13. Android Activity State Changes by Example

The previous chapters have discussed in some detail the different states and lifecycles of the activities that comprise an Android application. In this chapter, we will put the theory of handling activity state changes into practice through the creation of an example application. The purpose of this example application is to provide a real world demonstration of an activity as it passes through a variety of different states within the Android runtime.

In the next chapter, entitled *"Saving and Restoring the State of an Android Activity"*, the example project constructed in this chapter will be extended to demonstrate the saving and restoration of dynamic activity state.

13.1 Creating the State Change Example Project

The first step in this exercise is to create the new project. Begin by launching Android Studio and, if necessary, closing any currently open projects using the *File -> Close Project* menu option so that the Welcome screen appears.

Select the *Start a new Android Studio project* quick start option from the welcome screen and, within the resulting new project dialog, enter *StateChange* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of a Basic Activity named *StateChangeActivity* with a corresponding layout named *activity_state_change.*

Upon completion of the project creation process, the *StateChange* project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window with the *content_state_change.xml* layout file pre-loaded into the Layout Editor as illustrated in Figure 13-1.

The next action to take involves the design of the content area of the user interface for the activity. This is stored in a file named *content_state_change.xml* which should already be loaded into the Layout Editor tool. If it is not, navigate to it in the project tool window where it can

be found in the *app* -> *res* -> *layout* folder. Once located, double-clicking on the file will load it into the Android Studio Layout Editor tool.

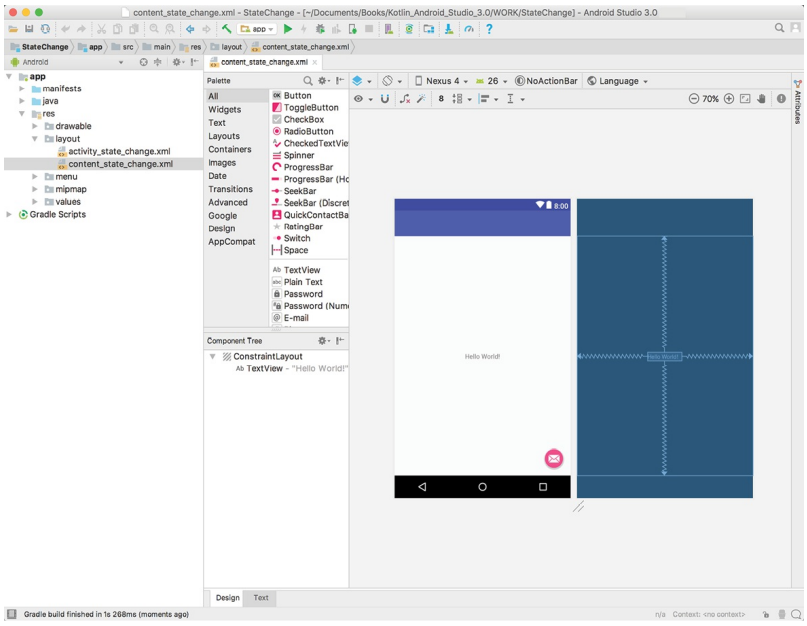

Figure 13-1

13.2 Designing the User Interface

With the user interface layout loaded into the Layout Editor tool, it is now time to design the user interface for the example application. Instead of the "Hello world!" TextView currently present in the user interface design, the activity actually requires an EditText view. Select the TextView object in the Layout Editor canvas and press the Delete key on the keyboard to remove it from the design.

From the Palette located on the left side of the Layout Editor, select the *Text* category and, from the list of text components, click and drag a *Plain Text* component over to the visual representation of the device screen. Move the component to the center of the display so that the center guidelines appear and drop it into place so that the layout resembles that of Figure 13-2.

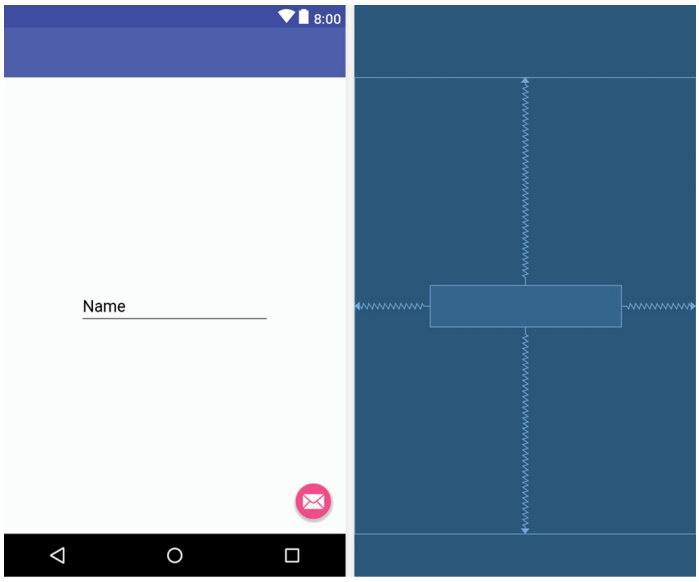

Figure 13-2

When using the EditText widget it is necessary to specify an *input type* for the view. This simply defines the type of text or data that will be entered by the user. For example, if the input type is set to *Phone*, the user will be restricted to entering numerical digits into the view. Alternatively, if the input type is set to *TextCapCharacters*, the input will default to upper case characters. Input type settings may also be combined.

For the purposes of this example, we will set the input type to support general text input. To do so, select the EditText widget in the layout and locate the *inputType* entry within the Attributes tool window. Click on the current setting to open the list of options and, within the list, switch off *textPersonName* and enable *text* before clicking on the OK button.

By default the EditText is displaying text which reads "Name". Remaining within the Attributes panel, delete this from the *text* property field so that the view is blank within the layout.

13.3 Overriding the Activity Lifecycle Methods

At this point, the project contains a single activity named *StateChangeActivity*, which is derived from the Android *AppCompatActivity* class. The source code for this activity is contained within the *StateChangeActivity.java* file which should already be open in an editor session and represented by a tab in the editor tab bar. In the event that the file is no longer open, navigate to it in the Project tool window panel (*app -> java*

-> *com.ebookfrenzy.statechange* -> *StateChangeActivity*) and double-click on it to load the file into the editor. Once loaded the code should read as follows:

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class StateChangeActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_state_change);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab =
              (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                     Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
    }

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

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

So far the only lifecycle method overridden by the activity is the *onCreate()* method which has been implemented to call the super class instance of the method before setting up the user interface for the activity. We will now modify this method so that it outputs a diagnostic message in the Android Studio Logcat panel each time it executes. For this, we will use the *Log* class, which requires that we import *android.util.Log* and declare a tag that will enable us to filter these messages in the log output:

```
package com.ebookfrenzy.statechange;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.util.Log;

public class StateChangeActivity extends AppCompatActivity {

    private static final String TAG = "StateChange";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_state_change);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        FloatingActionButton fab = (FloatingActionButton)
findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                     Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });

        Log.i(TAG, "onCreate");
    }
.
.
}
```

The next task is to override some more methods, with each one containing a corresponding log call. These override methods may be added manually or generated using the *Alt-Insert* keyboard shortcut as outlined in the chapter entitled *"The Basics of the Android Studio Code Editor"*. Note that the Log calls will still need to be added manually if the methods are being auto-generated:

```
@Override
protected void onStart() {
    super.onStart();
    Log.i(TAG, "onStart");
}

@Override
protected void onResume() {
    super.onResume();
    Log.i(TAG, "onResume");
}

@Override
protected void onPause() {
    super.onPause();
```

```
    Log.i(TAG, "onPause");
}

@Override
protected void onStop() {
    super.onStop();
    Log.i(TAG, "onStop");
}

@Override
protected void onRestart() {
    super.onRestart();
    Log.i(TAG, "onRestart");
}

@Override
protected void onDestroy() {
    super.onDestroy();
    Log.i(TAG, "onDestroy");
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.i(TAG, "onSaveInstanceState");
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    Log.i(TAG, "onRestoreInstanceState");
}
```

13.4 Filtering the Logcat Panel

The purpose of the code added to the overridden methods in *StateChangeActivity.java* is to output logging information to the *Logcat* tool window. This output can be configured to display all events relating to the device or emulator session, or restricted to those events that relate to the currently selected app. The output can also be further restricted to only those log events that match a specified filter.

Display the Logcat tool window and click on the filter menu (marked as B in

Figure 13-3) to review the available options. When this menu is set to *Show only selected application*, only those messages relating to the app selected in the menu marked as A will be displayed in the Logcat panel. Choosing *No Filter*, on the other hand, will display all the messages generated by the device or emulator.

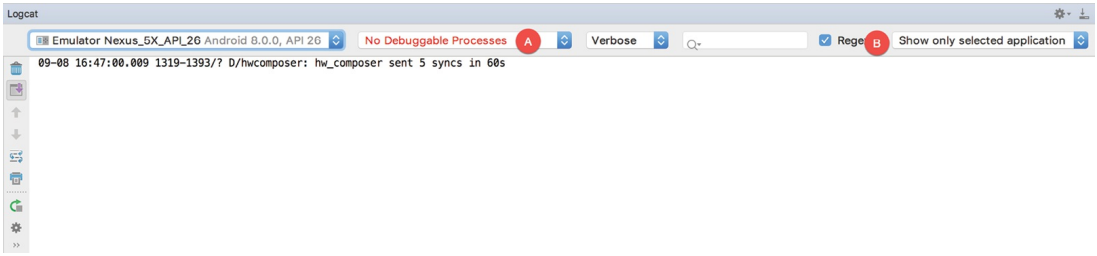

Figure 13-3

Before running the application, it is worth demonstrating the creation of a filter which, when selected, will further restrict the log output to ensure that only those log messages containing the tag declared in our activity are displayed.

From the filter menu (B), select the *Edit Filter Configuration* menu option. In the *Create New Logcat Filter* dialog (Figure 13-4), name the filter *Lifecycle* and, in the *Log Tag* field, enter the Tag value declared in *StateChangeActivity.java* (in the above code example this was *StateChange*).

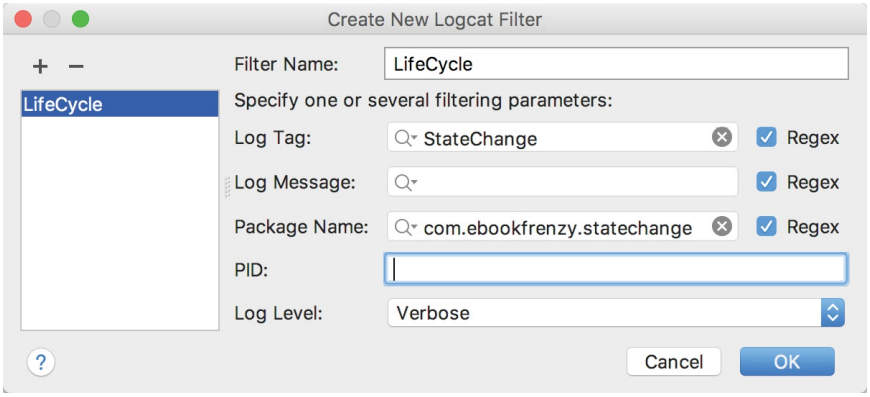

Figure 13-4

Enter the package identifier in the *Package Name* field (clicking on the search icon in the text field will drop down a menu from which the package name may be selected) and, when the changes are complete, click on the *OK* button to create the filter and dismiss the dialog. Instead of listing *No Filters,* the newly created filter should now be selected in the Logcat tool window.

13.5 Running the Application

For optimal results, the application should be run on a physical Android device, details of which can be found in the chapter entitled *"Testing Android Studio Apps on a Physical Android Device"*. With the device configured and connected to the development computer, click on the run button represented by a green triangle located in the Android Studio toolbar as shown in Figure 13-5 below, select the *Run -> Run…* menu option or use the Shift+F10 keyboard shortcut:

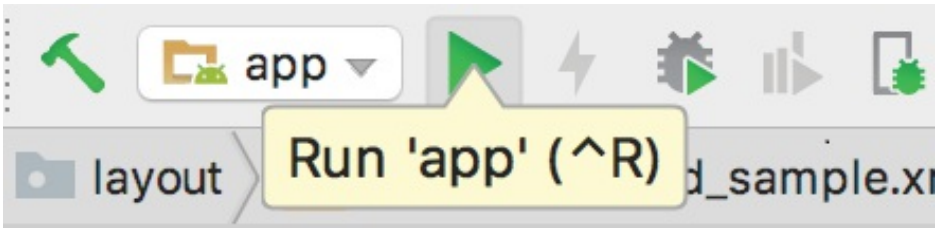

Figure 13-5

Select the physical Android device from the *Choose Device* dialog if it appears (assuming that you have not already configured it to be the default target). After Android Studio has built the application and installed it on the device it should start up and be running in the foreground.

A review of the Logcat panel should indicate which methods have so far been triggered (taking care to ensure that the *Lifecycle* filter created in the preceding section is selected to filter out log events that are not currently of interest to us):

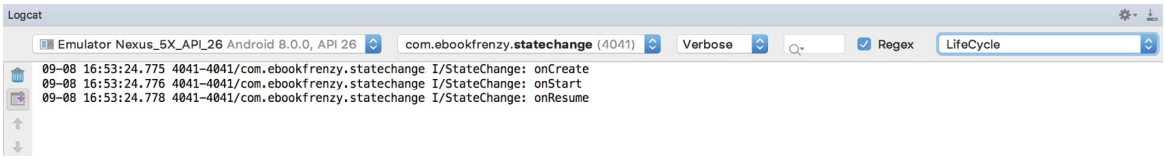

Figure 13-6

13.6 Experimenting with the Activity

With the diagnostics working, it is now time to exercise the application with a view to gaining an understanding of the activity lifecycle state changes. To begin with, consider the initial sequence of log events in the Logcat panel:

```
onCreate
onStart
onResume
```

Clearly, the initial state changes are exactly as outlined in *"Understanding*

Android Application and Activity Lifecycles". Note, however, that a call was not made to *onRestoreInstanceState()* since the Android runtime detected that there was no state to restore in this situation.

Tap on the Home icon in the bottom status bar on the device display and note the sequence of method calls reported in the log as follows:

```
onPause
onSaveInstanceState
onStop
```

In this case, the runtime has noticed that the activity is no longer in the foreground, is not visible to the user and has stopped the activity, but not without providing an opportunity for the activity to save the dynamic state. Depending on whether the runtime ultimately destroyed the activity or simply restarted it, the activity will either be notified it has been restarted via a call to *onRestart()* or will go through the creation sequence again when the user returns to the activity.

As outlined in *"Understanding Android Application and Activity Lifecycles"*, the destruction and recreation of an activity can be triggered by making a configuration change to the device, such as rotating from portrait to landscape. To see this in action, simply rotate the device while the *StateChange* application is in the foreground. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. The resulting sequence of method calls in the log should read as follows:

```
onPause
onSaveInstanceState
onStop
onDestroy
onCreate
onStart
onRestoreInstanceState
onResume
```

Clearly, the runtime system has given the activity an opportunity to save state before being destroyed and restarted.

13.7 Summary

The old adage that a picture is worth a thousand words holds just as true for examples when learning a new programming paradigm. In this chapter, we

have created an example Android application for the purpose of demonstrating the different lifecycle states through which an activity is likely to pass. In the course of developing the project in this chapter, we also looked at a mechanism for generating diagnostic logging information from within an activity.

In the next chapter, we will extend the *StateChange* example project to demonstrate how to save and restore an activity's dynamic state.