21. Manual XML Layout Design in Android Studio

While the design of layouts using the Android Studio Layout Editor tool greatly improves productivity, it is still possible to create XML layouts by manually editing the underlying XML. This chapter will introduce the basics of the Android XML layout file format.

21.1 Manually Creating an XML Layout

The structure of an XML layout file is actually quite straightforward and follows the hierarchical approach of the view tree. The first line of an XML resource file should ideally include the following standard declaration:

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

This declaration should be followed by the root element of the layout, typically a container view such as a layout manager. This is represented by both opening and closing tags and any properties that need to be set on the view. The following XML, for example, declares a ConstraintLayout view as the root element, assigns the ID *activity_main* and sets *match_parent* properties such that it fills all the available space of the device display:

```
<android.support.constraint.ConstraintLayout
    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/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.ebookfrenzy.myapplication.MainActivity">


</android.support.constraint.ConstraintLayout>
```

Note that the layout element is also configured with padding on each side of 16dp (density independent pixels). Any specification of spacing in an Android layout must be specified using one of the following units of measurement:

- **in** – Inches.
- **mm** – Millimeters.
- **pt** – Points (1/72 of an inch).
- **dp** – Density-independent pixels. An abstract unit of measurement based on the physical density of the device display relative to a 160dpi display baseline.
- **sp** – Scale-independent pixels. Similar to dp but scaled based on the user's font preference.
- **px** – Actual screen pixels. Use is not recommended since different displays will have different pixels per inch. Use *dp* in preference to this unit.

Any children that need to be added to the ConstraintLayout parent must be *nested* within the opening and closing tags. In the following example a Button widget has been added as a child of the

ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="com.ebookfrenzy.myapplication.MainActivity">

    <Button
        android:text="Button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button" />

</android.support.constraint.ConstraintLayout>
```

As currently implemented, the button has no constraint connections. At runtime, therefore, the button will appear in the top left-hand corner of the screen (though indented 16dp by the padding assigned to the parent layout). If opposing constraints are added to the sides of the button, however, it will appear centered within the layout:

```
<Button
    android:text="Button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button"
    app:layout_constraintLeft_toLeftOf="@+id/activity_main"
    app:layout_constraintTop_toTopOf="@+id/activity_main"
    app:layout_constraintRight_toRightOf="@+id/activity_main"
    app:layout_constraintBottom_toBottomOf="@+id/activity_main" />
```

Note that each of the constraints is attached to the element named *activity_main* which is, in this case, the parent ConstraintLayout instance.

To add a second widget to the layout, simply embed it within the body of ConstraintLayout element. The following modification, for example, adds a TextView widget to the layout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
```

```
    tools:context="com.ebookfrenzy.myapplication.MainActivity">

    <Button
        android:text="Button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/button"
        app:layout_constraintLeft_toLeftOf="@+id/activity_main"
        app:layout_constraintTop_toTopOf="@+id/activity_main"
        app:layout_constraintRight_toRightOf="@+id/activity_main"
        app:layout_constraintBottom_toBottomOf="@+id/activity_main" />

    <TextView
        android:text="TextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textView" />

</android.support.constraint.ConstraintLayout>
```

Once again, the absence of constraints on the newly added TextView will cause it to appear in the top left-hand corner of the layout at runtime. The following modifications add opposing constraints connected to the parent layout to center the widget horizontally, together with a constraint connecting the bottom of the TextView to the top of the button with a margin of 72dp:

```
<TextView
    android:text="TextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/textView"
    app:layout_constraintLeft_toLeftOf="@+id/activity_main"
    app:layout_constraintRight_toRightOf="@+id/activity_main"
    app:layout_constraintBottom_toTopOf="@+id/button"
    android:layout_marginBottom="72dp" />
```

Also, note that the Button and TextView views have a number of properties declared. Both views have been assigned IDs and configured to display text strings represented by string resources named *button_string* and *text_string* respectively. Additionally, the *wrap_content* height and width properties have been declared on both objects so that they are sized to accommodate the content (in this case the text referenced by the string resource value).

Viewed from within the Preview panel of the Layout Editor in Text mode, the above layout will be rendered as shown in Figure 21-1:

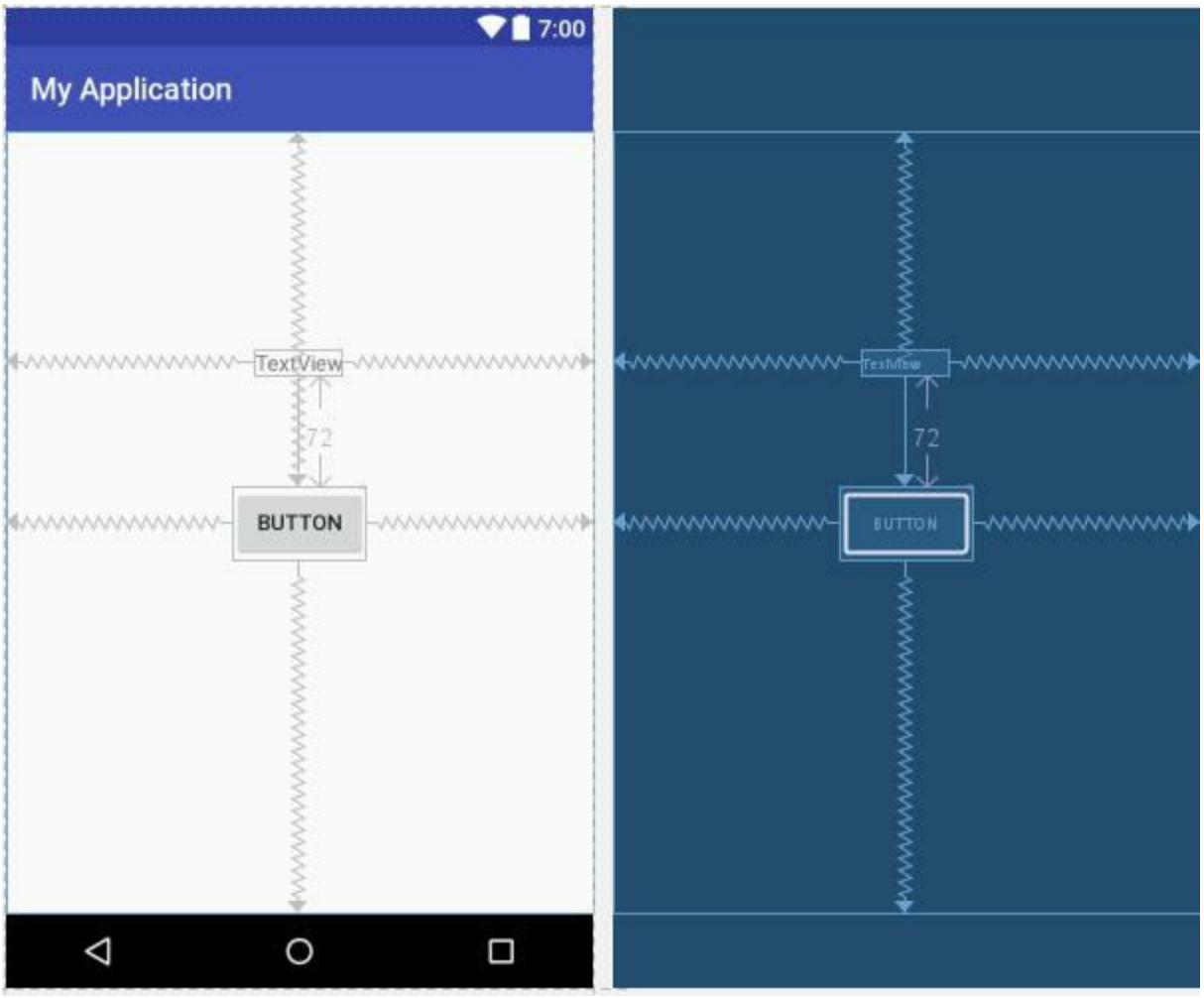

Figure 21-1

21.2 Manual XML vs. Visual Layout Design

When to write XML manually as opposed to using the Layout Editor tool in design mode is a matter of personal preference. There are, however, advantages to using design mode.

First, design mode will generally be quicker given that it avoids the necessity to type lines of XML. Additionally, design mode avoids the need to learn the intricacies of the various property values of the Android SDK view classes. Rather than continually refer to the Android documentation to find the correct keywords and values, most properties can be located by referring to the Properties panel.

All the advantages of design mode aside, it is important to keep in mind that the two approaches to user interface design are in no way mutually exclusive. As an application developer, it is quite likely that you will end up creating user interfaces within design mode while performing fine-tuning and layout tweaks of the design by directly editing the generated XML resources. Both views of the interface design are, after all, displayed side by side within the Android Studio environment making it easy to work seamlessly on both the XML and the visual layout.

21.3 Summary

The Android Studio Layout Editor tool provides a visually intuitive method for designing user interfaces. Using a drag and drop paradigm combined with a set of property editors, the tool provides considerable productivity benefits to the application developer.

User interface designs may also be implemented by manually writing the XML layout resource files, the format of which is well structured and easily understood.

The fact that the Layout Editor tool generates XML resource files means that these two approaches to interface design can be combined to provide a “best of both worlds” approach to user interface development.

22. Managing Constraints using Constraint Sets

Up until this point in the book, all user interface design tasks have been performed using the Android Studio Layout Editor tool, either in text or design mode. An alternative to writing XML resource files or using the Android Studio Layout Editor is to write Java code to directly create, configure and manipulate the view objects that comprise the user interface of an Android activity. Within the context of this chapter, we will explore some of the advantages and disadvantages of writing Java code to create a user interface before describing some of the key concepts such as view properties and the creation and management of layout constraints.

In the next chapter, an example project will be created and used to demonstrate some of the typical steps involved in this approach to Android user interface creation.

22.1 Java Code vs. XML Layout Files

There are a number of key advantages to using XML resource files to design a user interface as opposed to writing Java code. In fact, Google goes to considerable lengths in the Android documentation to extol the virtues of XML resources over Java code. As discussed in the previous chapter, one key advantage to the XML approach includes the ability to use the Android Studio Layout Editor tool, which, itself, generates XML resources. A second advantage is that once an application has been created, changes to user interface screens can be made by simply modifying the XML file, thereby avoiding the necessity to recompile the application. Also, even when hand writing XML layouts, it is possible to get instant feedback on the appearance of the user interface using the preview feature of the Android Studio Layout Editor tool. In order to test the appearance of a Java created user interface the developer will, inevitably, repeatedly cycle through a loop of writing code, compiling and testing in order to complete the design work.

In terms of the strengths of the Java coding approach to layout creation, perhaps the most significant advantage that Java has over XML resource files comes into play when dealing with dynamic user interfaces. XML resource files are inherently most useful when defining static layouts, in other words layouts that are unlikely to change significantly from one invocation of an activity to the next. Java code, on the other hand, is ideal for creating user interfaces dynamically at run-time. This is particularly useful in situations where the user interface may appear differently each time the activity executes subject to external factors.

A knowledge of working with user interface components in Java code can also be useful when dynamic changes to a static XML resource based layout need to be performed in real-time as the activity is running.

Finally, some developers simply prefer to write Java code than to use layout tools and XML, regardless of the advantages offered by the latter approaches.

22.2 Creating Views

As previously established, the Android SDK includes a toolbox of view classes designed to meet most of the basic user interface design needs. The creation of a view in Java is simply a matter of creating instances of these classes, passing through as an argument a reference to the activity with which that view is to be associated.

The first view (typically a container view to which additional child views can be added) is displayed to the user via a call to the *setContentView()* method of the activity. Additional views may be added to the root view via calls to the object's *addView()* method.

When working with Java code to manipulate views contained in XML layout resource files, it is necessary to obtain the ID of the view. The same rule holds true for views created in Java. As such, it is necessary to assign an ID to any view for which certain types of access will be required in subsequent Java code. This is achieved via a call to the *setId()* method of the view object in question. In later code, the ID for a view may be obtained via a subsequent call to the object's *getId()* method.

22.3 View Properties

Each view class has associated with it a range of *properties*. These property settings are set directly on the view instances and generally define how the view object will appear or behave. Examples of properties are the text that appears on a Button object, or the background color of a ConstraintLayout view. Each view class within the Android SDK has a pre-defined set of methods that allow the user to *set* and *get* these property values. The Button class, for example, has a *setText()* method which can be called from within Java code to set the text displayed on the button to a specific string value. The background color of a *ConstraintLayout* object, on the other hand, can be set with a call to the object's *setBackgroundColor()* method.

22.4 Constraint Sets

While property settings are internal to view objects and dictate how a view appears and behaves, *constraint sets* are used to control how a view appears relative to its parent view and other sibling views. Every ConstraintLayout instance has associated with it a set of constraints that define how its child views are positioned and constrained.

The key to working with constraint sets in Java code is the *ConstraintSet* class. This class contains a range of methods that allow tasks such as creating, configuring and applying constraints to a ConstraintLayout instance. In addition, the current constraints for a ConstraintLayout instance may be copied into a ConstraintSet object and used to apply the same constraints to other layouts (with or without modifications).

A ConstraintSet instance is created just like any other Java object:

```
ConstraintSet set = new ConstraintSet();
```

Once a constraint set has been created, methods can be called on the instance to perform a wide range of tasks.

22.4.1 Establishing Connections

The *connect()* method of the ConstraintSet class is used to establish constraint connections between views. The following code configures a constraint set in which the left-hand side of a Button view is connected to the right-hand side of an EditText view with a margin of 70dp:

```
set.connect(button1.getId(), ConstraintSet.LEFT,
            editText1.getId(), ConstraintSet.RIGHT, 70);
```

22.4.2 Applying Constraints to a Layout

Once the constraint set is configured, it must be applied to a ConstraintLayout instance before it will take effect. A constraint set is applied via a call to the *applyTo()* method, passing through a reference

to the layout object to which the settings are to be applied:

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

22.4.3 Parent Constraint Connections

Connections may also be established between a child view and its parent ConstraintLayout by referencing the ConstraintSet.PARENT_ID constant. In the following example, the constraint set is configured to connect the top edge of a Button view to the top of the parent layout with a margin of 100dp:

```
set.connect(button1.getId(), ConstraintSet.TOP,
                  ConstraintSet.PARENT_ID, ConstraintSet.TOP, 100);
```

22.4.4 Sizing Constraints

A number of methods are available for controlling the sizing behavior of views. The following code, for example, sets the horizontal size of a Button view to *wrap_content* and the vertical size of an ImageView instance to a maximum of 250dp:

```
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);
set.constrainMaxHeight(imageView1.getId(), 250);
```

22.4.5 Constraint Bias

As outlined in the chapter entitled *A Guide to using ConstraintLayout in Android Studio*, when a view has opposing constraints it is centered along the axis of the constraints (i.e. horizontally or vertically). This centering can be adjusted by applying a bias along the particular axis of constraint. When using the Android Studio Layout Editor, this is achieved using the controls in the properties tool window. When working with a constraint set, however, bias can be added using the *setHorizontalBias()* and *setVerticalBias()* methods, referencing the view ID and the bias as a floating point value between 0 and 1.

The following code, for example, constrains the left and right-hand sides of a Button to the corresponding sides of the parent layout before applying a 25% horizontal bias:

```
set.connect(button1.getId(), ConstraintSet.LEFT,
                ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);
set.connect(button1.getId(), ConstraintSet.RIGHT,
                ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);
set.setHorizontalBias(button1.getId(), 0.25f);
```

22.4.6 Alignment Constraints

Alignments may also be applied using a constraint set. The full set of alignment options available with the Android Studio Layout Editor may also be configured using a constraint set via the *centerVertically()* and *centerHorizontally()* methods, both of which take a variety of arguments depending on the alignment being configured. In addition, the *center()* method may be used to center a view between two other views.

In the code below, button2 is positioned so that it is aligned horizontally with button1:

```
set.centerHorizontally(button2.getId(), button1.getId());
```

22.4.7 Copying and Applying Constraint Sets

The current constraint set for a ConstraintLayout instance may be copied into a constraint set object

using the *clone()* method. The following line of code, for example, copies the constraint settings from a ConstraintLayout instance named *myLayout* into a constraint set object:

```
set.clone(myLayout);
```

Once copied, the constraint set may be applied directly to another layout or, as in the following example, modified before being applied to the second layout:

```
ConstraintSet set = new ConstraintSet();
set.clone(myLayout);
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);
set.applyTo(mySecondLayout);
```

22.4.8 ConstraintLayout Chains

Vertical and horizontal chains may also be created within a constraint set using the *createHorizontalChain()* and *createVerticalChain()* methods. The syntax for using these methods is as follows:

```
createHorizontalChain(int leftId, int leftSide, int rightId,
     int rightSide, int[] chainIds, float[] weights, int style);

createVerticalChain(int topId, int topSide, int bottomId,
     int bottomSide, int[] chainIds, float[] weights, int style)
```

Based on the above syntax, the following example creates a horizontal spread chain that starts with button1 and ends with button4. In between these views are button2 and button3 with weighting set to zero for both:

```
int[] chainViews = {button2.getId(), button3.getId()};
float[] chainWeights = {0, 0};

set.createHorizontalChain(button1.getId(), ConstraintSet.LEFT,
                          button4.getId(), ConstraintSet.RIGHT,
                          chainViews, chainWeights,
                          ConstraintSet.CHAIN_SPREAD);
```

A view can be removed from a chain by passing the ID of the view to be removed through to either the *removeFromHorizontalChain()* or *removeFromVerticalChain()* methods. A view may be added to an existing chain using either the *addToHorizontalChain()* or *addToVerticalChain()* methods. In both cases the methods take as arguments the IDs of the views between which the new view is to be inserted as follows:

```
set.addToHorizontalChain (newViewId, leftViewId, rightViewId);
```

22.4.9 Guidelines

Guidelines are added to a constraint set using the *create()* method and then positioned using the *setGuidelineBegin()*, *setGuidelineEnd()* or *setGuidelinePercent()* methods. In the following code, a vertical guideline is created and positioned 50% across the width of the parent layout. The left side of a button view is then connected to the guideline with no margin:

```
set.create(R.id.myGuidelineId, ConstraintSet.VERTICAL_GUIDELINE);
set.setGuidelinePercent(R.id.myGuidelineId, 0.5f);

set.connect(button.getId(), ConstraintSet.LEFT,
           R.id.myGuidelineId, ConstraintSet.RIGHT, 0);
```

```
set.applyTo(layout);
```

22.4.10 Removing Constraints

A constraint may be removed from a view in a constraint set using the *clear()* method, passing through as arguments the view ID and the anchor point for which the constraint is to be removed:

```
set.clear(button.getId(), ConstraintSet.LEFT);
```

Similarly, all of the constraints on a view may be removed in a single step by referencing only the view in the *clear()* method call:

```
set.clear(button.getId()
```

22.4.11 Scaling

The scale of a view within a layout may be adjusted using the ConstraintSet *setScaleX()* and *setScaleY()* methods which take as arguments the view on which the operation is to be performed together with a float value indicating the scale. In the following code, a button object is scaled to twice its original width and half the height:

```
set.setScaleX(myButton.getId(), 2f);
set.setScaleY(myButton.getId(), 0.5f);
```

22.4.12 Rotation

A view may be rotated on either the X or Y axis using the *setRotationX()* and *setRotationY()* methods respectively both of which must be passed the ID of the view to be rotated and a float value representing the degree of rotation to be performed. The pivot point on which the rotation is to take place may be defined via a call to the *setTransformPivot()*, *setTransformPivotX()* and *setTransformPivotY()* methods. The following code rotates a button view 30 degrees on the Y axis using a pivot point located at point 500, 500:

```
set.setTransformPivot(button.getId(), 500, 500);
set.setRotationY(button.getId(), 30);
set.applyTo(layout);
```

Having covered the theory of constraint sets and user interface creation from within Java code, the next chapter will work through the creation of an example application with the objective of putting this theory into practice. For more details on the ConstraintSet class, refer to the reference guide at the following URL:

https://developer.android.com/reference/android/support/constraint/ConstraintSet.html

22.5 Summary

As an alternative to writing XML layout resource files or using the Android Studio Layout Editor tool, Android user interfaces may also be dynamically created in Java code.

Creating layouts in Java code consists of creating instances of view classes and setting properties on those objects to define required appearance and behavior.

How a view is positioned and sized relative to its ConstraintLayout parent view and any sibling views is defined through the use of constraint sets. A constraint set is represented by an instance of the ConstraintSet class which, once created, can be configured using a wide range of method calls to perform tasks such as establishing constraint connections, controlling view sizing behavior and

creating chains.

With the basics of the ConstraintSet class covered in this chapter, the next chapter will work through a tutorial that puts these features to practical use.

23. An Android ConstraintSet Tutorial

The previous chapter introduced the basic concepts of creating and modifying user interface layouts in Java code using the ConstraintLayout and ConstraintSet classes. This chapter will take these concepts and put them into practice through the creation of an example layout created entirely in Java code and without using the Android Studio Layout Editor tool.

23.1 Creating the Example Project in Android Studio

Launch Android Studio and select the *Start a new Android Studio project* option from the quick start menu in the welcome screen.

In the new project configuration dialog, enter *JavaLayout* 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 an Empty Activity named *JavaLayoutActivity* with a corresponding layout named *activity_java_layout*.

Once the project has been created, the *JavaLayoutActivity.java* file should automatically load into the editing panel. As we have come to expect, Android Studio has created a template activity and overridden the *onCreate()* method, providing an ideal location for Java code to be added to create a user interface.

23.2 Adding Views to an Activity

The *onCreate()* method is currently designed to use a resource layout file for the user interface. Begin, therefore, by deleting this line from the method:

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

The next modification to the *onCreate()* method is to write some Java code to add a ConstraintLayout object with a single Button view child to the activity. This involves the creation of new instances of the ConstraintLayout and Button classes. The Button view then needs to be added as a child to the ConstraintLayout view which, in turn, is displayed via a call to the *setContentView()* method of the activity instance:

```
package com.ebookfrenzy.javalayout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintSet;
import android.support.constraint.ConstraintLayout;
import android.widget.Button;
import android.widget.EditText;

public class JavaLayoutActivity extends AppCompatActivity {
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button myButton = new Button(this);
        ConstraintLayout myLayout = new ConstraintLayout(this);
        myLayout.addView(myButton);
        setContentView(myLayout);
    }
}
```

When new instances of user interface objects are created in this way, the constructor methods must be passed the context within which the object is being created which, in this case, is the current activity. Since the above code resides within the activity class, the context is simply referenced by the standard Java *this* keyword:

```
Button myButton = new Button(this);
```

Once the above additions have been made, compile and run the application (either on a physical device or an emulator). Once launched, the visible result will be a button containing no text appearing in the top left-hand corner of the ConstraintLayout view as shown in Figure 23-1:

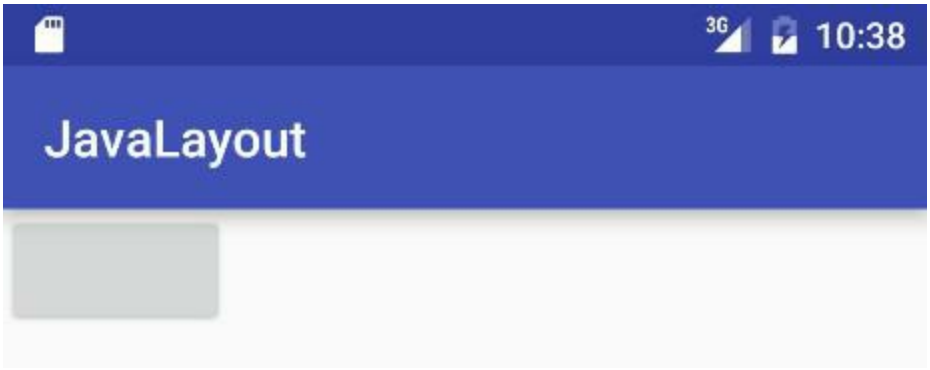

Figure 23-1

23.3 Setting View Properties

For the purposes of this exercise, we need the background of the ConstraintLayout view to be blue and the Button view to display text that reads “Press Me” on a yellow background. Both of these tasks can be achieved by setting properties on the views in the Java code as outlined in the following code fragment:

```
.
.
import android.graphics.Color;

public class JavaLayoutActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button myButton = new Button(this);
        myButton.setText("Press Me");
        myButton.setBackgroundColor(Color.YELLOW);

        ConstraintLayout myLayout = new ConstraintLayout(this);
        myLayout.setBackgroundColor(Color.BLUE);
```

```
        myLayout.addView(myButton);
        setContentView(myLayout);

    }
}
```

When the application is now compiled and run, the layout will reflect the property settings such that the layout will appear with a blue background and the button will display the assigned text on a yellow background.

23.4 Creating View IDs

When the layout is complete it will consist of a Button and an EditText view. Before these views can be referenced within the methods of the ConstraintSet class, they must be assigned unique view IDs. The first step in this process is to create a new resource file containing these ID values.

Right click on the *app -> res -> values* folder, select the *New -> Values resource file* menu option and name the new resource file *id.xml*. With the resource file created, edit it so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item name="myButton" type="id" />
    <item name="myEditText" type="id" />
</resources>
```

At this point in the tutorial, only the Button has been created, so edit the *onCreate()* method to assign the corresponding ID to the object:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Button myButton = new Button(this);
    myButton.setText("Press Me");
    myButton.setBackgroundColor(Color.YELLOW);
    myButton.setId(R.id.myButton);
.
.
}
```

23.5 Configuring the Constraint Set

In the absence of any constraints, the ConstraintLayout view has placed the Button view in the top left corner of the display. In order to instruct the layout view to place the button in a different location, in this case centered both horizontally and vertically, it will be necessary to create a ConstraintSet instance, initialize it with the appropriate settings and apply it to the parent layout.

For this example, the button needs to be configured so that the width and height are constrained to the size of the text it is displaying and the view centered within the parent layout. Edit the *onCreate()* method once more to make these changes:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    Button myButton = new Button(this);
```

```
        myButton.setText("Press Me");
        myButton.setBackgroundColor(Color.YELLOW);
        myButton.setId(R.id.myButton);

        ConstraintLayout myLayout = new ConstraintLayout(this);
        myLayout.setBackgroundColor(Color.BLUE);

        myLayout.addView(myButton);
        setContentView(myLayout);

        ConstraintSet set = new ConstraintSet();

        set.constrainHeight(myButton.getId(),
                            ConstraintSet.WRAP_CONTENT);
        set.constrainWidth(myButton.getId(),
                            ConstraintSet.WRAP_CONTENT);

        set.connect(myButton.getId(), ConstraintSet.LEFT,
                    ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);
        set.connect(myButton.getId(), ConstraintSet.RIGHT,
                    ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);
        set.connect(myButton.getId(), ConstraintSet.TOP,
                    ConstraintSet.PARENT_ID, ConstraintSet.TOP, 0);
        set.connect(myButton.getId(), ConstraintSet.BOTTOM,
                    ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

        set.applyTo(myLayout);
    }
```

With the initial constraints configured, compile and run the application and verify that the Button view now appears in the center of the layout:

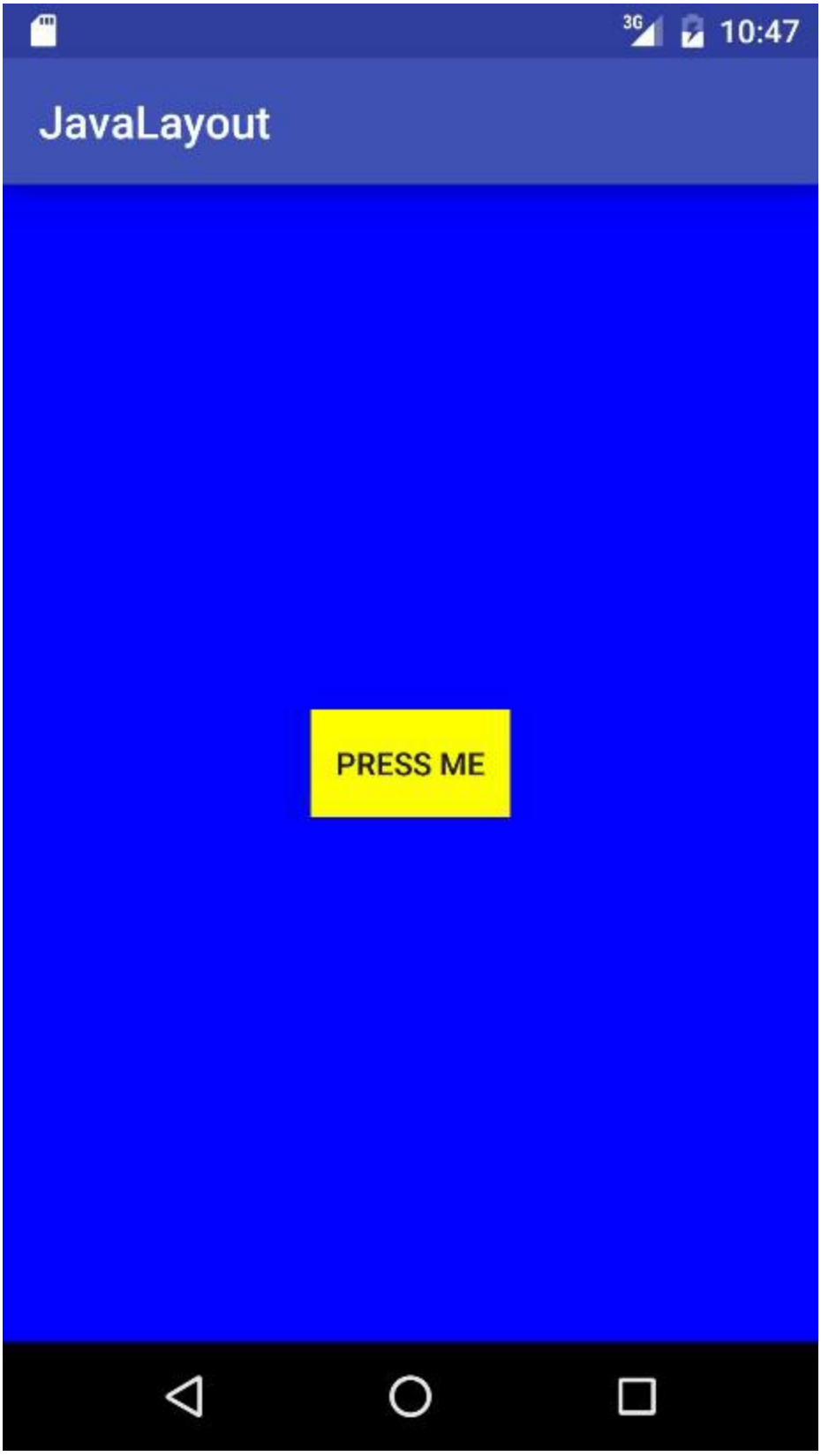

Figure 23-2

23.6 Adding the EditText View

The next item to be added to the layout is the EditText view. The first step is to create the EditText object, assign it the ID as declared in the *id.xml* resource file and add it to the layout. The code changes to achieve these steps now need to be made to the *onCreate()* method as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
```

```
        Button myButton = new Button(this);
        myButton.setText("Press Me");
        myButton.setBackgroundColor(Color.YELLOW);
        myButton.setId(R.id.myButton);

        EditText myEditText = new EditText(this);
        myEditText.setId(R.id.myEditText);

        ConstraintLayout myLayout = new ConstraintLayout(this);
        myLayout.setBackgroundColor(Color.BLUE);

        myLayout.addView(myButton);
        myLayout.addView(myEditText);

        setContentView(myLayout);
    .
    .
    }
```

The EditText widget is intended to be sized subject to the content it is displaying, centered horizontally within the layout and positioned 70dp above the existing Button view. Add code to the *onCreate()* method so that it reads as follows:

```
    .
    .
    .
    set.connect(myButton.getId(), ConstraintSet.LEFT,
                ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);
    set.connect(myButton.getId(), ConstraintSet.RIGHT,
                ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);
    set.connect(myButton.getId(), ConstraintSet.TOP,
                ConstraintSet.PARENT_ID, ConstraintSet.TOP, 0);
    set.connect(myButton.getId(), ConstraintSet.BOTTOM,
                ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

    set.constrainHeight(myEditText.getId(),
                            ConstraintSet.WRAP_CONTENT);
    set.constrainWidth(myEditText.getId(),
                            ConstraintSet.WRAP_CONTENT);

    set.connect(myEditText.getId(), ConstraintSet.LEFT,
                ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);
    set.connect(myEditText.getId(), ConstraintSet.RIGHT,
                ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);
    set.connect(myEditText.getId(), ConstraintSet.BOTTOM,
                myButton.getId(), ConstraintSet.TOP, 70);

    set.applyTo(myLayout);
```

A test run of the application should show the EditText field centered above the button with a margin of 70dp.

23.7 Converting Density Independent Pixels (dp) to Pixels (px)

The next task in this exercise is to set the width of the EditText view to 200dp. As outlined in the

chapter entitled *Designing an Android User Interface using the Graphical Layout Tool*, when setting sizes and positions in user interface layouts it is better to use density independent pixels (dp) rather than pixels (px). In order to set a position using dp it is necessary to convert a dp value to a px value at runtime, taking into consideration the density of the device display. In order, therefore, to set the width of the EditText view to 200dp, the following code needs to be added to the *onCreate()* method:

```
package com.ebookfrenzy.javalayout;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.constraint.ConstraintLayout;
import android.support.constraint.ConstraintSet;
import android.widget.Button;
import android.widget.EditText;
import android.graphics.Color;
import android.content.res.Resources;
import android.util.TypedValue;

public class JavaLayoutActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        Button myButton = new Button(this);
        myButton.setText("Press Me");
        myButton.setBackgroundColor(Color.YELLOW);
        myButton.setId(R.id.myButton);

        EditText myEditText = new EditText(this);
        myEditText.setId(R.id.myEditText);

        Resources r = getResources();
        int px = (int) TypedValue.applyDimension(
                TypedValue.COMPLEX_UNIT_DIP, 200,
                r.getDisplayMetrics());

        myEditText.setWidth(px);
.
.
}
```

Compile and run the application one more time and note that the width of the EditText view has changed as illustrated in Figure 23-3:

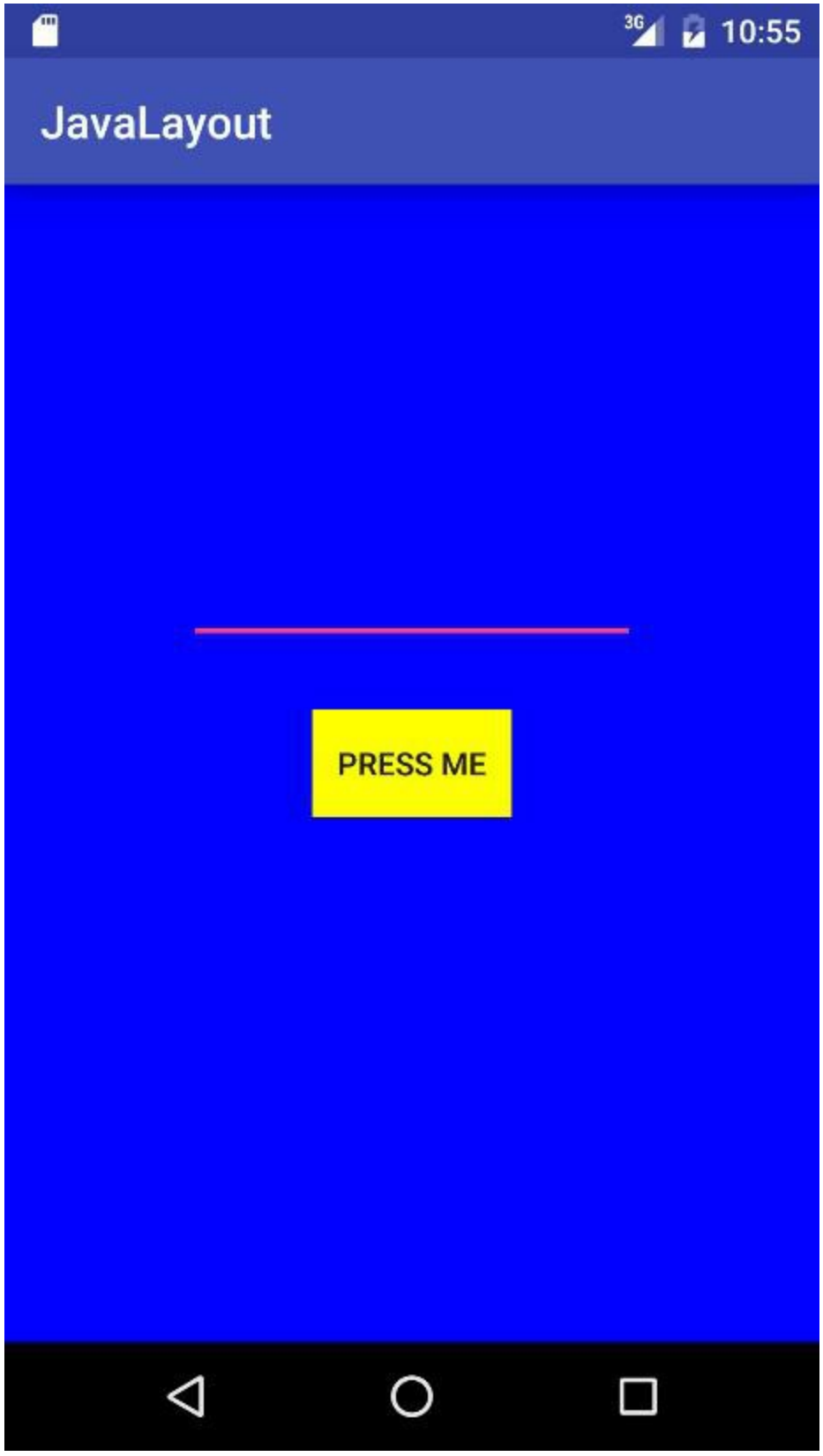

Figure 23-3

23.8 Summary

The example activity created in this chapter has, of course, created a similar user interface (the change in background color and view type notwithstanding) as that created in the earlier *Manual XML Layout Design in Android Studio* chapter. If nothing else, this chapter should have provided an appreciation of the level to which the Android Studio Layout Editor tool and XML resources shield the developer from many of the complexities of creating Android user interface layouts.

There are, however, instances where it makes sense to create a user interface in Java. This approach

is most useful, for example, when creating dynamic user interface layouts.

24. An Overview and Example of Android Event Handling

Much has been covered in the previous chapters relating to the design of user interfaces for Android applications. An area that has yet to be covered, however, involves the way in which a user's interaction with the user interface triggers the underlying activity to perform a task. In other words, we know from the previous chapters how to create a user interface containing a button view, but not how to make something happen within the application when it is touched by the user.

The primary objective of this chapter, therefore, is to provide an overview of event handling in Android applications together with an Android Studio based example project.

24.1 Understanding Android Events

Events in Android can take a variety of different forms, but are usually generated in response to an external action. The most common form of events, particularly for devices such as tablets and smartphones, involve some form of interaction with the touch screen. Such events fall into the category of *input events*.

The Android framework maintains an *event queue* into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis. In the case of an input event such as a touch on the screen, the event is passed to the view positioned at the location on the screen where the touch took place. In addition to the event notification, the view is also passed a range of information (depending on the event type) about the nature of the event such as the coordinates of the point of contact between the user's fingertip and the screen.

In order to be able to handle the event that it has been passed, the view must have in place an *event listener*. The Android View class, from which all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method. In order to be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback. For example, if a button is to respond to a *click* event (the equivalent to the user touching and releasing the button view as though clicking on a physical button) it must both register the *View.onClickListener* event listener (via a call to the target view's *setOnClickListener()* method) and implement the corresponding *onClick()* callback method. In the event that a "click" event is detected on the screen at the location of the button view, the Android framework will call the *onClick()* method of that view when that event is removed from the event queue. It is, of course, within the implementation of the *onClick()* callback method that any tasks should be performed or other methods called in response to the button click.

24.2 Using the *android:onClick* Resource

Before exploring event listeners in more detail it is worth noting that a shortcut is available when all that is required is for a callback method to be called when a user "clicks" on a button view in the user interface. Consider a user interface layout containing a button view named *button1* with the requirement that when the user touches the button, a method called *buttonClick()* declared in the activity class is called. All that is required to implement this behavior is to write the *buttonClick()* method (which takes as an argument a reference to the view that triggered the click event) and add a

single line to the declaration of the button view in the XML file. For example:

```
<Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="buttonClick"
        android:text="Click me" />
```

This provides a simple way to capture click events. It does not, however, provide the range of options offered by event handlers, which are the topic of the rest of this chapter. When working within Android Studio Layout Editor, the onClick property can be found and configured in the Properties panel when a suitable view type is selected in the device screen layout.

24.3 Event Listeners and Callback Methods

In the example activity outlined later in this chapter the steps involved in registering an event listener and implementing the callback method will be covered in detail. Before doing so, however, it is worth taking some time to outline the event listeners that are available in the Android framework and the callback methods associated with each one.

- **onClickListener** – Used to detect click style events whereby the user touches and then releases an area of the device display occupied by a view. Corresponds to the *onClick()* callback method which is passed a reference to the view that received the event as an argument.
- **onLongClickListener** – Used to detect when the user maintains the touch over a view for an extended period. Corresponds to the *onLongClick()* callback method which is passed as an argument the view that received the event.
- **onTouchListener** – Used to detect any form of contact with the touch screen including individual or multiple touches and gesture motions. Corresponding with the *onTouch()* callback, this topic will be covered in greater detail in the chapter entitled *Android Touch and Multi-touch Event Handling*. The callback method is passed as arguments the view that received the event and a MotionEvent object.
- **onCreateContextMenuListener** – Listens for the creation of a context menu as the result of a long click. Corresponds to the *onCreateContextMenu()* callback method. The callback is passed the menu, the view that received the event and a menu context object.
- **onFocusChangeListener** – Detects when focus moves away from the current view as the result of interaction with a track-ball or navigation key. Corresponds to the *onFocusChange()* callback method which is passed the view that received the event and a Boolean value to indicate whether focus was gained or lost.
- **onKeyListener** – Used to detect when a key on a device is pressed while a view has focus. Corresponds to the *onKey()* callback method. Passed as arguments are the view that received the event, the KeyCode of the physical key that was pressed and a KeyEvent object.

24.4 An Event Handling Example

In the remainder of this chapter, we will work through the creation of a simple Android Studio project designed to demonstrate the implementation of an event listener and corresponding callback method to detect when the user has clicked on a button. The code within the callback method will update a text view to indicate that the event has been processed.

Create a new project in Android Studio, entering *EventExample* 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 an Empty Activity named *EventExampleActivity* with corresponding layout file named *activity_event_example*.

24.5 Designing the User Interface

The user interface layout for the *EventExampleActivity* class in this example is to consist of a ConstraintLayout, a Button and a TextView as illustrated in Figure 24-1.

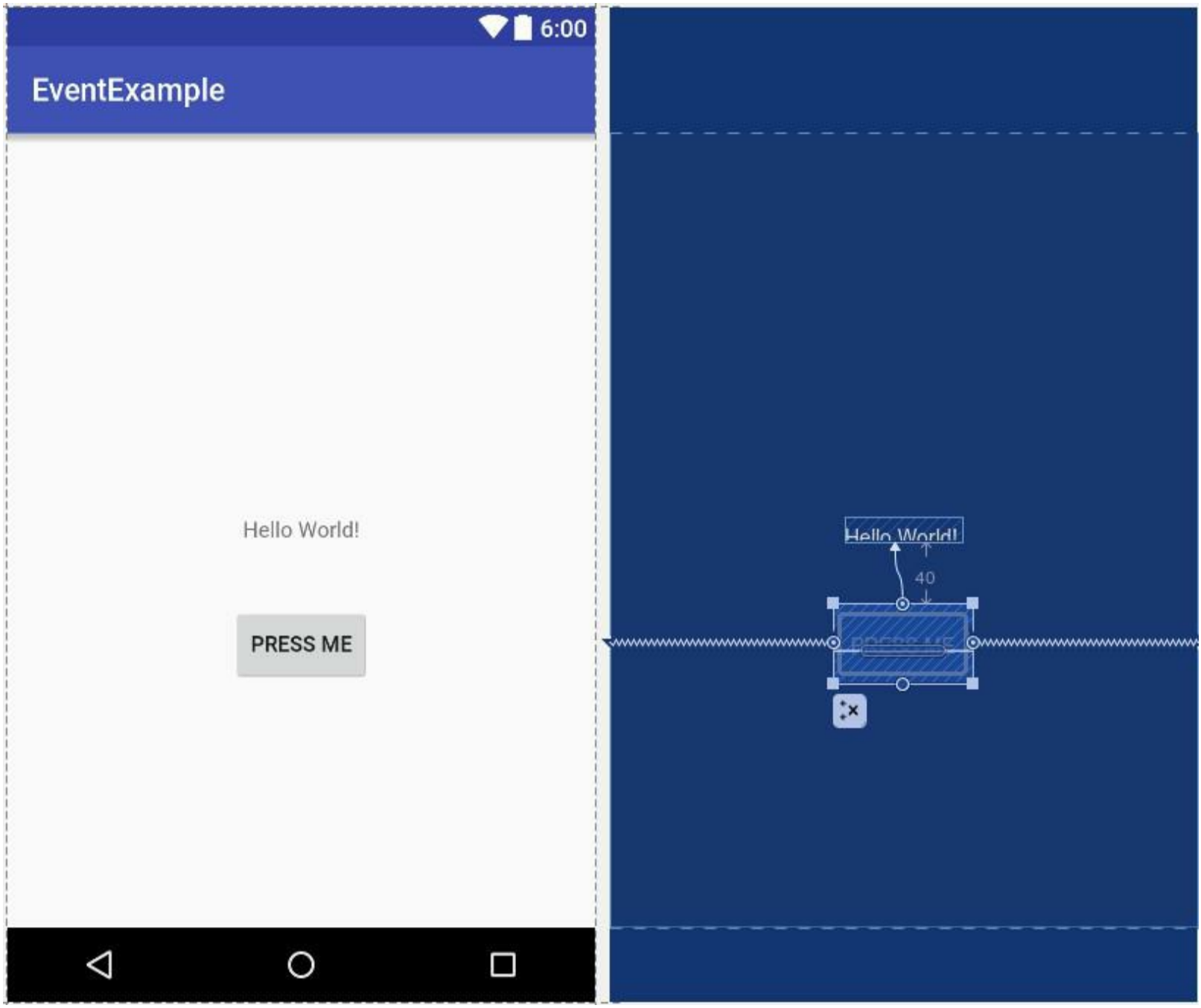

Figure 24-1

Locate and select the *activity_event_example.xml* file created by Android Studio (located in the Project tool window under *app -> res -> layouts*) and double-click on it to load it into the Layout Editor tool.

Make sure that Autoconnect is enabled, then drag a Button widget from the palette and move it so that it is positioned in the horizontal center of the layout and beneath the existing TextView widget. When correctly positioned, drop the widget into place so that appropriate constraints are added by the autoconnect system.

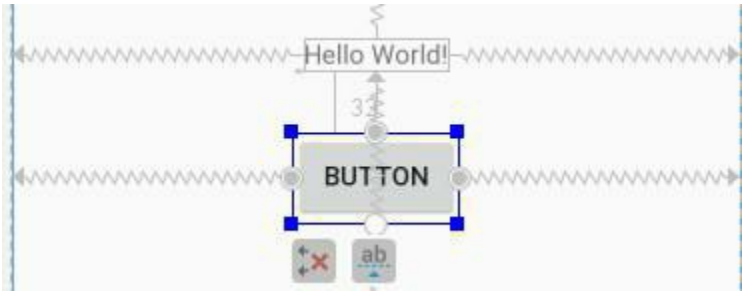

Figure 24-2

With the Button widget selected, use the Properties panel to set the text property to Press Me. Using the square red button located in the top right-hand corner of the Layout Editor (Figure 24-3), display the warnings list dialog and click the link to extract the text string on the button to a resource named *press_me*:

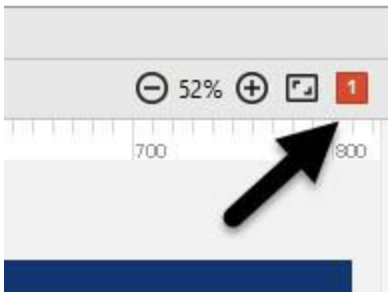

Figure 24-3

Select the "Hello World!" TextView widget and use the Properties panel to set the ID to *statusText*. Repeat this step to change the ID of the Button widget to *myButton*.

With the user interface layout now completed, the next step is to register the event listener and callback method.

24.6 The Event Listener and Callback Method

For the purposes of this example, an *onClickListener* needs to be registered for the *myButton* view. This is achieved by making a call to the *setOnClickListener()* method of the button view, passing through a new *onClickListener* object as an argument and implementing the *onClick()* callback method. Since this is a task that only needs to be performed when the activity is created, a good location is the *onCreate()* method of the EventExampleActivity class.

If the *EventExampleActivity.java* file is already open within an editor session, select it by clicking on the tab in the editor panel. Alternatively locate it within the Project tool window by navigating to (*app -> java -> com.ebookfrenzy.eventexample -> EventExampleActivity*) and double-click on it to load it into the code editor. Once loaded, locate the template *onCreate()* method and modify it to obtain a reference to the button view, register the event listener and implement the *onClick()* callback method:

```
package com.ebookfrenzy.eventexample;

import android.support.v7.app.AppCompatActivity;
```

```
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class EventExample extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_event_example);
        Button button = (Button)findViewById(R.id.myButton);

        button.setOnClickListener(
                new Button.OnClickListener() {
                    public void onClick(View v) {

                    }
                }
        );

    }
.
.
.
}
```

The above code has now registered the event listener on the button and implemented the *onClick()* method. If the application were to be run at this point, however, there would be no indication that the event listener installed on the button was working since there is, as yet, no code implemented within the body of the *onClick()* callback method. The goal for the example is to have a message appear on the TextView when the button is clicked, so some further code changes need to be made:

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

       Button button = (Button)findViewById(R.id.myButton);

      button.setOnClickListener(
              new Button.OnClickListener() {
                      public void onClick(View v) {
                        TextView statusText =
                           (TextView)findViewById(R.id.statusText);
                        statusText.setText("Button clicked");
                      }
              }
       );
}
```

Complete this phase of the tutorial by compiling and running the application on either an AVD emulator or physical Android device. On touching and releasing the button view (otherwise known as "clicking") the text view should change to display the "Button clicked" text.

24.7 Consuming Events

The detection of standard clicks (as opposed to long clicks) on views is a very simple case of event handling. The example will now be extended to include the detection of long click events which occur when the user clicks and holds a view on the screen and, in doing so, cover the topic of event consumption.

Consider the code for the *onClick()* method in the above section of this chapter. The callback is declared as *void* and, as such, does not return a value to the Android framework after it has finished executing.

The *onLongClick()* callback method of the *onLongClickListener* interface, on the other hand, is required to return a Boolean value to the Android framework. The purpose of this return value is to indicate to the Android runtime whether or not the callback has *consumed* the event. If the callback returns a *true* value, the event is discarded by the framework. If, on the other hand, the callback returns a *false* value the Android framework will consider the event still to be active and will consequently pass it along to the next matching event listener that is registered on the same view.

As with many programming concepts this is, perhaps, best demonstrated with an example. The first step is to add an event listener and callback method for long clicks to the button view in the example activity:

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

       Button button = (Button)findViewById(R.id.myButton);

      button.setOnClickListener(
              new Button.OnClickListener() {
                      public void onClick(View v) {
                        TextView statusText =
                      (TextView)findViewById(R.id.statusText);
                      statusText.setText("Button clicked");
                       }
              }
        );

        button.setOnLongClickListener(
                new Button.OnLongClickListener() {
                      public boolean onLongClick(View v) {
                        TextView statusText =
                            (TextView)findViewById(R.id.statusText);
                        statusText.setText("Long button click");
                        return true;
                }
              }
        );
    }
```

Clearly, when a long click is detected, the *onLongClick()* callback method will display "Long button click" on the text view. Note, however, that the callback method also returns a value of *true* to indicate that it has consumed the event. Run the application and press and hold the Button view until the "Long button click" text appears in the text view. On releasing the button, the text view continues to display the "Long button click" text indicating that the *onClick()* callback method was not called.

Next, modify the code such that the *onLongClick()* method now returns a *false* value:

```
button.setOnLongClickListener(
                        new Button.OnLongClickListener() {
                                public boolean onLongClick(View v) {
                                        TextView myTextView =
(TextView)findViewById(R.id.myTextView);
                                myTextView.setText("Long button click");
                                return false;
                                }
                        }
        );
```

Once again, compile and run the application and perform a long click on the button until the long click message appears. Upon releasing the button this time, however, note that the *onClick()* callback is also triggered and the text changes to "Button click". This is because the *false* value returned by the *onLongClick()* callback method indicated to the Android framework that the event was not consumed by the method and was eligible to be passed on to the next registered listener on the view. In this case, the runtime ascertained that the onClickListener on the button was also interested in events of this type and subsequently called the *onClick()* callback method.

24.8 **Summary**

A user interface is of little practical use if the views it contains do not do anything in response to user interaction. Android bridges the gap between the user interface and the back end code of the application through the concepts of event listeners and callback methods. The Android View class defines a set of event listeners, which can be registered on view objects. Each event listener also has associated with it a callback method.

When an event takes place on a view in a user interface, that event is placed into an event queue and handled on a first in, first out basis by the Android runtime. If the view on which the event took place has registered a listener that matches the type of event, the corresponding callback method is called. The callback method then performs any tasks required by the activity before returning. Some callback methods are required to return a Boolean value to indicate whether the event needs to be passed on to any other event listeners registered on the view or discarded by the system.

Having covered the basics of event handling, the next chapter will explore in some depth the topic of touch events with a particular emphasis on handling multiple touches.

25. A Guide to using Instant Run in Android Studio

Now that some of the basic concepts of Android development using Android Studio have been covered, now is a good time to introduce the Android Studio Instant Run feature. As all experienced developers know, every second spent waiting for an app to compile and run is time better spent writing and refining code.

25.1 Introducing Instant Run

Prior to the introduction of Instant Run, each time a change to a project needed to be tested Android Studio would recompile the code, convert it to Dex format, generate the APK package file and install it on the device or emulator. Having performed these steps the app would finally be launched ready for testing. Even on a fast development system this is a process that takes a considerable amount of time to complete. It is not uncommon for it to take a minute or more for this process to complete for a large application.

Instant Run, in contrast, allows many code and resource changes within a project to be reflected nearly instantaneously within the app while it is already running on a device or emulator session.

Consider, for the purposes of an example, an app being developed in Android Studio which has already been launched on a device or emulator. If changes are made to resource settings or the code within a method, Instant Run will push the updated code and resources to the running app and dynamically "swap" the changes. The changes are then reflected in the running app without the need to build, deploy and relaunch the entire app. In many cases, this allows changes to be tested in a fraction of the time it would take without Instant Run.

25.2 Understanding Instant Run Swapping Levels

Not all project changes are fully supported by Instant Run and different changes result in a different level of "swap" being performed. There are three levels of Instant Run support, referred to as hot, warm and cold swapping:

- **Hot Swapping** – Hot swapping occurs when the code within an existing method implementation is changed. The new method implementation is used next time it is called by the app. A hot swap occurs instantaneously and, if configured, is accompanied by a toast message on the device screen that reads "Applied code changes without activity restart".
- **Warm Swapping** – When a change is made to a resource file of the project (for example a layout change or the modification of a string or color resource setting) an Instant Run warm swap is performed. A warm swap involves the restarting of the currently running activity. Typically the screen will flicker as the activity restarts. A warm swap is reported on the device screen by a toast message that reads "Applied changes, restarted activity".
- **Cold Swapping** – Structural code changes such as the addition of a new method, a change to the signature of an existing method or a change to the class hierarchy of the project triggers a cold swap in which the entire app is restarted. In some conditions, such as the addition of new image resources to the project, the application package file (APK) will also be reinstalled during the swap.

25.3 Enabling and Disabling Instant Run

Instant Run is enabled and disabled via the Android Studio Settings screen. To view the current settings begin by selecting the *File -> Settings...* menu option. Within the Settings dialog select the *Build, Execution, Deployment* entry in the left-hand panel followed by *Instant Run* as shown in Figure 25-1:

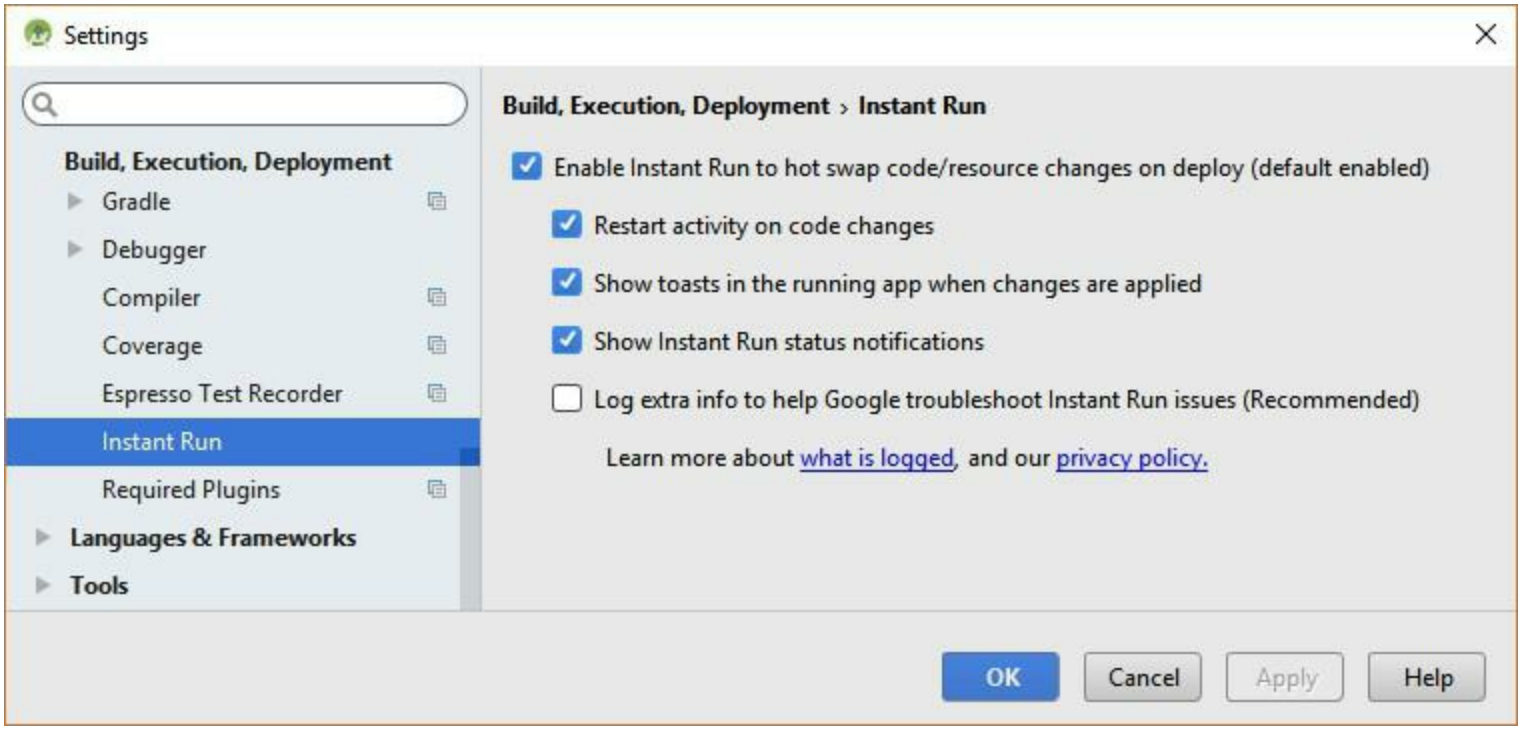

Figure 25-1

The options provided in the panel apply only to the current project. Each new project will start with the default settings. The first option controls whether or not Instant Run is enabled by default each time the project is opened in Android Studio. The *Restart activity on code changes* option forces Instant Run to restart the current activity every time a change is made, regardless of whether a hot swap could have been performed. The next option controls whether or not messages are displayed within Android Studio and the app indicating the type of Instant Run level performed. Finally, an option is provided to allow additional log information to be provided to Google to help in improving the reliability of the Instant Run feature.

25.4 Using Instant Run

When a project has been loaded into Android Studio, but is not yet running on a device or emulator, it can be launched as usual using either the run (marked A in Figure 25-2) or debug (B) button located in the toolbar:

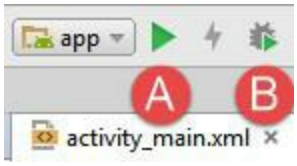

Figure 25-2

After the app has launched and is running, Android Studio will indicate the availability of Instant Run by enabling the *Apply Changes* button located immediately to the right of the run button as highlighted in Figure 25-3:

Figure 25-3

When it is enabled, clicking on the Apply Changes button will use Instant Run to update the running app.

25.5 An Instant Run Tutorial

Begin by launching Android Studio and creating a new project. Within the *New Project* dialog, enter *InstantRunDemo* 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 23: Android 6.0 (Marshmallow). Continue to proceed through the screens, requesting the creation of a Basic Activity named *InstantRunDemoActivity* with a corresponding layout named *activity_instant_run_demo*.

Click on the *Finish* button to initiate the project creation process.

25.6 Triggering an Instant Run Hot Swap

Begin by clicking on the run button and selecting a suitable emulator or physical device as the run target. After clicking the run button, track the amount of time before the example app appears on the device or emulator.

Once running, click on the action button (the button displaying an envelope icon located in the lower right-hand corner of the screen). Note that a Snackbar instance appears displaying text which reads "Replace with your own action" as shown in Figure 25-4:

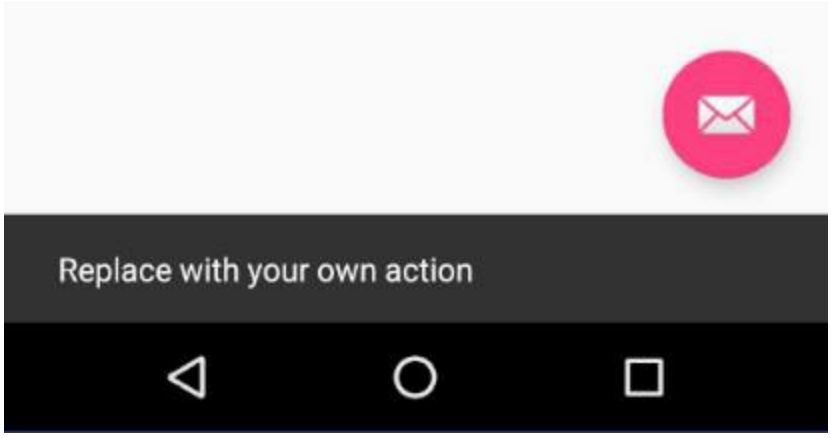

Figure 25-4

Once the app is running, the Apply Changes button should have been enabled indicating the availability of Instant Run. To see this in action, edit the *InstantRunDemoActivity.java* file, locate the *onCreate* method and modify the action code so that a different message is displayed when the action button is selected:

```
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Instant Run is Amazing!",
```

```
                                        Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
```

With the code change implemented, click on the Apply Changes button and note that the toast message appears within a few seconds indicating the app has been updated. Tap the action button and note that the new message is now displayed in the Snackbar. Instant Run has successfully performed a hot swap.

25.7 Triggering an Instant Run Warm Swap

Any resource change should result in Instant Run performing a warm swap. Within Android Studio select the *app -> res -> layout -> content_instant_run_demo.xml* layout file. With the Layout Editor tool in Design mode, select the ConstraintLayout view within the Component Tree panel, switch the Properties tool window to expert mode and locate the *background* property. Click on the button displaying three dots next to the background property text field, select a color from the Resources dialog and click on *OK*. With the background color of the activity content modified, click on the Apply Changes button once again. This time a warm swap will be performed and the currently running activity should quickly restart to adopt the new background color setting.

25.8 Triggering an Instant Run Cold Swap

As previously described, a cold swap triggers a complete restart of the running app. To experience an Instant Run cold swap, edit the *InstantRunDemoActivity.java* file and add a new method after the *onCreate* method as follows:

```
    public void demoMethod() {

    }
```

Click on the Apply Changes button and note that the app now has to terminate and restart to accommodate the addition of the new method. Within Android Studio a message will appear indicating that the app was restarted due to a method being added:

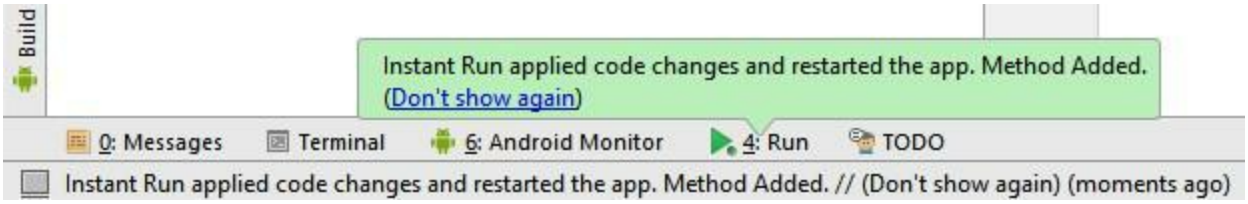

Figure 25-5

25.9 The Run Button

When no apps are running, the run button appears as shown in Figure 25-3. When an app is running, however, an additional green dot appears in the bottom right-hand corner of the button as shown in Figure 25-6 below:

Figure 25-6

Although the Instant Run feature has improved significantly since being introduced it can still occasionally produce unexpected results when performing hot or warm swaps. It is worth being aware, therefore, that clicking the run button when an app is currently running will force a cold swap

to be performed regardless of the changes made to the project.

25.10 **Summary**

Instant Run is a feature introduced with Android Studio 2 designed to significantly accelerate the code, build and run cycle. Using a swapping mechanism, Instant Run is able to push updates to the running application, in many cases without the need to re-install or even restart the app. Instant Run provides a number of different levels of support depending on the nature of the modification being applied to the project. These levels are referred to as hot, warm and cold swapping. This chapter has introduced the concepts of Instant Run and worked through some demonstrations of the different levels of swapping.