
XAM312

# Customizing the ListView in Xamarin Forms

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

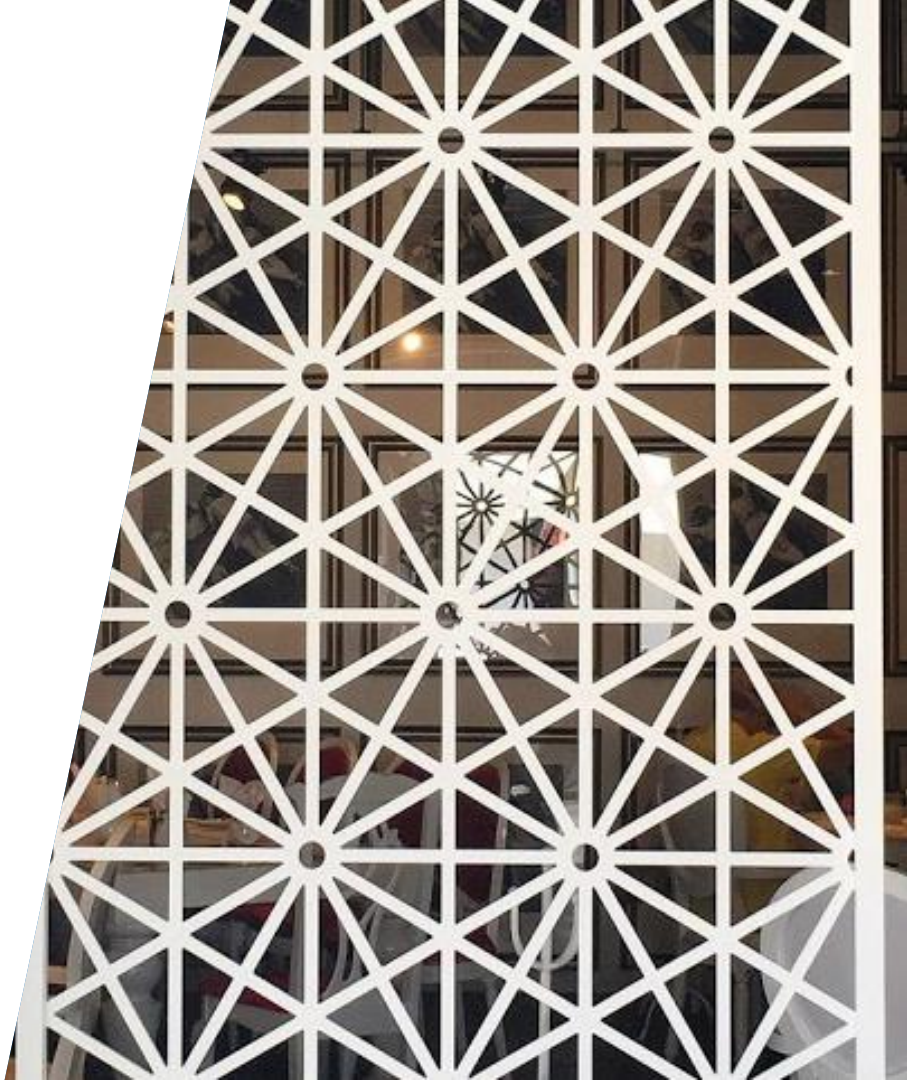
# Objectives

1. Create custom cell definitions
2. Add headers and footers to a ListView
3. Display grouped data in a ListView
4. Customize the cell view based on data
5. Optimize ListView Performance

# Create Custom Cell Definitions

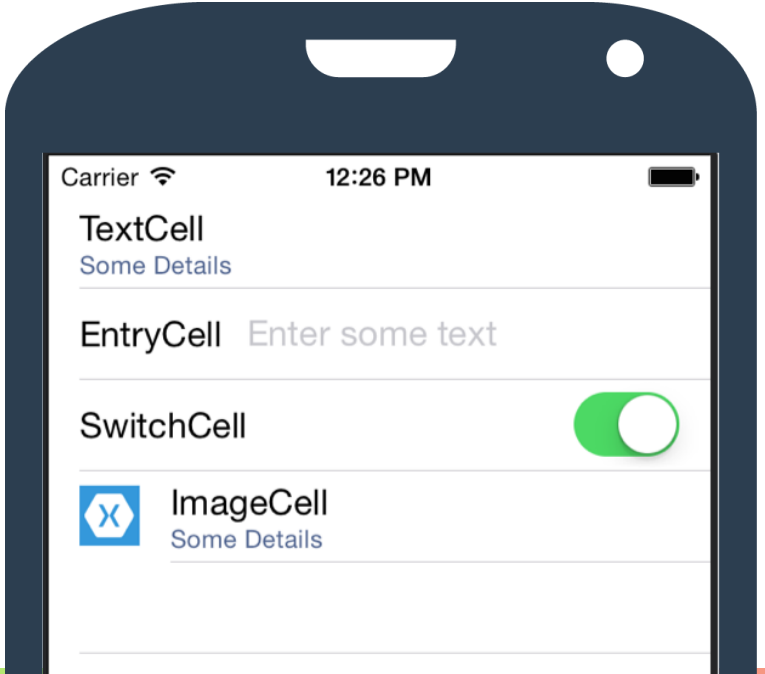
# Tasks

1. Define custom cells using ViewCell
2. Create Data Templates in code
3. Define unique row visuals



# Reminder: Cell Styles

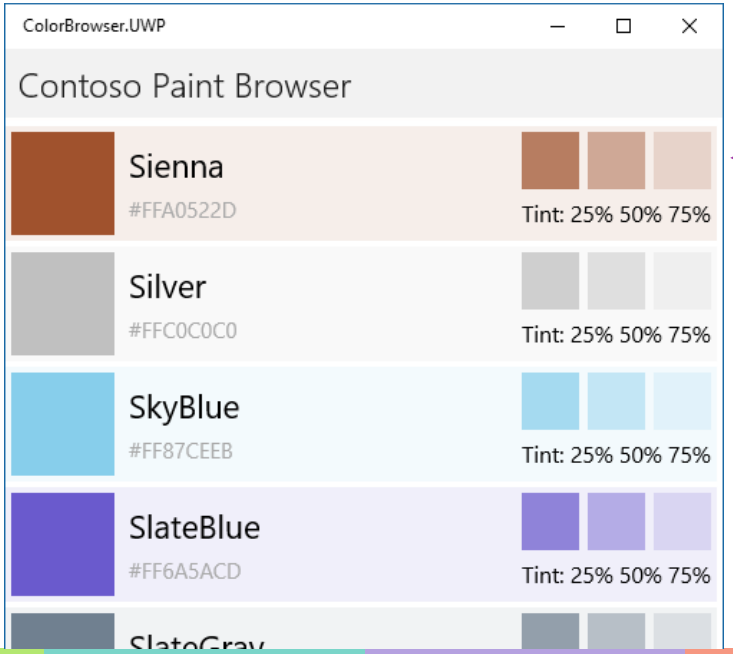
- `ListView` provides several built-in cell styles for the most common scenarios



# Custom cells

- You can create your own **`ListView`** cell to present a custom cell layout

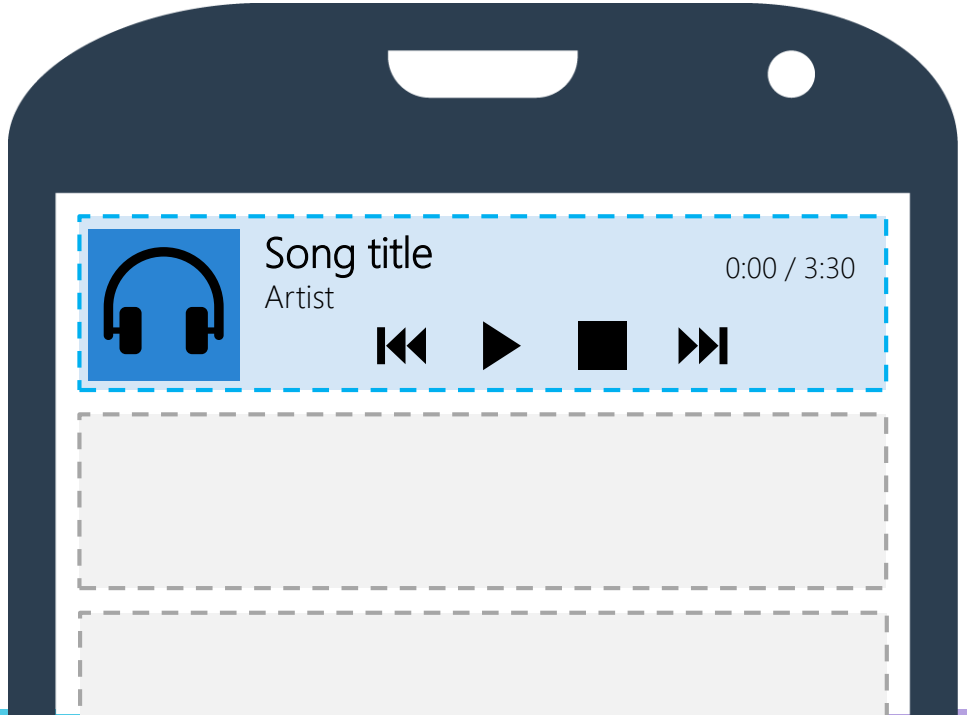
A custom cell lets you use elements not provided in the default cells




A custom cell can present multiple visual elements

# What is a ViewCell?

- ViewCell is a ListView cell that displays a developer defined view




ViewCell hosts your custom layout

# Define a ViewCell in XAML

- Can define a **`ViewCell`** in XAML and assign it to the **`ListView`**'s **`DataTemplate`**

```
<ListView>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>

            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
    ...
</ListView>
```

**`ViewCell`** hosts your custom layout

Your custom layout goes here

# Define ViewCell content in XAML

- A **`ViewCell`** holds a single child that defines your custom layout

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

The child is often a layout which allows multiple visual elements to be added

# Dynamic content in a ViewCell

- **ViewCell**s can utilize the **BindingContext** for the **ListView** to display dynamic content

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

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

# Define ViewCells in code

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

```
public class ContactViewCell : ViewCell
{
  public ContactViewCell()
  {
    var name = new Label();
    var toggle = new Switch();

    name.SetBinding(Label.TextProperty, new Binding("Name"));
    toggle.SetBinding(Switch.IsToggledProperty, new Binding("Favorite"));

    this.View = new StackLayout { Children = { name, switch } };
  }
}
```

# Applying custom view cells in code

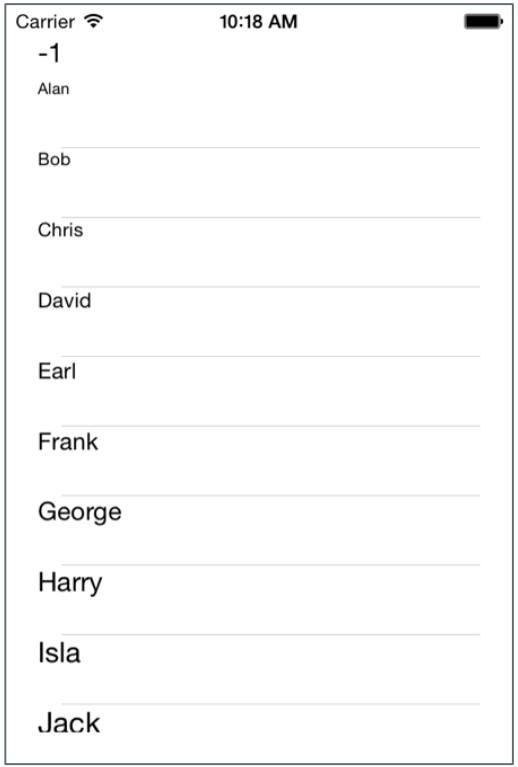
- **`ViewCell`**s must be assigned to a **`DataTemplate`** which are then added to the ListView's **`ItemTemplate`** property

```
var template = new DataTemplate (()=> return new ContactViewCell());

var list = new ListView ();

list.ItemTemplate = template;
```

Constructor takes a **`Func`** that creates the **`ViewCell`**

# ViewCell default row height

- When using a **`ViewCell`**, the **`ListView`** estimates the required height based on content and will set the height of every cell to fit the tallest cell



# Setting the ListView row height

- Specify the height to be used for all rows in a **`ListView`** by setting the **`RowHeight`** property

```
<ListView RowHeight="20">
```

# Individual Exercise

Provide a custom cell template for a ListView

Xamarin University

# Variable-sized rows

- **`ListView`** can display different heights for each cell based on the cell's content by setting **`HasUnevenRows`** to **`true`**

```
<ListView
    HasUnevenRows="true">
```

# Set cell height

- Set the cell's **Height** property to help the **ListView** determine the amount of vertical space the cell requires

```
public class MyContactViewCell : ViewCell
{
    ...
    protected override void OnBindingContextChanged()
    {
        base.OnBindingContextChanged();
        var contact = (MyContact)BindingContext;

        this.Height = contact.IsFavorite ? 20 : 15;
    }
}
```

# Runtime row resizing

- Individual **ListView** row height can be recalculated at runtime by calling **ForceUpdateSize** 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();
    }
}
```

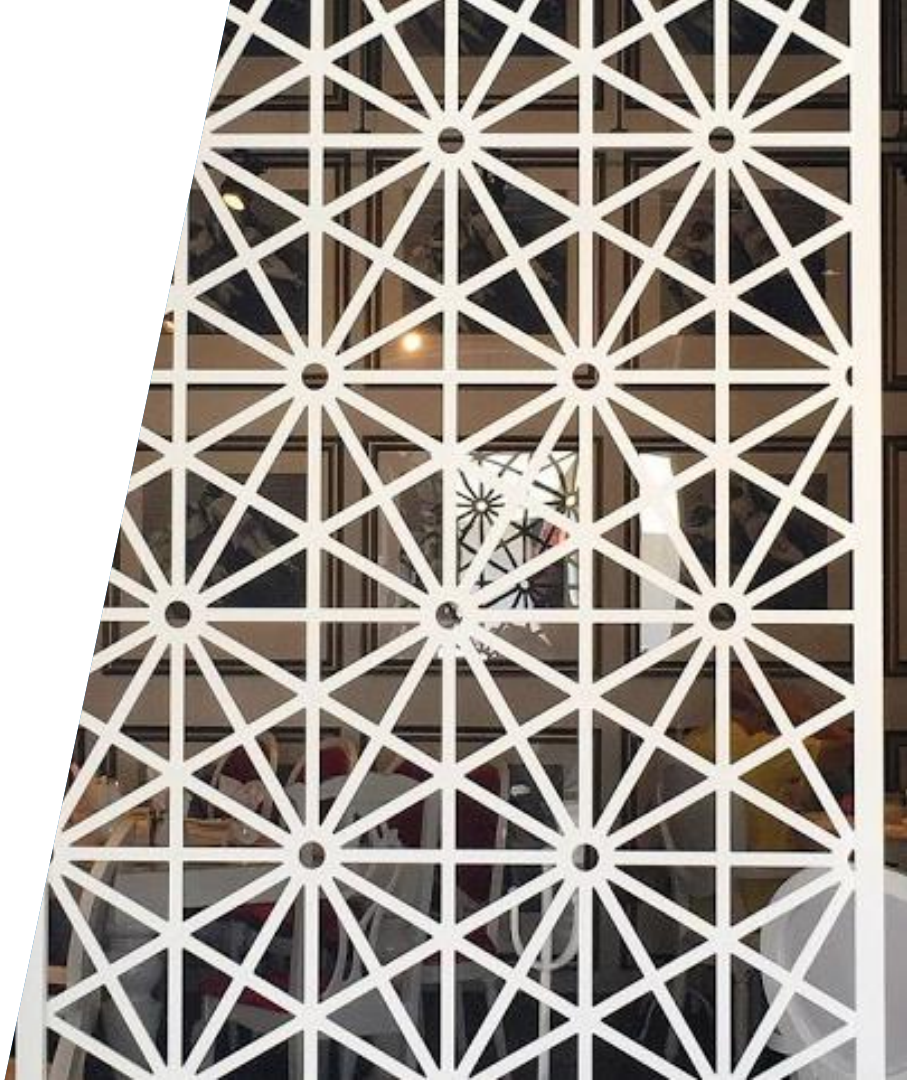
Update the height of the child elements

Call **ForceUpdateSize** to update the cell

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

# Summary

1. Define custom cells using ViewCell
2. Create Data Templates in code
3. Define unique row visuals

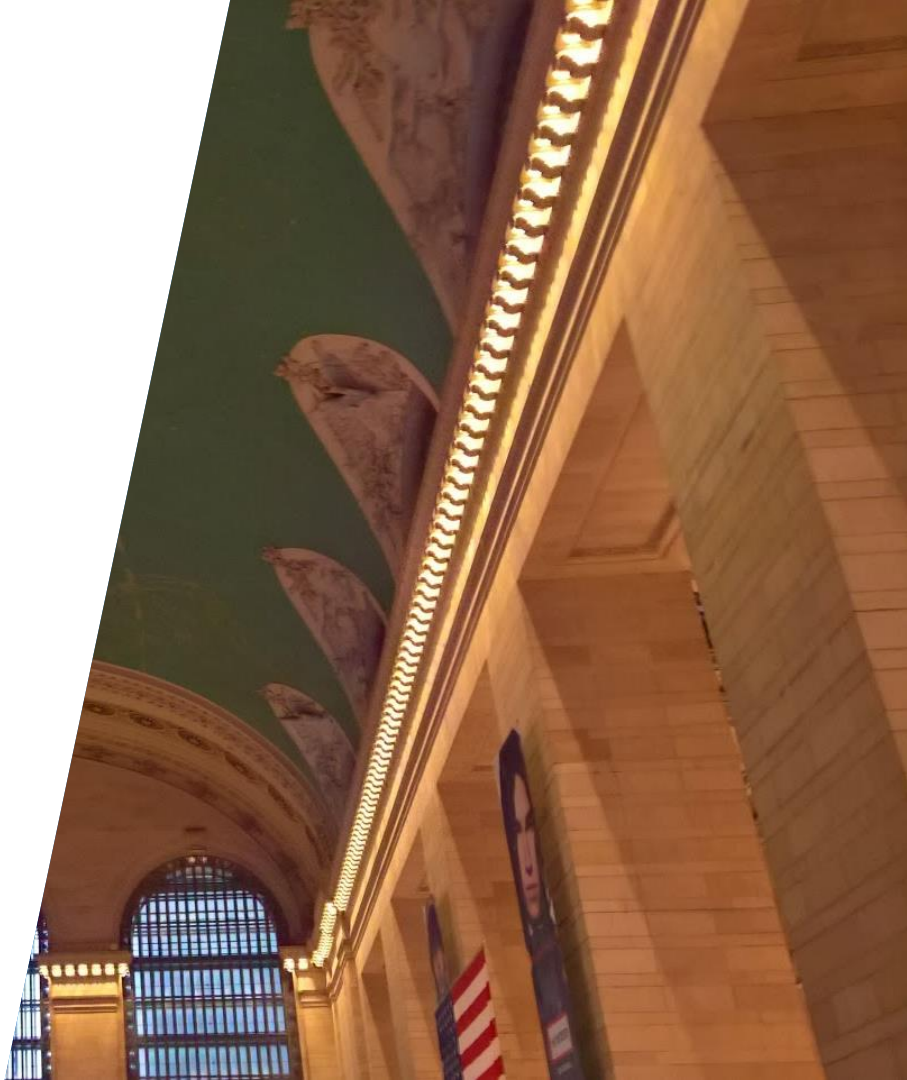


# Add Headers and Footers to a ListView

Xamarin University

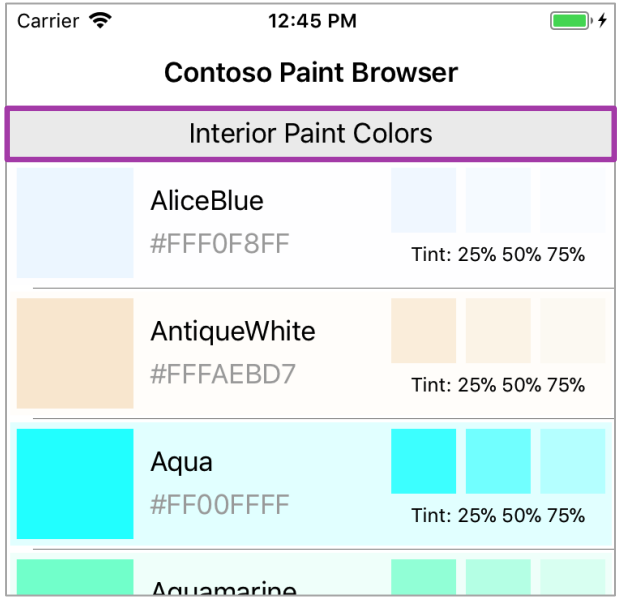
# Tasks

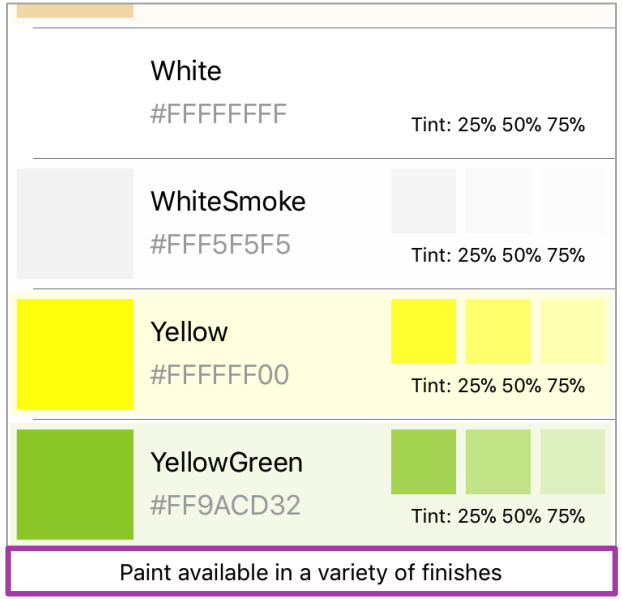
1. Define a header
2. Define a footer
3. Create a dynamic header or footer
4. Set the binding context for a header or footer



# ListView header and footer

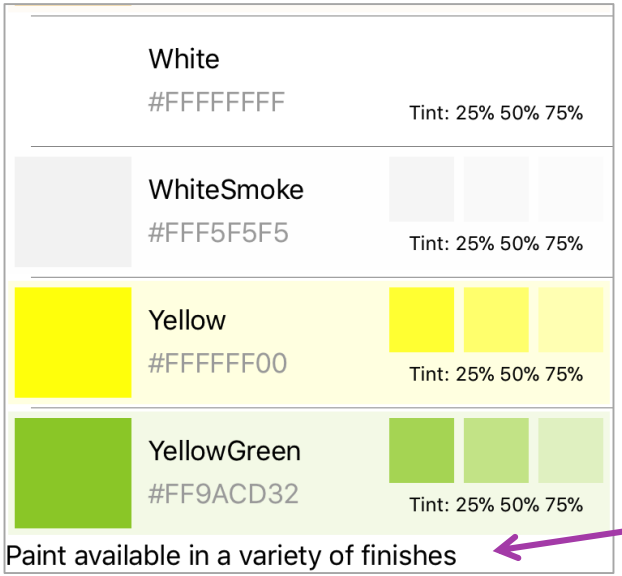
- **`ListView`** supports header and footer – which are rendered at the top and bottom of the **`ListView`** control



# Setting the header and footer

- **`Header`** and **`Footer`** property define objects which are rendered directly into the **`ListView`**




Assigning the Header or Footer properties causes a plain **`Label`** to be rendered

```
<ListView ...
    Footer="Paint available ...">
```

# 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="Interior Paint Colors" />
    </ContentView>
</ListView.Header>
```

# Binding Headers & Footers

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

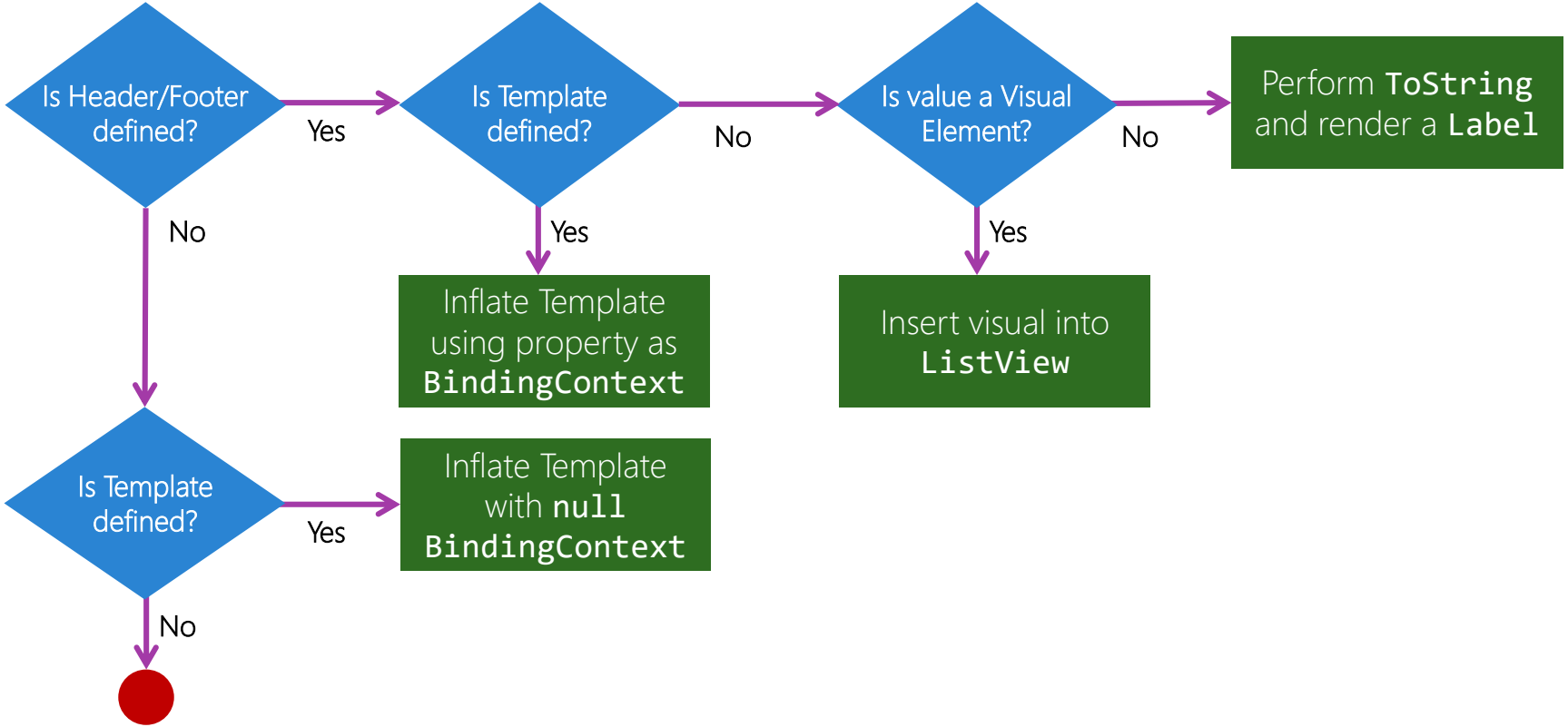
```
<ListView Header="{Binding HeaderText}">
   ...
   <ListView.HeaderTemplate>
      <DataTemplate>
         <Label FontSize="Large" TextColor="Blue"
             Text="{Binding .}" />
      </DataTemplate>
   </ListView.HeaderTemplate >
</ListView>
```

# Headers & Footers binding properties

- **Header** and **Footer** properties are then data bound to properties which populate the **HeaderTemplate** and **FooterTemplate**

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

# Populating the header/footer data



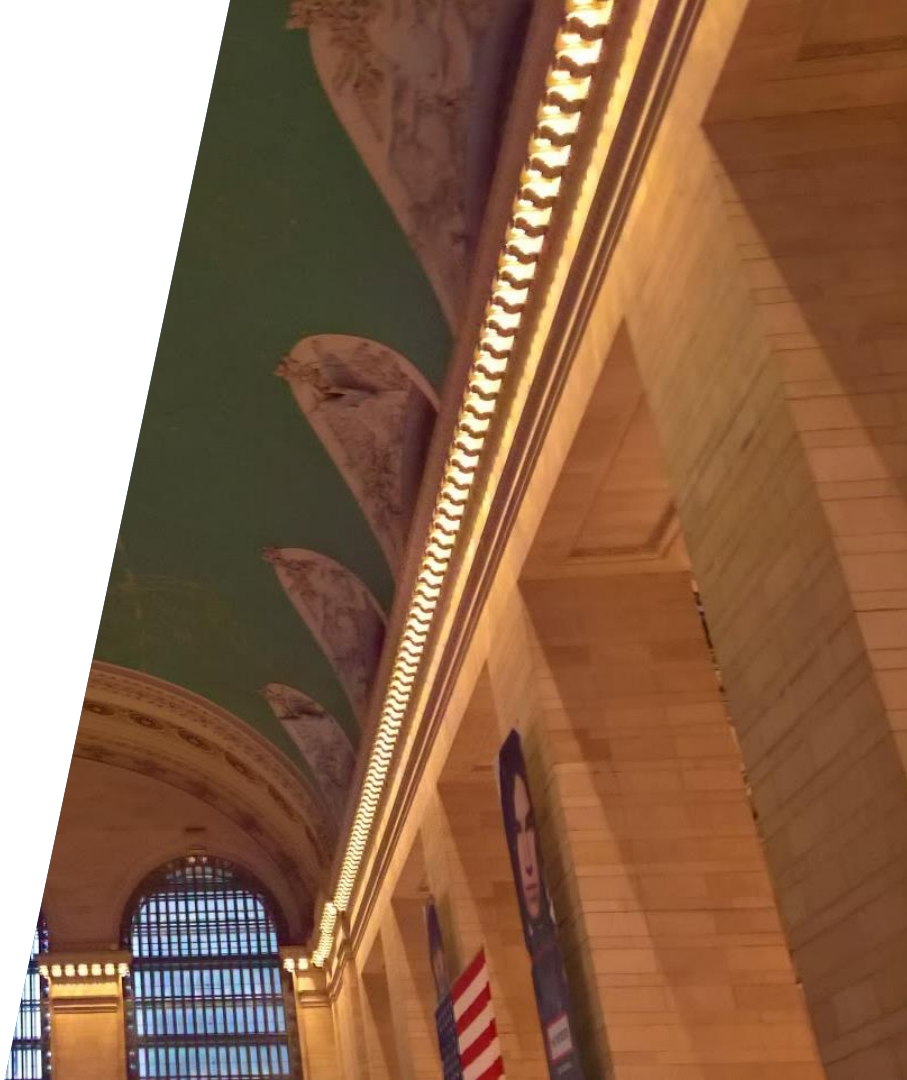
# Individual Exercise

Add a header and footer to the ListView

Xamarin University

# Summary

1. Define a header
2. Define a footer
3. Create a dynamic header or footer
4. Set the binding context for a header or footer



# Separating your data into Groups

Xamarin University

# Tasks

1. Sort data in a ListView
2. Filter data in a ListView
3. Group data in ListView

# Sorting ListView data

- Sort the data displayed in the list view by either modifying or replacing the underlying collection

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

# Filtering displayed data

- Filtering is performed on the data collection

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

# Grouping

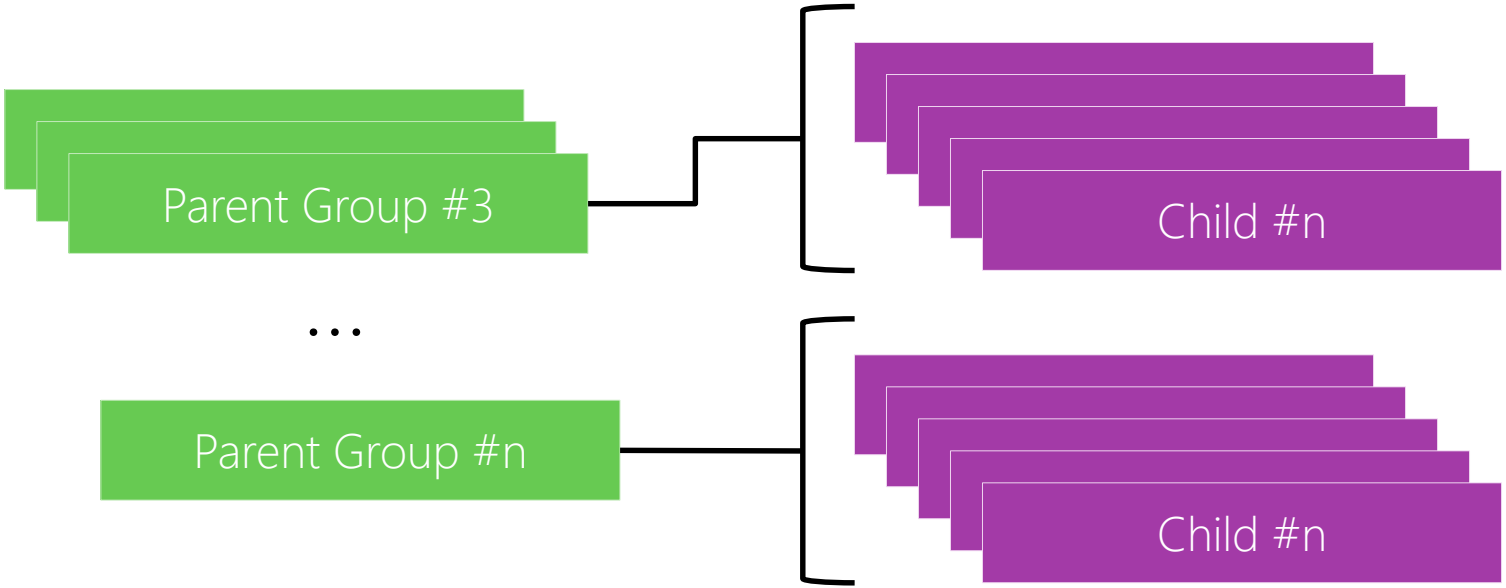
- **`ListView`** has built-in support to provide visual grouping of data

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

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

# Supplying Grouped Data

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



# Supplying Grouped Data

- The parent object must provide a 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`**

# Populating with grouped data

- Can use LINQ to group data

```
ILookup groupedContacts = Contacts.All
   .OrderBy(c => c.Name)
   .ToLookup(a => a.Name[0].ToString());

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

# Adding a group header

- Can 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

# Adding a group header

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

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

# Adding a Quick Index

- **ListView** supports 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 will "jump" to the group

# Individual Exercise

Add grouping and a Quick Jump Index

Xamarin University

# Summary

1. Sort data in a ListView
2. Filter data in a ListView
3. Group data in ListView

# Customize the cell view based on data

Xamarin University

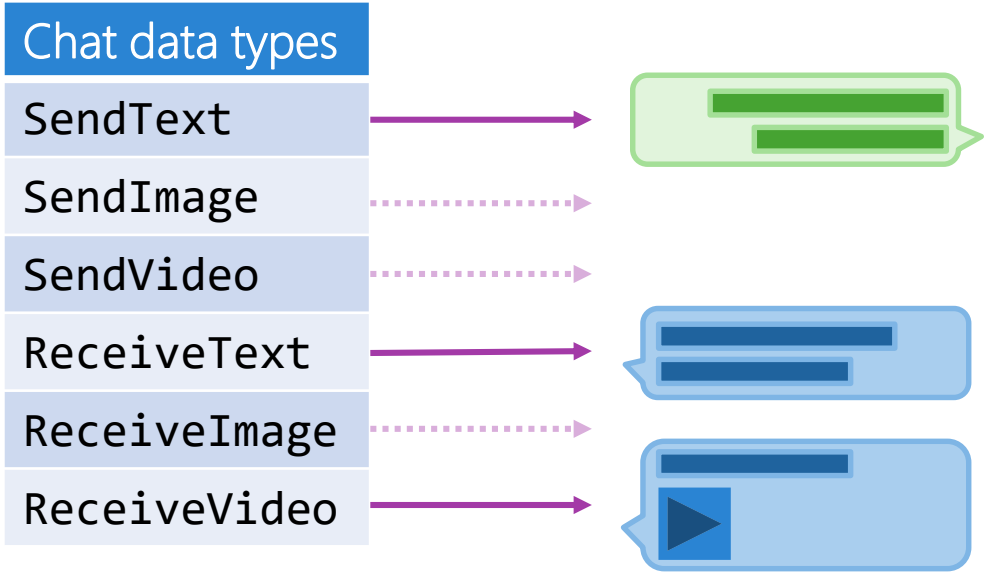
# Not all data is the same

- Different cell visualizations may be required when the type of data varies within the bound collection

E.g. a chat app may show Images, emojis, multimedia and text

# What is a data template selector

- A **`DataTemplateSelector`** is a **`DataTemplate`** that has the ability to provide different **`DataTemplate`**s based on the bound data element



# Create a DataTemplateSelector

- Create a new class that derives from **`DataTemplateSelector`** and override **`OnSelectTemplate`**

```
class ChatSelector : Xamarin.Forms.DataTemplateSelector
{
   protected override DataTemplate OnSelectTemplate (object item,
                                                      BindableObject container)
   {
      ...
   }
}
```

# Return different DataTemplates

- **`OnSelectTemplate`** receives the data-bound object for each cell – you can perform runtime checks on the **`item`** param to decide which template to use

```
class ChatSelector : Xamarin.Forms.DataTemplateSelector
{
    public DataTemplate sentTemplate { get; set; }
    public DataTemplate receivedTemplate { get; set; }

    protected override DataTemplate OnSelectTemplate (object item,
                                                       BindableObject container)
    {
        Message m = (Message)item; // Model
        return m.IsSentMessage ? sentTemplate : receivedTemplate;
    }
}
```

# Cache DataTemplates

- Only create one instance of a **`DataTemplate`** per cell within a **`DataTemplateSelector`**

```
class ChatSelector : Xamarin.Forms.DataTemplateSelector
{
   public DataTemplate sentTemplate { get; set; }
   public DataTemplate receivedTemplate { get; set; }

   public ChatSelector()
   {
      sentTemplate     = new DataTemplate(typeof(SentViewCell));
      receivedTemplate = new DataTemplate(typeof(ReceivedViewCell));
   }
   ...
}
```

# 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="ChatSelector" />
   </ResourceDictionary>
</ContentPage.Resources>

<ListView x:Name="listMessages"
        ItemTemplate="{StaticResource ChatSelector}"
        ItemsSource="{Binding ChatHistory}"
        HasUnevenRows="True"
        ... />
```

# Individual Exercise

Display different cells based on the bound-data

Xamarin University

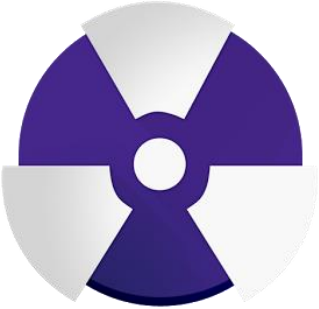
# Optimize ListView Performance

# Tasks

- Configure caching strategy
- Optimize layout
- Optimize properties and setters

# 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

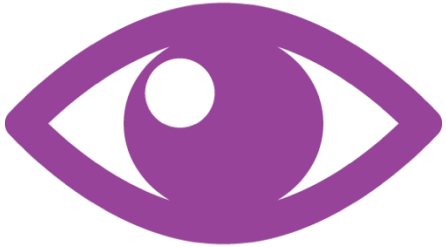
# Structure of a ListView row

- 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

# Caching strategies

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

**Retain Element (default)**

**`ListView`** generates a **`Cell`** for *every item* in the list and keeps them in memory

**Recycle Element**

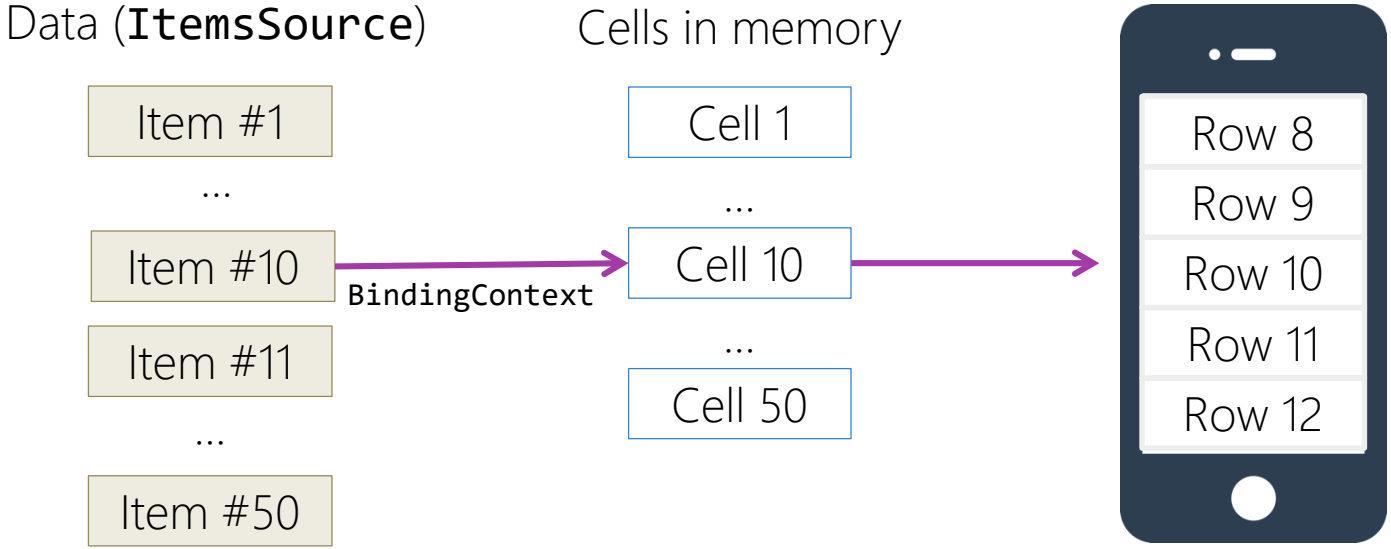
**`ListView`** reuses allocated cells as it's scrolled

**Recycle Element and Data Template**

**`ListView`** reuses allocated cells and keeps reference to its data template

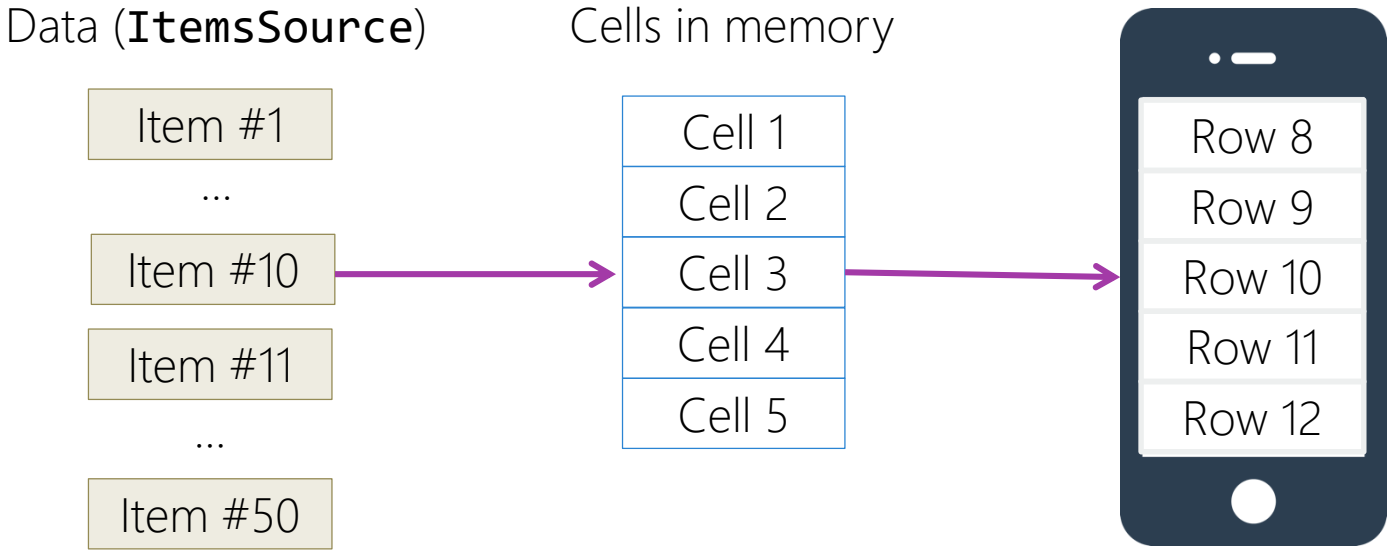
# Retain Element

- The **`RetainElement`** caching strategy tells the **`ListView`** to generate a cell for each item in the bound data collection



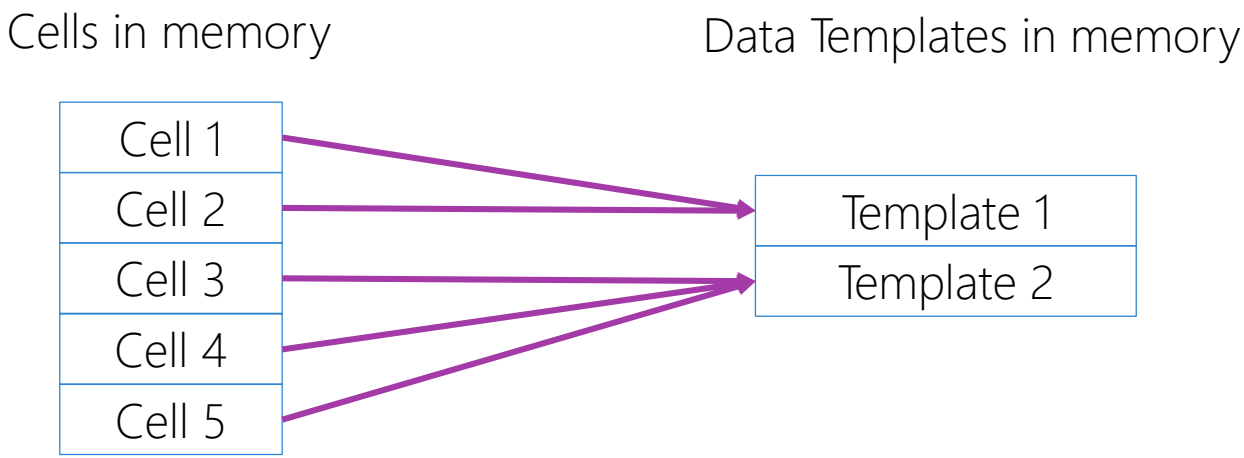
# Recycle Element

- The **`RecycleElement`** caching strategy tells the **`ListView`** to reuse cells that have scrolled off-screen



# Recycle Element and Data Template

- The **`RecycleElementAndDataTemplate`** caching strategy tells the **`ListView`** to cache the cell's data template with cell



# Turn on ListView recycling

- Cell recycling must be set when the list view is created and can be done in XAML or in C#

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

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

# Enable Fast Renders (experimental)

- Xamarin.Forms fast renderers are optional optimized renderers for several common native Android controls

```
public class MainActivity :...
{
   protected override void OnCreate (Bundle bundle)
   {
      ...
      global::Xamarin.Forms.Forms.SetFlags("FastRenderers_Experimental");

      global::Xamarin.Forms.Forms.Init (this, bundle);
      LoadApplication (new SmartHome.App ());
   }
}
```

# Demonstration

Enable cell recycling

# Optimize your data source

- LINQ produces **IEnumerable** expressions that don't support random access and will be slow to retrieve individual items

```
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();
```

Can use **ToList()** to create a concrete list, or pass the result into as new **ObservableCollection<T>**

# Avoid complex layouts

- 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>
```

`Spacing` on the `StackLayout` gives us exactly the same result but minimizes the layout pass complexity

# 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>
```

# Remove 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

- Configure caching strategy
- Optimize layout
- Optimize properties and setters

# Thank You!

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

Microsoft