# Displaying Data in Lists

Xamarin Android Level 1, Chapter 4

Xamarin Inc.

# Overview

This chapter 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.

The structure of this chapter is as follows:

- **Visual Appearance** – Parts of the `ListView` control and how they work.
- **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.
- **Custom appearance** – Changing the style of the `ListView` with custom layouts, fonts and colors.

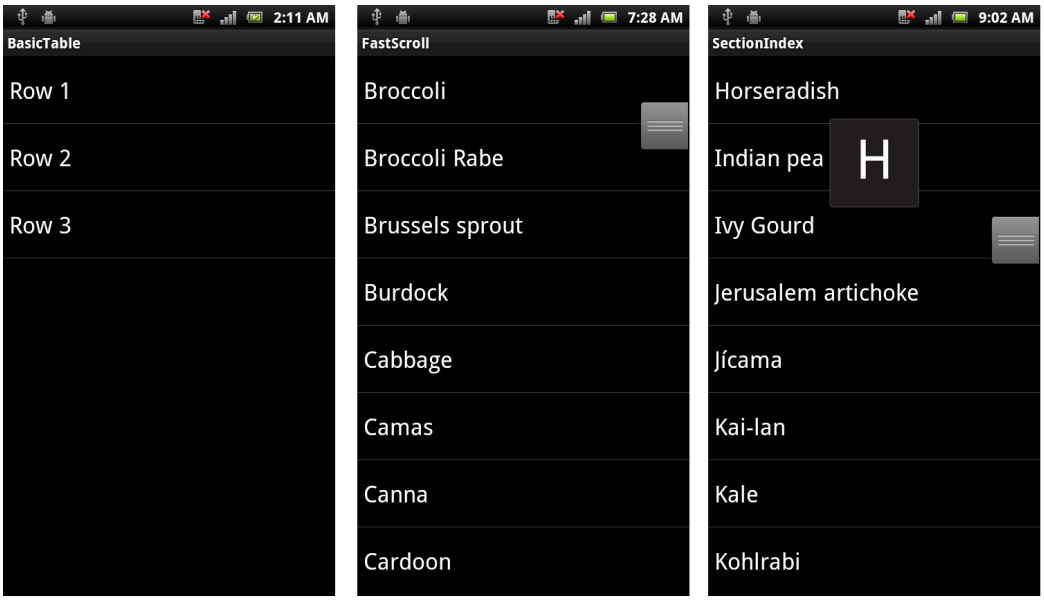
Because of their power and flexibility, `ListView` controls are one of the most commonly used controls in an Android application. In fact, it would be difficult to find any application of complexity that didn't use them. As such, their usage is an enormous subject. This chapter provides an introductory 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.

# ListView Parts & Functionality

A `ListView` consists of the following parts:

- **Rows** – The visible representation of the data in the list.
- **Adapter** – A non-visual class that binds the data source to the list view.
- **Fast Scrolling** – A handle that lets the user scroll the length of the list.
- **Section Index** – A user interface element that floats over the scrolling rows to indicate where in the list the current rows are located.

These screenshots use a basic `ListView` control to show how Fast Scrolling and Section Index are rendered:




Plain old rows | Fast Scrolling | Section Index

The elements that make up a `ListView` are described in more detail below:

## Rows

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. There are examples in this chapter of using built-in layouts and others explaining how to define custom layouts.

## Adapter

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.

## Fast Scrolling

When a `ListView` contains many rows of data fast-scrolling can be enabled to help the user navigate to any part of the list. The fast-scrolling 'scroll bar' can be optionally enabled (and customized in API level 11 and higher).

## Section Index

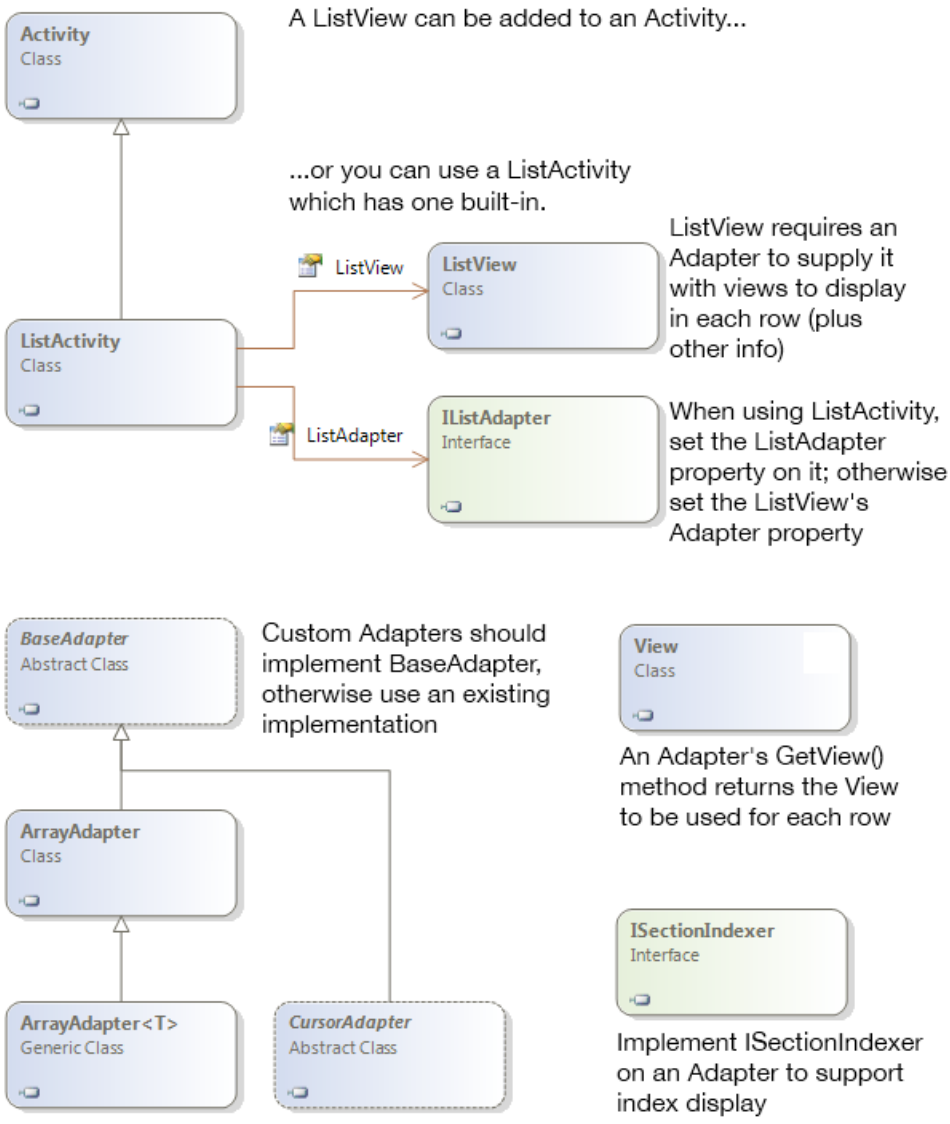
While scrolling through long lists, the optional section index provides the user with feedback on what part of the list they are currently viewing. It is only appropriate on long lists, typically in conjunction with fast scrolling.

## Content Provider

`ContentProviders` allow you to access data exposed by other applications (including Android system data like contacts, media and calendar information).

# 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 chapter contains simple examples that use an `ArrayAdapter` as well as more complex examples that require custom implementations of `BaseAdapter` or `CursorAdapter`.

# 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` and `ArrayAdapter` classes 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 each row. You can use built-in resources such as those in `Android.Resource.Layout` so you don't need to write your own.

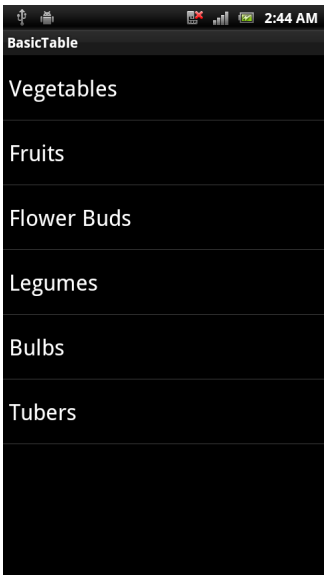
## Using ListActivity to Display A List and Respond to Selection

Now that we understand the constituent components of a `ListView`, and the basic idea behind them, let's actually create an application that uses them. In this walkthrough, we'll subclass `ListActivity` (which hosts a built-in `ListView`) to create a simple list and respond to user selections.

1. Start by creating a new Xamarin.Android project called `BasicTable`.
2. Next, rename the class `Activity1` to `HomeScreen`, and then edit the file so that it uses an `ArrayAdapter<string>` to display a list of strings in the `ListView`:

```
[Activity(Label = "BasicTable", MainLauncher = true, Icon =
"@drawable/icon")]
public class HomeScreen : ListActivity {
    string[] items;
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        items = new string[] { "Vegetables","Fruits","Flower
Buds","Legumes","Bulbs","Tubers" };
        ListAdapter = new ArrayAdapter<String>(this,
Android.Resource.Layout.SimpleListItem1, items);
    }
```

3. Now, run the application, it should look something like the following:



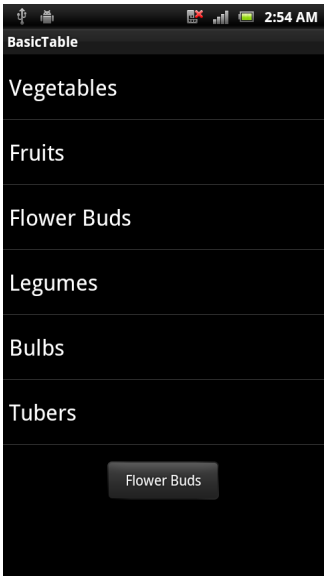

At this point the application will display a list, but will not respond to any user selections. Often times, we want a `ListView` to not only display rows of data but also allow the user to interact with in some fashion (such as playing a song, or calling a contact, or showing another screen). Let's upgrade this to allow a user to select a row and then respond to that selection.

In order to respond to user selection, we need to override the virtual method on `ListActivity` called `OnListItemClicked`, which is invoked when a user touches a row in the list.

4. To do this, let's modify the class `HomeScreen`, and override the method `OnListItemClick` to display a toast message when it's selected:

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

5. Now, let's run the application, and select an item in the list. A Toast should appear, similar to the following screen shot:




As we can see, it's very easy to implement a basic `ListView` and respond to selection, but let's explore `ListViews` and `Adapters` a bit further.

## Creating a Custom Adapter

`ArrayAdapter<T>` is great because of its simplicity, but it's extremely limited. Often an application will have a collection of business entities, rather than just strings that you want to bind. For example, if your data consists of a collection of `Employee` classes then you might want the list to display the names of each employee. To customize the display of data in a `ListView` the application must use a subclass of `BaseAdapter` and override the following three members:

- **Count** – To tell the control how many rows are in the data.
- **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.
- **GetItemId** – Return a row identifier (typically the row number, although it can be any long value that you like).

Let's extend our application to use a custom adapter and a user-defined class as its data.

1. To start, add a new class to the application, and name it `HomeScreenAdapter`. Edit the class so that it contains the following:

```
public class HomeScreenAdapter : BaseAdapter<Flora> {
    Flora[] items;
    Activity context;
    public HomeScreenAdapter(Activity context, Flora[] items) : base() {
        this.context = context;
        this.items = items;
    }
    public override long GetItemId(int position)
    {
```

```
            return position;
        }
        public override Flora this[int position] {
            get { return items[position]; }
        }
        public override int Count {
            get { return items.Length; }
        }
        public override View GetView(int position, View convertView, ViewGroup
    parent)
        {
            View view = convertView; // re-use an existing view, if one is
    available
            if (view == null) // otherwise create a new one
                view =
    context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1,
    null);
            view.FindViewById<TextView>(Android.Resource.Id.Text1).Text =
    items[position].Heading;
            return view;
        }
    }
```

`HomeScreenAdapter` is a fully functional adapter for which encapsulates our list and overrides `GetView` to return a `View` for displaying each row.

2. Next, create a custom class called Flora to the project. This is a class that will hold data about the vegetables:

```
/// <summary>
///   A simple class for holding the data that is read from VegeData.txt
/// </summary>
public class Flora
{
    public string Name { get; set; }
    public override string ToString()
    {
        return Name;
    }
}
```

## REUSING VIEWS IN A LISTVIEW

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 the adapter should create a new `View` for the row.

The following code snippet (from our sample above) illustrates the pattern to reuse those views:

```
public override View GetView(int position, View convertView, ViewGroup
parent)
{
    View view = convertView; // re-use an existing view, if one is
supplied
    if (view == null) // otherwise create a new one
        view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1,
null);
    // set view properties to reflect data for the given row
    view.FindViewById<TextView>(Android.Resource.Id.Text1).Text =
items[position].Heading;
    // return the view, populated with data, for display
    return view;
}
```

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.

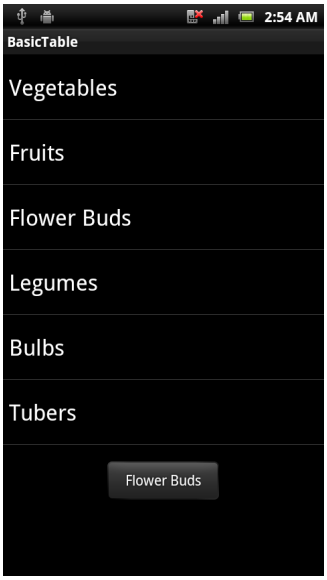
3. The final step is to modify the Activity to use `HomeScreenAdapter`, and not the `ArrayAdapter`. Edit the class `HomeScreen` to so that the `OnCreate` method is as follows:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    items = new Flora[] {
        new Flora { Name = "Vegetables"},
        new Flora { Name = "Fruits"},
        new Flora { Name = "Flower Buds"},
        new Flora { Name = "Legumes"},
        new Flora { Name = "Bulbs"},
        new Flora { Name = "Tubers"}
    };

    ListAdapter = new HomeScreenAdapter(this, items);
}
```

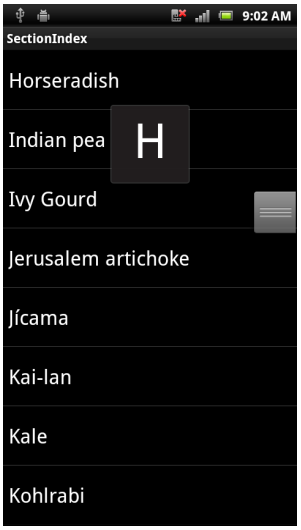
Because this example uses the same row layout (`SimpleListItem1`), when the application is run, it should look identical to the previous example, but now displays data from custom objects:




Custom adapters like this are much more common in real-world applications because data is often encapsulated in custom objects. For a working example of this sample, see the `04 - Displaying Data in Lists - Walkthrough 1 - Source Code.zip` file.

# Implementing Fast-Scrolling and Adding a Section Index

For lists that can become long, Android has two features that allow for quick navigation through the list; Fast-Scrolling, and Section Indexing. Fast Scrolling allows the user to quickly scroll through the list by providing a scroll widget on the right side of the list. And a Section Index provides an overlay that displays information (such as a letter) to indicate what section of the list the user is currently in. The following screenshot illustrates these two features in action:




## ENABLING FAST SCROLLING

Enabling fast scrolling is as easy as simple as setting the `FastScrollEnabled` property on a `ListView` to `true`:

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

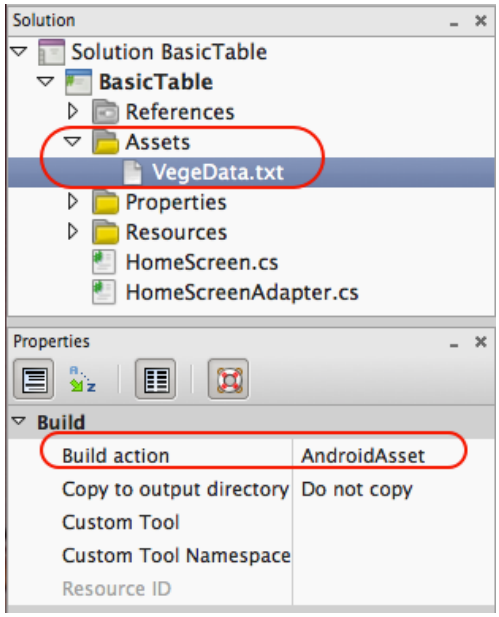
## ENABLING SECTION INDEXES

To enable the section index, we have to implement the `ISectionIndexer` interface on our custom adapter, which has the following three methods:

- **GetSections** – Provides the complete list of section index titles that could be displayed. This method requires an array of Java Objects so the code needs to create a `Java.Lang.Object[]` from a .NET collection. In our example it returns a list of the initial characters in the list as `Java.Lang.String`.
- **GetPositionForSection** – Returns the first row position for a given section index.
- **GetSectionForPosition** – Returns the section index to be displayed for a given row.

Let's create a new application that utilizes fast-scrolling and section indexes. Because these features are more applicable to large lists, we're going to create a longer list of vegetables. The `HomeScreen` Activity will be changed to load a large list of vegetables from a text file that has been embedded as an *AndroidAsset* into the application.

1. Start by opening the solution from the previous walkthrough in this chapter.
2. Next, download the text file `04 - Displaying Data in Lists - Walkthrough 1 - VegeData.txt` and rename it `VegeData.txt`.
3. It may be necessary to create a new folder in the application called `Assets`. If so, create that folder. Copy the file `VegeData.txt` into the folder, and ensure that the **Build Action** of the file is set to **AndroidAsset**, as shown in the following screenshot:



4. Now, create a new method in the `HomeScreen` activity that will read the text file and create a `Flora` object for each line in `VegeData.txt`, and place those objects into a sorted `Flora[]`:

```
/// <summary>
///   Loads the list of vegetables from the text file VegeData.txt which
is an Android Asset.
/// </summary>
/// <returns> A sorted list of vegetables. </returns>
private Flora[] LoadListOfVegetablesFromAssets()
{
    var veges = new List<Flora>();
    var seedDataStream = Assets.Open(@"VegeData.txt");
    using (var reader = new StreamReader(seedDataStream))
    {
        while (!reader.EndOfStream)
        {
            var flora = new Flora { Name = reader.ReadLine() };
            veges.Add(flora);
        }
    }
    veges.Sort((x, y) => String.Compare(x.Name, y.Name,
StringComparison.Ordinal));
    items = veges.ToArray();

    return items;
}
```

5. Next modify the `OnCreate` method to get a list of vegetables `LoadListOfVegetablesFromAssets` and pass that list to our custom adapter. The following code snippet shows the new `OnCreate` method:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    items = LoadListOfVegetablesFromAssets();
    ListAdapter = new HomeScreenAdapter(this, items);
    ListView.FastScrollEnabled = true;
}
```

Notice in the previous code snippet that we have enabled fast scrolling on the `ListView`.

6. Modify `HomeScreenAdapter` so that it implements `ISectionIndexer`. The code snippet below shows the changes that need to be made to `HomeScreenAdapter`:

```
public class HomeScreenAdapter : BaseAdapter<string>, ISectionIndexer
{
    // Code omitted for clarity

    // -- Code for the ISectionIndexer implementation follows --
    string[] sections;
    Java.Lang.Object[] sectionsObjects;
    Dictionary<string, int> alphaIndex;
```

```
        public int GetPositionForSection(int section)
        {
            return alphaIndex [sections [section]];
        }

        public int GetSectionForPosition(int position)
        {
            return 1;
        }

        public Java.Lang.Object[] GetSections()
        {
            return sectionsObjects;
        }
    }
```

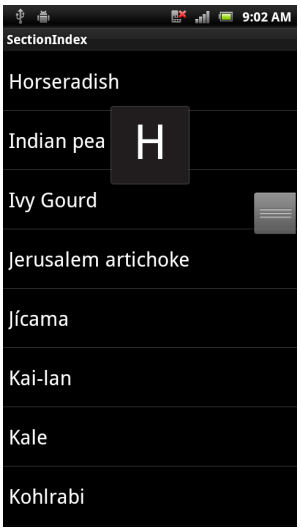
In the above code, there are two instance variables: `alphaIndex` and `sectionsObjects`. `alphaIndex` is an internal list that will map a row in the list with a given letter of the alphabet. The `sectionsObjects` array will be used by the `ListView` to display a preview letter while scrolling.

7. The final change that must be made on the adapter is to initialize the `sectionObjects` and `alphaIndex` from the list of vegetables. The adapter will do this by looping through every row in the vegetable list and extracting the first character of the title (the items must already be sorted for this to work). Edit the class `HomeScreenAdapter` and create a method called `BuildSectionIndex` to perform this task:

```
  private void BuildSectionIndex()
  {
      alphaIndex = new Dictionary<string, int>();
      for (var i = 0; i < items.Length; i++)
      {
          // Use the first character in the name as a key.
          var key = items[i].Name.Substring(0,1);
          if (!alphaIndex.ContainsKey(key))
          {
              alphaIndex.Add(key, i);
          }
      }

      sections = new string[alphaIndex.Keys.Count];
      alphaIndex.Keys.CopyTo(sections, 0);
      sectionsObjects = new Object[sections.Length];
      for (var i = 0; i < sections.Length; i++)
      {
          sectionsObjects[i] = new String(sections[i]);
      }
  }
```

8. Finally, let's run the application and scroll through the list. It should appear similar to the following screenshot:




Note that section index titles don't need to map 1:1 to actual sections. This is why the `GetPositionForSection` method exists. `GetPositionForSection` gives you an opportunity to map whatever indices are in your index list to whatever sections are in your list view. For example, you may have a "z" in your index, but you may not have a table section for every letter, so instead of "z" mapping to 26, it may map to 25 or 24, or whatever section index "z" should map to.

# Customizing the ListView's Appearance

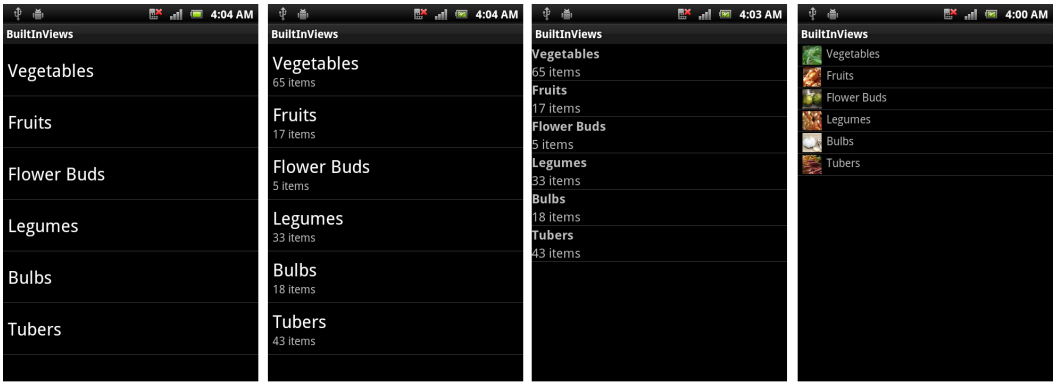
The ListView control is highly customizable. It offers several different built-in styles styles, as well the functionality to customize each row with a custom view.

## Built-in Row Views

There are four built-in Views that can be referenced using `Android.Resource.Layout`:

- **SimpleListItem1** – Single line of text.
- **SimpleListItem2** – Two lines of text.
- **TwoLineListItem** – Two lines of text.
- **ActivityListItem** – Single line of text with an image.

Each built-in row view has a built in style associated with it, as illustrated in the following screenshots:




SimpleListItem1 SimpleListItem2 TwoLineListItem ActivityListItem

These built in styles can be found in the `Android.Resource.Layout` class and instantiated like any other view. For instance, the following code snippet would go in the `GetView` method of an adapter and would create the first layout style:

```
view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1,
null);
```

The view's properties can then be set by referencing the standard control identifiers `Text1`, `Text2` and `Icon` under `Android.Resource.Id` (do not set properties that the view does not contain or an exception will be thrown):

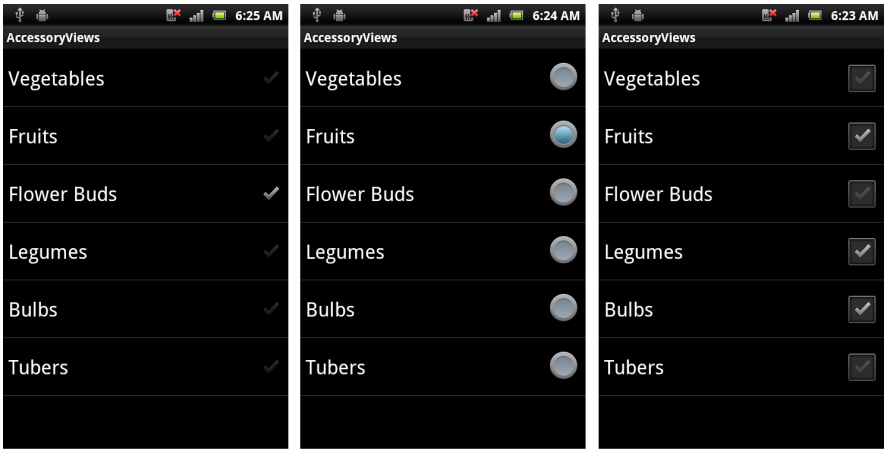
```
view.FindViewById<TextView>(Android.Resource.Id.Text1).Text =
item.Heading;
view.FindViewById<TextView>(Android.Resource.Id.Text2).Text =
item.SubHeading;
view.FindViewById<ImageView>(Android.Resource.Id.Icon).SetImageResource(it
em.ImageResourceId); // only use with ActivityListItem
```

## Accessories

Rows can also have these accessories added to the right of the view to indicate selection state

- ➔ **SimpleListItemChecked** – To create a single-selection list with a tick as the indicator.
- ➔ **SimpleListItemSingleChoice** – To create radio-button-type lists.
- ➔ **SimpleListItemMulitpleChoice** – To create checkbox-type lists.

The aforementioned accessories are illustrated in the following screens, in their respective order:




Checked SingleChoice MultipleChoice

To display one of these accessories, pass the required layout resource ID to the adapter then manually set the selection state for the required rows. For example, this code snippet shows how to enable a checked accessory:

```
ListAdapter = new ArrayAdapter<String>(this,
Android.Resource.Layout.SimpleListItemChecked, items);
```

The `ListView` itself supports different selection modes, regardless of the accessory being displayed. To avoid confusion, use `Single` selection mode with `Checked` and `SingleChoice` accessories and the `Multiple` mode with the `MultipleChoice` style. The selection mode is controlled by the `ChoiceMode` property of the `ListView`.

## SELECTING ITEMS PROGRAMMATICALLY

Manually setting which items are 'selected' is done with the `SetItemChecked` method (it can be called multiple times for multiple selection):

```
// Set the initially checked row ("Fruits")
lv.SetItemChecked(1, true);
```

The code also needs to detect single selections differently from multiple selections. To determine which row has been selected in `Single` mode use the `CheckedItemPosition` integer property:

```
FindViewById<ListView>(Android.Resource.Id.List).CheckedItemPosition
```

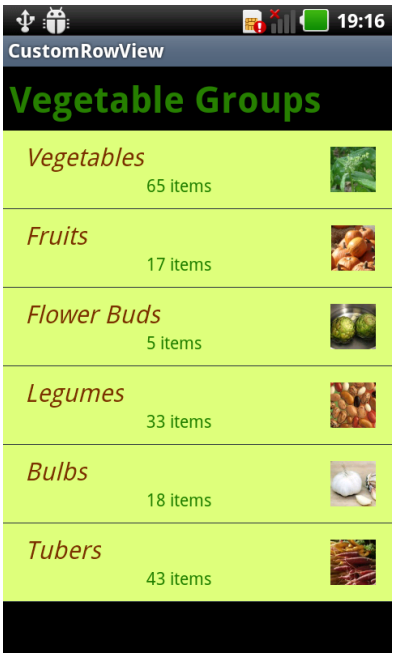
To determine which rows have been selected in `Multiple` mode you need to loop through the `CheckedItemPositions` property which returns a `SparseBooleanArray`. A sparse array is like a dictionary that only contains entries where the value has been changed, so you must traverse the entire array looking for `true` values to know what has been selected in the list as illustrated in the following code snippet:

```
var sparseArray =
FindViewById<ListView>(Android.Resource.Id.List).CheckedItemPositions;
for (var i = 0; i < sparseArray.Size(); i++ )
    Console.Write(sparseArray.KeyAt(i) + "=" + sparseArray.ValueAt(i) +
",");
Console.WriteLine();
```

# Creating Custom Row Layouts

While the four built-in row styles cover a lot of common use cases, it's often necessary to create custom rows to display data the way you want. For these scenarios, Android allows you to create custom views and use them as rows in a ListView. Custom views are generally declared as AXML files in the `Resources/Layout` directory and then loaded by a custom adapter. The view can contain any number of display classes (such as *TextViews*, *ImageViews* and other controls) with custom colors, fonts and layout.

In this walkthrough, we'll create an application that will display a list similar to the one displayed in the following screenshot using custom views as rows:



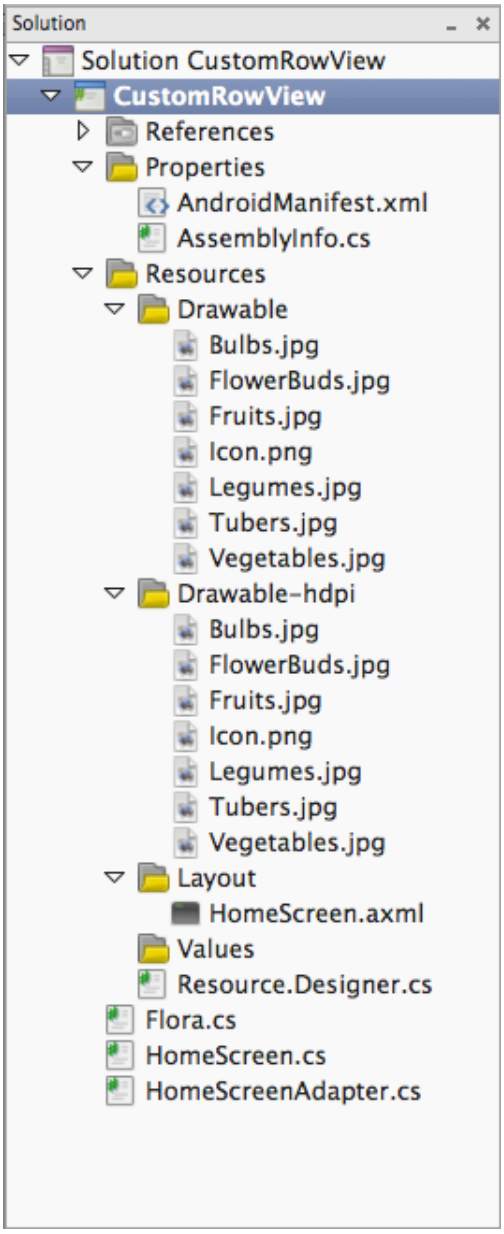

This walkthrough differs from the previous walkthrough in a number of ways:

- ➔ It inherits from `Activity`, not `ListActivity`. You can customize rows for any `ListView`, however other controls can also be included in an `Activity` layout (such as a heading, buttons or other user interface elements). This example adds a heading above the `ListView` to illustrate.
- ➔ It requires an AXML layout file for the screen; in the previous examples the `ListActivity` does not require a layout file. This AXML contains a `ListView` control declaration.
- ➔ It requires an AXML layout file to render each row. This AXML file contains the text and image controls with custom font and color settings. This custom layout will be inflated by the adapter as necessary.
- ➔ It uses an optional custom selector XML file to set the appearance of the row when it is selected.

➔ `ItemClick` must be declared differently (an event handler is attached to `ListView.ItemClick` rather than an overriding `OnListItemClick` in `ListActivity`).

These changes are detailed below, starting with creating the activity's view and the custom row view and then covering the modifications to the Adapter and Activity to render them.

To get started with this walkthrough, download the zip file `04 - Display Data in Lists - CustomRowView 1.zip` and extract the contents. This zip file is a starter project for this walkthrough with the following contents:




➔ **Drawable** – this folder contains the default resource images to be displayed in a row.

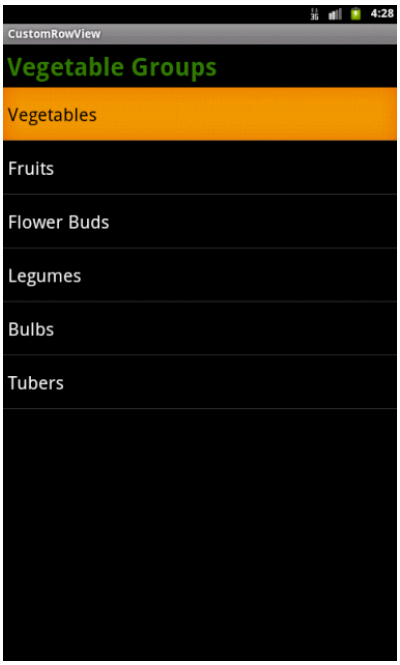
- ➔ **Drawable-hdpi** – this folder contains the resource images to be displayed in a row for HDPI devices.
- ➔ **Layout/HomeScreen.axml** – this is a layout file for the activity that contains a `ListView` and a `TextView`.
- ➔ **HomeScreen.cs** – this file contains the class `HomeScreen`, which is just a regular `Activity` (not a `ListActivity`).
- ➔ **HomeScreenAdapter.cs** – this file contains the Adapter that will bind the data
- ➔ **Flora.cs** – this file contains a class called `Flora` that will hold the data to be displayed in a row.

Because `HomeScreen` no longer inherits from `ListActivity` it doesn't have a default view, so the layout AXML file `HomeScreen.axml` will be used. The layout has a heading (using a `TextView`) and a `ListView` to display data. The layout is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <TextView android:id="@+id/Heading"
    android:text="Vegetable Groups"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#00000000"
    android:textSize="30dp"
    android:textColor="#FF267F00"
    android:textStyle="bold"
    android:padding="5dp"
  />
  <ListView android:id="@+id/List"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:cacheColorHint="#FFDAFF7F"
    />
</LinearLayout>
```

The benefit of using an `Activity` with a custom layout (instead of a `ListActivity`) lies in being able to add additional controls to the screen, such as the heading `TextView` in this example.

At this point, if the application is run, it should be very similar in appearance to the following screenshot:




Let's start in on creating custom row views.

1. First, create a new layout that will be used for a list row called `/Resources/Layout/CustomView.axml` with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="8dp">
  <LinearLayout android:id="@+id/Text"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:paddingLeft="10dip">
    <TextView
      android:id="@+id/Text1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textColor="#FF7F3300"
      android:textSize="20dip"
      android:textStyle="italic"
      />
    <TextView
      android:id="@+id/Text2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="14dip"
      android:textColor="#FF267F00"
      android:paddingLeft="100dip"
      />
  </LinearLayout>
```

```
  <ImageView
    android:id="@+id/Image"
    android:layout_width="48dp"
    android:layout_height="48dp"
    android:padding="5dp"
    android:src="@drawable/icon"
    android:layout_alignParentRight="true"
    />
</RelativeLayout >
```

This layout defines two `TextViews` and one `ImageView` in each row.

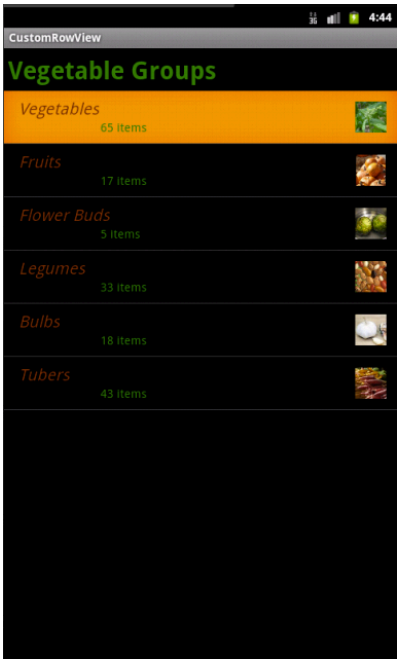
2. Next, we need to change the `GetView` method in the `HomeScreenAdapter` so that it uses our custom view and then sets the various view properties on that view to show the data for that particular `Flora`:

```
public override View GetView(int position, View convertView, ViewGroup
parent)
{
    var item = items[position];
    View view = convertView;
    if (view == null)
    { // no view to re-use, create new
        view = context.LayoutInflater.Inflate(Resource.Layout.CustomView,
null);
    }
    view.FindViewById<TextView>(Resource.Id.Text1).Text = item.Heading;
    view.FindViewById<TextView>(Resource.Id.Text2).Text = item.SubHeading;

view.FindViewById<ImageView>(Resource.Id.Image).SetImageResource(item.Imag
eResourceId);
        return view;
    }
```

This code inflates our custom view, rather than a built in one, and then assigns the data to the appropriate sub-views. For example, the image is displayed in the `ImageView` with the ID `Resource.Id.ImageView`, the heading in `Resource.Id.Text1`, and the subheading in `Resource.Id.Text2`. The View making up the row is then returned from the method.

3. Now, if we run the application, we should get something similar to the following:




The application is now using the custom layout for each row, but let's customize it a bit more to make it more colorful.

4. First, create a new color resource file `/Resources/Values/Colors.xml` with these contents:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="cellback">#FFDAFF7F</color>
</resources>
```

This file defines a color resource named `cellback` that can be referenced by other layout files in the application.

5. Next create a new drawable resource that will be used to set the color for each row. Add another new file named `/Resources/Drawable/CustomSelector.xml` with a new state list drawable of the following:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
<item android:state_pressed="false"
   android:state_selected="false"
   android:drawable="@color/cellback" />
<item android:state_pressed="true" >
   <shape>
      <gradient
         android:startColor="#E77A26"
         android:endColor="#E77A26"
         android:angle="270" />
   </shape>
</item>
<item android:state_selected="true"
   android:state_pressed="false"
```

```
    android:drawable="@color/cellback" />
</selector>
```

This state-list selector controls what color is displayed for the selected row and what color is displayed for an unselected row. Of particular interest is how the color `cellback` is referenced:

```
android:drawable="@color/cellback"
```

`@color` tells Android to locate a color resource with the name that follows the `/`.

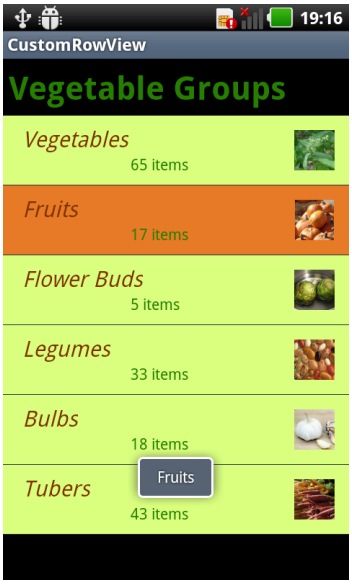
For more information about state list selectors, consult Android's documentation.

6. Next, let's edit the `CustomView.axml` file and set the background attribute of the `RelativeLayout` element to use the custom selector state-list that was created in the previous step:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="@drawable/CustomSelector"
    android:padding="8dp"
```

Notice that the state-list drawable resource that was created in the previous step is assigned to the `background` attribute of the `RelativeLayout`. The `@drawable` syntax tells Android to use a drawable resource named `CustomSelector`.

7. Run the application and select a row. The application should appear similar to the following screenshot:




Congratulations! You've now successfully created a `ListView` that uses custom views for its rows.

## PREVENTING FLICKER ON CUSTOM LAYOUTS

Android attempts to improve the performance of `ListView` scrolling by caching layout information. If you have long scrolling lists of data you should also set the the `android:cacheColorHint` property on the `ListView` declaration in the Activity's AXML definition (to the same color value as your custom row layout's background).

Failure to include this hint could result in a 'flicker' as the user scrolls through a list with custom row background colors.

> **Note:** While a custom row layout can contain many different controls, scrolling performance can be affected by complex designs and using images (especially if they have to be loaded over the network). See Google's Making ListView Scrolling Smooth document for more information on addressing scrolling performance issues.

# Using CursorAdapters

Android provides adapter classes specifically to display data from an SQLite database:

- **SimpleCursorAdapter** – Similar to an `ArrayAdapter` because it can be used without subclassing. Simply provide the required parameters (such as a cursor and layout information) in the constructor and then assign to a `ListView`.
- **CursorAdapter** – A base class that you can inherit from when you need more control over the binding of data values to layout controls (for example, hiding/showing controls or changing their properties).

Cursor adapters provide a high-performance way to scroll through long lists of data that are stored in SQLite. The consuming code must define an SQL query in a `Cursor` object and then describe how to create and populate the views for each row.

For more details on using a `CursorAdapter`, consult Xamarin's documentation on ListViews and Adapters.

# ListView and the Activity Lifecycle

All of the examples in this chapter to this point perform 'setup tasks' in the Activity's `OnCreate` method. The examples generally use small data sets that do not change, so re-loading the data more frequently is unnecessary.

The next chapter, *Responding to the Activity Lifecycle*, will discuss the lifecycle of an Android application, the various states an application goes through, and how an application should respond to these state changes.

# Summary

The `ListView` class provides a flexible way to present data, whether it is a short menu or a long scrolling list. It provides usability features like fast scrolling, indexes and single or multiple selection to help you build mobile-friendly user interfaces for your applications.

We covered a lot of ground in this chapter. 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` and use a more complex data type. The custom layout also made use of a state-list drawable resource to display which row was selected. Finally, we modified our ListView to implement `ISectionIndex` so that the application could display section indexes during scrolling.