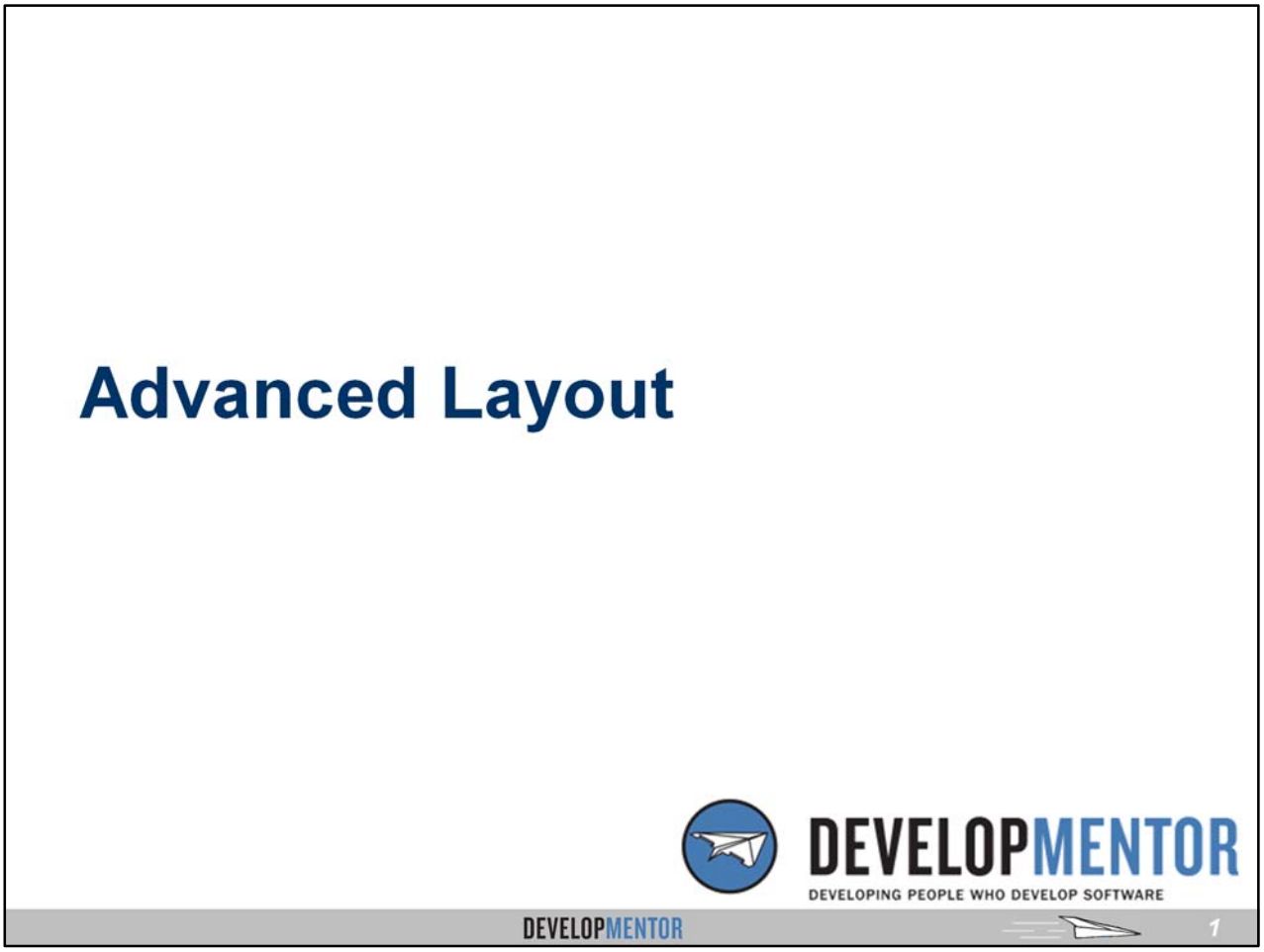
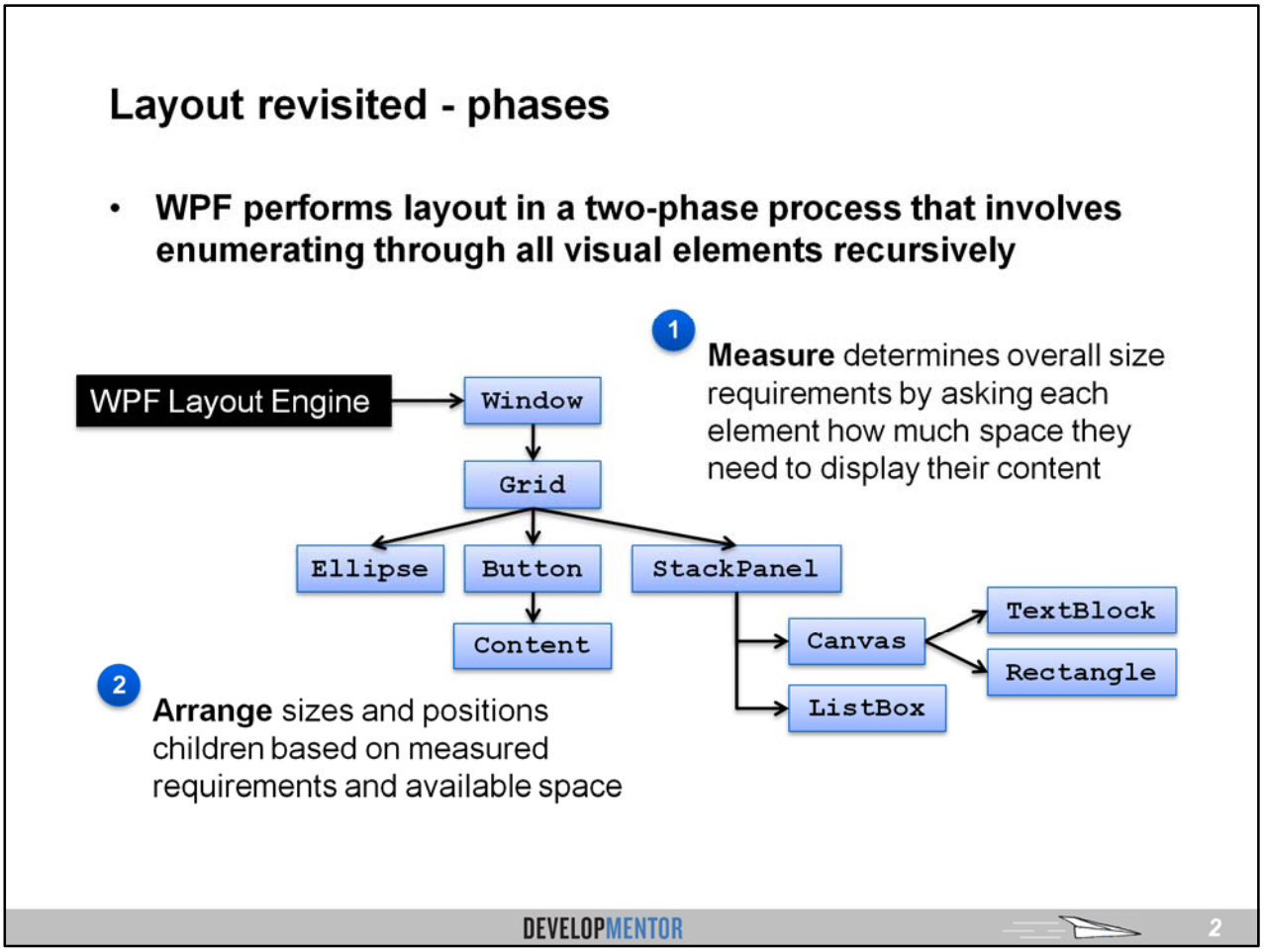
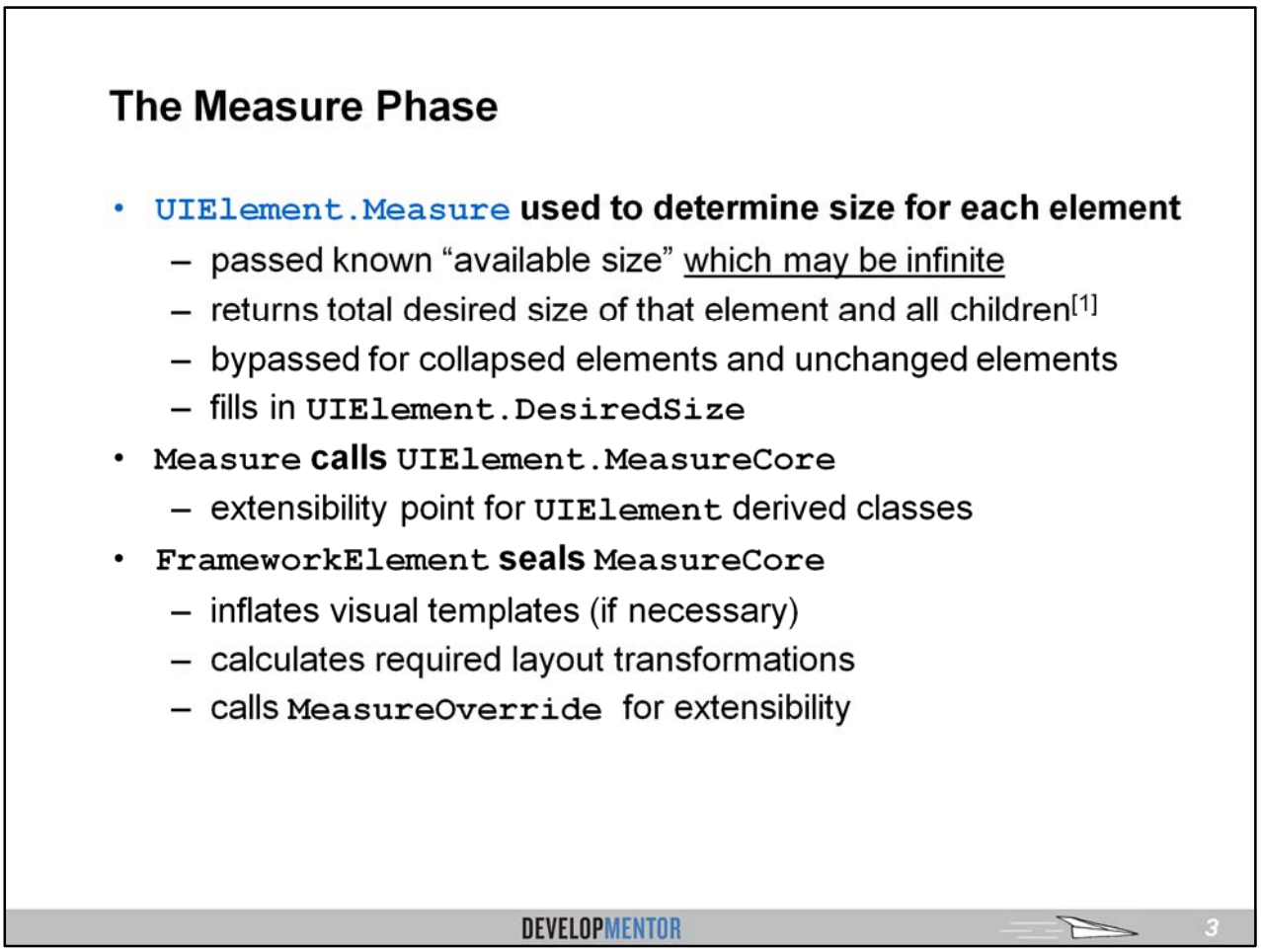
Appendix: Advanced Layout




Layout revisited - phases
• WPF performs layout in a two-phase process that involves enumerating through all visual elements recursively
1
Measure determines overall size requirements by asking each element how much space they need to display their content
WPF Layout Engine
Window
Grid
Ellipse
Button
StackPanel
Content
Canvas
TextBlock
Rectangle
ListBox
2
Arrange sizes and positions children based on measured requirements and available space
DEVELOPMENTOR
2



[1] It must not return infinity in either direction – WPF will throw an exception if this occurs.

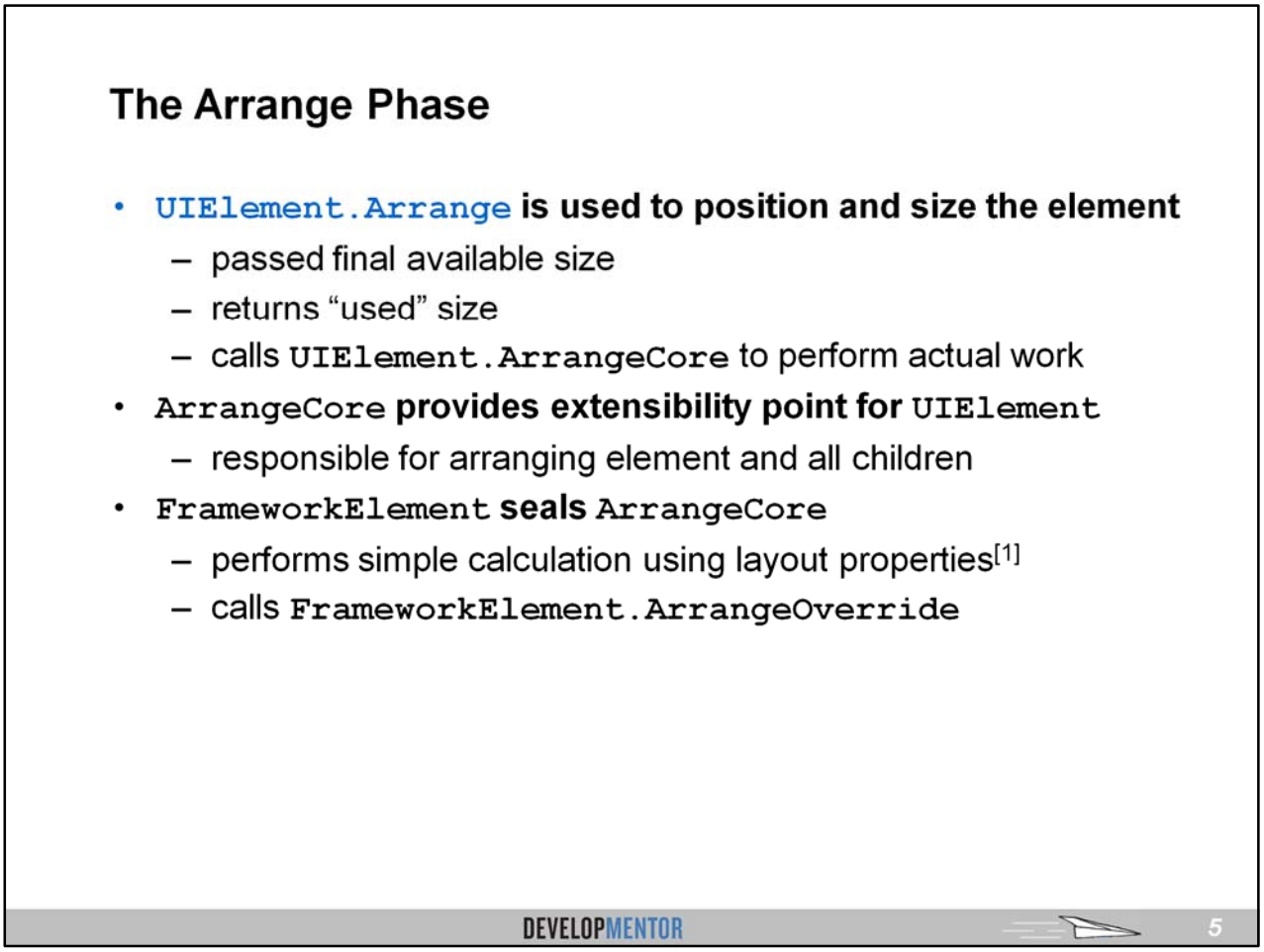
FrameworkElement.MeasureOverride

- `MeasureOverride` **used to provide custom sizing logic**
  - must call `Measure` on all children or they will not be sized
  - can restrict available space of each child if necessary
  - returns `Size(0,0)` by default[1]

```
protected override Size MeasureOverride(Size availableSize)
{
   foreach (UIElement child in VisualChildren)
   {
      child.Measure(availableSize);
      availableSize.Deflate(child.DesiredSize);
   }
   Size desired = ... sum of children's DesiredSize ...;
   return desired;
}
```

DEVELOPMENTOR 4

[1] Given this, it is not necessary to ever call base.MeasureOverride(availableSize) as it never does anything.



[1] These include Horizontal/Vertical alignment, Margin and ClipToBounds.

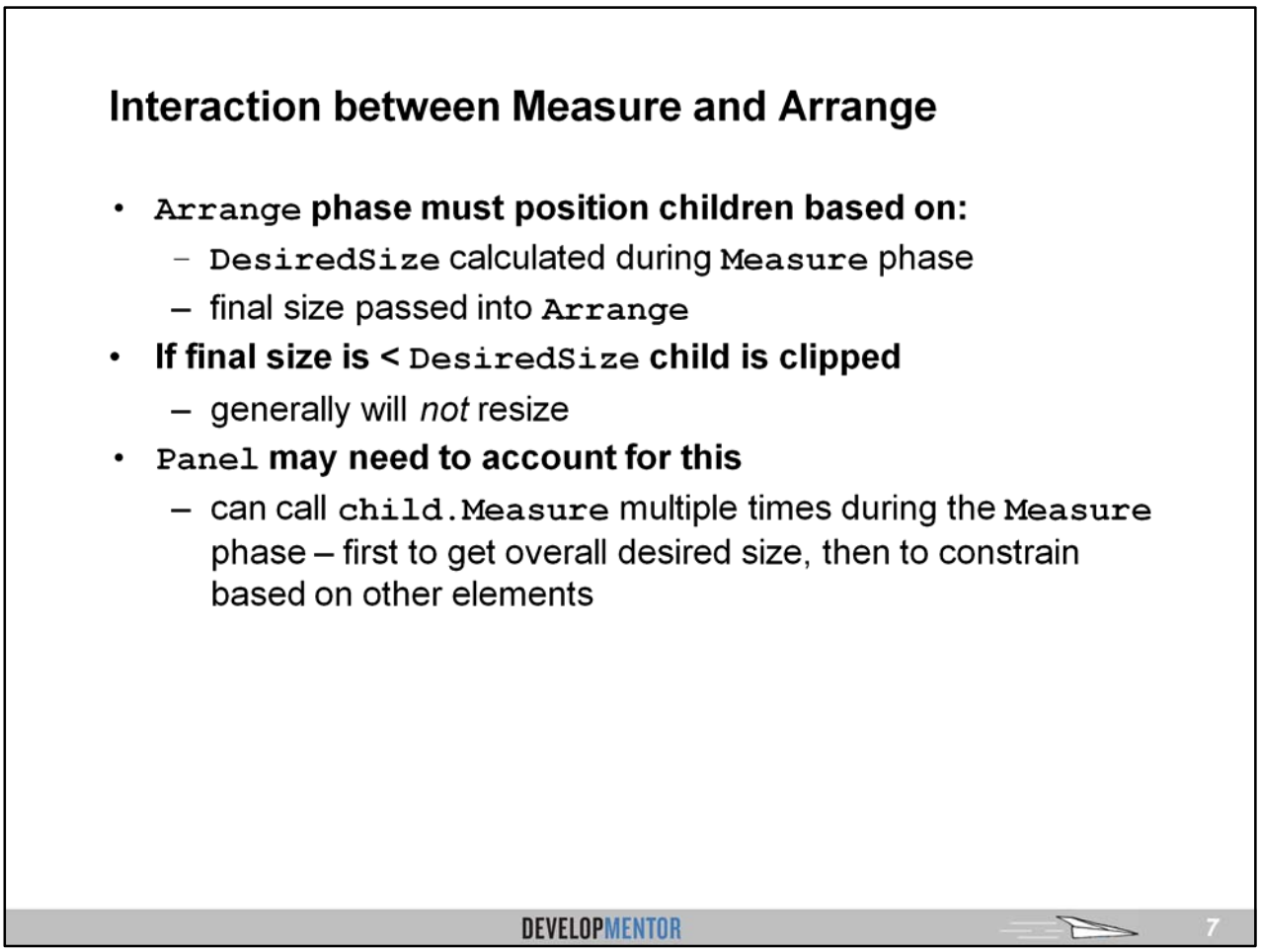
FrameworkElement.ArrangeOverride

- `ArrangeOverride` **is extension point for** `FrameworkElement`
  - responsible for positioning any children within passed size
  - should return final rendering size[1]
  - common to use values calculated by measure phase

```
protected override Size ArrangeOverride(Size finalSize)
{
    foreach (UIElement child in VisualChildren)
    {
       double childX = 0, childY = 0;
       Size childSize = finalSize;
       child.Arrange(new Rect(childX, childY, childSize));
       childY += child.DesiredSize.Height;
       childSize.Height -= child.DesiredSize.Height;
    }
    return finalSize;
}
```

DEVELOPMENTOR 6

[1] The base method returns the passed "finalSize". As with MeasureOverride, it is not necessary to call the base class when this is overridden.


Interaction between Measure and Arrange
• Arrange phase must position children based on:
– DesiredSize calculated during Measure phase
– final size passed into Arrange
• If final size is < DesiredSize child is clipped
– generally will *not* resize
• Panel may need to account for this
– can call child.Measure multiple times during the Measure phase – first to get overall desired size, then to constrain based on other elements
DEVELOPMENTOR
7

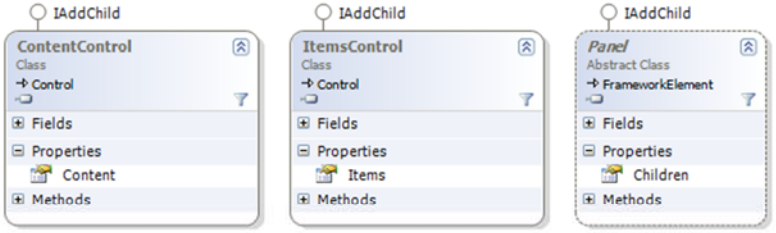
What triggers layout?

- **Layout is generally triggered by size changes in elements**
  - by DPs marked as `AffectsMeasure` or `AffectsArrange`
- **Can also happen because**
  - `UIElement.InvalidateVisual` is called
  - an item becomes visible which was previously collapsed
  - a layout transform is applied
  - a call to `InvalidateMeasure` or `InvalidateArrange`
- `Measure` **always implies a call to** `Arrange`
  - but the reverse is not necessarily true

Appendix: Advanced Layout

Dealing with visual children

- **Primary layout container is the `Panel`**
  - holds `UIElements` in `Children` collection
- **.. But some controls also contain children**
  - `ContentControl.Content`, `ItemsControl.Items`, etc.

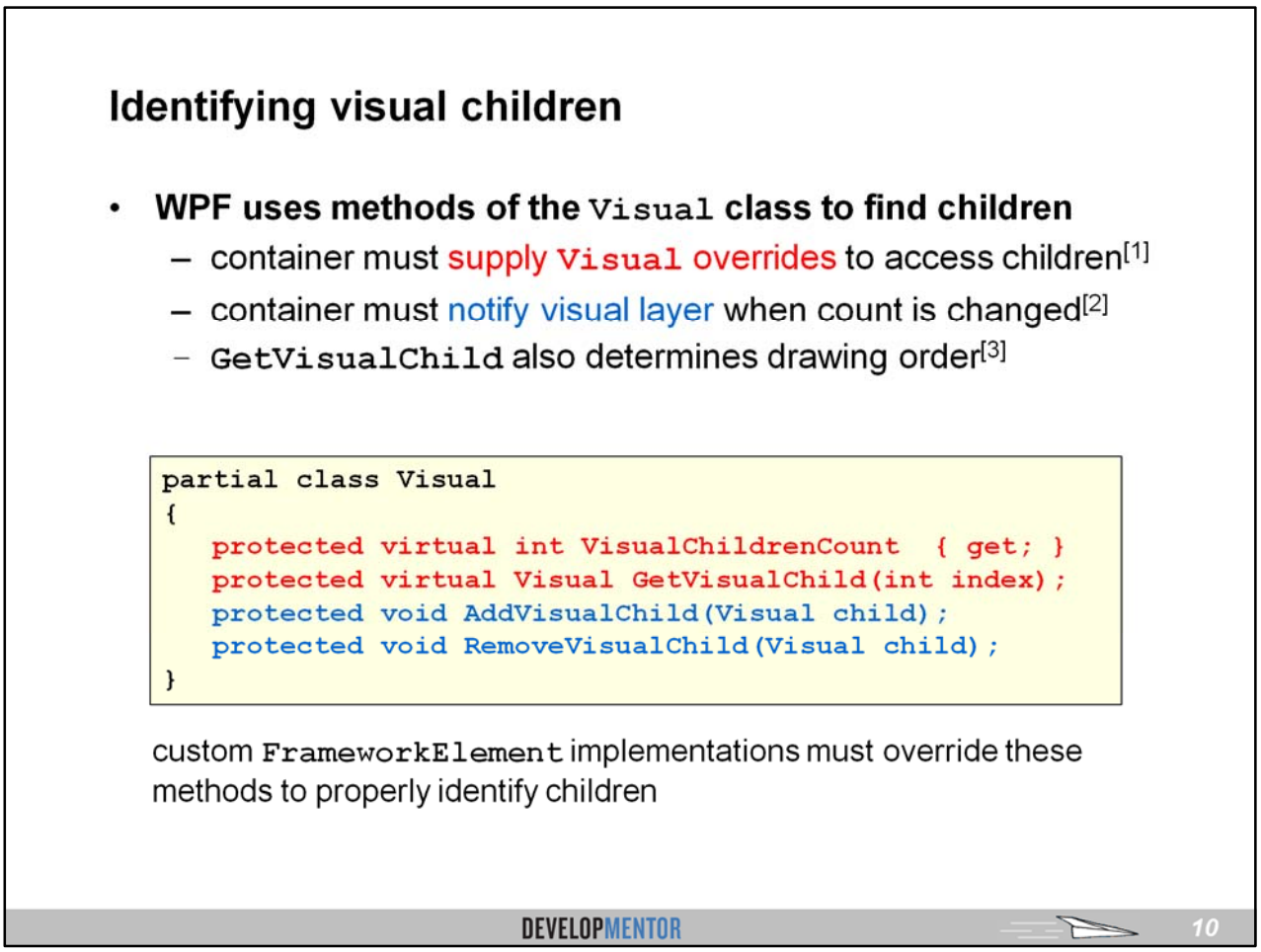


each type uses a different property to manage the child elements -- WPF really needs some consistent way to identify visual children...

[1] The default implementation in Visual does not allow any children – it will always return zero for the count and throw an exception if you try to get a child.

[2] Calling the AddVisualChild and RemoveVisualChild methods is required. Without this, the child will not be added to the visual tree.

[3] This is simply based on the order it returns the children in. Visuals returned at lower indexes are drawn lower in the Z-order.



[1] The default implementation in Visual does not allow any children – it will always return zero for the count and throw an exception if you try to get a child.

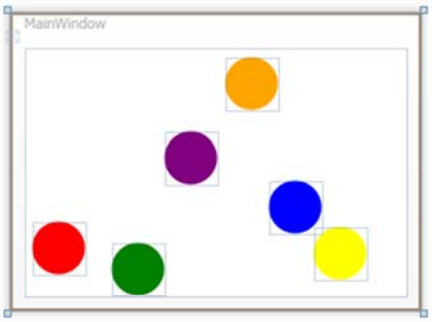
[2] Calling the AddVisualChild and RemoveVisualChild methods is required. Without this, the child will not be added to the visual tree.

[3] This is simply based on the order it returns the children in. Visuals returned at lower indexes are drawn lower in the Z-order.

Creating custom layout panels

- **Reusable layout designs can be created with custom panels**
  - where existing panels are not sufficient or complex layout model used in several places

```
<app:RandomPositionPanel>
   <Ellipse Fill="Red"
      Width="40" Height="40" />
   <Ellipse Fill="Blue"
      Width="40" Height="40" />
   <Ellipse Fill="Green"
      Width="40" Height="40" />
   <Ellipse Fill="Orange"
      Width="40" Height="40" />
   <Ellipse Fill="Purple"
      Width="40" Height="40" />
   <Ellipse Fill="Yellow"
      Width="40" Height="40" />
</app:RandomPositionPanel>
```



here the panel always randomizes the position of the elements

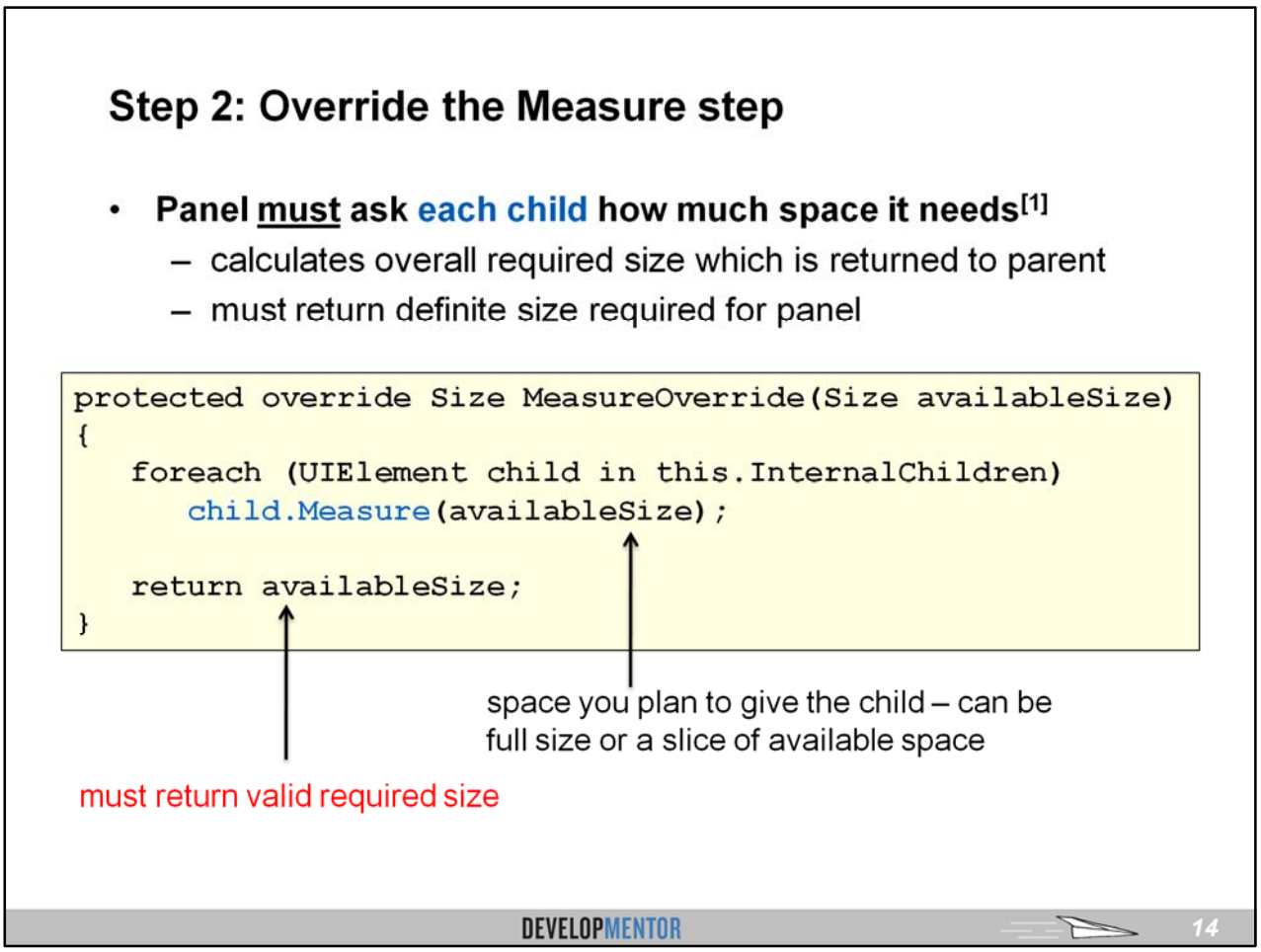
Panel layout process

- **Panel must implement the two-phase layout process**
  - **Measure phase** – determine how much size is necessary by asking each child for the desired size and adding any panel required size
  - **Arrange phase** – panel locates and sizes each child based on the total amount of space given by WPF
- **Creating a custom panel involves 3 steps:**
  1. derive a class from `Panel`
  2. override `MeasureOverride` to process the measure phase
  3. override `ArrangeOverride` to process the arrange phase

Step 1: Creating a Custom Panel

- Create a class derived from the base `Panel` class

```
public class RandomPositionPanel : Panel
{
   ...
}
```



[1] You can pass in Double.PositiveInfinity for Width/Height as the available size in the child.Measure call to represent "infinite" space and get an idea of how much space the child might use. Just be prepared to receive these values back as DesiredSize because some child controls (primarily panels) will eat up as much as you tell them is available.

Ex: child.Measure(new Size(Double.PositiveInfinity, Double.PositiveInfinity));

Be careful if you do this as the control may return infinity back to you and that's not a legal value for you to return to WPF.

Step 3: Override the Arrange step

- **Panel tells each child its new position and size**
  - can use `child.DesiredSize` to get measured size

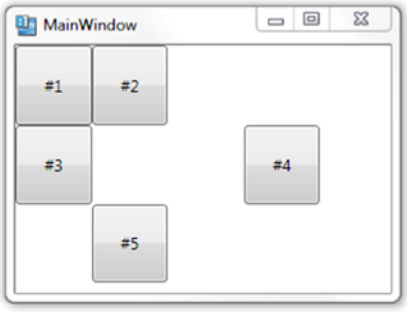
```
private readonly Random _rnd = new Random();
protected override Size ArrangeOverride(Size finalSize)
{
   foreach (UIElement child in this.InternalChildren)
   {
      int childX = _rnd.Next((int)(finalSize.Width -
                              child.DesiredSize.Width));
      int childY = _rnd.Next((int)(finalSize.Height -
                              child.DesiredSize.Height));
      child.Arrange(new Rect(new Point(childX, childY),
                             child.DesiredSize));
   }
   return finalSize;
}
```

returns "used" size – can be different than final size if it did not use it all

Adding layout specific information to children

- Some panels require additional layout information to work
  - `DockPanel` uses `DockPanel.Dock`
  - `Grid` uses `Grid.Row|Column`
  - `Canvas` uses `Canvas.Top|Left`

```
<app:AutoGrid>
   <Button>#1</Button>
   <Button Content="#2"
     app:AutoGrid.Column="1" />
   <Button>#3</Button>
   <Button Content="#4"
     app:AutoGrid.Column="2" />
   <Button Content="#5"
     app:AutoGrid.Column="1"/>
</app:AutoGrid>
```



auto-sizing grid requires column placement

Adding layout information to children

- **Custom panels can also supply layout-specific properties**
  - defined as attached properties on the panel class
  - properties that change layout should be marked as `AffectsParentMeasure` and/or `AffectsParentArrange`

```
public partial class AutoGrid : Panel
{
   public readonly static DependencyProperty ColumnProperty =
        DependencyProperty.RegisterAttached("Column",
          typeof(int), typeof(AutoGrid),
          new FrameworkPropertyMetadata(0,
          FrameworkPropertyMetadataOptions.AffectsParentArrange));

   public static int GetColumn(UIElement e) { ... }
   public static void SetColumn(UIElement e, int column) { ... }
}
```


Making the Designer aware of Attached Properties

- `AttachedPropertyBrowsableForChildrenAttribute` **enables designer support for attached properties**
  - will show property on each child element in logical tree
  - applied on getter static method

```
partial class AutoGrid
{
    [AttachedPropertyBrowsableForChildren]
    public static int GetColumn(UIElement e)
    {
        ...
    }
}
```


Using layout properties

- Use properties in `Measure|ArrangeOverride`

```
public partial class AutoGrid: Panel
{
   protected override void MeasureOverride(Size availableSize)
   {
      int currentRow = 0, lastColumn = 0;
      ...
      foreach (UIElement child in this.InternalChildren)
      {
         int column = GetColumn(child);
         if (column == 0 || column <= lastColumn)
            ++currentRow;
         ...
      }
   }
}
```

calculate row based on requested column – this is used to determine how much space the panel needs to layout all the children..

Overriding existing properties

- **Sometimes default property values are inappropriate**
  - can override with new defaults specific to the panel
  - replaces existing metadata with new information

```
partial class AutoGrid : Panel
{
   public AutoGrid()
   {
      HorizontalAlignmentProperty.OverrideMetadata(
            typeof(AutoGrid),
            new FrameworkPropertyMetadata(
                  HorizontalAlignment.Left,
            FrameworkPropertyMetadataOptions.AffectsMeasure));
      ...
   }
}
```


want the `HorizontalAlignment` property for the panel to default to Left

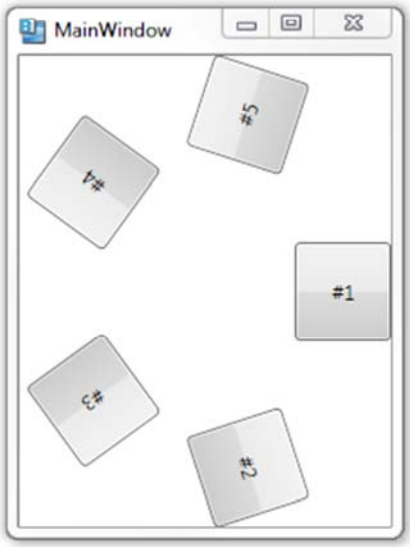
Adding panel properties and events

- **Properties applied to the panel should be defined as Dependency Properties**
  - allows for data binding and value change detection
  - make sure to add appropriate metadata
- **Events should be declared as Routed Events**
  - generally use only bubbled or direct events

```
partial class AutoGrid : Panel
{
   public static readonly DependencyProperty MinColumnSizeProperty =
      DependencyProperty.Register("MinColumnSize",
        typeof(int), typeof(AutoGrid),
        new FrameworkPropertyMetadata(0,
            FrameworkPropertyMetadataOptions.AffectsMeasure));
   ...
}
```


Transforming child elements

- **Panels can apply transforms to children to affect visualization**



here children are laid out around an ellipse and each is rotated clockwise as they go around by applying a `RotateTransform` to each child

Applying Transforms

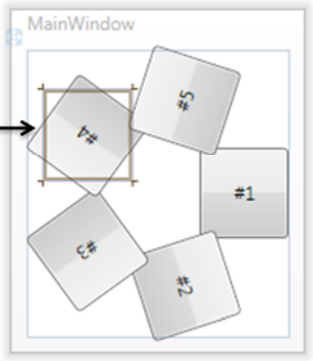
- **Transforms should always be applied in `Arrange`**
  - prefer `RenderTransform` for performance

```
protected override Size ArrangeOverride(Size finalSize)
{
   double x, y, angle;
   Size maxItemSize;
   ...
   foreach (UIElement child in this.InternalChildren)
   {
      ...
      child.RenderTransformOrigin = new Point(0.5, 0.5);
      child.RenderTransform = new RotateTransform(
                                    angle * 180 / Math.PI);

      child.Arrange(new Rect(new Point(x, y), maxItemSize));
      ...
   }

   return finalSize;
}
```


Arrange + Render Transforms

- **WPF always arranges `UIElement`s in a rectangular fashion**
  - makes size calculations much easier (and faster)
- **When applying render transforms always position and size elements using *non-transformed* coordinates**
  - remember that transformation is performed when it is rendered

gray rectangle shows actual layout "space" reserved for visual



Knowing when layout changes are happening

- **Use `SizeChanged` event to detect changes in element size**
  - override `OnChildDesiredSizeChanged` to monitor children
- **Use `LayoutUpdated` event to monitor for layout changes**
  - raised after arrange phase on each element in visual tree
  - does not indicate what caused layout change
  - can be used to provide layout animations

```
public partial class AnimatingPanel : Panel
{
   public AnimatingPanel()
   {
      LayoutUpdated += new OnLayoutUpdated;
   }
   void OnLayoutUpdated(object sender, EventArgs e)
   {
      DoAnimations();
   }
}
```


Summary

- **Layout system in WPF is quite complex and powerful**
  - can end up being a performance bottleneck
- **Layout is processed in two phases**
  - Measure to calculate required size
  - Arrange to position and size elements
  - recursively processed through entire visual tree
- **Panels allow any layout algorithm to be captured and reused**
  - anytime you need to arrange visual children, consider a panel
- **Custom panels are easily created**
  - supply `MeasureOverride` and `ArrangeOverride`
  - make sure to call `Measure/Arrange` on all children

