Взаємодія з додатком за допомогою подій

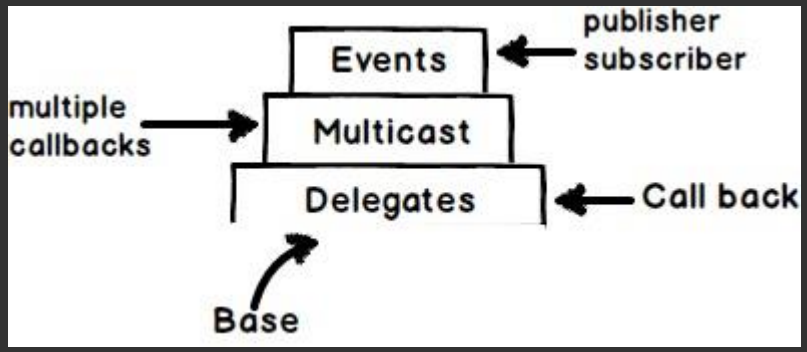
Питання 3.3.

Події та елементи управління

- Для елементів управління в WPF визначено багато подій, які умовно можна поділити на кілька груп:
  - Події клавіатури
  - Події миші
  - Події стилуса
  - Події сенсорного екрана/мультитач
  - Події життєвого циклу

Делегати як основа для подій

- Делегати використовуються для інкапсуляції та передачі методів у якості параметрів для інших методів.
  - Інкапсулюватись може іменований або анонімний метод.
  - Тип-делегат визначає сигнатуру методу, а при створенні екземпляру делегата можна associate з будь-яким методом з такою ж сигнатурою.
  - Події інкапсулюють делегат та реалізують модель publisher-subscriber.



Процес створення делегата

- Делегат повинен створюватись за допомогою методу або лямбда-виразу, що мають сумісні вхідні параметри та тип повернення.
  - Делегати, що були сконструйовані з використанням іменованого методу, можуть інкапсулювати або статичний метод, або метод екземпляра.
  - Для використання з анонімними методами делегат і пов'язаний з ним код повинні оголошуватись разом.
- Три кроки при створенні делегата:
  1. Оголосити делегат: **delegate void Del(int x);**
  2. Визначити (іменований) метод: **void DoWork(int k) { /* ... */ }**
  3. Створити екземпляр делегата, використовуючи метод в якості параметру: **Del d = obj.DoWork;**

Приклад оголошення та використання делегата

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass {
    static void Main() {
        MathClass m = new MathClass();
        Del d = m.MultiplyNumbers; // Delegate instantiation using "MultiplyNumbers"

        // Invoke the delegate object.
        System.Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++) { d(i, 2);  }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n) {
        System.Console.Write(m * n + " ");
    }
}
```

Делегат, зіставлений одночасно зі статичним методом та методом екземпляра, повертає інформацію від кожного з них

```
// Declare a delegate
delegate void Del();

class SampleClass {
    public void InstanceMethod() {
        System.Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod() {
        System.Console.WriteLine("A message from the static method.");
    }
}
class TestSampleClass {
    static void Main() {
        SampleClass sc = new SampleClass();
        Del d = sc.InstanceMethod; // Map the delegate to the instance method:
        d();
        d = SampleClass.StaticMethod; // Map to the static method:
        d();
    }
}
```

Делегати та анонімні методи

- Створюючи анонімні методи, фактично передається блок коду в якості параметра делегата.
- Приклади:
  - // Create a handler for a click event.
    button1.Click += delegate(System.Object o, System.EventArgs e)
    { System.Windows.Forms.MessageBox.Show("Click!"); };
  - // Create a delegate.
    delegate void Del(int x); // Instantiate the delegate using an anonymous method.
    Del d = delegate(int k) { /* ... */ };

Багатоадресні (Multicast) делегати

- Об'єкти делегатів дозволяють за допомогою оператора + присвоювати кілька об'єктів одному делегату.
  - Багатоадресний делегат містить список присвоєних йому делегатів, які почергово викликаються.
  - Об'єднувати можна тільки делегати одного типу.
  - За допомогою оператора «-» можна видалити делегат зі складу багатоадресного делегату.
  - Багатоадресні делегати повинні мітити тільки методи, які повертають void, інакше виникне run-time exception.

Приклад роботи з багатоадресним делегатом

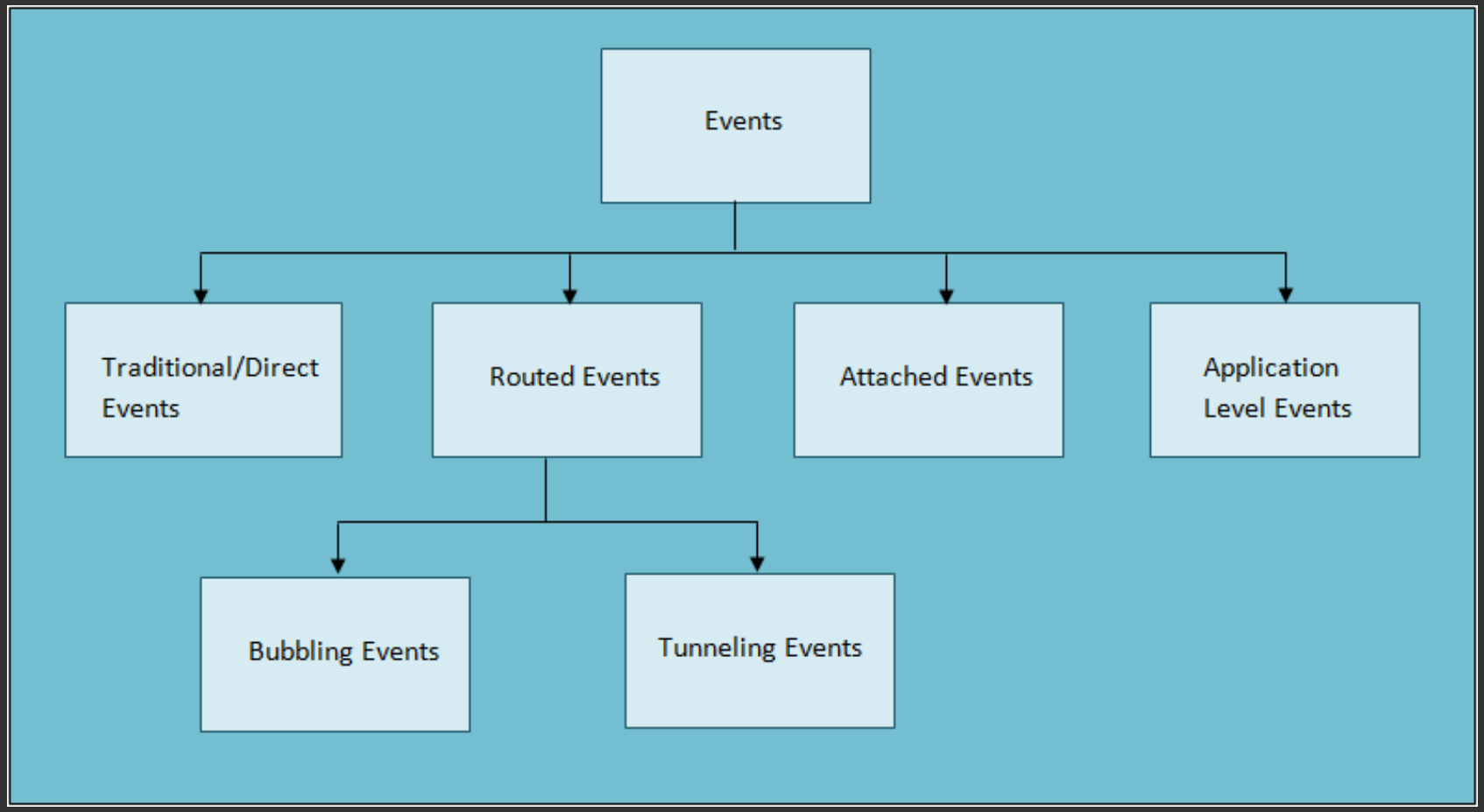
```
using System;
delegate void Mdelegate(int x, int y);

public class Oper {
    public static void Add(int x, int y) {
        Console.WriteLine("{0} + {1} = {2}", x, y, x + y);
    }

    public static void Sub(int x, int y) {
        Console.WriteLine("{0} - {1} = {2}", x, y, x - y);
    }
}

public class CSharpApp {
    static void Main() {
        Mdelegate del = new Mdelegate(Oper.Add);

        del += new Mdelegate(Oper.Sub);
        del(6, 4);
        del -= new Mdelegate(Oper.Sub);
        del(2, 8);
    }
}
```

```
6 + 4 = 10
6 - 4 = 2
2 + 8 = 10
```

Області застосування делегатів

- Делегати використовуються у наступних випадках:
  - Обробка подій (Event handlers)
  - Функції/методи зворотного виклику (Callbacks)
  - LINQ
  - Реалізація шаблонів проектування
- У делегатів немає специфічних можливостей, які неможливо реалізувати за допомогою звичайних методів.
  - Делегати дозволяють методам передаватись в якості параметрів.
  - Як і інтерфейси, делегати дозволяють decouple та узагальнювати код.
  - За потреби вирішити, який метод викликати під час виконання, використовується делегат.
  - Делегати забезпечують спосіб спеціалізації поведінки класу без його субкласування.

Види подій у WPF-додатках



Традиційні (прямі) події

- Існують ще з часів windows forms.
  - Виникають тільки від елемента управління, де були згенеровані.

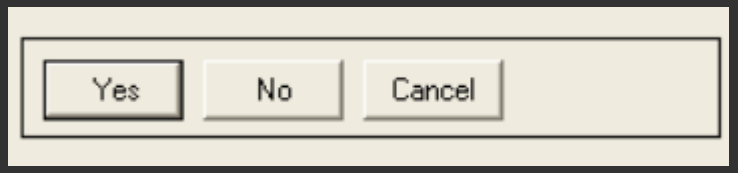
Top-level сценарії для маршрутизованих подій

- **Управління композицією та інкапсуляцією:**
  - Різні елементи управління в WPF мають rich content model.
  - Проте подія прив'язана до елемента, навіть якщо користувач взаємодіяв з його контентом.
- **Точки прикріплення Singular handler:**
  - У Windows Forms потрібно було прикріпляти (attach) один обробник багато разів, щоб обробляти події, згенеровані кількома елементами.
  - Маршрутизовані події дозволяють прикріпляти обробник лише раз та використовувати його логіку за необхідності там, де виникла подія.

Приклад спільного обробника події (xaml-частина)

```
<Border Height="50" Width="300" BorderBrush="Gray" BorderThickness="1">
  <StackPanel Background="LightGray" Orientation="Horizontal"
Button.Click="CommonClickHandler">
    <Button Name="YesButton" Width="Auto" >Yes</Button>
    <Button Name="NoButton" Width="Auto" >No</Button>
    <Button Name="CancelButton" Width="Auto" >Cancel</Button>
  </StackPanel>
</Border>
```

- Три кнопки



Приклад спільного обробника події (C#-частина)

```
private void CommonClickHandler(object sender, RoutedEventArgs e)
{
  FrameworkElement feSource = e.Source as FrameworkElement;
  switch (feSource.Name)
  {
    case "YesButton":  // do something here ...
      break;
    case "NoButton":  // do something ...
      break;
    case "CancelButton":  // do something ...
      break;
  }
  e.Handled=true;
}
```

Top-level сценарії для маршрутизованих подій

- **Обробка класу (Class handling):**
  - Маршрутизовані події дозволяють визначений у класі статичний обробник.
  - Він має можливість обробити подію до того, як це зможуть зробити прикріплені до екземпляру класу обробники.
- **Посилання на подію без рефлексії:**
  - Деякі коди розмітки та бізнес-логіки вимагають мати можливість ідентифікувати конкретну подію.
  - Маршрутизована подія створює поле типу RoutedEvent в якості ідентифікатора.

Реалізація маршрутизованих подій

- Маршрутизована подія зареєстрована в системі подій WPF.
  - Екземпляр RoutedEvent, отриманий після реєстрації, зазвичай зберігається як public static readonly поле класу, що реєструє (та володіє) маршрутизованою подією.
  - Підключення до такої ж за назвою CLR-події здійснюється шляхом переозначення реалізацій властивостей залежності add та remove CLR-події.

```
public static readonly RoutedEvent TapEvent =
        EventManager.RegisterRoutedEvent( "Tap",      RoutingStrategy.Bubble,
        typeof(RoutedEventHandler),          typeof(MyButtonSimple));

// Provide CLR accessors for the event
public event RoutedEventHandler Tap
{
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

Приклад: Click event handler

- RoutedEventArgs передається другим аргументом.
- Цей об'єкт містить аргументи події:
  - Handled – булеве значення, яке вказує, чи оброблюється подія. Якщо e.Handled = True, то подія оброблена та не буде виникати для подальших у візуальному дереві елементів управління.
  - Source – елемент управління, який генерує подію.
  - OriginalSource – первинний елемент управління, який згенерував подію. Зазвичай, той же, що і Source за винятком деяких складених елементів управління.
  - RoutedEvent – додаткові деталі, що стосуються події.

Причини використання маршрутизованих подій

- ***Можливі сценарії:*** визначення спільних обробників для елементів зі спільним коренем, компонування власних елементів управління або визначення власних класів, що описують елементи управління.
  - Будь-який об'єкт UIElement або ContentElement може бути слухачем (event listener) для будь-якої маршрутизованої події.

- Маршрутизовані події також можуть використовуватись для комунікації по дереву елементів, оскільки дані від події зберігаються для кожного елемента з маршруту.
  - Один елемент може щось змінити в цих даних, причому ця зміна буде видною наступному елементу по маршруту.

Причини використання маршрутизованих подій

- При реалізації власних подій можна розглядати наступні принципи:
  - Деякі можливості стилізації та шаблонізації WPF, як EventSetter та EventTrigger, вимагають, щоб подія, на яку посилаються, була маршрутизованою.
  - Маршрутизовані події підтримують class handling mechanism: клас може описувати статичні методи, які зможуть обробляти маршрутизовані події до того, як до них отримають доступ зареєстровані instance handlers.
  - Це дозволяє забезпечити подійно-орієнтовані поведінки класу, які буде неможливо випадково придушити (suppress) обробником екземеляру.

Маршрутизовані події і XAML

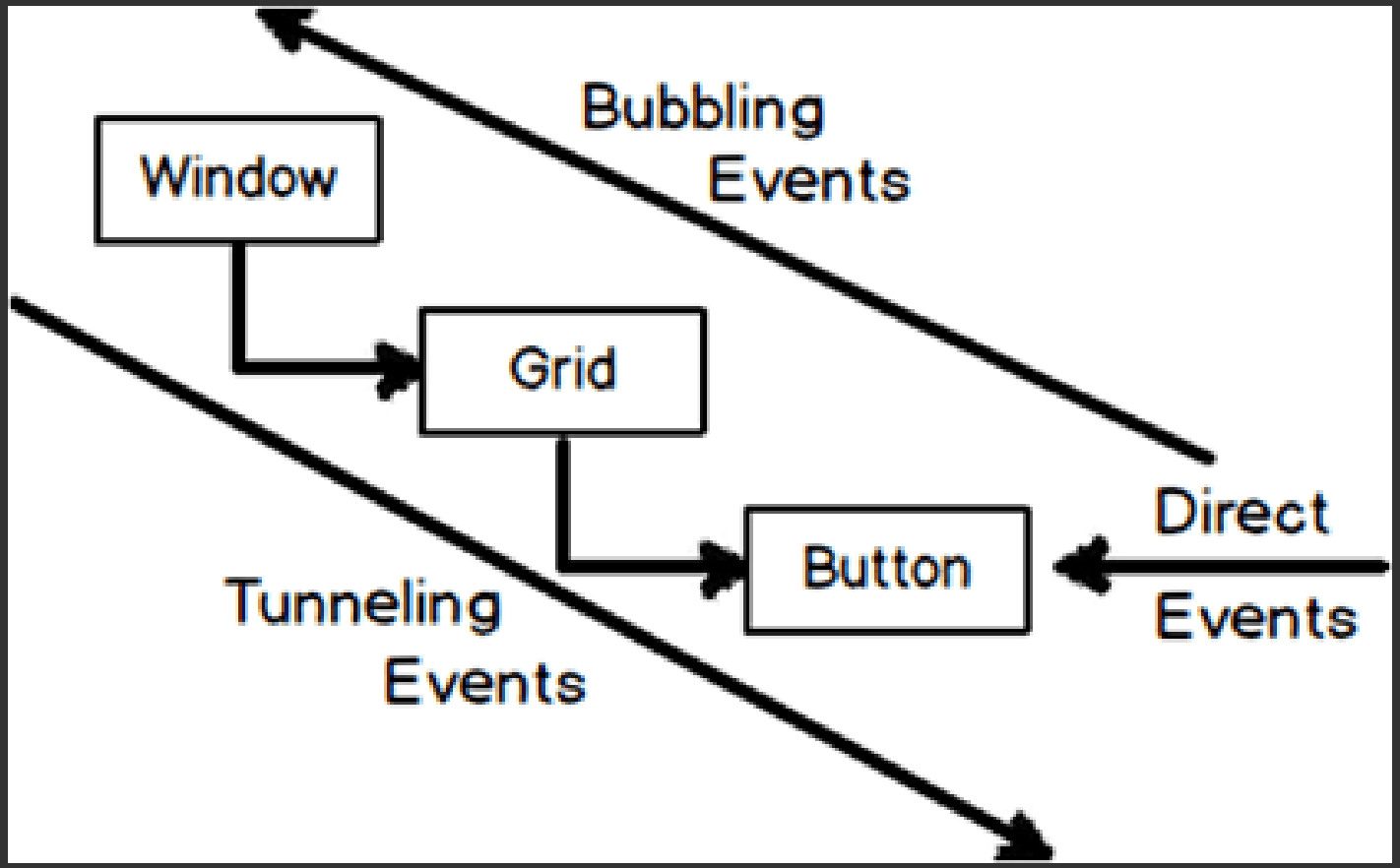
```
<Button Click="b1SetColor">button</Button>


void b1SetColor(object sender, RoutedEventArgs args)
{
  //logic to handle the Click event
}
```

- Синтаксис XAML для додавання стандартних обробників CLR-подій аналогічний до обробки маршрутизованої події, оскільки насправді обробники додаються до обгортки CLR-події, яка містить реалізовану маршрутизацію всередині.

Стратегії маршрутизації

- **Bubbling:** обробники подій викликаються в джерелі події.
  - Маршрутизована подія прямує до кореня дерева через батьківські елементи.
  - Стратегія використовується більшістю маршрутизованих подій.
  - Bubbling routed events загалом використовуються для повідомлення про зміни стану чи вводу для окремих елементів інтерфейсу користувача.
- **Direct:** тільки сам source element може викликати обробник у відповідь на подію.
  - Аналогічно до «маршрутизації" подій у Windows Forms.
  - На відміну від стандартних CLR-подій, direct routed events підтримують class handling та можуть використовуватись EventSetter та EventTrigger.
- **Tunneling:** спочатку обробники подій викликаються в корені element tree.
  - Маршрут події від кореня по дочірніх елементах до елементу, що є routed event source (the element that raised the routed event).
  - Часто використовуються або обробляються як складова частина елементу управління.
  - Такі події можуть навмисно придушуватись або замінятись подіями елементу управління загалом.
  - Події вводу в WPF часто реалізуються як пара tunneling/bubbling.

Стратегії маршрутизації



Обробка маршрутизованої події

- RoutedEventHandler - делегат для базового обробника маршрутизованої події.
  - Для спеціалізованих під елемент управління чи сценарій маршрутизованих подій делегати, що лежать в основі обробника, також стають більш спеціалізованими для передачі відповідних даних події.
  - Наприклад, при поширеному сценарії вводу може оброблятись маршрутизована подія DragEnter: обробнику слід реалізувати делегат DragEventHandler.
  - Використовуючи найбільш конкретний делегат, можна обробити DragEventArgs та зчитати властивість Data, яка містить корисне навантаження буферу обміну (clipboard payload) операції перетягування (drag).

Приклад обробки маршрутизованої події

```
<StackPanel Name="dpanel" Button.Click="HandleClick" >
        <StackPanel.Resources>
                <Style TargetType="{x:Type Button}">
                        <Setter Property="Height" Value="20"/>
                        <Setter Property="Width" Value="250"/>
                        <Setter Property="HorizontalAlignment"
                                                Value="Left"/>
                </Style>
        </StackPanel.Resources>
        <Button Name="Button1">Item 1</Button>
        <Button Name="Button2">Item 2</Button>
        <TextBlock Name="results"/>
</StackPanel>
```

```
public partial class RoutedEventHandle : StackPanel {
        StringBuilder eventstr = new StringBuilder();
        void HandleClick(object sender, RoutedEventArgs args) {
                // Get the element that handled the event.
                FrameworkElement fe = (FrameworkElement)sender;
                eventstr.Append("Event handled by element named ");
                eventstr.Append(fe.Name); eventstr.Append("\n");
                // Get the element that raised the event.
                FrameworkElement fe2 = (FrameworkElement)args.Source;
                eventstr.Append("Event originated from source element
                of type ");
                eventstr.Append(args.Source.GetType().ToString());
                eventstr.Append(" with Name ");
                eventstr.Append(fe2.Name);
                eventstr.Append("\n");
                // Get the routing strategy.
                eventstr.Append("Event used routing strategy ");
                eventstr.Append(args.RoutedEvent.RoutingStrategy);
                eventstr.Append("\n");
                results.Text = eventstr.ToString();
        }
}
```

- Обробники маршрутизованих подій також можуть додаватись з використанням допоміжного методу AddHandler().
  - Проте існуючі маршрутизовані події WPF загалом мають власні (backing) реалізації implementations логіки додавання та видалення.
  - Це дозволяє додавати обробники маршрутизованих подій за допомогою специфічного для мови програмування синтаксису.

```
void MakeButton() {
        Button b2 = new Button();
        b2.AddHandler(Button.ClickEvent, new RoutedEventHandler(Onb2Click));
}

void Onb2Click(object sender, RoutedEventArgs e) {
        //logic to handle the Click event
}
```

Операторний синтаксис C# для додавання обробників подій

```
void MakeButton2() {
        Button b2 = new Button();
        b2.Click += new RoutedEventHandler(Onb2Click2);
}

void Onb2Click2(object sender, RoutedEventArgs e) {
        //logic to handle the Click event
}
```

Додавання обробника подій у коді

```
<TextBlock Name="text1">Start by clicking the button below</TextBlock>
<Button Name="b1" Click="MakeButton">Make new button and add handler to it
</Button>
```

```
public partial class RoutedEventAddRemoveHandler {
        void MakeButton(object sender, RoutedEventArgs e) {
                Button b2 = new Button();
                b2.Content = "New Button";
                // Associate event handler to the button. You can
                remove the event
                // handler using "-=" syntax rather than "+=".
                b2.Click += new RoutedEventHandler(Onb2Click);
                root.Children.Insert(root.Children.Count, b2);
                DockPanel.SetDock(b2, Dock.Top);
                text1.Text = "Now click the second button...";
                b1.IsEnabled = false;
        }
        void Onb2Click(object sender, RoutedEventArgs e) {
                text1.Text = "New Button (b2) Was Clicked!!";
        }
```

Властивість Handled та поняття обробленості події

- Властивість Handled задіює будь-який обробник подій за маршрутом, щоб позначити маршрутизовану подію обробленою (Handled = true).
  - Після обробки для одного елементу з маршруту спільні дані події знову оголошуються (report) кожному слухачу на маршруті.
- Значення Handled впливає на те, як маршрутизована подія обробляється або оголошується протягом руху по маршруту.
  - Якщо Handled = true отримано в даних для маршрутизованої події, обробники, які її прослуховують для інших елементів, загалом більше не викликаються для конкретно цього екземпляру події.
  - Для найбільш поширених сценаріїв Handled = true означає «зупинку» маршрутизації (як тунельованої, так і bubbling, а також для будь-якої події, обробленої в точці маршруту обробником класу).

Механізм "handledEventsToo"

- Слухачі все ще можуть запускати обробники у відповідь на маршрутизовані події, для яких Handled = true.
  - Маршрутизація події не завжди зупиняється навіть при мітці handled.
- Механізм може використовуватись тільки в коді або в EventSetter:
  - У коді викликається WPF-метод AddHandler(RoutedEvent, Delegate, Boolean) для додавання обробника замість специфічного для МП синтаксису. Значення handledEventsToo встановлюється рівним true.
  - В EventSetter атрибуту HandledEventsToo присвоюється значення true.

Рекомендації щодо використання Handled

- У додатках досить поширена практика – обробляти тільки bubbling routed event для об'єкту, який її згенерував, та не брати до уваги характеристики маршрутизації події взагалі.

- Проте хорошою практикою вважається відмічати подію як «handled» всередині даних події, щоб уникати непередбачених побічних ефектів, якщо існує елемент вище (далі) по дереву, який також матиме обробник для цієї ж маршрутизованої події.

Класи-обробники (Class Handlers)

- Якщо оголошується клас, породжений від DependencyObject, також можна визначити та прикріпити обробник класу для маршрутизованої події, яка буде оголошеним або успадкованим членом класу.
  - Класові обробники викликаються раніше, ніж слухачі екземпляру, що прикріплені до екземпляру цього класу, whenever a routed event reaches an element instance in its route.
- Деякі елементи управління WPF мають притаманну їм обробку класу для деяких маршрутизованих подій.
  - Може здаватись, що маршрутизована подія не генерується, проте насправді вона має клас-обробник (class handled).
  - Маршрутизована подія ще може потенційно обробитись за допомогою обробників екземпляру, якщо будуть задіяні спеціальні програмні техніки.
  - Також багато базових класів та елементів управління розкривають віртуальні методи, які можна використати для переозначення поведінки класової обробки.

Прикріплені події в WPF

- Мова XAML визначає спеціальний тип подій – ***attached event***.
  - Прикріплена подія дозволяє додавати обробник для конкретної події до довільного елементу.
  - The element handling the event need not define or inherit the attached event, and neither the object potentially raising the event nor the destination handling instance must define or otherwise "own" that event as a class member.
- Система вводу WPF широко використовує прикріплені події.
  - Проте майже всі вони націлені на базові елементи.
  - Події вводу потім з'являються як еквівалентні маршрутизовані події, які є членами класу базового елементу.
  - Наприклад, прикріплена подія Mouse.MouseDown може бути оброблена простіше для будь-якого елементу, породженого від UIElement, використовуючи MouseDown над UIElement, а не за допомогою синтаксису прикріпленої події в XAML або коді.

Синтаксис для прикріпленої події

- У XAML синтаксис прикріпленої події задається так: ТипВласника.НазваПодії.
  - Вказування власника дозволяє прикріпляти подію до будь-якого елементу, що може бути інстанційованим.
  - Зверніть увагу на префікс aqua; тут він обов'язковий, оскільки прикріплена подія є кастомною та належить custom mapped xmlns.

```
<aqua:Aquarium Name="theAquarium" Height="600" Width="800"
aqua:AquariumFilter.NeedsCleaning="WashMe"/>
```

Сценарії використання прикріплених подій

- Хоч WPF визначає багато прикріплених подій, сценаріїв для їх прямого використання чи обробки дуже мало.
  - Загалом прикріплені події слугують архітектурним цілям, but is then forwarded to a non-attached (backed with a CLR event "wrapper") routed event.
  - Наприклад, underlying прикріплена подія Mouse.MouseDown може простіше оброблятись будь-яким елементом, породженим від UIElement, використовуючи MouseDown над цим UIElement, а не синтаксис прикріплених подій у XAML або коді.
  - Прикріплені події корисні з точки зору архітектури, оскільки дозволяють подальше розширення списку пристроїв введення.
  - Гіпотетичному пристрою буде потрібно лише згенерувати подію Mouse.MouseDown, щоб симулювати ввід мишею, а не виконувати наслідування від класу Mouse.
  - Проте цей сценарій включає обробку подій в коді, обробка прикріпленої події в XAML у цьому сценарії нерелевантна.

Обробка прикріплених подій у WPF

- Процес обробки в основному той же, що й для маршрутизованих подій.
  - Відмінності з'являються у визначенні джерела події та її розкриття класом в якості члена цього класу.
  - Проте, як зазначалось, існуючі прикрілені події WPF не призначені для обробки в WPF.
  - Частіше, мета події – дати можливість складеному елементу (composited element) оголосити стан батьківського елементу в композиції.
  - Тоді подія зазвичай генерується в коді й також покладається на class handling у відповідному батьківському класі.
  - Наприклад, від items всередині Selector очікується генерація прикріпленої події Selected, яка потім обробляється класом Selector та потенційно перетворюється ним в іншу маршрутизовану подію – SelectionChanged.

Визначення власної події: маршрутизована vs прикріплена

- При наслідуванні від загальних базових класів WPF можна реалізувати власні прикріплені події, включаючи відповідні методи у клас та використовуючи вже представлені допоміжні методи.
- Шаблон наступний:
  - ***Метод AddEventNameHandler з 2 параметрами: ідентифікатор події та обробник, що додається.*** Метод повинен бути публічним, статичним та не повертати значення.
  - ***Метод RemoveEventNameHandler з двома параметрами: ідентифікатор події та обробник, який видалятиметься.*** Характеристики аналогічні.
    - Ці методи організують доступ з коду до сховища обробників подій для прикріпленої події.
    - Метод-аксесор AddEventNameHandler підтримує XAML-обробку, коли атрибути обробника прикріпленої події оголошуються для елементу.

Представлений шаблон недостатньо точний на практиці

- Реалізація XAML reader може мати різні схеми ідентифікування underlying подій у supporting language та архітектурі.
  - Це одна з причин, чому WPF реалізує прикріплені властивості як маршрутизовані; ідентифікатор для події (RoutedEvent) вже визначено в системі подій WPF.
- Представлена реалізація **Add*EventName*Handler** для прикріпленої події WPF складається з виклику AddHandler з маршрутизованою подією та обробником у якості аргументів.
  - Дана стратегія реалізації та система маршрутизованих подій загалом обмежує обробку прикріплених подій до породжених класів від UIElement або ContentElement, оскільки тільки такі класи мають реалізації методу AddHandler().

Прикріплена подія NeedsCleaning над owner-класом Aquarium

```
public static readonly RoutedEvent NeedsCleaningEvent =
EventManager.RegisterRoutedEvent("NeedsCleaning", RoutingStrategy.Bubble,
typeof(RoutedEventHandler), typeof(AquariumFilter));

public static void AddNeedsCleaningHandler(DependencyObject d,
RoutedEventHandler handler) {
        UIElement uie = d as UIElement;
        if (uie != null) {
                uie.AddHandler(AquariumFilter.NeedsCleaningEvent, handler);
        }
}

public static void RemoveNeedsCleaningHandler(DependencyObject d,
RoutedEventHandler handler) {
        UIElement uie = d as UIElement;
        if (uie != null) {
                uie.RemoveHandler(AquariumFilter.NeedsCleaningEvent,handler);
        }
}
```

Raising a WPF Attached Event

- Генерувати визначені в WPF прикріплені події в коді зазвичай не потрібно – вони слідують концепції «служби», а за генерацію відповідають службові класи на зразок InputManager.
- Проте при визначенні власної прикріпленої події можна використати RaiseEvent для генерування прикріпленої події для будь-якого елементу UIElement або ContentElement.
  - Генерування маршрутизованої події (прикріпленої або ні) вимагає оголошення конкретного елементу в дереві елементів у якості джерела події (event source);
  - Це джерело оголошується в якості RaiseEvent caller.
  - Визначення елементу-джерела для дерева покладається на Ваш service's responsibility

Дякую за увагу!