목차

[1장 Application과 Window(25) 2](#_Toc124888284)

[2장 브러시(49) 3](#_Toc124888285)

[3장 컨텐트(75) 4](#_Toc124888286)

[4장 버튼과 컨트롤(99) 5](#_Toc124888287)

[5장 StackPanel과 WrapPanel(129) 6](#_Toc124888288)

[6장 DockPanel과 Grid(151) 7](#_Toc124888289)

[7장 캔버스(183) 8](#_Toc124888290)

[8장 의존 프로퍼티(195) 9](#_Toc124888291)

[9장 입력 이벤트의 라우팅(213) 10](#_Toc124888292)

[10장 커스텀 엘리먼트(245) 12](#_Toc124888293)

[11장 단일 자식 엘리먼트 13](#_Toc124888294)

1장 Application과 Window(25)

WPF를 사용할 때는 STAThread 속성을 Main에 사용해야 한다. -> 멀티스레드가 되지 않는다.

WPF app은 일반적으로 Window, Application타입의 객체를 생성하는 것으로 시작한다.

Window는 표준 애플리케이션 창을 생성할 때 사용하며, 띄워진 창과 관련된 기능을 담당한다.

Application은 Run 메소드를 호출함으로써 메시지 루프를 형성하고 이 메시지 루프를 통해서 키보드, 마우스 등을 이용한 사용자 입력을 받을 수 있게 도와준다.

사실상 프로그램이 하는 일은 **이벤트에 대해 반응하는 것이 전부**라고 할 수 있다. 그 중 키보드, 마우스, 스타일러스와 관련된 몇 가지의 이벤트는 UIElement클래스에 정의돼있고 Window클래스는 이 모든 이벤트를 상속받는다.

Application클래스의 Current는 프로그램이 생성한 Application클래스를 반환한다. 이때 여기에는 MainWindow를 가지고 있으며 이는 만들어진 여러 window객체 중에서 내가 메인창으로 설정한 객체를 저장한다.

Application 클래스에는 시작할 때 사용되는 OnStartup, 종료할 떄 사용되는 OnExit가 있으며 이들은 각각 Run메소드가 호출된 직후, 반환되기 직전에 일어나는 이벤트에서 이용된다.

MainWindow뿐 아니라 여러 창을 만들 수 있는데 이들은 모두 동등한 지위를 가진다. 이때 이들을 작업표시줄에서 보이게 하는 속성은 Window클래스 안의 ShowInTaskbar이다. 또한 각 창들을 따로 직접 닫거나 메인창이 닫히면 한꺼번에 닫히게 할 수 있는데 이를 설정하는 것은 Application클래스의 ShutdownMode에서 설정할 수 있다.

각 창들은 Window클래스 안의 Owner를 이용해 소유관계를 정할 수 있다. 소유관계를 정하게 되면 종속된 창은 항상 소유자의 앞에 나타나고, 소유자를 닫으면 모두 같이 닫히게 된다.

대화창은 크게 모달리스(modeless)와 모달로 구분된다. 모달 대화창은 대화창을 불러낸 창으로 전환할 수 없는 반면에 모달리스 대화창은 대화창의 전환에 제약이 없다. 이 때 모달 대화창은 Run메소드에서 불러오는 메시지루프에 참여하지 않기 때문에 모달이 될 수 있는 것이다.(입력을 별개로 받는다?)

Window클래스의 OnKeyDown, OnKeyUp를 이용해 키보드를 누르고 떼는 것에 대한 이벤트를 활용할 수 있다. 이때 만약에 특정 유니코드 문자를 얻는 것이 목적이라면 OnTextInput을 이용해야한다.

Window클래스에서 WindowStyle, ResizeMode, WindowState, Topmost, Background 등을 이용해 창의 상태를 정할 수 있다.

2장 브러시(49)

WPF에서는 색상을 다루기 위해 System.Windows.Media의 Color 구조체를 이용한다. Color는 R, G, B, A라는 byte타입의 프로퍼티가 있고, 각각은 3원색과 투명도를 결정한다. 그리고 Colors클래스에는 이미 만들어져 있는 색들이 있으며 이를 Color clr = Colors.Red와 같이 활용할 수 있다.

WPF의 창을 꾸미기 위해서는 Color를 직접 활용하는 것이 아니라 이를 포함하고 여러 기능을 가지고 있는 Brush클래스를 활용해야 한다.

그 중 SolidColorBrush클래스는 단색으로 색을 칠하는 기능을 가지고 있다.

Brush 객체의 상태를 변경하기만 해도 관련된 내용을 창에 실제로 적용하는 코드를 만들지 않아도 변경된 상태가 자동으로 창에 적용되는 것을 알 수 있다. 이는 Brush클래스가 Freezable클래스를 상속받았기 때문이다.

Freezable클래스에는 Changed라는 이벤트가 구현되어 있으며 이 이벤트는 객체에서 변경된 내용을 바로 창에 적용한다. 반면에 Brushes클래스에 있는 이미 만들어져 있는 Brush객체를 가져다 쓰면 해당 객체를 변경하려고 시도할 때 예외가 발생하는 것을 알 수 있다. 이는 마찬가지로 Freezable클래스를 상속받았지만 해당 클래스의 IsFrozen변수가 True인 상태인 객체이기 때문이다. IsFrozen이 True인 경우에는 객체의 상태를 바꿀 수 없으며, 이는 객체 변경으로 인한 이벤트 확인을 건너뛰게 해줌으로써 프로그램의 성능을 향상시킨다. 또한 기존에 IsFrozen이 False인 객체도Freezable클래스의 Freeze메소드를 사용해 IsFrozen을 True로 만들 수 있으며, 이를 다시 변경가능한 객체로 만드는 방법은 없다. 대신에 Freezable클래스의 Clone메소드를 이용해 변경가능한 객체로 다시 복사해서 사용할 수 있다.

GradientBrush클래스는 두 개 이상의 색을 점진적으로 섞어서 칠할 때 사용할 수 있다.

그 중 LinearGradientBrush는 두 개의 Color, 두 개의 Point 객체를 이용해 각 Point객체가 각각의 기준색을 가지고 두 점을 이용해 만든 선을 기준으로 수직하게 색을 섞어서 칠한다. 이 때의 Point객체는 (0, 0) ~ (1, 1)사이의 값을 이용해야 하며 각각 좌측상단, 우측하단을 의미한다. 또한 두 점 대신에 각도를 넣어서 그라데이션을 만들 수도 있다. 그리고 이 클래스의 SpreadMethod를 이용해 범위에서 벗어난 곳에 대한 색을 어떻게 칠할 것인지 정할 수 있다. 중간에 그라데이션에 이용될 색을 추가하기 위해서는 이 클래스의 GradientStops에 새로운 GradientStop 객체를 추가해주면 된다.

RadialGradientBrush는 타원형으로 색을 칠할 때 이용하며, Center, RadiusX, RadiusY, GradientOrigin을 이용해 그라데이션의 모양을 바꾼다.

이와 같은 Brush는 Background 뿐만 아니라 Window의 OpacityMask, BorderBrush, Foreground에도 활용할 수 있다.

3장 컨텐트(75)

Window클래스의 Content는 원하는 객체를 저장하는 용도로 사용한다. 이는 문자열, 비트맵, 그림, 등과 WPF에서 지원하는 여러가지 컨트롤을 이용할 수 있다. 다만 Content에는 오직 하나의 객체만을 할당할 수 있다. Content는 ContentControl클래스에 있는 것이고 Window는 이를 상속받아 사용하는 것이다.

Window가 상속받은 Control클래스에는 폰트에 관련된 내용들이 있다. 관련된 변수들을 이용해 폰트의 모양을 바꿀 수 있다.

Window에 있는 SizeToContent는 컨텐트의 크기에 맞춰 창의 크기를 조절하는데 이용한다.

Content에는 문자열 외에도 다양한 형태의 내용을 넣을 수 있다. 일반적인 객체가 들어온 경우에는 해당 객체의 ToString을 이용해 출력하게 된다.

UIElement는 키보드, 마우스, 스타일러스 펜등을 처리하는 것이 구현돼있다. 또한 객체의 그래픽 표시를 얻기 위해 호출되는 OnRender메소드도 가지고 있다.

Content는 단순히 텍스트 출력만 지원하는 것이 아니라 자연스러운 그래픽을 출력하는 것을 목적으로 한다. 그러므로 Content의 동작에 관련된 모든 객체는 UIElement를 상속받은 것과 그렇지 않은 것으로 나눌 수 있다. 전자는 UIElement의 OnRender메소드를 이용해 출력하고 후자는 ToString을 이용해 출력하게 된다.

BitmapImage클래스는 실제 이미지를 메모리에 로딩하는 기능을 수행하고, Image클래스는 이를 출력할 때 사용한다. 즉, Image객체는 Content를 이용해 이미지를 표시하는데 이용한다. Image클래스의 Stretch, StretchDirection 등을 이용해 사진의 표시방법을 여러 방식으로 바꿀 수 있다.

FrameworkElement클래스는 UIElement클래스를 직접 상속하고 있는 유일한 클래스이다. 이 클래스에는 WPF의 레이아웃에 중요한 변수들이 있으며 이 변수들을 수정해 이를 상속하고 있는 객체들의 레이아웃을 정해줄 수 있다. Image클래스 또한 이를 상속하고 있다.

Image클래스는 래스터 이미지를 화면에 출력하는 반면에 비슷하게 그림을 그리는 Shape클래스는 이차원 벡터그래픽을 구현한다. 이 클래스로 그려지는 그래픽의 결과는 Content의 크기에 의존하게 된다.

Content에 텍스트를 표시할 때는 사실 TextBlock클래스의 객체를 이용하고 있는 것이다. 이 클래스는 Inline클래스 컬렉션인 Inlines를 프로퍼티로 가지고 있는데 이를 이용해서 텍스트 내에서 세세한 설정을 할 수 있다.

Inline클래스는 ContentElement를 상속하고 있는 FrameworkContentElement를 상속하고 있다. 이는 UIElement, FrameworkElement와 유사한 관계를 가지고 있지만 OnRender를 가지고 있지는 않기 때문에 직접 화면상에 그리는 역할을 수행하지 않는다. 대신에 UIElement를 상속받은 객체의 특정 속성이 되어 그려지는 모양을 바꾸는데 일조한다. 그러므로 FrameworkElement를 상속받은 TextBlock객체 내의 Inline클래스는 TextBlock의 일부 텍스트의 속성을 변경하는데 사용된다고 볼 수 있다. 또한 ContentElement에는 UIElement처럼 사용자 입력에 관련된 이벤트도 정의되어 있으므로 일부 텍스트에만 이벤트를 발생시키는 것도 가능하다.

4장 버튼과 컨트롤(99)

WPF에서의 컨트롤은 사용자와 능동적으로 상호작용하는 요소들을 의미한다. Control클래스를 상속받은 클래스는 여러 개 존재하고, 대표적으로 버튼, 리스트 박스, 스크롤바, 에디트 필드, 메뉴, 툴바 등의 기능을 제공한다. 이들은 System.Windows.Controls와 System.Windows.Controls.Primitives 네임스페이스에 정의되어 있다.

Button클래스로 만드는 Button객체 또한 ContentControl을 상속받고 있기 때문에 자신만의 Content를 가지고 있다. 그러므로 Button은 다양한 객체를 받아들이는 Content객체의 특징을 모두 사용할 수 있다.

Button의 메소드 중에 Focus메소드를 이용하면 해당 버튼을 가선택할 수 있다. 또한 텍스트 문자열의 글자 앞에 \_를 입력하면 해당 \_ 다음 문자와 alt를 이용해 바로가기 키를 만들어 사용할 수 있다. 그리고 Button객체의 IsDefault가 true인 경우에는 enter키에 반응하게 되고, IsCancel이 true인 경우에는 ESC키에 반응하게 된다. ButtonBase클래스에 있는 ClickMode를 이용해 마우스 클릭의 움직임에 따른 행동을 정할 수 있다.

ButtonBase, MenuItem을 포함한 몇 개의 클래스에는 Command가 정의되어 있다. 이를 이용해 붙여넣기와 같은 시스템의 명령들을 쉽게 이용할 수 있다. 또한 CommandBinding클래스를 이용해 Command와 이벤트 핸들러를 Binding해서 이용할 수 있고, 이를 UIElement클래스에 속해 있는 CommandBindings객체를 이용해 관리할 수 있다. Command바인딩을 이용하게 되면 바인딩한 시스템 명령을 이용하더라도 바인딩한 객체를 이용하는 것과 똑같이 이벤트를 일으킬 수 있는 장점이 있다. 또한 상황에 따른 예외처리도 좀더 쉽게 처리할 수 있게 된다.

ToggleButton에 해당하는 CheckBox, RadioButton은 눌린 상태와 눌리지 않은 상태를 확인하는데 사용한다. 그 중 checkbox는 네모난 선택칸을 이용해 선택할 수 있고, RadioButton은 여러 개의 선택 사항 중 1개를 동그란 선택칸을 통해서 선택한다.

SetBinding을 이용해 객체(A)의 값을 다른 객체(B)의 값과 바인딩할 수 있다. 이때 바인딩을 시도하는 객체(A)의 DataContext에 다른 객체(B)를 저장해 어떤 객체와 바인딩을 하려는 것인지 알려줘야 한다.

ToolTip을 이용하면 마우스 포인터가 위에서 잠깐 멈출 때 텍스트가 나타나도록 만들 수 있다.

Label은 기본적으로는 텍스트 문자열을 출력하는 데 이용한다. 이때 특이한 점은 Label자체는 문자열만 사용할 수 있으므로 단축키를 지정하게 되면 Label의 컨트롤에 포커스가 가는 것이 아니라 Label 내에 있는 다음 컨트롤로 포커스가 이동한다. 그렇기 때문에 주로 입력에 사용하는 TextBox와 연계해서 이용하는 경우가 많다.

TextBox클래스는 입력을 하는데 이용하고, 내부에는 Text 프로퍼티가 있어서 최초 문자열을 지정하거나 사용자가 입력하는 문자열을 입력하는데 사용한다. 또한 Control을 상속받고 있으므로, 이와 관련된 기능을 이용할 수 있다. 하지만 TextBlock과 다르게 모든 텍스트가 같은 색, 폰트를 이용해야한다. Textbox의 AcceptsReturn을 true로 함으로써 Textbox객체 내에서 줄넘김이 가능해진다.

반면에 RichTextBox는 다양한 글자와 문단 서식이 지원된다. RichTextBox의 내용은 객체 내의 FlowDocument객체인 Document를 이용해 타 객체로 주고받게 된다. 이때 RichTextBox는 일반 텍스트형식, RTF, XML 등의 포맷을 지원하기 때문에 입출력시 이를 활용할 수 있다. RichTextBox는 모든 키 누름을 스스로 처리하기 때문에 단축키를 활용하기 위해서는 preview이벤트를 이용해야 한다. preview이벤트는 키보드 누름이 RichTextBox에 도달하기 전에 미리 입력을 확인해 이벤트 인자의 Handled를 true로 만들어 이미 처리된 이벤트로 만들어 RichTextBox에서 이용되지 않게 만들 수 있다.

5장 StackPanel과 WrapPanel(129)

일반적인 Content에는 오직 하나의 객체만을 지정할 수 있다. 그래서 WPF에는 이 문제를 해결할 수 있는 클래스가 몇 개 설계되어 있다. 이들을 Panel이라고 하며, 이를 활용해 layout을 만들 수 있다.

패널들은 Panel클래스를 상속받고 있고, 이들의 가장 큰 특징은 내부 요소들의 크기와 위치를 정해진 레이아웃에 맞게 자동으로 조정해줄 수 있으며, Children을 이용해 내부에 2개 이상의 요소들을 저장할 수 있다는 것이며, 이를 동적 레이아웃이라고 한다.

Panel클래스에는 Children이란 프로퍼티를 이용해 자식 요소들을 저장할 수 있다. Children은 UIElement객체들의 컬렉션이므로 UIElement를 상속하고 있는 객체들을 모두 넣어줄 수 있다.

StackPanel객체는 내부 요소들을 수직 또는 수평으로 정렬해서 보여준다. StackPanel객체의 위치나 크기는 내부 요소의 크기, 위치와 본인의 alignment상태에 따라서 달라지게 된다. 또한 SizeToContent를 이용해 자동으로 조절할 수도 있다.

AddHandler메소드를 이용하면 해당 객체 내에서 일어나는 특정 이벤트를 지정해 그 이벤트가 일어나는지 확인할 수 있다. 또한 그 이벤트가 일어났을 때 특정 이벤트 핸들러를 이용해 그 이벤트를 처리할 것이지 정해줄 수 있다. 이런 경우에는 이벤트 핸들러 내의 sender와 이벤트 매개변수의 주체가 달라질 수 있다.

ScrollViewer객체를 이용하면 창에 공간이 부족한 경우 보는 영역을 조절할 수 있게 해주는 스크롤을 지원해주는 창을 사용할 수 있다. 이때 ScrollBarVisibility변수를 통해서 어떻게 스크롤바가 나타나게 할 것인지 정할 수 있다. 이때 스크롤바 내부의 스크롤 조절 버튼은 버튼으로 인식되므로 이벤트 사용에 주의를 할 필요가 있다. 또한 ScrollViewer말고도 Viewbox와 같은 방법을 이용해 공간에 맞게 모든 객체의 크기를 조절해서 보여주는 방법도 있다.

RadioButton객체는 다른 버튼들과는 다르게 자기 자신만으로 이용하면 큰 의미가 없고 GroupBox클래스를 활용해 묶어줘야 여러 선택지 중에서 1개만 선택할 수 있게 해주는 본연의 기능을 활용할 수 있다.

WrapPanel은 StackPanel과 유사한 특징을 가지고 있다. WrapPanel객체는 만약에 공간이 부족하면 다음 요소를 자동으로 다음 줄에서 표현해주는 것이 StackPanel과의 다른 점이다.

6장 DockPanel과 Grid(151)

DockPanel클래스는 창의 내부의 다른 요소가 특정 방향에 도킹되어 있는 형태로 만들어져야할 때 사용하는 패널이다. 이에 대한 설정은 첨부 프로퍼티(attached property)를 이용한다. 예를 들어 ctrl을 DockingPanel객체 dock에 도킹하게 하려면 dock의 자식으로 ctrl을 추가한 뒤, DockPanel.SetDock(ctrl, Dock.Rigth) 혹은 ctrl.SetValue(DockPanel.DockProperty, DockRight)와 같은 메소드를 이용해 DockProperty를 바꿔야 한다.

Grid클래스는 행과 열의 크기, 개수를 자유롭게 조절할 수 있는 기능을 지원하는 패널이다. 원하는 크기로 고정해놓을 수도 있고, 내부 요소 크기를 기준으로 자동으로 조절하는 방법도 있고, 남는 공간을 기준으로 비율을 따져가면서 조절하는 방법도 있다. Grid클래스는 RowDefinitions, ColumnDefinitions에 연관된 객체를 넣어서 행과 열의 개수를 정할 수 있다. 그리고 요소들을 Grid객체에 넣기 위해서는 객체의 자식으로 요소를 설정한 뒤, DockPanel에서 사용했던 방법처럼 첨부 프로퍼티를 설정하는 메소드인 Grid.SetRow와 Grid.SetColumn을 이용해 어느 행과 열에 요소를 배치할 것인지 결정할 수 있다. 또한 SetRowSpan, SetColumnSpan을 이용해 여러 행, 열에 걸쳐서 배치할 수도 있다.

GridSplitter클래스는 그리드 패널의 크기를 조절하는데 이용한다. 이는 Control클래스를 상속하고 있는 Thumb클래스를 상속하고 있다. GridSplitter객체는 다른 요소와 셀을 공유할 수 있기 때문에 다른 요소에 가려지거나 다른 요소를 가릴 수 있다. 그래서 주로 GridSplitter객체만을 위한 그리드를 만들어서 이용하는 경우가 많다. GridSplitter객체는 Alignment와 Resizebehavior, ResizeDirection을 조절해 작동 방식을 바꿀 수 있다.

Scrollbar클래스는 Control클래스를 상속받는 RangeBase클래스를 상속하고 있다. Orientation을 이용해 가로나 세로로 배치할 수 있으며, Minimum, Maximum값을 정해서 Value의 범위를 정할 수 있다. 또한 SmallChange, LargeChange를 조절해 스크롤의 움직임 정도를 조절할 수 있다. 이때 Scrollbar객체의 내부 값들은 double값임을 주의해야 한다.

Grid와 유사하게 UniformGrid가 있으며 이는 행, 열이 모두 같은 폭, 넓이를 가진다. 또한 Grid와 다르게 RowDefinition, ColumnDefinition을 정하는 대신에 행과 열의 수를 Rows, Columns에 지정해주는 방식으로 사용한다. 그리고 UniformGrid에는 위치를 정하는 첨부 프로퍼티가 없는 대신에 자식을 추가하면 첫 자리부터 순서대로 들어가게 된다. 이때 Rows나 Columns의 값에 0을 넣게 되면, 해당 부분은 자식 수에 따라 적정한 값이 자동으로 정해진다.

7장 캔버스(183)

Canvas클래스는 내부 요소의 위치를 좌표로 지정하는 전통적인 그래픽 환경에 가장 가까운 레이아웃이다. 각 UIElement들은 x, y left, top과 같은 프로퍼티가 없으므로 캔버스 내의 위치를 설정할 대는 Canvas.SetLeft, CanvasSetTop과 같은 메소드를 이용해 위치를 지정해야 한다. 이는 DockPanel처럼 첨부 프로퍼티와 연결돼있다. 또한 마찬가지로 자동으로 크기가 조절되지 않으므로 Width, Height 등의 값도 명시적으로 정해줘야 한다. 또한 Canvas패널의 경우에는 내부 요소들을 겹치는 것이 가능하며, 먼저 추가한 요소가 뒤에 추가한 요소에게 가려진다.

Polygon클래스는 Points가 정의되어 있으며 이들은 Point를 저장한다. 그리고 FillRule에 정해진 설정에 맞춰 Points를 기준으로 그림을 그린다.

Border클래스는 FrameworkElement를 상속하고 있는 Decorator를 상속하고 있다. 이는 단색의 경계선을 출력할 수 있는 기능을 가지고 있는 것이 특징이다.

8장 의존 프로퍼티(195)

WPF에서는 부모 요소의 특성을 그대로 따라서 자식 요소에 적용되는 프로퍼티들이 있다. 예를 들면 특정 패널의 FontSize가 16이라면 내부에 있는 버튼의 FontSize도 자동으로 16으로 정해진다. 또한 이러한 경우에 버튼의 FontSize를 명시적으로 수정한 후에는 부모 패널의 값을 변경하더라도 자동으로 적용되지 않는다. 이것을 프로퍼티 계승이라고 하며, 이러한 특성들을 저장하고 자동으로 WPF프로그램에 적용시키기 위해서 사용하는 것이 의존 프로퍼티이다.

기본적으로 의존 프로퍼티는 DependencyProperty클래스의 객체이다. 이는 위에서 말한 기능들을 하기 위해서 필요한 각종 값들을 미리 저장하고 있는 객체이다. 이 각종 값들을 우리는 메타데이터라고 부르고, PropertyMetadata클래스에 관련된 내용이 정의되어 있다. 또한 의존 프로퍼티를 이용해 값의 변경에 따라 자동으로 WPF프로그램이 바뀌게 하기 위해서는 내가 만든 의존 프로퍼티를 DependencyProperty에 등록할 필요가 있다. 그러므로 의존 프로퍼티를 사용하기 위해서는 DependencyProperty객체를 만들고, 해당 의존프로퍼티에 관련된 PropertyMetadata객체를 만든 뒤, 이를 DependencyProperty.Register 정적 메소드를 이용해 등록해야 한다.

의존 프로퍼티의 주 값은 메타데이터의 일부이므로 이를 직접 활용할 수 없다. 그러므로 이를 얻기 위해서는 DependencyObject클래스에 정의되어 있는 SetValue, GetValue메소드를 이용해 값에 접근해야 한다. 그러므로 의존 프로퍼티를 이용할 모든 클래스는 DependencyObject를 상속해야만 한다. 이를 마치고 나면 기본 C#에 있는 프로퍼티와 비슷하게 사용할 수 있게 하기 위해서 C# 프로퍼티의 get, set을 이용해 일반적인 C#프로퍼티인 것처럼 활용할 수 있게 된다.

한 개의 의존 프로퍼티를 여러 클래스에서 사용하는 경우에는 각 클래스가 해당 프로퍼티를 따로 정의해야 한다. 이를 새로 정의하기 위해서 메타데이터 각 클래스용으로 따로 만들어야 한다. 하지만 처음 등록하는 의존 프로퍼티가 아니기 때문에 Register메소드를 사용해 이를 등록하는 것이 아니라, 이미 만들었던 의존 프로퍼티 클래스의 AddOwner 정적 메소드와 OverrideMetadata 정적 메소드를 이용해 의존 프로퍼티의 사용자 클래스를 추가로 등록해야 한다.

이전의 DockPanel에서는 SetDock을 이용해 특정 요소의 DockProperty를 직접 정해준 적이 있다. 이때의 DockProperty는 생각해보면 모든 패널에서 쓰이는 것이 아니라 DockPanel에서만 사용되기 때문에 이를 위해 Dock프로퍼티를 다른 클래스(예 : UIElement)에 미리 만들어 두는 것은 좋은 생각이 아니다. 그렇기 때문에 DockProperty를 사용하는 요소이더라도 이 프로퍼티에 관련된 객체를 저장하고 있는 클래스는 없다.

이런 경우를 위해서 해당 의존 프로퍼티를 처음 등록할 때 DependencyProperty.RegisterAttached메소드를 이용해서 등록한다. 그리고 해당 프로퍼티를 사용하게 되는 다른 클래스에는 정의되어 있지 않기 때문에, 이 프로퍼티가 존재하는 클래스의 정적 메소드를 이용해서 해당 프로퍼티를 사용하는 객체를 엮어주는 방식으로 의존 프로퍼티를 등록해주는 것이다. 이러한 형태의 의존 프로퍼티는 클래스에 해당 프로퍼티를 첨부하는 형태이므로 첨부 프로퍼티(attached property)라고 부른다.

9장 입력 이벤트의 라우팅(213)

WPF에서의 주된 사용자 입력은 키보드, 마우스, 스타일러스 펜으로 3가지이다. 입력 이벤트들은 RoutedEventArgs클래스를 상속하는 InputEventArgs클래스를 상속하고 있는 객체들을 전달받아서 사용한다.

WPF에서는 RoutedEventArgs클래스를 통해서 이벤트의 라우팅을 지원한다. 즉, 이 클래스를 활용하면 어떤 요소에서 이벤트가 발생했을 때, 이벤트가 발생한 요소와 서로 연결되어 있는 다른 요소에서 해당 이벤트를 쉽게 갖다 쓸 수 있다.

이때, RoutedEventArgs클래스는 처리할 이벤트 및 처리방식을 담고있는 RoutedEvent객체, 현재 담고있는 이벤트가 처리되어 추가적으로 라우팅이 필요한지를 나타내는 Handled변수, 사용자가 직관적으로 생각했을 때 이벤트를 발생시켰다고 생각되는 위치를 담고있는 Source, 실제로 이벤트가 발생한 위치를 담고있는 OriginalSource로 이루어져있다.

이를 이용해 RoutedEvent를 활용한 이벤트를 주로 어떻게 구성하는지 ButtonBase클래스에 있는 Click이벤트를 통해 살펴보면, Click이벤트는 RoutedEvent객체인 ClickEvent, ClickEventHandler델리게이트를 가진 event객체인 Click, 실제로 Click이벤트를 발생시킬 때 활용하는 OnClick메소드로 구성되어 있다.

Click이벤트를 사용하기 위해서는 먼저 ClickEvent객체를 EventManager클래스에 등록할 필요가 있다. EventManager.RegisterRoutedEvent 정적 메소드를 이용해 RoutedEvent객체를 등록할 수 있는데, 이를 위해서는 이벤트의 이름, 라우팅의 방법, 해당 이벤트가 다룰 핸들러의 종류(RoutedEventHandler클래스), 해당 이벤트가 등록된 클래스의 타입이 필요하다. 이때 라우팅의 방법으로는 터널 다운, 버블 업이 있다. 터널 다운은 이벤트가 발생하면 눈에 보이는 요소들의 루트로부터 실제 이벤트가 발생한 요소까지 순서대로 이벤트가 발생한다.(window->grid->button순) 반면에 버블 업은 이벤트가 발생했을 때, 눈에 보이는 요소들 중 실제 이벤트가 발생한 요소부터 루트까지 순서대로 이벤트가 발생하게 된다. (button->grid->window순) 주로 터널 다운 라우팅 방법으로 처리되는 이벤트에는 Preview라는 이름이 접두사로 붙게 된다.

Click객체는 일반 C#에서 이벤트를 활용하듯이 RoutedEvent객체를 활용할 수 있도록 도와준다. 이를 위해서 Click객체는 핸들러가 등록되면 자동으로 UIElement.AddHandler메소드를 이용해 ClickEvent에 등록받은 핸들러를 추가한다.

위의 방법과 별개로 현재 요소에는 없는 이벤트이지만 해당 요소의 트리에 포함된 다른 요소의 이벤트를 라우팅하고 싶은 경우가 생길 수 있다. 이런 경우에는 해당 요소객체의 AddHandler메소드를 활용해서 직접 원하는 이벤트와 핸들러를 등록할 수도 있다.

OnClick메소드는 실제로 Click이벤트를 발생시키고 싶은 곳에서 이 메소드를 불러오게 되면 Click이벤트가 발생한다. 이 때, 메소드의 내부에서는 ClickEvent객체와 이벤트를 일으킨 객체를 소스(source)로 활용해 RoutedEventArgs객체를 만들고, 이 객체를 UIElement클래스로부터 상속받은 RaiseEvent메소드에 전달해 이벤트를 일으키게 된다.

이러한 입력에 관한 이벤트들은 대부분 이미 정의되어 있으므로, 실제 사용할 때는 event객체에 원하는 핸들러를 추가등록해서 사용하거나 이벤트가 있는 클래스를 상속한 뒤, On메소드를 오버라이드해서 사용할 수 있다.

마우스, 스타일러스 이벤트에서는 내가 선택한 위치를 기준으로 가장 앞에 있는 요소가 이벤트의 소스(source)에 해당한다. 즉, 해당 요소는 눈에 보여야하고 활성화되어 있어야 한다.(요소의 Visibility가 Visibility.Visible, IsEnabled가 true) 또한 키보드 이벤트에서는 입력 포커스를 가지고 있는 요소가 소스가 된다.

이때 마우스 전용 RoutedEventArgs클래스로 MouseEventArgs클래스가 있는데 이는 마우스의 상태에 대한 각종 정보를 담고 있다. 또한 이에 관련된 이벤트를 처리하는 방법 중 대표적인 것으로는 OnMouseDown, OnMouseUp이 있는데 각각의 이벤트가 같은 요소에서 일어날 것이라는 보장이 없기 때문에 이들이 항상 쌍으로 나타나는 것은 아니다. 그러므로 만약에 마우스 버튼을 누른 상태에서 마우스 포인터를 완전히 WPF프로그램 밖으로 이동한 후, 버튼을 떼면 OnMouseUp이 일어나지 않을 수도 있다.

이러한 마우스의 움직임은 CaptureMouse메소드를 이용해 보완할 수 있다. 예를 들어 OnMouseDown을 오버라이드할 때, 그 내부에서 CaptureMouse메소드를 호출하면 프로그램 밖으로 마우스가 나가더라도 마우스의 움직임을 계속 확인하게 된다. 이는 ReleaseMouseCapture를 호출해서 캡처를 멈출 수 있다. 마우스 캡처는 윈도우의 적절한 동작을 방해할 가능성이 있기 때문에 상황에 따라서 윈도우가 마우스 캡처를 일방적으로 풀 수 있다. 만약 MouseUp이벤트를 받고도 마우스 캡처를 중지하지 않았다면, 사용자가 다른 창을 클릭할 때 마우스 캡처가 중지되고, 모든 규칙을 잘 지켰더라도 윈도우가 시스템 모달 대화상자를 띄울 필요가 있다면 마우스 캡처는 중지된다. 마지막으로 마우스 캡처를 이용하는 경우에는 OnMouseUp이 사용되지 않아서 원래 처리해야할 부분을 처리하지 못하는 문제가 생길 수 있으므로, LostMouseCapture이벤트에 대한 핸들러를 만들어서 필요한 정리 작업을 해야 한다.

MouseEventArgs와 비슷하게 키보드 입력과 관계된 이벤트를 위해서 KeyEventArgs클래스와 TextCompositionEventArgs클래스가 존재한다. 이 중 KeyEventArgs는 키보드 입력 그 자체를 위한 것이고, TextCompositionEventArgs는 누른 키에 대응하는 문자를 위한 것이다.

10장 커스텀 엘리먼트(245)

사용자 요소를 만들 때는 FrameworkElement나 Control을 상속해서 만들게 되는 경우가 많다. 이때 Control의 경우는 의무적으로 구현해야 하는 것이 많은 편이기 때문에 해당 요소들이 반드시 필요한 것이 아니라면 굳이 사용할 필요는 없다. 이 사용에 대해서 가장 큰 힌트를 주는 프로퍼티는 Focusable이다. Focusable은 UIElement에서 정의될 때 기본값이 false로 정의됐지만, Control에서 오버라이드 하면서 true가 됐으므로, 만약에 키보드 입력 포커스를 받을 만한 사용자 요소라면 Control로 상속하면 좋은 경우가 많다.

OnRender 메소드는 UIElement에 정의되어 있다. 이는 그림을 그리는 기능을 담당하는데 OnRender에서 그리는 내용은 바로 화면에 보이는 것이 아니라 WPF시스템에 의해 출력이 보류되다가, 출력이 필요한 경우에 자동으로 호출된다. 또한 InvalidtateVisual메소드를 명시적으로 호출해 OnRender를 사용할 수도 있다. 또한 OnRender를 통해서 그려지는 그래픽 객체는 마우스 이벤트를 처리하기도 한다.

OnRender에서는 그릴 크기를 결정하기 위해서 RenderSize프로퍼티를 이용한다. 이는 UIElement에서 여러 가지 요소에 영향을 받아 계산된다.

사용자 요소가 출력되기 위해서는 이에 대한 크기를 알려줄 방법이 있어야 할 것이다. 이를 계산하는 것을 담당하는 메소드가 FrameworkElement클래스에 MeasureOverride이며, 입력값으로 그려지는 요소가 가질 수 있는 최대 크기(availableSize)를 받고 출력값으로 요소가 그려지기 위해서 필요하다고 판단되는 크기(desiredSize)를 내놓아야 한다. 이때, availableSize와 desiredSize를 비교해서 만약 desiredSize가 더 크다고 판단이 된다면 이는 렌더링을 시도한 영역이 실제로 그릴 수 있는 영역보다 크다고 판단되어, availableSize에 맞춰 렌더링의 클리핑이 일어난다. 이와 같은 MeasureOverride가 호출되어 렌더링 영역을 정하고 나면 항상 그 뒤에는 OnRender가 연이어서 호출된다. 그리고 MeasureOverride를 오버라이드 하지 않은 경우에는 desiredSize로 0이 반환되므로, OnRender측에서 RenderSize를 잘 조절해야 한다.

의존 프로퍼티를 Register를 이용해 등록할 때, 메타데이터의 옵션으로 AffectsRender를 넣으면, 해당 프로퍼티가 변경된 경우에 InvalidateVisual메소드가 호출되어, OnRender를 사용하게 된다. 또한 AffectsMeasure를 넣으면 마찬가지로 InvalidateMeasure메소드가 호출되며, 이는 MeasureOverride를 호출하게 해서 결과적으로 OnRender를 호출하게 된다.

11장 단일 자식 엘리먼트(267)

FrameworkElement나 Control을 상속받는 많은 클래스들은 자식 요소를 갖는 경우가 많다. 그리고 이런 자식 요소를 수용하기 위해서, 주로 1개의 프로퍼티와 4개의 메소드를 오버라이딩한다.

VisualChildrenCount : Visual 클래스에 정의된 읽기 전용 프로퍼티로 현재 요소가 관리하는 자식의 수를 알 수 있도록 오버라이딩 해야 한다.  
GetVisualChild : Visual 클래스에 정의된 메소드로 인덱스를 이용해 자신의 자식 요소 중 인덱스에 해당하는 것을 반환한다. 이 때, 인덱스에 문제가 있다면, null을 반환하도록 구현하는 것이 아니라 예외를 던지도록 구현해야 한다.  
MeasureOverride : FrameworElement 클래스에 정의된 메소드로 현재 요소의 요구 크기를 계산해서 반환한다. 그 과정에서 자식의 Measure 메소드를 호출해 그 자식의 DesiredSize 프로퍼티를 확인할 필요가 있다.  
ArrangeOverride : FrameworElement 클래스에 정의된 메소드로 요소의 최종 레이아웃 크기를 받게 되면, 각 자식 요소들에 대한 Arrange를 사용해, 그 표면 위에 배치한다.  
OnRender : 요소를 그리는 데 사용하며, 자식 요소들은 여기서 그려진 내용보다 더 앞에 그려지게 된다.

위의 메소드 중 MeasureOverride, ArrangeOverride, OnRedner는 각각이 차례로 호출되며, 앞의 호출이 빠진 채로 뒤의 호출만 일어날 수도 있다. 자식 요소를 여러 개 가지게 되는 요소들은 주로 패널(panel)로 분류되며, 이 장에서는 단일 자식을 가진 요소만을 다룬다.

자식 요소를 유지하고 관리하는 것은 그 자식 요소를 가진 클래스의 책임이므로, AddVisualChild, AddLogicalChild, RemoveVisualChild, RemoveLogicalChild와 같은 메소드를 호출해 의존 프로퍼티의 상속과 이벤트 라우팅을 위해 필요한 비주얼 논리 트리를 관리해야 한다. 이를 위해서 주로 위의 메소드를 활용한 관리를 구현한 UIElement객체라고 가정된 child 프로퍼티를 만들어서 사용하게 된다.

MeasureOverride와 ArrangeOverride는 자식 요소의 Measure나 Arrange를 불러오게 되는 경우가 많은데, 이렇게 자식 요소에서 호출된 메소드는 때때로 자식 요소의 MeasureOverride, ArrangeOverride를 불러온다. 이와 같은 방식으로 결과적으로 가장 밑단의 자식요소까지 고려한 계산이 진행된다. 특히 MeasureOverride는 계산 결과로 DesiredSize 프로퍼티가 바뀌는데 이는 본인이 직접 사용하는 경우는 적으나 부모 요소에 이를 활용해 계산하는 경우가 많다. 또한 ArrangeOverride는 인자로 sizeFinal을 받아서 이를 자식 요소의 DesiredSize와 함께 활용해 자식 요소의 배치를 결정짓는다. 그리고 반환하는 값은 그대로 RenderSize가 되므로 주로 sizeFinal을 그대로 반환한다.

위의 내용들을 활용하면 어떤 요소의 역할을 적당히 나눠서 이를 클래스로 표현할 수 있다. 예를 들어 어떤 커스텀 버튼을 만들기 위해, Decorator 클래스를 상속하는 ButtonDecorator 클래스를 만들고, 요소의 실제 모양과 버튼의 상태에 대해서 다루는 내용을 넣는다. 그리고 Decorator객체를 가지고 있는 실제 Button 클래스를 만들어, Decorator를 Button의 자식 요소로 만들고, 외형이나 상태에 관련된 내용을 프로퍼티와 이벤트를 이용해 연결한다. 이를 통해서 실제 이벤트와 관련된 기능적인 내용은 Button 클래스에서 처리하고, 외형적인 부분은 Decorator 클래스에서 처리하도록 만들 수 있다.

커스텀 요소를 만드는 다른 방법으로 템플릿을 활용할 수도 있다. 이를 활용하기 위한 핵심 프로퍼티는 Control 클래스에 정의된 Template이다. 이는 ControlTemplate 타입이며, 객체를 전달하면 해당 ControlTemplate 객체에 이미 설정된 프로퍼티들을 전달받은 객체에 설정해준다.

이를 위해서 먼저 ControlTemplate 객체를 만들어야 한다. 또한 해당 객체가 가지는 특정한 타입의 실제 값을 설정하기 위해서 FrameworkElementFactory 객체를 만들어서 간접적으로 만든 객체에 값을 설정해주고, 템플릿 객체에 이를 넣어준다. 또한 비슷하게 어떤 이벤트에 관련된 내용을 설정하고 싶다면, Trigger 객체를 만들어, 어떤 상황에서 이벤트가 일어날 지 간접적으로 설정하고, Setter 객체를 만들어 실제로 이벤트가 일어났을 때 어떤 것이 바뀌게 될지를 간접적으로 설정해주고, 이를 Trigger 객체에 넣어준다. 그 후, 템플릿의 Triggers에 Trigger객체를 추가해주면 해당 템플릿을 통해 만든 컨트롤은 위에서 간접적으로 설정해둔 이벤트를 작동시킬 수 있다.

OnRender 메소드에서는 DrawingContext를 활용할 수 있고 이는 그려야 하는 것에 대한 정보를 담고 있다. 마찬가지로 이를 활용할 수 있는 클래스로는 DrawingVisual 클래스가 있는데 DrawingVisual 객체를 만들고, DrawingContext 객체에 연결한 뒤 DrawingContext 객체를 활용해 그림을 그리면 그린 그림 자체를 DrawingVisual 클래스에 저장할 수 있다. 이때 DrawingVisual 클래스는 Visual 클래스에서 직접 파생된 것이므로 그 자체로는 MeasureOverride, ArrangeOverride와 같은 메소드가 없다. 그리고 당연하게도 DawingVisual 객체 또한 자식 요소로서 등록해서 사용할 수 있다.