ListViews + Adapters in Android

Evolve Fundamentals Track, Chapter 6

Xamarin Inc.

Overview

This tutorial introduces the `ListView` class and the different types of *Adapters* you can use with it. The discussion will begin with an overview of the `ListView` class itself before introducing progressively more complex examples of how to use it.

Because of their power and flexibility, `ListView` controls are one of the most commonly used controls in an Android application. In fact, it would be difficult to find any application of complexity that didn't use them. As such, their usage is an enormous subject. This tutorial provides an introductory examination of some of the most common usages of the `ListView` control and its intent is to get you up to speed on using it quickly, but is by no means comprehensive. For a much more in depth look at the `ListView` control, see the ListViews and Adapters guide on Xamarin's web site.

A `ListView` consists of the following parts:

- **Rows** – The visible representation of the data in the list. Each row has its own `View`. The view can be either one of the built-in views defined in `Android.Resources`, or a custom view. Each row can use the same view layout or they can all be different.
- **Adapter** – A non-visual class that binds the data source to the list view. The `ListView` control requires an `Adapter` to supply the formatted `View` for each row. Android has built-in Adapters and Views that can be used, or custom classes can be created.

The structure of this tutorial is as follows:

- **Classes** – Overview of the classes used to display a `ListView`.
- **Displaying Data in a ListView** – How to display a simple list of data; how to implement `ListView's` usability features; how to use different built-in row layouts; and how Adapters save memory by re-using row views.

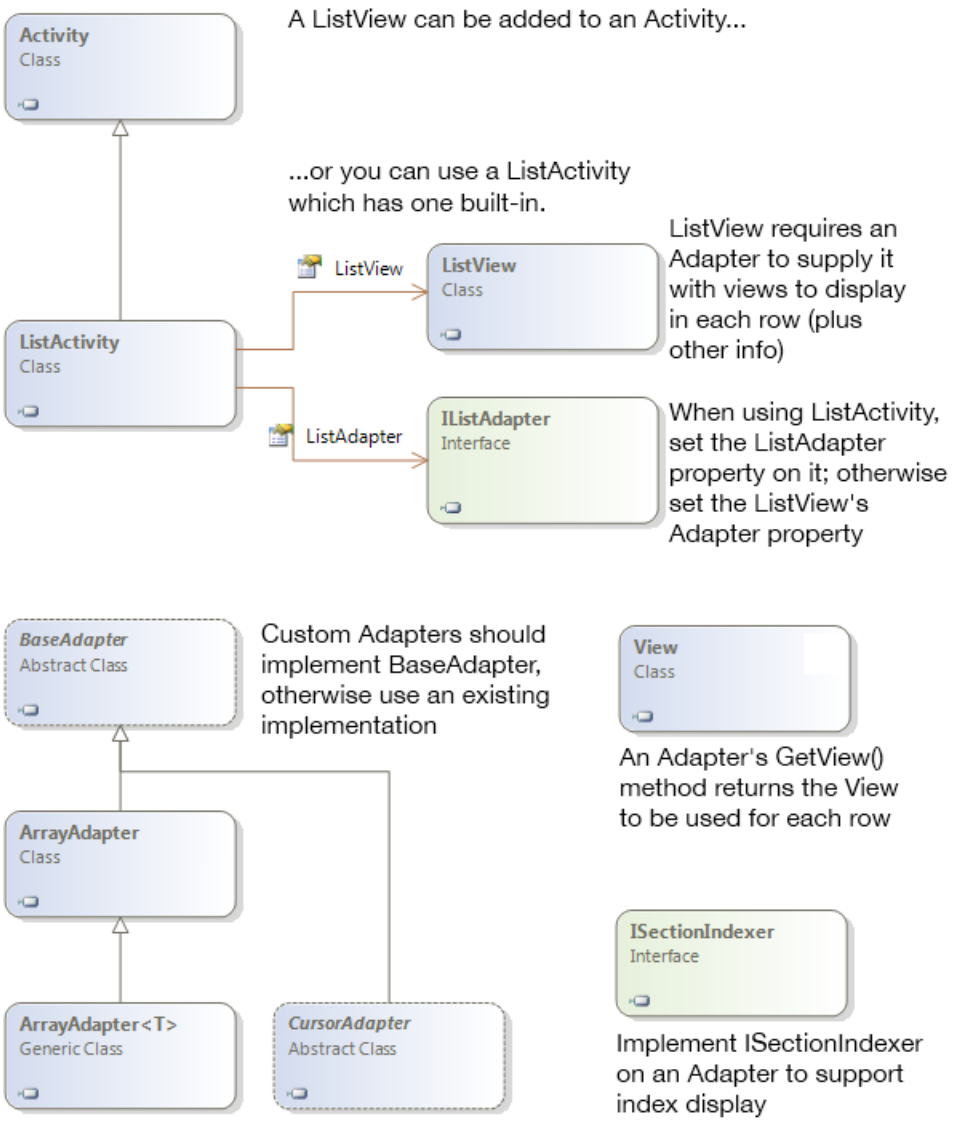
The code samples are structured to incrementally add functionality as you progress through the chapter. There are seven sample projects included in the solution:

- **ListsAndroid_demo1** – Implement a basic table
- **ListsAndroid_demo2** – Add an index
- **ListsAndroid_demo3** – Plain or Grouped style with a header
- **ListsAndroid_demo4** – Alternate built-in cell layouts
- **ListsAndroid_demo5** – Custom cell layouts
- **ListsAndroid_demo6** – Using tables for navigation

The code introduced in this chapter is already present in the samples, but commented out. Use the **Tasks** window in your IDE to quickly find the commented-out code that is marked with a `//TODO:` task identifier.

Classes Overview

The primary classes used to display `ListViews` are shown here:



The purpose of each class is described below:

- **ListView** – user interface element that displays a scrollable collection of rows. On phones it usually uses up the entire screen (in which case, the `ListActivity` class can be used) or it could be part of a larger layout on phones or tablet devices.
- **View** – a View in Android can be any user interface element, but in the context of a `ListView` it requires a `View` to be supplied for each row.
- **BaseAdapter** – Base class for Adapter implementations to bind a `ListView` to a data source.

- **ArrayAdapter** – Built-in Adapter class that binds an array of strings to a `ListView` for display. The generic `ArrayAdapter<T>` does the same for other types.
- **CursorAdapter** – Use `CursorAdapter` or `SimpleCursorAdapter` to display data based on an SQLite query.

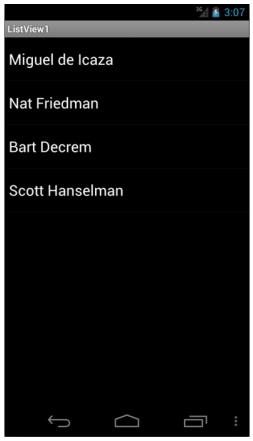
This tutorial covers how to create a custom implementation of BaseAdapter. For more information on ArrayAdapter or CursorAdapter, please consult Xamarin's documentation on ListViews and Adapters.

Populating a ListView with Data

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

The built-in adapters take a view resource ID as a parameter that gets used for the layout for each row. You can use built-in resources such as those in `Android.Resource.Layout` so you don't need to write your own.

Now that we understand the constituent components of a `ListView`, and the basic idea behind them, we'll use the **ListsAndroid_demo1** sample project to try it out. The following screenshot shows Speakers activity when we are done implementing this functionality:



Setup the Project

To get things setup, download the file `ListsAndroid.zip`, and un-zip it. The `ListsAndroid` solution contains all the code for this chapter. The demo projects already contain some files that we will use - here is a quick description of them:

- **Resources/values/String**.xml – the string resources have been expanded to include the various strings that we will need for this application.

- Models/Speaker.cs – the Speaker class is a business entity that holds data about the speaker.
- Models/Sessions.cs – this file holds the Sessions class, which contains some information about sessions that make up a conference.
- Assets/images/speakers – this folder holds headshots for each speaker in the conference. Each file in this directory will be embedded into the application.

If you run the ListsAndroid_demo1 project the screen will be empty, very much like the following screenshot:

With a basic application working, let's move on to implementing the ListView control.

Using an Array Adapter

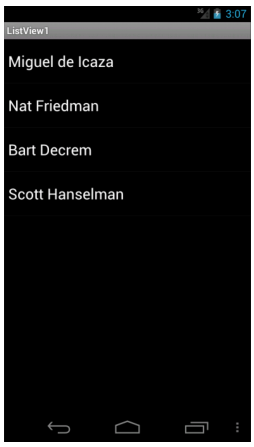
The simplest way to display data in a list is to use the built-in ArrayAdapter class to bind a collection of data to a ListView control. In this case ArrayAdapter uses three things to render a list:

- A context – the code passes in a reference to the current activity
- A layout resource ID – the sample uses the built-in `Android.Resource.Layout.SimpleListItem1` reference to a simple cell layout with one `TextView`. The `ArrayAdapter` will automatically know to place the data item in the Text property of this layout.
- An array of data – a simple string array.

The code in the `OnCreate` method looks like this

```
items = new string[] { "Miguel de Icaza", "Nat Friedman", "Bart Decrem",
"Scott Hanselman" };
ListAdapter = new ArrayAdapter<String>(this,
Android.Resource.Layout.SimpleListItem1, items);
```

Running the sample with this code shows the following output.



Responding to user interactions

To respond to touches on a specific item in the list, implement the `OnListItemClick` method. The example shows a toast popup message with the following code:

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

It's very simple to display a list in Xamarin.Android using just these lines of code, but there are also many options to display more complex data.

Creating a Custom Adapter

The data for the **ListsAndroid_demo2** project is a list of `Speaker` objects, rather than a simple string array. Our application must subclass `BaseAdapter<T>` to create a custom adapter so that we can select which property of the Speaker objects to display in the view. Override the following three members to display these objects:

- **Count** – To tell the control how many rows are in the data.
- **GetItemId** – Return a row identifier (typically the row number, although it can be any long value that you like).
- **GetView** – To return a `View` for each row, populated with data. This method has a parameter for the `ListView` to pass in an existing, unused row for re-use. We are using the built-in `Android.Resource.Layout.SimpleListItem1` view.

To demonstrate displaying a longer list of data, the method `PopulateSpeakerData` has been added to the SpeakersAdapter.cs file. It returns a hardcoded list of `Speaker` objects.

Re-using Cell Views

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 inflate a new View based on the `speaker_row` layout. We then obtain a reference for the various widgets in the view, and set the text (or image) for the widget. Once all the widgets are updated, we return the View that can now be displayed to the user.

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.

Building a Custom Adapter

Let's examine code in the **ListsAndroid_demo2** project. The `SpeakersAdapter` class contains the following elements:

1. The constructor populates the data and activity fields:

```
public class SpeakersAdapter: BaseAdapter<Speaker>
{
    List<Speaker> data;
    Activity context;
    public Speakers2Adapter(Activity activity, List<Speaker> speakers)
    {
        data = (from s in speakers orderby s.Name select s).ToList ();
        context = activity;
    }
```

2. The three overridden methods return information based on the data being displayed:

```
    public override long GetItemId(int position)
    {
        return position;
    }
    public override Speaker this[int position]
    {
        get { return data[position]; }
    }
    public override int Count
    {
        get { return data.Count; }
    }
```

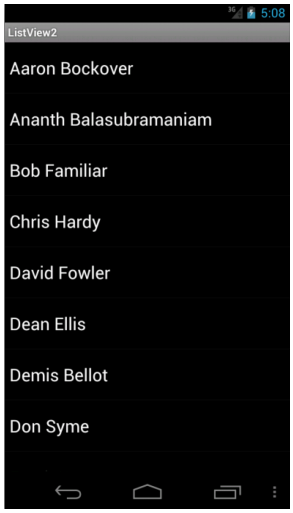
3. And finally, the GetView method implements the view re-use strategy (highlighted) and then assigns the `Speaker.Name` property as the value to be displayed in the view:

```
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;
}
```

After these changes we can use the custom adapter with this code in the `SpeakersActivity OnCreate` method:

```
speakers = GetSpeakerData ();
adapter = new Speakers2Adapter(this, speakers);
ListView.Adapter = adapter;
```

The app now looks like this:

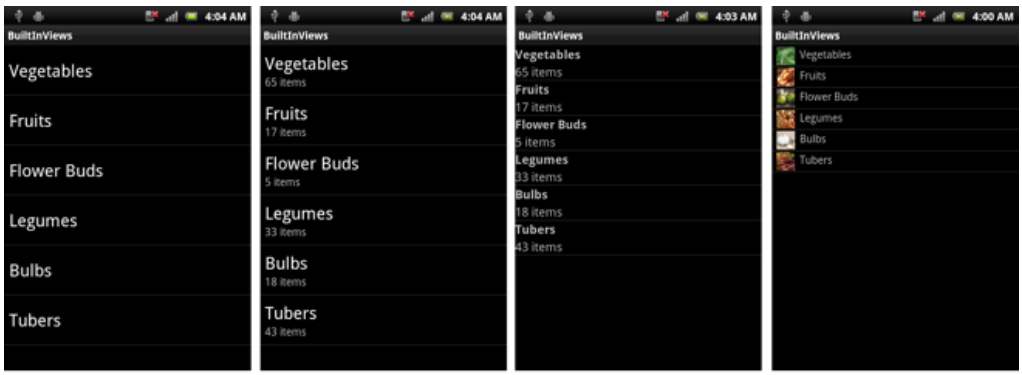


Changing the built-in Cell Style

In addition to the built-in `Android.Resource.Layout.SimpleListItem1` view used in previous examples, Android includes other view layouts you can use:

- SimpleListItem1
- SimpleListItem2
- TwoLineListItem
- ActivityListItem

An example of each type is shown below:



Uncomment different lines in the **ListsAndroid_demo3** `SpeakersAdapter` class to try these different cell styles:

```
public override View GetView(int position, View convertView, ViewGroup
parent)
{
    var view = convertView;
    if (view == null) {
       //TODO: Demo3: uncomment different layouts to see how they look
        //view =
_activity.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1,
null);
        view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem2,
null);
        //view =
_activity.LayoutInflater.Inflate(Android.Resource.Layout.TwoLineListItem,
null);
        //view =
_activity.LayoutInflater.Inflate(Android.Resource.Layout.ActivityListItem,
null);  // image
    }
    var speaker = data[position];
    //TODO: Demo3: change which UI controls are populated, depending on
which layout is used
    view.FindViewById<TextView>(Android.Resource.Id.Text1).Text =
speaker.Name;
    view.FindViewById<TextView>(Android.Resource.Id.Text2).Text =
speaker.Company;
    // ActivityListItem only
    //view.FindViewById<ImageView>
(Android.Resource.Id.Icon).SetImageDrawable (GetHeadShot
(speaker.HeadhuhshotUrl));
    return view;
}
```

This screenshot shows an example using our data with the `ActivityListItem` style:

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.

In the **ListsAndroid_demo4** project we want to display a headshot, company name, and the speaker's Twitter handle. The following screenshot shows an example of one such row:

Review the Android view file `speaker_row.axml` in the folder `Resources/layout`. Here is the XML for the layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:minWidth="25px"
    android:minHeight="25px"
    android:paddingBottom="4dp">
    <ImageView
        android:src="@android:drawable/ic_menu_gallery"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:id="@+id/headshotImageView"
        android:layout_centerVertical="true" />
    <TextView
        android:text="Small Text"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:id="@+id/twitterTextView"
        android:layout_below="@+id/speakerNameTextView"
        android:layout_alignParentRight="true"
        android:paddingRight="10dp" />
    <TextView
        android:text="Small Text"
        android:textAppearance="?android:attr/textAppearanceSmall"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/companyNameTextView"
        android:layout_toLeftOf="@+id/twitterTextView"
        android:layout_below="@+id/speakerNameTextView"
        android:layout_toRightOf="@+id/headshotImageView"
        android:paddingLeft="15dp" />
    <TextView
        android:text="Large Text"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/speakerNameTextView"
        android:layout_below="@+id/speakerNameTextView"
        android:layout_toRightOf="@+id/headshotImageView"
        android:paddingLeft="5dp" />
</RelativeLayout>
```

Now that we have a layout that we can use for a row, we need to implement the `GetView` method as shown in the following code snippet. The new custom layout is referred to by its resource ID (highlighted):

```
public override View GetView(int position, View convertView, ViewGroup
parent)
{
    var view = convertView;
    if (view == null) {
        view = context.LayoutInflater.Inflate(Resource.Layout.speaker_row,
null);
    }
    var speaker = data[position];
    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;
    return view;
}
```

Now if you run the application, and scroll through the list, you should be able to scroll through the **Speakers** tab and see a list of speakers with some information about them, as shown in the following screenshot:

Adding 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.

In the **ListAndroid_demo5** project, the `SpeakersActivity` class sets the `FastScrollEnabled` property of the ListView

```
ListView.FastScrollEnabled = true;
```

This allows the ListView scrolling speed to accelerate and move more quickly through the cells. The ListAndroid_demo5 project has had additional speaker data added to better demonstrate how the fast scrolling tabs helps, as shown in this screenshot:

The repeated data in this screenshot is on purpose – we needed to copy-paste lots of data to make the fast scrolling and index work.

Adding an Index

The **ListAndroid_demo5** sample also demonstrates how to add an index to a fast scrolling list view. The index helps the user identify where they are in long lists, especially when scrolling quickly.

To implement the index display your BaseAdapter subclass must implement the `ISectionIndexer` interface, which requires the following methods:

- **GetPositionForSection** – A lookup method that tells the ListView the row position where the specified section starts.
- **GetSectionForPosition** – A lookup method that tells the ListView which section a specified row belongs to.
- **GetSections** – The collection of strings to display for each section. This must be an array of `Java.Lang.Object` to satisfy the interface.

The sample implementation code in the `SpeakersAdapter` is in three parts. The first part is simply adding additional class-level fields to store the section information, to be used when scrolling through this data:

```
string[] sections;
Java.Lang.Object[] sectionsObjects;
Dictionary<string, int> alphaIndex;
```

The second part occurs in the constructor where the data is parsed to determine which sections to display. This is a very simple algorithm for demonstration purposes – depending on the data in your application you may write something very different.

```
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]);
}
```

The third part is the implementation of the interface methods. Because we have already pre-calculated the sections for this data, these methods are simple to implement:

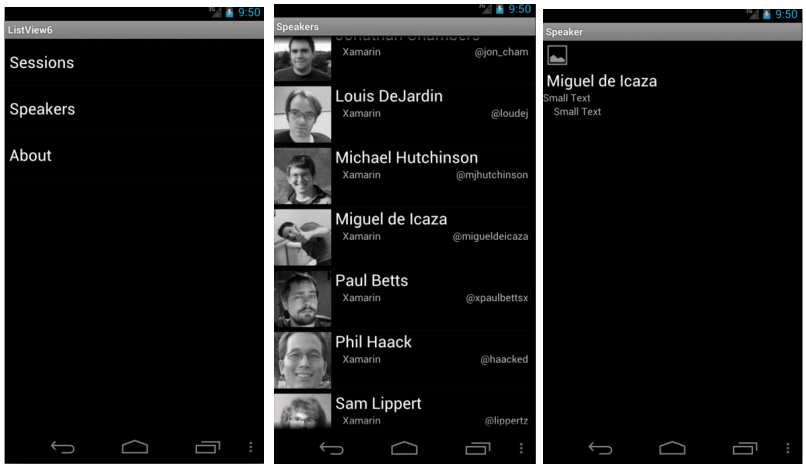
```
public int GetPositionForSection(int section)
{
    return alphaIndex[sections[section]];
}
public int GetSectionForPosition(int position)
{
    int prevSection = 0;
    for (int i = 0; i < sections.Length; i++) {
        if (GetPositionForSection(i) > position && prevSection <=
position) {
    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

The index is shown in this screenshot – the large letter “M” changes as the user scrolls through the list using the “tab”.

The repeated data in this screenshot is on purpose – we needed to copy-paste lots of data to make the fast scrolling and index work.

Using ListViews for Navigation

Many menu-driven apps use ListViews to help the user navigate through a set of hierarchical menu screens. The **ListsAndroid_demo6** sample project shows an example of how to construct a simple three level menu using the concepts covered in this chapter. 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. An example of the screenflow is shown below:



It is very easy to create this menu structure with ListViews.

Menu screen

Similar to the first example in this chapter, the menu screen is driven by a hardcoded array of strings.

```
items = new string[] { "Sessions", "Speakers", "About" };
ListAdapter = new ArrayAdapter<String>(this,
Android.Resource.Layout.SimpleListItem1, items);
```

The `OnListItemClick` method is also hardcoded to display a different activity depending on the item selected:

```
protected override void OnListItemClick(ListView l, View v, int position,
long id)
{
    var intent = new Intent(this, typeof(SessionsActivity));
    if (position == 1)
        intent = new Intent (this, typeof(SpeakersActivity));
    else if (position == 2)
        intent = new Intent (this, typeof(AboutActivity));
    StartActivity(intent);
}
```

List and detail screens

The subsequent screens in the hierarchy are taken directly from earlier examples in this chapter. The built-in “back” functionality allows the user to easily return to the previous screen.

Summary

The `ListView` class provides a flexible way to present data, whether it is a short menu or a long scrolling list. We covered a lot of ground in this tutorial. First, we examined how to use `ListActivity` with the built in Adapter `ArrayAdapter<T>`. Then, by subclassing `BaseAdapter` we achieved the flexibility to customize the appearance of the `ListView` to display a more complex data type.

We also learned about fast scrolling and indexes, and how to use ListViews to create a basic navigation hierarchy.