XAM280

Using ListView in Xamarin.Forms

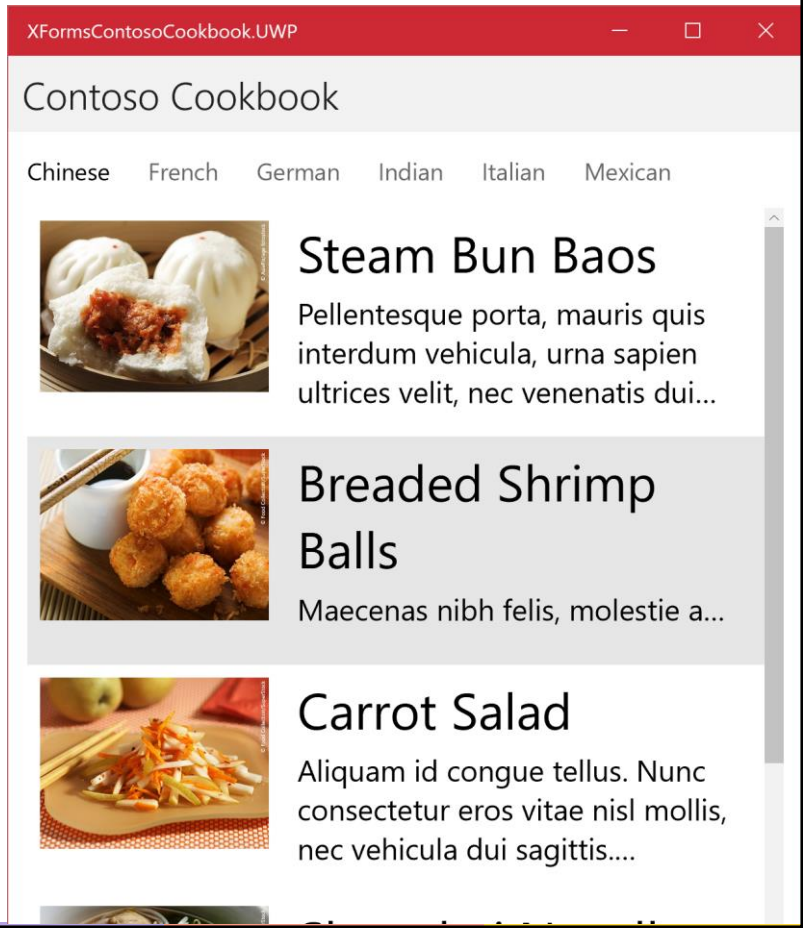
Download class materials from university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Display a collection with ListView
2. Add and remove items dynamically
3. Customize ListView rows



Display a collection with ListView

Tasks

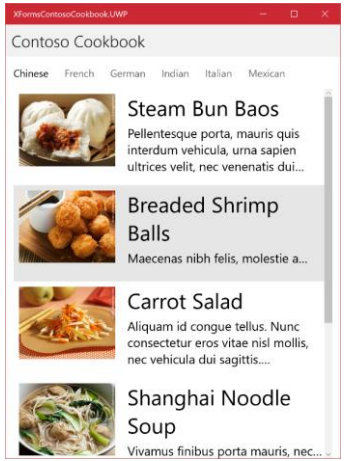
1. Provide data to a ListView
2. Manage selection in a ListView

Displaying Lists of Data

- **`ListView`** enables a common navigation style in your Xamarin.Forms applications – displaying homogenous data in a scrollable, interactive way

`UITableView`

`ListView`



`ListView`

Providing Data to a ListView

- ListView generates rows at runtime from a *collection source* assigned to the **`ItemsSource`** property

`ItemsSource` takes data in the form of an **`IEnumerable<T>`** - arrays, lists, LINQ expressions, etc.

Each object in the **`IEnumerable`** data source becomes a row in the **`ListView`**

Breaded Shrimp Balls

Maecenas nibh felis, molestie a est in, mattis congue neque. Vivamus at lorem viverra, viverra lacus in, pellentesque diam. Cum sociis na...

Carrot Salad

Aliquam id congue tellus. Nunc consectetur eros vitae nisl mollis, nec vehicula dui sagittis. Suspendisse at convallis turpis.

Shanghai Noodle Soup

Vivamus finibus porta mauris, nec condimentum purus facilisis vitae. Etiam leo velit, pretium vehicula venenatis nec, commodo nec ant...

Setting the ItemsSource property

- **ItemsSource** must be set to an **IEnumerable** data source

```
listView.ItemsSource = Cookbook.Recipes;
```

or

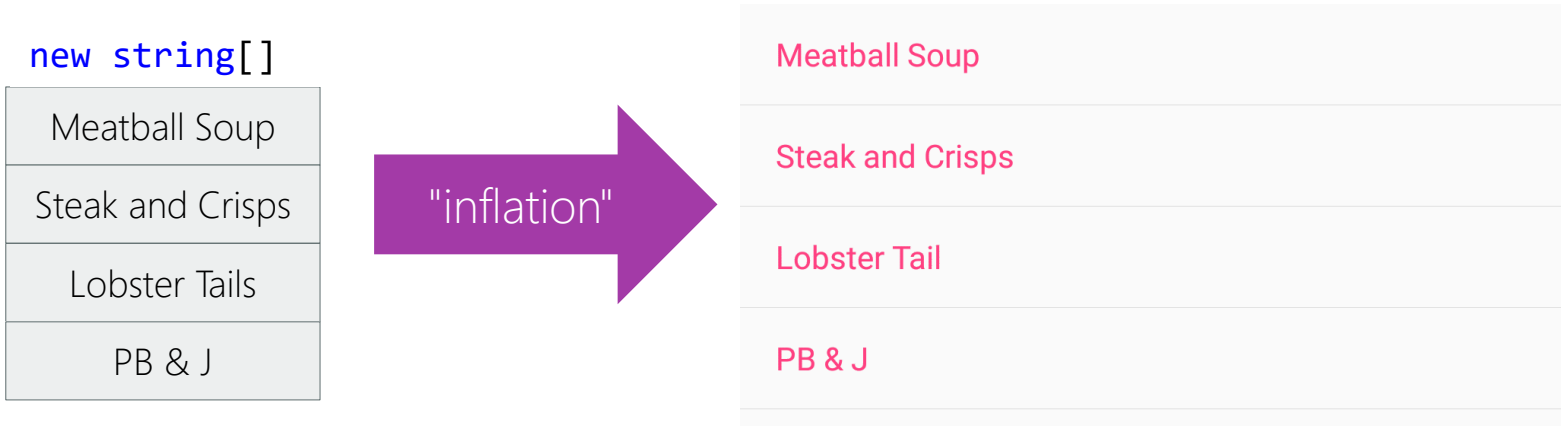
```
<ListView x:Name="listView" ...
  ItemsSource="{x:Static data:Cookbook.Recipes}" ...>
```

ItemsSource can data bind to a property of a model that exposes an **IEnumerable** or **IList**

```
public static class Cookbook
{
   public static IList<Recipe> Recipes
        { set; private set; }
}
```

Creating the rows

- The **`ListView`** will then generate a single row in the scrolling list for each item present in the collection



by default, it will use **`ToString`** on each item that is visible and create a **`Label`** to display the text in the **`ListView`**

Individual Exercise

Display a list of items with a ListView

Managing Selection

- Set or retrieve the current selection with the **`SelectedItem`** property

```
listView.SelectedItem = Cookbook.Recipes.Last();
...

Recipe currentRecipe = (Recipe) listView.SelectedItem;
```

- Can also use **data binding** to manage selection

```
<ListView ...
    SelectedItem="{Binding SelectedRecipe, Mode=TwoWay}">
```

No need to deal with selection events with this approach, can treat selection as "activation" and place code into your property setters

Dealing with Activation

- Can separate "activation" from selection using **`ItemTapped`** event – this can be useful for master / detail navigation

```
<ListView ItemTapped="OnRecipeTapped" ...>
```

```
async void OnRecipeTapped(object sender, ItemTappedEventArgs e)
{
    Recipe selection = (Recipe) e.Item;
    await Navigation.PushAsync(new DetailsPage(selection));
}
```

Individual Exercise

Selecting a row

Xamarin University

Summary

1. Provide data to a ListView
2. Manage selection in a ListView

Add and remove items dynamically

Tasks

- Add, remove and update data in the ListView
- Make UI-safe collection changes
- Modify collections in the background

Working with dynamic data

- Sometimes, the list of items we want to display is *dynamic* in nature – we add and remove elements over time as the application runs

New data available

Data removed

Data sorted/moved or changed

Adding and Removing ListView items

- There are no explicit APIs for adding and removing ListView items, instead you modify the collection of data in the **`ItemsSource`** property

```
Cookbook.Recipes.Add(new Recipe { Name = "Mac n Cheese" });
```

```
Cookbook.Recipes.RemoveAt(0);
```

```
Cookbook.Recipes[0] = new Recipe { Name = "Golden Heaven Food" }
```

Modifying Collections

- But .. adding, removing or replacing items in the collection at runtime *will not alter the UI* unless the collection reports **collection change notifications**

```
public static class Cookbook
{
   public static List<Recipe> Recipes
      { get; private set; }
}
```

List<T> doesn't know anything about Xamarin.Forms…

```
Cookbook.Recipes.Add(new Recipe { Name = "Lobster Bisque" });
```

… so this change only happens in the collection .. not the UI!

INotifyCollectionChanged

- Microsoft defined the **`INotifyCollectionChanged`** interface to provide this notification – any collections which supply data to a UI element must implement this interface

```
namespace System.Collections.Specialized
{
    public interface INotifyCollectionChanged
    {
        event NotifyCollectionChangedEventHandler CollectionChanged;
    }
}
```

ObservableCollection

- Can use **`ObservableCollection<T>`** as the underlying collection type – this implements the necessary collection change notifications

```
public static class Cookbook
{
   public static IList<Recipe> Recipes { get; private set;}

   static Cookbook() {
      Recipes = new ObservableCollection<Recipe>();
   }
}
```

Can expose an interface so implementation can be changed if / when necessary

Individual Exercise

Working with mutable lists

Xamarin University

Modifying collections

- Normally, changes to **`ObservableCollection<T>`** must be done on the UI thread – otherwise you will get an exception at runtime

```
void OnProcessRevisions(Recipe[] recipes)
{
    Device.BeginInvokeOnMainThread(() => {
        foreach (var r in recipes) {
            Recipes.Add(r);
        }
    });
}
```

Must make sure to switch to the UI thread before altering the collection data

Modifying collections in the background

- Alternatively, can instruct the binding system to manage that collection in a *thread-safe fashion*

```
BindingBase.EnableCollectionSynchronization(
    Recipes,
    null,
    (list, context, action, writeAccess) => {
        lock (list) {
          action();
        }
    }
);
```

Pass the instance of the collection that is assigned to the ListView

Modifying collections in the background

- Alternatively, can instruct the binding system to manage that collection in a *thread-safe fashion*

```
BindingBase.EnableCollectionSynchronizat
    Recipes,
    null,
    (list, context, action, writeAccess)
        lock (list) {
          action();
        }
    }
);
```

Can supply an optional *context* parameter which will be passed to the locking method each time, or use **null** if you don't need it

Modifying collections in the background

- Alternatively, can instruct the binding system to manage that collection in a *thread-safe fashion*

```
BindingBase.EnableCollectionSynchronization(
    Recipes,
    null,
    (list, context, action, writeAccess) => {
        lock (  ) {
```

Must pass in a delegate that the **ListView** will use to access the collection. Method is passed the Collection, Context, an **Action** to run, and whether this call will alter the collection (to distinguish read & write)

Modifying collections in the background

- Alternatively, can instruct the binding system to manage that collection in a thread-safe fashion

```
BindingBase.EnableCollectionSynchronization(
    Recipes,
    null,
    (list, context, action, writeAccess) => {
        lock (list) {
          action();
        }
    }
);
```

Method used to protect the collection – this can use whatever locking mechanism you prefer but must invoke the passed `Action`

Flash Quiz

Flash Quiz

① If you intend to alter the collection providing the data, you should use a _____ to make sure the UI is notified about the changes

a) List<T>

b) NotifyableCollection<T>

c) ObservableCollection<T>

d) Any collection type will work

Flash Quiz

① If you intend to alter the collection providing the data, you should use a _____ to make sure the UI is notified about the changes

a) List<T>

b) NotifyableCollection<T>

c) ObservableCollection<T>

d) Any collection type will work

Flash Quiz

② To add a new item to the ListView you can _____.

a) ListView.Items.Add(…)

b) ListView.ItemsSource.Add(…);

c) ListView.Add(…);

d) None of the above

Flash Quiz

② To add a new item to the ListView you can _____.

a) ListView.Items.Add(…)

b) ListView.ItemsSource.Add(…);

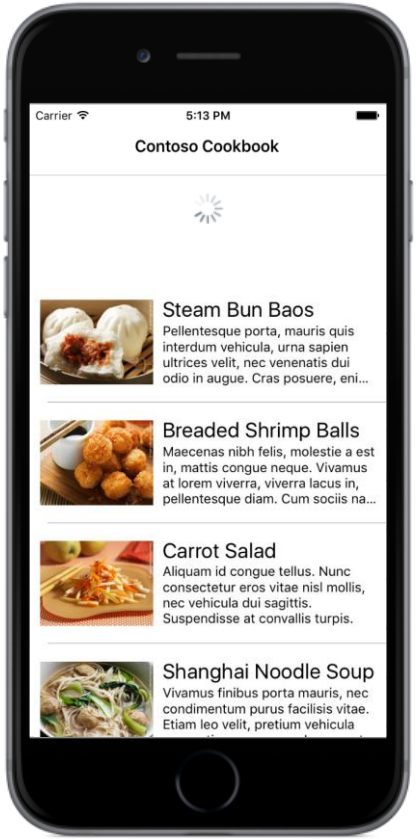
c) ListView.Add(…);

d) None of the above

Pull to refresh

- A very popular gesture used with **`ListViews`** that display external data is "pull-to-refresh" to get new data from the external source

- Refresh is activated by "pulling down" on the **`ListView`** – indicator is shown while the data is being updated



Implementing Pull to refresh

- Must turn on support through **`IsPullToRefreshEnabled`**

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

can be data bound or changed at runtime if you want to turn support on and off

Implementing Pull to refresh

- Control raises **Refreshing** event when refresh gesture is detected

```
<ListView ... IsPullToRefreshEnabled="True"
        Refreshing="OnRefreshing" >
```

```
void OnRefreshing(object sender, EventArgs e)
{
    ... // Code to do the refresh goes here ..
    ... // This is called on the UI thread!
}
```

Implementing Pull to refresh

- Must set **`IsRefreshing`** to false when refresh is complete

```
void OnRefreshing(object sender, EventArgs e)
{
    ... // Code to do the refresh goes here ..
    ((ListView)sender).IsRefreshing = false;
}
```

Implementing Pull to refresh [MVVM]

- Can also use **RefreshCommand** property to implement refresh logic as a MVVM-compatible command

```
<ListView ... IsPullToRefreshEnabled="True"
   IsRefreshing="{Binding IsRefreshing, Mode=TwoWay}" >
   RefreshCommand="{Binding RefreshCommand}" >
```

```
public class TheViewModel : INotifyPropertyChanged
{
   public ICommand RefreshCommand { get; private set; }
   public bool IsRefreshing ...
```

Manually starting a Refresh

- Can manually start and stop a refresh using the **`BeginRefresh`** and **`EndRefresh`** methods; this is useful if you have some other way to perform a refresh but want the same built-in experience

```
ListView lv;
void OnServerUpdatedData(object sender, EventArgs e)
{
   lv.BeginRefresh();
   ... // Update data
   lv.EndRefresh();
}
```

Group Exercise

Add Pull to Refresh support

Xamarin University

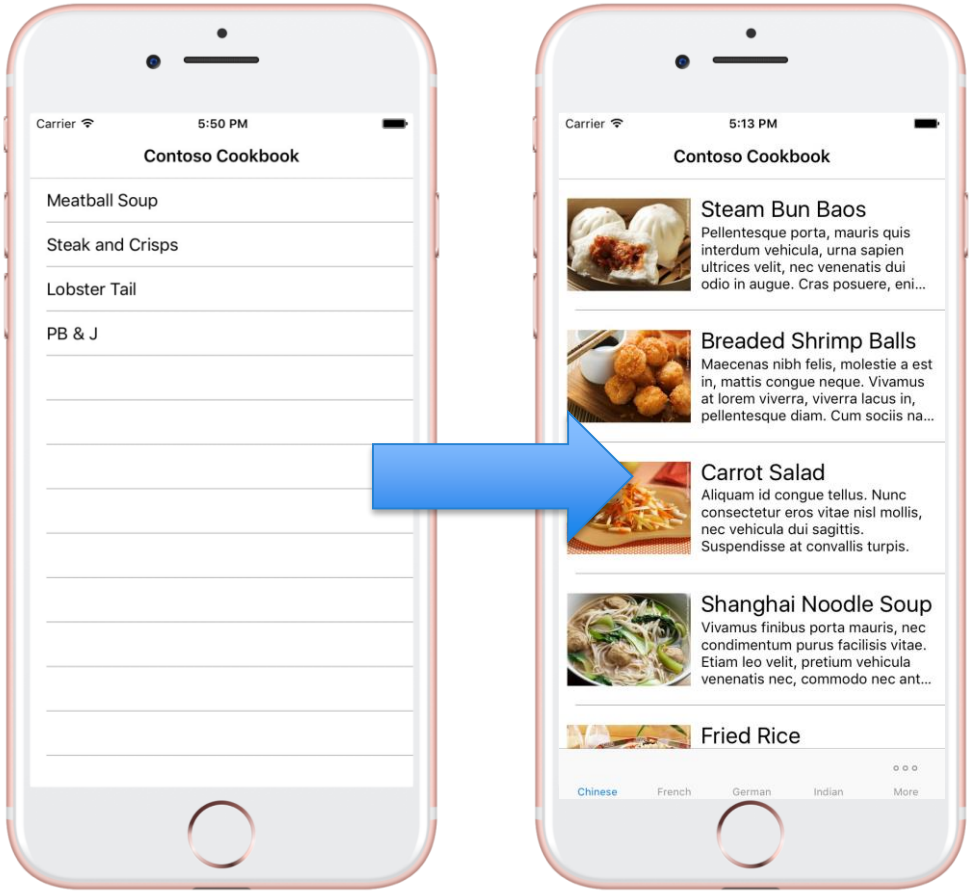
Summary

- Add, remove and update data in the ListView
- Make UI-safe collection changes
- Modify collections in the background

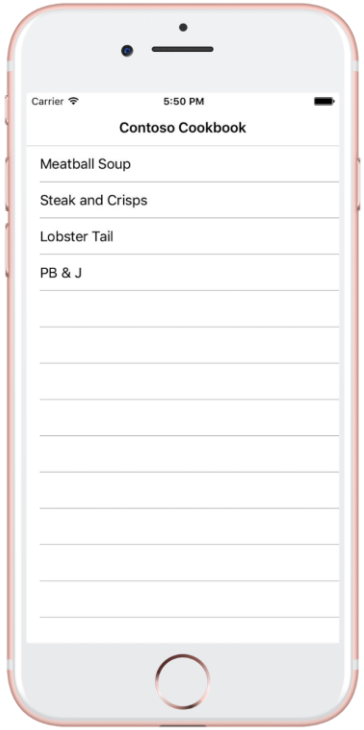
Customize ListView rows

Tasks

1. Alter the row visuals
2. Use Data Templates
3. Change the ListView separator
4. Use built-in cell templates



Displaying ListView Items



- Default behavior for **`ListView`** is to use **`ToString()`** method and display a single string for each row
- Acceptable for basic data, but has little to no visual customization of colors, position, or even data displayed

Mutating data

- A second problem with using the default visualization is that it is considered *read-only*

- If the data *inside* the object is changed at runtime, the **`ListView`** will *not* see the change – *even if a property change notification is raised!*

Try changing the data for a record and then going back … see what happens ..

Altering the row visuals

- Can customize the row by setting **`ItemTemplate`** property

`ItemTemplate` describes visual representation for each row

Breaded Shrimp Balls

Maecenas nibh felis, molestie a est in, mattis congue neque. Vivamus at lorem viverra, viverra lacus in, pellentesque diam. Cum sociis na...

Carrot Salad

Aliquam id congue tellus. Nunc consectetur eros vitae nisl mollis, nec vehicula dui sagittis. Suspendisse at convallis turpis.

Shanghai Noodle Soup

Vivamus finibus porta mauris, nec condimentum purus facilisis vitae.

Setting an ItemTemplate [XAML]

- **DataTemplate** provides visual "instructions" for each row

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

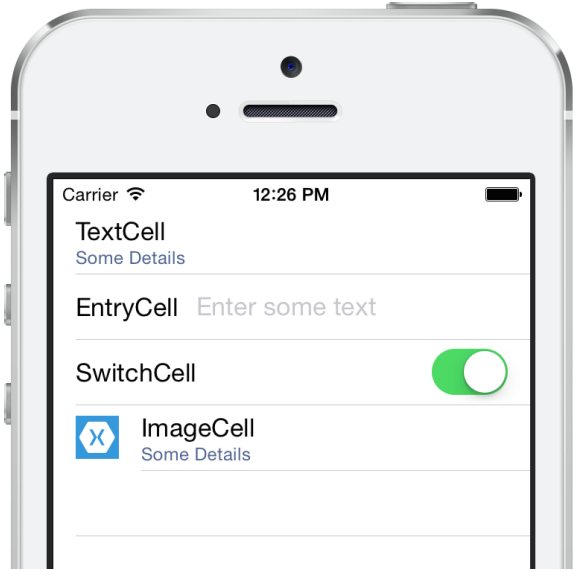
```
listView.ItemTemplate =
      new DataTemplate(...);
```

ListView uses the **DataTemplate** definition to create the runtime visualization, once per row in the **ItemsSource**

Data Template

- **`DataTemplate`** must describe a **`Cell`**, several built-in variations available

`TextCell`	Text + Details
`EntryCell`	Editable Text + Label
`SwitchCell`	Switch + Label
`ImageCell`	Image + Text + Details



Providing Data

- Cell provides "template" for each row, bindings used to fill in the details

```
<ListView.ItemTemplate>
   <DataTemplate>
      <TextCell Text="{Binding Name}"
           DetailColor="Gray" Detail="{Binding PrepTime}" />
   </DataTemplate>
</ListView.ItemTemplate>
```

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

Setting an ItemTemplate [C#]

- Can create and assign a **DataTemplate** in C# for more dynamic content or if you prefer to not use XAML

```
var dt = new DataTemplate(typeof(TextCell));
dt.SetBinding(TextCell.TextProperty, "Name");
dt.SetBinding(TextCell.DetailProperty, "PrepTime");
dt.SetValue(TextCell.DetailColorProperty, Color.Gray);

contactList.ItemTemplate = dt;
```

Use **SetBinding** for data-bound values that come from the **BindingContext** and **SetValue** for static values to set in the template

Defining custom cells

- Can use derived class to keep bindings with definition

```
listView.ItemTemplate =
    new DataTemplate(typeof(CustomTextCell));
```

```
public class CustomTextCell : TextCell
{
   public CustomTextCell() {
      this.SetBinding(TextCell.TextProperty,
                      new Binding("Name"));
      this.SetBinding(TextCell.DetailProperty,
                      new Binding("PrepTime"));
   }
}
```

Bindings can be set in the constructor with this approach

Setting an ItemTemplate [C#]

- **DataTemplate** can also use a callback function – this can be used to dynamically select a specific template based on runtime characteristics

```
listView.ItemTemplate = new DataTemplate(
    () => new TextCell()) {
      Bindings = {
        { TextCell.TextProperty, new Binding("Name") },
        { TextCell.DetailProperty, new Binding("Email") },
     }
};
```

Can set a **Bindings** dictionary property to establish the required bindings

Flash Quiz

Flash Quiz

① Data Templates can be defined in code or XAML

a) True

b) False

Flash Quiz

① Data Templates can be defined in code or XAML

a) True

b) False

Flash Quiz

② For ListView, the Data Template must define a _____ type

a) View
b) Visual
c) Cell
d) ViewCell

Flash Quiz

② For ListView, the Data Template must define a _____ type

a) View

b) Visual

c) Cell

d) ViewCell

Flash Quiz

③ Which is <u>not</u> a built-in Cell type?

a) TextCell

b) ImageCell

c) SliderCell

d) All of these are available

Flash Quiz

③ Which is not a built-in Cell type?

a) TextCell

b) ImageCell

c) SliderCell

d) All of these are available

Individual Exercise

Using the built-in ImageCell

Changing the separator

- Xamarin.Forms supports the ability to change the line that separates each displayed entry through two properties:

```
<ListView ... SeparatorVisibility="None">
```

Can be None or Default, currently there is no option for Always as not all platforms have separators as part of their UX

Changing the separator

- Xamarin.Forms supports the ability to change the line that separates each displayed entry through two properties:

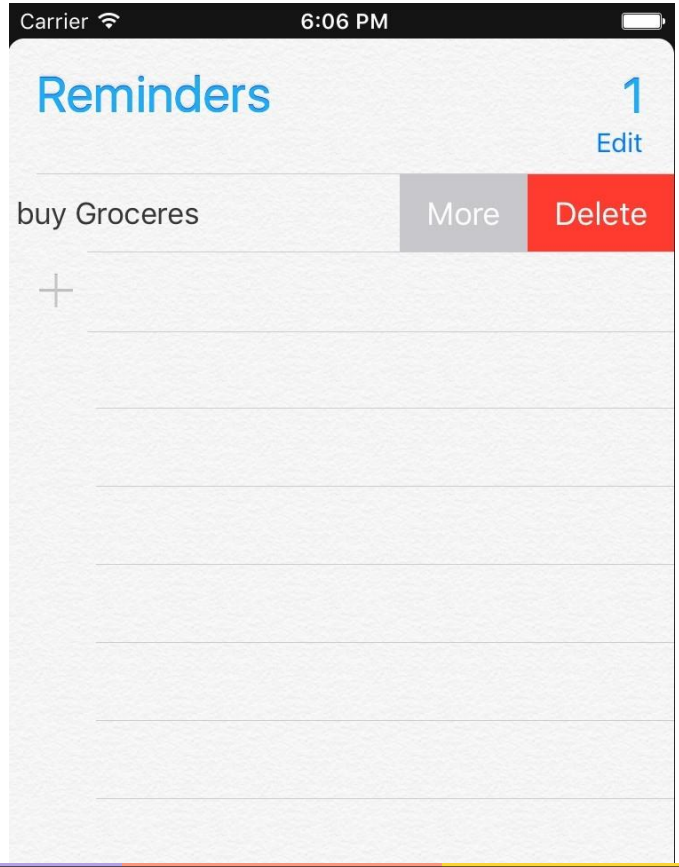
```
<ListView ... SeparatorVisibility="None">
```

```
<ListView ... SeparatorColor="#90C0C0C0">
```

Can supply an Argb value to set the color, or one of the known colors – including **Color.Default**

Context actions

- Rows can have *actions* associated with them which are displayed through "swipe to the left" gesture (iOS) or a long-click gesture (other platforms)
- Allows for one or more "actions" to be displayed and invoked inline with the row
- Applied through a **`ContextAction`** property array on the **`Cell`** definition



Adding Context Actions

- Each item in the **`ContextActions`** collection is a **`MenuItem`** – can set **`Text`** and provide an event handler to process the action

```
<TextCell ...>
    <TextCell.ContextActions>
       <MenuItem Clicked="OnMarkAsRead" Text="MarkAsRead" />
       <MenuItem Clicked="OnDelete" Text="Delete"
                  IsDestructive="true" />
   </TextCell.ContextActions>
</TextCell>
```

Button is rendered with platform "danger" background, on iOS this results in a red button

Getting the data out of the handler

- Can grab **BindingContext** from **MenuItem** sender to determine the data item being interacted with in the event handler

```
void OnDelete(object sender, EventArgs e)
{
   MenuItem item = (MenuItem)sender;
   Recipe recipe = (Recipe)item.BindingContext;
   Cookbook.Recipes.Remove(recipe);
}
```

Never forget that the **BindingContext** is the underlying *model* data for the row – this is almost always what you want to work with

Commanding support in MenuItem

- MenuItem also has **Command** and **CommandParameter** properties useful for MVVM style implementations

```
<TextCell ...>
    <TextCell.ContextActions>
        ...
       <MenuItem Text="Delete" IsDestructive="True"
                 Command="{Binding DeleteCommand}"
                 CommandParameter="{Binding .}" />
   </TextCell.ContextActions>
</TextCell>
```

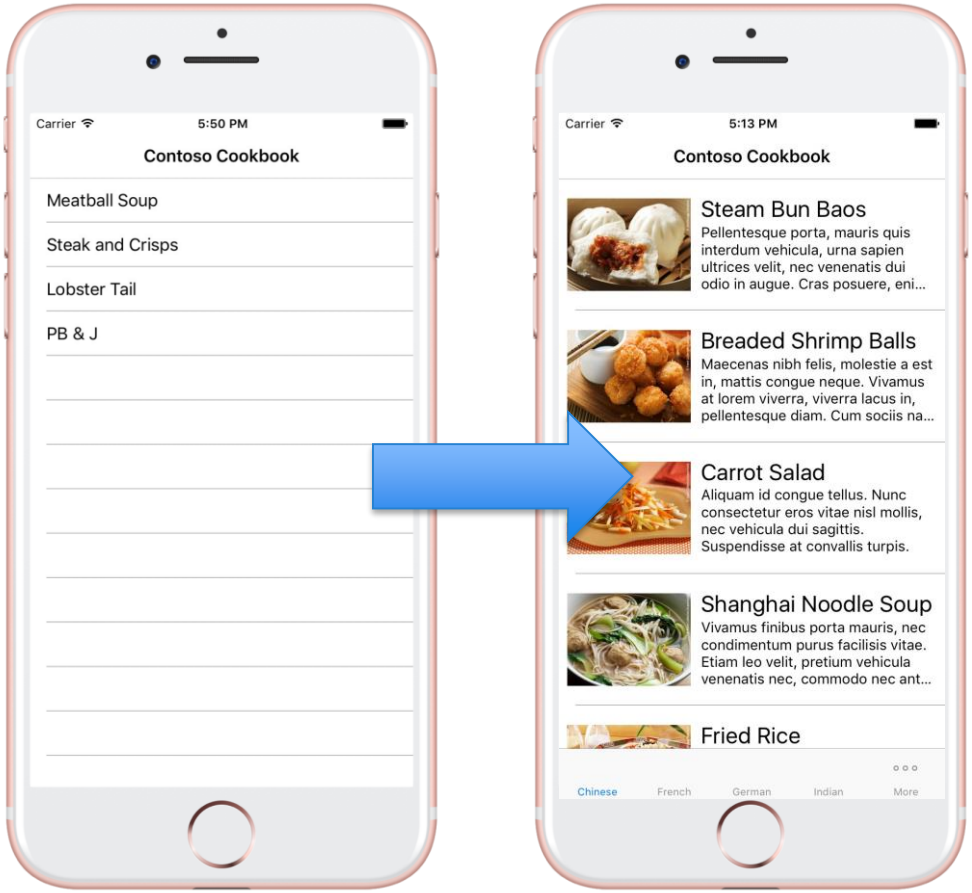
Can bind **CommandParameter** to **BindingContext** to get access to the specific element this action is being invoked on

Homework Exercise

Add a context action to each row

Summary

1. Alter the row visuals
2. Use Data Templates
3. Change the ListView separator
4. Use built-in cell templates



Thank You!

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

Microsoft