35. Working with the Floating Action Button and Snackbar

One of the objectives of this chapter is to provide an overview of the concepts of material design. Originally introduced as part of Android 5.0, material design is a set of design guidelines that dictate how the Android user interface, and that of the apps running on Android, appear and behave.

As part of the implementation of the material design concepts, Google also introduced the Android Design Support Library. This library contains a number of different components that allow many of the key features of material design to be built into Android applications. Two of these components, the floating action button and Snackbar, will also be covered in this chapter prior to introducing many of the other components in subsequent chapters.

35.1 The Material Design

The overall appearance of the Android environment is defined by the principles of material design. Material design was created by the Android team at Google and dictates that the elements that make up the user interface of Android and the apps that run on it appear and behave in a certain way in terms of behavior, shadowing, animation and style. One of the tenets of the material design is that the elements of a user interface appear to have physical depth and a sense that items are constructed in layers of physical material. A button, for example, appears to be raised above the surface of the layout in which it resides through the use of shadowing effects. Pressing the button causes the button to flex and lift as though made of a thin material that ripples when released.

Material design also dictates the layout and behavior of many standard user interface elements. A key example is the way in which the app bar located at the top of the screen should appear and the way in which it should behave in relation to scrolling activities taking place within the main content of the activity.

In fact, material design covers a wide range of areas from recommended color styles to the way in which objects are animated. A full description of the material design concepts and guidelines can be found online at the following link and is recommended reading for all Android developers:

https://www.google.com/design/spec/material-design/introduction.html

35.2 The Design Library

Many of the building blocks needed to implement Android applications that adopt the principles of material design are contained within the Android Design Support Library. This library contains a collection of user interface components that can be included in Android applications to implement much of the look, feel and behavior of material design. Two of the components from this library, the floating action button and Snackbar, will be covered in this chapter, while others will be introduced in later chapters.

35.3 The Floating Action Button (FAB)

The floating action button is a button which appears to float above the surface of the user interface of an app and is generally used to promote the most common action within a user interface screen. A floating action button might, for example, be placed on a screen to allow the user to add an entry to a

list of contacts or to send an email from within the app. Figure 35-1, for example, highlights the floating action button that allows the user to add a new contact within the standard Android Contacts app:

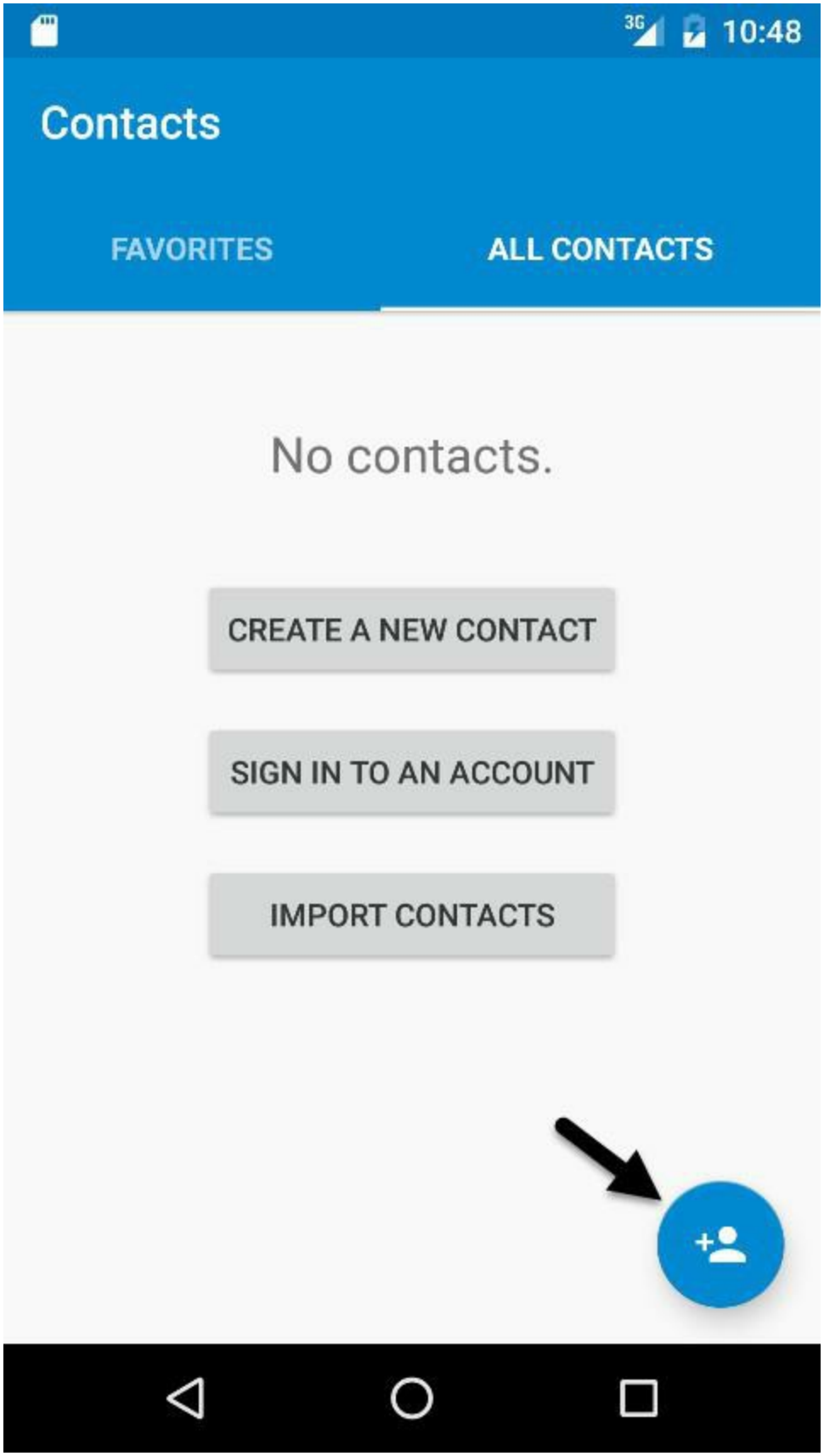

Figure 35-1

To conform with the material design guidelines, there are a number of rules that should be followed when using floating action buttons. Floating action buttons must be circular and can be either 56 x 56dp (Default) or 40 x 40dp (Mini) in size. The button should be positioned a minimum of 16dp from

the edge of the screen on phones and 24dp on desktops and tablet devices. Regardless of the size, the button must contain an interior icon that is 24x24dp in size and it is recommended that each user interface screen have only one floating action button.

Floating action buttons can be animated or designed to morph into other items when touched. A floating action button could, for example, rotate when tapped or morph into another element such as a toolbar or panel listing related actions.

35.4 The Snackbar

The Snackbar component provides a way to present the user with information in the form of a panel that appears at the bottom of the screen as shown in Figure 35-2. Snackbar instances contain a brief text message and an optional action button which will perform a task when tapped by the user. Once displayed, a Snackbar will either timeout automatically or can be removed manually by the user via a swiping action. During the appearance of the Snackbar the app will continue to function and respond to user interactions in the normal manner.

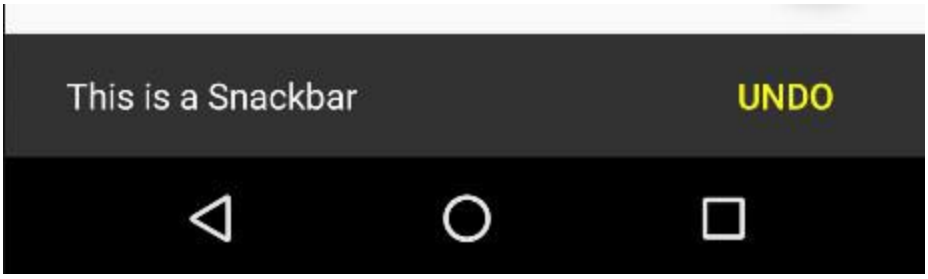

Figure 35-2

In the remainder of this chapter an example application will be created that makes use of the basic features of the floating action button and Snackbar to add entries to a list of items.

35.5 Creating the Example Project

Create a new project in Android Studio, entering *FabExample* 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).

Although it is possible to manually add a floating action button to an activity, it is much easier to use the Basic Activity template which includes a floating action button by default. Continue to proceed through the screens, therefore, requesting the creation of a Basic Activity named *FabExampleActivity* with corresponding layout and menu files named *activity_fab_example* and *menu_fab_example* respectively.

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

35.6 Reviewing the Project

Since the Basic Activity template was selected, the activity contains two layout files. The *activity_fab_example.xml* file consists of a CoordinatorLayout manager containing entries for an app bar, a toolbar and a floating action button.

The *content_fab_example.xml* file represents the layout of the content area of the activity and contains a ConstraintLayout instance and a TextView. This file is embedded into the *activity_fab_example.xml* file via the following include directive:

```
<include layout="@layout/content_fab_example" />
```

The floating action button element within the *activity_fab_example.xml* file reads as follows:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:src="@android:drawable/ic_dialog_email" />
```

This declares that the button is to appear in the bottom right-hand corner of the screen with margins represented by the *fab_margin* identifier in the *values/dimens.xml* file (which in this case is set to 16dp). The XML further declares that the interior icon for the button is to take the form of the standard drawable built-in email icon.

The blank template has also configured the floating action button to display a Snackbar instance when tapped by the user. The code to implement this can be found in the *onCreate()* method of the *FabExampleActivity.java* file and reads as follows:

```
FloatingActionButton fab =
              (FloatingActionButton) findViewById(R.id.fab);

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

The code obtains a reference to the floating action button via the button's ID and adds to it an onClickListener handler to be called when the button is tapped. This method simply displays a Snackbar instance configured with a message but no actions.

Finally, open the module level *build.gradle* file (*Gradle Scripts -> build.gradle (Module: App)*) and note that the Android design support library has been added as a dependency:

```
compile 'com.android.support:design:25.1.0'
```

When the project is compiled and run the floating action button will appear at the bottom of the screen as shown in Figure 35-3:

Figure 35-3

Tapping the floating action button will trigger the onClickListener handler method causing the

Snackbar to appear at the bottom of the screen:

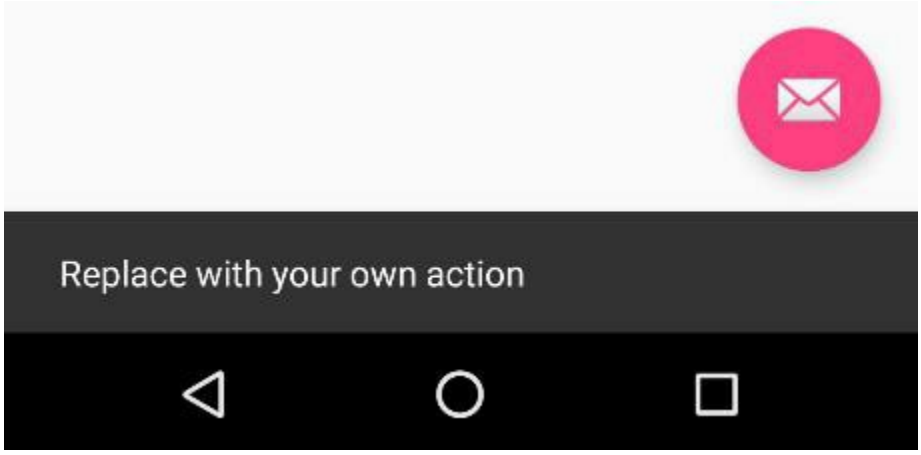

Figure 35-4

When the Snackbar appears on a narrower device (as is the case in Figure 35-4 above) note that the floating action button is moved up to make room for the Snackbar to appear. This is handled for us automatically by the CoordinatorLayout container in the *activity_fab_example.xml* layout resource file.

35.7 Changing the Floating Action Button

Since the objective of this example is to configure the floating action button to add entries to a list, the email icon currently displayed on the button needs to be changed to something more indicative of the action being performed. The icon that will be used for the button is named *ic_add_entry.png* and can be found in the *project_icons* folder of the sample code download available from the following URL:

http://www.ebookfrenzy.com/retail/androidstudio23/index.php

Locate this image in the file system navigator for your operating system and copy the image file. Right-click on the *app -> res -> drawable* entry in the Project tool window and select Paste from the menu to add the file to the folder:

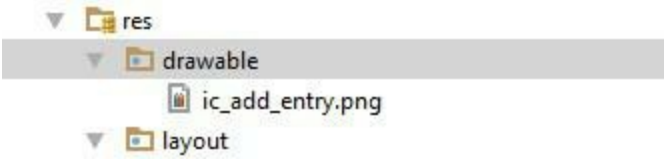

Figure 35-5

Next, edit the *activity_fab_example.xml* file and change the image source for the icon from *@android:drawable/ic_dialog_email* to *@drawable/ic_add_entry* as follows:

```
<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:src="@drawable/ic_add_entry" />
```

Within the layout preview, the interior icon for the button will have changed to a plus sign.

The background color of the floating action button is defined by the *accentColor* property of the prevailing theme used by the application. The color assigned to this value is declared in the *colors.xml* file located under *app -> res -> values* in the Project tool window. Instead of editing this

XML file directly a better approach is to use the Android Studio Theme Editor.

Select the *Tools -> Android -> Theme Editor* menu option to display the Theme Editor as illustrated in Figure 35-6:

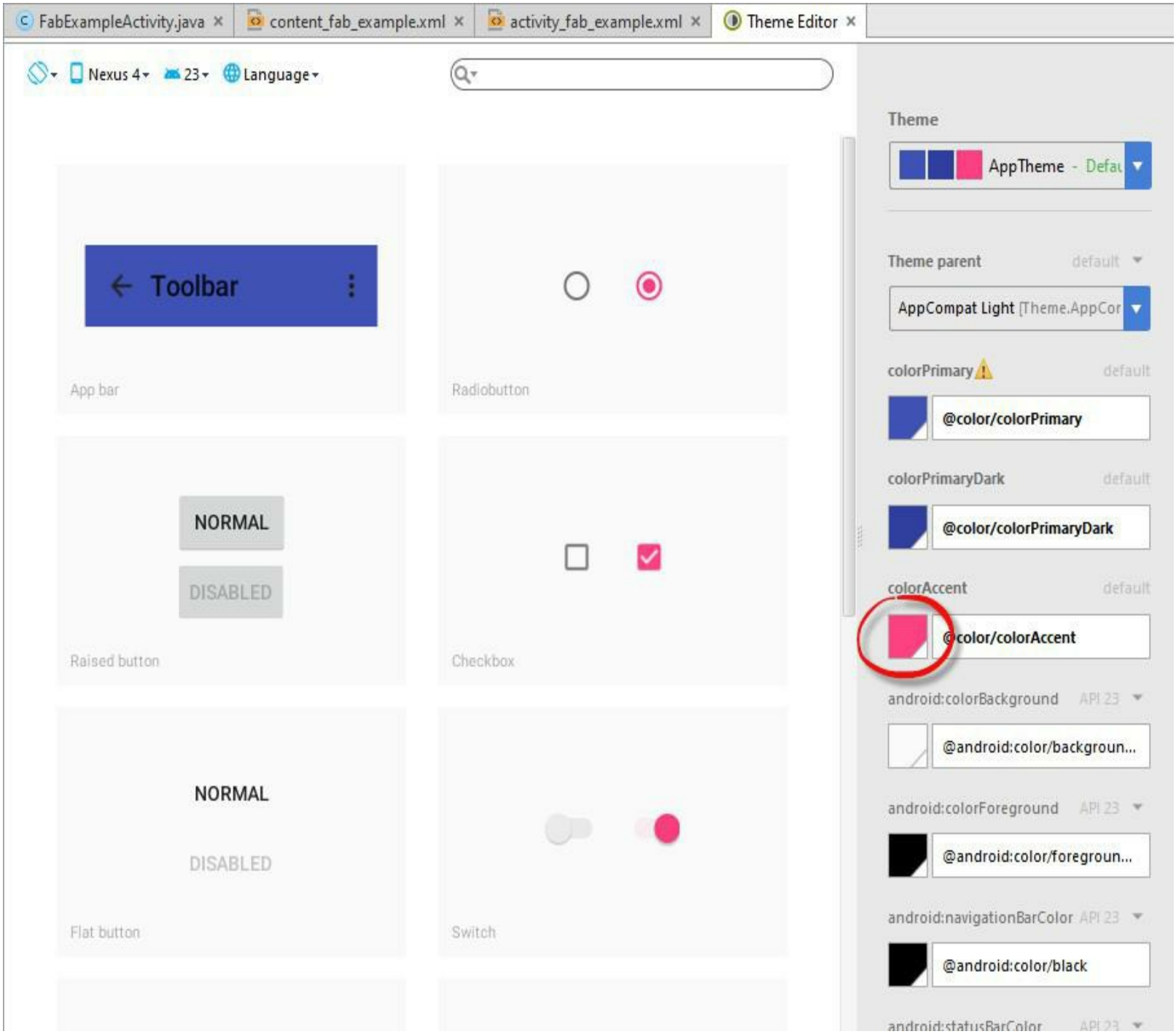

Figure 35-6

Click on the color swatch for the *colorAccent* setting (highlighted in the figure above) to display the color resource dialog. Within the color resource dialog, enter *holo_orange_light* into the search field and select the color from the list:

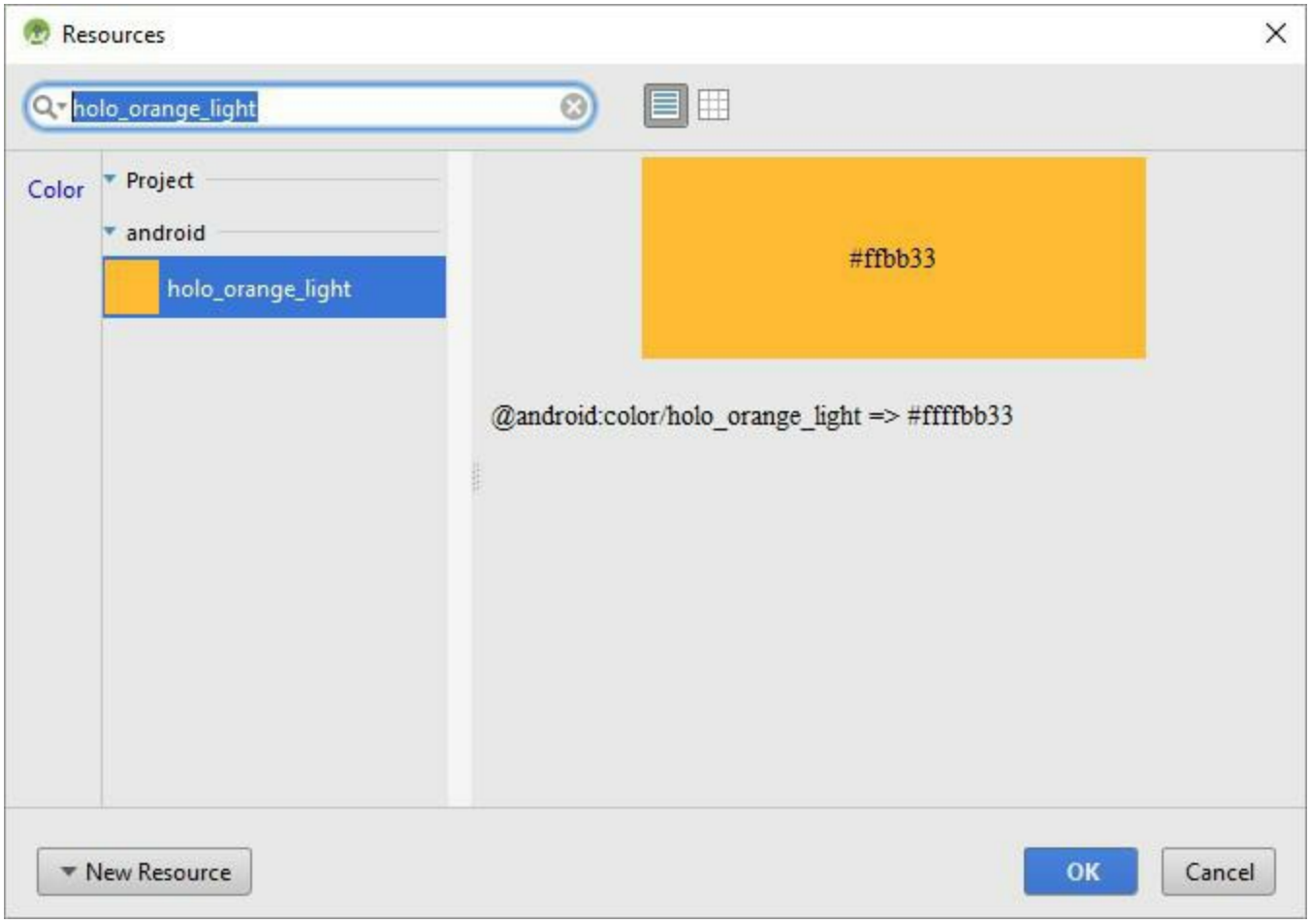

Figure 35-7

Click on the *OK* button to apply the new accentColor setting and return to the *activity_fab_example.xml* and verify that the floating action button now appears with an orange background.

35.8 Adding the ListView to the Content Layout

The next step in this tutorial is to add the ListView instance to the *content_fab_example.xml* file. The ListView class provides a way to display items in a list format and can be found in the *Containers* section of the Layout Editor tool palette.

Load the *content_fab_example.xml* file into the Layout Editor tool, select Design mode if necessary, and select and delete the default TextView object. Locate the ListView object in the Containers category of the palette and, with autoconnect mode enabled, drag and drop it onto the center of the layout canvas. Select the ListView object and change the ID to *listView* within the Properties tool window. The Layout Editor should have sized the ListView to fill the entire container and established constraints on all four edges as illustrated in Figure 35-8:

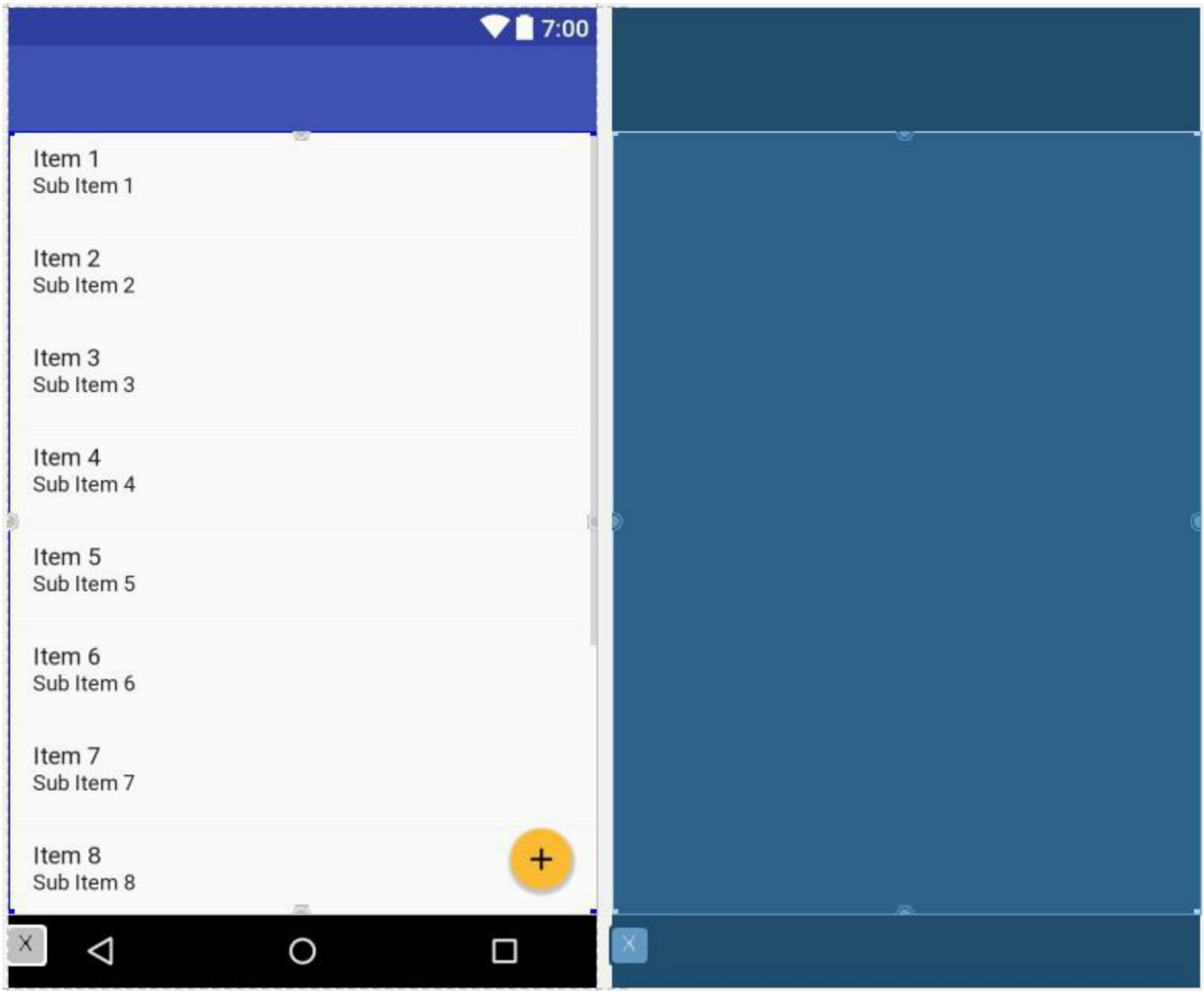

Figure 35-8

35.9 Adding Items to the ListView

Each time the floating action button is tapped by the user, a new item will be added to the ListView in the form of the prevailing time and date. To achieve this, some changes need to be made to the *FabExampleActivity.java* file.

Begin by modifying the *onCreate()* method to obtain a reference to the ListView instance and to initialize an adapter instance to allow us to add items to the list in the form of an array:

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

```
import java.util.ArrayList;

public class FabExampleActivity extends AppCompatActivity {

    ArrayList<String> listItems = new ArrayList<String>();
    ArrayAdapter<String> adapter;
    private ListView myListView;

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

        myListView = (ListView) findViewById(R.id.listView);

        adapter = new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1,
                listItems);
        myListView.setAdapter(adapter);

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

The ListView needs an array of items to display, an adapter to manage the items in that array and a layout definition to dictate how items are to be presented to the user.

In the above code changes, the items are stored in an ArrayList instance assigned to an adapter that takes the form of an ArrayAdapter. The items added to the list will be displayed in the ListView using the *simple_list_item_1* layout, a built-in layout that is provided with Android to display simple string based items in a ListView instance.

Next, edit the onClickListener code for the floating action button to display a different message in the Snackbar and to call a method to add an item to the list:

```
FloatingActionButton fab = (FloatingActionButton)
                                findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        addListItem();
        Snackbar.make(view, "Item added to list",
              Snackbar.LENGTH_LONG)
```

```
                .setAction("Action", null).show();

        }
    });
```

Remaining within the *FabExampleActivity.java* file, add the *addListItem()* method as follows:

```
package com.ebookfrenzy.fabexample;
.
.
.
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

public class FabExampleActivity extends AppCompatActivity {
.
.
    private void addListItem() {

        SimpleDateFormat dateformat =
                new SimpleDateFormat("HH:mm:ss MM/dd/yyyy",
                        Locale.US);
        listItems.add(dateformat.format(new Date()));
        adapter.notifyDataSetChanged();
    }
.
.
}
```

The code in the *addListItem()* method identifies and formats the current date and time and adds it to the list items array. The array adapter assigned to the ListView is then notified that the list data has changed, causing the ListView to update to display the latest list items.

Compile and run the app and test that tapping the floating action button adds new time and date entries to the ListView, displaying the Snackbar each time as shown in Figure 35-9:

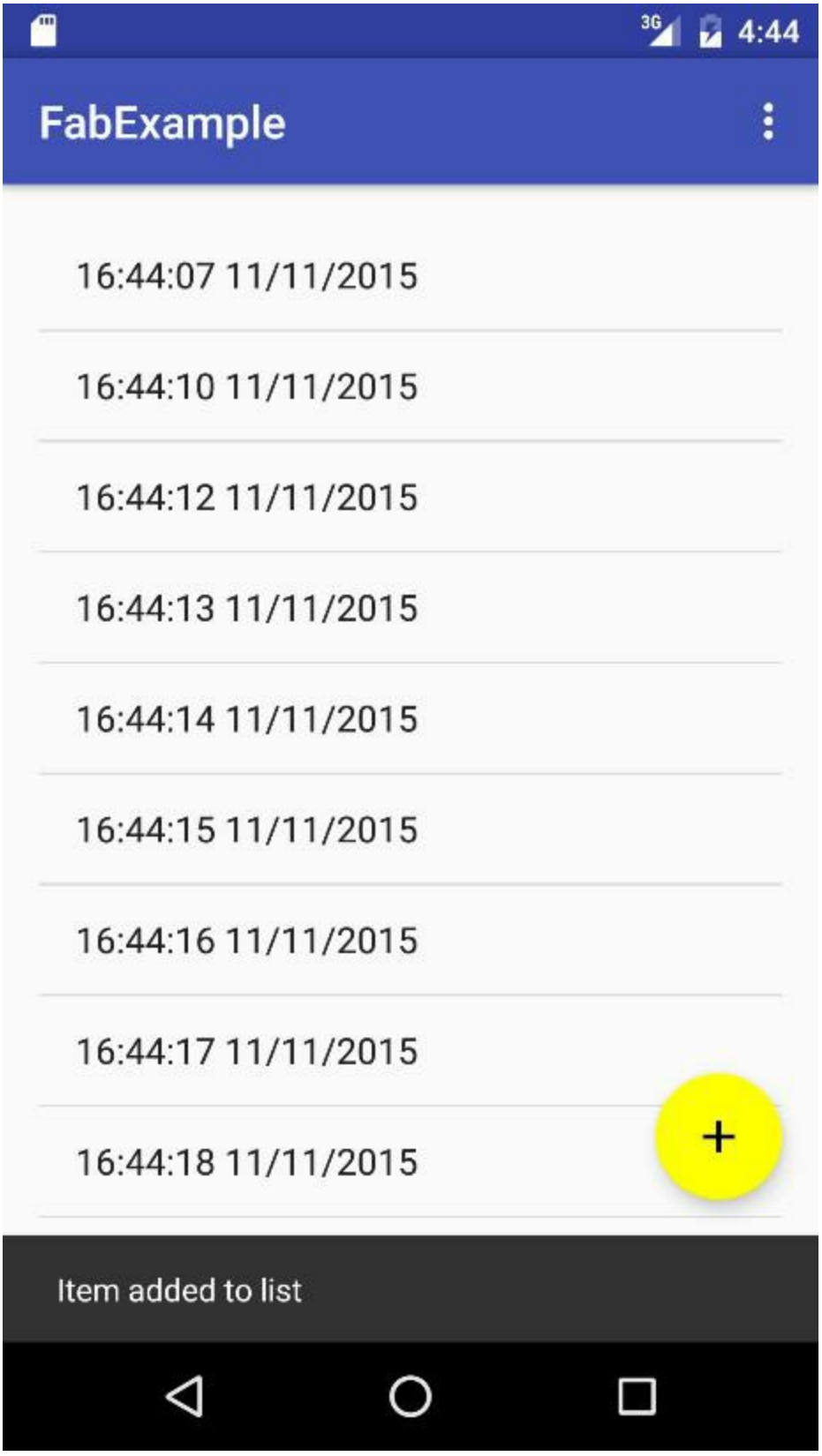

Figure 35-9

35.10 Adding an Action to the Snackbar

The final task in this project is to add an action to the Snackbar that allows the user to undo the most recent addition to the list. Edit the *FabExampleActivity.java* file and modify the Snackbar creation code to add an action titled "Undo" configured with an onClickListener named *undoOnClickListener*:

```
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        addListItem();
```

```
            Snackbar.make(view, "Item added to list",
                    Snackbar.LENGTH_LONG)
                      .setAction("Undo", undoOnClickListener).show();


        }
    });
```

Within the *FabExampleActivity.java* file add the listener handler:

```
    View.OnClickListener undoOnClickListener = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            listItems.remove(listItems.size() -1);
            adapter.notifyDataSetChanged();
            Snackbar.make(view, "Item removed", Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show();
        }
    };
```

The code in the onClick method identifies the location of the last item in the list array and removes it from the list before triggering the list view to perform an update. A new Snackbar is then displayed indicating that the last item has been removed from the list.

Run the app once again and add some items to the list. On the final addition, tap the Undo button in the Snackbar (Figure 35-10) to remove the last item from the list:

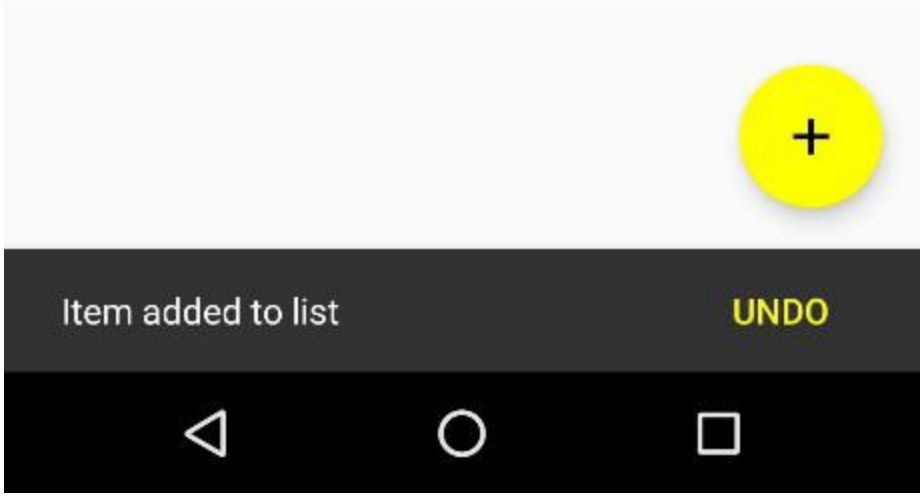

Figure 35-10

It is also worth noting that the Undo button appears using the same color assigned to the accentColor property via the Theme Editor earlier in the chapter.

35.11 Summary

This chapter has provided a general overview of material design, the floating action button and Snackbar before working through an example project that makes use of these features

Both the floating action button and the Snackbar are part of the material design approach to user interface implementation in Android. The floating action button provides a way to promote the most common action within a particular screen of an Android application. The Snackbar provides a way for an application to both present information to the user and also allow the user to take action upon it.

36. Creating a Tabbed Interface using the TabLayout Component

The previous chapter outlined the concept of material design in Android and introduced two of the components provided by the design support library in the form of the floating action button and the Snackbar. This chapter will demonstrate how to use another of the design library components, the TabLayout, which can be combined with the ViewPager class to create a tab based interface within an Android activity.

36.1 An Introduction to the ViewPager

Although not part of the design support library, the ViewPager is a useful companion class when used in conjunction with the TabLayout component to implement a tabbed user interface. The primary role of the ViewPager is to allow the user to flip through different pages of information where each page is most typically represented by a layout fragment. The fragments that are associated with the ViewPager are managed by an instance of the FragmentPagerAdapter class.

At a minimum the pager adapter assigned to a ViewPager must implement two methods. The first, named *getCount()*, must return the total number of page fragments available to be displayed to the user. The second method, *getItem()*, is passed a page number and must return the corresponding fragment object ready to be presented to the user.

36.2 An Overview of the TabLayout Component

As previously discussed, TabLayout is one of the components introduced as part of material design and is included in the design support library. The purpose of the TabLayout is to present the user with a row of tabs which can be selected to display different pages to the user. The tabs can be fixed or scrollable, whereby the user can swipe left or right to view more tabs than will currently fit on the display. The information displayed on a tab can be text-based, an image or a combination of text and images. Figure 36-1, for example, shows the tab bar for the Android phone app consisting of three tabs displaying images:

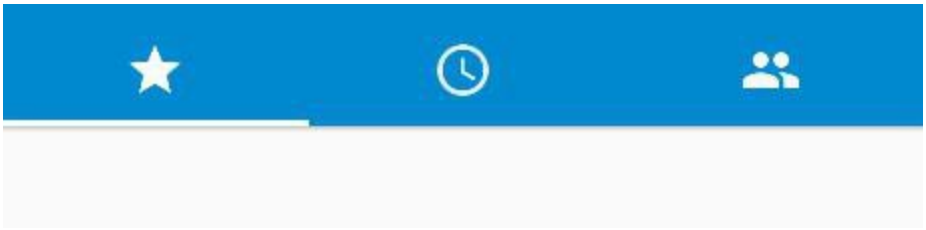

Figure 36-1

Figure 36-2, on the other hand, shows a TabLayout configuration consisting of four tabs displaying text in a scrollable configuration:

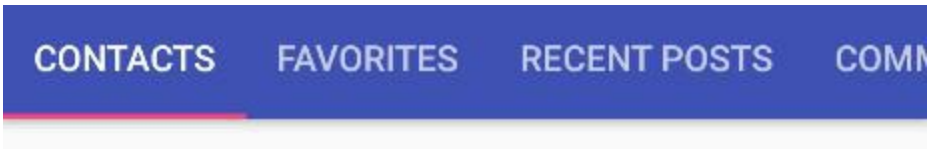

Figure 36-2

The remainder of this chapter will work through the creation of an example project that demonstrates

the use of the TabLayout component together with a ViewPager and four fragments.

36.3 Creating the TabLayoutDemo Project

Create a new project in Android Studio, entering *TabLayoutDemo* 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 through the configuration screens requesting the creation of a Basic Activity named *TabLayoutDemoActivity* with corresponding layout and menu files named *activity_tab_layout_demo* and *menu_tab_layout_demo* respectively. Click on the *Finish* button to initiate the project creation process.

Once the project has been created, load the *content_tab_layout_demo.xml* file into the Layout Editor tool, select "Hello World" TextView object, and then delete it.

36.4 Creating the First Fragment

Each of the tabs on the TabLayout will display a different fragment when selected. Create the first of these fragments by right-clicking on the *app -> java -> com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Fragment -> Fragment (Blank)* option. In the resulting dialog, enter *Tab1Fragment* into the *Fragment Name:* field and *fragment_tab1* into the *Fragment Layout Name:* field. Enable the *Create layout XML?* option and disable both the *Include fragment factory methods?* and *Include interface callbacks?* options before clicking on the *Finish* button to create the new fragment:

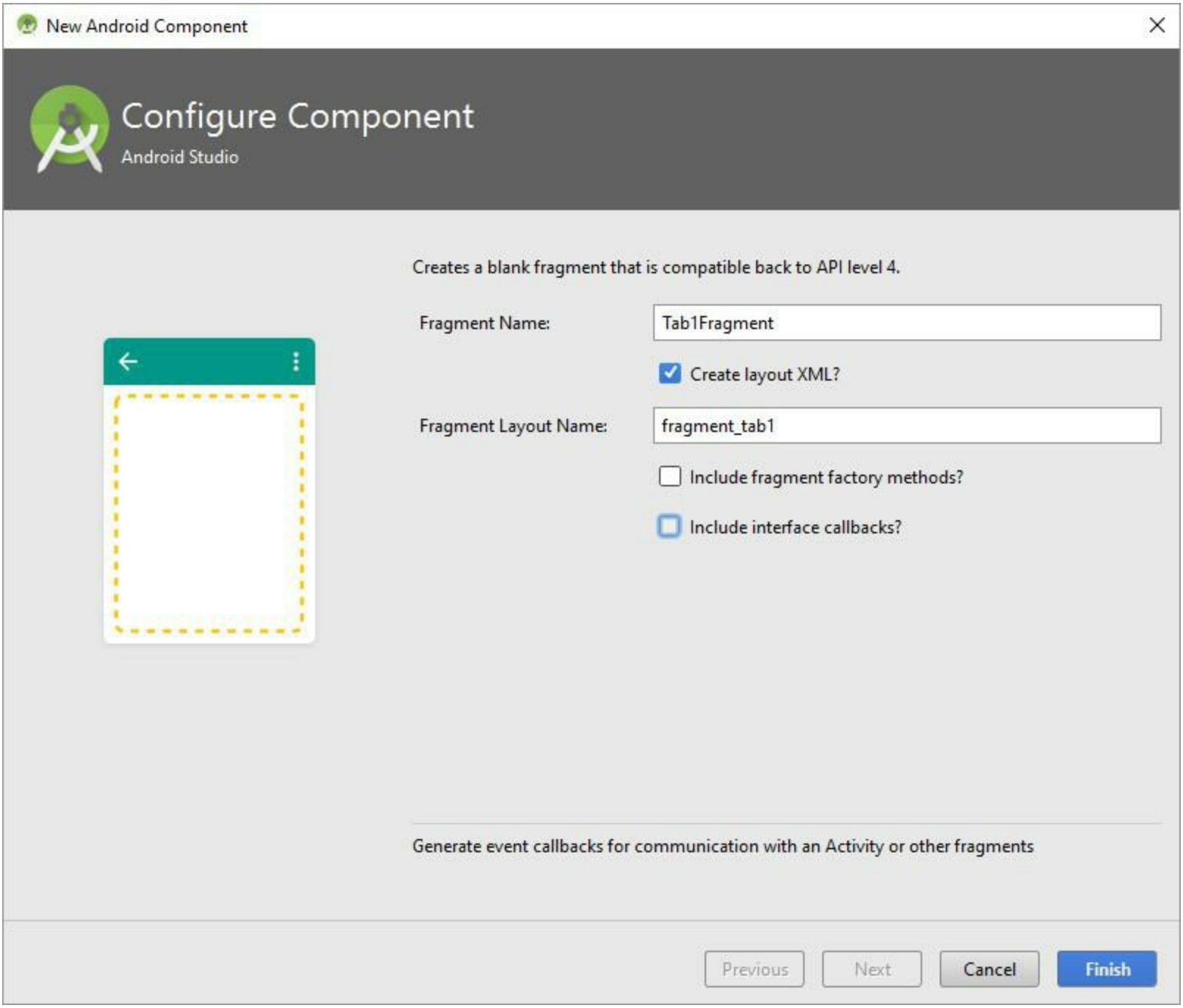

Figure 36-3

Load the newly created *fragment_tab1.xml* file (located under *app -> res -> layout*) into the Layout Editor tool, right-click on the FrameLayout entry in the Component Tree panel and select the *Convert FrameLayout to ConstraintLayout* menu option. In the resulting dialog, verify that all conversion options are selected before clicking on OK.

Once the layout has been converted to a ConstraintLayout, delete the TextView from the layout. From the Palette, locate the TextView widget and drag and drop it so that it is positioned in the center of the layout. Edit the text property on the object so that it reads "Tab 1 Fragment" and set the layout_width property to *wrap_content*. Extract the string to a resource named *tab_1_fragment*, at which point the layout should match that of Figure 36-4:

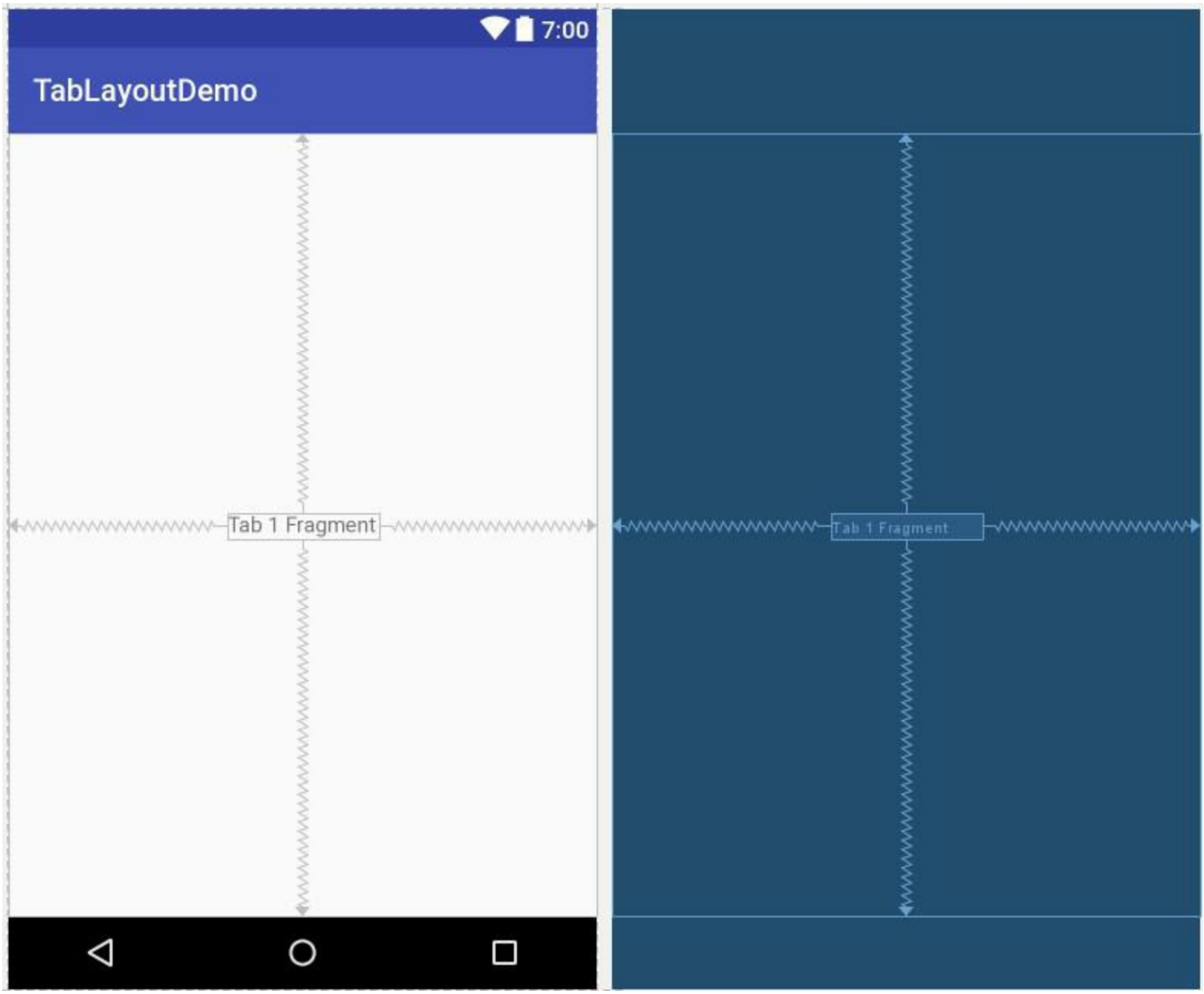

Figure 36-4

36.5 Duplicating the Fragments

So far, the project contains one of the four required fragments. Instead of creating the remaining three fragments using the previous steps it would be quicker to duplicate the first fragment. Each fragment consists of a layout XML file and a Java class file, each of which needs to be duplicated.

Right-click on the *fragment_tab1.xml* file in the Project tool window and select the Copy option from the resulting menu. Right-click on the *layout* entry, this time selecting the Paste option. In the resulting dialog, name the new layout file *fragment_tab2.xml* before clicking the *OK* button. Edit the new *fragment_tab2.xml* file and change the text on the Text View to "Tab 2 Fragment", following the usual steps to extract the string to a resource named *tab_2_fragment*.

To duplicate the Tab1Fragment class file, right-click on the class listed under *app -> java -> com.ebookfrenzy.tablayoutdemo* and select Copy. Right-click on the *com.ebookfrenzy.tablayoutdemo* entry and select Paste. In the Copy Class dialog, enter Tab2Fragment into the *New name:* field and click on OK. Edit the new *Tab2Fragment.java* file and change the *onCreateView()* method to inflate the *fragment_tab2* layout file:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_tab2, container, false);
}
```

Perform the above duplication steps twice more to create the fragment layout and class files for the remaining two fragments. On completion of these steps the project structure should match that of Figure 36-5:

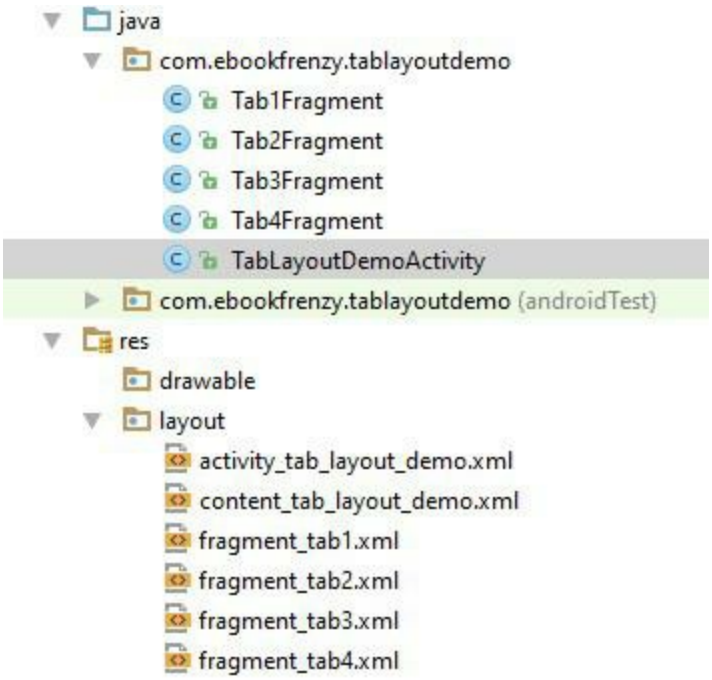

Figure 36-5

36.6 Adding the TabLayout and ViewPager

With the fragment creation process now complete, the next step is to add the TabLayout and ViewPager to the main activity layout file. Edit the *activity_tab_layout_demo.xml* file and add these elements as outlined in the following XML listing. Note that the TabLayout component is embedded into the AppBarLayout element while the ViewPager is placed after the AppBarLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context=".TabLayoutDemoActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
```

```
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

        <android.support.design.widget.TabLayout
            android:id="@+id/tab_layout"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabMode="fixed"
            app:tabGravity="fill"/>

    </android.support.design.widget.AppBarLayout>

    <android.support.v4.view.ViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
    />

    <include layout="@layout/content_tab_layout_demo" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

36.7 Creating the Pager Adapter

This example will use the ViewPager approach to handling the fragments assigned to the TabLayout tabs. With the ViewPager added to the layout resource file, a new class which subclasses FragmentPagerAdapter needs to be added to the project to manage the fragments that will be displayed when the tab items are selected by the user.

Add a new class to the project by right-clicking on the *com.ebookfrenzy.tablayoutdemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, enter *TabPagerAdapter* into the *Name:* field and click *OK*.

Edit the *TabPagerAdapter.java* file so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class TabPagerAdapter extends FragmentPagerAdapter {

    int tabCount;
```

```
    public TabPagerAdapter(FragmentManager fm, int numberOfTabs) {
        super(fm);
        this.tabCount = numberOfTabs;
    }

    @Override
    public Fragment getItem(int position) {

        switch (position) {
            case 0:
                Tab1Fragment tab1 = new Tab1Fragment();
                return tab1;
            case 1:
                Tab2Fragment tab2 = new Tab2Fragment();
                return tab2;
            case 2:
                Tab3Fragment tab3 = new Tab3Fragment();
                return tab3;
            case 3:
                Tab4Fragment tab4 = new Tab4Fragment();
                return tab4;
            default:
                return null;
        }
    }

    @Override
    public int getCount() {
        return tabCount;
    }
}
```

The class is declared as extending the FragmentPagerAdapter class and a constructor is implemented allowing the number of pages required to be passed to the class when an instance is created. The *getItem()* method will be called when a specific page is required. A switch statement is used to identify the page number being requested and to return a corresponding fragment instance. Finally, the *getCount()* method simply returns the count value passed through when the object instance was created.

36.8 Performing the Initialization Tasks

The remaining tasks involve initializing the TabLayout, ViewPager and TabPagerAdapter instances. All of these tasks will be performed in the *onCreate()* method of the *TabLayoutDemoActivity.java* file. Edit this file and modify the *onCreate()* method so that it reads as follows:

```
package com.ebookfrenzy.tablayoutdemo;

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

```
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;

public class TabLayoutDemoActivity extends AppCompatActivity {

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

        TabLayout tabLayout =
              (TabLayout) findViewById(R.id.tab_layout);

        tabLayout.addTab(tabLayout.newTab().setText("Tab 1 Item"));
        tabLayout.addTab(tabLayout.newTab().setText("Tab 2 Item"));
        tabLayout.addTab(tabLayout.newTab().setText("Tab 3 Item"));
        tabLayout.addTab(tabLayout.newTab().setText("Tab 4 Item"));

        final ViewPager viewPager =
              (ViewPager) findViewById(R.id.pager);
        final PagerAdapter adapter = new TabPagerAdapter
                (getSupportFragmentManager(),
                              tabLayout.getTabCount());
        viewPager.setAdapter(adapter);

        viewPager.addOnPageChangeListener(new
              TabLayout.TabLayoutOnPageChangeListener(tabLayout));
        tabLayout.addOnTabSelectedListener(new
              TabLayout.OnTabSelectedListener() {
            @Override
            public void onTabSelected(TabLayout.Tab tab) {
                viewPager.setCurrentItem(tab.getPosition());
            }

            @Override
            public void onTabUnselected(TabLayout.Tab tab) {

            }

            @Override
            public void onTabReselected(TabLayout.Tab tab) {

            }

        });
.
.
.
    }
.
.
.
}
```

The code begins by obtaining a reference to the TabLayout object that was added to the *activity_tab_layout_demo.xml* file and creating four tabs, assigning the text to appear on each:

```
TabLayout tabLayout =
        (TabLayout) findViewById(R.id.tab_layout);

tabLayout.addTab(tabLayout.newTab().setText("Tab 1 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 2 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 3 Item"));
tabLayout.addTab(tabLayout.newTab().setText("Tab 4 Item"));
```

A reference to the ViewPager instance in the layout file is then obtained and an instance of the TabPagerAdapter class created. Note that the code to create the TabPagerAdapter instance passes through the number of tabs that have been assigned to the TabLayout component. The TabPagerAdapter instance is then assigned as the adapter for the ViewPager and the TabLayout component added to the page change listener:

```
final ViewPager viewPager = (ViewPager) findViewById(R.id.pager);
final PagerAdapter adapter = new TabPagerAdapter
                (getSupportFragmentManager(),
                      tabLayout.getTabCount());
viewPager.setAdapter(adapter);

viewPager.addOnPageChangeListener(new
        TabLayout.TabLayoutOnPageChangeListener(tabLayout));
```

Finally, the onTabSelectedListener is configured on the TabLayout instance and the *onTabSelected()* method implemented to set the current page on the ViewPager based on the currently selected tab number. For the sake of completeness the other listener methods are added as stubs:

```
tabLayout.setOnTabSelectedListener(new
                      TabLayout.OnTabSelectedListener()
{
        @Override
        public void onTabSelected(TabLayout.Tab tab) {
               viewPager.setCurrentItem(tab.getPosition());
        }

        @Override
        public void onTabUnselected(TabLayout.Tab tab) {

        }

        @Override
        public void onTabReselected(TabLayout.Tab tab) {

        }
});
```

36.9 **Testing the Application**

Compile and run the app on a device or emulator and make sure that selecting a tab causes the corresponding fragment to appear in the content area of the screen:

Figure 36-6

36.10 Customizing the TabLayout

The TabLayout in this example project is configured using *fixed* mode. This mode works well for a limited number of tabs with short titles. A greater number of tabs or longer titles can quickly become a problem when using fixed mode as illustrated by Figure 36-7:

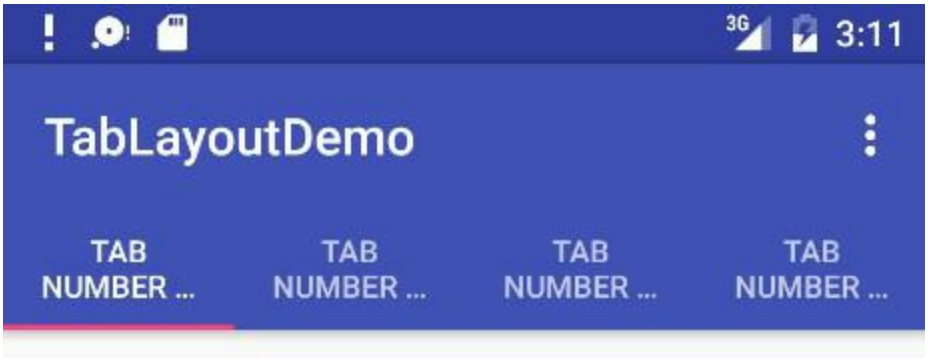

Figure 36-7

In an effort to fit the tabs into the available display width the TabLayout has used multiple lines of text. Even so, the second line is clearly truncated making it impossible to see the full title. The best

solution to this problem is to switch the TabLayout to *scrollable* mode. In this mode the titles appear in full length, single line format allowing the user to swipe to scroll horizontally through the available items as demonstrated in Figure 36-8:

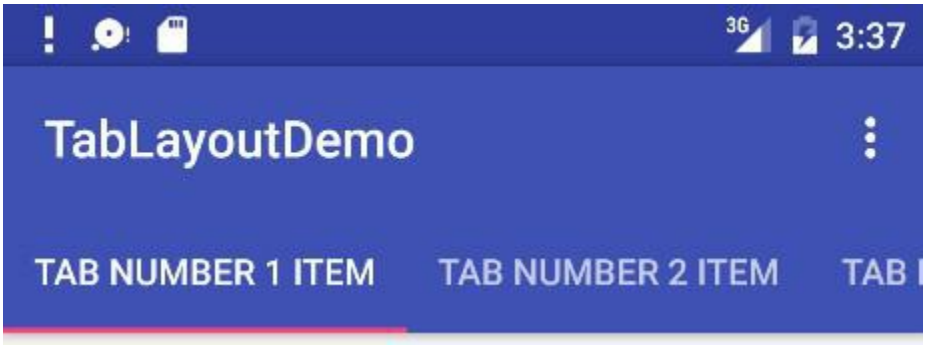

Figure 36-8

To switch a TabLayout to scrollable mode, simply change the *app:tabMode* property in the *activity_tab_layout_demo.xml* layout resource file from "fixed" to "scrollable":

```
<android.support.design.widget.TabLayout
    android:id="@+id/tab_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:tabMode="scrollable"
    app:tabGravity="fill"/>
</android.support.design.widget.AppBarLayout>
```

When in fixed mode, the TabLayout may be configured to control how the tab items are displayed to take up the available space on the screen. This is controlled via the *app:tabGravity* property, the results of which are more noticeable on wider displays such as tablets in landscape orientation. When set to "fill", for example, the items will be distributed evenly across the width of the TabLayout as shown in Figure 36-9:

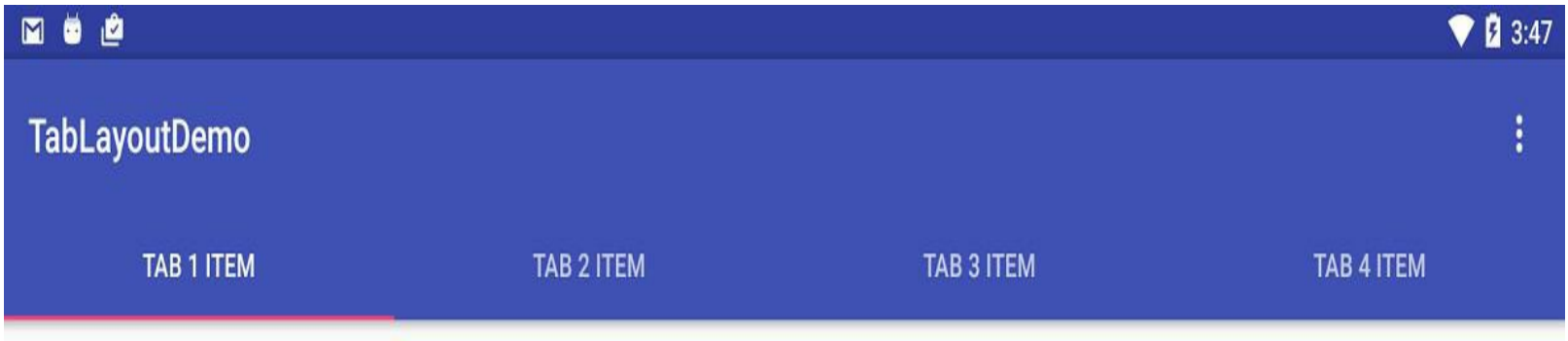

Figure 36-9

Changing the property value to "center" will cause the items to be positioned relative to the center of the tab bar:

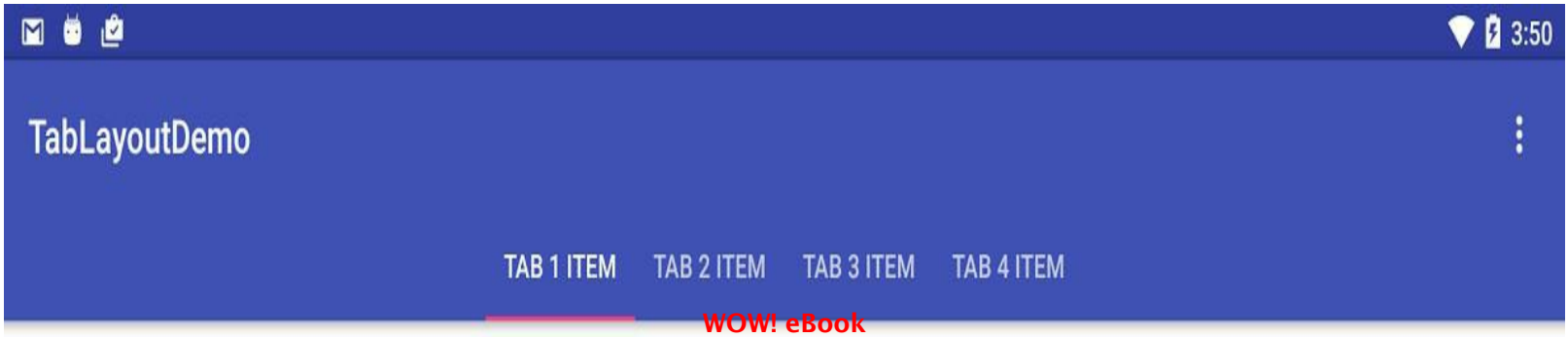

Figure 36-10

Before proceeding to the final step in this chapter, revert the tabMode and tabGravity properties in the *activity_tab_layout_demo.xml* file to "fixed" and "fill" respectively.

36.11 Displaying Icon Tab Items

The last step in this tutorial is to replace the text based tabs with icons. To achieve this, modify the *onCreate()* method in the *TabLayoutDemoActivity.java* file to assign some built-in drawable icons to the tab items:

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

    TabLayout tabLayout = (TabLayout) findViewById(R.id.tab_layout);

    tabLayout.addTab(tabLayout.newTab().setIcon(
                    android.R.drawable.ic_dialog_email));
    tabLayout.addTab(tabLayout.newTab().setIcon(
                    android.R.drawable.ic_dialog_dialer));
    tabLayout.addTab(tabLayout.newTab().setIcon(
                    android.R.drawable.ic_dialog_map));
    tabLayout.addTab(tabLayout.newTab().setIcon(
                    android.R.drawable.ic_dialog_info));

    final ViewPager viewPager =
              (ViewPager) findViewById(R.id.pager);
.
.
.
}
```

Instead of using the *setText()* method of the tab item, the code is now calling the *setIcon()* method and passing through a drawable icon reference. When compiled and run, the tab bar should now appear as shown in Figure 36-11. Note if using Instant Run that it will be necessary to trigger a warm swap using Ctrl-Shift-R for the changes to take effect:

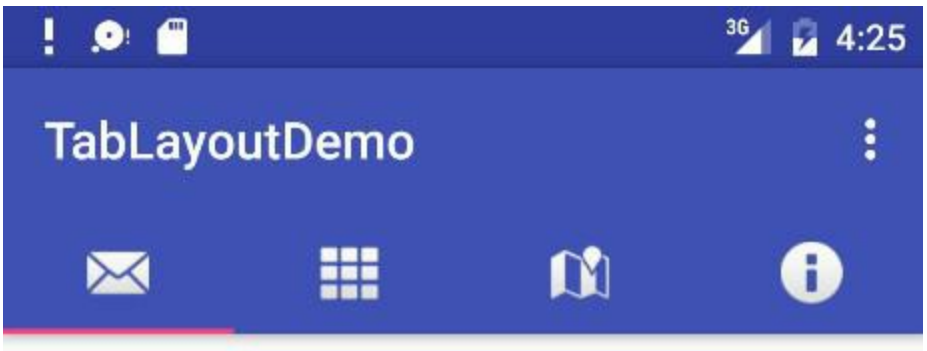

Figure 36-11

36.12 Summary

TabLayout is one of the components introduced as part of the Android material design implementation. The purpose of the TabLayout component is to present a series of tab items which,

when selected, display different content to the user. The tab items can display text, images or a combination of both. When combined with the ViewPager class and fragments, tab layouts can be created with relative ease, with each tab item selection displaying a different fragment.

37. Working with the RecyclerView and CardView Widgets

The RecyclerView and CardView widgets work together to provide scrollable lists of information to the user in which the information is presented in the form of individual cards. Details of both classes will be covered in this chapter before working through the design and implementation of an example project.

37.1 An Overview of the RecyclerView

Much like the ListView class outlined in the chapter entitled *Working with the Floating Action Button and Snackbar*, the purpose of the RecyclerView is to allow information to be presented to the user in the form of a scrollable list. The RecyclerView, however, provides a number of advantages over the ListView. In particular, the RecyclerView is significantly more efficient in the way it manages the views that make up a list, essentially reusing existing views that make up list items as they scroll off the screen instead if creating new ones (hence the name "recycler"). This both increases the performance and reduces the resources used by a list, a feature that is of particular benefit when presenting large amounts of data to the user.

Unlike the ListView, the RecyclerView also provides a choice of three built-in layout managers to control the way in which the list items are presented to the user:

- **LinearLayoutManager** – The list items are presented as either a horizontal or vertical scrolling list.

Figure 37-1

- **GridLayoutManager** – The list items are presented in grid format. This manager is best used when the list items of are of uniform size.

Figure 37-2

- **StaggeredGridLayoutManager** - The list items are presented in a staggered grid format. This manager is best used when the list items are not of uniform size.

Figure 37-3

For situations where none of the three built-in managers provide the necessary layout, custom layout managers may be implemented by subclassing the RecyclerView.LayoutManager class.

Each list item displayed in a RecyclerView is created as an instance of the ViewHolder class. The ViewHolder instance contains everything necessary for the RecyclerView to display the list item, including the information to be displayed and the view layout used to display the item.

As with the ListView, the RecyclerView depends on an adapter to act as the intermediary between the RecyclerView instance and the data that is to be displayed to the user. The adapter is created as a subclass of the RecyclerView.Adapter class and must, at a minimum, implement the following methods, which will be called at various points by the RecyclerView object to which the adapter is assigned:

- **getItemCount()** – This method must return a count of the number of items that are to be displayed in the list.
- **onCreateViewHolder()** – This method creates and returns a ViewHolder object initialized with the view that is to be used to display the data. This view is typically created by inflating the XML layout file.
- **onBindViewHolder()** – This method is passed the ViewHolder object created by the *onCreateViewHolder()* method together with an integer value indicating the list item that is about to be displayed. Contained within the ViewHolder object is the layout assigned by the *onCreateViewHolder()* method. It is the responsibility of the *onBindViewHolder()* method to populate the views in the layout with the text and graphics corresponding to the specified item and to return the object to the RecyclerView where it will be presented to the user.

Adding a RecyclerView to a layout is simply a matter of adding the appropriate element to the XML layout file of the activity in which it is to appear. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".CardStuffActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"/>

    <android.support.design.widget.AppBarLayout
```

```
android:layout_height="wrap_content"
        android:layout_width="match_parent"
android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent" android:layout_height="?
attr/actionBarSize"
            android:background="?attr/colorPrimary"
app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>
.
.
.
}
```

In the above example the RecyclerView has been embedded into the CoordinatorLayout of a main activity layout file along with the AppBar and Toolbar. This provides some additional features, such as configuring the Toolbar and AppBar to scroll off the screen when the user scrolls up within the RecyclerView (a topic covered in more detail in the chapter entitled *Working with the AppBar and Collapsing Toolbar Layouts*).

37.2 An Overview of the CardView

The CardView class is a user interface view that allows information to be presented in groups using a card metaphor. Cards are usually presented in lists using a RecyclerView instance and may be configured to appear with shadow effects and rounded corners. Figure 37-4, for example, shows three CardView instances configured to display a layout consisting of an ImageView and two TextViews:

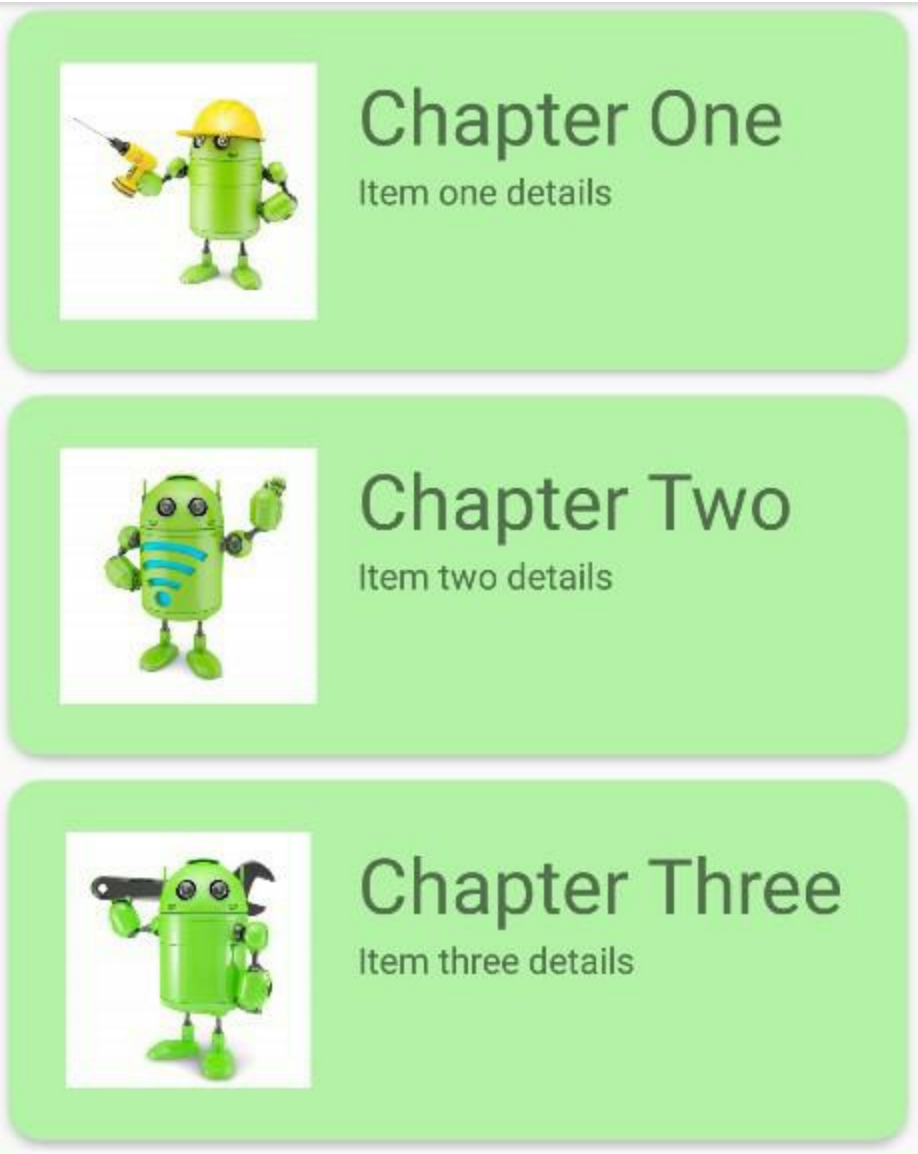

Figure 37-4

The user interface layout to be presented with a CardView instance is defined within an XML layout resource file and loaded into the CardView at runtime. The CardView layout can contain a layout of any complexity using the standard layout managers such as RelativeLayout and LinearLayout. The following XML layout file represents a card view layout consisting of a RelativeLayout and a single ImageView. The card is configured to be elevated to create shadowing effect and to appear with rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
    <android.support.v7.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/card_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="5dp"
        card_view:cardCornerRadius="12dp"
        card_view:cardElevation="3dp"
        card_view:contentPadding="4dp">

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
            android:padding="16dp" >

            <ImageView
                android:layout_width="100dp"
                android:layout_height="100dp"
                android:id="@+id/item_image"
                android:layout_alignParentLeft="true"
                android:layout_alignParentTop="true"
                android:layout_marginRight="16dp" />
        </RelativeLayout>
</android.support.v7.widget.CardView>
```

When combined with the RecyclerView to create a scrollable list of cards, the *onCreateViewHolder()* method of the recycler view inflates the layout resource file for the card, assigns it to the ViewHolder instance and returns it to the RecyclerView instance.

37.3 Adding the Libraries to the Project

In order to use the RecyclerView and CardView components, the corresponding libraries must be added to the Gradle build dependencies for the project. Within the module level *build.gradle* file, therefore, the following lines need to be added to the *dependencies* section:

```
dependencies {
.
.
    compile 'com.android.support:recyclerview-v7:25.2.0'
    compile 'com.android.support:cardview-v7:25.2.0'
}
```

37.4 Summary

This chapter has introduced the Android RecyclerView and CardView components. The RecyclerView provides a resource efficient way to display scrollable lists of views within an Android app. The CardView is useful when presenting groups of data (such as a list of names and addresses) in the form of cards. As previously outlined, and demonstrated in the tutorial contained in the next chapter, the RecyclerView and CardView are particularly useful when combined.

38. An Android RecyclerView and CardView Tutorial

In this chapter an example project will be created that makes use of both the CardView and RecyclerView components to create a scrollable list of cards. The completed app will display a list of cards containing images and text. In addition to displaying the list of cards, the project will be implemented such that selecting a card causes a messages to be displayed to the user indicating which card was tapped.

38.1 Creating the CardDemo Project

Create a new project in Android Studio, entering *CardDemo* 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) and continue to proceed through the screens.

In the next chapter, the scroll handling features of the AppBar, Toolbar and CoordinatorLayout layout will be demonstrated using this project. On the activity selection screen, therefore, request the creation of a Basic Activity named *CardDemoActivity* with corresponding layout and menu files named *activity_card_demo* and *menu_card_demo* respectively. Click on the *Finish* button to initiate the project creation process.

Once the project has been created, load the *content_card_demo.xml* file into the Layout Editor tool and select and delete the "Hello World" TextView object.

38.2 Removing the Floating Action Button

Since the Basic Activity was selected, the layout includes a floating action button which is not required for this project. Load the *activity_card_demo.xml* layout file into the Layout Editor tool, select the floating action button and tap the keyboard delete key to remove the object from the layout. Edit the *CardDemoActivity.java* file and remove the floating action button code from the onCreate method as follows:

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

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

38.3 Adding the RecyclerView and CardView Libraries

Within the Project tool window locate and select the module level *build.gradle* file and modify the dependencies section of the file to add the support library dependencies for the RecyclerView and CardView:

```
dependencies {
.
.
.
    compile 'com.android.support:appcompat-v7:25.2.0'
    compile 'com.android.support.constraint:constraint-layout:1.0.0'
    compile 'com.android.support:design:25.1.0'
    compile 'com.android.support:recyclerview-v7:25.2.0'
    compile 'com.android.support:cardview-v7:25.2.0'
    testCompile 'junit:junit:4.12'
}
```

When prompted to do so, resync the new Gradle build configuration by clicking on the *Sync Now* link in the warning bar.

38.4 Designing the CardView Layout

The layout of the views contained within the cards will be defined within a separate XML layout file. Within the Project tool window right click on the *app -> res -> layout* entry and select the New -> *Layout resource file* menu option. In the New Resource Dialog enter *card_layout* into the *File name:* field and *android.support.v7.widget.CardView* into the root element field before clicking on the *OK* button.

Load the *card_layout.xml* file into the Layout Editor tool, switch to Text mode and modify the layout so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/card_view"
    android:layout_margin="5dp"
    card_view:cardBackgroundColor="#81C784"
    card_view:cardCornerRadius="12dp"
    card_view:cardElevation="3dp"
    card_view:contentPadding="4dp" >

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp" >

        <ImageView
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:id="@+id/item_image"
            android:layout_alignParentStart="true"
            android:layout_alignParentLeft="true"
```

```
            android:layout_alignParentTop="true"
            android:layout_marginEnd="16dp"
            android:layout_marginRight="16dp" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/item_title"
            android:layout_toEndOf="@+id/item_image"
            android:layout_toRightOf="@+id/item_image"
            android:layout_alignParentTop="true"
            android:textSize="30sp" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/item_detail"
            android:layout_toEndOf="@+id/item_image"
            android:layout_toRightOf="@+id/item_image"
            android:layout_below="@+id/item_title" />

    </RelativeLayout>
</android.support.v7.widget.CardView>
```

38.5 Adding the RecyclerView

Select the *activity_card_demo.xml* layout file and modify it to add the RecyclerView component immediately before the AppBarLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    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:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:contextcom.ebookfrenzy.carddemo.CardDemoActivity">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    <android.support.design.widget.AppBarLayout
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">
.
.
.
```

38.6 Creating the RecyclerView Adapter

As outlined in the previous chapter, the RecyclerView needs to have an adapter to handle the creation of the list items. Add this new class to the project by right-clicking on the *app -> java ->*

com.ebookfrenzy.carddemo entry in the Project tool window and selecting the *New -> Java Class* menu option. In the Create New Class dialog, enter *RecyclerAdapter* into the *Name:* field before clicking on the *OK* button to create the new Java class file.

Edit the new *RecyclerAdapter.java* file to add some import directives and to declare that the class now extends RecyclerView.Adapter. Rather than create a separate class to provide the data to be displayed, some basic arrays will also be added to the adapter to act as the data for the app:

```
package com.ebookfrenzy.carddemo;

import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;

public class RecyclerAdapter extends
RecyclerView.Adapter<RecyclerAdapter.ViewHolder> {

    private String[] titles = {"Chapter One",
            "Chapter Two",
            "Chapter Three",
            "Chapter Four",
            "Chapter Five",
            "Chapter Six",
            "Chapter Seven",
            "Chapter Eight"};

    private String[] details = {"Item one details",
            "Item two details", "Item three details",
            "Item four details", "Item file details",
            "Item six details", "Item seven details",
            "Item eight details"};

   private int[] images = { R.drawable.android_image_1,
                            R.drawable.android_image_2,
                            R.drawable.android_image_3,
                            R.drawable.android_image_4,
                            R.drawable.android_image_5,
                            R.drawable.android_image_6,
                            R.drawable.android_image_7,
                            R.drawable.android_image_8 };
}
```

Within the RecyclerAdapter class we now need our own implementation of the ViewHolder class configured to reference the view elements in the *card_layout.xml* file. Remaining within the *RecyclerAdapter.java* file implement this class as follows:

```
.
.
.
public class RecyclerAdapter  extends
RecyclerView.Adapter<RecyclerAdapter.ViewHolder> {
.
.
```

```
.
   class ViewHolder extends RecyclerView.ViewHolder {

        public ImageView itemImage;
        public TextView itemTitle;
        public TextView itemDetail;

        public ViewHolder(View itemView) {
            super(itemView);
            itemImage =
               (ImageView)itemView.findViewById(R.id.item_image);
            itemTitle =
               (TextView)itemView.findViewById(R.id.item_title);
            itemDetail =
               (TextView)itemView.findViewById(R.id.item_detail);
        }
    }
.
.
.
}
```

The ViewHolder class contains an ImageView and two TextView variables together with a constructor method that initializes those variables with references to the three view items in the *card_layout.xml* file.

The next item to be added to the *RecyclerAdapter.java* file is the implementation of the *onCreateViewHolder()* method:

```
@Override
public ViewHolder onCreateViewHolder(ViewGroup viewGroup, int i) {
    View v = LayoutInflater.from(viewGroup.getContext())
                .inflate(R.layout.card_layout, viewGroup, false);
    ViewHolder viewHolder = new ViewHolder(v);
    return viewHolder;
}
```

This method will be called by the RecyclerView to obtain a ViewHolder object. It inflates the view hierarchy *card_layout.xml* file and creates an instance of our ViewHolder class initialized with the view hierarchy before returning it to the RecyclerView.

The purpose of the *onBindViewHolder()* method is to populate the view hierarchy within the ViewHolder object with the data to be displayed. It is passed the ViewHolder object and an integer value indicating the list item that is to be displayed. This method should now be added, using the item number as an index into the data arrays. This data is then displayed on the layout views using the references created in the constructor method of the ViewHolder class:

```
@Override
public void onBindViewHolder(ViewHolder viewHolder, int i) {
    viewHolder.itemTitle.setText(titles[i]);
    viewHolder.itemDetail.setText(details[i]);
    viewHolder.itemImage.setImageResource(images[i]);
}
```

The final requirement for the adapter class is an implementation of the *getItem()* method which, in this case, simply returns the number of items in the *titles* array:

```
    @Override
    public int getItemCount() {
        return titles.length;
    }
```

38.7 Adding the Image Files

In addition to the two TextViews, the card layout also contains an ImageView on which the Recycler adapter has been configured to display images. Before the project can be tested these images must be added. The images that will be used for the project are named *android_image_<n>.jpg* and can be found in the *project_icons* folder of the sample code download available from the following URL:

http://www.ebookfrenzy.com/retail/androidstudio23/index.php

Locate these images in the file system navigator for your operating system and select and copy the eight images. Right click on the *app -> res -> drawable* entry in the Project tool window and select Paste to add the files to the folder:

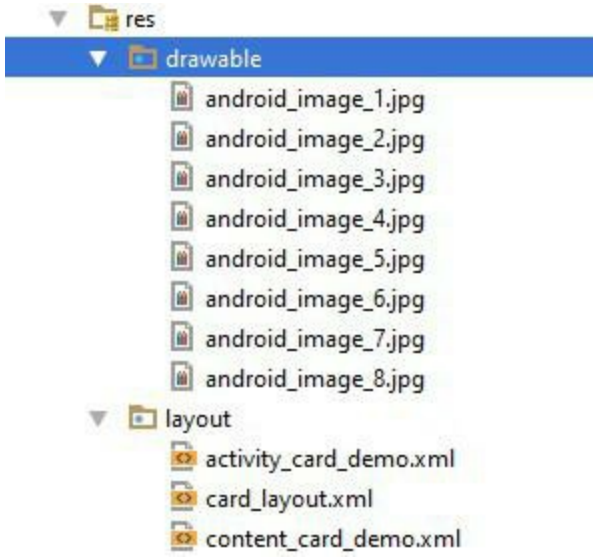

Figure 38-1

38.8 Initializing the RecyclerView Component

At this point the project consists of a RecyclerView instance, an XML layout file for the CardView instances and an adapter for the RecyclerView. The last step before testing the progress so far is to initialize the RecyclerView with a layout manager, create an instance of the adapter and assign that instance to the RecyclerView object. For the purposes of this example, the RecyclerView will be configured to use the LinearLayoutManager layout option. Edit the *CardDemoActivity.java* file and modify the *onCreate()* method to implement this initialization code:

```
package com.ebookfrenzy.carddemo;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;

public class CardDemoActivity extends AppCompatActivity {
```

```
    RecyclerView recyclerView;
    RecyclerView.LayoutManager layoutManager;
    RecyclerView.Adapter adapter;

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

        recyclerView =
              (RecyclerView) findViewById(R.id.recycler_view);

        layoutManager = new LinearLayoutManager(this);
        recyclerView.setLayoutManager(layoutManager);

        adapter = new RecyclerAdapter();
        recyclerView.setAdapter(adapter);
    }
.
.
}
```

38.9 Testing the Application

Compile and run the app on a physical device or emulator session and scroll through the different card items in the list:

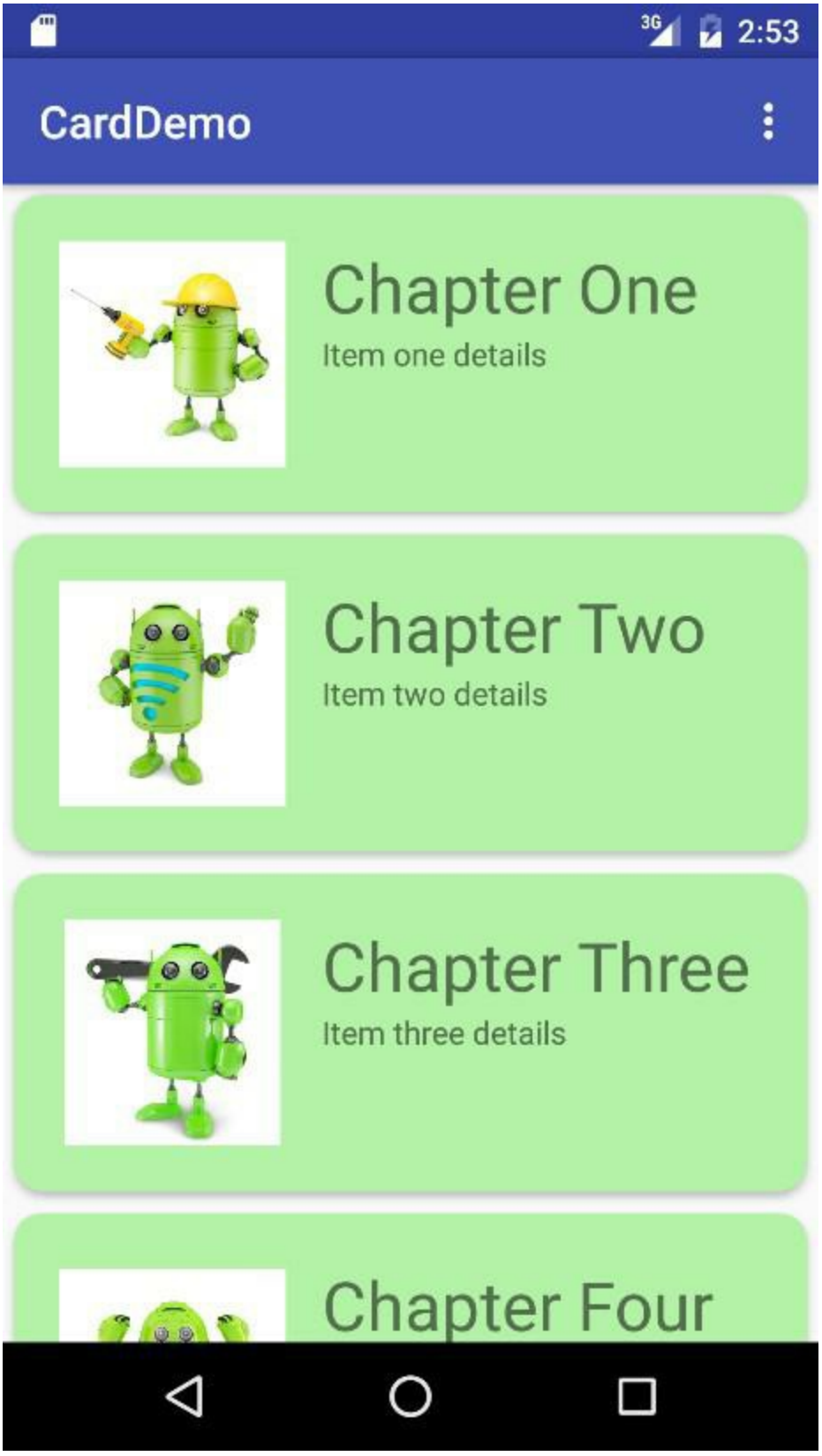

Figure 38-2

38.10 Responding to Card Selections

The last phase of this project is to make the cards in the list selectable so that clicking on a card triggers an event within the app. For this example, the cards will be configured to present a message on the display when tapped by the user. To respond to clicks, the ViewHolder class needs to be modified to assign an onClickListener on each item view. Edit the *RecyclerAdapter.java* file and modify the ViewHolder class declaration so that it reads as follows:

```
import android.support.design.widget.Snackbar;
```

```
.
.
.
class ViewHolder extends RecyclerView.ViewHolder{

    public int currentItem;
    public ImageView itemImage;
    public TextView itemTitle;
    public TextView itemDetail;

    public ViewHolder(View itemView) {
        super(itemView);
        itemImage = (ImageView)itemView.findViewById(R.id.item_image);
        itemTitle = (TextView)itemView.findViewById(R.id.item_title);
        itemDetail =
                (TextView)itemView.findViewById(R.id.item_detail);

        itemView.setOnClickListener(new View.OnClickListener() {
            @Override public void onClick(View v) {

            }
        });
    }
}
```

Within the body of the onClick handler, code can now be added to display a message indicating that the card has been clicked. Given that the actions performed as a result of a click will likely depend on which card was tapped it is also important to identify the selected card. This information can be obtained via a call to the *getAdapterPosition()* method of the RecyclerView.ViewHolder class. Remaining within the *RecyclerAdapter.java* file, add code to the *onClick* handler so it reads as follows:

```
@override
public void onClick(View v) {

    int position = getAdapterPosition();

    Snackbar.make(v, "Click detected on item " + position,
            Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
        }
});
```

The last task is to enable the material design ripple effect that appears when items are tapped within Android applications. This simply involves the addition of some properties to the declaration of the CardView instance in the *card_layout.xml* file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/card_view"
    android:layout_margin="5dp"
    card_view:cardBackgroundColor="#81C784"
```

```
        card_view:cardCornerRadius="12dp"
        card_view:cardElevation="3dp"
        card_view:contentPadding="4dp"
        android:foreground="?selectableItemBackground"
        android:clickable="true" >
```

Run the app once again and verify that tapping a card in the list triggers both the standard ripple effect at the point of contact and the appearance of a Snackbar reporting the number of the selected item.

38.11 **Summary**

This chapter has worked through the steps involved in combining the CardView and RecyclerView components to display a scrollable list of card based items. The example also covered the detection of clicks on list items, including the identification of the selected item and the enabling of the ripple effect visual feedback on the tapped CardView instance.