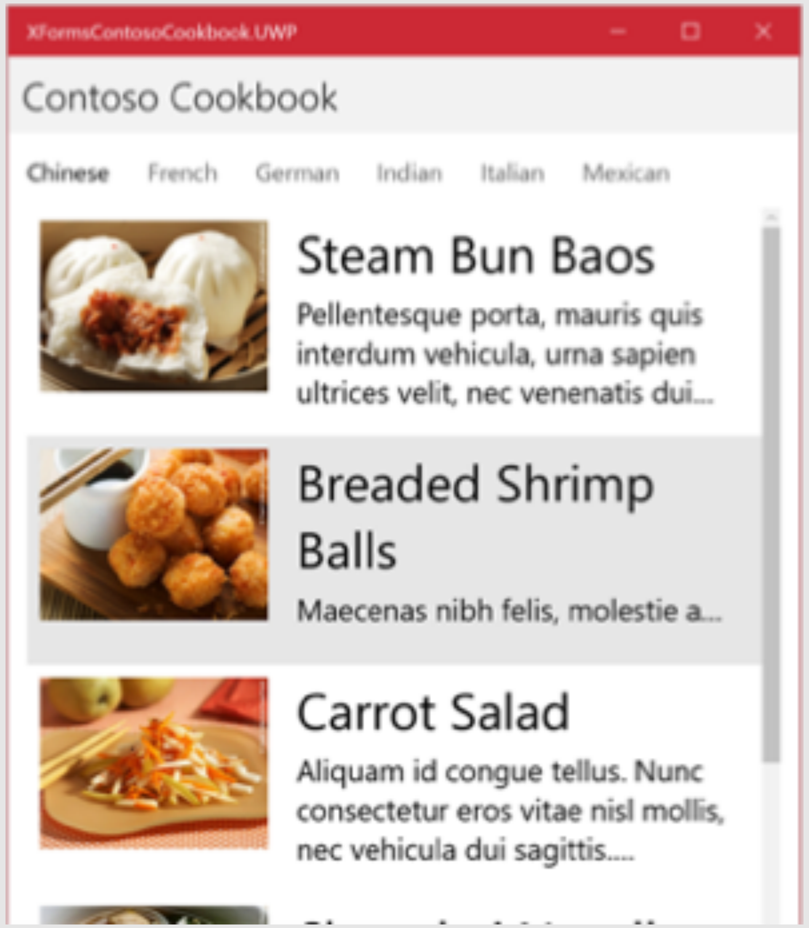
Displaying collections in Xamarin.Forms

Displaying Lists of Data

Many applications display scrollable lists of homogenous data, for example to show master/detail lists

Xamarin.Forms includes the **`ListView`** control which maps to the native list control for this purpose



Steam Bun Baos

Pellentesque porta, mauris quis interdum vehicula, urna sapien ultrices velit, nec venenatis dui...

Breaded Shrimp Balls

Maecenas nibh felis, molestie a...

Carrot Salad

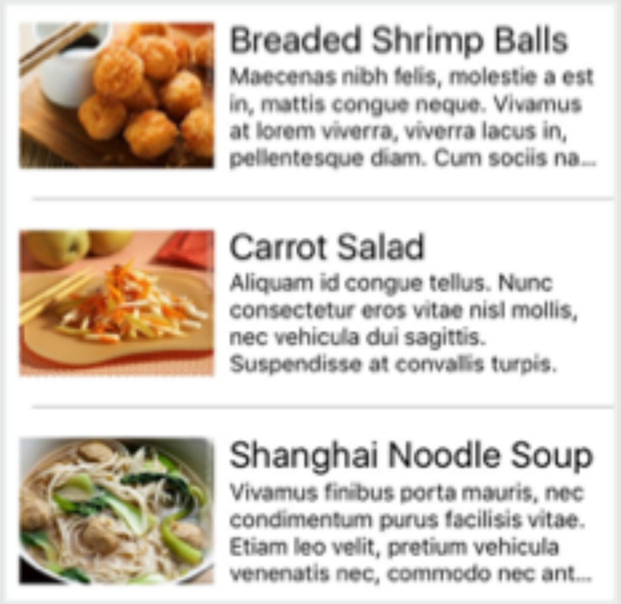
Aliquam id congue tellus. Nunc consectetur eros vitae nisl mollis, nec vehicula dui sagittis....

Providing Data to a ListView

`ListView` generates rows at runtime from a collection source

`ItemsSource` takes data in the form of an **`IEnumerable<T>`**

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

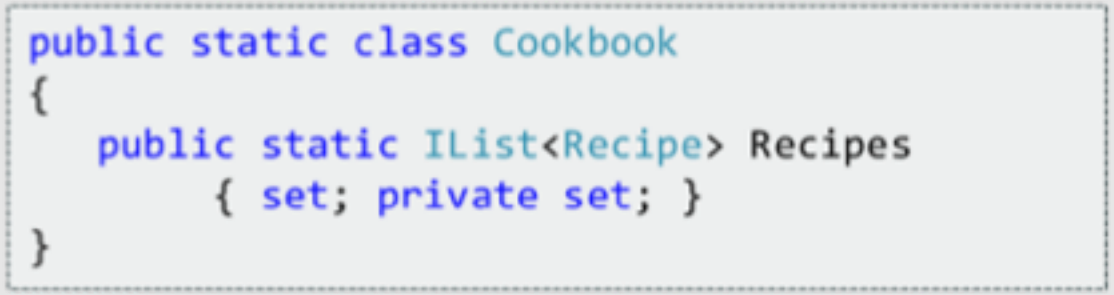
`ItemsSource` must be set to an `IEnumerable` data source

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

or

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

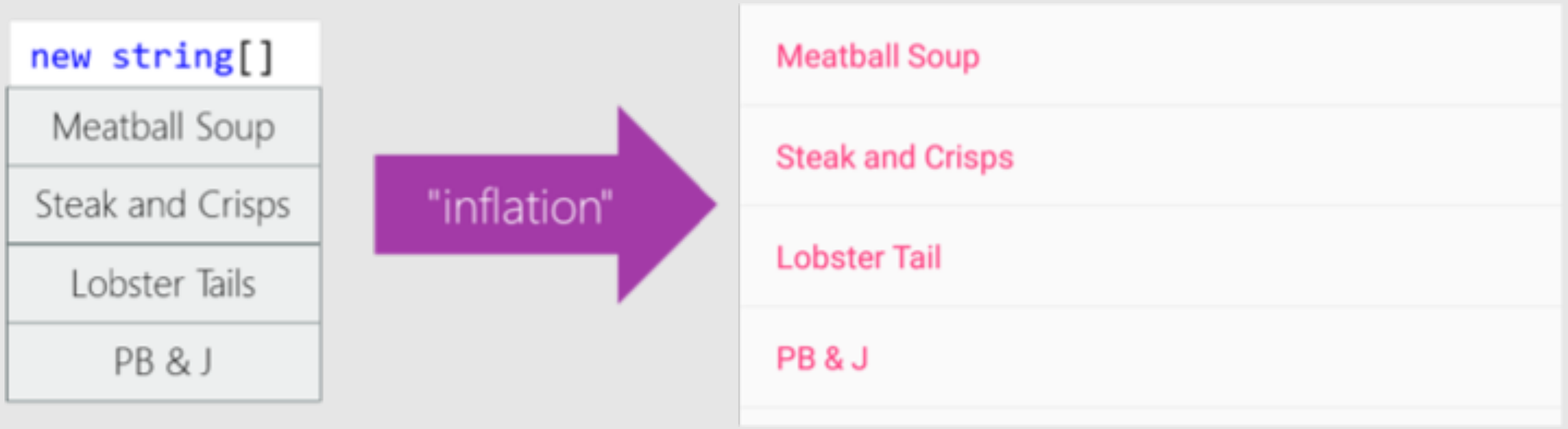
`ItemsSource` can data bind to a property of a model that exposes an `IEnumerable` or `IList`



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

Creating the rows

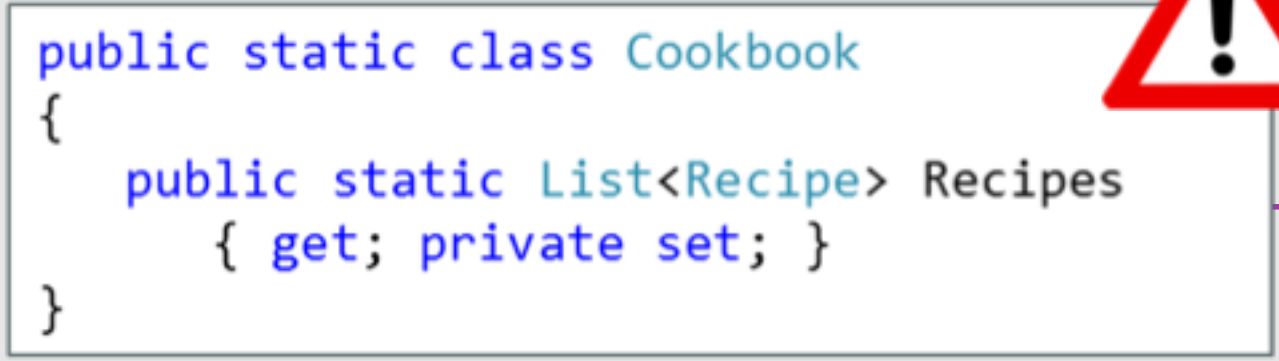
The **`ListView`** will then generate a single row in the scrolling list for each item present in the collection



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

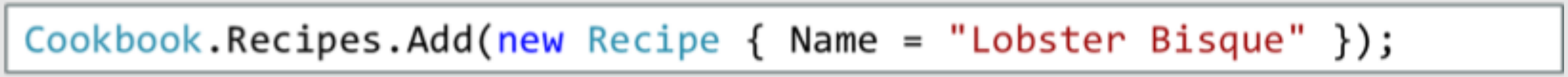
Modifying Collections

Just like property changes, 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!

ObservableCollection

Can use **`ObservableCollection<T>`** as the underlying collection type, this is just like **`List<T>`** but it also provides collection change notifications

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

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

Exercise #17

Add a Page with a ListView to display quiz results

Managing Selection

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

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

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

... or 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

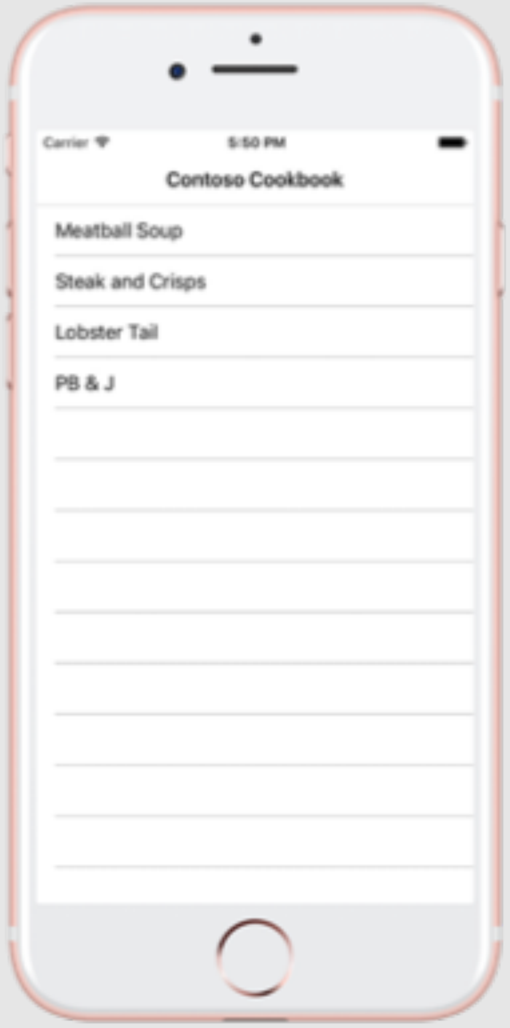
```
<ListView ItemTapped="OnRecipeTapped" ...>
```

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

Exercise #18

Display question explanation when tapping an item"

Displaying ListView Items



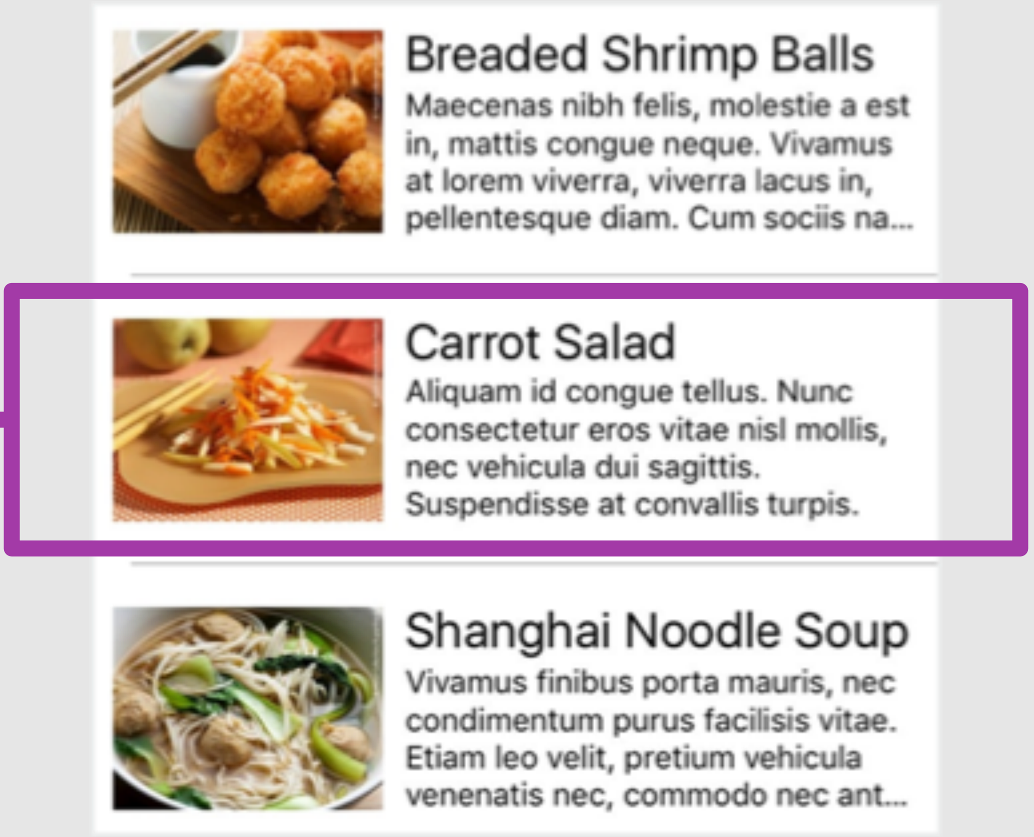
Default behavior for **`ListView`** is to use **`ToString()`** method and display a single string for each row

Acceptable for basic data, but has little to no visual customization of colors, position, or even data displayed

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. Etiam leo velit, pretium vehicula venenatis nec, commodo nec ant...

Setting the ItemTemplate property

`DataTemplate` provides visual "instructions" for each row

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

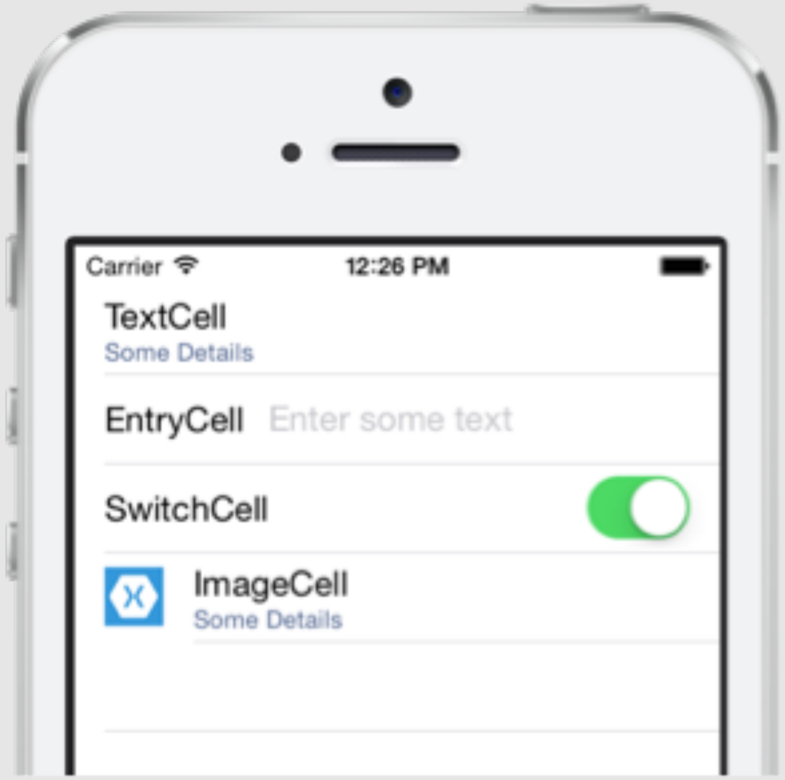
```
contactList.ItemTemplate =
        new DataTemplate(...);
```

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

Data Template

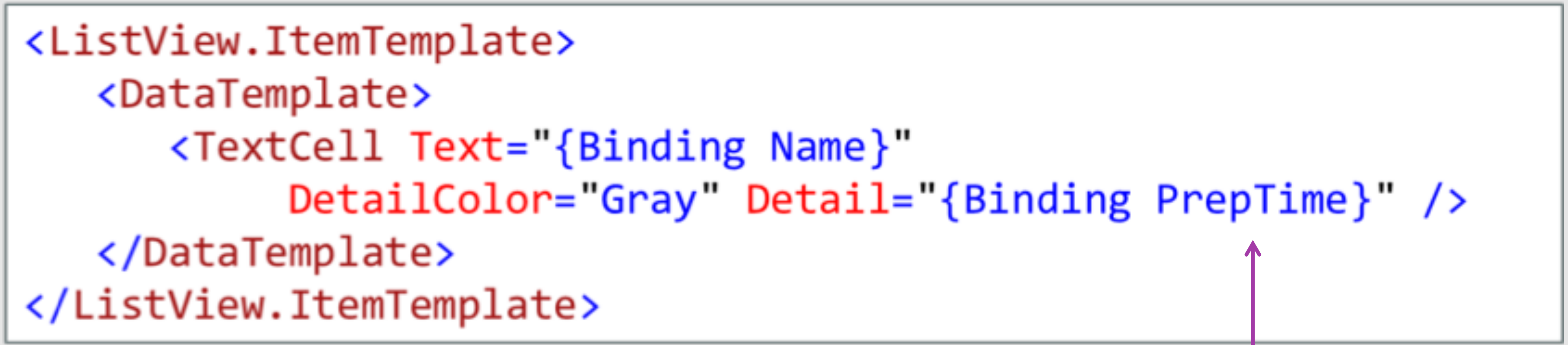
`DataTemplate` must describe a **`Cell`**, several built-in variations available

`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>
```

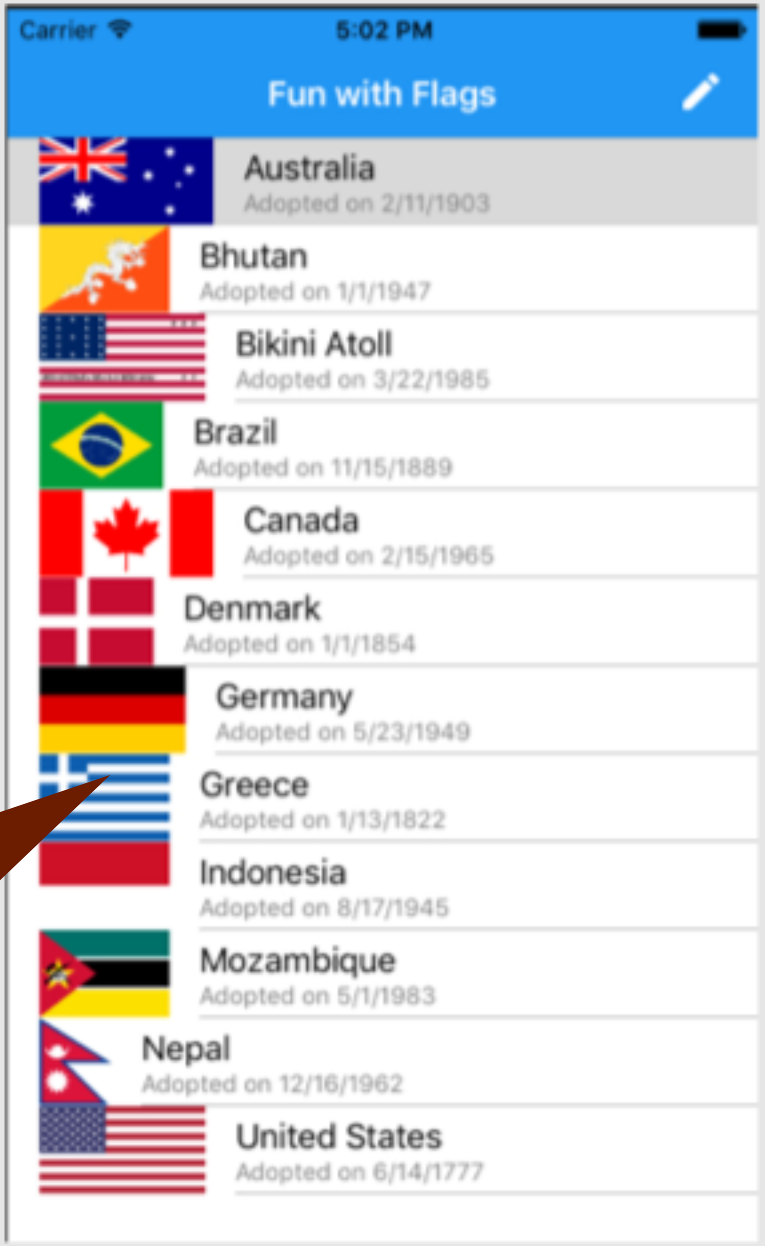
`BindingContext` for the generated row will be a single item from the `ItemsSource`

Customizing the View

Sometimes we need to customize the cell template

- does not fit the data
- need custom layout or colors
- maybe just 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



Australia
Adopted on 2/11/1903
Bhutan
Adopted on 1/1/1947
Bikini Atoll
Adopted on 3/22/1985
Brazil
Adopted on 11/15/1889
Canada
Adopted on 2/15/1965
Denmark
Adopted on 1/1/1854
Germany
Adopted on 5/23/1949
Greece
Adopted on 1/13/1822
Indonesia
Adopted on 8/17/1945
Mozambique
Adopted on 5/1/1983
Nepal
Adopted on 12/16/1962
United States
Adopted on 6/14/1777

Introducing: ViewCell

Can use **`ViewCell`** style for a custom visualization of any type

```
<DataTemplate>
   <ViewCell>
      <StackLayout Padding="5">
       <Label Font="20" TextColor="Black" Text="{Binding Country}" />
       <Label Font="14" TextColor="Blue" Text="{Binding DateAdopted}" />
      </StackLayout>
   </ViewCell>
</DataTemplate>
```

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

Working with visual properties

Assume a business requirement is to change the color of the person's name in the UI if they are a main character in the show

```
partial class PersonViewModel
{
    public Color NameColor { get; }
}
```

Avoid this! `Color` is a Xamarin.Forms specific type

```
partial class PersonViewModel
{
   public string NameColor { get; }
}
```

… this is better but still not ideal – colors should be determined by the designer role and view code

What we *really* want to do here is to have our UI change based on state properties such as `bool` or enumerations – we could do this with bindings and value converters

Working with visual properties

Assume a business requirement is to change the color of the employee's name in the UI if they are a supervisor

```
partial class
{
    public Co
}
```

```
partial class PersonViewModel
{
    public bool IsMainCharacter {
        get { ... }
    }
}
```

Let's expose a boolean property indicating whether the importance of the character to the show

is a
pecific type

… this is better but still not ideal – colors should be determined by the designer role and view code

```
iewModel

olor { get; }
}
```

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsMainCharacter}"
                   Value="True">
         <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">

                          ype="Label"
                          ="{Binding IsMainCharacter}"
                   Value="True">
         <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Assign default value – this is used when no trigger is matched

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsMainCharacter}"
                    lue="True">
                    erty="TextColor" Value="Blue" />

   </Label.Triggers>
</Label>
```

Can have zero or more *triggers* in the Triggers property

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsMainCharacter}"
                   Value="True">
                      rty="TextColor" Value="Blue" />
```

`DataTrigger` identifies a specific binding value: a public property in our ViewModel

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextColor="Gray">
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding IsMainCharacter}"
                   Value="True">
         <Setter Property="TextColor     ue="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

… and a comparison test for that binding; e.g. when IsMainCharacter = true

Visual Behavior through properties

Data Triggers support dynamic UI property changes based on bindings with conditional tests

```
<Label Text="{Binding Name}" TextC
   <Label.Triggers>
      <DataTrigger TargetType="Label"
                   Binding="{Binding I    inCharacter}"
                   Value="True">
        <Setter Property="TextColor" Value="Blue" />
      </DataTrigger>
   </Label.Triggers>
</Label>
```

Has one or more setters to apply when the trigger condition is matched – this will change property values at runtime

This is completely dynamic and is driven completely through the binding engine – so if the property changes at runtime, the trigger is re-evaluated and applied or removed!

Exercise #19

Polish the results page using a DataTrigger

ListView performance

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

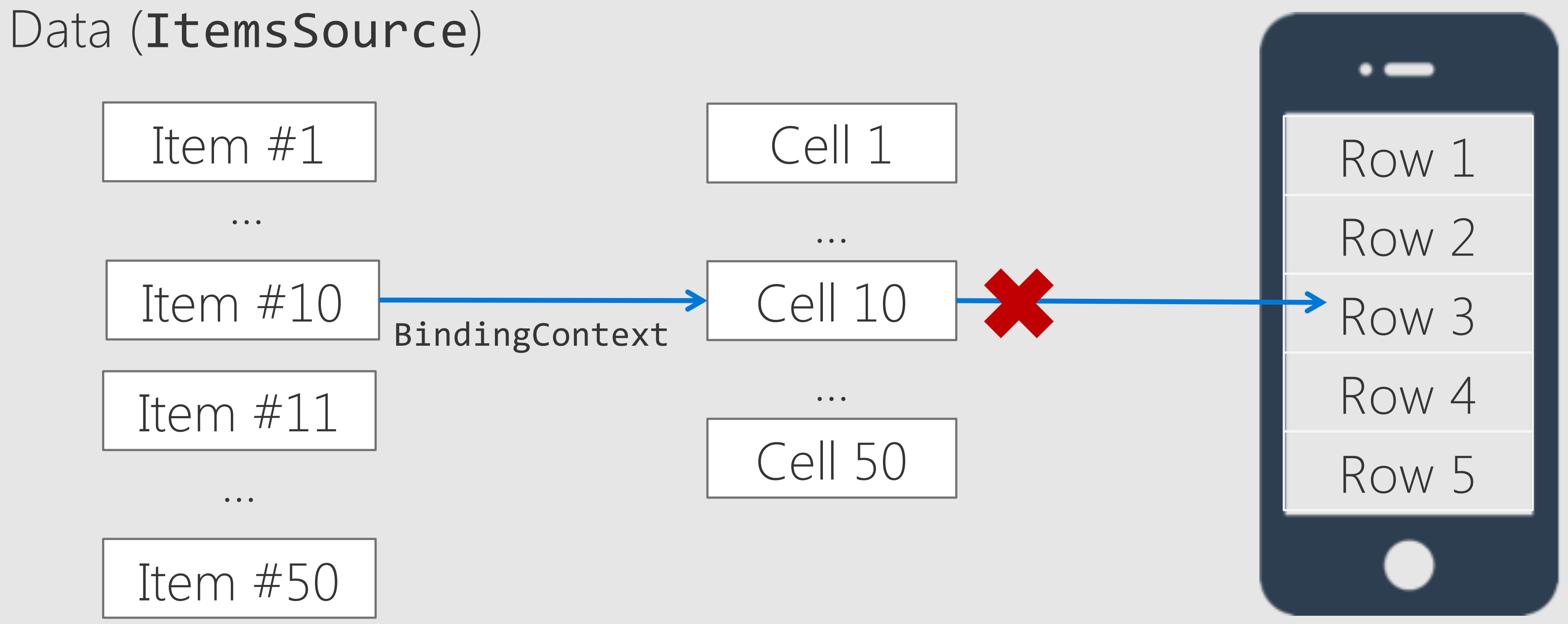
```
<ListView CachingStrategy="RecycleElement" ... />
```

or

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

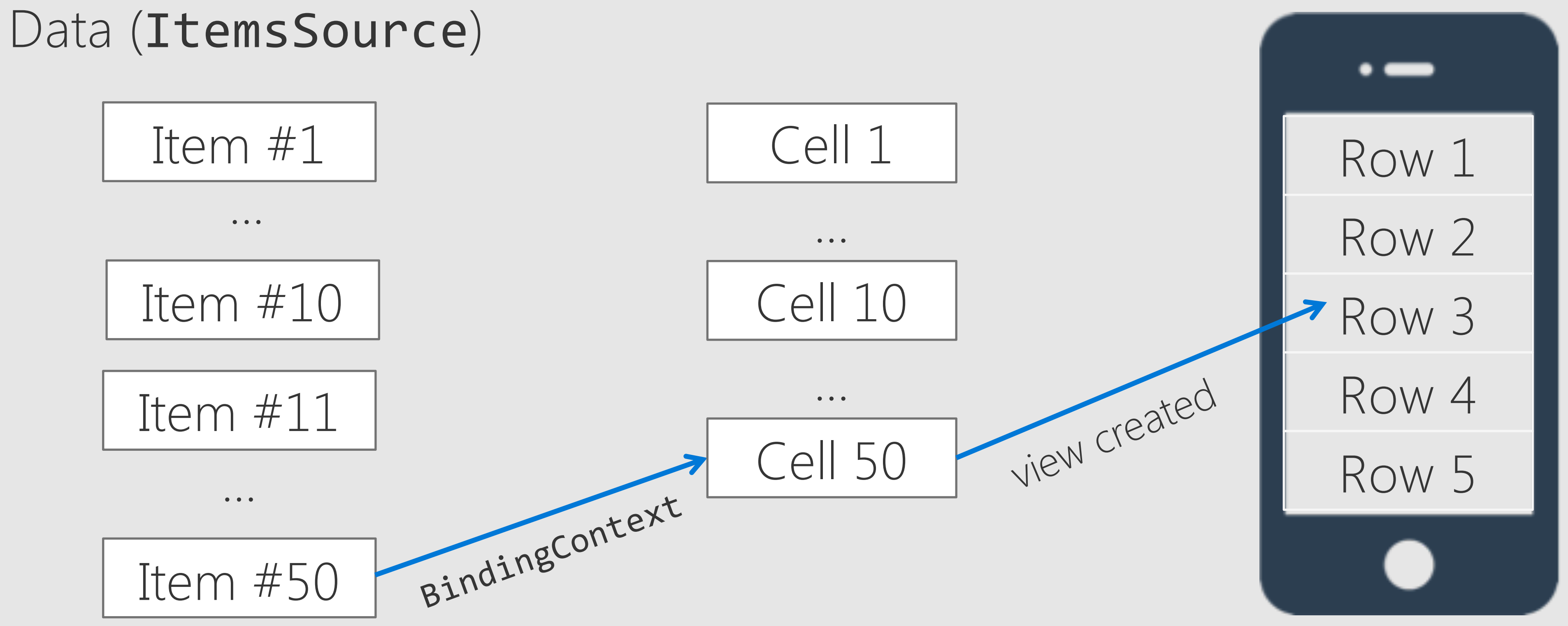
When cell recycling is OFF

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



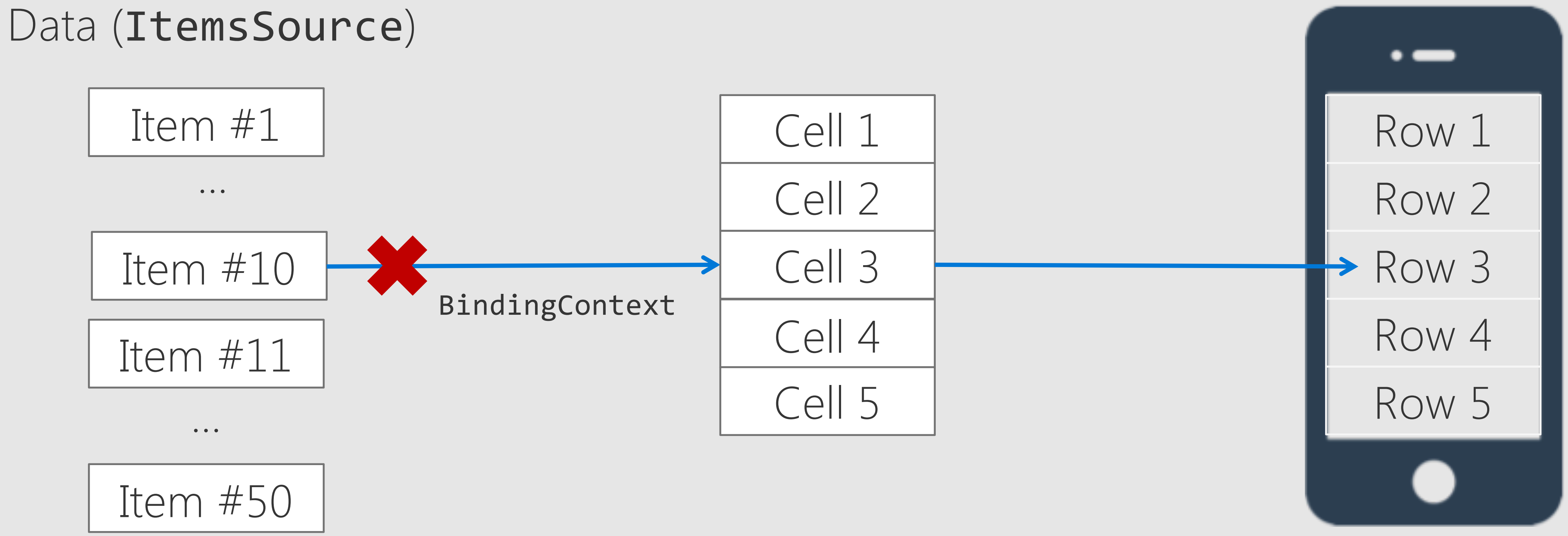
When cell recycling is OFF

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



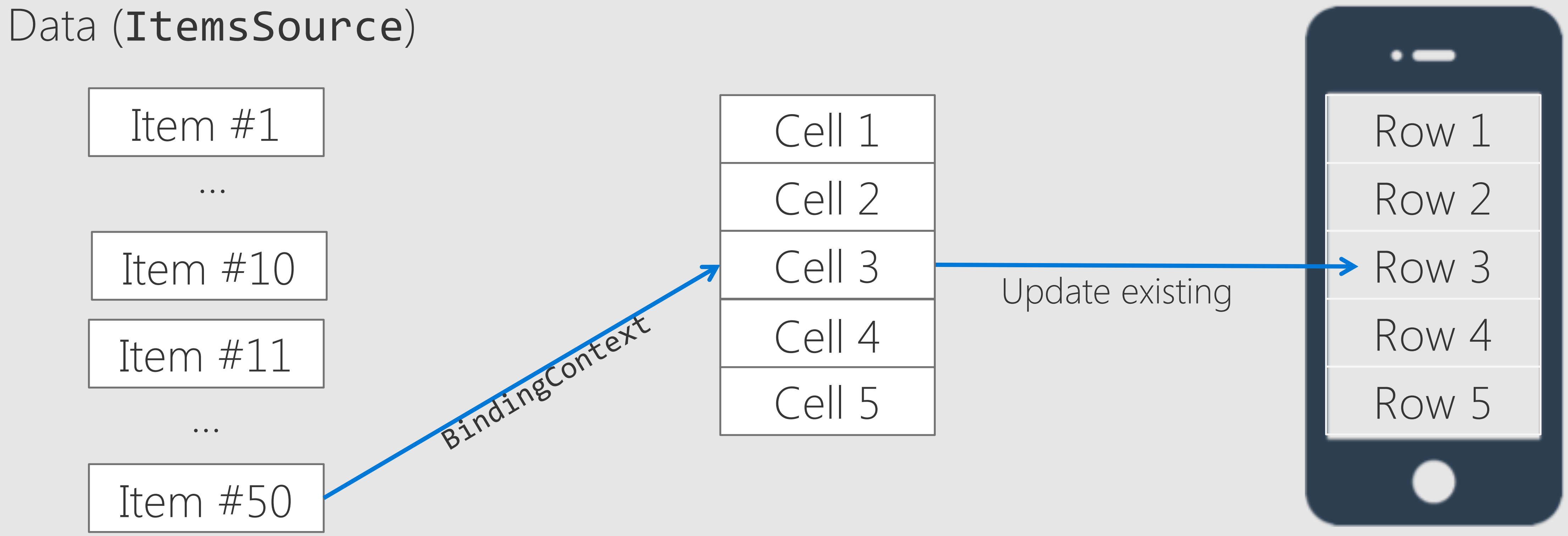
When cell recycling is ON

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



Exercise #20

Turn on cell recycling

Summary

ListView is a common control used to display scrollable, interactive lists of data

Each item in the supplied data collection is turned into a visual row

Supports row customization through the use of data templates and cells

Can enable cell recycling for high performance scrolling