Part 1 - ListView Parts and Functionality

A `ListView` consists of the following parts:

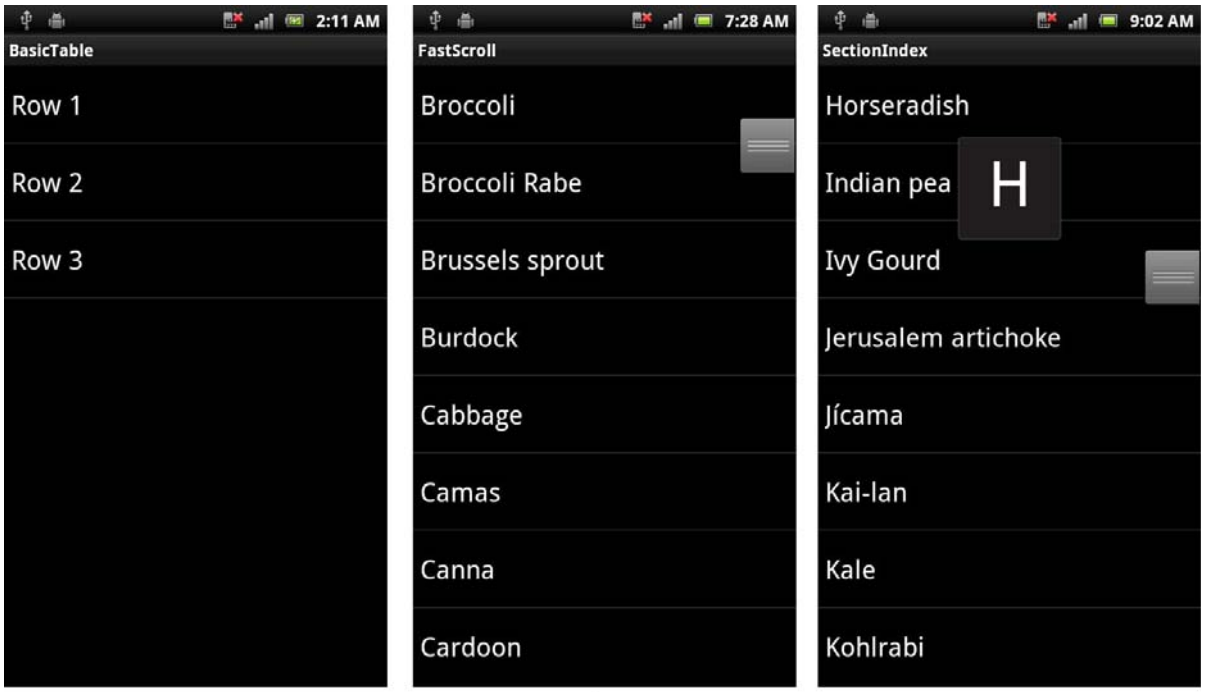
Rows – The visible representation of the data in the list.

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 document 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

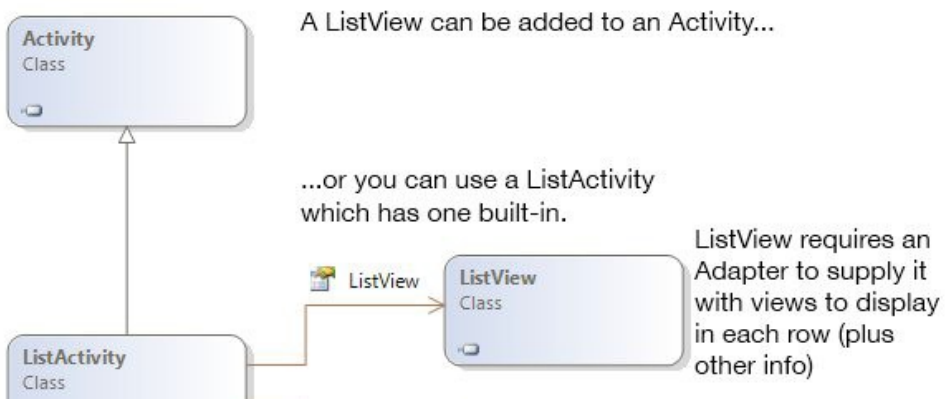
When a `ListView` contains many rows of data fast-scrolling can be enabled to help the user navigate to any part of the list. The fast-scrolling 'scroll bar' can be optionally enabled (and customized in API level 11 and higher).

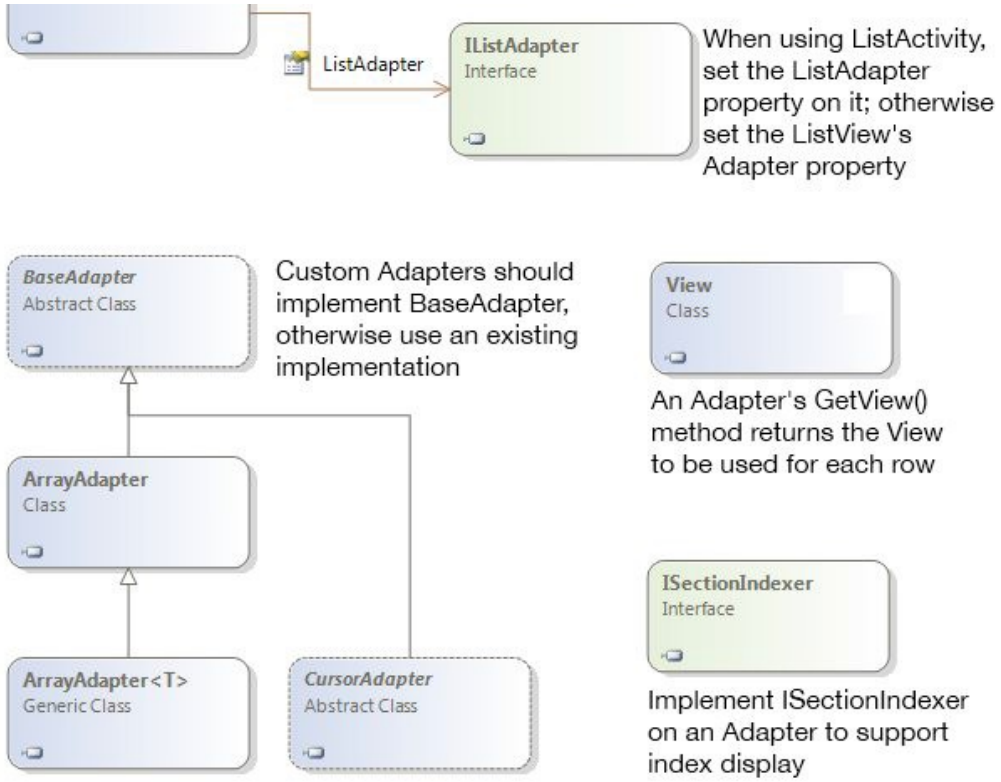
Section Index

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.

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

Part 2 - Populating a ListView With Data

To add rows to a `ListView` you need to add it to your layout and implement an `IListAdapter` with methods that the `ListView` calls to populate itself. Android includes 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.

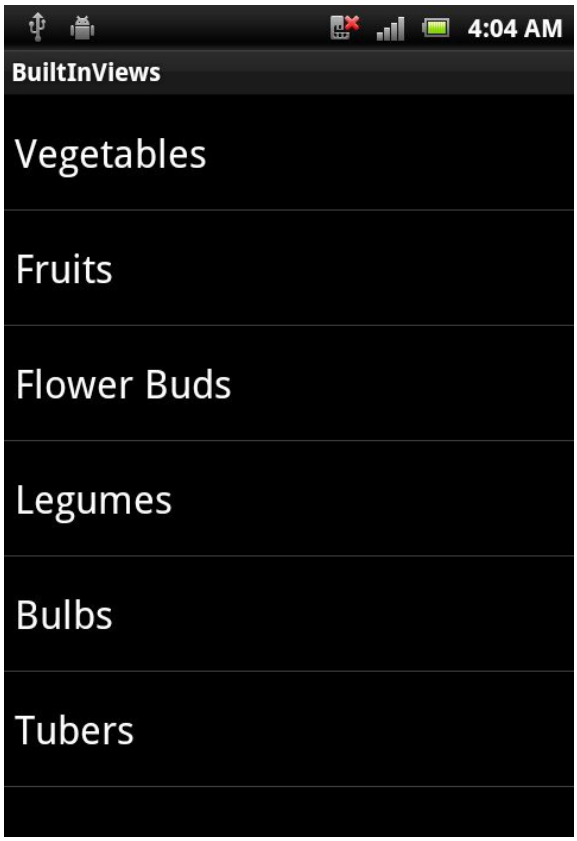
Using ListActivity and ArrayAdapter<String>

The example `BasicTable/HomeScreen.cs` demonstrates how to use these classes to display a `ListView` in only a few lines of code:

```
[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);

   }

   protected override void OnListItemClick(ListView l, View v, int
position, long id)
}
```

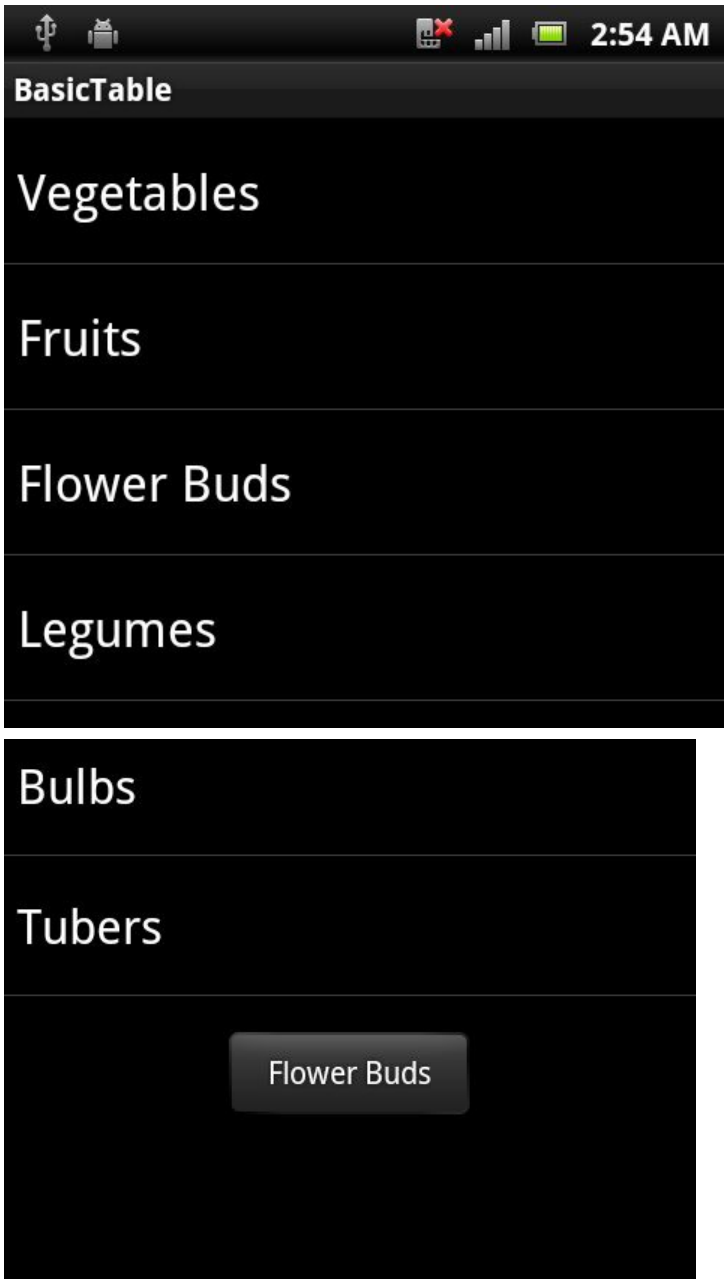
Handling Row Clicks

Usually a `ListView` will also allow the user to touch a row to perform some action (such as playing a song, or calling a contact, or showing another screen). To respond to user touches there needs to be one more method implemented in the `ListActivity` – `OnListItemClick`
– like this:




```
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();

}
```

Now the user can touch a row and a `Toast` alert will appear:



Implementing a ListAdapter

`ArrayAdapter<string>` is great because of its simplicity, but it's extremely limited. However, often times you 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 to list to just display the names of each employee. To customize the behavior of a `ListView` to control what data is displayed you must implement a subclass of `BaseAdapter` overriding the following four items:

- 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).
- this[int] indexer – To return the data associated with a particular row number.

The example code in `BasicTableAdapter/HomeScreenAdapter.cs` demonstrates how to subclass `BaseAdapter`:

```
public class HomeScreenAdapter : BaseAdapter<string> {
   string[] items;
   Activity context;

   public HomeScreenAdapter(Activity context, string[] items) : base()

{

       this.context = context;
       this.items = items;
   }

   public override long GetItemId(int position)

  {

       return position;

   }

   public override string 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];
       return view;

   }

}
```

Using a Custom Adapter

Using the custom adapter is similar to the built-in `ArrayAdapter`, passing in a `context` and the `string[]` of values to display:

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

Because this example uses the same row layout (`SimpleListItem1`) the resulting application will look identical to the previous example.

Row View Re-Use

In this example there are only six items. Since the screen can fit eight, no row re-use required. When displaying hundreds or thousands of rows, however, it would be a waste of memory to create hundreds or thousands of `View` objects when only eight 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 available it passes an unused view in the `convertView` parameter. If this value is null then your code should create a new view instance, otherwise you can re-set the properties of that object and re-use it.

The `GetView` method should follow this pattern to re-use row views:

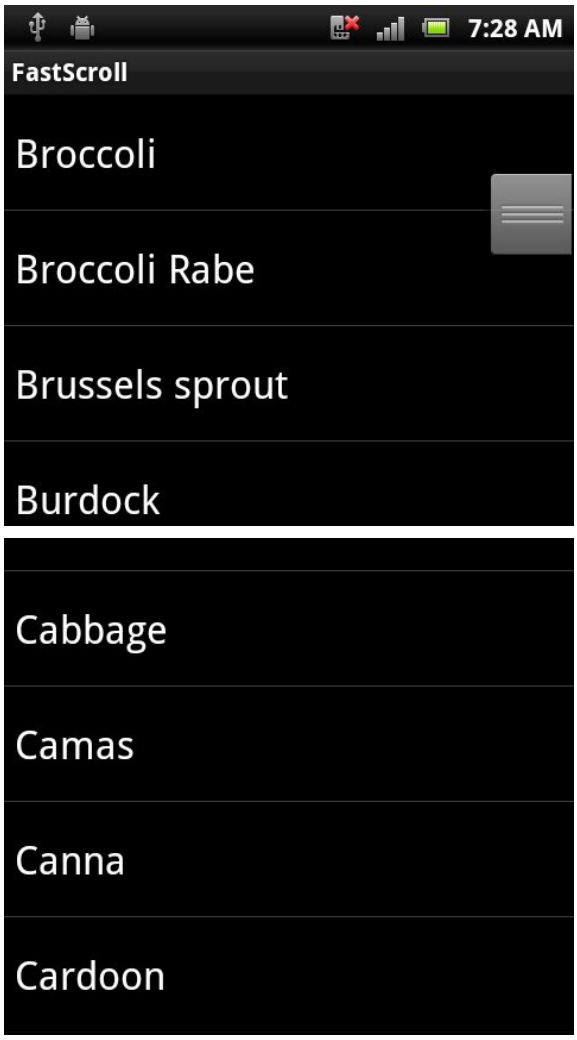
```
public override View GetView(int position, View convertView, ViewGroup
parent)
{
   View view = convertView; // re-use an existing view, if one is
supplied
   if (view == null) // otherwise create a new one
       view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1,
null);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

Some adapter implementations (such as the `CursorAdapter`) don't have a `GetView` method, rather they require two different methods `NewView` and `BindView` which enforce row re-use by separating the responsibilities of `GetView` into two methods. There is a `CursorAdapter` example later in the document.

Enabling Fast Scrolling

Fast Scrolling helps the user to scroll through long lists by providing an additional 'handle' that acts as a scroll bar to directly access a part of the list. This screenshot shows the fast scroll handle:



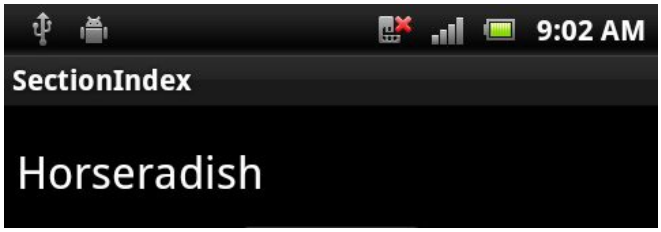
Causing the fast scrolling handle to appear is as simple as setting the `FastScrollEnabled`

property to `true`:

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

Adding a Section Index

A section index provides additional feedback for users when they are fast-scrolling through a long list – it shows which 'section' they have scrolled to. To cause the section index to appear the Adapter subclass must implement the `ISectionIndexer` interface to supply the index text depending on the rows being displayed:



To implement `ISectionIndexer` you need to add three methods to an adapter:

- 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.

The example `SectionIndex/HomeScreenAdapter.cs` file implements those methods, and some additional code in the constructor. The constructor builds the section index by looping through every row and extracting the first character of the title (the items must already be sorted for this to work).

```
alphaIndex = new Dictionary<string, int>();

for (int i = 0; i < items.Length; i++) { // loop through items
   var key = items[i][0].ToString();
   if (!alphaIndex.ContainsKey(key))

       alphaIndex.Add(key, i); // add each 'new' letter to the index

}

sections = new string[alphaIndex.Keys.Count];
alphaIndex.Keys.CopyTo(sections, 0); // convert letters list to
string[]
// Interface requires a Java.Lang.Object[], so we create one here
sectionsObjects = new Java.Lang.Object[sections.Length];
for (int i = 0; i < sections.Length; i++) {
   sectionsObjects[i] = new Java.Lang.String(sections[i]);
}
```

With the data structures created, the `ISectionIndexer` methods are very simple:

```
public Java.Lang.Object[] GetSections()

{

   return sectionsObjects;

}

public int GetPositionForSection(int section)

{

   return alphaIndexer[sections[section]];

}

public int GetSectionForPosition(int position)

        {      // this method isn't called in this example, but code is
provided for completeness
```

```
        int prevSection = 0;
        for (int i = 0; i < sections.Length; i++)
        {
            if (GetPositionForSection(i) > position)
            {
                break;
            }
            prevSection = i;
        }
        return prevSection;
    }
```

Your section index titles don't need to map 1:1 to your 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.

Part 3 - Customizing a ListView's Appearance

The appearance of a ListView is dictated by the layout of the rows being displayed. To change the appearance of a `ListView`> use a different row layout.

Built-in Row Views

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

TestListItem – Single line of text with minimal formatting.

SimpleListItem1 – Single line of text.

SimpleListItem2 – Two lines of text.

SimpleSelectableListItem – Single line of text that supports single or multiple item selection (added in API level 11).

SimpleListItemActivated1 – Similar to SimpleListItem1, but the background color indicates when a row is selected (added in API level 11).

SimpleListItemActivated2 – Similar to SimpleListItem2, but the background color indicates when a row is selected (added in API level 11).

SimpleListItemChecked – Displays check marks to indicate selection.
SimpleListItemMultipleChoice – Displays check boxes to indicate multiple-choice

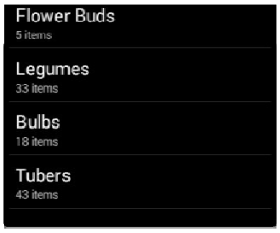
selection. SimpleListItemSingleChoice – Displays radio buttons to indicate mutually-exclusive selection. TwoLineListItem – Two lines of text.
ActivityListItem – Single line of text with an image.

SimpleExpandableListItem – Groups rows by categories, and each group can be expanded or collapsed. Each built-in row view has a built in style associated with it. These screenshots show how each view appears:

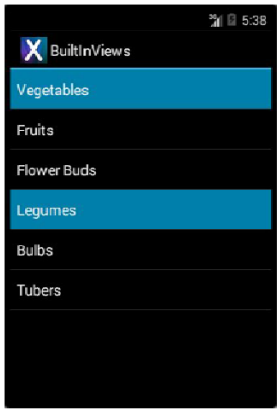
TestListItem

SimpleSelectableListItem

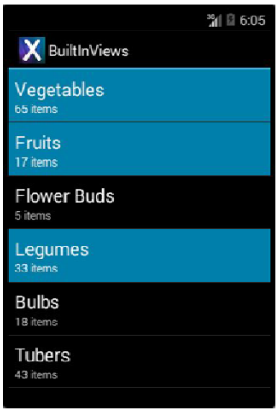
SimpleListItem1



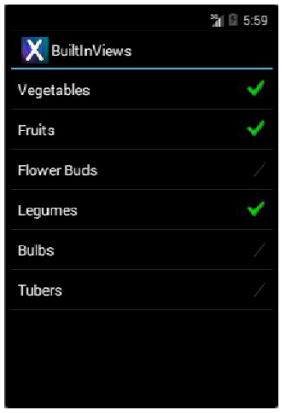
SimpleListItem2



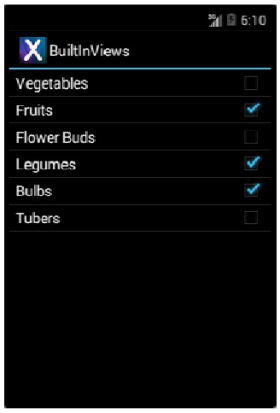
SimpleListItemActivated1



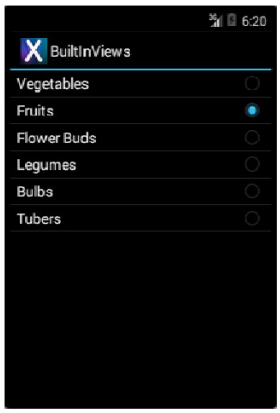
SimpleListItemActivated2



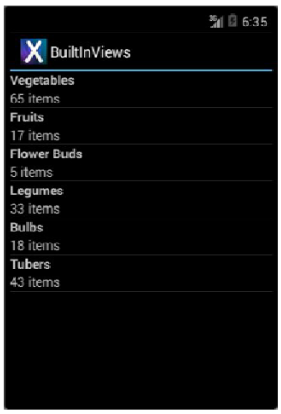
SimpleListItemChecked



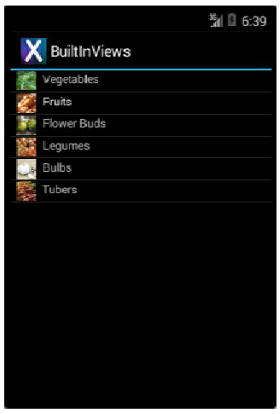
SimpleListItemMultipleChoice



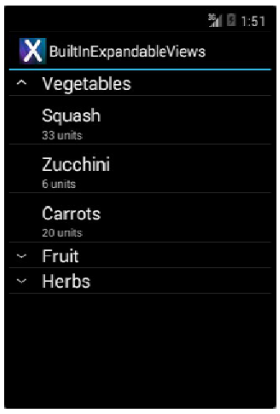
SimpleListItemSingleChoice



TwoLineListItem



ActivityListItem



SimpleExpandableListItem

The `BuiltInViews/HomeScreenAdapter.cs` sample file (in the BuiltInViews solution) contains the code to produce the non-expandable list item screens. The view is set in the `GetView` method like this:

```
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(item.ImageResourceId); // only
use with ActivityListItem
```

The `BuiltInExpandableViews/ExpandableScreenAdapter.cs` sample file (in the BuiltInViews solution) contains the code to produce the SimpleExpandableListItem screen. The group view is set in the `GetGroupView` method like this:

```
view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleExpandableListItem1,
null);
```

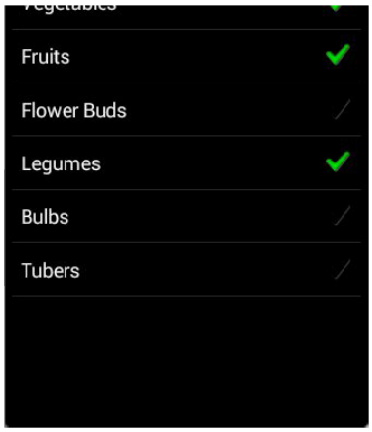
The child view is set in the `GetChildView` method like this:

```
view =
context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleExpandableListItem2,
null);
```

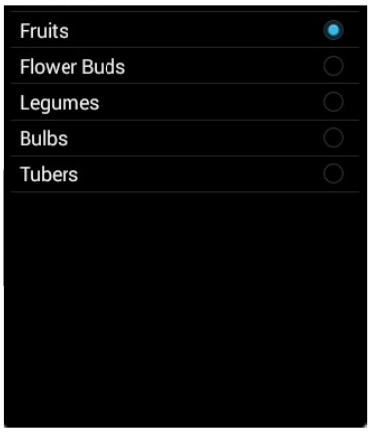
The properties for the group view and the child view can then be set by referencing the standard `Text1` and `Text2` control identifiers as shown above. The SimpleExpandableListItem screenshot (shown above) provides an example of a one-line group view (SimpleExpandableListItem1) and a two-line child view (SimpleExpandableListItem2). Alternately, the group view can be configured for two lines (SimpleExpandableListItem2) and the child view can be configured for one line (SimpleExpandableListItem1), or both group view and child view can have the same number of lines.

Accessories

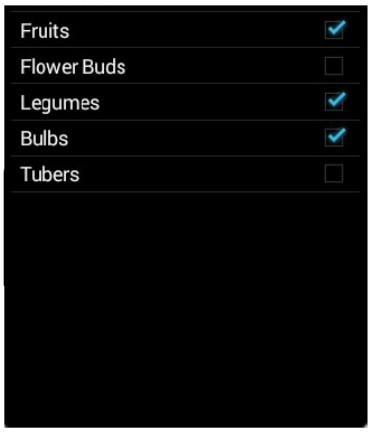
Rows can have accessories added to the right of the view to indicate selection state: SimpleListItemChecked – Creates a single-selection list with a check as the indicator. SimpleListItemSingleChoice – Creates radio-button-type lists where only one choice is possible. SimpleListItemMultipleChoice – Creates checkbox-type lists where multiple choices are possible. The aforementioned accessories are illustrated in the following screens, in their respective order:



SimpleListItemChecked



SimpleListItemSingleChoice



SimpleListItemMultipleChoice

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. This line of code shows how to create and assign an `Adapter` using
one of these layouts:

```
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`.

Handling API Level

Earlier versions of Xamarin.Android implemented enumerations as integer properties. The latest version has introduced proper .NET enumeration types which makes it much easier to discover the potential options.

Depending on which API level you are targeting, `ChoiceMode` is either an integer or an enumeration. The sample file `AccessoryViews/HomeScreen.cs` has a block commented out if you wish to target the Gingerbread API:

```
// For targeting Gingerbread the ChoiceMode is an int, otherwise it is an
// enumeration.


lv.ChoiceMode = Android.Widget.ChoiceMode.Single; // 1
//lv.ChoiceMode = Android.Widget.ChoiceMode.Multiple; // 2
//lv.ChoiceMode = Android.Widget.ChoiceMode.None; // 0
// Use this block if targeting Gingerbread or lower
/*
lv.ChoiceMode = Android.Widget.ChoiceMode.Single; // Single
//lv.ChoiceMode = 0; // none
//lv.ChoiceMode = 2; // Multiple
//lv.ChoiceMode = 3; // MultipleModal
*/
```

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` `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

The four built-in row views are very simple. To display more complex layouts (such as a list of emails, or tweets, or contact info) a custom view is required. Custom views are generally declared as AXML files in the `Resources/Layout` directory and then loaded using their resource Id 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.

This example differs from the previous examples in a number of ways:

- 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.
- 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.
- Requires an AXML layout file to render each row. This AXML file contains the text and image controls with custom font and color settings.
- Uses an optional custom selector XML file to set the appearance of the row when it is selected. The `Adapter` implementation returns a custom layout from the `GetView` override.
- `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.

Adding a ListView to an Activity Layout

Because `HomeScreen` no longer inherits from `ListActivity` it doesn't have a default view, so a layout AXML file must be created for the HomeScreen's view. For this example, the view will have a heading (using a `TextView`) and a `ListView` to display data. The layout is defined in the `/Resources/Layout/HomeScreen.axml` file which 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.

Creating a Custom Row Layout

Another AXML layout file is required to contain the custom layout for each row that will appear in the list view. In this example the row will have a green background, brown text and right-aligned image. The Android XML markup to declare this layout is described in `/Resources/Layout/CustomView.axml`:

```
<?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:background="#FFDAFF7F"
   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 >
```

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 article for more information on addressing scrolling performance issues.

Referencing a Custom Row View

The implementation of the custom adapter example is in `HomeScreenAdapter.cs`. The key method is `GetView` where it loads the custom AXML using the resource ID `Resource.Layout.CustomView`, and then sets properties on each of the controls in the view before returning it. The complete adapter class is shown:

```
public class HomeScreenAdapter : BaseAdapter<TableItem> {
   List<TableItem> items;
    Activity context;

   public HomeScreenAdapter(Activity context, List<TableItem> items)

         : base()

   {

       this.context = context;
       this.items = items;
   }

   public override long GetItemId(int position)

   {

       return position;

   }

   public override TableItem this[int position]

   {

       get { return items[position]; }

   }

   public override int Count

   {

       get { return items.Count; }

   }

   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,

       view.FindViewById<TextView>(Resource.Id.Text1).Text = item.Heading;
       view.FindViewById<TextView>(Resource.Id.Text2).Text = item.SubHeading;
null);  view.FindViewById<ImageView>
```

```
(Resource.Id.Image).SetImageResource(item.ImageResourceId);
        return view;
   }

}
```

Referencing the Custom ListView in the Activity

Because the `HomeScreen` class now inherits from `Activity`, a `ListView` field is declared in the class to hold a reference to the control declared in the AXML:

```
ListView listView;
```

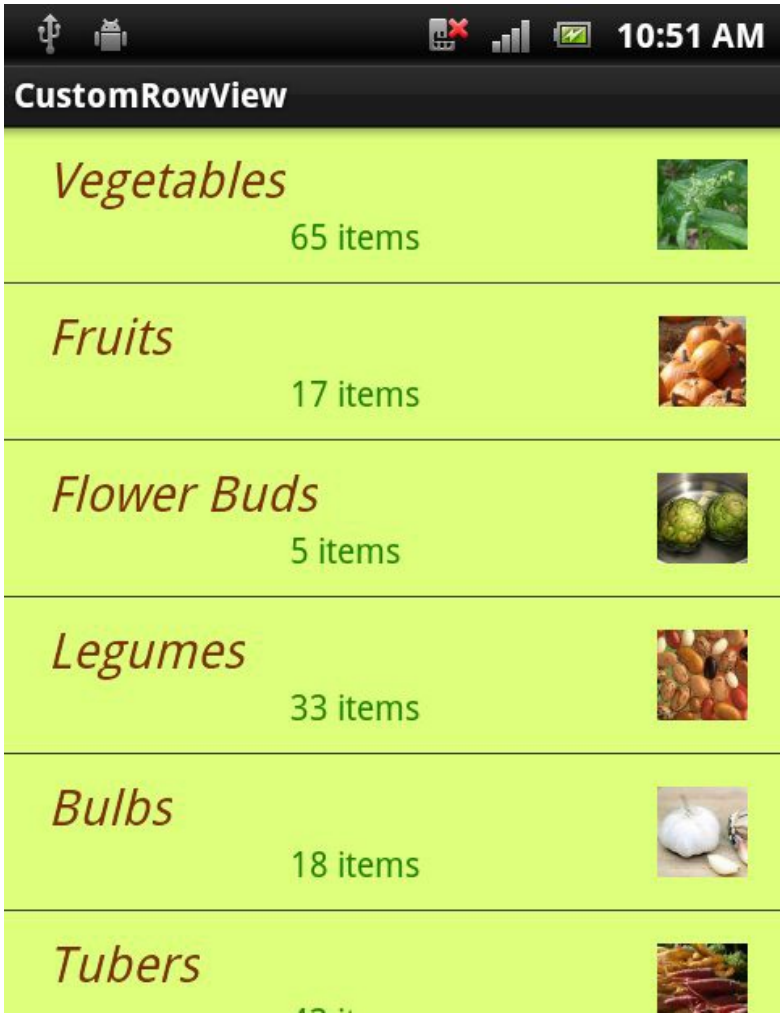
The class must then load the Activity's custom layout AXML using the `SetContentView` method. It can then find the `ListView` control in the layout then creates and assigns the adapter and assigns the click handler. The code for the OnCreate method is shown here:

```
SetContentView(Resource.Layout.HomeScreen); // loads the HomeScreen.axml as
this activity's view
listView = FindViewById<ListView>(Resource.Id.List); // get reference to the

ListView in the layout

// populate the listview with data

listView.Adapter = new HomeScreenAdapter(this, tableItems);
listView.ItemClick += OnListItemClick;  // to be defined
```

Finally the `ItemClick` handler must be defined; in this case it just displays a `Toast` message:

```
void OnListItemClick(object sender, AdapterView.ItemClickEventArgs e)

{

   var listView = sender as ListView;
   var t = tableItems[e.Position];
   Android.Widget.Toast.MakeText(this, t.Heading,
Android.Widget.ToastLength.Short).Show();
}
```

The resulting screen looks like this:



Customizing the Row Selector Color

When a row is touched it should be highlighted for user feedback. When a custom view specifies as background color as `CustomView.axml` does, it also overrides the selection highlight. This line of code in `CustomView.axml` sets the background to light green, but it also means there is no visual indicator when the row is touched:

```
android:background="#FFDAFF7F"
```

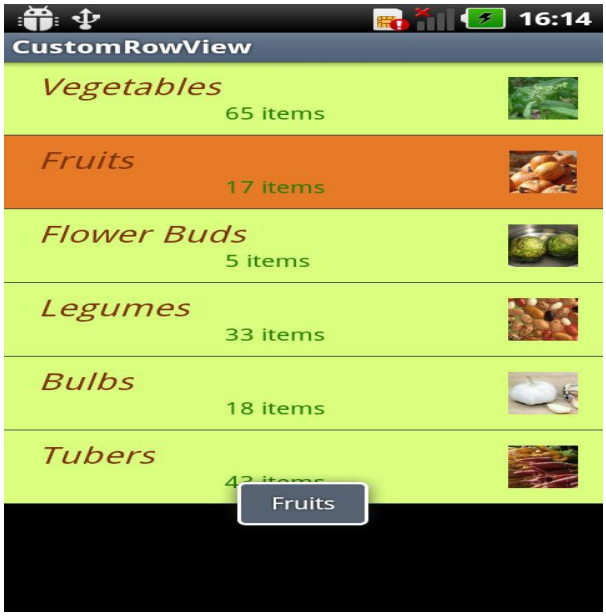
To re-enable the highlight behavior, and also to customize the color that is used, set the background attribute to a custom selector instead. The selector will declare both the default background color as well as the highlight color. The file `/Resources/Drawable/CustomSelector.xml` contains the following declaration:

```
<?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>
```

To reference the custom selector, change the background attribute in `CustomView.axml` to:

```
android:background="@drawable/CustomSelector"
```

A selected row and the corresponding `Toast` message now looks like this:



Preventing Flickering 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.

Part 4 - Using CursorAdapters

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

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.

Creating an SQLite Database

To demonstrate cursor adapters requires a simple SQLite database implementation. The code in SimpleCursorTableAdapter/VegetableDatabase.cs contains the code and SQL to create a table and populate it with some data. The complete `VegetableDatabase` class is shown here:

```
class VegetableDatabase  : SQLiteOpenHelper {

   public static readonly string create_table_sql =

       "CREATE TABLE [vegetables] ([_id] INTEGER PRIMARY KEY AUTOINCREMENT
NOT NULL UNIQUE, [name] TEXT NOT NULL UNIQUE)";
   public static readonly string DatabaseName = "vegetables.db";
   public static readonly int DatabaseVersion = 1;
   public VegetableDatabase(Context context) : base(context, DatabaseName,
null, DatabaseVersion) { }
   public override void OnCreate(SQLiteDatabase db)

   {

       db.ExecSQL(create_table_sql);

       // seed with data

       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Vegetables')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Fruits')");
```

```
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Flower Buds')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Legumes')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Bulbs')");
        db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Tubers')");
    }

    public override void OnUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion)
    {   // not required until second version :)
        throw new NotImplementedException();
    }

}
```

The `VegetableDatabase` class will be instantiated in the `OnCreate` method of the `HomeScreen` activity. The `SQLiteOpenHelper` base class manages the setup of the database file and ensures that the SQL in its `OnCreate` method is only run once. This class is used in the following two examples for `SimpleCursorAdapter` and `CursorAdapter`.

The cursor query *must* have an integer column `_id` for the `CursorAdapter` to work. If the underlying table does not have an integer column named `_id` then use a column alias for another unique integer in the `RawQuery` that makes up the cursor. Refer to the [Android docs]() for further information.

Creating the Cursor

The examples use a `RawQuery` to turn an SQL query into a `Cursor` object. The column list that is returned from the cursor defines the data columns that are available for display in the cursor adapter. The code that creates the database in the SimpleCursorTableAdapter/HomeScreen.cs `OnCreate` method is shown here:

```
vdb = new VegetableDatabase(this);

cursor = vdb.ReadableDatabase.RawQuery("SELECT * FROM vegetables", null); //
cursor query
StartManagingCursor(cursor);

// use either SimpleCursorAdapter or CursorAdapter subclass here!
```

Any code that calls `StartManagingCursor` should also call `StopManagingCursor`. The examples use `OnCreate` to start, and `OnDestroy` to close the cursor. The `OnDestroy` method contains this code:

```
StopManagingCursor(cursor);
cursor.Close();
```

Once an application has a SQLite database available and has created a cursor object as shown, it can utilize either a `SimpleCursorAdapter` or a subclass of `CusorAdapter` to display rows in a `ListView`.

Using SimpleCursorAdapter

`SimpleCursorAdapter` is like the `ArrayAdapter`, but specialized for use with SQLite. It does not require subclassing – just set some simple parameters when creating the object and then assign it to a `ListView`'s `Adapter` property.

The parameters for the SimpleCursorAdapter constructor are:

Context – A reference to the containing Activity.

Layout – The resource ID of the row view to use.

ICursor – A cursor containing the SQLite query for the data to display.

From string array – An array of strings corresponding to the names of columns in the cursor.

To integer array – An array of layout IDs that correspond to the controls in the row layout. The value of the column specified in the `from` array will be bound to the ControlID specified in this array at the same index.

The `from` and `to` arrays must have the same number of entries because they form a mapping from the data source to the layout controls in the view.

The SimpleCursorTableAdapter/HomeScreen.cs sample code wires up a `SimpleCursorAdapter`

like this:

```
// which columns map to which layout
controls string[] fromColumns = new
string[] {"name"};
int[] toControlIDs = new int[] {Android.Resource.Id.Text1};

// use a SimpleCursorAdapter

listView.Adapter = new SimpleCursorAdapter
(this, Android.Resource.Layout.SimpleListItem1,
cursor,
       fromColumns,
       toControlIDs);
```

`SimpleCursorAdapter` is a fast and simple way to display SQLite data in a `ListView`. The main limitation is that it can only bind column values to display controls, it does not allow you to change other aspects of the row layout (for example, showing/hiding controls or changing properties).

Subclassing CursorAdapter

A `CursorAdapter` subclass has the same performance benefits as the `SimpleCursorAdapter` for displaying data from SQLite, but it also gives you complete control over the creation and layout of each row view. The `CursorAdapter` implementation is very different from subclassing `BaseAdapter` because it does not override `GetView`, `GetItemId`, `Count` or `this[]` indexer.

Given a working SQLite database, you only need to override two methods to create a `CursorAdapter` subclass:

BindView – Given a view, update it to display the data in the provided cursor.
NewView – Called when the `ListView` requires a new view to display. The `CursorAdapter` will take care of recycling views (unlike the `GetView` method on regular Adapters).

The adapter subclasses in earlier examples have methods to return the number of rows and to retrieve the current item – the `CursorAdapter` does not require these methods because that information can be gleaned from the cursor itself. By splitting the creation and population of each view into these two methods, the `CursorAdapter` enforces view re-use. This is in contrast to a regular adapter where it's possible to ignore the `convertView` parameter of the `BaseAdapter.GetView` method.

Implementing the CursorAdapter

The code in CursorTableAdapter/HomeScreenCursorAdapter.cs contains a `CursorAdapter` subclass. It stores a context reference passed into the constructor so that it can access a `LayoutInflater` in the `NewView` method. The complete class looks like this:

```
public class HomeScreenCursorAdapter : CursorAdapter {
   Activity context;
   public HomeScreenCursorAdapter(Activity context, ICursor c)
       : base(context, c)
   {
       this.context = context;
   }
   public override void BindView(View view, Context context, ICursor
   cursor)
   {
       var textView = view.FindViewById<TextView>
(Android.Resource.Id.Text1);
       textView.Text = cursor.GetString(1); // 'name' is column 1 in the
cursor query
   }
   public override View NewView(Context context, ICursor cursor, ViewGroup
parent)
   {
       return
this.context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem
1, parent, false);
   }
```

```
}
```

Assigning the CursorAdapter

In the `Activity` that will display the `ListView`, create the cursor and `CursorAdapter` then assign it to the list view.

The code that performs this action in the CursorTableAdapter/HomeScreen.cs `OnCreate` method is shown here:

```
// create the cursor

vdb = new VegetableDatabase(this);

cursor = vdb.ReadableDatabase.RawQuery("SELECT * FROM vegetables",
null); StartManagingCursor(cursor);


// create the CursorAdapter

listView.Adapter = (IListAdapter)new HomeScreenCursorAdapter(this, cursor,
false);
```

The `OnDestroy` method contains the `StopManagingCursor` method call described previously.

CursorAdapters can also be used to display data from a ContentProvider. ContentProviders allow you to access data exposed by other applications (including Android system data like contacts, media and calendar information).

The preferred way to access a ContentProvider is with a CursorLoader using the LoaderManager. LoaderManager was introduced in Android 3.0 (API Level 11, Honeycomb) to move blocking tasks off the main thread, and using a CursorLoader allows the data to be loaded in a thread before being bound to a ListView for display.

Refer to the Intro to ContentProviders document for more information.

Part 6 - ListView and the Activity Lifecycle

Activities go through certain states as your application runs, such as starting up, running, being paused and being stopped. For more information, and specific guidelines on handling state transitions, see the Activity Lifecycle Tutorial. It is important to understand the activity lifecycle and place your `ListView` code in the correct locations.

All of the examples in this document perform 'setup tasks' in the Activity's `OnCreate`

method and (when required) perform 'teardown' in `OnDestroy`. The examples generally use small data sets that do not change, so re-loading the data more frequently is unnecessary.

However, if your data is frequently changing or uses a lot of memory it might be appropriate to use different lifecycle methods to populate and refresh your `ListView`. For example, if the underlying data is constantly changing (or may be affected by updates on other activities) then creating the adapter in `OnStart` or `OnResume` will ensure the latest data is displayed each time the Activity is shown.

If the Adapter uses resources like memory, or a managed cursor, remember to release those resources in the complementary method to where they were instantiated (eg. objects created in `OnStart` can be disposed of in `OnStop`).

Configuration Changes

It's important to remember that configuration changes – especially screen rotation and keyboard visibility – can cause the current activity to be destroyed and re-created (unless you specify otherwise using the `ConfigurationChanges` attribute). This means that under normal conditions, rotating a device will cause a `ListView` and `Adapter` to be re-created and (unless you have written code in `OnPause` and `OnResume`) the scroll position and row selection states will be lost.

The following attribute would prevent an activity from being destroyed and recreated as a
result of configuration changes:

```
[Activity(ConfigurationChanges="keyboardHidden|orientation")]
```

The Activity should then override `OnConfigurationChanged` to respond to those changes appropriately. For more details on how to handle configuration changes see the documentation.

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.

This article introduced some simple examples using the built-in features of the `ListActivity` and then discussed custom implementations of `ListView` that allowed for colorful layouts and using an SQLite database.

It also briefly touched on the relevance of the activity lifecycle on your `ListView` implementation.