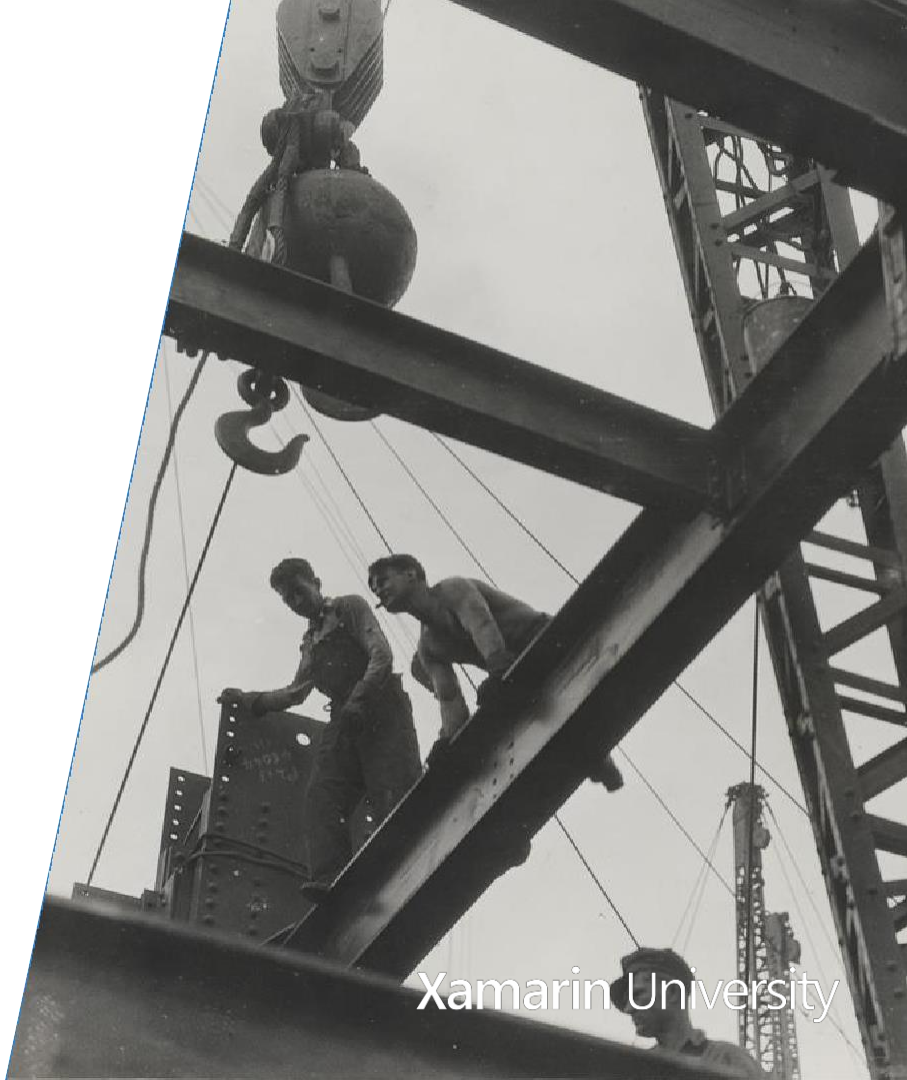
Layout in Xamarin.Forms

Microsoft

Objectives

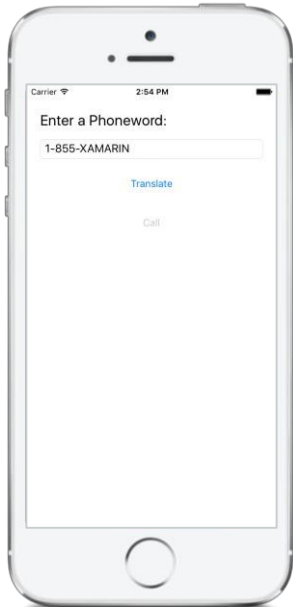
1. Specify the size of a view
2. Arrange views with `StackLayout`
3. Apply Attached Properties
4. Arrange views with `Grid`
5. Scroll a layout with `ScrollView`

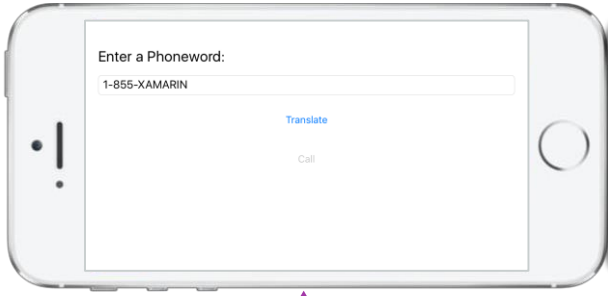


Motivation

- Using layout containers to calculate view size and position helps your UI adapt to varied screen dimensions and resolutions

E.g. you request that views are "stacked" one after the other

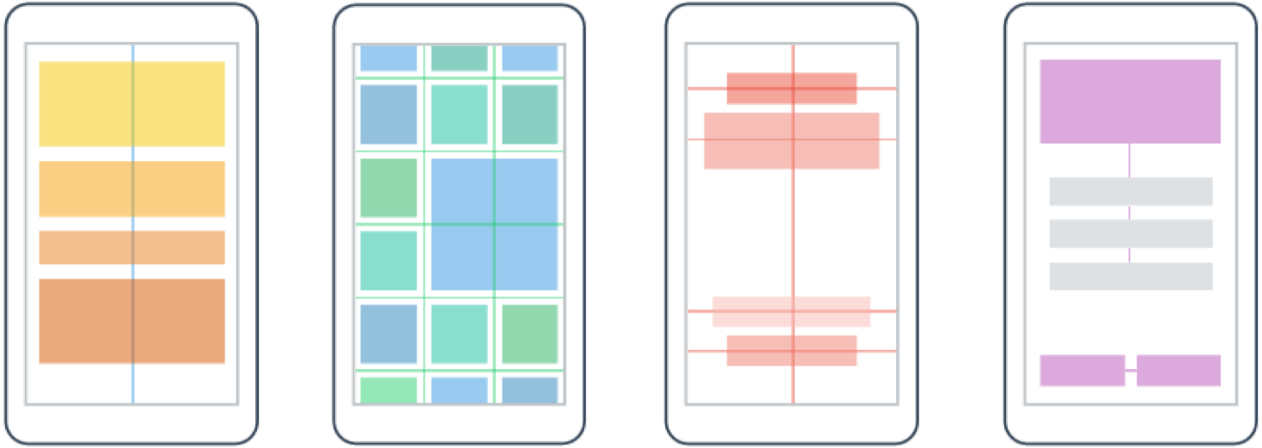




Sizes/positions are recalculated automatically when device rotates

What is a layout?

- A *layout* is a Xamarin.Forms container that determines the size and position for a collection of children

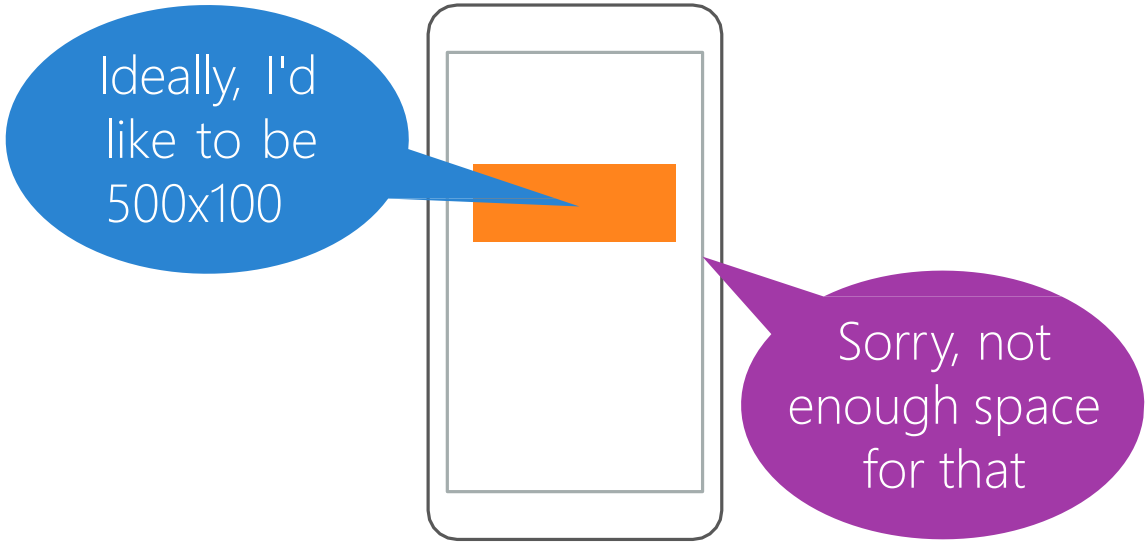


StackLayout Grid AbsoluteLayout RelativeLayout

Covered in this course

Sizing collaboration

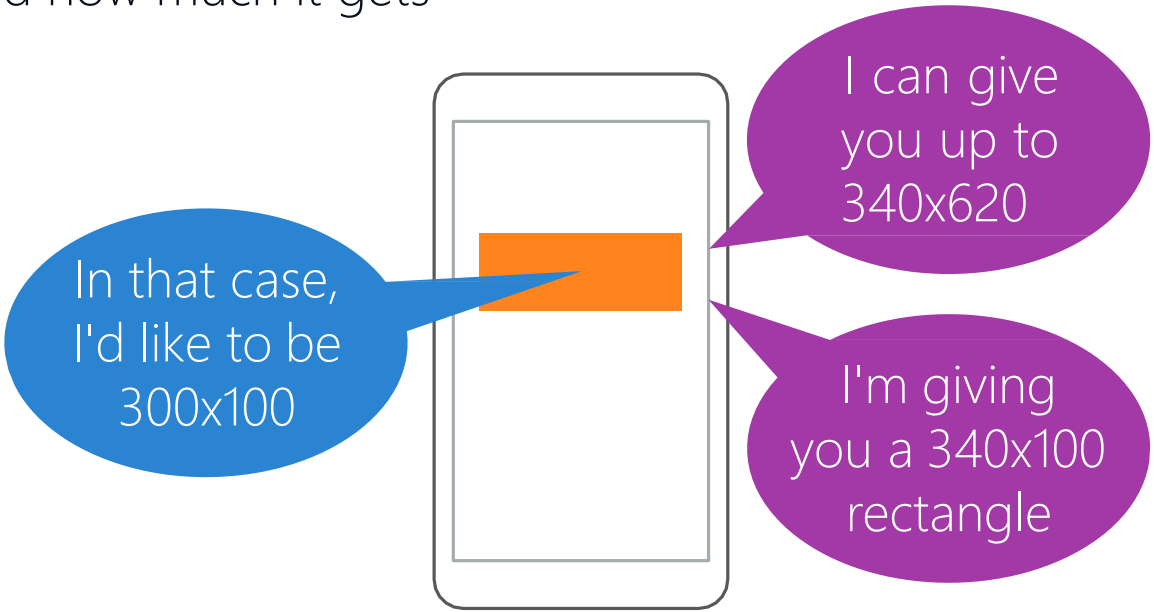
- The rendered size of a view is a collaboration between the view itself and its layout container



View Layout

Layout algorithm

- Layout panel asks each child how much room it would like, but then tells each child how much it gets

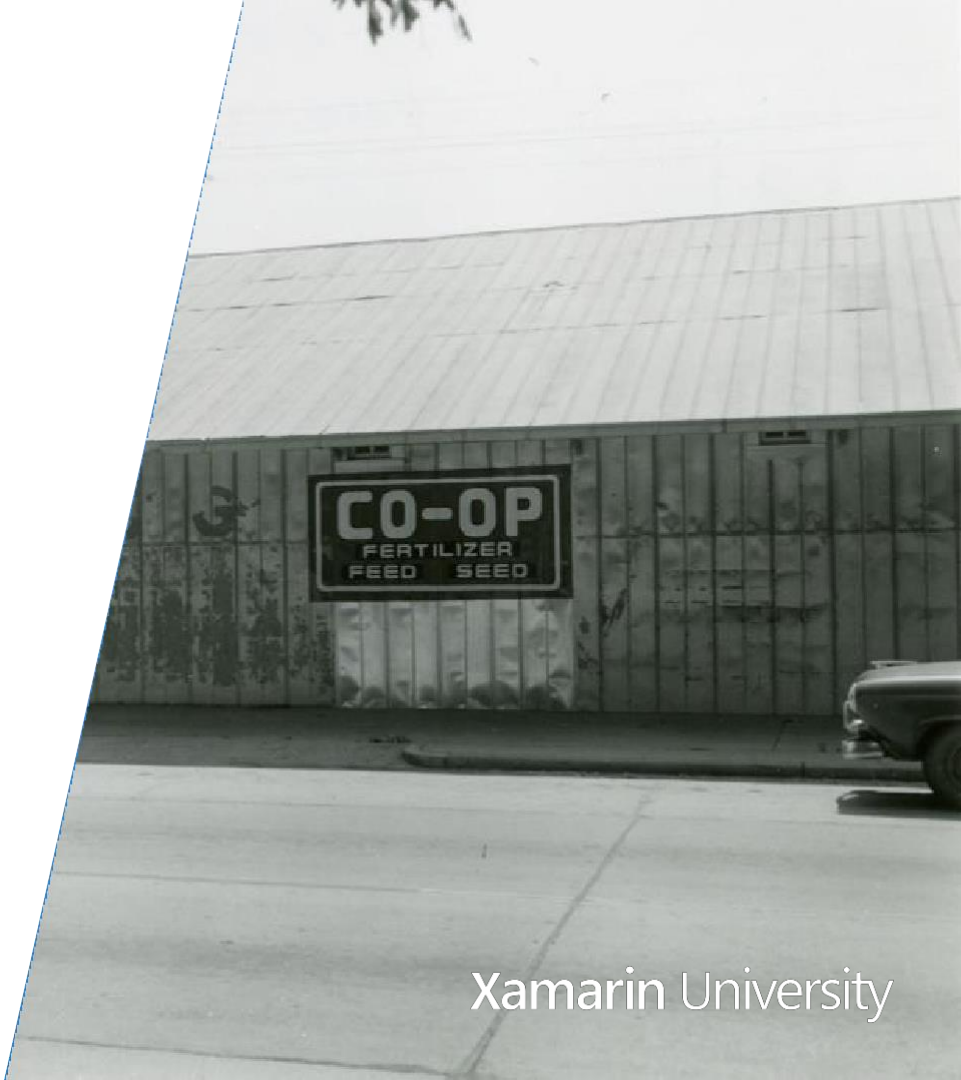


View Layout

Specify the size of a view

Tasks

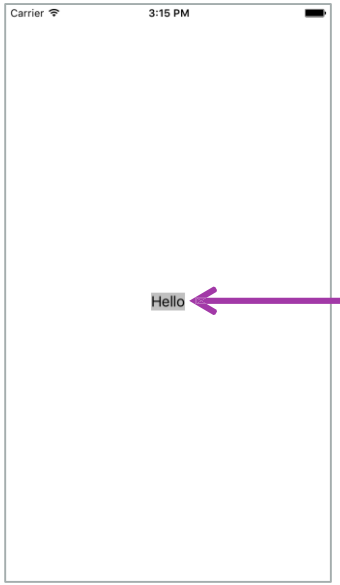
1. Specify preferred size of an Element
2. Set layout options



Default view sizing

- By default, most views try to size themselves just large enough to hold their content (we will see other factors that influence size)

```
<Label
    Text="Hello"
    BackgroundColor="Silver"
    VerticalOptions="Center"
    HorizontalOptions="Center"   />
```

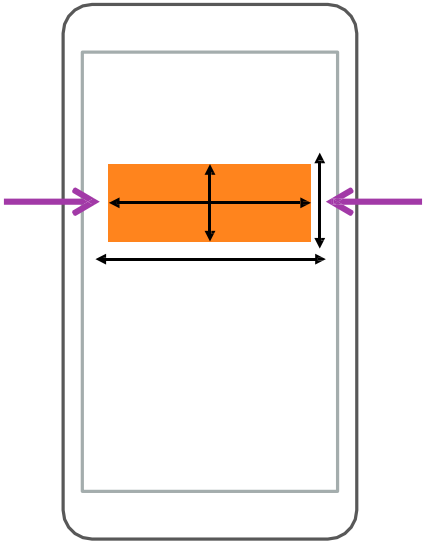


E.g. by default **Label**s are sized based on their text

View preferences

- A view has four properties that influence its rendered size; they are all requests and may be overruled by the layout container

`WidthRequest`
`HeightRequest`
(preferred size)



`VerticalOptions`
`HorizontalOptions`
(preferred position in the rectangle allocated by the container)

Sizing requests

- A view can request a desired width and height

Preferred size is stored in the view, but read and interpreted by its layout container

```
<Label
    Text="Hello"
    WidthRequest="100"
    HeightRequest="300"
    BackgroundColor="Silver" />
```

Size units

- Explicit sizes in Xamarin.Forms have no intrinsic units; the values are interpreted by each platform according to that platform's rules

Effective pixels in UWP

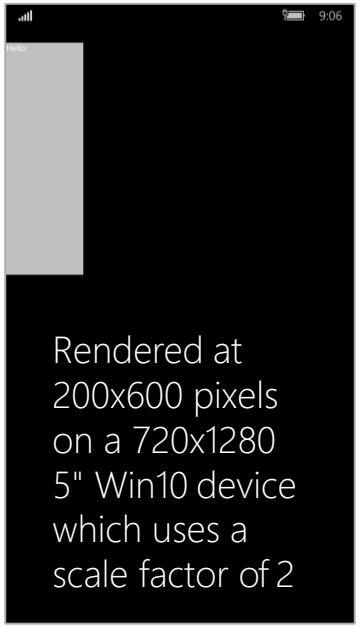
Points in iOS

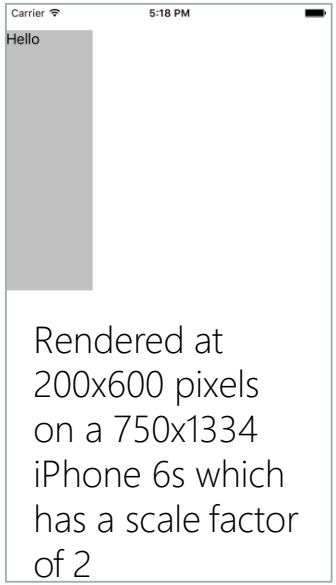
Density-independent pixels in Android

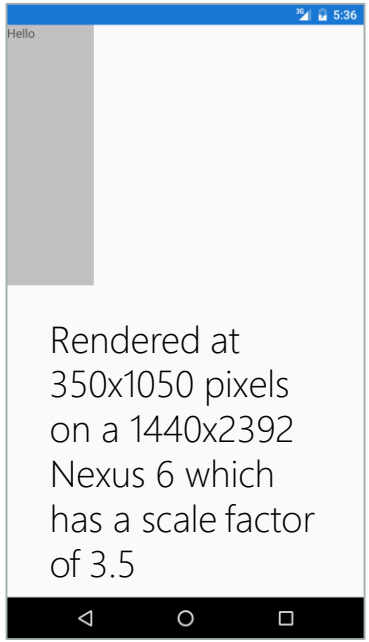
Platform rendering

- Sizes set in Xamarin.Forms are passed to the underlying platform; the platform will scale the values based on screen size and resolution

```
<Label
    Text="Hello"
    WidthRequest="100"
    HeightRequest="300"
    BackgroundColor="Silver" />
```





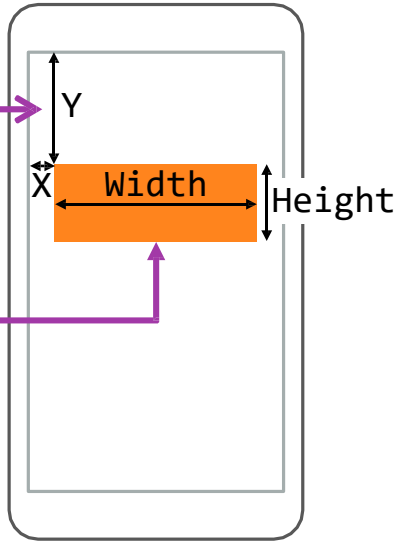


Reported sizes

- Visual elements report their size/location via properties that are set during layout; the values are expressed in platform-independent units

Coordinates of the view's rectangle within the parent

Bounds reports all four values in one **Rectangle** object



Size of the view

Layout requests

- A view can specify layout requests

```
public class View : ...
{
    public LayoutOptions HorizontalOptions { get; set; }
    public LayoutOptions VerticalOptions   { get; set; }
    ...
}
```

Layout preferences are stored in the view, but read and interpreted by the layout container

What are LayoutOptions?

- The **`LayoutOptions`** struct encapsulates two layout preferences

```
public struct LayoutOptions
{
    public LayoutAlignment Alignment { get; set; }
    public bool            Expands   { get; set; }
    ...
}
```

```
public enum LayoutAlignment
{
    Start, Center, End, Fill
}
```

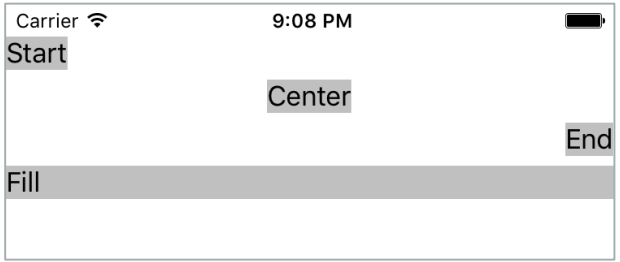
Location within the rectangle given by the container

Used only by **`StackLayout`**, indicates if the view would like extra space if available

Alignment

- A view's preferred alignment determines its position and size within the rectangle allocated for it by its container

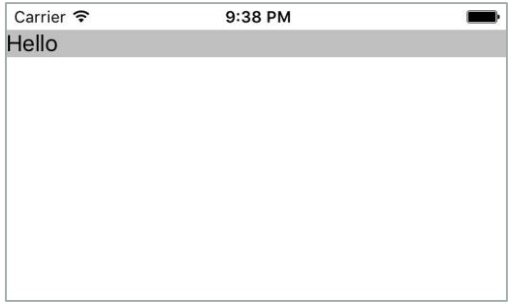
```
<StackLayout>
   <Label Text="Start"  HorizontalOptions="Start"  BackgroundColor="Silver" />
   <Label Text="Center" HorizontalOptions="Center" BackgroundColor="Silver" />
   <Label Text="End"    HorizontalOptions="End"    BackgroundColor="Silver" />
   <Label Text="Fill"   HorizontalOptions="Fill"   BackgroundColor="Silver" />
</StackLayout>
```



Size requests vs. Fill

- The **`Fill`** layout option generally overrides size preferences

```
<StackLayout>
   <Label Text="Hello"
      WidthRequest="100"
      HorizontalOptions="Fill"
      BackgroundColor="Silver" />
</StackLayout>
```



`Fill` causes **`WidthRequest`** to be ignored here

Alignment default

- Horizontal and vertical alignment options generally default to **`Fill`**

```
<Label Text="Hello" HorizontalOptions="Fill" VerticalOptions="Fill" />
```

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

The declaration of these labels is equivalent because of the defaults

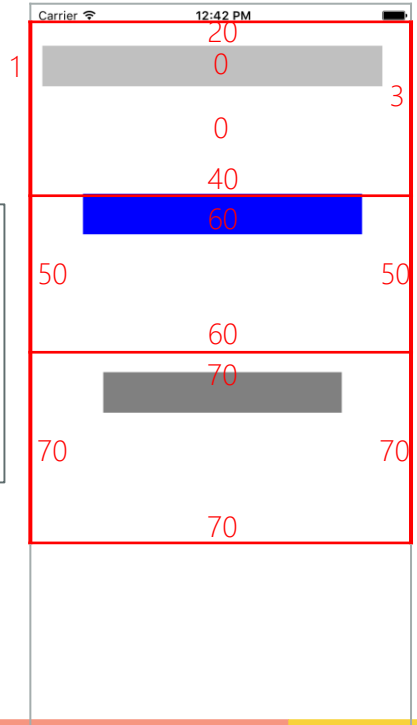
Group Exercise

Explore alignment options

Margin

- *Margin* is extra space around the outside of a view (available in all views, including containers)

```
<StackLayout>
    <BoxView Color="Silver" Margin="10,20,30,40"/>
    <BoxView Color="Blue"   Margin="50,60"/>
    <BoxView Color="Gray"   Margin="70"/>
</StackLayout>
```



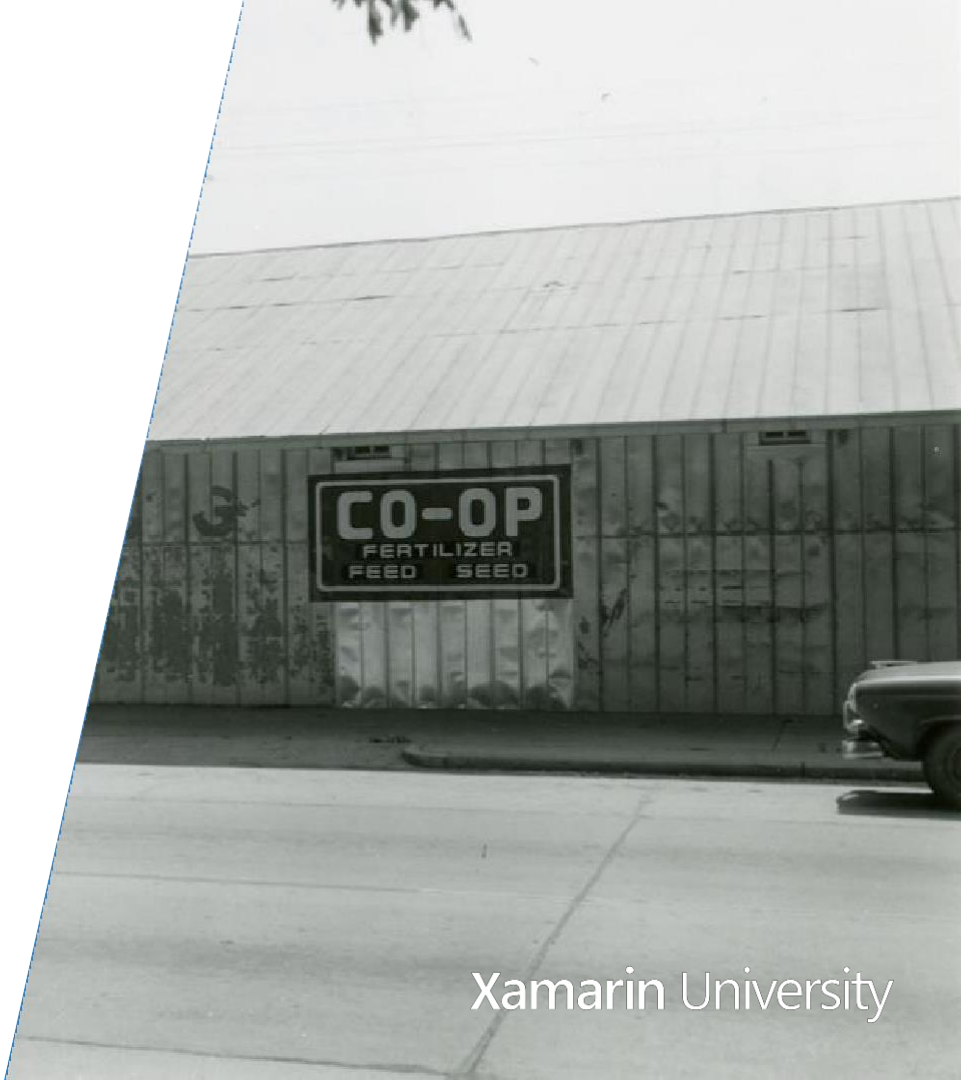
Padding

- *Padding* is extra space on the inside of a layout that creates a gap between the children and the layout itself (applicable only to layouts)

```
<StackLayout Padding="20,40,60,80">
    ...
</StackLayout>
```

Summary

1. Specify preferred size of an Element
2. Set layout options



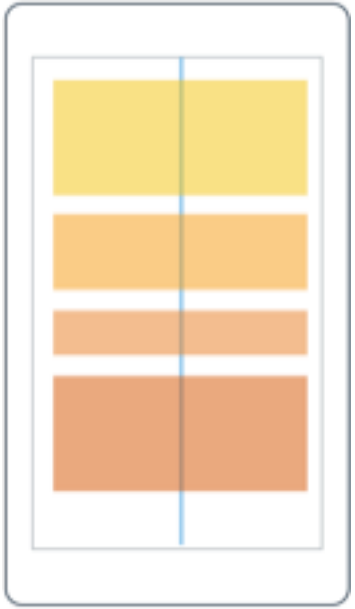
Arrange views with StackLayout

Tasks

1. Add views to a **`StackLayout`** in code and XAML
2. Specify layout orientation
3. Use **`Expands`** to request extra space

What is StackLayout?

- `StackLayout` arranges its children in a single column or a single row



StackLayout children

- **`StackLayout`** holds a collection of child views

The views this panel will display

```
public abstract class Layout<T> : ...
{
    public IList<T> Children { get { ... } }
}
```

```
public class StackLayout : Layout<View>
{
}
```

Stores **`Views`**

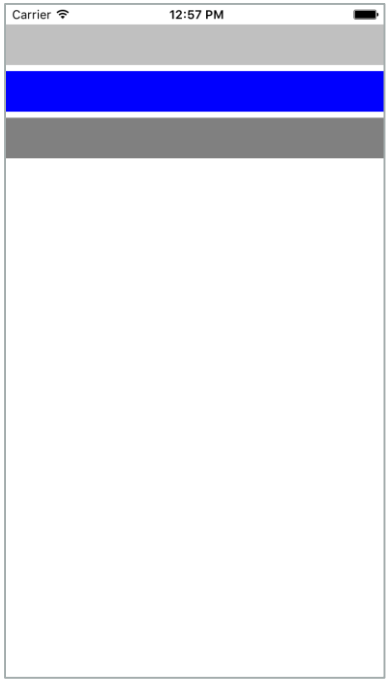
Adding children [code]

- You can add/remove children from a **`StackLayout`** using code

```
<StackLayout x:Name="stack" />
```

```
var a = new BoxView() { BackgroundColor = Color.Silver };
var b = new BoxView() { BackgroundColor = Color.Blue   };
var c = new BoxView() { BackgroundColor = Color.Gray   };

stack.Children.Add(a);
stack.Children.Add(b);
stack.Children.Add(c);
```

Dynamically add views to the panel

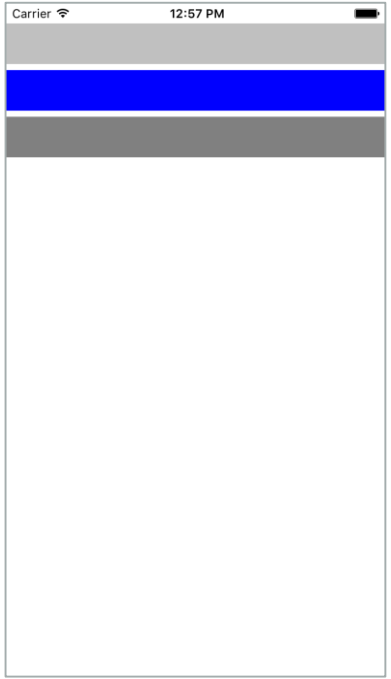


Adding children [XAML]

- You can add children to a **`StackLayout`** in XAML

Views are added to the **`Children`** collection because that is the Content property

```
<StackLayout>
    <BoxView Color="Silver" />
    <BoxView Color="Blue"   />
    <BoxView Color="Gray"   />
</StackLayout>
```



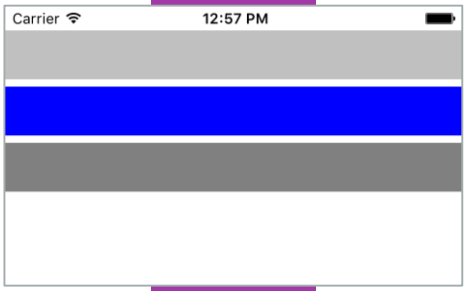
This course will prefer XAML because it is more common than code.

Child ordering

- Child layout order is determined by the order they were added to the **`Children`** collection (applies to both code and XAML)

Textual order determines layout order

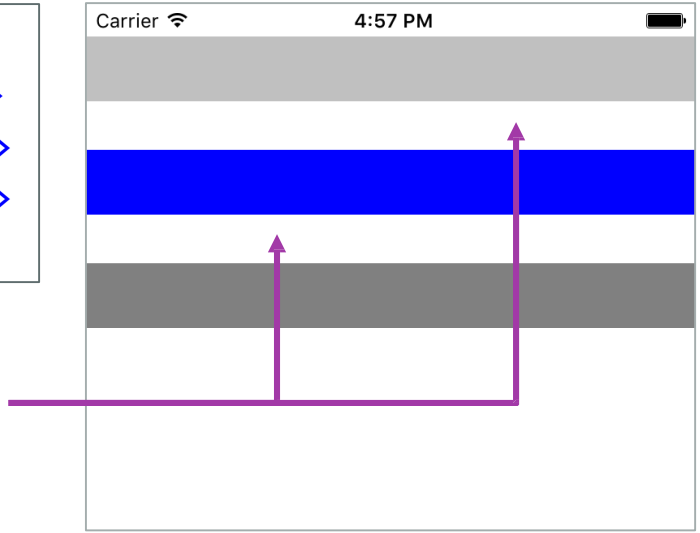
```
<StackLayout>
    <BoxView Color="Silver" />
    <BoxView Color="Blue"   />
    <BoxView Color="Gray"   />
</StackLayout>
```



StackLayout child spacing

- **`StackLayout`**'s **`Spacing`** separates the children (the default is 6)

```
<StackLayout Spacing="30">
    <BoxView Color="Silver" />
    <BoxView Color="Blue"   />
    <BoxView Color="Gray"   />
</StackLayout>
```



Space added between every child

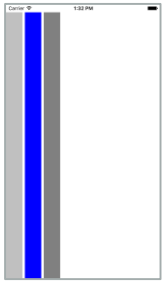
StackLayout orientation

- **`StackLayout`**'s **`Orientation`** property lets you choose a vertical column or a horizontal row

Vertical is the default →

```
<StackLayout Orientation="Vertical">
    <BoxView Color="Silver" />
    <BoxView Color="Blue"   />
    <BoxView Color="Gray"   />
</StackLayout>
```

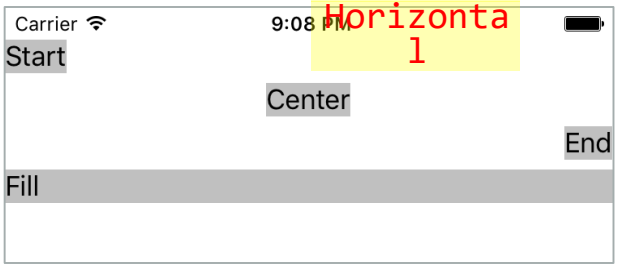
```
<StackLayout Orientation="Horizontal">
    <BoxView Color="Silver" />
    <BoxView Color="Blue"   />
    <BoxView Color="Gray"   />
</StackLayout>
```



LayoutOptions against orientation

- In the direction *opposite* of its orientation, **`StackLayout`** uses the **`Start`**, **`Center`**, **`End`**, and **`Fill`** layout options

```
<StackLayout Orientation="Vertical">
   <Label ...  HorizontalOptions="Start"  />
   <Label ...  HorizontalOptions="Center" />
   <Label ...  HorizontalOptions="End"    />
   <Label ...  HorizontalOptions="Fill"   />
</StackLayout>
```



These *horizontal* options are used by a *vertical* **`StackLayout`**

LayoutOptions with orientation

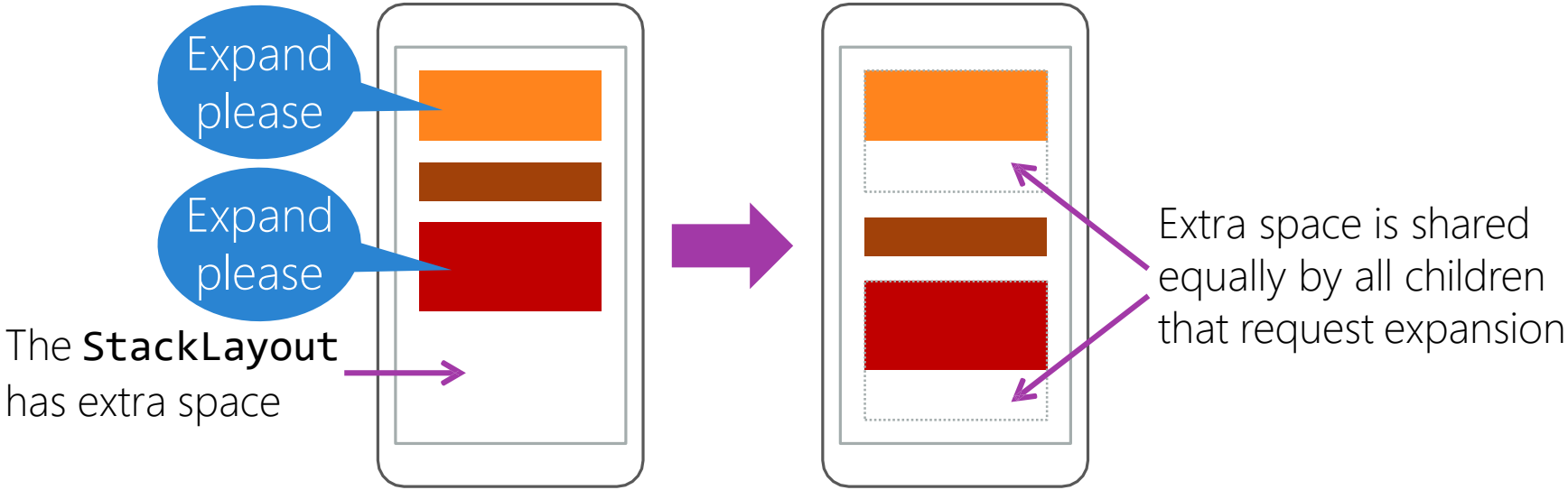
- In the direction of its orientation, **`StackLayout`** ignores the **`Start`**, **`Center`**, **`End`**, and **`Fill`** layout options

```
<StackLayout Orientation="Vertical">
   <Label ... VerticalOptions="Start"  />
   <Label ... VerticalOptions="Center" />
   <Label ... VerticalOptions="End"    />
   <Label ... VerticalOptions="Fill"   />
</StackLayout>
```

These *vertical* options are ignored by a *vertical* **`StackLayout`**

What is expansion?

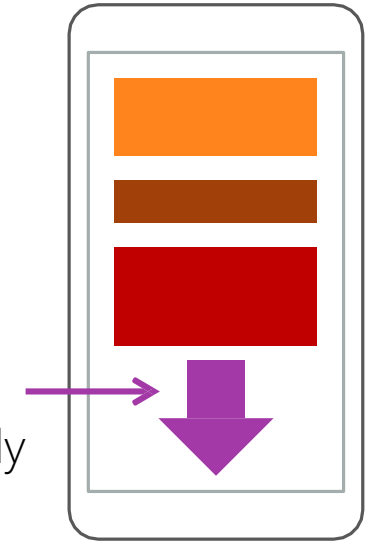
- A view's *expansion* setting determines whether it would like the **`StackLayout`** to allocate available extra space to its rectangle



Expansion direction

- `StackLayout` expands children only in the direction of its orientation

E.g. a vertical
`StackLayout`
expands vertically



How much extra space?

- **`StackLayout`** determines the amount of extra space using its standard layout calculation as if there were no expansion

```
<StackLayout Orientation="Vertical">
   <Label Text="One"   HeightRequest="100" ... />
   <Label Text="Two"                       ... />
   <Label Text="Three" HeightRequest="50"  ... />
</StackLayout>
```

Uses requested size if provided or "default" size if not

How to specify expansion?

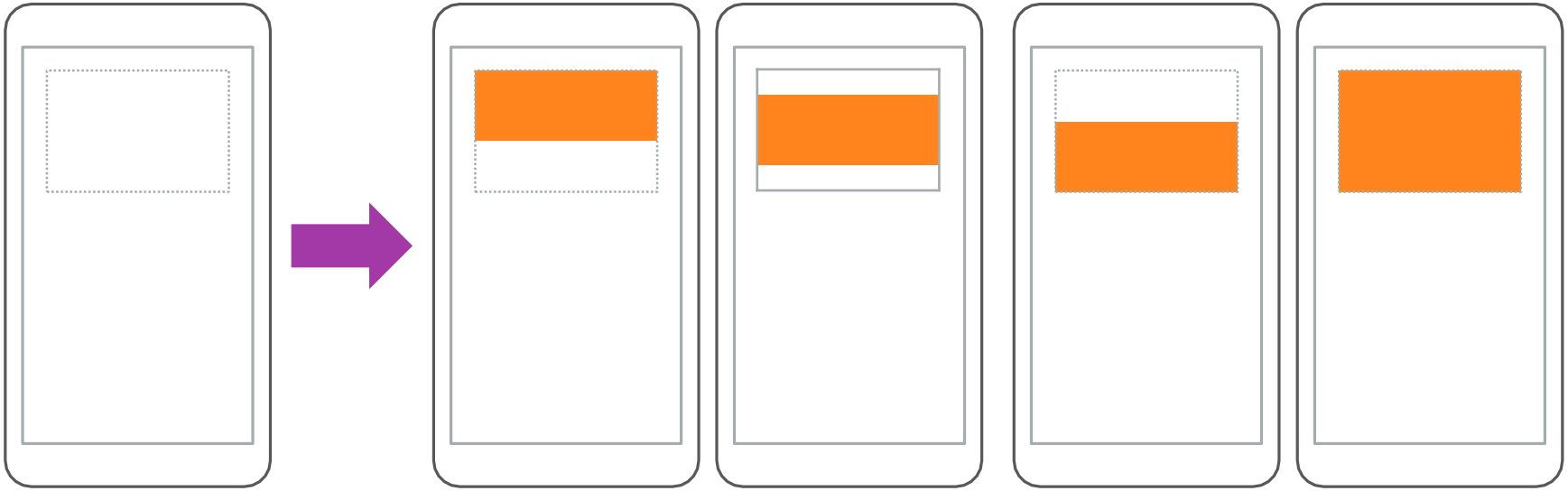
- To request expansion, use the "**...AndExpand**" version of the layout options in the direction of the **StackLayout**'s orientation

```
<StackLayout Orientation="Vertical">
   <Label ...  VerticalOptions="StartAndExpand"   />
   <Label ...  VerticalOptions="CenterAndExpand"  />
   <Label ...  VerticalOptions="EndAndExpand"     />
   <Label ...  VerticalOptions="FillAndExpand"    />
</StackLayout>
```

These settings give a **LayoutOptions** instance with **Expands** set to **true**

Expansion vs. view size

- Enabling expansion can change the size of the view's layout rectangle, but doesn't change the size of the view unless it uses `FillAndExpand`



No expansion against orientation

- In the direction *opposite* of its orientation, adding "**...AndExpand**" to the layout options has no effect (there is no expansion in that direction)

```
<StackLayout Orientation="Vertical">
   <Label ... HorizontalOptions="Start"            />
   <Label ... HorizontalOptions="StartAndExpand"   />
   <Label ... HorizontalOptions="Center"           />
   <Label ... HorizontalOptions="CenterAndExpand" />
   <Label ... HorizontalOptions="End"              />
   <Label ... HorizontalOptions="EndAndExpand"     />
   <Label ... HorizontalOptions="Fill"             />
   <Label ... HorizontalOptions="FillAndExpand"    />
</StackLayout>
```

Same

Same

Same

Same

Individual Exercise

Use StackLayout to build a UI

Summary

1. Add views to a **`StackLayout`** in code and XAML
2. Specify layout orientation
3. Use **`Expands`** to request extra space

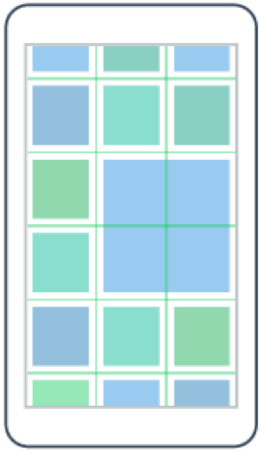
Apply Attached Properties

Tasks

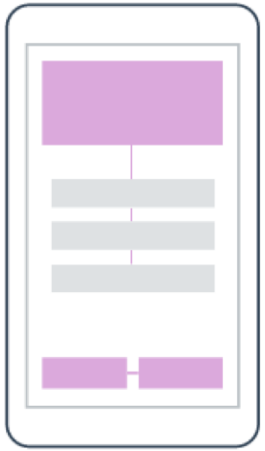
1. Apply an Attached Property in code
2. Apply an Attached Property in XAML

Motivation

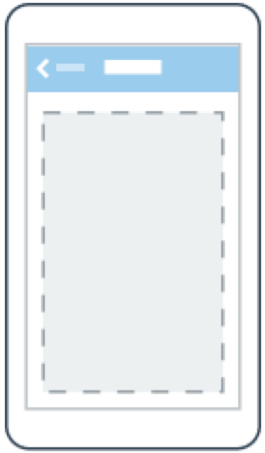
- Some properties are only needed in specific situations



Row/column needed when in a **`Grid`**



Constraints needed when in a **`RelativeLayout`**



Request for a back button needed when in a **`NavigationPage`**

Union is a bad solution

- Do not mix all potential properties into a base class; it would make each object larger and the base class harder to understand

Needed when in a **Grid** layout

Needed when in a **RelativeLayout**

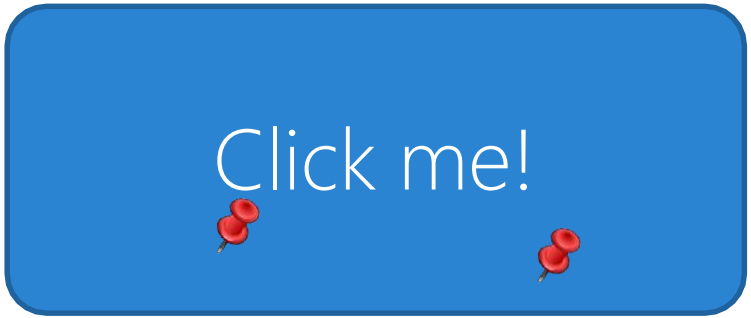
Needed when in a **NavigationPage**

```
public class MyBaseClass
{
    public int Row    { get; set; }
    public int Column { get; set; }

    public Constraint WidthConstraint  { get; set; }
    public Constraint HeightConstraint { get; set; }

    public bool HasBackButton { get; set; }
}
```

What is an attached property?

- An *attached property* is a property that is defined in one class but set on objects of other types

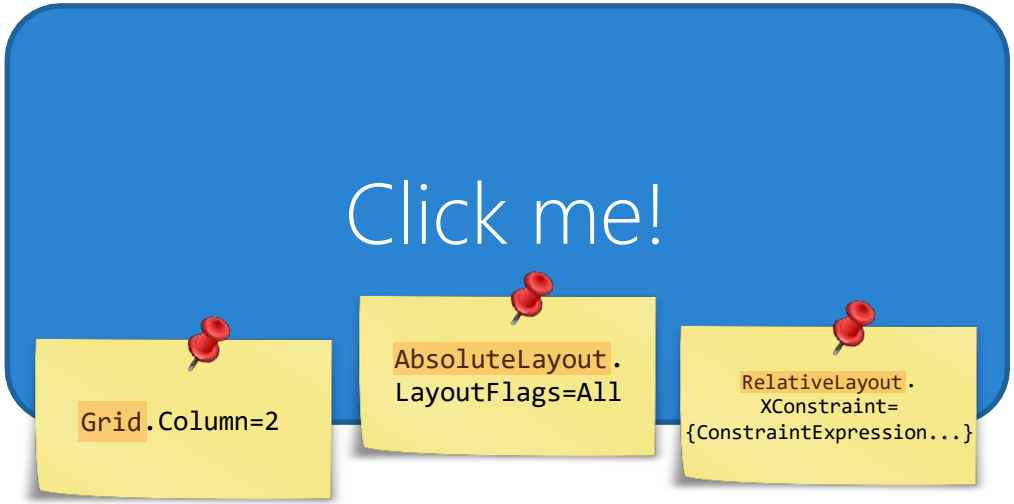
`Button` does not have **`Row`**/**`Column`** properties



They are defined in **`Grid`** and attached to objects of other types as needed

Multiple attached properties

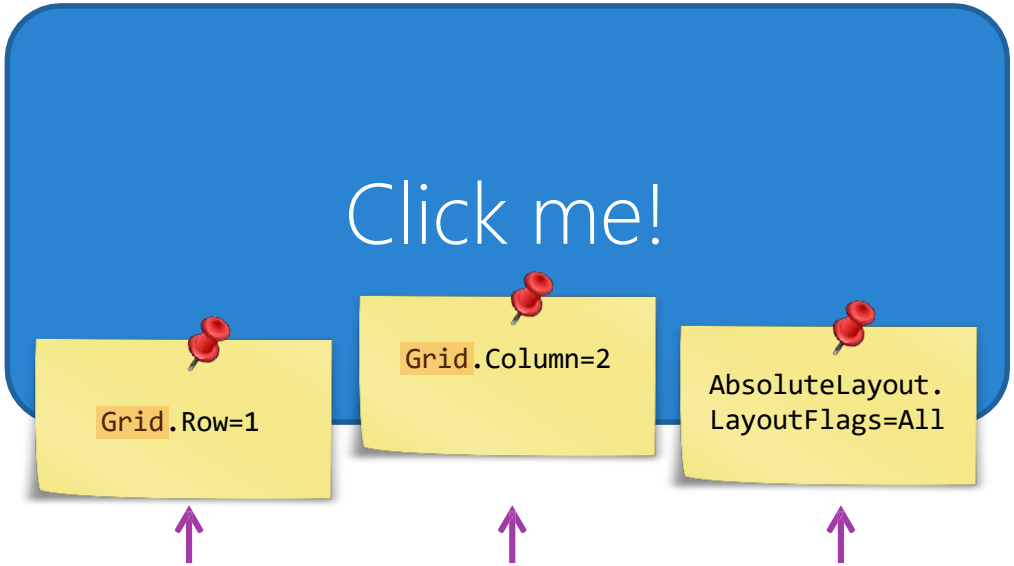
- You can attached properties from multiple classes to an object



Who consumes attached properties?

- Typically, a container will look for attached properties on its children

When this button is in **Grid**…



…the grid reads the attached properties it needs…and ignores the others

Attached property infrastructure

- Support for creating attached properties is built-in to Xamarin.Forms

Registration

```
public sealed class BindableProperty
{  ...
   public static BindableProperty CreateAttached(...) { ... }
}
```

Value storage

```
public abstract class BindableObject : ...
{  ...
   public object GetValue(BindableProperty property) { ... }
   public void   SetValue(BindableProperty property, object value) { ...
}                        }
```

How to define an attached property

- The owner of an attached property defines the property and access methods

```
public partial class Grid : Layout<View>
{  ...
   public static readonly BindableProperty RowProperty = BindableProperty.CreateAttached(...);

   public static int  GetRow(BindableObject bindable) { ... }
   public static void SetRow(BindableObject bindable, int value) { ... }
}
```

Get/set methods

The property definition (the **Property** suffix is used by convention)

Apply an attached property in code

- In code, use the static **`Set`** method to apply an attached property

Attach row and column settings to a button

```
var button = new Button();

Grid.SetRow    (button, 1);
Grid.SetColumn(button, 2);
```

```
public partial class Grid : Layout<View>
{  ...
   public static readonly BindableProperty RowProperty = BindableProperty.CreateAttached(...);

   public static int  GetRow(BindableObject bindable) { ... }
   public static void SetRow(BindableObject bindable, int value) { ... }
}
```

Apply an attached property in XAML

- In XAML, use the owning class name and the attached property name (without the **`Property`** suffix)

Attach row and column settings to a button

```
<Button Grid.Row="1" Grid.Column="2" ... />
```

```
public partial class Grid : Layout<View>
{  ...
   public static readonly BindableProperty RowProperty = BindableProperty.CreateAttached(...);

   public static int  GetRow(BindableObject bindable) { ... }
   public static void SetRow(BindableObject bindable, int value) { ... }
}
```

Summary

1. Apply an Attached Property in code
2. Apply an Attached Property in XAML

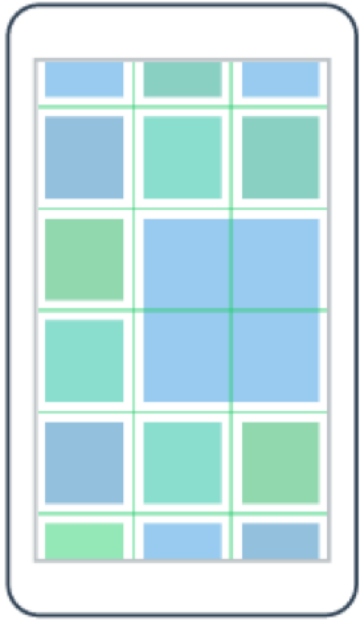
Arrange views with Grid

Tasks

1. Specify grid row and column sizes
2. Add children to grid cells

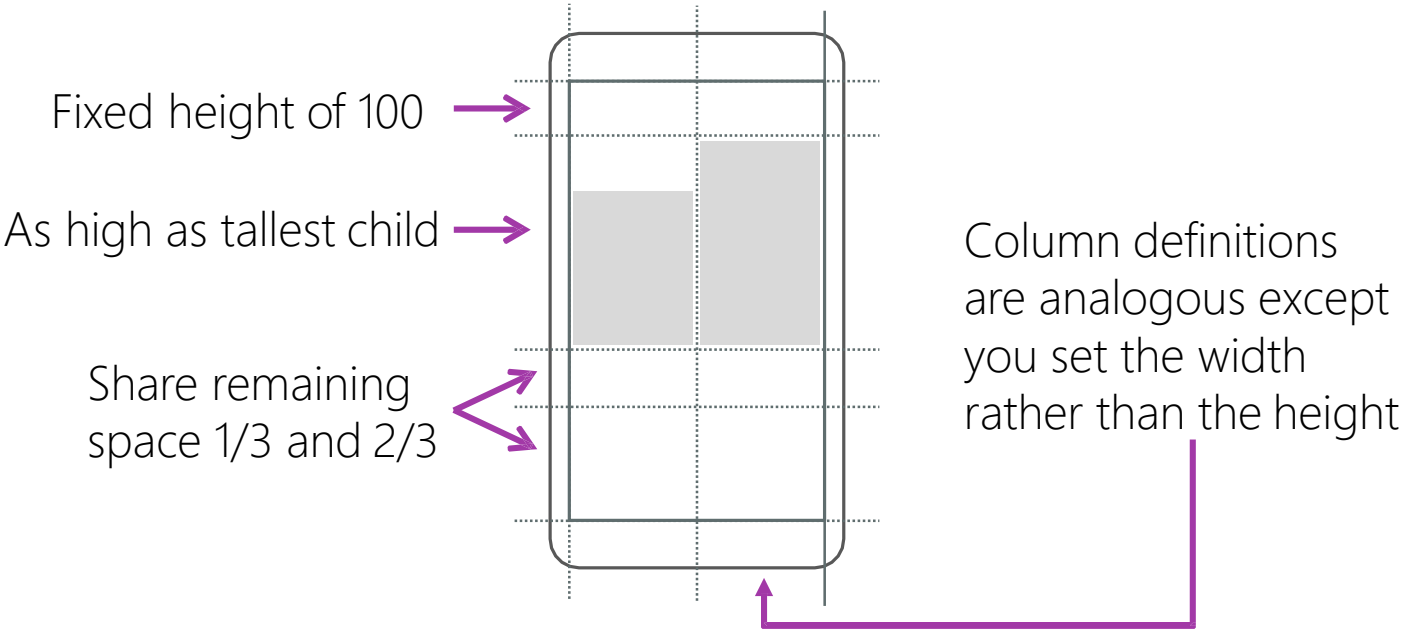
What is Grid?

- **`Grid`** places its children into cells formed from rows and columns



Grid rows/columns

- You specify the shape of the grid by defining each row and column individually



Row/column definitions

- There are dedicated classes that define a row or a column

Specify row height

```
public sealed class RowDefinition : ...
{   ...
    public GridLength Height { get; set; }
}
```

Specify column width

```
public sealed class ColumnDefinition : ...
{   ...
    public GridLength Width { get; set; }
}
```

What is GridLength?

- **`GridLength`** encapsulates two things: unit and value

```
public struct GridLength
{   ...
    public GridUnitType GridUnitType { get; }
    public double       Value        { get; }
}
```

Units can be: **`Absolute`**, **`Auto`**, **`Star`**

Absolute GridLength

- **`Absolute GridLength`** specifies a fixed row height or column width

```
var row = new RowDefinition() { Height = new GridLength(100) };
```

```
<RowDefinition Height="100" />
```

Value is in platform-independent units

Auto GridLength

- **`Auto GridLength`** lets the row height or column width adapt, it automatically becomes the size of the largest child

```
var row = new RowDefinition() {Height = new GridLength(1, GridUnitType.Auto)};
```

```
<RowDefinition Height="Auto" />
```

Value is irrelevant for **`Auto`**, it is typical to use 1 as the value when creating in code

Star GridLength

- **`Star GridLength`** shares the available space proportionally among all rows/columns that use star sizing

```
var row = new RowDefinition() { Height = new GridLength(2.5, GridUnitType.Star) };
```

```
<RowDefinition Height="2.5*" />
```

XAML type converter uses `*` instead of the **`Star`** used in code.
Note: **`"1*"`** and **`"*"`** are equivalent in XAML.

Grid row/column collections

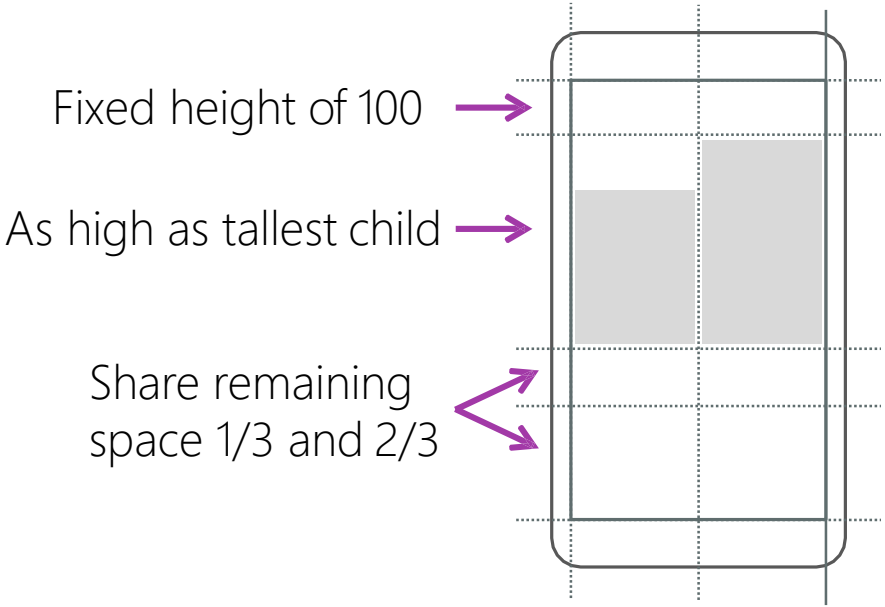
- **Grid** contains collections for the row and column definitions

```
public partial class Grid : Layout<View>
{
    ...
    public ColumnDefinitionCollection ColumnDefinitions { get; set; }
    public RowDefinitionCollection    RowDefinitions    { get; set; }
}
```

You add items to these collections
to create the rows/columns

Grid example

- It is common to mix different **`GridLength`** settings in the same grid



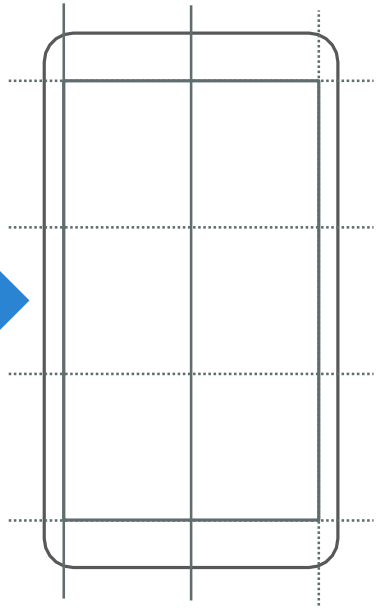

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition Height="100"  />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="1*"   />
      <RowDefinition Height="2*"   />
   </Grid.RowDefinitions>
   ...
</Grid>
```

Default size

- Rows and columns default to **"1*"** size

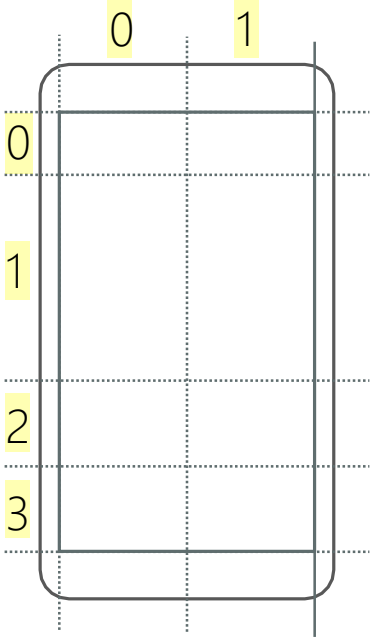
```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>

   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>
   ...
</Grid>
```

Yields a uniform 3x2 grid



Row/column numbering

- The row/column numbering starts at 0



Grid positioning properties

- **`Grid`** defines four attached properties used to position children

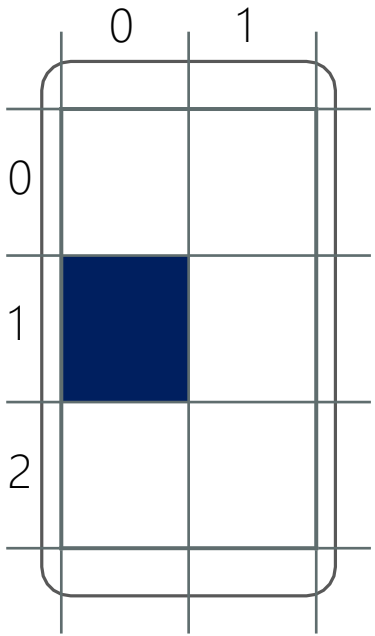
ATTACHED PROPERTY	VALUE
Column	An integer that represents the Column in which the item will appear.
ColumnSpan	An integer that represents the number of Columns that the item will span.
Row	An integer that represents the row in which the item will appear.
RowSpan	An integer that represents the number of rows that the item will span.

Cell specification

- Apply the **`Row`** and **`Column`** attached properties to each child

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>

   <BoxView Grid.Row="1" Grid.Column="0"
            BackgroundColor="Navy" />
</Grid>
```

Specify row/ column

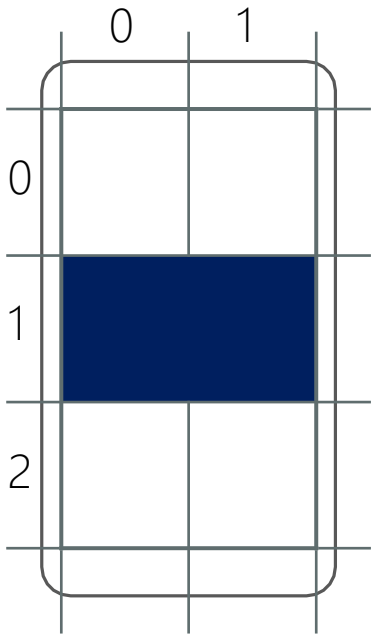


Span specification

- Apply **`RowSpan`** and **`ColumnSpan`** to each child as needed

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>

   <BoxView Grid.Row="1" Grid.Column="0"
            Grid.ColumnSpan="2"
            BackgroundColor="Navy" />
</Grid>
```

Specify span

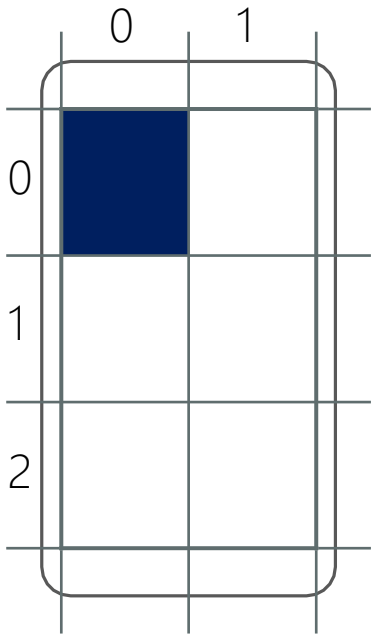


Cell and span defaults

- Cell locations default to 0 and spans default to 1

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>

   <BoxView BackgroundColor="Navy" />
</Grid>
```

Placed in cell (0,0)

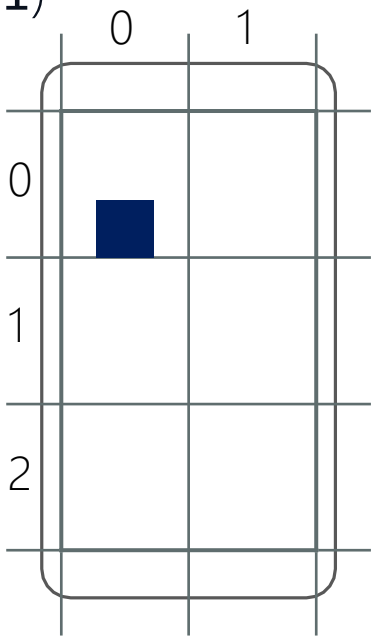


Layout options

- A view's horizontal and vertical layout options control how it is sized and positioned within its grid cell (the default is **`Fill`**)

Position within the cell

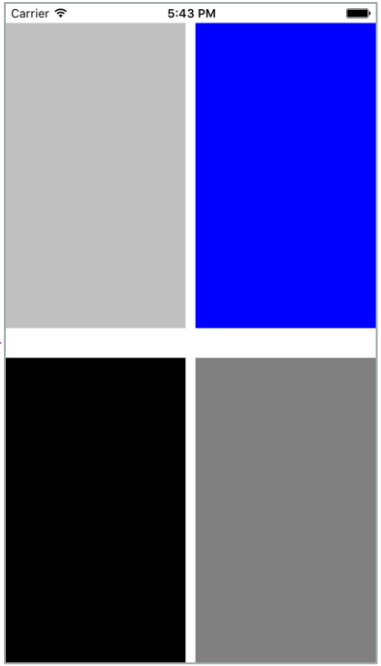
```
<Grid>
    ...
    <BoxView HorizontalOptions="Center"
             VerticalOptions="End"
             BackgroundColor="Navy"
             WidthRequest="50"
             HeightRequest="50"  />
</Grid>
```



Grid child spacing

- **`Grid`**'s **`RowSpacing`** and **`ColumnSpacing`** properties separate the children (they both default to 6)

```
<Grid RowSpacing="30" ColumnSpacing="10">
   <BoxView Color="Silver" Grid.Row="0" Grid.Column="0" />
   <BoxView Color="Blue"   Grid.Row="0" Grid.Column="1" />
   <BoxView Color="Black"  Grid.Row="1" Grid.Column="0" />
   <BoxView Color="Gray"   Grid.Row="1" Grid.Column="1" />
</Grid>
```



Row spacing

Column spacing

Grid Children

- **Grid** redefines its **Children** to use a custom list that provides several overloaded **Add** methods

```
public partial class Grid : Layout<View>
{
    ...
    public new IGridList<View> Children { get { ... } }
}
```

This property hides the inherited one

Can specify row/column when adding children

Add children programmatically

- **IGridList** provides several **Add** methods that are more specialized than typically found in a list

```
var grid = new Grid();
int row, column;
...
grid.Children.Add(label, column, row);

grid.Children.Add(button, column, column+1, row, row+2);
```

Yields a **ColumnSpan** of 1

Yields a **RowSpan** of 2

Auto-generated rows/columns

- **`Grid`** will automatically generate equal-sized rows/columns based on the position of the children you add

```
<Grid>
   <Button Grid.Row="1" Grid.Column="1" Text="OK"     />
   <Button Grid.Row="2" Grid.Column="0" Text="Cancel" />
</Grid>
```

Maximum index is 2 so the grid will have 3 rows

Maximum index is 1 so the grid will have 2 columns

Individual Exercise

Use Grid to build a UI

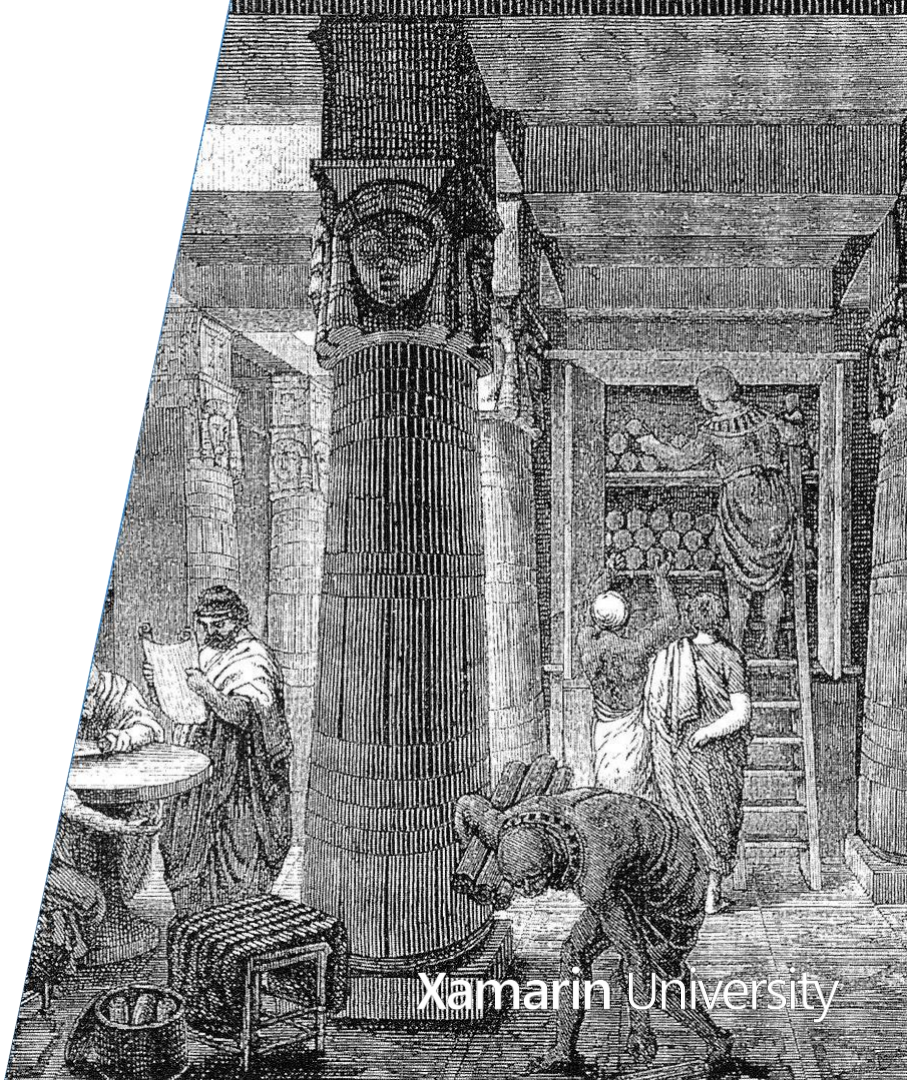
Summary

1. Specify grid row and column sizes
2. Add children to grid cells

Scroll a layout with ScrollView

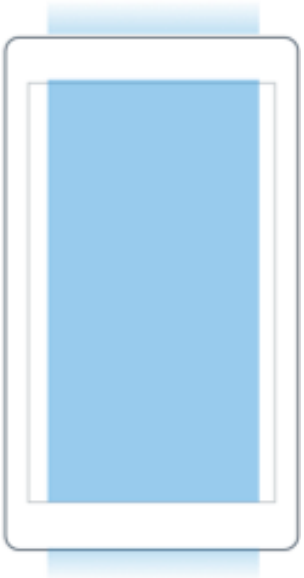
Tasks

1. Use `ScrollView` to add scrolling
2. Set the scroll direction



What is ScrollView?

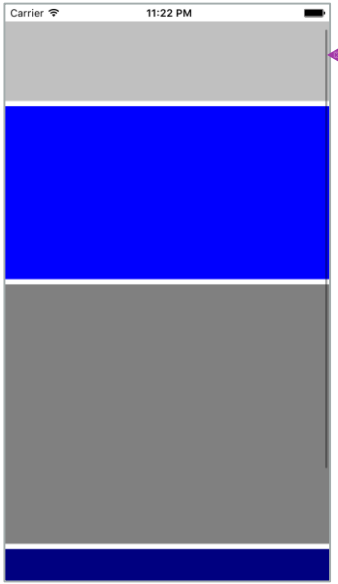
- `ScrollView` adds scrolling to a single piece of content; the content can be an individual view or a layout container



How to use ScrollView

- Wrap a **`ScrollView`** around a single element to add scrolling

```
<ScrollView>
   <StackLayout>
      <BoxView Color="Silver" HeightRequest="100" />
      <BoxView Color="Blue"   HeightRequest="200" />
      <BoxView Color="Gray"   HeightRequest="300" />
      <BoxView Color="Navy"   HeightRequest="200" />
   </StackLayout>
</ScrollView>
```



Scroll indicator

ScrollView orientation

- **`ScrollView`** lets you control the scroll direction: **`Vertical`** (the default), **`Horizontal`**, or **`Both`**

```
<ScrollView Orientation="Both">
   <Image Source="monkey.jpg"
          HeightRequest="1000"
          WidthRequest="1000" />
</ScrollView>
```

Image is larger than its container

Vertical indicator

Horizontal indicator

Do not nest scrolling views

- Generally, do not nest **`ScrollView`**s or a **`ListView`** in a **`ScrollView`**, it often creates non-intuitive behavior

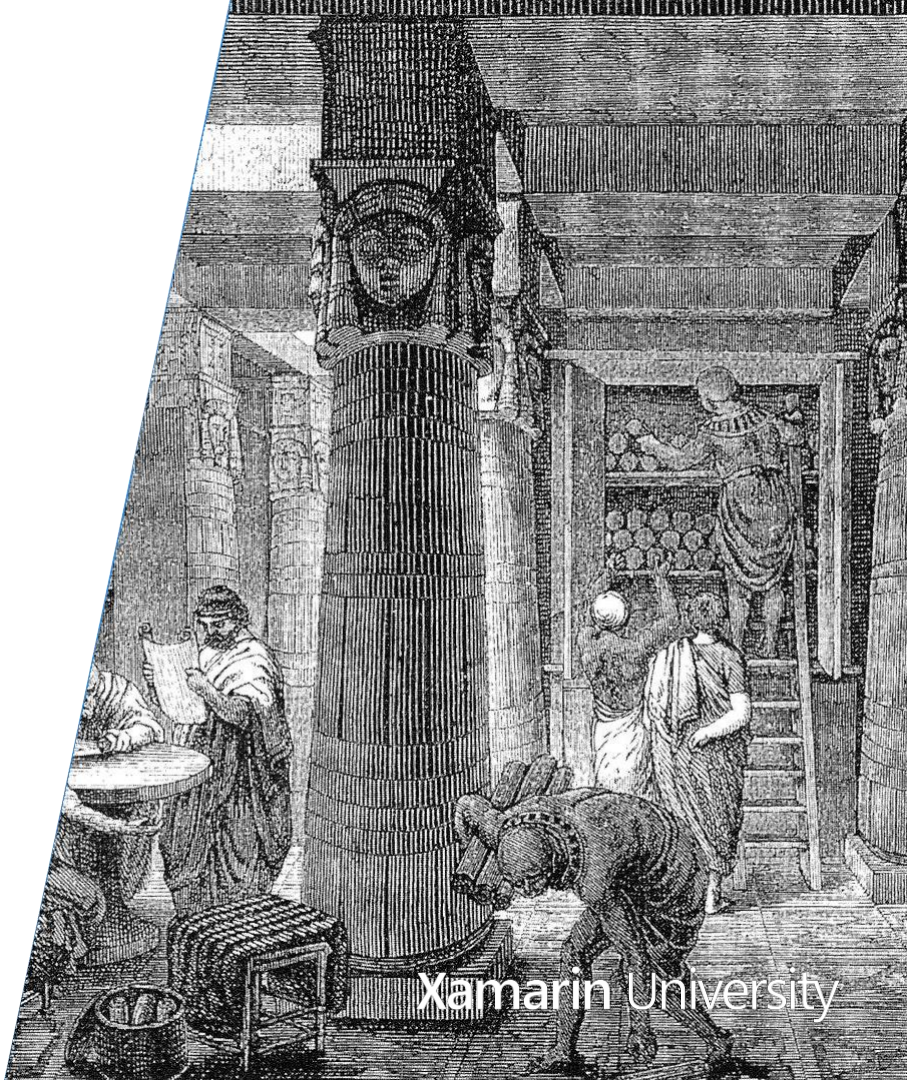
```
<ScrollView>
   <StackLayout>
      <Label Text="Prefix" />
      <Entry x:Name="prefix" />
      <ListView x:Name="listView" />
   </StackLayout>
</ScrollView>
```

Can scroll the entire **`ScrollView`**

Can scroll just the **`ListView`** content

Summary

1. Use `ScrollView` to add scrolling
2. Set the scroll direction



Thank You!

Microsoft