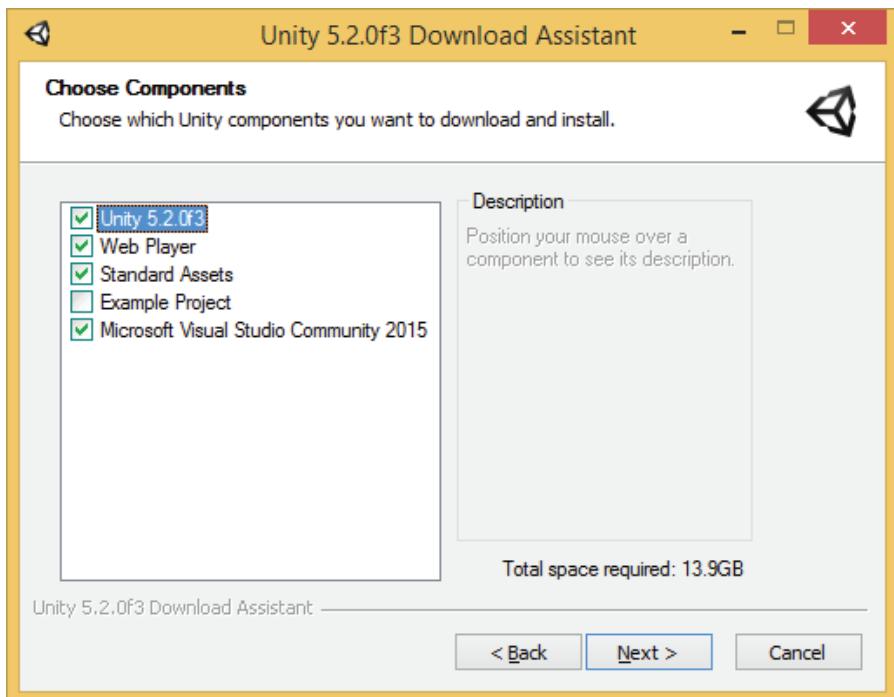
Обидва інструменти мають посилання на встановлення один одного. Якщо у вас новий ПК з Visual Studio 2015, ви можете виявити, що в діалоговому вікні **New Project** є нова категорія – **Game** (Гра). Її не можна використовувати для створення нових проектів, тому що вона лише містить посилання на деякі популярні ігрові фреймворки, включно з Unity.



Виберіть команду **Install Unity**, і Visual Studio допоможе вам інсталювати Unity та інструменти Visual Studio Tools для Unity:

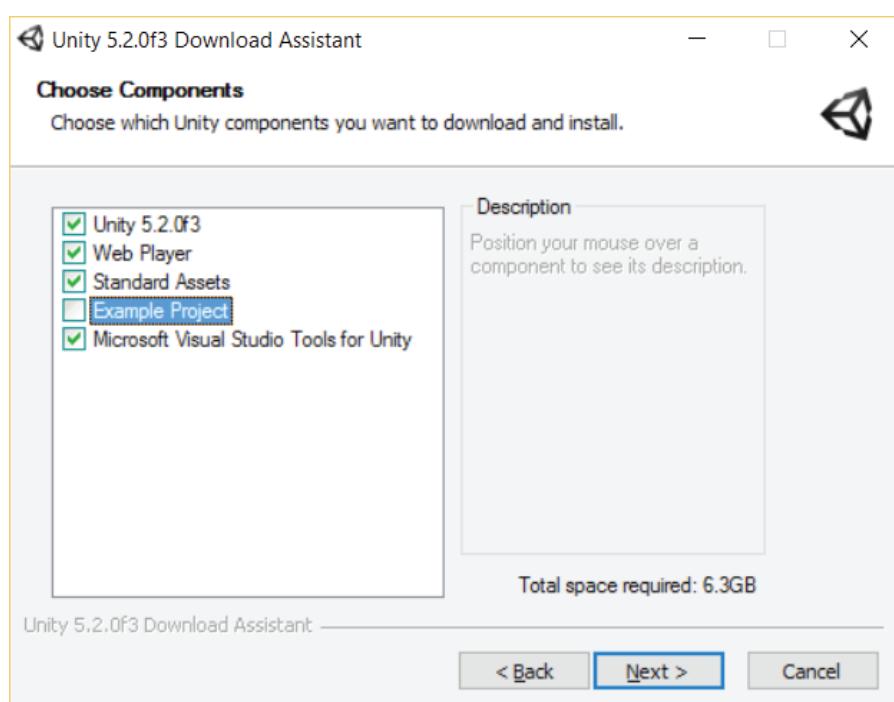


У разі використання інсталятора Unity інтеграція навіть ще вища: користувачі можуть обрати та встановити Visual Studio Community Edition прямо з інсталятора Unity:



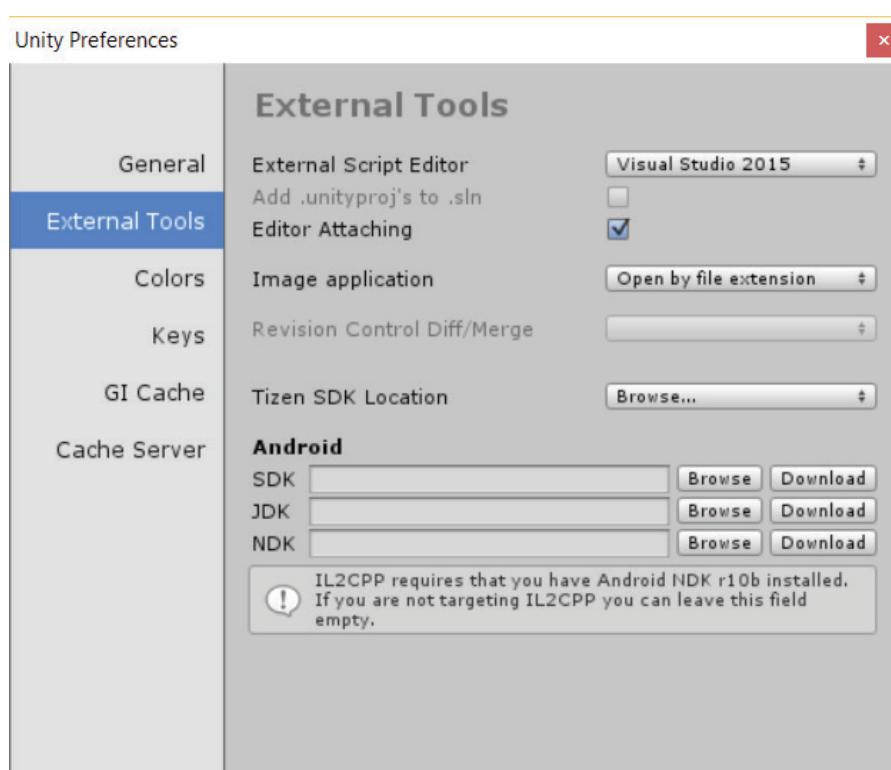
Зверніть особливу увагу на те, що, починаючи з випуску Visual Studio Community Edition, розробники ігор можуть використовувати всі важливі функції Visual Studio безкоштовно, включно з розширеннями (плагінами) та функціями налагодження. Якщо ви бажаєте скачати VS Community Edition окремо, відвідайте веб-сайт <https://www.visualstudio.com/>.

Середовище Visual Studio Community Edition має деякі обмеження щодо ліцензування. Наприклад, його можна використовувати лише для невеликих груп (до 5 осіб), але якщо ви працюєте у великий компанії, можна обрати професійний або корпоративний варіант Visual Studio. У цьому разі інсталятор Unity розпізнає наявну версію Visual Studio і запропонує встановити тільки ті інструменти Visual Studio Tools для Unity, які потрібні для інтеграції Unity та Visual Studio 2015:

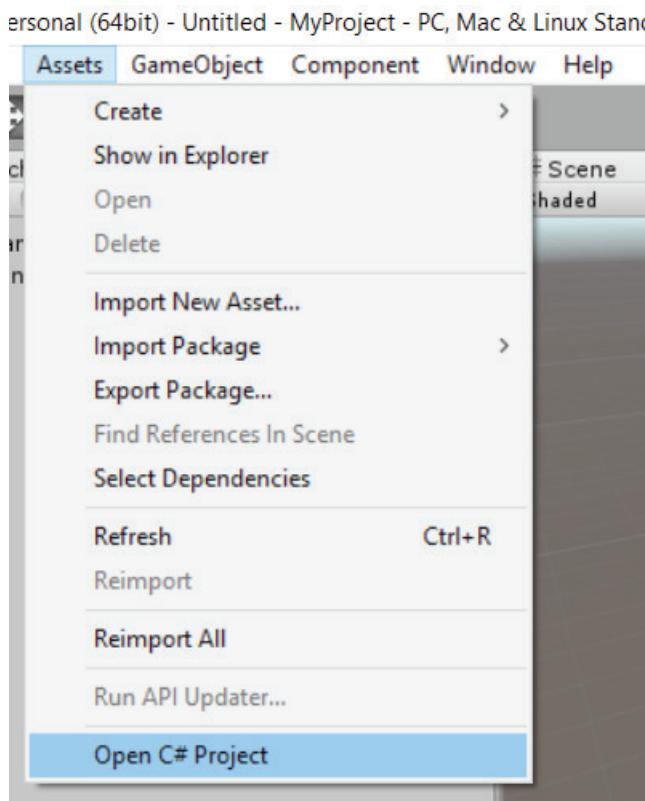


Використовуючи інсталятор Unity чи Visual Studio, справді важко пропустити можливість встановлення інструментів Microsoft Visual Studio Tools для Unity. Розгляньмо, як використовувати ці інструменти.

Версія Unity3D 5.2 має вбудовану підтримку інструментів Visual Studio. Таким чином, додавати жодні пакети не потрібно. Варто лише створити новий проект або відкрити наявний. Щоб переконатися, що Visual Studio є редактором за замовчуванням, виберіть пункти меню **Edit > Preferences** і відкрийте налаштування **Unity**, що містить інформацію про зовнішні інструменти, включно з редактором сценаріїв:

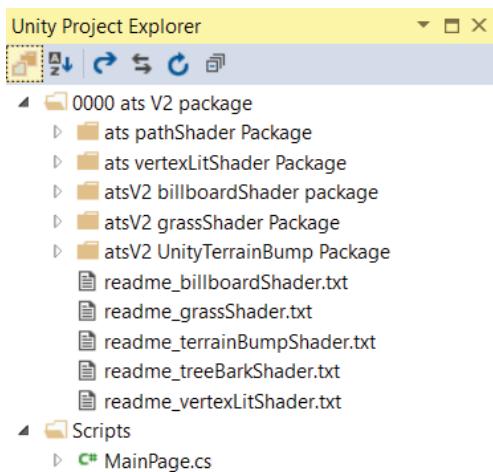


На цьому етапі ви можете почати роботу з вашим проектом Unity, створювати сценарії на C#, об'єкти тощо. Щоб відкрити проект у Visual Studio, просто виберіть пункт меню **Open in C#**.



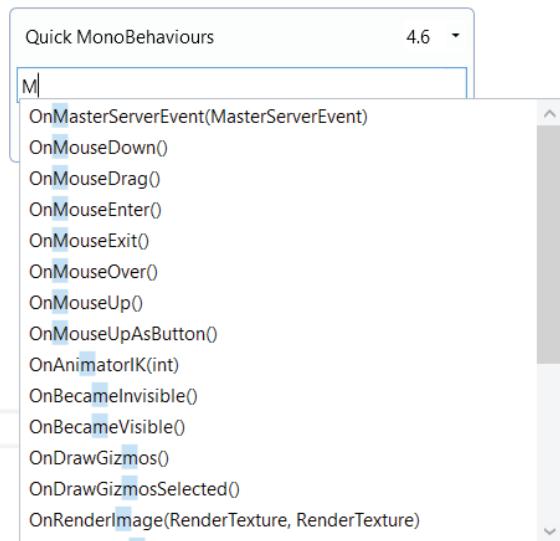
Після цього Unity відкриє ваш проект у Visual Studio. Розглянемо деякі можливості, які можна використовувати у Visual Studio.

Перш за все, можна скористатися Unity Project Explorer (**Shift+Alt+E**):

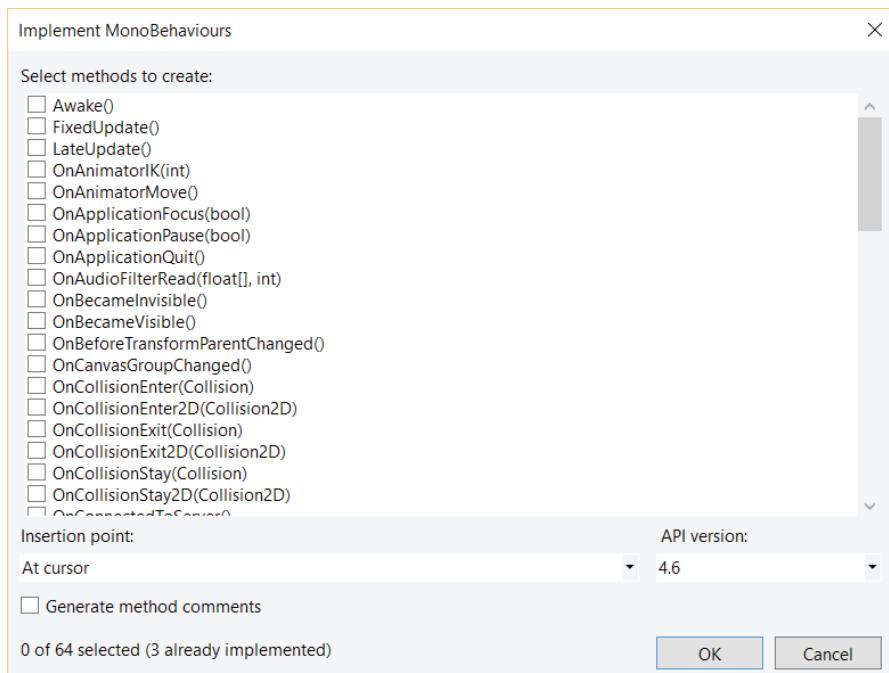


Це вікно схоже на вікно **Project** в Unity і може так само відображати файли проекту. Отже якщо вам потрібно знайти певні файли дуже швидко, можна скористатися тим самим способом, що й у Unity. Unity Project Explorer і Solution Explorer показують файли проекту різними способами. Особливо це стосується великих проектів.

Наступні два вікна дають можливість швидко перевизначити методи класу **MonoBehaviour**. Щоб відкрити вікно **Quick MonoBehaviour**, натисніть комбінацію клавіш **Ctrl+Shift+Q**. Просто почніть вводити назву методу, і у з'явиться спливаючий список назв:



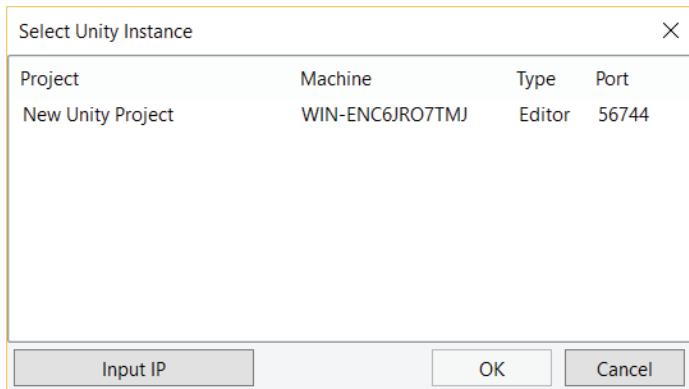
Наступне вікно ви можете відкрити, натиснувши комбінацію **Ctrl+Shift+M**:



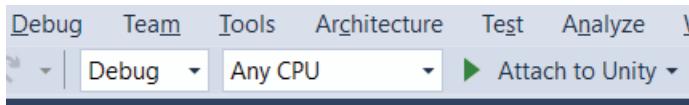
Використання майстра **MonoBehaviour** дає змогу одночасно створювати кілька методів-заглушок.

Серед функцій Visual Studio для Unity виокремлюють ще дві – підтримка шейдерів редактування та інтеграція з виводом Unity. Завдяки першій функції можна побачити розфарбування залежно від синтаксису та форматування, якщо працювати із шейдерами у Visual Studio. Друга функція дає змогу побачити помилки та попередження Unity у вікні помилок Visual Studio.

І, нарешті, найважливіша функція – налагодження. Ви можете підключитися до налагоджувача Unity, вибравши команду **Debug > Attach Unity Debugger**. У вікні **Select Unity Instance** відобразяться всі доступні екземпляри Unity, і можна буде вибрати будь-який із них:



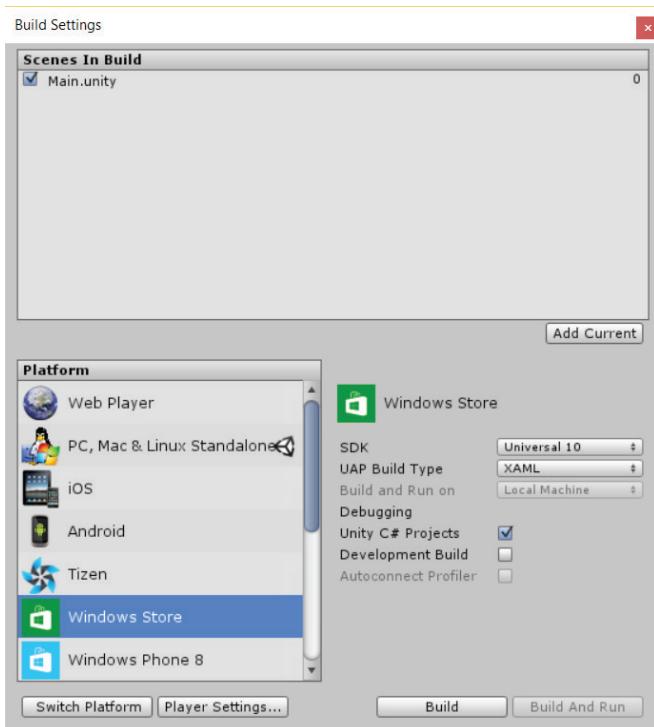
Або можна просто натиснути кнопку **Attach to Unity** на стандартній панелі інструментів.



Після приєднання Visual Studio відкрийте редактор Unity і використовуйте функції **Play/Stop**. Звичайно, Visual Studio підтримує точки зупину, дає можливість обчислювати вирази та змінні й застосовувати інші функції налагодження.

## Публікація гри Unity в Магазині Windows

Якщо ви готові опублікувати вашу гру в Магазині, почніть із вікна **Build Settings** (Налаштування збірки) у Unity:



Для програм на платформі Windows 10 потрібно вибрати Universal 10 в якості SDK, а також тип збірки поміж C # і DirectX. Тип проекту підказує, які типи хост-проектів будуть використовуватися. Ми будемо використовувати XAML/C#, але можна також вибрати DirectX (D3D).

Наступний важливий параметр – **Debugging Unity C# Project**. Якщо ви не оберете його, Unity скомпілює весь код і створить новий хост-проект, який посилається на бібліотеки DLL із вашою грою. У цьому разі код гри у Visual Studio буде неможливо налагодити. Але якщо ви оберете цей параметр, Unity включить ще один проект із вихідним кодом вашої гри, і це даст змогу налагоджувати його.

Звичайно, деякі розробники можуть запитати, чому ми повинні налагоджувати код фінального проекту, особливо якщо користуємося інструментами Visual Studio для Unity і вже перевірили і налагодили гру в редакторі Unity. Основна проблема полягає в тому, що редактор Unity використовує базу Mono для запуску гри в програвачі. Але після того, як ви створили остаточне рішення для Магазину і плануєте створити та розгорнути пакет на платформі Windows 10, Visual Studio буде використовувати .NET Framework для версії налагодження і .NET Core для

релізу. Ось чому в деяких випадках гру потрібно налаштовувати, уникаючи деяких просторів імен, таких як **System.IO**, **System.Net**, **System.Reflection** тощо. Інколи можна використовувати бібліотеку **WinRTLegacy** як обгортку для старої функціональності, а часом потрібно змінювати класи безпосередньо (наприклад, використовувати **Windows.Storage** замість **System.IO**).

Отже, можливості налагодження дуже важливі для остаточного рішення, тому що потенційні проблеми містяться й у реальних програмах на платформі Windows 10, а не лише в програвачі Unity.

Перш ніж натиснути кнопку **Build**, рекомендуємо вибрати **Player Settings** і переглянути налаштування для програми щодо Магазину Windows.



У цьому вікні ви можете додати багато властивостей у файл маніфесту програми. Зазвичай, майже всі налаштування можна додати у Visual Studio. Таким чином, можна обрати оптимальний варіант.

Після того, як ви натиснете кнопку **Build**, Unity створить остаточне рішення для Магазину і включить в себе два або три проекти (залежно від варіанту налагодження): бібліотеки, вихідний код і сам проект для Магазину. Крім того, Unity створить три конфігурації збірки: **Debug**, **Release** і **Master**. У разі вибору конфігурацій **Debug** і **Release** Visual Studio буде використовувати .NET CLR, щоб побудувати й запустити програму, але для налагодження програми вам потрібна конфігурація **Master**. Ця конфігурація підтримує .NET Native і дає можливість публікації в Магазині.

Отже, на цьому кроці слід лише переконатися, що програма успішно працює за межами програвача Unity. Але якщо у вас виникли проблеми з кодом через міграцію з Mono на .NET, слід просто знайти та використати класи з Windows Runtime.

На наступному кроці потрібно додати функціональність, специфічну для Windows 10. Наприклад, можна використовувати динамічні плитки, сповіщення тощо. Під час налагодження ігор це не повинно зайняти багато часу, але планувати такі кроки потрібно заздалегідь.

Завершивши налагодження, ви можете перевірити продуктивність, відтестувати програму і опублікувати її в Магазині.

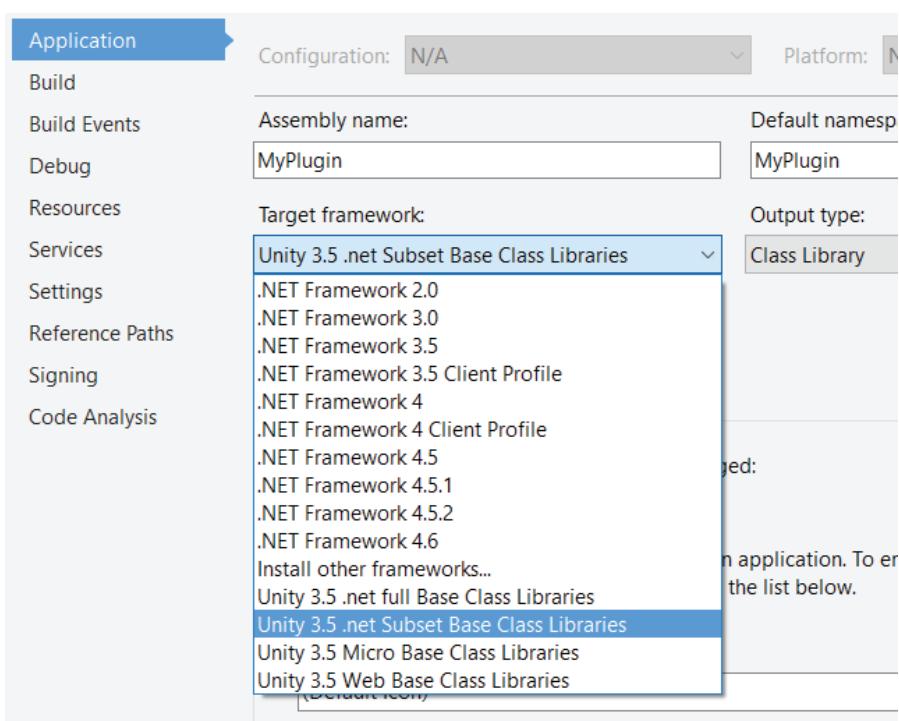
## Як створювати плагіни для Unity

Як правило, підготувати пакет для Магазину легко, важче вирішити проблеми з плагінами. Якщо у вас є двійкові файли, які підтримують Магазин Windows – можна просто використовувати їх.

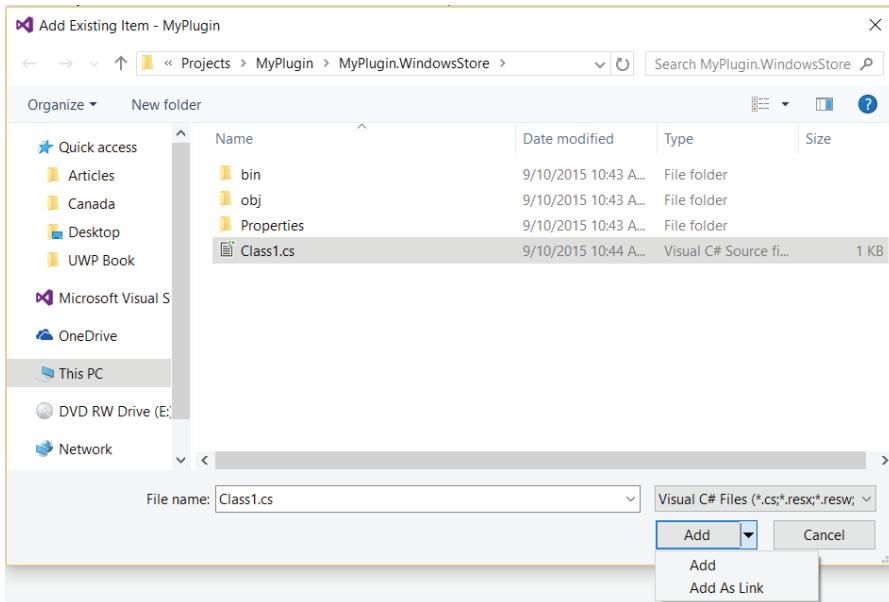
Але якщо ви не маєте доступу до вихідного коду плагіна і його не підтримують Магазин, це становить проблему. Ви можете попросити розробника включити підтримку плагіна в Магазин Windows або використовувати інший плагін.

Нарешті, ви можете створити повністю новий плагін або використовувати існуючий вихідний код для його міграції на UWP. Це дуже поширенна задача, і ми приділимо їй увагу.

Насамперед необхідно створити нове рішення, яке буде містити принаймні проекти Windows Runtime Component і Classic Class Library. Перший проект Unity використає для того, щоб створити рішення UWP, а другий – у Unity Player. Зверніть увагу на те, що проект Classic Class Library за замовчуванням використовує .Net Framework 4.6, так що вам потрібно відкрити налаштування проекту та змінити фреймворк на Unity 3.5 .Net Base Class Library:



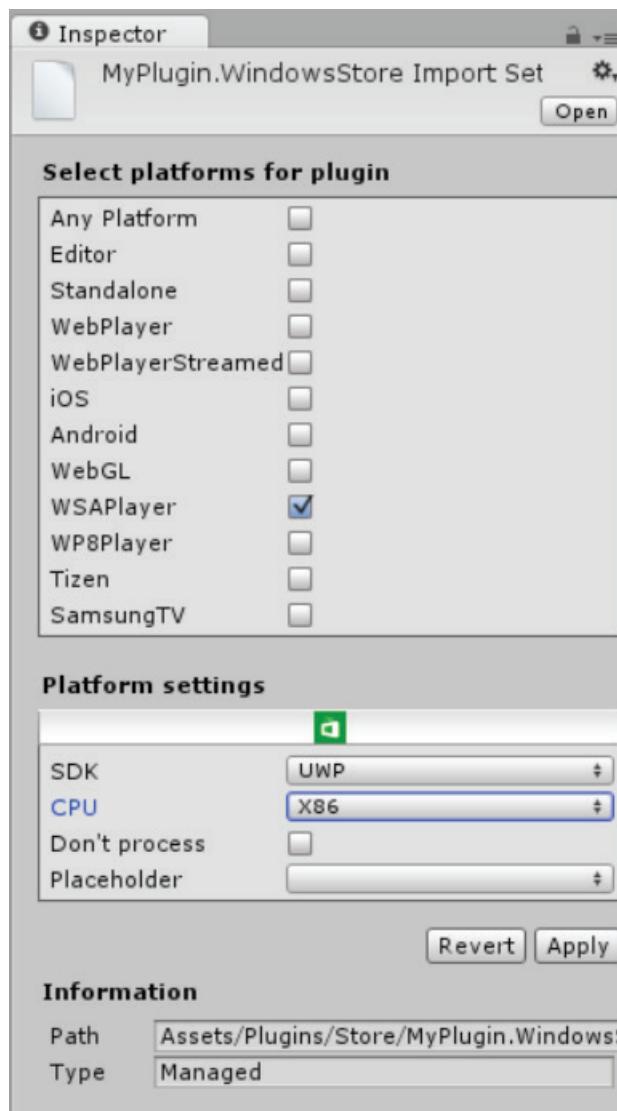
Не дублюйте наявний код (або новий код) в обох проектах. Просто додайте вихідні файли в одному із цих проектів і використайте функцію Add As Link, щоб додати посилання:



Завдяки цьому ви можете працювати з тим самим набором файлів для обох проектів.

Звичайно, за таких умов слід використовувати деякі директиви препроцесора, щоб активувати умовну компіляцію. У нашому випадку використовуємо **UNITY\_EDITOR** і **UNITY\_WSA** як директиви для редактора і програм Магазину Windows. Перейдіть на вкладку **Build** у властивостях проекту і додайте відповідний символ **Conditional compilation symbol**.

Готовий код можна скомпілювати та додати всі бібліотеки проекту Unity. У попередній версії вам потрібно було розмістити бібліотеку в спеціальних папках, але з Unity 5.2 ви його можна розмістити в будь-якому місці. Натисніть на кожну бібліотеку і вкажіть платформу на вкладці **Inspector**:



Процес завершено. Поза нашою увагою залишилося багато питань (інструменти діагностики, зв'язки між мостами Магазину Windows та іграми тощо), але наданої інформації достатньо для того, щоб почати роботу.