## FUNDAMENTALS LESSON 02 SCRIPT

## ListViews and Adapters

INSTRUCTOR OVERVIEW

ESTIMATED TIME: 2 hours

LABS

* Lab 01 – Six projects building on each one.

## Lecture Prep

* Read through the **Lecture Preparation Steps** document prior to starting the class. You can find this in the Xamarin University internal repository.
* Copy the lab assets to your machine – go ahead and load up the completed project before you begin so you can show it when you start – you will be starting demo6, consider starting the emulator now as well.
* Copy the ListView Instructor project to your machine as well – this is the project you will be using which has the code removed so you can type it in.

## Slides 01-05: Script

### Slide 01: Xamarin University

### 

Hello there and welcome to Xamarin University!

My name is \_\_\_\_\_\_\_\_\_\_ and I'll be your instructor today. This session is an introduction to Android ListViews and Adapters. Because of their flexibility and power, ListView controls are one of the most commonly used controls in an Android application. Most applications use one or more of them to display lists of data. We won’t be able to talk about all the capabilities of this complex control in the limited time we have, but we will cover the main points of using it. At the end of the lecture, I’ll point you to some resources you can read in the Xamarin documentation for more information.

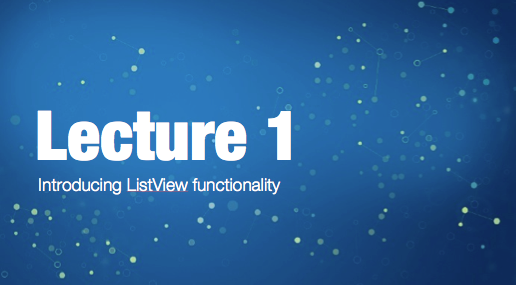
Throughout this session we will be working on a conference application. The goal is for you to work on the lab exercises with me so that at the end you have exactly the same project that I do there locally on your machine.

It is very important for everyone to follow along with the steps, so if you have a problem, a question or just need a little more time then please interrupt me and we’ll make sure you are able to keep up. I’m going to turn on everyone’s microphone so we can all have a very interactive class, but if you have to slip off or have a side conversation then please make sure to mute your line so it doesn’t distract everyone else. Finally, you should have received an email with a link for the lab resources. Make sure you’ve downloaded those and placed them in an easily accessible location on your computer. If you are using a virtual machine, then go ahead and copy the resources onto the VM.

Because the primary goal is for you to code along with me, I am going to assume that everyone’s system is setup properly and that you are either running on a Mac or Windows, or using the preconfigured VM from Xamarin. Please make sure you’ve run the TaskPro application sample and gotten it to execute properly in the Android Emulator – if you’ve not done that, we recommend you drop off the line and verify your environment and then reschedule this class for a later date.

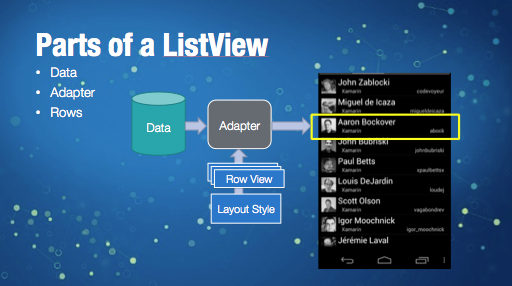
Are there any questions before we get started?

### Slide 02: Lecture 1



Our goal for today is to work with the ListView control, and by extension the different adapter types that you use to supply data to those controls. A ListView is used primarily to display *lists* of data. It is the equivalent to the ListBox in the Windows world. You can customize how the list is rendered, decide what happens when the user taps on an item and manage scrolling. Before we start looking at code, it’s helpful to understand all the pieces that make up a ListView.

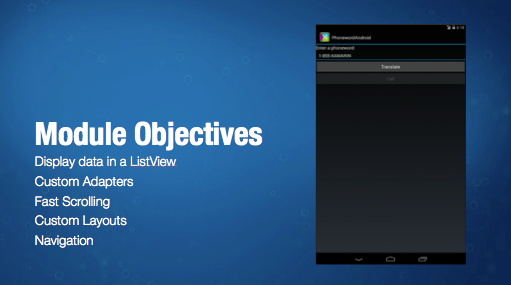
### Slide 03: Parts of a ListView



A ListView consists of several elements. First, it has the data to display. This can be as simple as an array of strings like you saw in the Introduction to Android lab, or you can use complex objects of your own design to manage the data. The point is, this is just data that you need to display on the screen and it’s going to be contained in some form of enumerable collection. For each item in the collection, Android will render a row. The row, surrounded by the yellow box on the slide, is the visual representation of the data in the list. Each row has it’s own View to render that data and the View, determined by the Layout Style, can either be a standard, built-in layout style, or a custom view created by the developer. In addition, rows can all use the same layout style, or they can use different styles based on the content.

The creation of the row’s View and population of the data into the row is done by the Adapter class. The adapter is a non-visual class which binds the data to the ListView control and decides how to format each row as it’s created. Android has several built-in Adapters, but it’s very common to create your own which you will see in the lab.

### Slide 04: Module Objectives



As I mentioned, we will be creating a conference application that will display a list of speakers and sessions. I will be creating the application as we go, but you have a pre-built solution that you will be able to work through to be able to keep the pace light and interactive. As a first step, let me show you the finished application we will be building. Go ahead and load the ListViewsAndroid solution that is located in the ListViewsAndroid\_Begin folder in the lab assets you downloaded. This is the solution we will be working in the entire time and it has six different projects in it, each one will show a specific feature of the ListView:

* **ListViewsInAndroid\_demo1** – Implement a basic table using ArrayAdapter<T>.
* **ListViewsInAndroid\_demo2** – Use a custom Adapter
* **ListViewsInAndroid\_demo3** – Use different built-in row cell styles.
* **ListViewsInAndroid\_demo4** – Using a custom row cell style.
* **ListViewsInAndroid\_demo5** – Add fast scrolling and section indexes.
* **ListViewsInAndroid\_demo6** – Use ListViews for navigation.

Switch to Xamarin Studio and show the completed Demo6 project – launch it in the emulator and go through each screen showing the navigation. Point out that each screen is a ListView displaying different content.

### Slide 05: Let's Get Started



Now that we’ve seen the final project, let’s go ahead and create it. Set your startup project to ListViewsInAndroid\_demo1 by right-clicking on the project and selecting Set As Startup Project. I am going to open a slightly different project, which doesn’t contain all the code so we can add it, and I can tell you what each piece does. Your solution has a set of TODO comments that you can see if you open your Task Pad or Task List if you are using Visual Studio. As we go through the lab exercise, I will point out each TODO we are on and you can double-click on it in the Task Pad and it will jump you to that code.

## Lecture 1: Lecture 1 Exercise

**INSTRUCTOR TASKS FOR THIS LAB**

* Work through six different exercises contain in the solution that build on each other. The projects are all in the \_Instructor solution – make sure you are familiar with the projects.

### Populating a ListView with Data

To start out, once you’ve got the first project selected as active, let’s build it and make sure it runs in the emulator. This will also make sure the emulator is up and running.

* Set ListViewsInAndroid\_demo1 as the active project.
* Build and run it – show a blank screen with no data.

To display a ListView, it must be added to the layout of an Activity. Then the Activity must implement the interface IListAdapter that provides the methods that the ListView calls to populate itself with data. Android provides a built-in ListActivity class that you can use without defining any custom layout XML or code. The ListActivity automatically creates the ListView and exposes a ListAdapter property to supply the row views to display. This property must be assigned to an Adapter class.

This first project is using a ListActivity as the main launching activity. Go ahead and open up the **SpeakersActivity.cs** file and you should find some commented out code, which creates an array of strings and then assigns, a StringArrayAdapter<string> to the ListAdapter property.

TODO: Demo1 – Step 1 – uncomment this to display data in the list

The simplest way to display data in a list is to use the built-in ArrayAdapter class to bind a collection of data to a ListView control. ArrayAdapter<T> uses three things to render a list:

* A context – a reference to the current activity
* A layout resource ID which identifies which of the built-in layouts to use to generate the row view. We’ll use one that just displays a single line of text.
* An array of data based on the generic type. We will be using an array of strings here, so each item in the array will end up as a TextView on the screen.

Open **SpeakersActivity.cs** and in the **OnCreate** method, add an array of strings (use a global class field) and assign a new StringArrayAdapter<string>

string[] items;  
  
protected override void OnCreate(Bundle bundle)  
{  
    base.OnCreate(bundle);  
  
    items = new string[] { "Name 1", "Name 2", "Name 3", "Name 4" };  
    ListAdapter = new ArrayAdapter<String>(this, Android.Resource.Layout.SimpleListItem1, items);  
}

* Build and run the app again to show the list of data.
* Tap on an item – show that it has no behavior.

### Responding to User Interactions

We’d like some behavior when we tap on the item – to enable that, we need to override the OnListItemClick method – this is part of the IListAdapter interface that is used to communicate with the ListView. All we need to do when we use a ListActivity is to override the method.

TODO: Demo1 – Step 2 – uncomment this to display a toast notification on row click

Stop the program and switch back to Xamarin Studio – override the OnListItemClick method and add in a Toast notification:

protected override void OnListItemClick(ListView l, View v, int position, long id)  
{  
    var item = items[position];  
    Toast.MakeText(this, item,  ToastLength.Short).Show();  
}

Build and run the app again and show the Toast notifications.

### Creating a Custom Adapter

Let’s switch to the next project in our solution – ListViewsInAndroid\_demo2.

Make the \_demo2 project the active project (right-click, select Set as Active)

In this project, our data is now a list of Speaker objects.

* Open the Speaker.cs source file in the Model folder and show the data structure.
* Show the Speakers.cs file as well which has a GetSpeakerData method.

The ArrayAdapter<T> class won’t know how to display these objects – it would use the ToString() implementation to try to turn each one into a text element.

If you like, you can add in a quick line to use ArrayAdapter<Speaker> against the data to show the ToString() being called. It’s an optional part but interesting to see if you are a fast typist. The code is presented below and would go into the SpeakersActivity.cs file:

ListAdapter = new ArrayAdapter<Speaker>(this, Android.Resource.Layout.SimpleListItem1,  
  Speakers.GetSpeakerData());

In order to customize the output we need to write an Adapter. This class has already been written for you in your project – it’s in the source file **SpeakersAdapter**.cs. We are going to create it now from scratch and go through each required override:

Add a new source file – SpeakersAdapter.cs to the project. Write the following code – explain each method as you write it:

public class SpeakersAdapter: BaseAdapter<Speaker>  
{  
    private readonly List<Speaker> data;  
    private readonly Activity context;  
  
    public SpeakersAdapter(Activity activity, IEnumerable<Speaker> speakers)   
    {  
        data = speakers.OrderBy(s => s.Name).ToList();  
        context = activity;  
    }  
  
    public override long GetItemId(int position)  
    {  
        return position;  
    }  
  
    public override Speaker this[int index]  
    {  
        get { return data[index]; }  
    }  
  
    public override int Count  
    {  
        get { return data.Count; }  
    }  
}

The last method we need to override is the GetView method. This is used to generate the view for each row presented in the ListView.

Add the override for GetView

When a ListView is displaying hundreds or thousands of rows, it would be a waste of memory to create a View object for each row, especially when only eight or so rows fit on the screen at a time. To avoid this situation, when a row disappears from the screen its view is placed in a queue for re-use. As the user scrolls, the ListView calls GetView to request new views to display. If a re-usable row is available then it will be passed in to GetView as the convertView parameter. If a re-usable row is not available, then null will be passed in, and we create a new view to display the row.

Custom adapter implementations should *always* re-use the convertView object before creating new views to ensure they do not run out of memory when displaying long lists.

When creating our view, we can either use an existing, predefined Android layout or create a custom view either in code or from an AXML file. Let’s start by using some of the existing layouts provided by the system.

Implement the following method to the code, explain the passed parameters and that the SimpleListItem1 layout used here has a defined TextView element which the Id **Resource.Id.Text1** which we can use to locate it and set the text. Also highlight (again) the reuse strategy used here.

public override View GetView(int position, View convertView, ViewGroup parent)  
{  
    var view = convertView;  
    if (view == null) {  
        view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, null);  
    }  
  
    var speaker = data[position];  
    var text = view.FindViewById<TextView>(Android.Resource.Id.Text1);  
    text.Text = speaker.Name;  
  
    return view;  
}

Next, we need to use this new Adapter in our code. There are two TODO items you need to do in your project for this, both in the SpeakersActivity.cs file:

TODO: Demo2 – Step 1 – Add Speaker Adapter

TODO: Demo2 – Step 2 – assign the adapter

Add the Speaker Adapter into your SpeakersActivity and use it in the OnCreate method:

public class SpeakersActivity : ListActivity   
{  
    SpeakersAdapter adapter;  
  
    protected override void OnCreate(Bundle bundle)  
    {  
        base.OnCreate(bundle);  
  
        adapter = new SpeakersAdapter(this, Speakers.GetSpeakerData());  
        ListAdapter = adapter;  
    }

}

Build and run the app to show the results. This has no activity when you tap – direct the student to uncomment the next step and run that to get the toast back. If you have time you can add the code below for the same thing, otherwise just move to the next task.

protected override void OnListItemClick(ListView l, View v, int position, long id)  
{  
    var item = adapter[position].Name;  
    Toast.MakeText(this, item, ToastLength.Short).Show();  
}

Break Time Check

### Changing the Cell Style

Let’s try out some of the other built-in layout styles for ListViews. Switch to the third demo project in the solution. Go ahead and open the SpeakersAdapter code – you should find a TODO comment here that is just before a series of styles to assign.

* Make sure to switch to \_demo3.
* In the SpeakersAdapter.cs file, use each of the following layout styles in the LayoutInflator.Inflate calls and build + run the application to show each layout style, make sure to point out the code in the GetView method that populates each piece.

Android.Resource.Layout.SimpleListItem2

Android.Resource.Layout.TwoLineListItem

Android.Resource.Layout.ActivityListItem

Break Time Check

### Creating a Custom Cell View

Most applications will want to have a custom layout for ListView cells. New layouts are defined using XML files in the Resources/layout folder, and then referenced in code. Switch to the ListViewsInAndroid\_demo4 project, we’re going to create a custom row view as part of our adapter implementation. In particularly, we’d like to display a headshot, company name, and the speaker’s Twitter handle.

In your project you have a custom AXML file called speaker\_row.axml. Go ahead and open that file to see the layout. I’m going to create that view here so you can see how it is built.

|  |  |
| --- | --- |
| Expand the Resources/Layout folder and add a new Android Layout file called speaker\_row.axml. It is helpful to turn on the Document Pad here to delete the LinearLayout, which is there by default. | |
| Delete the LinearLayout root element | |
| Drag a RelativeLayout into the design surface | |
| Drag an ImageView into the design surface – it should position on the left | Set the Id to @+id/headshotImageView  Set the **Layout\_Width** to **80dp**  Set the **Layout\_Height** to **80dp**  Check **Layout\_Center\_Vertical**  Check **Layout\_Align\_Parent\_Left** |
| Drag Text(Large) below ImageView in document view | Set the **Id** to  **@+id/speakerNameTextView**  **Layout\_To\_RightOf** to @+id/headshotImageView  **Layout\_Margin\_Left** to 5dp |
| Drag Text(Small) below speakerNameTextView in document view | Set the **Id** to @+id/companyNameTextView  **Layout\_To\_RightOf** to @+id/headshotImageView  **Layout\_Below** to @+id/speakerNameTextView  **Layout\_Margin\_Left** to 10dp |
| Drag Text(Small) below companyNameTextView in document view | Set the **id** to @+id/twitterTextView  **Layout\_To\_RightOf** to @+id/companyNameTextView  **Layout\_Below** to @+id/speakerNameTextView  **Layout\_Margin\_Left** to 15dp  **Layout\_Margin\_Right** to 5dp  Check **Layout\_Align\_Parent\_Right**  **Gravity** to Right |

Now, let’s use this new layout in our Adapter code.

// TODO: Demo 4 – Step 3 – remove this code with the built-in id

// TODO: Demo 4 – Step 4 – uncomment these lines to set the UI controls in the custom view

You can also delete the existing code that sets the view if you like.

Open the SpeakersAdapter.cs and change the inflate code to use the new AXML – remove the existing view assignment.

view = context.LayoutInflater.Inflate(Resource.Layout.speaker\_row2, null);

Add code to locate each UI element and set the proper data, this replaces the existing view.FindViewById<TextView> in the code.

var imageView = view.FindViewById<ImageView>(Resource.Id.headshotImageView);  
var headshot = GetHeadShot(speaker.HeadshotUrl);  
imageView.SetImageDrawable(headshot);  
  
var speakerNameView = view.FindViewById<TextView>(Resource.Id.speakerNameTextView);  
speakerNameView.Text = speaker.Name;  
  
var companyNameTextView = view.FindViewById<TextView> (Resource.Id.companyNameTextView);  
companyNameTextView.Text = speaker.Company;  
  
var twitterHandleView = view.FindViewById<TextView>(Resource.Id.twitterTextView);  
twitterHandleView.Text = "@" + speaker.TwitterHandle;

Build and run the application to show the new cell style.

### Fast Scrolling

When your list of data gets really long, it can be time consuming for users to scroll all the way through it. Android provides a fast-scrolling feature to help solve this problem – a “tab” appears while scroll that lets you move through the list much faster. Go ahead and switch to the ListViewsInAndroid\_demo5 project and open up the SpeakersActivity. There you should find a TODO:

//TODO: Demo 5 – Step 1 – uncomment to enable fast scrolling

Turn on Fast Scrolling at the end of OnCreate:

ListView.FastScrollEnabled = true;

Build and run the application to show the tab button when you scroll.

This allows the ListView scrolling speed to accelerate and move more quickly through the cells. This demo project has had additional speaker data added to better demonstrate how the fast scrolling tabs helps. However, when you get this much data, it can be hard to tell exactly where you are in the scrolling – so Android allows you to add an “index” to the ListView. This is like a tooltip popup that displays while you scroll and it helps the user identify where they are in long lists, especially when scrolling quickly.

To implement the index display the **BaseAdapter** subclass must implement the ISectionIndexer interface. Let’s do that now. Open the **SpeakersAdapter** class and add the ISectionIndexer interface onto the class. There is no TODO marker for this – just add it after the BaseAdapter<T> definition on the type. Next there are two TODO markers you need to uncomment to add the feature:

TODO: Demo 5 – Step 2 – Implement ISectionIndexer

TODO: Demo 5 – Step 3 – setup the indexer support

While you uncomment that code, let me add it to my version of the project.

Open **SpeakersAdapter** and add the **ISectionIndexer** definition to the class – show how to implement the interface automatically through the IDE (OPTION+ENTER with the cursor on the interface declaration).

Add the following implementation, explain that we will use pre-calculated values to display – however you could also dynamically generate data for the indexes if desired.

private string[] sections;  
private Java.Lang.Object[] sectionsObjects;  
private Dictionary<string, int> alphaIndex;  
  
public int GetPositionForSection(int section)  
{  
    return alphaIndex[sections[section]];  
}  
  
public int GetSectionForPosition(int position)  
{  
    int prevSection = 0;   
    for (int i = 0; i < sections.Length; i++) {  
        if (GetPositionForSection(i) > position && prevSection <= position) {   
            prevSection = i; break;   
        }  
        prevSection = i;   
    }   
    return prevSection;   
}  
  
public Java.Lang.Object[] GetSections()  
{  
    return sectionsObjects;  
}

Now that we’ve got the indexer implementation, let’s fill in our simple dictionary with data to display.

Add this code at the end of the constructor – explain how we are adding the first letter of each name to display.

alphaIndex = new Dictionary<string, int>();  
for (int i = 0; i < data.Count; i++) {  
    var key = data[i].Name[0].ToString();  // first character of name  
    if (!alphaIndex.ContainsKey(key))   
        alphaIndex.Add(key, i);  
}  
sections = new string[alphaIndex.Keys.Count];  
alphaIndex.Keys.CopyTo(sections, 0);  
sectionsObjects = new Java.Lang.Object[sections.Length];  
for (int i = 0; i < sections.Length; i++) {  
    sectionsObjects[i] = new Java.Lang.String(sections[i]);  
}

Build and run the application and show the index tooltip by scrolling the data.

### Using ListViews for Navigation

Many menu-driven apps use ListViews to help the user navigate through a set of hierarchical menu screens. The final project ListViewsInAndroid\_demo6 shows an example of how to construct a simple three level menu using ListViews. Android devices have a built-in mechanism for navigating backwards through these menu screens – older devices have a hardware ‘back’ button on the device and newer versions of Android provide a back button on the screen.

Set the last project to be active and build and run it. Show the navigation – then walk through the code starting with the **MenuActivity** which is the main launcher activity.

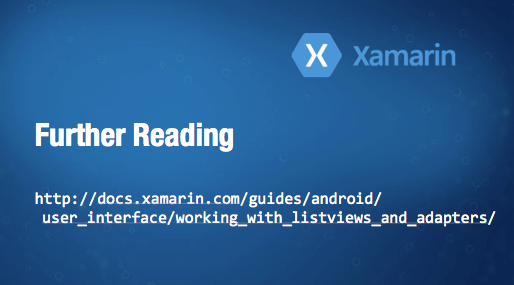
## Slides 06-08: Script

### Slide 06: Lab Summary



We’ve created a new Android application using Xamarin Studio, explored the default project template, seen how to build it and execute it using the Android emulator and used the Xamarin Android Designer to make edits to the screen. We’ll use these skills every time we create or edit an Android application going forward so I want to make sure everyone has a solid understanding of what we’ve done so far. Are there any questions?

### Slide 07: Further Reading



Before we end the session, I want to point out some additional resources available to you after the class is over. You'll of course have all of the code and resource files we used to build the labs today including completed versions of the labs, feel free to go back through a lab exercise if you don’t feel completely confident with the tools we’ve used. In addition, Xamarin has extensive documentation available online – in particular I’d suggest you read through the ListViews and Adapters tutorials shown here on the slide. The documentation expands on a lot of the concepts we’ve covered today and is periodically updated with new examples and API coverage so it’s worth checking it out.

### Slide 08: Thank You



Thank you for attending this lecture on ListViews and Adapters in Android – remember you can work through the labs again on your own to reinforce what we’ve covered today. Let’s open up the Q&A to answer any questions you have.