Lab 1 - Setting up the User’s Profile UI

Getting started

This is the first in a series of labs that allow you to develop the MyRuns App. The goal of the app is to capture and display (using maps) walks and runs using your Android phone.

The first lab simply focuses on developing a simple UI for capturing your profile; that is, your name, email, phone number, gender and major. The work flow is as follows: you present the user with a view that allows them to input and save their profile. When the App is opened again the profile as saved should be displayed, allowing the user to change any field.

We assume that you have installed the environment (use Android 2.3.3) and completed the Hello World tutorial. Before you start this lab complete the View Tutorial (see the class webpage for details) and read as much as chapter 3 and 4 as you can – you can use the book as reference but it’s also very readable. Chapter 3 introduces the UI, views and layouts and Chapter 4 goes into a lot more detail on your UI using views.

This lab mainly deals with activities and views, which we talked about in lecture 2.

Here is a summary of the code and what you have to do: 1) create project; 2) build the layout; 3) add event handling to handle user input of the profile; 4) save the profile as persistent data; and 5) restore the profile to the view/screen. In fact, 5 will come before 4 in the flow but the first time through the code there will be no saved context.

Because this is the first Android lab I provide a number of code snippets from the solution I showed you in class.

OK. Let’s get started.

Create a project

Create a project named MyRuns with a main activity called “ActivityProfile”. We will add to this project as we go along; that is, each new lab will add code to this project. This week we will create a single activity and screen (view).

Now let’s discuss the screen layout for the UI.

Design the Layout

The UI in Android is controlled by xml file in res/layout. Typically, each activity will be associated with a layout xml file. When the activity is created the `onCreate` event is fired and `setContentView` executed:

```
29
30      /** Called when the activity is first created. */
31      @Override
32      public void onCreate(Bundle savedInstanceState) {
33          super.onCreate(savedInstanceState);
34          setContentView(R.layout.main);
35
36
```

Figure 1 Using findViewById() and ID to get a reference

`setContentView` renders the layout declared in main.xml to the screen. If you run the app at this point it will display the title only.

Android does provide a "drag and drop" method as part of the graphical layout -- so you can us the xml or the graphical approach for your layout, or both -- I like to use both: you can easily switch between both modes -- for example, to see how the xml code you have written renders graphically.

The xml layout file consists of a simple linear or hierarchical list of widgets and layouts. If you have done the Views tutorial you are familiar with different types of layouts - from simple to more complex. You can specify the properties of each item in the xml, such id, size, position, color, alignment, padding, margin, etc.

Figure 1 shows the layout you need to "code" up using xml and/or the graphical tool.

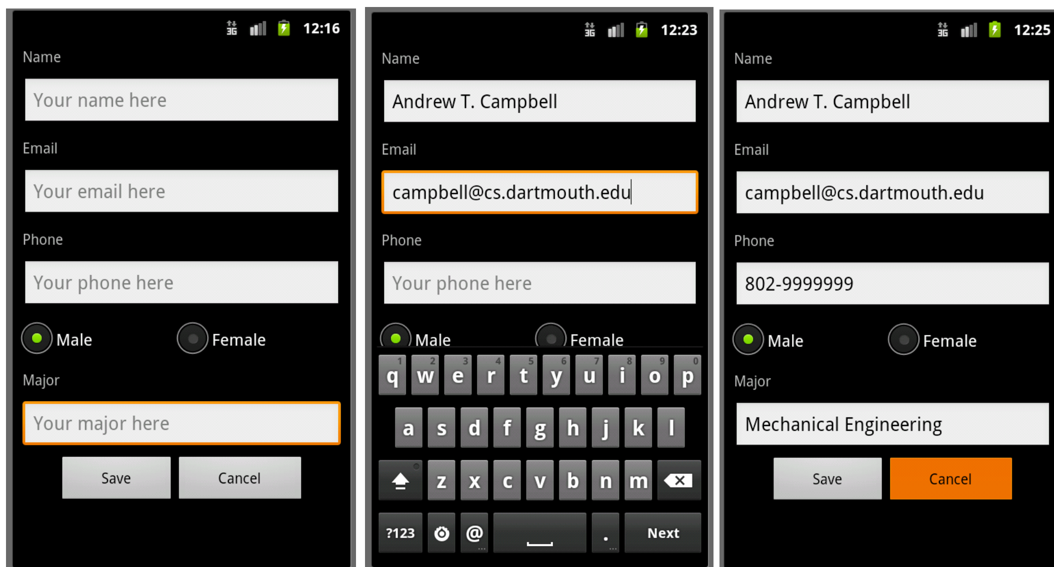

Figure 1(a): Profile layout design **Figure 1(b): User enters data** **Figure 1 (c): Now click save or cancel**

The book and tutorial will provide enough background to code up (xml) the layout in Figure 1(a). A screen comprises a layout with views - a view is a widget that has an appearance on the screen such as the `TextView` used for the phone, email and name in Figure 1(a). Other widgets are the `RadioGroup` for male and female. One or more views as in Figure 1(a) can be grouped together into a `ViewGroup` (e.g., `LinearLayout` which is a view itself) to provide the layout allowing you to order the appearance and sequence of views.

All views are in Figure 1(a) are in a root vertical linear layout; the views we use include `TextView` (i.e., Name, Email, Phone and Major), the editable boxes `EditText` used to enters data for name, email, etc. And, the gender `RadioButton` is grouped into a `RadioGroup`. Because the java code has to read in user input from the EditText and RadioButton views (which are presented as objects by the running code) it needs to get a reference to the view i.e., widget object – it does this by using the name of the ID that you provide in the xml of the view e.g., (android:id = "@+id/buttonSave") shown in Figure 2, which, again, you will refer to in your code when you want to read or restore the name of the user. It's not necessary to give every view an ID such as TextViews (which are hard code labels that do not change). You do need references to objects that your code needs to read and write to – and as the programmer you name and create these IDs – "buttonSave" in Figure 2.

As shown in Figure 1 we use "hints" to tell the user what to input: "your email here". Hints can be added to EditText views by using the "android:hint property.

Note, that all view properties can be specified programmatically but that is burdensome. Just use the xml or graphical layout to do this in a declarative manner. It is more maintainable to do the UI like this.

There are many tweaks to the position of widgets (e.g., `Button` view used for save and cancel) relative to each other. In the case of the save and cancel buttons they are "weighted" in proportion; you can do this by right clicking (see Figure 3) on the save and cancel widgets in the graphical layout mode or add the property directly to the xml, as show in Figure 2. Using right click you can change almost anything to do with the layout/view properties. Try changing some of the properties in the graphical layout – margins, gravity, etc. – have fun. Again, make a change in the graphical domain and checkout the xml representation. Once you get familiar with layouts and views you can simply write the xml directly. But until then the graphical mode is useful to learn representations of views.

```xml
<LinearLayout
    android:id="@+id/linearLayout1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <Button
        android:id="@+id/buttonSave"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="40dp"
        android:layout_weight="1"
        android:text="@string/save" />

    <Button
        android:id="@+id/buttonCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="40dp"
        android:layout_weight="1"
        android:text="@string/cancel" />

</LinearLayout>
```

Figure 2: xml button views and their properties

From Figure 2 you can see that both buttons have the same weight: `android:layout_weight="1"`. Play with the weights and other properties directly in the xml and switch to graphical layout to see the results. Note, in the examples in Figure 2 I have give the objects the IDs: `android:id="@+id/buttonCancel"` and `android:id="@+id/buttonSave"` so I can refer to the button's in my java code to see if the user has clicked save or cancel (discussed below). Also note, that because the two buttons are side by side we use a separate LinearLayout just for these two buttons – therefore this layout is embedded in the parent or root layout.

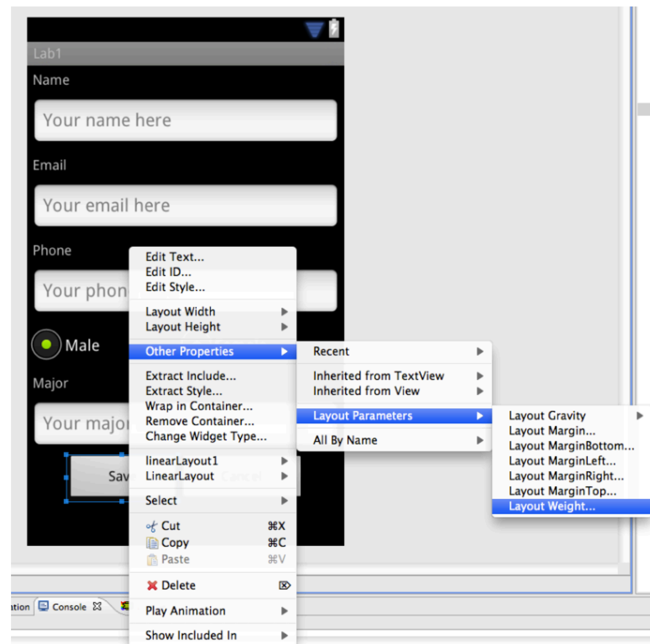

Figure 3 Right Click on a view to change properties: in this case the "layout weight"

Once you have completed the layout shown in Figure 1 you can "run" your code. The java code will render the layout to the screen. You can write to (e.g., enter your name) or click (e.g., click save) any view on the screen – alas, nothing will happen, that is, if you entered you profile details and click save, nothing will be saved – there is no code set up to deal with the user's event – e.g., data entry and click save.

We need some event handling code – next section.

Event handling

Once the user clicks the "save" button we need to store all the information input by the user from all the view objects to persistent storage – we will discuss how we store context in the next section. But first we need an event handler or call back to deal with the user's action once they click save or cancel. Assume for now the user hit the save button. Let's discuss that how we do that.

The code set up for creating an event handler is shown in Figure 4. Remember that all initial code should go in the `OnCreate()` event. We create a handler to handle the event when either the save or cancel buttons are clicked – take a close look at the code snippet in Figure 4.

```
30    /** Called when the activity is first created. */
31    @Override
32    public void onCreate(Bundle savedInstanceState) {
33        super.onCreate(savedInstanceState);
34        setContentView(R.layout.main);
35
36        Button btn = ((Button) findViewById(R.id.buttonSave));
37
38        // EVENT HANDLING PART: If Save button hit create handler to save the data
39
40        View.OnClickListener myListener = new View.OnClickListener() {
41                @Override
42                public void onClick(View v) {
43                    saveProifleFromScreenToSharedPreferences();
44                    Toast.makeText(getApplicationContext(),
45                            "Profile saved.", Toast.LENGTH_SHORT).show();
46                    // Close the activity
47                    finish();
48                }
49            };
50
51        btn.setOnClickListener(myListener);
52
53        loadProfileFromSharedPreferencesToScreen();
54
55    }
```

Figure 4 Code snippet for event handling

First we get a reference to the save button object using `findViewById()` using the id of the button that we specified in the xml when we created the widget. We reference the object using `R.id.buttonSave`.

After that we call the button's `setOnClickListener` method, which takes a `View.OnClickListener` object as the initialization argument; that is, we create a new temporary `View.OnClickListener` on the fly by just calling "new". Next, we need to override the default `onClick` function so we can call our new method (we insert our own code to execute when the event fires); that is, `saveProifleFromScreenToSharedPreferences()` – we need to implement this method to save all the user input (see next section).

We also use a `Toast` to inform the user what is going on "Profiled saved". Toast is a way to "flash" a feed back message to the user – it floats over the screen for a second or two (defined by `LENGTH_SHORT`) and then disappears. Finally, the method `finish()` is called to exist the handler.

The onCreate() method also calls `loadProfileFromSharedPreferencesToScreen()`, which is used to reload the stored profile (assuming the application has run before and the user input and hit the save button). But first let's consider when the user clicks save and we need to store the input data in the views; the work will be done by the code you write for the `saveProifleFromScreenToSharedPreferences()` method – discussed next.

Shared Preference

Android provides handy mechanisms for storing user data; checkout the note on "using shared preference" for background information:

http://developer.android.com/guide/topics/data/data-storage.html#pref

Shared preference is a simple way to save the user profile for this lab. The code in the goggle notes is similar to the code you will need to write to store and restore the user's profile data. Shared preference is a key-value method for saving small amounts of program content – other techniques such as SQLite can be used for more heavy duty storage needs; we will discuss SQLite in a future lab.

The code snippet shown in Figure 5 is from the solution and while not trying to give everything away shows how shared preference is used to get the users name (e.g., Andrew T. Campbell) from the screen and store it initially in an `editor`. Once all data is captured in the editor it needs to be committed to be stored. Here is the snippet `saveProifleFromScreenToSharedPreferences()`

```
68
69        Log.d(TAG, "entered saveProifleFromScreenToSharedPreferences");
70
71        SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
72
73        SharedPreferences.Editor editor = prefs.edit();
74
75        editor.putString(PROFILE_NAME, ((EditText) findViewById(R.id.editTextName)).getText().toString());
76
```

Figure 5 Code snippet for saving the profile

The SharedPreferences class in the snippet allows you to save and store primitive data, e.g., strings this case. Note, that data will persist. To store data the code uses `edit()` to get a `SharedPreferences.Editor`. Then adds the name the user has input using `putString()`. Finally, you need to commit the new values using `commit()` method – not shown in the snippet.

We also need to implement the code to restore any stored data. Here is a snippet from `loadProfileFromSharedPreferencesToScreen()`.To read values, use `SharedPreferences` methods `getString()`. I have explained most of the major design and coding issues for the assignment. You will have to fill in some gaps.

```
99
100         SharedPreferences prefs = getSharedPreferences(PREFS_NAME, MODE_PRIVATE);
101
102         // hints are not text stored in object
103
104         prefItemStr =  prefs.getString(PROFILE_NAME, "");
105         if (prefItemStr.length() > 0) {
106             ((EditText) findViewById(R.id.editTextName)).setText(prefItemStr);
107         }
108
```

Figure 6 Code snippet for restoring data

Restarting your App

Once you have saved your profile you should be able to simple start the app again and it should display your saved profile as shown in Figure 7.

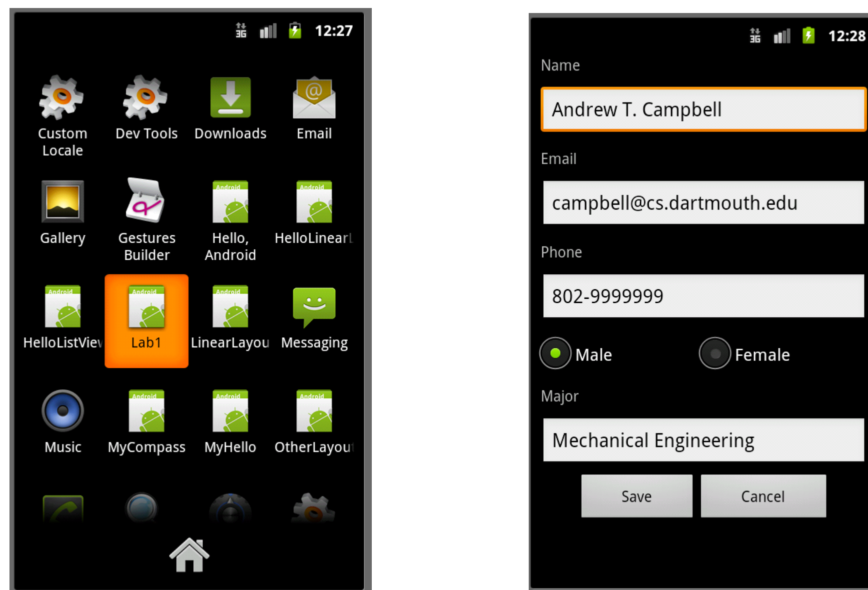

Figure 7 Once you save the profile the app with exit. Start the app again. It displays the saved profile

Tips

Style: Always code defensively; for example, the first time your code runs there is no profile saved. How do you code for this event?

Layout: You can set different keyboard layouts for numerical and text input so that for instance the code presents a right keyboard type for the right input. Figure 2(b) shows the keyboard for email entry.

You should remove the title bar.

Check cancel works too.

Debugging: You can use `Log.d()` in android and TAG that you can display in the LogCat. Checkout

http://developer.android.com/reference/android/util/Log.html

When you run the debugger and filter on TAG you can see all your Log.d() print outs. It allows you to print objects out and see the control flow through the code. The text book has a good section on using `Log.()` and setting filters on TAGs. Check it out. Note, it will take a little time – and many different bugs – to feel comfortable with debugging Android.

If your program crashed, don't panic. Look at the system log in the LogCat, it will print details of the function and line (in red) associated with the crash.