XAM312

Customizing the ListView in Xamarin Forms

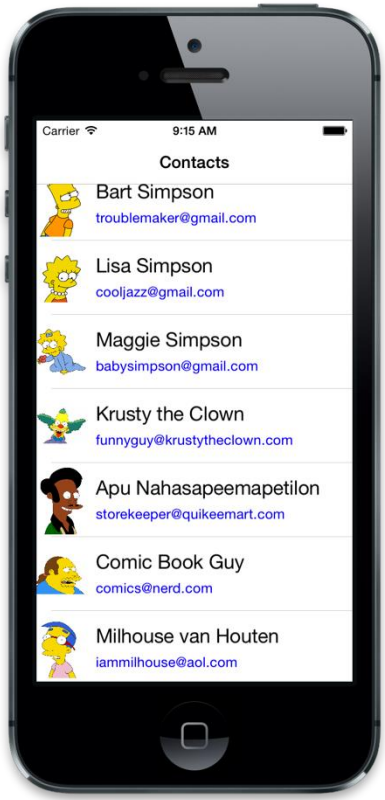
Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Creating custom cell definitions
2. Adding headers and footers
3. Separating your data into Groups
4. Performance Tuning your ListViews

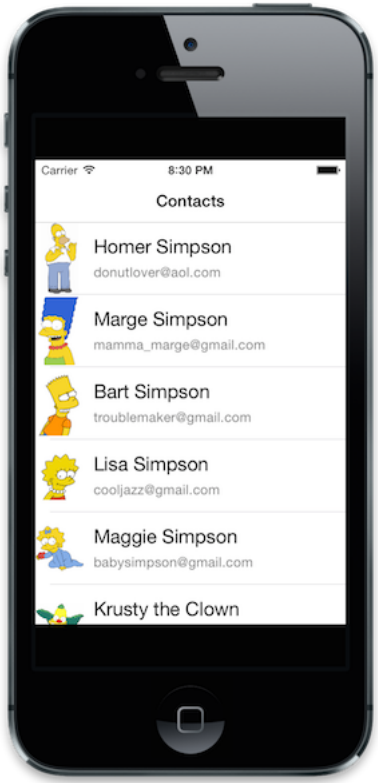


Creating Custom Cell Definitions

Xamarin University

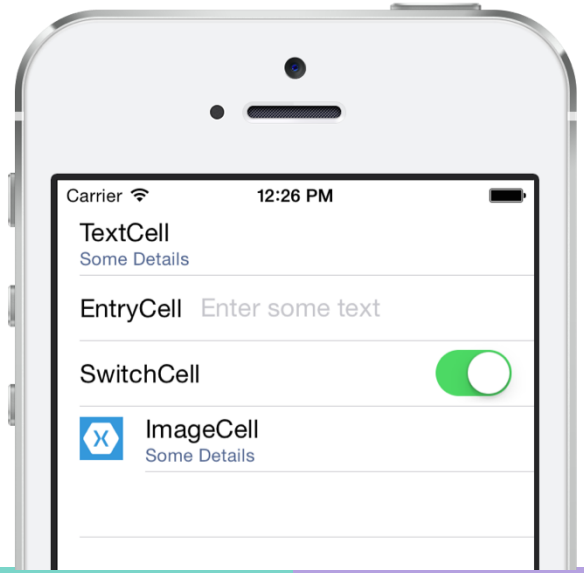
Tasks

1. Using ViewCells
2. Creating Data Templates in code
3. Creating Unique Row Visuals



Reminder: Cell Styles

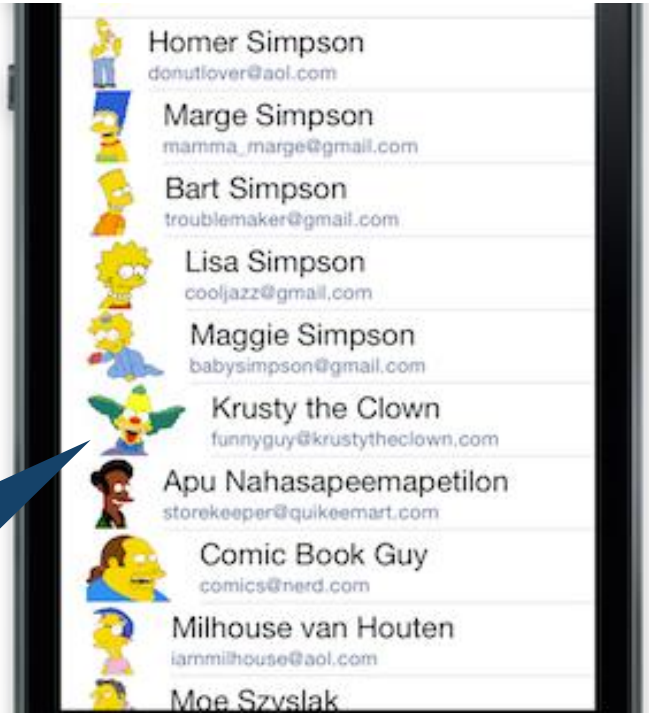
- `ListView` lets you define the visuals for each row through a cell style – there are several built-in variations



Customizing the cells

- Sometimes we need to customize the cell template
  - does not fit the data
  - need custom layout or colors
  - maybe you want something unique!

Images are different sizes and it pushes the text over – no way to control that in the `ImageCell`, would have to alter the image sizes which might not be possible



Introducing: ViewCell

- Can define a **`ViewCell`** to create a custom cell visualization of any type to display your data in a **`ListView`**

```
<DataTemplate>
   <ViewCell>
      <StackLayout Padding="5">
         <Label FontSize="20" TextColor="Black" Text="{Binding Name}" />
         <Label FontSize="14" TextColor="Blue" Text="{Binding Email}" />
      </StackLayout>
   </ViewCell>
</DataTemplate>
```

You define custom XAML to represent a row

Introducing: ViewCell

- Can define a **`ViewCell`** to create a custom cell visualization of any type to display your data in a **`ListView`**

```
<DataTemplate>
   <ViewCell>
      <StackLayout Padding="5">
         <Label FontSize="20" TextColor="Black" Text="{Binding Name}" />
         <Label FontSize="14" TextColor="Blue" Text="{Binding Email}" />
      </StackLayout>
   </ViewCell>
</DataTemplate>
```

`BindingContext` for the generated row will be a single item from the **`ItemsSource`**

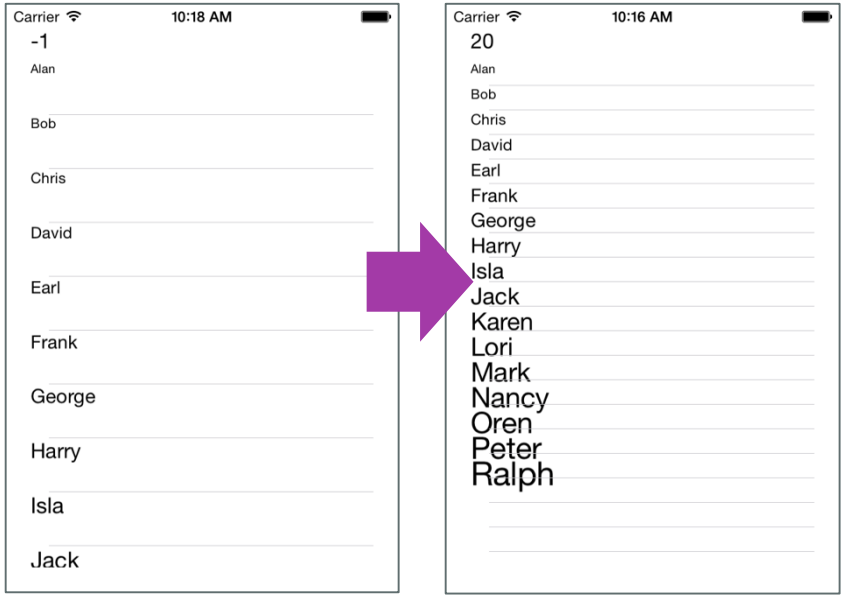
ViewCells in code

- Can define custom cells programmatically by deriving from **`ViewCell`**

```
public class NameViewCell : ViewCell
{
   public NameViewCell() {
      Label name = new Label();
      name.SetBinding(Label.TextProperty, new Binding("Name"));
      Switch toggle = new Switch();
      toggle.SetBinding(Switch.IsToggledProperty,
                        new Binding("Favorite"));
      View = new StackLayout { Children = { name, switch } };
   }
}
```

Controlling the row height

- By default, the **`ListView`** uses the same height for every cell – it's fixed in size for each default cell style
- For **`ViewCell`**, it will attempt to *estimate* the required height based on content; results will vary
- Can specify the height to be used for all rows explicitly by setting the **`ListView.RowHeight`** property



default behavior

`RowHeight` = 20

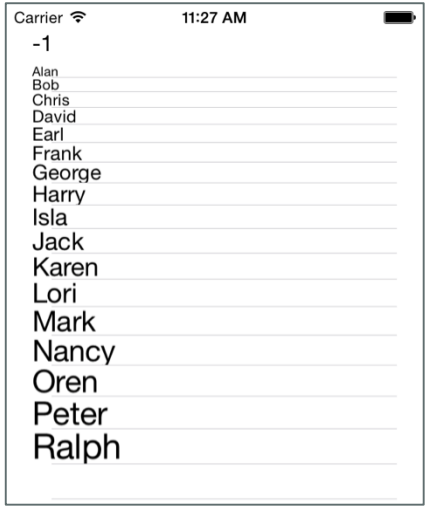
Individual Exercise

Providing a custom cell template for our ListView

Xamarin University

Variable-sized rows

- If the content size changes on a row-by-row basis, can set the **`HasUnevenRows`** property to get the **`ListView`** to size for each row
- Uses **`Cell.Height`** property instead of **`RowHeight`** if > 0
- Generally requires derived **`ViewCell`** class to set **`Height`** property based on the data




```
<ListView ... HasUnevenRows="True">
   <ListView.ItemTemplate>
      <DataTemplate>
         <local:TextViewCell />
      </DataTemplate>
   </ListView.ItemTemplate>
</ListView>
```

```
public class TextViewCell : ViewCell
{
   static MyFontSizeConverter fontConverter = new MyFontSizeConverter();

   public TextViewCell() {
      Label text = new Label();
      text.SetBinding(Label.TextProperty, new Xamarin.Forms.Binding("."));
      text.SetBinding(Label.FontSizeProperty,
             new Xamarin.Forms.Binding(".", converter: fontConverter));
      View = text;
   }

   protected override void OnBindingContextChanged() {
      base.OnBindingContextChanged();
      string text = BindingContext.ToString();
      Height = 10 + ((int)(text[0]) - 65);
   }
}
```

Runtime row resizing

- Individual **ListView** rows can be resized programmatically at runtime using the **ForceUpdateSize** method on the cell

```
void OnImageTapped(object sender, EventArgs args)
{
    var image = sender as Image;
    var viewCell = image.Parent.Parent as ViewCell;

    if (image.HeightRequest < 250)
    {
        image.HeightRequest = image.Height + 100;
        viewCell.ForceUpdateSize();
    }
}
```

Update the height of the child elements

Call **ForceUpdateSize** to update the cell

HasUnevenRows on the **ListView** must be set to **true** to resize at runtime

Using DataTriggers to customize rows

- By default, every row *shares* the same Data Template definition, can use *data triggers* in your template definition to change visuals at runtime

Default value assigned to property

```
<Label Text="{Binding Name}" TextColor="#008080">
   <Label.Triggers>
      <DataTrigger TargetType="Label" Binding="{Binding Gender}"
                   Value="Female">
         <Setter Property="TextColor" Value="#800080" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Trigger changes value based on binding evaluation

DataTemplateSelector

- Can use a **`DataTemplateSelector`** to provide a specific **`DataTemplate`** based on the model data being visualized

```
class CharacterSelector : Xamarin.Forms.DataTemplateSelector
{
    public DataTemplate LeadingCharacter { get; set; }
    public DataTemplate SupportingCharacter { get; set; }

    protected override DataTemplate OnSelectTemplate (object item,
                                                       BindableObject container)
    {
        Person p = (Person)item; // Model
        return p.IsMainCharacter ? LeadingCharacter : SupportingCharacter;
    }
}
```

Applying a template selector

- To associate a template selector, set the **`ListView.ItemTemplate`** property to an instance of your **`DataTemplateSelector`**

```
<ContentPage.Resources>
   <ResourceDictionary>
      <local:CharacterSelector x:Key="MyTemplateSelector" />
   </ResourceDictionary>
</ContentPage.Resources>

<ListView x:Name="MessagesListView"
        ItemTemplate="{StaticResource MyTemplateSelector}"
        ItemsSource="{Binding People}"
        HasUnevenRows="True"
        ... />
```

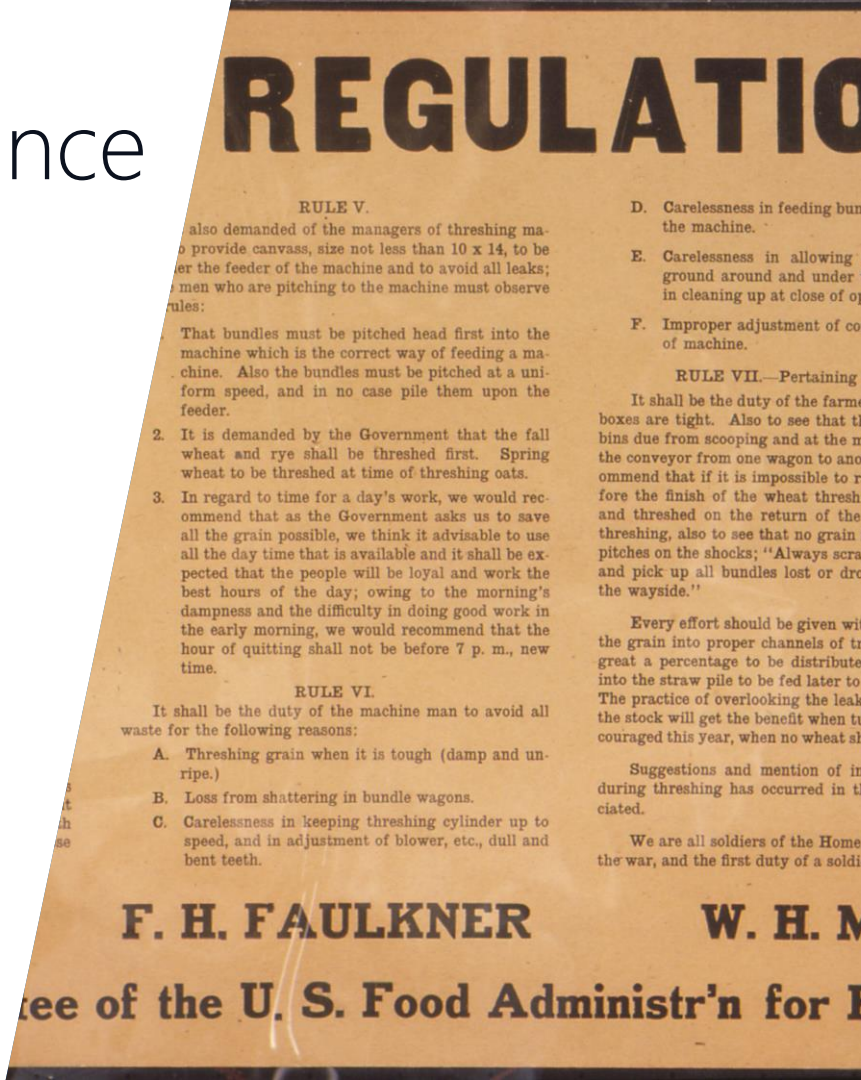
Template selector guidance

1. Minimize the number of returned data templates, particularly on Android

2. Must return the same template for a given model instance

3. Must return a `DataTemplate`, not another selector

4. Always reuse templates – do not allocate new instances each time



Flash Quiz

Xamarin University

Flash Quiz

① When **`HasUnevenRows`** is turned on, the **`ListView`** will use its **`RowHeight`** property if the **`ViewCell.Height`** property is zero

a) True

b) False

Flash Quiz

① When **`HasUnevenRows`** is turned on, the **`ListView`** will use its **`RowHeight`** property if the **`ViewCell.Height`** property is zero

a) True

b) False

Flash Quiz

② `DataTemplateSelector`s allow you to _____

a) build `DataTemplate`s in code

b) return a different `DataTemplate` based on the row

c) provide UI selection in your `DataTemplate`

Flash Quiz

② `DataTemplateSelector`s allow you to _____

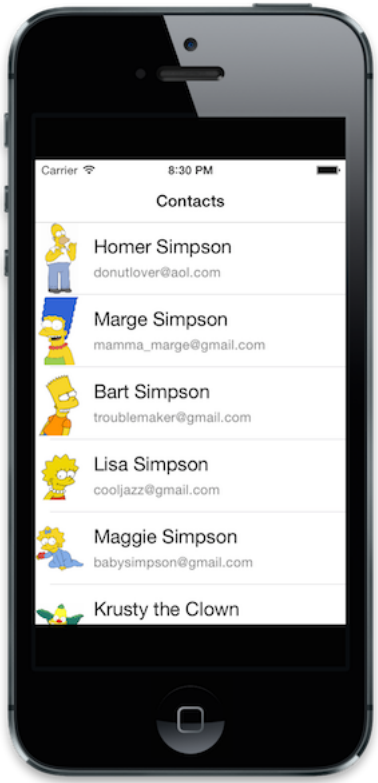
a) build `DataTemplate`s in code

b) return a different **`DataTemplate`** based on the row

c) provide UI selection in your `DataTemplate`

Summary

1. Using ViewCells
2. Creating Data Templates in code
3. Creating Unique Row Visuals

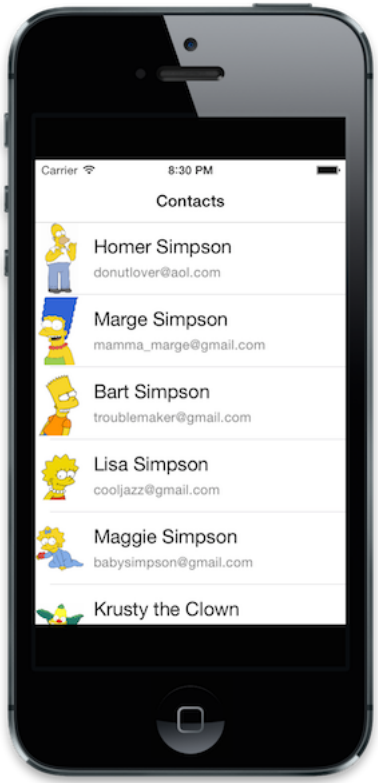


Adding Headers and Footers

Xamarin University

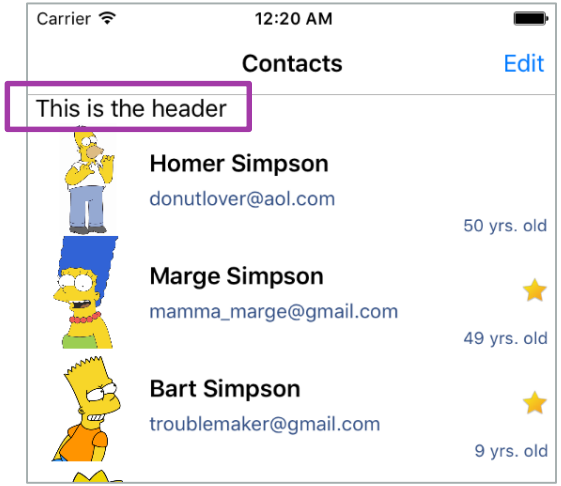
Tasks

1. Defining a header or footer
2. Creating a dynamic header or footer
3. Setting the binding context for a header or footer

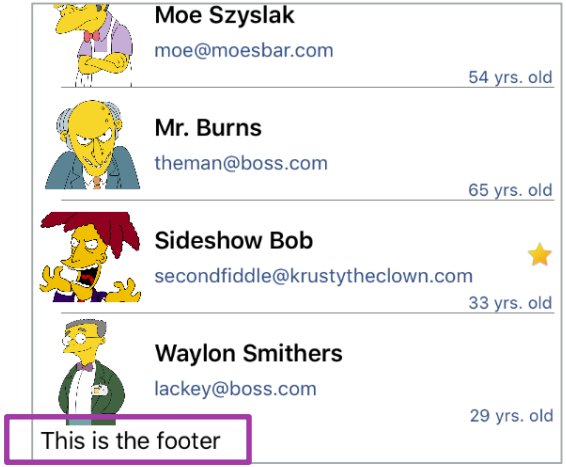


ListView header and footer

- **`ListView`** supports header and footer – which are rendered at the top and bottom of the **`ListView`** control

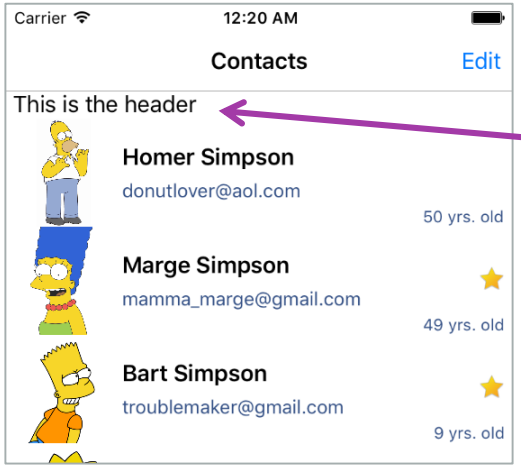


Headers and footers can be simple text or custom views



Setting the header and footer

- **`Header`** and **`Footer`** property define an object which is rendered directly into the **`ListView`** structure



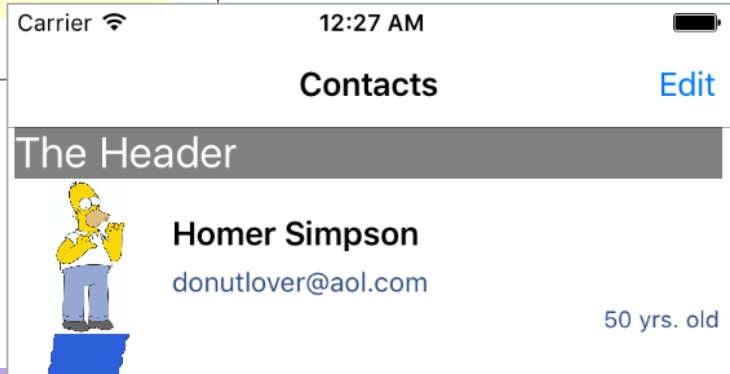

```
<ListView ...
    Header="This is the header"
    Footer="This is the footer">
```

Setting or data binding it to a string value causes a plain **`Label`** to be rendered

Setting the header and footer

- Can set the header or footer to a visual type to display custom visualizations

```
<ListView.Header>
   <ContentView BackgroundColor="Gray">
      <Label FontSize="Large" TextColor="White"
             Text="The Header" />
   </ContentView>
</ListView.Header>
```



Headers and Footers with MVVM

- Can define the header and footer as a **DataTemplate**; in this case, the **Header** and **Footer** properties are used as the **BindingContext**

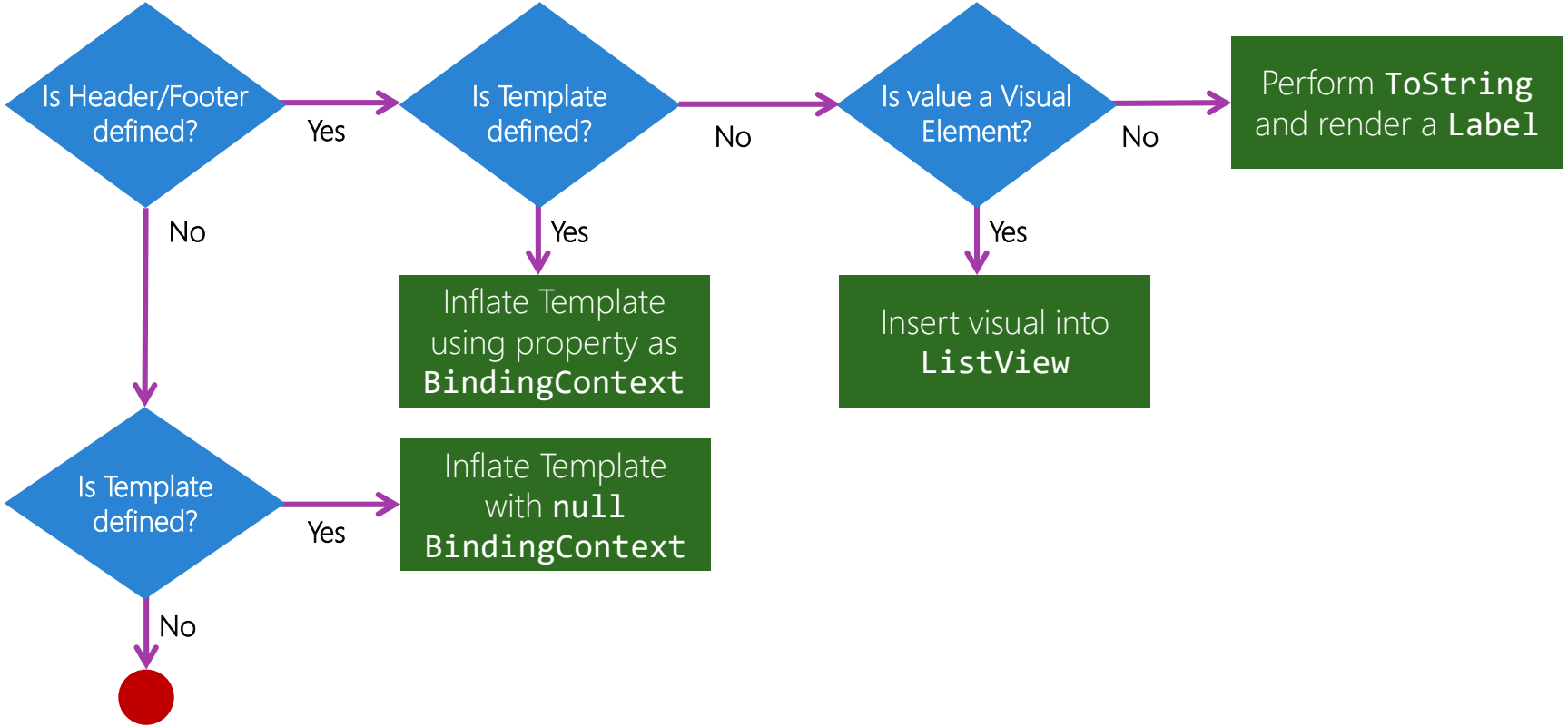
```
<ListView Header="{Binding HeaderText}">
   ...
   <ListView.HeaderTemplate>
      <DataTemplate>
         <Label FontSize="Large" TextColor="Blue"
             Text="{Binding .}" />
      </DataTemplate>
   </ListView.HeaderTemplate >
</ListView>
```

Headers and Footers with MVVM

- **Header** and **Footer** properties are then data bound to VM properties; which then populates the **HeaderTemplate** and **FooterTemplate**

```
public class MyViewModel : INotifyPropertyChanged
{
    string headerText;
    public string HeaderText {
        get { return headerText; }
        set { SetProperty(ref headerText, value); }
    ...
    public MyViewModel() {
        HeaderText = "The Header";
    }
}
```

Populating the header/footer data



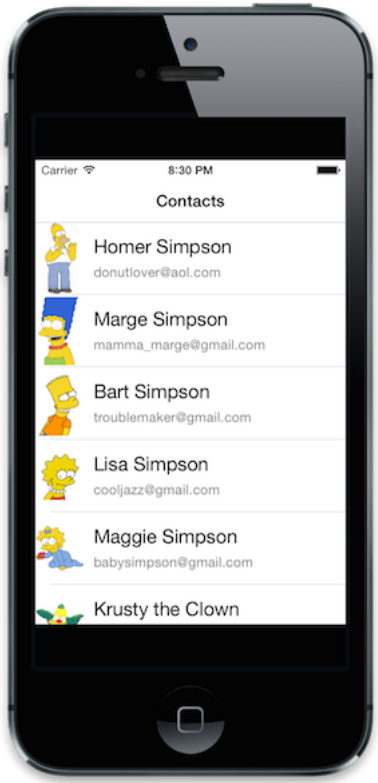
Individual Exercise

Add a header and footer to the ListView

Xamarin University

Summary

1. Defining a header or footer
2. Creating a dynamic header or footer
3. Setting the binding context for a header or footer

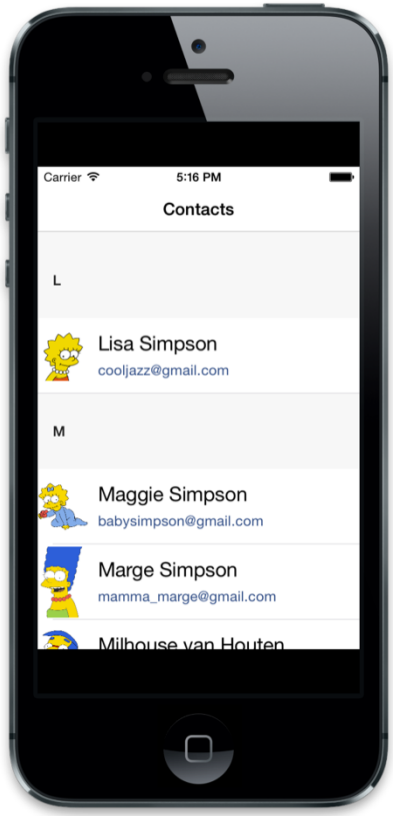


Separating your data into Groups

Xamarin University

Tasks

1. Sorting
2. Filtering
3. Grouping
4. Group Headers
5. Group Templates



Sorting

- Sorting can be done by modifying the underlying collection, or by replacing the **`ItemsSource`** property value

```
void OnSortAscending(object sender, EventArgs e)
{
    var data = Contacts.All;
    var sortedData = data.OrderBy(p => p.Name).ToList();
    contactList.ItemsSource = sortedData;
}
```

Faster to replace entire collection value than to remove/re-add all items

Filtering

- Filtering can be performed by double-buffering the collection; keeping a "raw" view with all items and a "UI" view with the specific filtered items

```
void OnFilter(object sender, EventArgs e)
{
    var data = Contacts.All;
    var filteredData = data.Where(p => p.Name.StartsWith("A"));
    contactList.ItemsSource = filteredData;
}
```

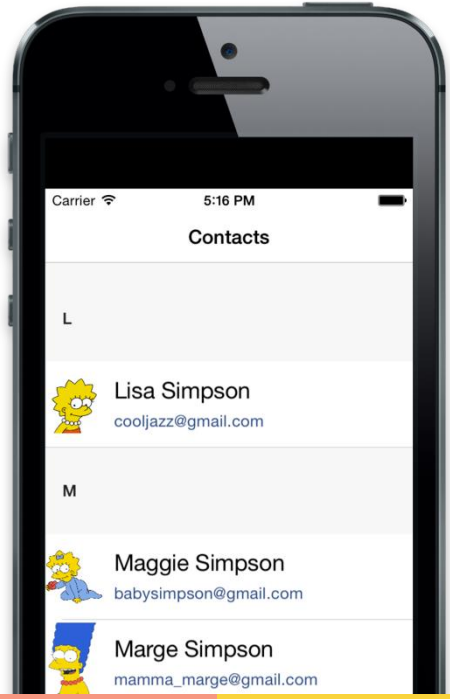
Notice here we are just using the LINQ query directly – e.g. assigning an **`IEnumerable`** to the **`ListView`**

Grouping

- **`ListView`** has built-in support to provide visual grouping of data

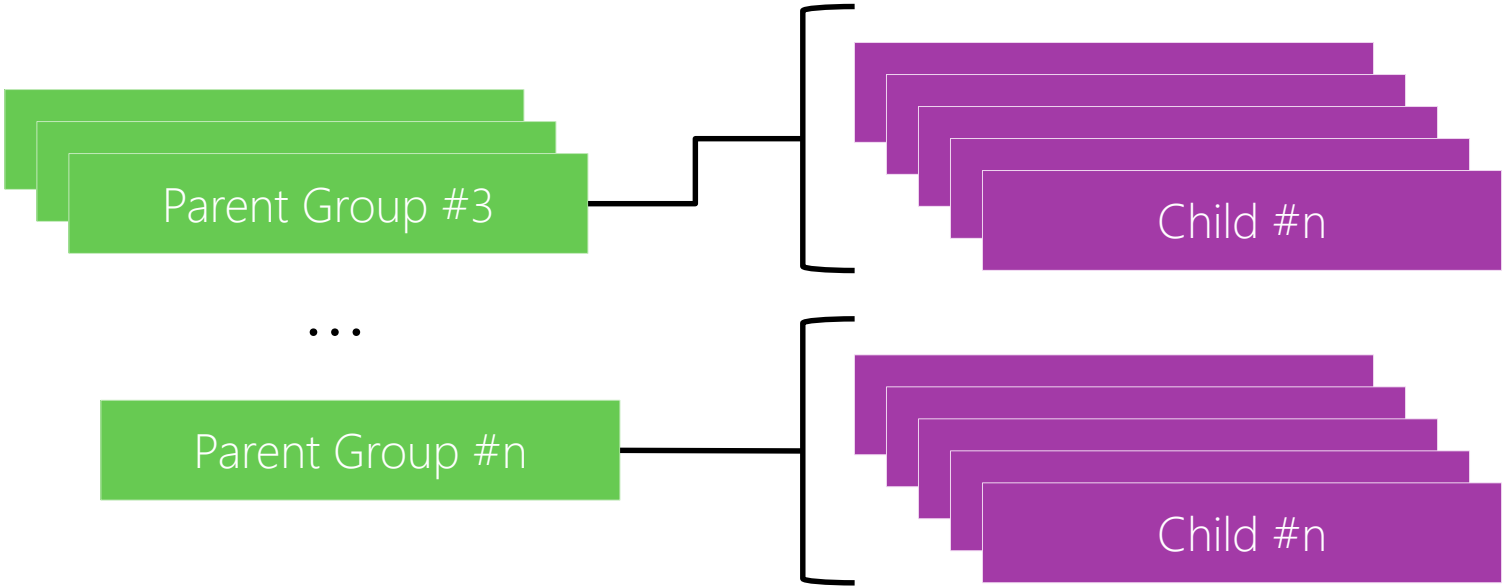
```
<ListView
    IsGroupingEnabled="True" ...>
```

feature is activated by setting the **`IsGroupingEnabled`** property



Supplying Grouped Data

- When grouping is activated, **`ListView`** expects data to be grouped in a parent-child fashion



Supplying Grouped Data

- Parent object must provide grouping property and implement **`IEnumerable`** for the children it owns

```
public class PersonGroup : ObservableCollection<Person>
{
    public string FirstLetter { get; set; }
    public string GroupName { get; set; }
    ...
}
```

Derive from an existing collection to expose the required **`IEnumerable`**

A more generic approach

- Can use a generic class to provide the data directly from LINQ's **GroupBy** expression

```
public class Grouping<K, T> : ObservableCollection<T>
{
    public K Key { get; private set; }
    public Grouping(K key, IEnumerable<T> items)
    {
        Key = key;
        foreach (var item in items)
            this.Items.Add(item);
    }
}
```

Populating with grouped data

- Can then use LINQ to group the data

```
var items = Contacts.All
  .OrderBy(c => c.Name)
  .GroupBy(c => c.Name[0].ToString(), c => c)
  .Select(g => new Grouping<string,Person>(g.Key, g))
  .ToList();

contactList.ItemsSource = items;
contactList.IsGroupingEnabled = true;
```

```
class Grouping<K, T> : ObservableCollection<T>
```

Adding a group header

- It is possible to add a header above each group in a **ListView**; can select a single property used to display a textual **Label**

```
<ListView
   GroupDisplayBinding="{Binding Key}" ...>
```

Value from binding is displayed at the top of the group

```
public class Grouping<K, T> : ObservableCollection<T>
{
   public K Key { get; private set; }
   ...
}
```

Adding a group header

- Can also supply the header as a full **`DataTemplate`** + **`Cell`** to allow for complete visual customization

```
<ListView.GroupHeaderTemplate>
   <DataTemplate>
     <TextCell Text="{Binding Key}"
          Detail="{Binding Count, StringFormat='{0} items'}" />
   </DataTemplate>
</ListView.GroupHeaderTemplate>
```

Group Exercise

Adding Grouping support to our Character List

Xamarin University

Adding a Quick Index

- iOS and WP support a "quick index" feature by setting the **GroupShortNameBinding** property; must supply a binding to a property that returns the string to use as the index

```
<ListView ItemsSource="{Binding .}"
        IsGroupingEnabled="true"
        GroupShortNameBinding="{Binding Key}">
```

String returned from **Binding** will be placed to the right of the **ListView**, tapping on the string will "jump" to that group

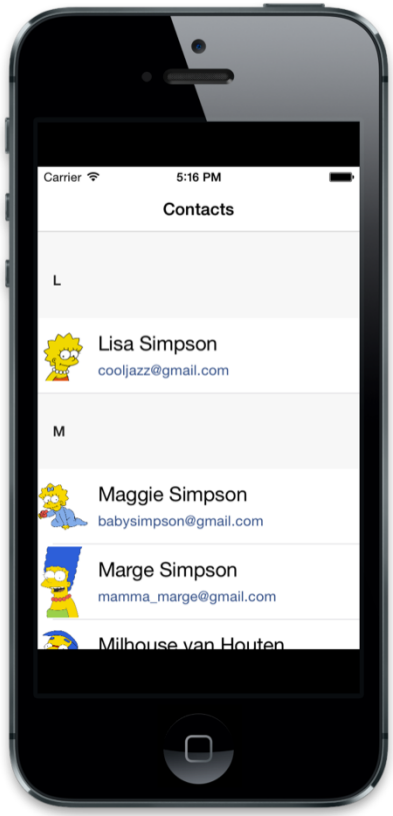
Individual Exercise

Add a Quick Index

Xamarin University

Summary

1. Sorting
2. Filtering
3. Grouping
4. Group Headers
5. Group Templates



Performance Tuning the ListView

Xamarin University

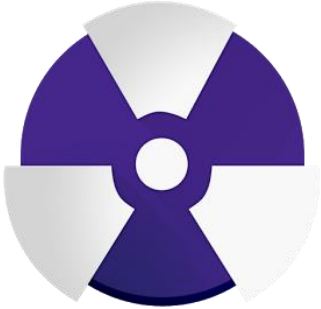
Performance Tips

- Several tweaks you can do to ListView to optimize it's performance based on the amount of data and the visualization being used

- Note that none of these are silver bullets – try each one as needed, but be prepared to profile and benchmark your application

Structure of a ListView row

- Each row is composed of two pieces that work together to display the information



Logical (Model) Information

- Data item from **`ItemsSource`** used as **`BindingContext`**
- **`Cell`**-derived object describes the desired visual layout

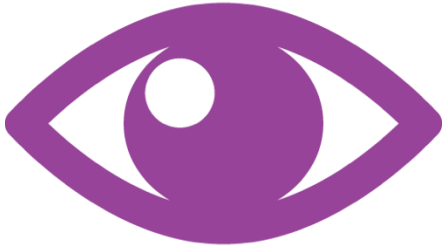
Structure of a ListView row

- Each row is composed of two pieces that work together to display the information

Cell Renderer which creates:

- iOS: `UITableViewCell`
- Android: `View`
- Windows: `ListViewItem`

Content is either native element, or generated from `ViewCell.View`



Visual Information

High performance lists

- Secret to high performance scrolling and visual rendering is *caching* and *reuse*
- Xamarin.Forms always uses native platform visual recycling/reuse
- However the *logical* element caching strategy can be decided by the developer

Configure caching strategy

- Xamarin.Forms 2.0 supports a new performance optimization related to how it generates Cells, called the *caching strategy*

RetainElement

`ListView` generates a **`Cell`** for *every item* in the list and keeps them around ("retains" them)

This is the *default* (old) behavior

Configure caching strategy

- Xamarin.Forms supports a performance optimization related to how it generates **`Cell`**s, called the *caching strategy*

RetainElement

RecycleElement

`ListView` minimizes the memory footprint by recycling cells as you scroll through the list

This is the preferred behavior

Turn on ListView recycling

- Most apps will benefit from recycling cells – new behavior must be set when **`ListView`** is created and cannot be changed at runtime

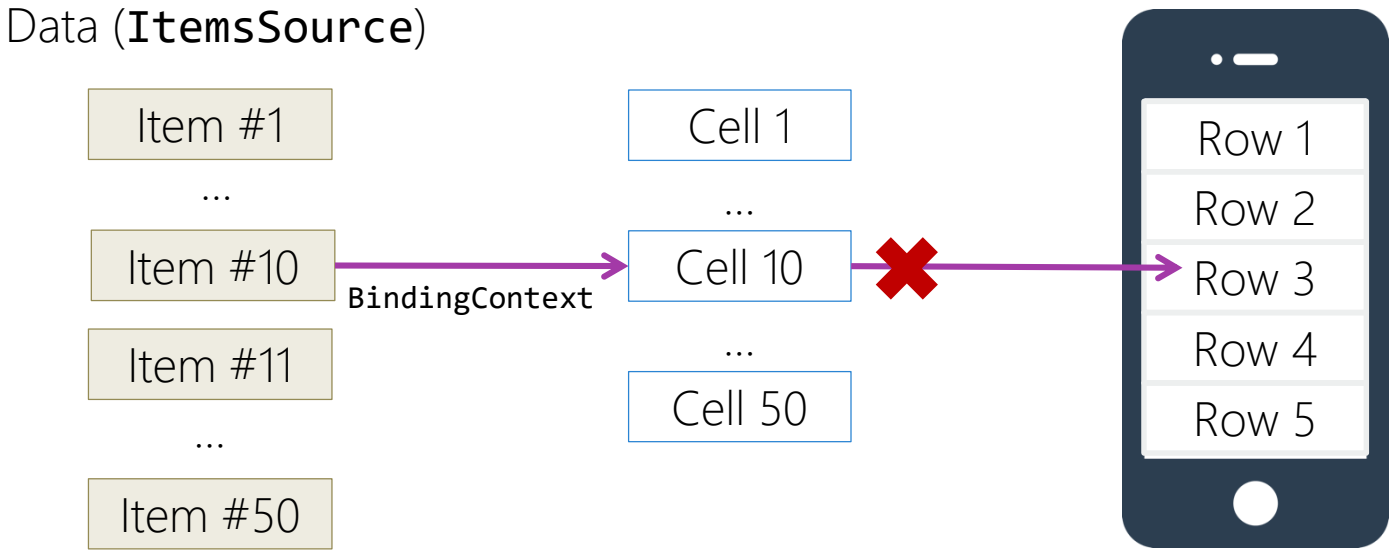
```
<ListView CachingStrategy="RecycleElement" ... />
```

or

```
var lv = new ListView(ListViewCachingStrategy.RecycleElement);
```

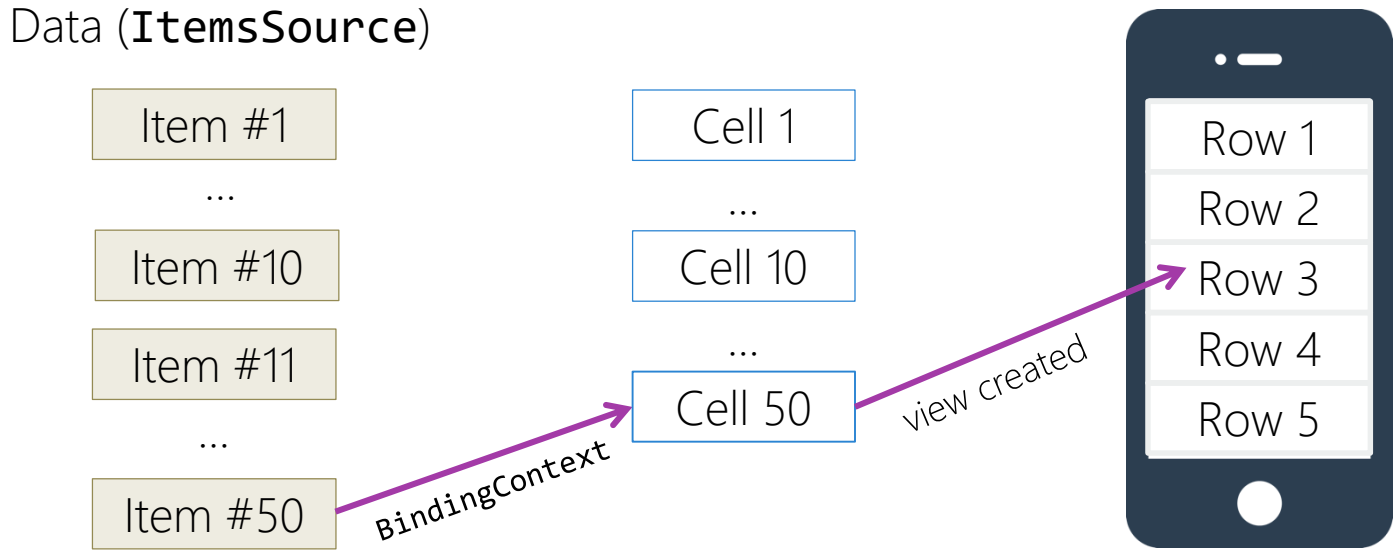
When cell recycling is OFF

- When cell recycling is off (default), a unique cell is created for each data item and used to populate the information in a visible row



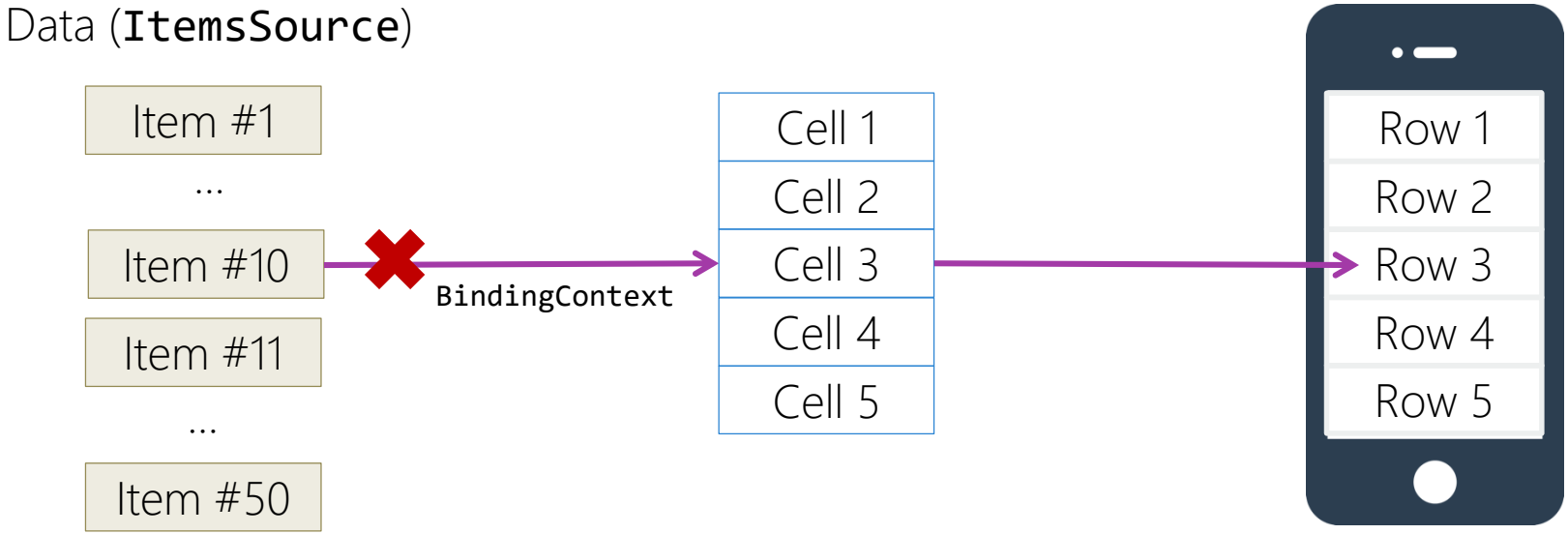
When cell recycling is OFF

- When cell recycling is off (default), a unique cell is created for each data item and used to populate the information in a visible row



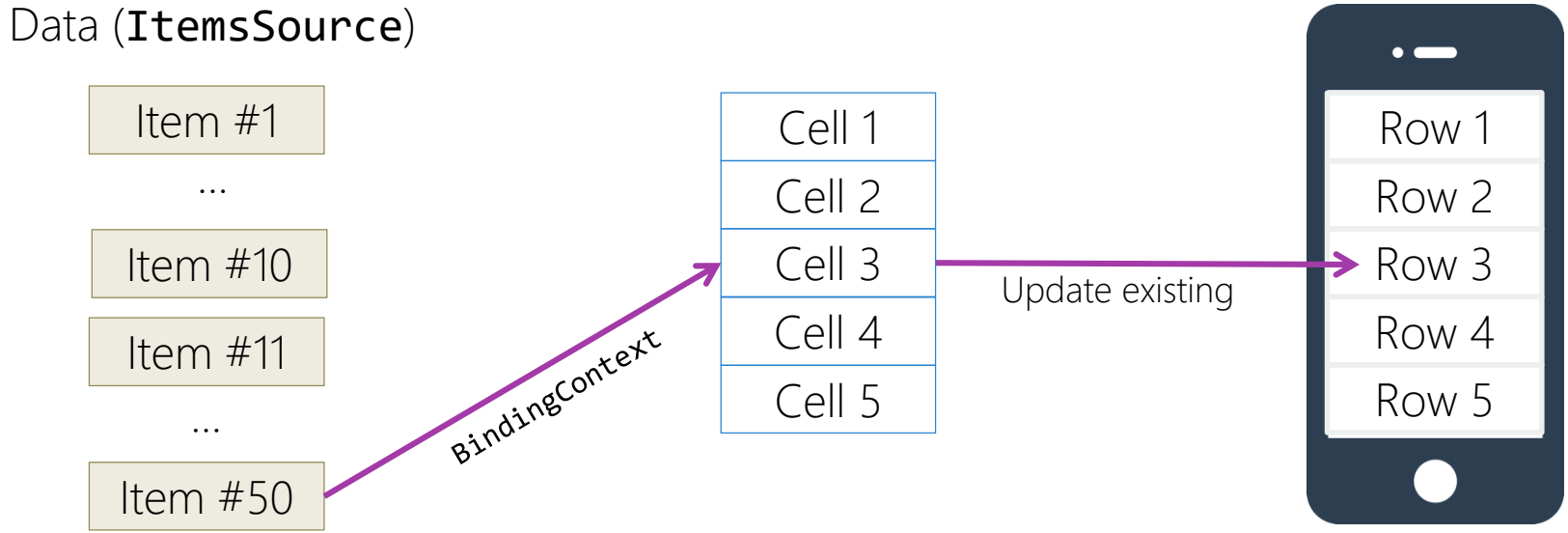
When cell recycling is ON

- When cell recycling is turned on, the cell is associated to a specific visual row and the **`BindingContext`** is changed to supply the data



When cell recycling is ON

- When cell recycling is turned on, the visual cell is kept and *reused* and the **`BindingContext`** is changed to point to the new data to visualize



When cell recycling is ON

- When cell re ... *used* and the **Bindin** ... visualize

Data (**ItemsSo**

Item #1

...

Item #10

Item #11

...

Item #50

BindingConte

Cell 5

Row 1

Row 2

Row 3

Row 4

Row 5

That means:

- All data about the cell must come from the binding context!
- Custom renderers must correctly update visuals from property change notifications

When should I retain vs. recycle

- If you have a large number of bindings on the cell (e.g. > 20)
- .. or if the cell visuals change a lot based on the binding context
- .. or if testing shows that RecycleElement is slower for you

Then prefer RetainElement

Individual Exercise

Turn on recycle caching

Xamarin University

Built-in cells vs. ViewCell

- Built-in cells (**`TextCell`**, **`SwitchCell`**, etc.) are mapped to native styles in each framework and are faster and lighter than using **`ViewCell`**

```
<ViewCell>
  <Label Text="{Binding Name}" ... />
</ViewCell>
```

If you can make it work, always use the built-in cell styles

```
<TextCell Text="{Binding Name}" ...>
```

Think about your data source

- Always prefer **IList<T>** over **IEnumerable<T>**

```
var items = Contacts.All
  .OrderBy(c => c.Name)
  .GroupBy(c => c.Name[0].ToString(), c => c)
  .Select(g => new Grouping<string,Person>(g.Key, g))
  .ToList();
```

LINQ always produces **IEnumerable** expressions – should always take them and use **ToList()** to turn them into a concrete list, or pass the result into as new **ObservableCollection<T>**

Simplify your visual design

- Work on minimizing your visual construction – try to display your UI with as few elements and as few property setters as possible

```
<StackLayout Orientation="Horizontal">
    <Label Margin="10,10,5,10" Text="Hello"/>
    <Label Margin="0,10,5,10" Text="{Binding FirstName}"/>
    <Label Margin="0,10,10,10" Text="{Binding LastName}"/>
</StackLayout>
```

Specifies **Margin** on each **Label** to provide uniform spacing around each of them

Simplify your visual design

- Work on minimizing your visual construction – try to display your UI with as few elements as possible

```
<StackLayout Orientation="Horizontal" Padding="10" Spacing="5">
    <Label Text="Hello"/>
    <Label Text="{Binding FirstName}"/>
    <Label Text="{Binding LastName}"/>
</StackLayout>
```

Know your layout properties! **Spacing** on the **StackLayout** gives us exactly the same result but minimizes the layout pass complexity

Simplify your visual design

- Avoid using expensive layout panels for a single element

```
<ViewCell>
   <Grid Padding="10" BackgroundColor="Gray">
       <Label Text="This is bad"/>
   </Grid>
</ViewCell>
```

Simplify your visual design

- Avoid using expensive layout panels for a single element

```
<ViewCell>
   <ContentView Padding="10" BackgroundColor="Gray">
       <Label Text="This is bad"/>
   </ContentView>
</ViewCell>
```

ContentView is a lighter weight container which can often host your content in exactly the same way as **Grid** or **StackLayout**

Avoid ScrollView

- Do not place **ListViews** into a scrollable control (e.g. **ScrollView**), instead use the **Header** and **HeaderTemplate** property to place scrollable fixed content at the top of the list

```
<ScrollView>
    <Label ... />
    <ListView />
</ScrollView>
```

```
<ListView.Header>
    <Label ... />
</ListView.Header>
```

Cut out unnecessary property setters

- Don't bother to set property values to the "defaults"

```
<ViewCell>
   <ContentView Padding="10">
       <Label Text="..."
          HorizontalTextAlignment="Start"
          VerticalTextAlignment="Start" />
   </ContentView>
</ViewCell>
```

This requires us to set the property (and store the value) to exactly what it was when the Label was created.. *every time we create a ViewCell!*

Optimizing your labels

- Labels are the most common visual element, and can be the most expensive because measuring text is expensive
  - Prefer `LineBreakMode.NoWrap`
  - Don't set `VerticalTextAlignment` unless needed
  - Don't update labels more often than necessary (avoid layout pass)
- Consider using a single `FormattedString` label instead of multiple labels for static text

Optimize your images

- Images are scaled / resized as they are drawn
  - Should use appropriately sized images to improve memory and render performance
  - Prefer .pngs for icons and "pixel-perfect" displays or transparent elements
  - Use .jpgs for larger photos – these are compressed and load faster
  - Use async task for background image downloads

Optimizing custom layouts (ViewCell)

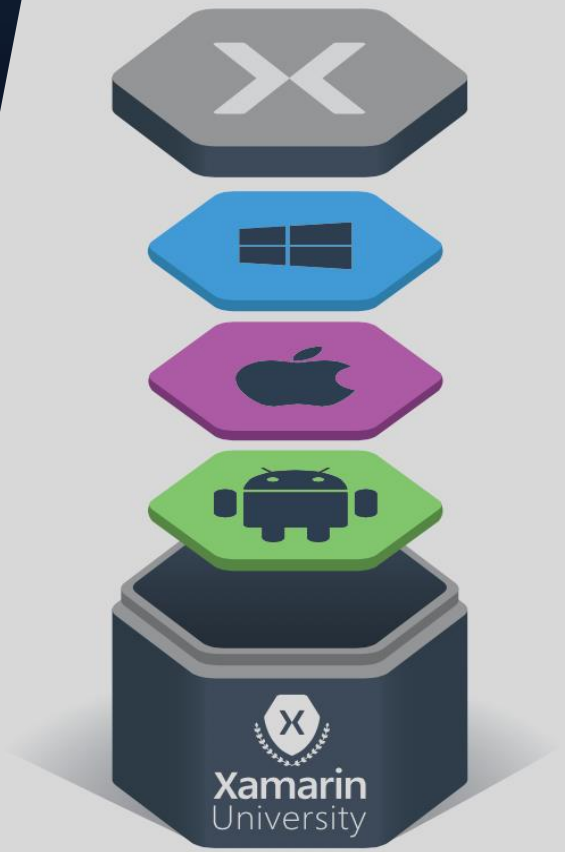
- ✓ **`Horizontal/VerticalOptions`** should be set to **`Fill`** or **`FillAndExpand`** (these are the defaults)
- ✓ Avoid nesting panels if possible
- ✓ When using **`StackLayout`**, one child ideally will be set to **`FillExpand`**
- ✓ Prefer **`AbsoluteLayout`** - can potentially do layouts in a single pass
- ✓ Avoid **`RelativeLayout`** for now if possible
- ✓ Avoid auto-sized columns/rows with **`Grid`**, fixed-sized are best
- ✓ Transparency is expensive, unless it's "0" or "1"
- ✓ XAMLC helps for XAML-based template when using **`Retain`**

Summary

- Performance is hard
- It's not always the frameworks fault
- Will likely require some testing, benchmarking and changes to optimize your app

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile



Microsoft