

CSCI 4100 Laboratory 2

More Android

Introduction

In the previous laboratory we explored the basic techniques used to construct Android applications. We weren't too concerned about how our applications looked and there is a major glitch in the applications that we produced. This glitch is illustrated in figures 1 and 2. Figure 1 is a cleaned up version our Hello World application from the previous lab. In this figure I have entered my name and it appears in the greeting.

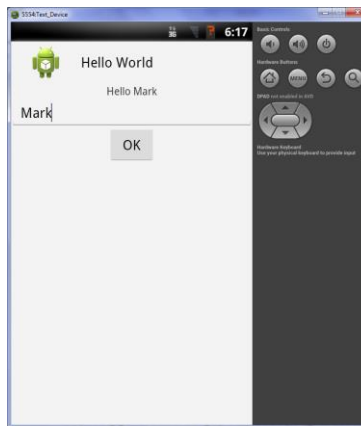


Figure 1 Hello World in vertical orientation

If I now rotate the application by 90 degrees I get the view in figure 2. In the simulator you can use the 7 and 9 keys on the numeric keypad to rotate the device. Your laptop doesn't have a numeric keypad so you can use cntl-F11 and cntl-F12 instead. Note that the greeting is now wrong. It has gone back to the original text in the layout file. How did this happen?

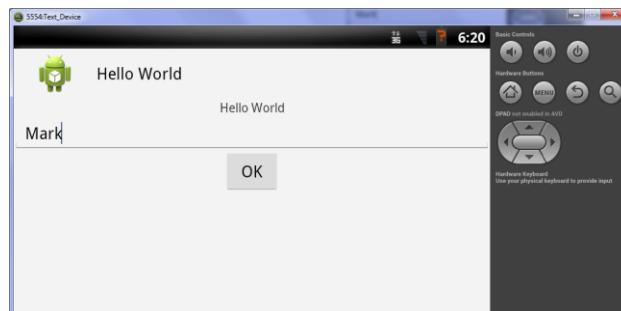


Figure 2 Hello World in the horizontal orientation

When we change the device's orientation Android destroys our activity and then creates it again. Why does it do this? It could be that we have different layout files for the horizontal and vertical orientations, and this gives our application the chance to load the appropriate layout file. But in

the process we have lost the information that is stored in the application; the user's name. In this laboratory we will learn how to fix this problem, as well as some of the basics of UI layout.

Layout

Start by creating a new Android application in the same way as in the first laboratory. We will call this new application Hello World Two. We will start by making this the same as the original Hello World application. You can either follow the steps in laboratory one to do this, or you can cut and paste from the original Hello World application. We now want to start improving the layout of this application. We will start with the textView1 widget. Select this widget in the hierarchy view in the right pane and then expand its layout parameters as shown in figure 3. Note that this widget has a fixed width of 157dp, regardless of its content. The first thing we need to do is change this property to wrap_content, which we can select from the button on the right of the value. The widget will now be wide enough to hold any content that we type (within reason). But, the text is still on the left side of the screen. We can change this by changing the Gravity property. Again press this button on the right of the value and select center from the list of choices. This will center the greeting at the top of the screen.

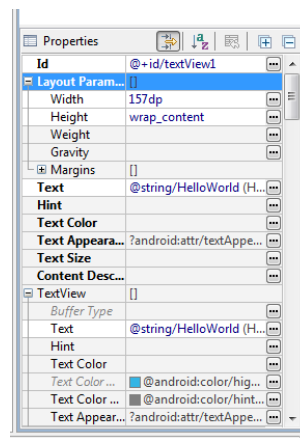


Figure 3 textView1 properties

For the button1 widget open its layout parameters property and also set the Gravity to center. This produces a nicer looking layout on the screen. Try running the application now and see how it handles reasonably long names. You can experiment with the different layout parameters and see what they do.

Persistence

Now we will deal with the glitch in our application where the greeting changes when we rotate the device. The main cause of our problem is that Android destroys our activity when we rotate the device so it can recreate the activity with possibly a different layout for the new orientation. Unfortunately, our application has some state that isn't saved when the device is rotated. You notice that the state of the text edit widget is automatically saved for us; we don't lose what we

have typed into it. For widgets like this Android automatically saves the state when the activity is destroyed, but for the classes that we create ourselves this doesn't happen. Note, the same problem would occur if the user answers the phone while our application is running. When the user answers the phone our activity will be destroyed and when we resume the application the content of the text view widget will have changed. Saving state when an activity is destroyed is called persistence, and we need to add it to our application.

Android makes it relatively easy to do persistence. First it provides a callback, `onSaveInstanceState()` that is called when our activity should save its state. This callback is called whenever there is the possibility that our activity will be destroyed. The parameter to this callback is a `Bundle`, which is a data structure that we can use to save the state. You will also notice that `onCreate()` also has a parameter, which is of type `Bundle`. We can use this parameter to retrieve the saved state for the activity. The `Bundle` is known as a key-value store. It is a collection of pairs, where the first member of the pair is a key and the second member is the value to be stored with that key. The key is a text string and it must be unique within the activity. The value can be any of the Java primitive types, and it can be a class if it implements the `Parcelable` interface. The `Bundle` class has a set of `put` procedures that can be used to add values to the bundle, and a set of `get` procedures that can be used to retrieve them. The first parameter to these procedures is the key that the procedure is operating on. For example, `getString(key)` will return the `String` value associated with the key, and `putString(key, string)` will store a new string value associated with the key.

We now know everything that we need to know to make our application persistent. What is the state that we need to save? It's the user's name that we enter through the edit text widget and then display in the text view widget. We will store this state in a class variable called `name`, and we will have a second variable called `nameTag` that we will use as the key for this variable. The declarations of these variables are:

```
private String name = "World";  
private final String nameTag = "nameTag";
```

The `onSaveInstanceState()` callback for our application is shown in figure 4. This method first calls `putString` on the `Bundle` parameter to save our state variable and then calls `onSaveInstanceState()` on the super class. This should be called after we have saved the state for the current class, and must always be called, otherwise the automatic persistence in Android might fail. Note that we have made a small change to the `setName` procedure so `name` is no longer a local variable and refers to the class level `name` variable. We need to make this modification so we have a value to save on the `onSaveInstanceState()` callback.

Figure 4 also shows the new version of the `onCreate()` method. This method is called each time that the activity is created. The first time it is called there will be no saved state information, and in this case the value of its parameter will be null. On subsequent calls the value of the parameter won't be null, indicating that there is saved state to be retrieved. In the latter case we

use `getString` to retrieve the user's name, retrieve the text view widget and then set the text in the text view widget to the greeting with the user's name. Note that we have to do this after we have called `setContentView`, otherwise there is no text view widget to retrieve.

```
public class MainActivity extends Activity {

    private String name = "World";
    private final String nameTag = "nameTag";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        if(savedInstanceState != null) {
            name = savedInstanceState.getString(nameTag);
            TextView label = (TextView) findViewById(R.id.textView1);
            label.setText("Hello "+name);
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        outState.putString(nameTag, name);
        super.onSaveInstanceState(outState);
    }

    public void setName(View view) {
        EditText nameText = (EditText) findViewById(R.id.editText1);
        TextView label = (TextView) findViewById(R.id.textView1);
        name = nameText.getText().toString();
        label.setText("Hello "+name);
    }
}
```

Figure 4 Modified Java code for persistence

Make the changes to your application code and check to see if it behaves currently when you rotate the device.

Multiple Activities

Android is based on dividing applications into a number of semi-independent components. So far the only component that we have seen is the activity, but we will see more later in the course. The idea is that these components should be developed separately with the aim of reusing them, either in the same application or in other applications. Android activities can even invoke activities in other applications. We will now construct an application that has two activities, so you can see how to add an activity to an application and pass control between activities. This will be yet another hello world application, but to break up the boredom we will call it hello UOIT. Figure 5 shows the activity that our application starts with, and figure 6 shows the second

activity. In the first activity the user enters his or her name and presses the OK button. This transfers control to the second activity where a customized greeting is displayed, along with a very bad UOIT logo. I grabbed the logo from the UOIT web page and by the time Android had processed it, it looked pretty bad. But, we don't need to worry about that for the current example.

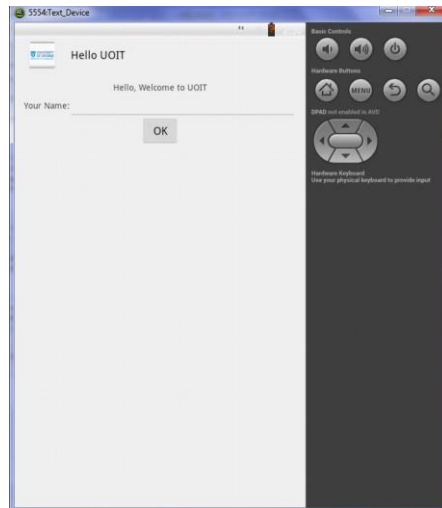


Figure 5 Hello UOIT first activity

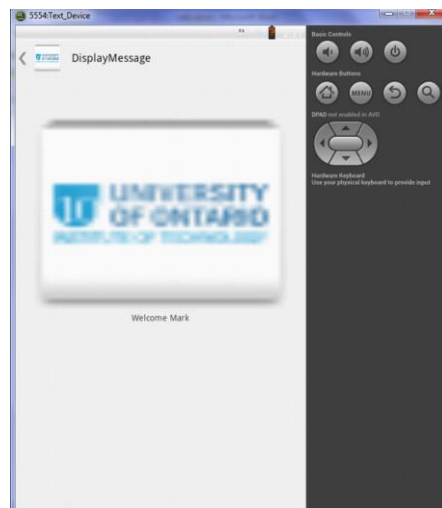


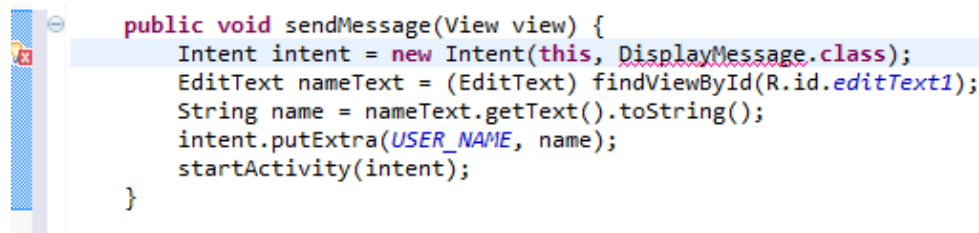
Figure 6 Hello UOIT second activity

We start this example in basically the same way as the previous example, except in this case we call out application Hello UOIT. Follow the usual steps to create a blank activity. We will start by designing the UI, which will be based on a vertical linear layout. In the hierarchy pane on the right select the relative layout, go to the refactor menu, select Android and then Change Layout. From the dialogue that appears select LinearLayout (Vertical) in the drop down menu, this will change the relative layout to a linear layout. Now change the text view widget and update the message it displays. You can either create a new string for this or edit the existing string used by

the widget (if you do this you need to save both the strings.xml and the layout file to get the message to update in the UI editor).

The second line of the activity uses a horizontal linear layout to hold a text view and an edit text widgets. Start by dragging a `LinearLayout(Horizontal)` onto the UI layout and then the two widgets that should be inside it. Next change the message displayed by the text view widget (again update the strings.xml file). Finally drag a button to a location below the horizontal linear layout and change its text to OK. For the button set its On Click property to `sendMessage`, the name of a method that we will produce next.

Now we turn our attention to the Java code and the `sendMessage` method. This method is called when the user presses the button and it will transfer control to our second activity. Android uses intents to transfer control between activities. An intent can specify a particular class to receive the intent, or it can use a more general condition and let Android search for an activity that matches the criterion. We will use the first approach, since we know the name of the class that implements the second activity. Intents can also carry data with them; in this case it will be the user's name which is displayed in the second intent. The Java code for the `sendMessage` method is shown in figure 7.

A screenshot of an IDE showing the implementation of the `sendMessage` method. The code is as follows:

```
public void sendMessage(View view) {  
    Intent intent = new Intent(this, DisplayMessage.class);  
    EditText nameText = (EditText) findViewById(R.id.editText1);  
    String name = nameText.getText().toString();  
    intent.putExtra(USER_NAME, name);  
    startActivity(intent);  
}
```

Figure 7 `sendMessage` method

At this point there is an error in this procedure since we haven't defined the class that we are transferring control to. The first statement in this method creates a new intent with the current class as the context and the `DisplayMessage` class as the target of the intent (we will define this class soon). The next two lines extract the user's name for the edit text widget using the same technique that we have seen in previous examples. The `putExtra()` procedure allows us to add data to our intent using the same key-value pair technique that we saw in persistence. The key is a text string which we have defined earlier in our class in the following way:

```
public static final String USER_NAME = "user name";
```

Finally we call the `startActivity` procedure to transfer control to the second activity. This completes our first activity.

Now we need to create the second activity. To do this click the new button on the task bar and instead of creating a new Android Application Project create a new Android Activity. Press the next button twice until you get to the dialogue shown in figure 8. There are some default values in this dialogue, but some of them aren't correct. Start by changing the activity name to

DisplayMessage, which is the name that we used for the new activity class. This removes all of the error messages. We would like our new activity to take part in navigation, so we need to assign it a parent. The parent is the name of our first activity, which we won't be able to find using the browse button. Instead we need to type it in by hand (note this is the name of the activity and not the class). In my case I used: `ca.uoit.MarkGreen.hellouoit.MainActivity`. If you get this wrong you will have a runtime error and will need to edit the manifest file to correct it. The completed dialogue is shown in figure 9. You can press the Finish button at this point.

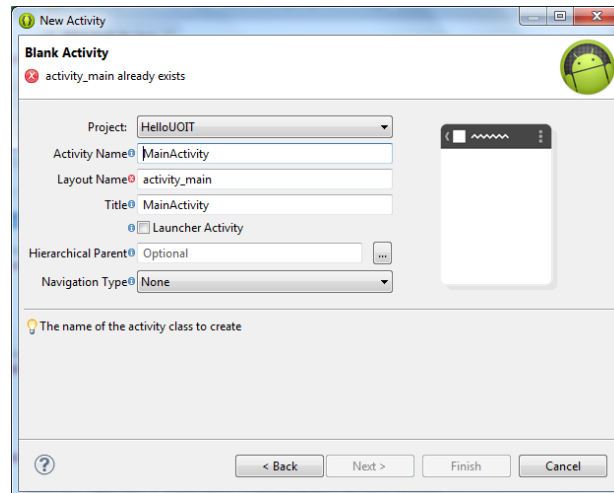


Figure 8 New activity dialogue

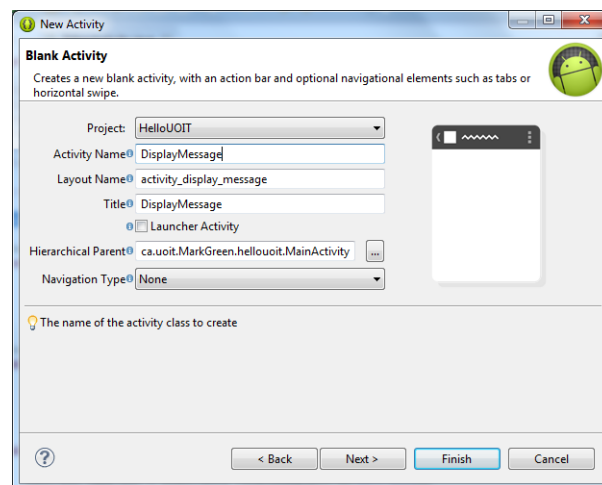


Figure 9 Completed new activity dialogue

We will start by designing the UI for the new activity. Again we have an initial UI layout that we need to modify. Remove the existing text view widget and then refactor the relative layout to be a vertical linear layout. From Images & Media in the pane to the left of the UI layout drag a new image view widget onto the UI layout. A dialogue will pop up asking for an icon for the image view widget. By default you will get the launcher icon. In this case press the Create New icon button and you will get the dialogue in figure 10. When you create a new icon you can either use the default name, which is the launcher icon, or you can enter a new name. By

accident I used the launcher icon name and was happy with the result. Press the Next button to go to the dialogue in figure 11.

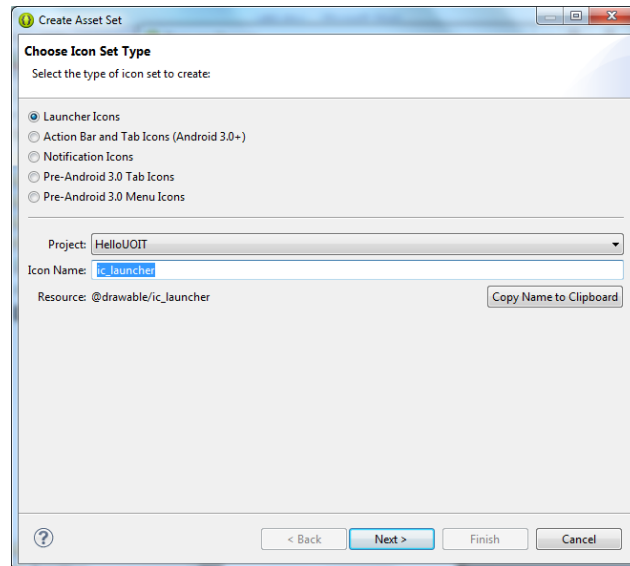


Figure 10 First step in creating a new icon

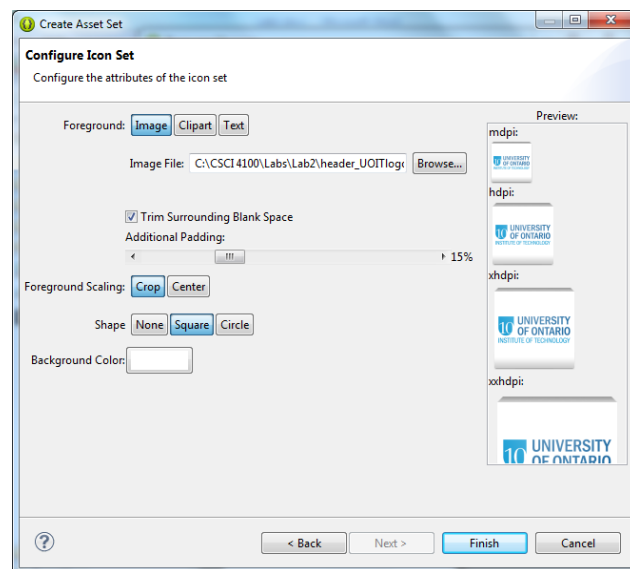


Figure 11 Second step in icon creation

In this case I used an image for the foreground; the logo I copied from the UOIT webpage (this image is available in the lab 2 folder on Blackboard). Then I changed the background to white (the default red background doesn't look very good with this image). You can now press the finish button. You will notice that the resulting icon is relatively small; we'll fix this in a few minutes. Now drag a text view widget below the image view widget, center it and change its text to welcome.

Now it's time to fix the layout problem. Open the Layout Parameters for the image view widget and for the Width property set it to fill_parent and the set the Weight property to 1. Open the Layout Parameters

for the text view widget and set its Weight property to 1 as well. You will now get the layout shown in figure 6. The Weight property specifies the widget's priority when it comes to allocating its screen space within its parent. When we set the Weight for both of the widgets to 1, they evenly divide the screen space, which gives us a reasonable layout. The layout is now done and it's time to write the Java code.

Open the DisplayMessage.java file and you will see that a lot more code has been provided for us. Most of this code handles navigation between activities and we don't need to worry about it. We just need to make a few changes to the onCreate() method to retrieve the intent that created it and the user's name so we can modify the message in the text view widget. The modified onCreate() code is shown in figure 12.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_display_message);
    // Show the Up button in the action bar.
    setupActionBar();
    Intent intent = getIntent();
    String name = intent.getStringExtra(MainActivity.USER_NAME);
    TextView message = (TextView) findViewById(R.id.textView1);
    message.setText("Welcome " + name);
}
```

Figure 12 Java code for onCreate

The second part of the method retrieves the information passed with the intent. The getIntent() procedure returns the intent that created the activity. We can then use its getStringExtra() method to retrieve the user's name from the intent and use this to change the message in the text view widget.

Once you have finished the layout and entered the code try running the application. If you press the arrow icon in the upper left corner of the second activity it should take you back to the first activity. If this doesn't work check your manifest file to ensure that the parent of the second activity has been set correctly. Figure 13 shows how it should look. Note that the circled names must all be the same otherwise navigation won't work.

```
> <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
        android:name="ca.uoit.MarkGreen.hellooit.MainActivity"
        android:label="@string/app_name"
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity
        android:name="ca.uoit.MarkGreen.hellooit.DisplayMessage"
        android:label="@string/title_activity_display_message"
        android:parentActivityName="ca.uoit.MarkGreen.hellooit.MainActivity"
        <meta-data
            android:name="android.support.design.navigation"
            android:value="ca.uoit.MarkGreen.hellooit.MainActivity" />
    </activity>
</application>
```

Figure 13 The AndroidManifest.xml file

Laboratory Report

In the Hello UOIT application when you navigate back to the first activity from the second activity you will notice that the user's name is no longer displayed in the edit text widget. After the first activity transfers control to the second activity it is destroyed, and then it is recreated when we return to it. Use persistence to store the user's name and return it to the edit text widget in the first activity. As an added feature change the message on the first line from "Hello, Welcome to UOIT" to "Welcome back to UOIT" when the user returns to the first activity. Show your application to the TA, or email it to him by the end of the day.