# 32. Animating User Interfaces with the Android Transitions Framework

The Android Transitions framework was introduced as part of the Android 4.4 KitKat release and is designed to make it easy for you, as an Android developer, to add animation effects to the views that make up the screens of your applications. As will be outlined in both this and subsequent chapters, animated effects such as making the views in a user interface gently fade in and out of sight and glide smoothly to new positions on the screen can be implemented with just a few simple lines of code when using the Transitions framework in Android Studio.

## 32.1 Introducing Android Transitions and Scenes

Transitions allow the changes made to the layout and appearance of the views in a user interface to be animated during application runtime. While there are a number of different ways to implement Transitions from within application code, perhaps the most powerful mechanism involves the use of *Scenes*. A scene represents either the entire layout of a user interface screen, or a subset of the layout (represented by a ViewGroup).

To implement transitions using this approach, scenes are defined that reflect the two different user interface states (these can be thought of as the "before" and "after" scenes). One scene, for example, might consist of a TextEdit, Button and TextView positioned near the top of the screen. The second scene might remove the Button view and move the remaining TextEdit and TextView objects to the bottom of the screen to make room for the introduction of a MapView instance. Using the transition framework, the changes between these two scenes can be animated so that the Button fades from view, the TextEdit and TextView slide to the new locations and the map gently fades into view.

Scenes can be created in code from ViewGroups, or implemented in layout resource files that are loaded into Scene instances at application runtime.

Transitions can also be implemented dynamically from within application code. Using this approach, scenes are created by referencing collections of user interface views in the form of ViewGroups with transitions then being performed on those elements using the TransitionManager class, which provides a range of methods for triggering and managing the transitions between scenes.

Perhaps the simplest form of transition involves the use of the *beginDelayedTransition()* method of the TransitionManager class. When called and passed the ViewGroup representing a scene, any subsequent changes to any views within that scene (such as moving, resizing, adding or deleting views) will be animated by the Transition framework.

The actual animation is handled by the Transition framework via instances of the *Transition* class. Transition instances are responsible for detecting changes to the size, position and visibility of the views within a scene and animating those changes accordingly.

By default, transitions will be animated using a set of criteria defined by the AutoTransition class. Custom transitions can be created either via settings in XML transition files or directly within code. Multiple transitions can be combined together in a TransitionSet and configured to be performed either in parallel or sequentially.

## 32.2 Using Interpolators with Transitions

The Transitions framework makes extensive use of the Android Animation framework to implement animation effects. This fact is largely incidental when using transitions since most of this work happens behind the scenes, thereby shielding the developer from some of the complexities of the Animation framework. One area where some knowledge of the Animation framework is beneficial when using Transitions, however, involves the concept of interpolators.

Interpolators are a feature of the Android Animation framework that allow animations to be modified in a number of pre-defined ways. At present the Animation framework provides the following interpolators, all of which are available for use in customizing transitions:

- **AccelerateDecelerateInterpolator** – By default, animation is performed at a constant rate. The AccelerateDecelerateInterpolator can be used to cause the animation to begin slowly and then speed up in the middle before slowing down towards the end of the sequence.
- **AccelerateInterpolator** – As the name suggests, the AccelerateInterpolator begins the animation slowly and accelerates at a specified rate with no deceleration at the end.
- **AnticipateInterpolator** – The AnticipateInterpolator provides an effect similar to that of a sling shot. The animated view moves in the opposite direction to the configured animation for a short distance before being flung forward in the correct direction. The amount of backward force can be controlled through the specification of a tension value.
- **AnticipateOvershootInterpolator** – Combines the effect provided by the AnticipateInterpolator with the animated object overshooting and then returning to the destination position on the screen.
- **BounceInterpolator** – Causes the animated view to bounce on arrival at its destination position.
- **CycleInterpolator** – Configures the animation to be repeated a specified number of times.
- **DecelerateInterpolator** – The DecelerateInterpolator causes the animation to begin quickly and then decelerate by a specified factor as it nears the end.
- **LinearInterpolator** – Used to specify that the animation is to be performed at a constant rate.
- **OvershootInterpolator** – Causes the animated view to overshoot the specified destination position before returning. The overshoot can be configured by specifying a tension value.

As will be demonstrated in this and later chapters, interpolators can be specified both in code and XML files.

## 32.3 Working with Scene Transitions

Scenes can be represented by the content of an Android Studio XML layout file. The following XML, for example, could be used to represent a scene consisting of three button views within a RelativeLayout parent:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/RelativeLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
```

```
        android:onClick="goToScene2"
        android:text="@string/one_string" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:onClick="goToScene1"
        android:text="@string/two_string" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/three_string" />

</RelativeLayout>
```

Assuming that the above layout resides in a file named *scene1_layout.xml* located in the *res/layout* folder of the project, the layout can be loaded into a scene using the *getSceneForLayout()* method of the Scene class. For example:

```
Scene scene1 = Scene.getSceneForLayout(rootContainer,
                R.layout.scene1_layout, this);
```

Note that the method call requires a reference to the root container. This is the view at the top of the view hierarchy in which the scene is to be displayed.

To display a scene to the user without any transition animation, the *enter()* method is called on the scene instance:

```
scene1.enter();
```

Transitions between two scenes using the default AutoTransition class can be triggered using the *go()* method of the TransitionManager class:

```
TransitionManager.go(scene2);
```

Scene instances can be created easily in code by bundling the view elements into one or more ViewGroups and then creating a scene from those groups. For example:

```
Scene scene1 = Scene(viewGroup1);
Scene scene2 = Scene(viewGroup2, viewGroup3);
```

## 32.4 Custom Transitions and TransitionSets in Code

The examples outlined so far in this chapter have used the default transition settings in which resizing, fading and motion are animated using pre-configured behavior. These can be modified by creating custom transitions which are then referenced during the transition process. Animations are categorized as either *change bounds* (relating to changes in the position and size of a view) and *fade* (relating to the visibility or otherwise of a view).

A single Transition can be created as follows:

```
Transition myChangeBounds = new ChangeBounds();
```

This new transition can then be used when performing a transition:

```
TransitionManager.go(scene2, myChangeBounds);
```

Multiple transitions may be bundled together into a TransitionSet instance. The following code, for example, creates a new TransitionSet object consisting of both change bounds and fade transition effects:

```
TransitionSet myTransition = new TransitionSet();
myTransition.addTransition(new ChangeBounds());
myTransition.addTransition(new Fade());
```

Transitions can be configured to target specific views (referenced by view ID). For example, the following code will configure the previous fade transition to target only the view with an ID that matches *myButton1*:

```
TransitionSet myTransition = new TransitionSet();
myTransition.addTransition(new ChangeBounds());
Transition fade = new Fade();
fade.addTarget(R.id.myButton1);
myTransition.addTransition(fade);
```

Additional aspects of the transition may also be customized, such as the duration of the animation. The following code specifies the duration over which the animation is to be performed:

```
Transition changeBounds = new ChangeBounds();
changeBounds.setDuration(2000);
```

As with Transition instances, once a TransitionSet instance has been created, it can be used in a transition via the TransitionManager class. For example:

```
TransitionManager.go(scene1, myTransition);
```

## 32.5 Custom Transitions and TransitionSets in XML

While custom transitions can be implemented in code, it is often easier to do so via XML transition files using the <fade> and <changeBounds> tags together with some additional options. The following XML includes a single changeBounds transition:

```
<?xml version="1.0" encoding="utf-8"?>
<changeBounds/>
```

As with the code based approach to working with transitions, each transition entry in a resource file may be customized. The XML below, for example, configures a duration for a change bounds transition:

```
<changeBounds android:duration="5000" >
```

Multiple transitions may be bundled together using the <transitionSet> element:

```
<?xml version="1.0" encoding="utf-8"?>
<transitionSet
  xmlns:android="http://schemas.android.com/apk/res/android" >

  <fade
    android:duration="2000"
    android:fadingMode="fade_out" />
```

```
    <changeBounds
      android:duration="5000" >

      <targets>
        <target android:targetId="@id/button2" />
      </targets>

    </changeBounds>

    <fade
      android:duration="2000"
      android:fadingMode="fade_in" />
</transitionSet>
```

Transitions contained within an XML resource file should be stored in the *res/transition* folder of the project in which they are being used and must be inflated before being referenced in the code of an application. The following code, for example, inflates the transition resources contained within a file named *transition.xml* and assigns the results to a reference named *myTransition*:

```
Transition myTransition = TransitionInflater.from(this)
    .inflateTransition(R.transition.transition);
```

Once inflated, the new transition can be referenced in the usual way:

```
TransitionManager.go(scene1, myTransition);
```

By default, transition effects within a TransitionSet are performed in parallel. To instruct the Transition framework to perform the animations sequentially, add the appropriate *android:transitionOrdering* property to the transitionSet element of the resource file:

```
<?xml version="1.0" encoding="utf-8"?>

<transitionSet
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:transitionOrdering="sequential">

    <fade
      android:duration="2000"
      android:fadingMode="fade_out" />

    <changeBounds
      android:duration="5000" >
  </changeBounds>
</transitionSet>
```

Change the value from “sequential” to “together” to indicate that the animation sequences are to be performed in parallel.

## 32.6 Working with Interpolators

As previously discussed, interpolators can be used to modify the behavior of a transition in a variety of ways and may be specified either in code or via the settings within a transition XML resource file.

When working in code, new interpolator instances can be created by calling the constructor method of the required interpolator class and, where appropriate, passing through values to further modify the interpolator behavior:

- AccelerateDecelerateInterpolator()
- AccelerateInterpolator(float factor)
- AnticipateInterpolator(float tension)
- AnticipateOvershootInterpolator(float tension)
- BounceInterpolator()
- CycleInterpolator(float cycles)
- DecelerateInterpolator(float factor)
- LinearInterpolator()
- OvershootInterpolator(float tension)

Once created, an interpolator instance can be attached to a transition using the *setInterpolator()* method of the Transition class. The following code, for example, adds a bounce interpolator to a change bounds transition:

```
Transition changeBounds = new ChangeBounds();
changeBounds.setInterpolator(new BounceInterpolator());
```

Similarly, the following code adds an accelerate interpolator to the same transition, specifying an acceleration factor of 1.2:

```
changeBounds.setInterpolator(new AccelerateInterpolator(1.2f));
```

In the case of XML based transition resources, a default interpolator is declared using the following syntax:

```
android:interpolator="@android:anim/<interpolator_element>"
```

In the above syntax, *<interpolator_element>* must be replaced by the resource ID of the corresponding interpolator selected from the following list:

- accelerate_decelerate_interpolator
- accelerate_interpolator
- anticipate_interpolator
- anticipate_overshoot_interpolator
- bounce_interpolator
- cycle_interpolator
- decelerate_interpolator
- linear_interpolator
- overshoot_interpolator

The following XML fragment, for example, adds a bounce interpolator to a change bounds transition contained within a transition set:

```
<?xml version="1.0" encoding="utf-8"?>
<transitionSet
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:transitionOrdering="sequential">

  <changeBounds
    android:interpolator="@android:anim/bounce_interpolator"
    android:duration="2000" />

  <fade
    android:duration="1000"
```

```
        android:fadingMode="fade_in" />
    </transitionSet>
```

This approach to adding interpolators to transitions within XML resources works well when the default behavior of the interpolator is required. The task becomes a little more complex when the default behavior of an interpolator needs to be changed. Take, for example, the cycle interpolator. The purpose of this interpolator is to make an animation or transition repeat a specified number of times. In the absence of a *cycles* attribute setting, the cycle interpolator will perform only one cycle. Unfortunately, there is no way to directly specify the number of cycles (or any other interpolator attribute for that matter) when adding an interpolator using the above technique. Instead, a custom interpolator must be created and then referenced within the transition file.

## 32.7 Creating a Custom Interpolator

A custom interpolator must be declared in a separate XML file and stored within the *res/anim* folder of the project. The name of the XML file will be used by the Android system as the resource ID for the custom interpolator.

Within the custom interpolator XML resource file, the syntax should read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<interpolatorElement
xmlns:android="http://schemas.android.com/apk/res/android"
android:attribute="value" />
```

In the above syntax, *interpolatorElement* must be replaced with the element name of the required interpolator selected from the following list:

- accelerateDecelerateInterpolator
- accelerateInterpolator
- anticipateInterpolator
- anticipateOvershootInterpolator
- bounceInterpolator
- cycleInterpolator
- decelerateInterpolator
- linearInterpolator
- overshootInterpolator

The *attribute* keyword is replaced by the name attribute of the interpolator for which the value is to be changed (for example *tension* to change the tension attribute of an overshoot interpolator). Finally, *value* represents the value to be assigned to the specified attribute. The following XML, for example, contains a custom cycle interpolator configured to cycle 7 times:

```
<?xml version="1.0" encoding="utf-8"?>
<cycleInterpolator
xmlns:android="http://schemas.android.com/apk/res/android"
android:cycles="7" />
```

Assuming that the above XML was stored in a resource file named *my_cycle.xml* stored in the *res/anim* project folder, the custom interpolator could be added to a transition resource file using the following XML syntax:

```
<changeBounds
  xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:duration="5000"
    android:interpolator="@anim/my_cycle" >
```

## 32.8 Using the beginDelayedTransition Method

Perhaps the simplest form of Transition based user interface animation involves the use of the *beginDelayedTransition()* method of the TransitionManager class. This method is passed a reference to the root view of the viewgroup representing the scene for which animation is required. Subsequent changes to the views within that sub view will then be animated using the default transition settings:

```
myLayout = (ViewGroup) findViewById(R.id.myLayout);
TransitionManager.beginDelayedTransition(myLayout);
// Make changes to the scene
```

If behavior other than the default animation behavior is required, simply pass a suitably configured Transition or TransitionSet instance through to the method call:

```
TransitionManager.beginDelayedTransition(myLayout, myTransition);
```

## 32.9 Summary

The Android 4.4 KitKat SDK release introduced the Transition Framework, the purpose of which is to simplify the task of adding animation to the views that make up the user interface of an Android application. With some simple configuration and a few lines of code, animation effects such as movement, visibility and resizing of views can be animated by making use of the Transition framework. A number of different approaches to implementing transitions are available involving a combination of Java code and XML resource files. The animation effects of transitions may also be enhanced through the use of a range of interpolators.

Having covered some of the theory of Transitions in Android, the next two chapters will put this theory into practice by working through some example Android Studio based transition implementations.

# 33. An Android Transition Tutorial using beginDelayedTransition

The previous chapter, entitled *Animating User Interfaces with the Android Transitions Framework*, provided an introduction to the animation of user interfaces using the Android Transitions framework. This chapter uses a tutorial based approach to demonstrate Android transitions in action using the *beginDelayedTransition()* method of the TransitionManager class.

The next chapter will create a more complex example that uses layout files and transition resource files to animate the transition from one scene to another within an application.

## 33.1 Creating the Android Studio TransitionDemo Project

Create a new project in Android Studio, entering *TransitionDemo* 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 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *TransitionDemoActivity* with a layout resource file named *activity_transition_demo*.

## 33.2 Preparing the Project Files

The first example transition animation will be implemented through the use of the *beginDelayedTransition()* method of the TransitionManager class. If Android Studio does not automatically load the file, locate and double-click on the *app -> res -> layout -> activity_transition_demo.xml* file in the Project tool window panel to load it into the Layout Editor tool.

Switch the Layout Editor to Design mode, drag a Button from the Widget section of the Layout Editor palette and position it in the top left-hand corner of the device screen layout. Once positioned, select the button and use the Properties tool window to specify an ID value of *myButton*.

Select the ConstraintLayout entry in the Component Tree tool window and use the Properties window to set the ID to *myLayout*.

## 33.3 Implementing beginDelayedTransition Animation

The objective for the initial phase of this tutorial is to implement a touch handler so that when the user taps on the layout view the button view moves to the lower right-hand corner of the screen.

Open the *TransitionDemoActivity.java* file (located in the Project tool window under *app -> java -> com.ebookfrenzy.transitiondemo*) and modify the *onCreate()* method to implement the onTouch handler:

```
package com.ebookfrenzy.transitiondemo;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintLayout;
import android.support.constraint.ConstraintSet;
import android.view.MotionEvent;
```

```
import android.widget.Button;
import android.view.View;

public class TransitionDemoActivity extends AppCompatActivity {

    ConstraintLayout myLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_transition_demo);

        myLayout = (ConstraintLayout) findViewById(R.id.myLayout);

        myLayout.setOnTouchListener(
                new ConstraintLayout.OnTouchListener() {
                    public boolean onTouch(View v,
                                           MotionEvent m) {
                        handleTouch();
                        return true;
                    }
                }
        );



    }
}
```

The above code simply sets up a touch listener on the ConstraintLayout container and configures it to call a method named *handleTouch()* when a touch is detected. The next task, therefore, is to implement the *handleTouch()* method as follows:

```
public void handleTouch() {
    Button button = (Button) findViewById(R.id.myButton);

    button.setMinimumWidth(500);
    button.setMinimumHeight(350);

    ConstraintSet set = new ConstraintSet();

    set.connect(R.id.myButton, ConstraintSet.BOTTOM,
          ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

    set.connect(R.id.myButton, ConstraintSet.RIGHT,
          ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);

    set.constrainWidth(R.id.myButton, ConstraintSet.WRAP_CONTENT);

    set.applyTo(myLayout);
}
```

This method obtains a reference to the button view in the user interface layout and sets new minimum height and width properties so that the button increases in size.

A ConstraintSet object is then created and configured with constraints that will position the button in the lower right-hand corner of the parent layout. This constraint set is then applied to the layout.

Test the code so far by compiling and running the application. Once launched, touch the background (not the button) and note that the button moves and resizes as illustrated in Figure 33-1:

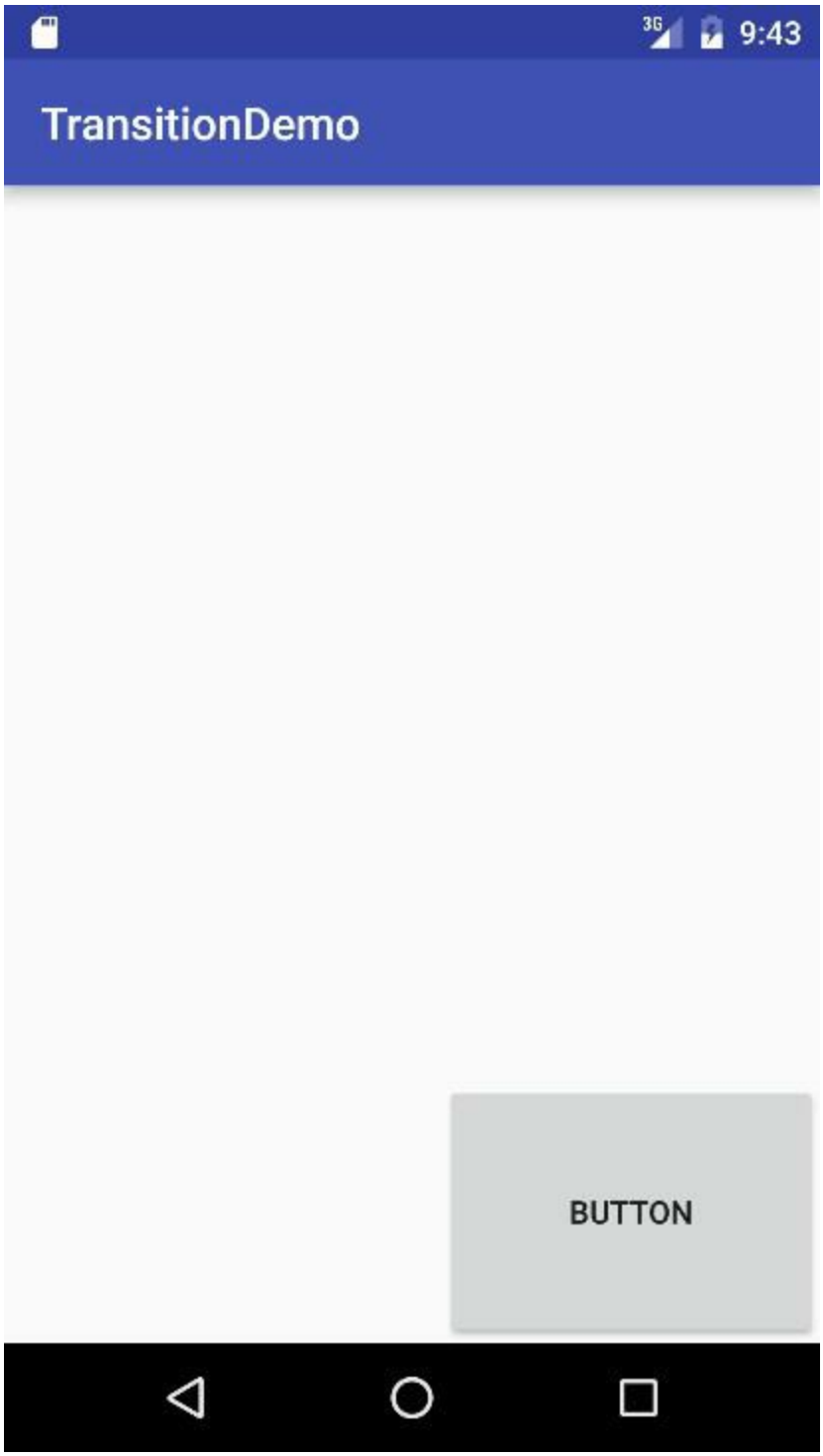


Figure 33-1

Although the layout changes took effect, they did so instantly and without any form of animation. This is where the call to the *beginDelayedTransition()* method of the TransitionManager class comes in. All that is needed to add animation to this layout change is the addition of a single line of code before the layout changes are implemented. Remaining within the *TransitionDemoActivity.java* file, modify the code as follows:

```
package com.ebookfrenzy.transitiondemo;
```

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintLayout;
import android.support.constraint.ConstraintSet;
import android.view.MotionEvent;
import android.view.View;
import android.widget.Button;
import android.transition.TransitionManager;

public class TransitionDemoActivity extends AppCompatActivity {

    ConstraintLayout myLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_transition_demo);

        myLayout = (ConstraintLayout) findViewById(R.id.myLayout);

        myLayout.setOnTouchListener(
                new ConstraintLayout.OnTouchListener() {
                    public boolean onTouch(View v,
                                           MotionEvent m) {
                        handleTouch();
                        return true;
                    }
                }
        );
    }

    public void handleTouch() {
        Button button = (Button) findViewById(R.id.myButton);

        TransitionManager.beginDelayedTransition(myLayout);

        ConstraintSet set = new ConstraintSet();

        button.setMinimumWidth(500);
        button.setMinimumHeight(350);

        set.connect(R.id.myButton, ConstraintSet.BOTTOM,
              ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

        set.connect(R.id.myButton, ConstraintSet.RIGHT,
              ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);

        set.constrainWidth(R.id.myButton, ConstraintSet.WRAP_CONTENT);

        set.applyTo(myLayout);
    }
}
```

.
.

```
    .
}
```

Compile and run the application once again and note that the transition is now animated.

## 33.4 Customizing the Transition

The final task in this example is to modify the changeBounds transition so that it is performed over a longer duration and incorporates a bounce effect when the view reaches its new screen location. This involves the creation of a Transition instance with appropriate duration interpolator settings which is, in turn, passed through as an argument to the *beginDelayedTransition()* method:

```
.
.
import android.transition.ChangeBounds;
import android.transition.Transition;
import android.view.animation.BounceInterpolator;
.
.
public void handleTouch() {
    Button button = (Button) findViewById(R.id.myButton);

    Transition changeBounds = new ChangeBounds();
    changeBounds.setDuration(3000);
    changeBounds.setInterpolator(new BounceInterpolator());

    TransitionManager.beginDelayedTransition(myLayout,
            changeBounds);

    TransitionManager.beginDelayedTransition(myLayout);

    ConstraintSet set = new ConstraintSet();

    button.setMinimumWidth(500);
    button.setMinimumHeight(350);

    set.connect(R.id.myButton, ConstraintSet.BOTTOM,
           ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

    set.connect(R.id.myButton, ConstraintSet.RIGHT,
           ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);

    set.constrainWidth(R.id.myButton, ConstraintSet.WRAP_CONTENT);

    set.applyTo(myLayout);
}
```

When the application is now executed, the animation will slow to match the new duration setting and the button will bounce on arrival at the bottom right-hand corner of the display.

## 33.5 Summary

The most basic form of transition animation involves the use of the *beginDelayedTransition()* method of the TransitionManager class. Once called, any changes in size and position of the views in the next user interface rendering frame, and within a defined view group, will be animated using the specified

transitions. This chapter has worked through a simple Android Studio example that demonstrates the use of this approach to implementing transitions.

# 34. Implementing Android Scene Transitions – A Tutorial

This chapter will build on the theory outlined in the chapter entitled *Animating User Interfaces with the Android Transitions Framework* by working through the creation of a project designed to demonstrate transitioning from one scene to another using the Android Transition framework.

## 34.1 An Overview of the Scene Transition Project

The application created in this chapter will consist of two scenes, each represented by an XML layout resource file. A transition will then be used to animate the changes from one scene to another. The first scene will consist of three button views. The second scene will contain two of the buttons from the first scene positioned at different locations on the screen. The third button will be absent from the second scene. Once the transition has been implemented, movement of the first two buttons will be animated with a bounce effect. The third button will gently fade into view as the application transitions back to the first scene from the second.

## 34.2 Creating the Android Studio SceneTransitions Project

Create a new project in Android Studio, entering *SceneTransitions* 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 19: Android 4.4 (KitKat). Continue to proceed through the screens, requesting the creation of an Empty Activity named *SceneTransitionsActivity* with a corresponding layout file named *activity_scene_transitions*.

## 34.3 Identifying and Preparing the Root Container

When working with transitions it is important to identify the root container for the scenes. This is essentially the parent layout container into which the scenes are going to be displayed. When the project was created, Android Studio created a layout resource file in the *app -> res -> layout* folder named *activity_scene_transitions.xml* and containing a single layout container and TextView. When the application is launched, this is the first layout that will be displayed to the user on the device screen and for the purposes of this example, a RelativeLayout manager within this layout will act as the root container for the two scenes.

Begin by locating the *activity_scene_transitions.xml* layout resource file and loading it into the Android Studio Layout Editor tool. Switch to Text mode and replace the existing XML with the following to implement the RelativeLayout with an ID of *rootContainer*:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/rootContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
tools:context="com.ebookfrenzy.scenetransitions.SceneTransitionsActivity">
```

```
</RelativeLayout>
```

## 34.4 Designing the First Scene

The first scene is going to consist of a layout containing three button views. Create this layout resource file by right-clicking on the *app -> res -> layout* entry in the Project tool window and selecting the *New -> Layout resource file...* menu option. In the resulting dialog, name the file *scene1_layout* and enter *android.support.constraint.ConstraintLayout* as the root element before clicking on *OK*.

When the newly created layout file has loaded into the Layout Editor tool, check that Autoconnect mode is enabled, drag a Button view from the Widgets section of the palette onto the layout canvas and position it in the top left-hand corner of the layout view so that the dashed margin guidelines appear as illustrated in Figure 34-1. Drop the Button view at this position, select it and change the text value in the Properties tool window to "One".

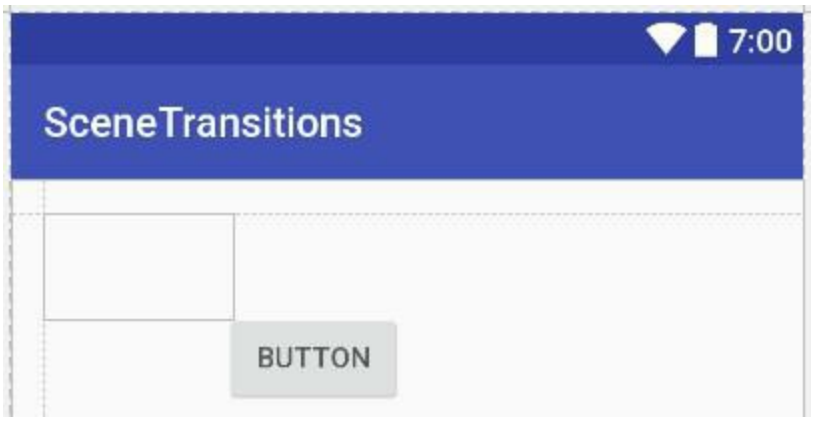


Figure 34-1

Drag a second Button view from the palette and position it in the top right-hand corner of the layout view so that the margin guidelines appear. Repeating the steps for the first button, assign text that reads "Two" to the button.

Drag a third Button view and position it so that it is centered both horizontally and vertically within the layout, this time configuring the button text to read "Three".

Click on the red warning button in the top right-hand corner of the Layout Editor and work through the list of I18N warnings, extracting the three button strings to resource values.

On completion of the above steps, the layout for the first scene should resemble that shown in Figure 34-2:

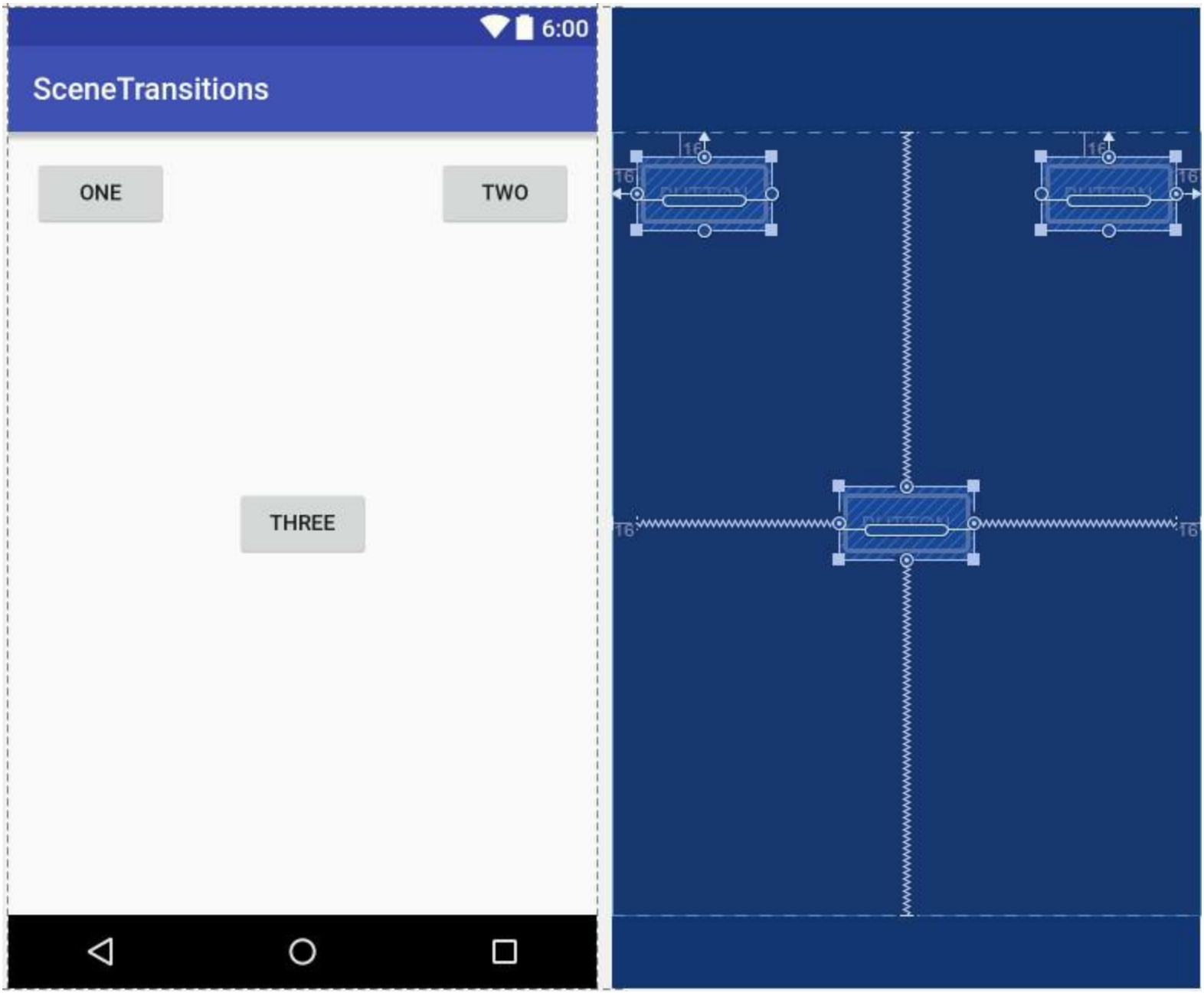


Figure 34-2

Select the “One” button and, using the properties tool window, configure the onClick attribute to call a method named *goToScene2*. Repeat this steps for the “Two” button, this time entering a method named *goToScene1* into the *onClick* field.

## 34.5 **Designing the Second Scene**

The second scene is simply a modified version of the first scene. The first and second buttons will still be present but will be located in the bottom right and left-hand corners of the layout respectively. The third button, on the other hand, will no longer be present in the second scene.

For the purposes of avoiding duplicated effort, the layout file for the second scene will be created by copying and modifying the *scene1_layout.xml* file. Within the Project tool window, locate the *app -> res -> layout -> scene1_layout.xml* file, right-click on it and select the *Copy* menu option. Right-click on the *layout* folder, this time selecting the *Paste* menu option and change the name of the file to *scene2_layout.xml* when prompted to do so.

Double-click on the new *scene2_layout.xml* file to load it into the Layout Editor tool and switch to Design mode if necessary. Right-click on the layout and select the *Clear all Constraints* option from

the resulting menu.

Select and delete the “Three” button and move the first and second buttons to the bottom right and bottom left locations as illustrated in Figure 34-3:

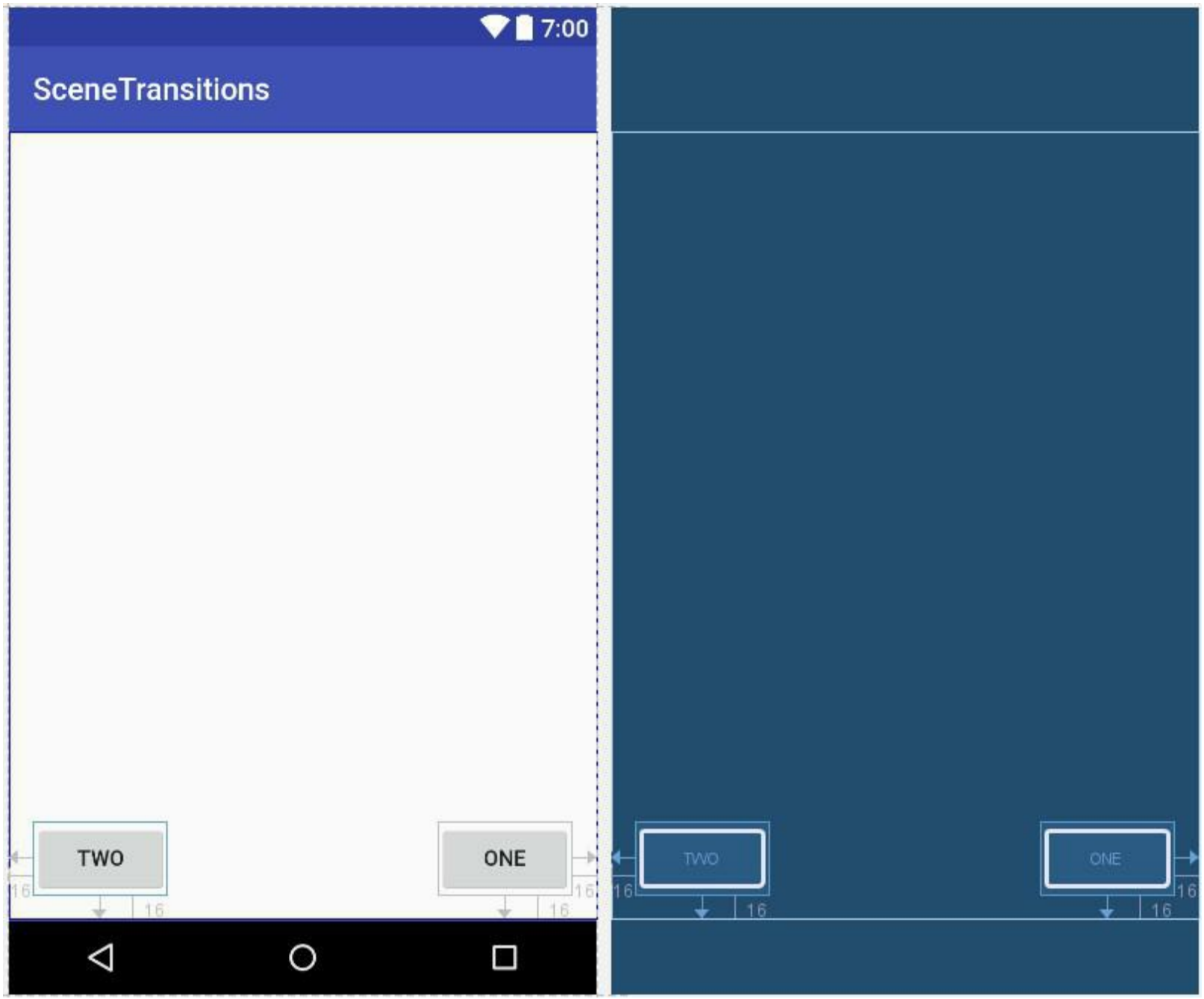


Figure 34-3

## 34.6 Entering the First Scene

If the application were to be run now, only the blank layout represented by the *activity_scene_transitions.xml* file would be displayed. Some code must, therefore, be added to the *onCreate()* method located in the *SceneTransitionsActivity.java* file so that the first scene is presented when the activity is created. This can be achieved as follows:

```
package com.ebookfrenzy.scenetransitions;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.transition.Scene;
import android.transition.Transition;
```

```
import android.transition.TransitionManager;
import android.view.ViewGroup;
import android.view.View;

public class SceneTransitionsActivity extends AppCompatActivity {

    ViewGroup rootContainer;
    Scene scene1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scene_transitions);

        rootContainer =
                (ViewGroup) findViewById(R.id.rootContainer);

        scene1 = Scene.getSceneForLayout(rootContainer,
                R.layout.scene1_layout, this);

        scene1.enter();
    }
.
.
}
```

The code added to the activity class declares some variables in which to store references to the root container and first scene and obtains a reference to the root container view. The *getSceneForLayout()* method of the Scene class is then used to create a scene from the layout contained in the *scene1_layout.xml* file to convert that layout into a scene. The scene is then entered via the *enter()* method call so that it is displayed to the user.

Compile and run the application at this point and verify that scene 1 is displayed after the application has launched.

## 34.7 Loading Scene 2

Before implementing the transition between the first and second scene it is first necessary to add some code to load the layout from the *scene2_layout.xml* file into a Scene instance. Remaining in the *SceneTransitionsActivity.java* file, therefore, add this code as follows:

```
public class SceneTransitionsActivity extends AppCompatActivity {

    ViewGroup rootContainer;
    Scene scene1;
    Scene scene2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_scene_transitions);

        rootContainer =
                (ViewGroup) findViewById(R.id.rootContainer);
```

```
            scene1 = Scene.getSceneForLayout(rootContainer,
                    R.layout.scene1_layout, this);

            scene2 = Scene.getSceneForLayout(rootContainer,
                    R.layout.scene2_layout, this);

            scene1.enter();
        }
    .
    .
    }
```

## 34.8 Implementing the Transitions

The first and second buttons have been configured to call methods named *goToScene2* and *goToScene1* respectively when selected. As the method names suggest, it is the responsibility of these methods to trigger the transitions between the two scenes. Add these two methods within the *SceneTransitionsActivity.java* file so that they read as follows:

```
public void goToScene2 (View view)
{
        TransitionManager.go(scene2);
}

public void goToScene1 (View view)
{
        TransitionManager.go(scene1);
}
```

Run the application and note that selecting the first two buttons causes the layout to switch between the two scenes. Since we have yet to configure any transitions, these layout changes are not yet animated.

## 34.9 Adding the Transition File

All of the transition effects for this project will be implemented within a single transition XML resource file. As outlined in the chapter entitled *Animating User Interfaces with the Android Transitions Framework*, transition resource files must be placed in the *app -> res -> transition* folder of the project. Begin, therefore, by right-clicking on the *res* folder in the Project tool window and selecting the *New -> Directory* menu option. In the resulting dialog, name the new folder *transition* and click on the *OK* button. Right-click on the new transition folder, this time selecting the *New -> File* option and name the new file *transition.xml*.

With the newly created *transition.xml* file selected and loaded into the editing panel, add the following XML content to add a transition set that enables the change bounds transition animation with a duration attribute setting:

```
<?xml version="1.0" encoding="utf-8"?>

<transitionSet
  xmlns:android="http://schemas.android.com/apk/res/android">

    <changeBounds
        android:duration="2000">
    </changeBounds>
```

```
</transitionSet>
```

## 34.10 Loading and Using the Transition Set

Although a transition resource file has been created and populated with a change bounds transition, this will have no effect until some code is added to load the transitions into a TransitionManager instance and reference it in the scene changes. The changes to achieve this are as follows:

```
package com.ebookfrenzy.scenetransitions;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.transition.Scene;
import android.transition.Transition;
import android.transition.TransitionInflater;
import android.transition.TransitionManager;
import android.view.ViewGroup;
import android.view.View;

public class SceneTransitionsActivity extends AppCompatActivity {

        ViewGroup rootContainer;
        Scene scene1;
        Scene scene2;
        Transition transitionMgr;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_scene_transitions);

                rootContainer =
                        (ViewGroup) findViewById(R.id.rootContainer);

                transitionMgr = TransitionInflater.from(this)
                            .inflateTransition(R.transition.transition);

                scene1 = Scene.getSceneForLayout(rootContainer,
                  R.layout.scene1_layout, this);

                scene2 = Scene.getSceneForLayout(rootContainer,
                  R.layout.scene2_layout, this);

                scene1.enter();
        }

        public void goToScene2 (View view)
        {
                TransitionManager.go(scene2, transitionMgr);
        }

        public void goToScene1 (View view)
        {
                TransitionManager.go(scene1, transitionMgr);
        }
```

```
.
.
}
```

When the application is now run the two buttons will gently glide to their new positions during the transition.

## 34.11 Configuring Additional Transitions

With the transition file integrated into the project, any number of additional transitions may be added to the file without the need to make any further changes to the Java source code of the activity. Take, for example, the following changes to the *transition.xml* file to add a bounce interpolator to the change bounds transition, introduce a fade-in transition targeted at the third button and to change the transitions such that they are performed sequentially:

```
<?xml version="1.0" encoding="utf-8"?>

<transitionSet
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:transitionOrdering="sequential" >


      <fade
        android:duration="2000"
        android:fadingMode="fade_in">

        <targets>
           <target android:targetId="@id/button3" />
        </targets>
      </fade>


    <changeBounds
        android:duration="2000"
        android:interpolator="@android:anim/bounce_interpolator">
    </changeBounds>
</transitionSet>
```

Buttons one and two will now bounce on arriving at the end destinations and button three will gently fade back into view when transitioning to scene 1 from scene 2.

Take some time to experiment with different transitions and interpolators by making changes to the *transition.xml* file and re-running the application.

## 34.12 Summary

Scene based transitions provide a flexible approach to animating user interface layout changes within an Android application. This chapter has demonstrated the steps involved in animating the transition between the scenes represented by two layout resource files. In addition, the example also used a transition XML resource file to configure the transition animation effects between the two scenes.