# More User Interface (UI)

Let's continue our discussion of the Android UI and its layout options.

## What this lecture will teach you

* Create ListView layout
* Use adapters and click listeners
* Create picker widgets
* Define some coding/naming style guidelines
* Save user data using SharedPreferfence
* Introduce higher-level UI components
* Storage options

## ListView Layout: Click on a State

A layout defines the visual structure for a user interface, such as the UI for an activity or app widget. You can declare a layout in two ways:

* **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
* **Instantiate layout elements at runtime**. Your application can create View and ViewGroup objects (and manipulate their properties) programmatically.

The view that we expect to see is

|  |  |
| --- | --- |
| **Main Screen** | **ListViewLayout Screen** |
|  |  |

First we add some a new buttons in the layout of the MainActivity hat will fire the ListViewLayoutActivity Activity Screen.

In this case the ListView defined not in the XML -- but in the code.

ListView is a view group that displays a list of scrollable items. The list items are automatically inserted to the list using an Adapter that pulls content from a source such as an array or database query and converts each item result into a view that's placed into the list.

First note that we do not extend activity in ListViewLayoutActivity.java rather ListViewLayoutActivity extends ListActivity.

ListActivity hosts a [ListView](https://developer.android.com/reference/android/widget/ListView.html) object that can be bound to different data sources, typically either an array or a Cursor, using a number of methods that help create, manage and control ListView-s.

**public class** ListViewLayoutActivity **extends** ListActivity {  
 **static final** String[] ***STATES*** = **new** String[] { **"Alabama"**, **"Alaska"**,

**Arizona"**,**"Arkansas"**, **"California"**, **"Colorado"**, **"Connecticut"**,

**"Delaware"**, **"Florida"**,**"Georgia"**, **"Hawaii"**, **"Idaho"**, **"Illinois"**,

**"Indiana"**, **"Iowa"**, **"Kansas"**, **"Kentucky"**, **"Louisiana"**, **"Maine"**,

**"Maryland"**, **"Massachusetts"**, **"Michigan"**,**"Minnesota"**, **"Mississippi"**,

**"Missouri"**, **"Montana"**, **"Nebraska"**, **"Nevada"**,**"New Hampshire"**,

**"New Jersey"**, **"New Mexico"**, **"New York"**, **"North Carolina"**,

**"North Dakota"**, **"Ohio"**, **"Oklahoma"**, **"Oregon"**, **"Pennsylvania"**,

**"Rhode Island"**,**"South Carolina"**, **"South Dakota"**, **"Tennessee"**, **"Texas"**,

**"Utah"**, **"Vermont"**, **"Virginia"**, **"Washington"**, **"West Virginia"**,

**"Wisconsin"**, **"Wyoming"**};  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 *// Don't have to do this anymore  
 // setContentView(R.layout.listview\_layout);  
  
 // Define a new adapter* ArrayAdapter<String> mAdapter = **new** ArrayAdapter<String>(**this**,  
 R.layout.***activity\_list\_view\_layout***, ***STATES***);  
  
 *// Assign the adapter to ListView object* setListAdapter(mAdapter);  
  
 *// Define the listener interface* AdapterView.OnItemClickListener mListener =

**new** AdapterView.OnItemClickListener() {  
 **public void** onItemClick(AdapterView<?> parent, View view,  
 **int** position, **long** id) {  
 *// When clicked, show a toast with the TextView text* Toast.*makeText*(getApplicationContext(),  
 ((TextView) view).getText() +

**" is one of the North America States!"**,  
 Toast.***LENGTH\_SHORT***).show();  
 }  
 };  
  
 *// Get the ListView and wired the listener* ListView listView = getListView();  
 listView.setOnItemClickListener(mListener);  
 }  
}

There are some steps performed in the code above

1. Initialize a new ArrayAdapter using a constructor to specify the layout for each string and the string array STATES. If you look at the XML you will see that only a TextView is defined -- for one of the elements in the ListView table: the height, width, padding and text size are all set up statically in the XML.

<**TextView xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="fill\_parent"  
 android:layout\_height="fill\_parent"  
 android:padding="10dp"  
 android:textSize="20sp"** >  
</**TextView**>

We specify the layout of individual rows in the list using a TextView in the XML. A ListAdapter constructor takes a parameter that specifies a layout resource for each row -- activity\_list\_view\_layout.xml-- that specifies the row template to use, that is: TextView.

ArrayAdapter<String> mAdapter = **new**

ArrayAdapter<String>(**this**,R.layout.***activity\_list\_view\_layout***, ***STATES***);

Where parameters are:

1. The app Context
2. The layout that contains a TextView for each string in the array
3. The string array
4. ListActivity is a specialized Activity that hosts a ListView and the Adapter is assigned to this ListView.

*// Assign the adapter to ListView* setListAdapter(mAdapter);

1. Interface definition for a callback to be invoked when an item in this AdapterView (Which is the ListView in our case) has been clicked (public static interface AdapterView.OnItemClickListener).

AdapterView.OnItemClickListener mListener =

**new** AdapterView.OnItemClickListener() {  
 **public void** onItemClick(AdapterView<?> parent, View view,  
 **int** position, **long** id) {  
 *// When clicked, show a toast with the TextView text* Toast.*makeText*(getApplicationContext(),  
 ((TextView) view).getText() + **" is one of the North America States!"**,  
 Toast.***LENGTH\_SHORT***).show();  
 }  
 };

parent AdapterView: The ListView where the click happened.

(ListView is a subclass of AdapterView)

view View: The view provided by the adapter that corresponds to a row)

position int: The position of the view in the adapter.

Id long: The row id of the item that was clicked.

1. callback is invoked

*// Get the ListView and wired the listener*ListView listView = getListView();  
listView.setOnItemClickListener(mListener);

## Some coding/naming style guidelines

There are a number of good guidelines out there for writing Android code. Please try and follow these field naming conventions:

* Non-public, non-static field names start with m.
* Static field names start with s.
* Other fields start with a lower-case letter.
* Public static final fields (constants) are ALL\_CAPS\_WITH\_UNDERSCORES.

For example:

public class MyClass {

public static final int SOME\_CONSTANT = 42;

public int publicField;

private static MyClass sSingleton;

int mPackagePrivate;

private int mPrivate;

protected int mProtected;

}

## Other layouts: using date and time pickers

Android provides a set of standard widgets for setting the date and time -- these are called Pickers http://developer.android.com/guide/topics/ui/controls/pickers.html.

First we add a new buttons in the layout of the MainActivity that will fire an activity about Date and Time.

Let create the Activity with the name DateAndTimeActivity with the respective layout with the name activity\_date\_and\_time\_layout.

When we run the program we want to have the following view.

|  |  |  |  |
| --- | --- | --- | --- |
| **Main Screen** | **DateAndTimeActivity Screen 1** | **DateAndTimeActivity Screen 2** | **DateAndTimeActivity Screen 3** |
|  |  |  |  |

The date\_time\_layout.xml is straightforward (**DateAndTimeActivity Screen 1**). A TextView is set up for displaying the date and time -- not the large than standard font (i.e., 30sp) is used. Two buttons are used to display the pickers, which are really wrapped dialog boxes. More on dialogs later. The onClick callbacks are set up in the XML as usual. The text displayed on each button is defined too.

|  |  |  |
| --- | --- | --- |
| <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="fill\_parent"  android:layout\_height="fill\_parent"  android:orientation="vertical" >  . . .  </LinearLayout> |  | |
| <TextView  android:id="@+id/dateTime"  android:layout\_width="fill\_parent"  android:layout\_height="wrap\_content"  android:text="Set me!"  android:textSize="30sp" /> |  | |
| <**Button  android:id="@+id/dateBtn"  android:layout\_width="fill\_parent"  android:layout\_height="wrap\_content"  android:onClick="onDateClicked"  android:text="@string/ui\_button\_date\_title"** /> |  | |
| **public void** onDateClicked(View v) {   DatePickerDialog.OnDateSetListener mDateListener = **new**  DatePickerDialog.OnDateSetListener() {  **public void** onDateSet(DatePicker view, **int** year, **int** monthOfYear,  **int** dayOfMonth) {  **mDateAndTime**.set(Calendar.***YEAR***, year);  **mDateAndTime**.set(Calendar.***MONTH***, monthOfYear);  **mDateAndTime**.set(Calendar.***DAY\_OF\_MONTH***, dayOfMonth);  updateDateAndTimeDisplay();  }  };   **new** DatePickerDialog(DateAndTimeActivity.**this**, mDateListener,  **mDateAndTime**.get(Calendar.***YEAR***),  **mDateAndTime**.get(Calendar.***MONTH***),  **mDateAndTime**.get(Calendar.***DAY\_OF\_MONTH***)).show();  } | | |
| 1. When a **callback** onDateClicked(View v)will be invoked,   the interface DatePickerDialog.OnDateSetListener is implemented here assigning the object **mDateAndTime** some information like year, month, day of month.  This information will be used when date picker dialog below is created.   1. Creates a new date picker dialog for the specified date.   **new** DatePickerDialog(DateAndTimeActivity.**this**, mDateListener,  **mDateAndTime**.get(Calendar.***YEAR***),  **mDateAndTime**.get(Calendar.***MONTH***),  **mDateAndTime**.get(Calendar.***DAY\_OF\_MONTH***)).show();  The object **mDateAndTime** is declared before its use.  Calendar **mDateAndTime** = Calendar.*getInstance*(); This declaration | | |
| <**Button  android:id="@+id/timeBtn"  android:layout\_width="fill\_parent"  android:layout\_height="wrap\_content"  android:onClick="onTimeClicked"  android:text="@string/ui\_button\_time\_title"** /> | |  |
| **public void** onTimeClicked(View v) {   TimePickerDialog.OnTimeSetListener mTimeListener = **new**  TimePickerDialog.OnTimeSetListener() {  **public void** onTimeSet(TimePicker view, **int** hourOfDay, **int** minute) {  **mDateAndTime**.set(Calendar.***HOUR\_OF\_DAY***, hourOfDay);  **mDateAndTime**.set(Calendar.***MINUTE***, minute);  updateDateAndTimeDisplay();  }  };   **new** TimePickerDialog(DateAndTimeActivity.**this**, mTimeListener,  **mDateAndTime**.get(Calendar.***HOUR\_OF\_DAY***),  **mDateAndTime**.get(Calendar.***MINUTE***), **true**).show();  } | | |
| 1. When a **callback** onTimeClicked(View v)will be invoked, the interface TimePickerDialog.OnTimeSetListener   is implemented here assigning the object **mDateAndTime** some information like hours of day, minutes  This information will be used when date picker dialog below is created.   1. Creates a new date picker dialog for the specified date.   **new** TimePickerDialog(DateAndTimeActivity.**this**, mTimeListener,  **mDateAndTime**.get(Calendar.***HOUR\_OF\_DAY***),  **mDateAndTime**.get(Calendar.***MINUTE***), **true**).show(); | | |

Take a look at the code below to see the callback processing on the DatePickerDialog object for example. The date is year, month, and day.

The TimePickerDialog provides a similar call back object so you can access the time as a 24.00 hour clock (if configured) or a 12 hour clock with AM/PM. Time is hours and minutes.

Let's discuss the code solution that handles the interaction with the pickers and display of the current time in the TextView -- as shown below in DateAndTimeActivity.java. The Calendar instance sets the data and time in onCreate() by calling the helper method updateDateAndTimeDisplay(); the method uses takes the current data and time from the Calendar and uses DateUtils object to format the date and time correctly for the TextView *dateTime* defined in the XML. DateUtils contains various date-related utilities for creating text for things like elapsed time and date ranges, strings for days of the week and months, and AM/PM text etc.

**public class** DateAndTimeActivity **extends** Activity {

TextView **mDisplayDateTime**;  
Calendar **mDateAndTime** = Calendar.*getInstance*();  
  
@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
  
 setContentView(R.layout.***activity\_date\_and\_time***);  
  
 **mDisplayDateTime** = (TextView) findViewById(R.id.***dateTime***);  
  
 updateDateAndTimeDisplay();  
  
}

. . .

**private void** updateDateAndTimeDisplay() {  
 **mDisplayDateTime**.setText(DateUtils.*formatDateTime*(**this**,  
 **mDateAndTime**.getTimeInMillis(), DateUtils.***FORMAT\_SHOW\_DATE*** | DateUtils.***FORMAT\_SHOW\_TIME***));  
}

}

Note, Android developers use (most of the time anonymous) objects[[1]](#footnote-1) to define specialized listeners such as TimePickerDialog.OnTimeSetListener -- new TimePickerDialog.OnTimeSetListener() -- which register a callback, to implement the program to take care of events such as set time. For example, in the case of new TimePickerDialog.OnTimeSetListener() it takes no specific parameters but creates a nested object to handle the callback. What the user clicks on *Done* to set the time onTimeSet(TimePicker view, int hourOfDay, int minute) method is called to implement the specific behavior of the user event.

**SharedPrefences: Storing user data**

What if you wanted to save user data so when the app opens again you don't have to renter some of the data. Android allows you to do this in a number of ways. You can use a simple SharedPreference object to store small amounts of user data. For more sophisticated data storage we will use databases and particular SQLite -- we will build an app that uses SQLite later in the course. But for now, let's assume we want to save a small amount of user data between invocations of our layout app.

First we add a new buttons in the layout of the MainActivity that will fire an activity SharedPreferences with the respective layout with the name activity\_shared\_preferences.

When the application run we need to have these views:

|  |  |
| --- | --- |
| **Main Screen** | **SharedPreferences** |
|  |  |

|  |
| --- |
| Let see in detail the XML layout of this Activity |
| <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="fill\_parent"  android:layout\_height="fill\_parent"  android:orientation="vertical" >  . . .  </LinearLayout> |  |
| <**TextView  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_margin="5dp"  android:text="@string/ui\_profile\_email\_title"** > </**TextView**> |  |
| <**EditText  android:id="@+id/editEmail"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:layout\_margin="5dp"  android:hint="@string/ui\_profile\_email\_hint"  android:inputType="textEmailAddress"  android:singleLine="true"** > </**EditText**> |  |
| <**TextView  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_margin="5dp"  android:text="@string/ui\_profile\_gender\_title"** > </**TextView**> |  |
| <**RadioGroup  android:id="@+id/radioGender"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:layout\_margin="5dp"  android:orientation="horizontal"** >   <**RadioButton  android:id="@+id/radioGenderF"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_weight="1"  android:text=**  **"@string/ui\_profile\_gender\_female"** />  <**RadioButton  android:id="@+id/radioGenderM"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_weight="1"**  **android:text="**  **@string/ui\_profile\_gender\_male"** /> </**RadioGroup**> |  |
| <**LinearLayout  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:layout\_margin="5dp"** >  <**Button  android:id="@+id/btnSave"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginLeft="40dp"  android:layout\_weight="1"  android:onClick="onSaveClicked"  android:text="@string/ui\_button\_save\_title"** >  </**Button**>  . . . |  |
| **public class** SharedPreferencesActivity **extends** Activity {   **private static final** String ***TAG*** = **"CS65"**;  **public static final** String ***PREFS\_MYRUNS*** = **"MyPrefs"**;  @Override **protected void** onCreate(Bundle savedInstanceState) {  **. . . .** }  **public void** onSaveClicked(View v) {   *// Save all information from the screen into a "shared preferences"  // using private helper function* saveUserData();    // go back in Main Activity  . . .   } | |
|  | |
| <**Button  android:id="@+id/btnCancel"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginRight="40dp"  android:layout\_weight="1"  android:onClick="onCancelClicked"  android:text="@string/ui\_button\_cancel\_title"** >  </**Button**> </**LinearLayout**> |  |
| **public void** onCancelClicked(View v) {   Toast.*makeText*(getApplicationContext(),  getString(R.string.***cancel\_message***), Toast.***LENGTH\_SHORT***).show();   // go back in Main Activity  . . .   } | |

Following this we discuss two helper methods used to support the saving and restoring the user data -- saveUserData() and loadUserData(). Frist, note that SharedPreferencesActivity extends Activity and onCreate() simply set the view and calls the helper to load any stored data. Note, the first time the app runs there is no stored data so this is an edge case that needs to be taken care of in the code -- we will discuss this edge case when we discuss the helper methods.

**public class** SharedPreferencesActivity **extends** Activity {

@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
  
 setContentView(R.layout.***activity\_shared\_preferences***);  
  
 *// Load user data to screen using the private helper function  
 // loadProfile* loadUserData();

}

}

**Helper functions - loadUserData() and saveUserData()**

loadUserData() first gets the key for the preference name which is used to obtain an instance of the SharedPreferences class. This is the object what we will store data using key/data pairs to save and restore (using the other helper function). Specifically, getSharedPreferences() has two parameters:

* name: the preferences file (mKey in our case). If a preferences file by this name does not exist, it will be created when you retrieve an editor (SharedPreferences.edit()) and then commit changes (Editor.commit()).
* mode:the operating mode. The MODE\_PRIVATE specifies that the preference file can only be accessed by the application that created it.

The getSharedPreferences() method retrieve the contents of the preferences file 'name', returning a SharedPreferences (i.e., mPrefs) through which you can retrieve and modify its values. Only one instance of the SharedPreferences object is returned to any callers for the same name, meaning they will see each other's edits as soon as they are made.

To update the value of preference we first use getString() to get the key -- for example, preference\_key\_profile\_email -- and then get the string value for the email using the mKey in this case. Note, if there is nothing stored for that key (i.e., no email details have been saved) then we use the default of an empty string. Once we have the value we update the EditText view on the screen by getting the view and then setting the restored value.

((EditText) findViewById(R.id.editEmail)).setText(mValue);

This is repeated for all views in the shared\_preferences\_layout.xml file, as shown above. The RadioButtons do not store strings but an integer value where -1 is the default used if nothing has already been saved and 0/1 for the gender buttons female/male from the order of the widgets in the

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* private helper functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  
  
// load the user data from shared preferences if there is no data make sure  
// that we set it to something reasonable***private void** loadUserData() {  
  
 *// We can also use log.d to print to the LogCat* Log.*d*(***TAG***, **"loadUserData()"**);  
  
 *// Load and update all profile views  
  
 // Get the shared preferences - create or retrieve the activity  
 // preference object* String mKey = getString(R.string.***preference\_name***);  
 SharedPreferences mPrefs = getSharedPreferences(mKey, ***MODE\_PRIVATE***);  
  
 *// Load the user email* mKey = getString(R.string.***preference\_key\_profile\_email***);  
 String mValue = mPrefs.getString(mKey, **" "**);  
 ((EditText) findViewById(R.id.***editEmail***)).setText(mValue);  
  
 *// Please Load gender info and set radio box* mKey = getString(R.string.***preference\_key\_profile\_gender***);  
  
 **int** mIntValue = mPrefs.getInt(mKey, -1);  
 *// In case there isn't one saved before:* **if** (mIntValue >= 0) {  
 *// Find the radio button that should be checked.* RadioButton radioBtn = (RadioButton) ((RadioGroup)

findViewById(R.id.***radioGender***))  
 .getChildAt(mIntValue);  
 *// Check the button.* radioBtn.setChecked(**true**);  
 Toast.*makeText*(getApplicationContext(),  
 **"number of the radioButton is : "** + mIntValue,  
 Toast.***LENGTH\_SHORT***).show();

}

}

*// load the user data from shared preferences if there is no data make sure  
// that we set it to something reasonable***private void** saveUserData() {  
  
 Log.*d*(***TAG***, **"saveUserData()"**);  
  
 *// Getting the shared preferences editor* String mKey = getString(R.string.***preference\_name***);  
 SharedPreferences mPrefs = getSharedPreferences(mKey, ***MODE\_PRIVATE***);  
  
 SharedPreferences.Editor mEditor = mPrefs.edit();  
 mEditor.clear();  
  
 *// Save email information* mKey = getString(R.string.***preference\_key\_profile\_email***);  
 String mValue = (String) ((EditText) findViewById(R.id.***editEmail***))  
 .getText().toString();  
 mEditor.putString(mKey, mValue);  
  
 *// Read which index the radio is checked.  
  
 // edit this out and use as a debug example  
 // interesting bug because you try and write an int to a string* mKey = getString(R.string.***preference\_key\_profile\_gender***);  
  
 RadioGroup mRadioGroup = (RadioGroup) findViewById(R.id.***radioGender***);  
 **int** mIntValue = mRadioGroup.indexOfChild(findViewById(mRadioGroup  
 .getCheckedRadioButtonId()));  
 mEditor.putInt(mKey, mIntValue);  
  
 *// Commit all the changes into the shared preference* mEditor.commit();  
  
 Toast.*makeText*(getApplicationContext(), **"saved name: "** + mValue,  
 Toast.***LENGTH\_SHORT***).show();  
  
}

Once the loadUserData() is complete the screen is updated with any stored data. Now, assume the user has changed these values (email and gender) and clicked the Save button. The following happens. The callback calls the saveUserData() helper function. The helper function gets the preference file key and gets the reference to the SharedPeference object. While same key/value pairs are used to store data in the object the difference here is that an editor is needed to update the values. We first create (SharedPreferences.Editor object) and clear an editor through the mPrefs.edit() method. Following this we get the key of the value we want to update and then use mEditor.putString() to change the value. To save all the changes to the preference file we use mEditor.commit(). Finally, the helper displays the toast informing the user that their email address and gender information has been successfully saved.

**Storage options**

As discussed above there are a number of options of storing persistent application data in Android:

* using Shared Preferences for primitive data such as int, strings, etc.
* using internal or external device storage; and
* using SQLite databases for storing structured data.

The type of storage depends on what your application is trying to achieve.

**Assignment**

Implement the whole project in your computer. Make the following updates

1. Create a new Button in MainActivity’s xml ListViewLayout2. This button will create a new Activity:

ListViewLayoutActivity2.java, where the implementation of the list is done from xml file: you have to use

<ListView syntax.

In code you have to write:

. . .

simpleList = (ListView)findViewById(R.id.simpleListView);

ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(this, R.layout.activity\_listview, R.id.textView, countryList);

simpleList.setAdapter(arrayAdapter);

Display in the list an array with different information.

2.

Use the code of SharedReferencesActivity.java, in LinearLayoutActivity.java, and decide to save some of information and have this information displayed when you click again LinearLayout Button.

1. **Objects**-**Anonymous** **objects**. Whenever an **object** is instantiated but is not assigned a reference variable, it is called **anonymous** **object** instantiation [↑](#footnote-ref-1)