

Community Experience Distilled

Android 4: New features for Application Development

Develop Android applications using the new features of Android Ice Cream Sandwich

Murat Aydin

[PACKT] PUBLISHING open source* community experience distilled

Android 4: New features for Application Development

Develop Android applications using the new features of Android Ice Cream Sandwich

Murat Aydin

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Android 4: New features for Application Development

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2012

Production Reference: 1171212

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-84951-952-6

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author
Murat Aydin

Reviewers
Rick Boyer
Ahmet Oguz Mermerkaya
Nathan Schwermann
Murat Yener

Acquisition Editor
Usha Iyer

Commissioning Editor
Meeta Rajani
Maria D'souza
Yogesh Dalvi

Technical Editor
Nitee Shetty

Project Coordinator
Esha Thakker

Proofreader
Maria Gould

Indexer
Monica Ajmera Mehta

Graphics
Aditi Gajjar

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Author

Murat Aydin is a Senior Software Engineer in a company that develops software technologies for defense systems, and an enthusiastic Android developer. He has several Android applications in Google Play. He is a Sun Certified Java Developer and has eight years of experience in developing web-based applications using Java technologies, desktop, and engineering applications using .NET technologies.

He earned his BSc degree in Computer Engineering from METU (Middle East Technical University) and his MSc degree in Software Engineering from METU.

He is a member of GDG Ankara (Google Developer Group Ankara, `www.gdgankara.org`). They organize several Android events in GDG Ankara such as Android Developer Days (`www.androiddeveloperdays.com`).

He is married and lives in Ankara with his wife Ülkü.

You can get in touch with him on Linkedin at `http://www.linkedin.com/pub/murat-ayd%C4%B1n/33/702/6a2`, or through his Twitter handle `@maydintr`, or you can also e-mail him at `maydin@gmail.com`.

I would like to thank my precious wife Ülkü for abiding with me during my long hours of work for this book. A special thanks to my parents and my siblings for supporting me. I would also like to thank Ahmet Oğuz Mermerkaya, Murat Yener, Rick Boyer, and Nathan Schwermann for reviewing my book. Finally, I want to thank the team at Packt Publishing for all their help.

About the Reviewers

Rick Boyer has over twenty years of professional programming experience, including developing applications on Windows, Windows CE, Windows Phone, and Android. With a passion for mobile, he now focuses exclusively on the Android platform with his consulting business, NightSky Development. He also runs the LinkedIn group, Published Android Developers (`http://goo.gl/Byilc`), where developers discuss issues related to publishing apps to the market.

You can contact him at `about.me\RickBoyer`.

Ahmet Oguz Mermerkaya is an Electronics Engineer but has always worked as a software developer. He has developed softwares on different platforms using C, C++, Java, UML, and Web (PHP, MySQL). He also has experience in extreme programming techniques and model-driven development. Currently, he is working on Android application development. He is the author of *Merhaba Android*, a turkish book about Android application development. He is also an active member of the GDG community in Turkey.

Nathan Schwermann is a graduate from the University of Kansas and has been developing applications for Android professionally for over two years. He is a strong supporter of backward compatibility and is very familiar with both Google's support library and its famous extension Actionbar Sherlock. He also reviewed *Android 3.0 Animations*, *Packt Publishing*.

You can contact Nathan anytime at `schwiz@gmail.com` if you would like to talk about Android, job offers, or arrange a meet up at Google IO or other popular Android events.

Murat Yener completed his BS and MS degree at Istanbul Technical University. He has taken part in several projects still in use at the ITU Informatics Institute. He has worked on Isbank's Core Banking exchange project as a J2EE developer. He has designed and completed several projects still on the market by Muse Systems. He has worked in TAV Airports Information Technologies as a Enterprise Java & Flex developer. He has worked for HSBC as a project leader responsible for business processes and rich client user interfaces. Currently he is employed at Eteration A.S. working on several projects including Eclipse Libra Tools, GWT, and Mobile applications (both on Android and iOS).

He is also leading the Google Technology User Group Istanbul since 2009 and is a regular speaker at conferences such as JavaOne, EclipseCon, EclipseIst, and GDG meetings.

I would like to thank Murat Aydin, who offered me the chance to review his wonderful book. I would also like to thank Naci Dai for being my mentor and providing the best work environment, Daniel Kurka for developing mgwt, the best mobile platform I ever worked on, and Nilay Coskun for all her support.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface **1**

Chapter 1: Action Bar for All **7**

Action bar **7**

Adding an action bar 9

Adding an ActionProvider **14**

Adding submenus to the ActionProvider 19

ShareActionProvider 21

Adding an action view **24**

Using the action bar for navigation **30**

Summary **35**

Chapter 2: A New Layout – GridLayout **37**

GridLayout **37**

Why to use GridLayout 39

Adding a GridLayout 39

Configuring GridLayout 43

A new view – Space **49**

Summary **51**

Chapter 3: Social APIs **53**

Basics of contacts in Android **53**

Using Social API **54**

Device user profile 65

Summary **65**

Chapter 4: Calendar APIs **67**

Using Calendar APIs **67**

Creating an event 68

Using Intents for creating events 73

Adding an attendee 73

Adding a reminder 74
Summary 76
Chapter 5: Fragments 77
Fragment basics 77
Fragment lifecycle 78
Creating and managing fragments 79
Programmatically adding a fragment 85
Event sharing with activity 86
Using multiple fragments in an activity 87
Types of fragments 91
ListFragment 91
DialogFragment 91
PreferenceFragment 96
WebViewFragment 99
Summary 104
Chapter 6: Supporting Different Screen Sizes 105
Android 4.0 supports different screen sizes 105
Using match_parent and wrap_content 107
Using dip instead of px 111
Omit using AbsoluteLayout 113
Providing different bitmap drawables for different screen densities 114
Providing different layouts for different screen sizes 115
Nine-patch 117
Summary 118
Chapter 7: Android Compatibility Package 119
What is Android Compability Package 119
How to use the Android Compatibility Package 120
Summary 126
Chapter 8: New Connectivity APIs – Android Beam and Wi-Fi Direct 127
Android Beam 127
Beaming NdefMessages 128
Wi-Fi Direct 133
Sample Wi-Fi Direct application 134
Summary 143
Chapter 9: Multiple APK Support
This chapter is available for download at `http://www.packtpub.com/sites/default/files/downloads/Multiple_APK_Support.pdf`.

Chapter 10: APIs with Android Jelly Bean

This chapter is available for download at
`http://www.packtpub.com/sites/default/files/downloads/Android_JellyBean.pdf`.

Index **145**

Preface

This book is a practical and hands-on guide for developing Android applications using new features of Android Ice Cream Sandwich (Android 4.0), with a step-by-step approach and clearly explained sample codes. You will learn the new APIs in Android 4.0 with these sample codes.

What this book covers

Chapter 1, *Action Bar for All*, introduces us to the action bar and shows us how to use and configure the action bar.

Chapter 2, *A New Layout – GridLayout*, introduces us to GridLayout and shows us how to use and configure GridLayout. GridLayout is a new layout introduced with Android Ice Cream Sandwich. This layout is an optimized layout and could be used instead of LinearLayout and RelativeLayout.

Chapter 3, *Social APIs*, covers the Social APIs that were introduced with Android Ice Cream Sandwich. This API makes it easy to integrate the social networks. Furthermore, high resolution photos can now be used as a contact's photo after Ice Cream Sandwich was released. This chapter shows Social API usage with examples.

Chapter 4, *Calendar APIs*, covers the Calendar APIs which were introduced with Android Ice Cream Sandwich for managing calendars. Event, attendee, alert, and reminder databases can be managed with these APIs. These APIs allow us to easily integrate calendars with our Android applications. This chapter shows how to use Calendar APIs with examples.

Chapter 5, *Fragments*, introduces us to the basics of fragments and how to use them.

Chapter 6, *Supporting Different Screen Sizes*, introduces us to the ways of designing user interfaces that support different screen sizes.

Chapter 7, *Android Compatibility Package*, introduces us to the Android Compatibility Package and shows us how to use it. The Android Compatibility Package is to allow the porting of the new APIs to the older versions of the Android platform.

Chapter 8, *New Connectivity APIs – Android Beam and Wi-Fi Direct*, introduces us to Android Beam, which uses the NFC hardware of the device and Wi-Fi Direct which allows devices to connect to each other without using wireless access points. This chapter will teach us the usage of Android Beam and Wi-Fi Direct.

Chapter 9, *Multiple APK Support*, introduces us to Multiple APK Support which is a new option in Google Play (Android Market) by which multiple versions of APKs could be uploaded for a single application.

This chapter is available for download at `http://www.packtpub.com/sites/default/files/downloads/Multiple_APK_Support.pdf`.

Chapter 10, *APIs with Android Jelly Bean*, covers Android Jelly Bean and the new APIs within it.

This chapter is available for download at `http://www.packtpub.com/sites/default/files/downloads/Android_JellyBean.pdf`.

What you need for this book

To follow the examples in this book, the Android Development Tools should be set up and ready. The necessary software list is as follows:

- Eclipse with ADT plugin
- Android SDK Tools
- Android platform tools
- The latest Android platform

The Operating Systems that can be used are as follows:

- Windows XP (32-bit), Vista (32- or 64-bit), or Windows 7 (32- or 64-bit)
- Mac OS X 10.5.8 or later (x86 only)
- Linux (tested on Ubuntu Linux, Lucid Lynx)
    - GNU C Library (glibc) 2.7 or later is required
    - On Ubuntu Linux, version 8.04 or later is required
    - 64-bit distributions must be capable of running 32-bit applications

The specifications for use of the Eclipse IDE is as follows:

- Eclipse 3.6.2 (Helios) or greater (Eclipse 3.5 (Galileo) is no longer supported with the latest version of ADT)
- Eclipse JDT plugin (included in most Eclipse IDE packages)
- JDK 6 (JRE alone is not sufficient)
- Android Development Tools plugin (recommended)

Who this book is for

This book is for developers who are experienced with the Android platform, but who may not be familiar with the new features and APIs of Android 4.0.

Android developers who want to learn about supporting multiple screen sizes and multiple Android versions; this book is also for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Implement `onCreateOptionsMenu` and `onOptionsItemSelected` methods."

A block of code is set as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/settings"
    android:title="Settings">
    </item>
    <item android:id="@+id/about" android:title="About">
    </item>

</menu>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
@Override
public void onPrepareSubMenu(SubMenu subMenu) {
  //In order to add submenus, we should override this method we
    dynamically created submenus
    subMenu.clear();
    subMenu.add("SubItem1").setOnMenuItemClickListener(this);
    subMenu.add("SubItem2").setOnMenuItemClickListener(this);
  }
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Insert** button and then click on the **List** button".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

The source code will also be available on the author's website at `www.ottodroid.net`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Action Bar for All

Action bar API was firstly introduced with Android 3.0. With Android Ice Cream Sandwich, action bar supports small screen sizes. This chapter shows how to use and configure the action bar.

The topics covered in this chapter are as follows:

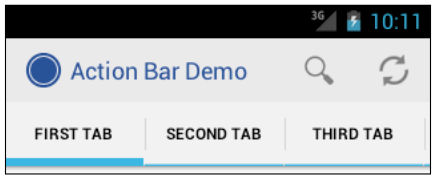
- Action bar types
- Adding an action bar
- Adding an ActionProvider and ShareActionProvider
- Adding an action view
- Using action bar for navigation

Action bar

Action bar is a user interface element located on top of the user's device screen. It provides actions and navigation capabilities to the user. Action bar has been available since API Level 11 (Android 3.0 Honeycomb) and after Ice Cream Sandwich was released, it supports small screen devices too. A sample Action Bar with tabs is shown in the following screenshot:



As it can be seen in the preceding screenshot, on the left of the bar there is an application logo and title, and then come the tabs for navigation. Lastly, the action buttons are placed after the tabs. The action buttons that do not fit to screen are displayed as an overflow menu with three dots on the right of the bar. In the previous screenshot, the action bar is displayed on a large screen device. However, in small screen devices, the Action Bar is displayed as a stack of bars as seen in the following screenshot:



As it can be seen in the preceding screenshot, there is not enough space to display all action bar items and the action bar is displayed with two bars on top of the screen.

Another type of action bar is the **split** action bar. In this type of action bar, action buttons are displayed in a bar at the bottom of the screen in narrow screens as shown in the following screenshot:



Adding an action bar

After Ice Cream Sandwich, Android doesn't require the menu button to reach the options menu. The best practice is to use action bar instead of the menu button. It is very easy to migrate from the options menu to the action bar. Now we are going to create a menu and then migrate that menu to the action bar.

Firstly, create an Android project and then add a menu that contains `Settings` and `About` as menu items. The resulting menu XML file should look like the following code block:



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

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

    <item android:id="@+id/settings"
    android:title="Settings"></item>

    <item android:id="@+id/about" android:title="About"></item>

</menu>
```

The layout XML for this sample is a `LinearLayout` layout with a `TextView` component in it as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

Implement the `onCreateOptionsMenu` and `onOptionsItemSelected` methods as shown in the following code block, in order to show the menu items:

```
package com.chapter1;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

public class Chapter1Activity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
            //Inflate the menu.xml of the android project
             //in order to create menu
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
    }

    @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    //According to selection, show the Toast message
    //of the selected button
    switch (item.getItemId()) {

    case R.id.settings:
      Toast.makeText(this, "Settings options menu button
      is pressed", Toast.LENGTH_LONG).show();
      return true;
    case R.id.about:
      Toast.makeText(this, "About options menu button is pressed",
      Toast.LENGTH_LONG).show();
      return true;
      default:
      return super.onOptionsItemSelected(item);
    }
  }
}
```

In order to display the action bar, the Android applications should target a minimum of API Level 11 in the `AndroidManifest.xml` file as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<!—set targetSDKversion to 11 because Action Bar is
     available since API Level 11-->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >

    < uses-sdk android:minSdkVersion="5"
              android:targetSdkVersion="11"  />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Chapter1Activity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

With this configuration, when the application runs on devices that have Android 3.0 or greater, the action bar will be displayed.

When we run this application on an emulator with API Level 15, we will see the overflow menu on the right of the action bar and the options menu buttons will be displayed when the overflow menu is pressed. In order to show the options menu buttons on the action bar (not as an overflow menu), just add `android:showAsAction="ifRoom|withText"` in the `item` tags of the menu XML file. The resulting menu XML file should look like the following code block:

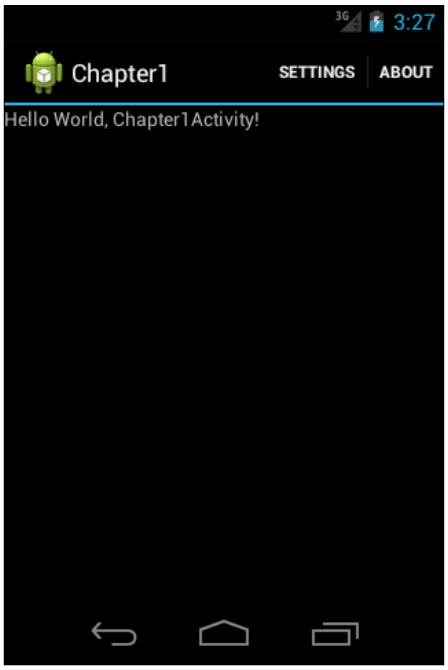
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
```

```
        <item android:id="@+id/settings" android:title="Settings"
        android:showAsAction="ifRoom|withText"></item>

        <item android:id="@+id/about" android:title="About"
        android:showAsAction="ifRoom|withText"></item>

    </menu>
```

If there is not enough space (`ifRoom`) to display the options menu buttons, the buttons will be displayed as an overflow menu. In order to show the options menu buttons with icon only (if an icon is provided), `withText` should be removed. When you run the application it will look like the following screenshot:



In some cases, you may not want to display the action bar. In order to remove the action bar, add `android:theme="@android:style/Theme.Holo.NoActionBar"` to the `activity` tag in the `AndroidManifest.xml` file. The resulting `AndroidManifest.xml` should look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-sdk android:minSdkVersion="5"
              android:targetSdkVersion="11"  />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Chapter1Activity"
            android:label="@string/app_name"
            android:theme="@android:style/Theme.Holo.NoActionBar" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

In order to show the action bar as a *split* action bar, add the `android:uiOptions="splitActionBarWhenNarrow"` application in the `activity` tag in `AndroidManifest.xml`. The resulting `AndroidManifest.xml` should look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="5"
        android:targetSdkVersion="11"  />

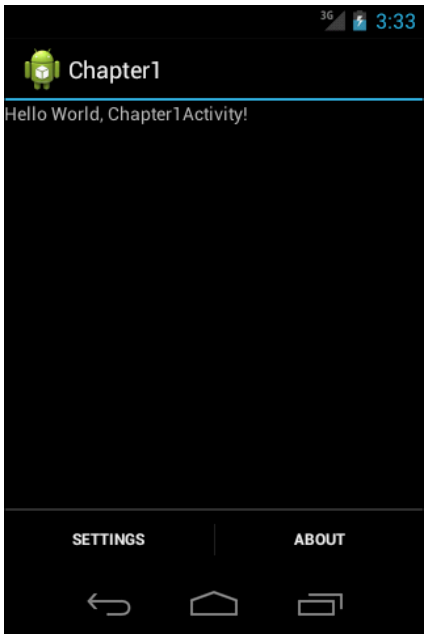
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:uiOptions="splitActionBarWhenNarrow">
        <activity
            android:name=".Chapter1Activity"
            android:label="@string/app_name"
            >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
```

```
            </intent-filter>
        </activity>
    </application>

</manifest>
```

When you run this application on an emulator, the screen will look like the following screenshot:



Adding an ActionProvider

In order to use a custom view instead of a simple button in action bar, the `ActionProvider` class could be the solution. **ActionProvider** has been available since API Level 14. ActionProvider can generate a custom view in the action bar, can generate submenus, and can handle events of the views that it generates. In order to create an ActionProvider, we should extend the `ActionProvider` class. The following code shows a sample class that extends the `ActionProvider` class and displays a custom layout instead of a simple button in action bar:

```
import android.content.Context;
import android.view.ActionProvider;
import android.view.LayoutInflater;
import android.view.View;
```

```
import android.widget.ImageButton;
import android.widget.Toast;

public class Chapter1ActionProvider extends ActionProvider {

  Context mContext;

  public Chapter1ActionProvider(Context context) {
    super(context);
    mContext = context;
  }

  @Override
  public View onCreateActionView() {
        //This method is the place where we generate a custom
        layout for the Action Bar menu item
     LayoutInflater layoutInflater =
     LayoutInflater.from(mContext);
         View view =
         layoutInflater.inflate(R.layout.action_provider, null);
         ImageButton button = (ImageButton)
         view.findViewById(R.id.button);

         button.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
          Toast.makeText(mContext, "Action Provider click",
          Toast.LENGTH_LONG).show();
             }
         });
    return view;
  }

  @Override
  public boolean onPerformDefaultAction() {
          //This is the method which is called when the Action Bar
          menu item is in overflow menu and clicked from there
          Toast.makeText(mContext, "Action Provider click",
          Toast.LENGTH_LONG).show();
    return true;
  }
}
```

We have to add a constructor and override the `onCreateActionView()` method. In the constructor, we assign `Context` to a variable because we are going to need it in further implementations. The `onCreateActionView()` method is the place where we generate a custom layout for the action bar menu item. `onPerformDefaultAction()` is the method which is called when the action bar menu item is in the overflow menu and is clicked from there. If the ActionProvider provides submenus, this method is never called. The layout XML for the custom layout used in the `onCreateActionView()` method is shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="center"
    android:focusable="true"
    android:addStatesFromChildren="true"
    android:background="?android:attr/actionBarItemBackground"
    style="?android:attr/actionButtonStyle">

    <ImageButton android:id="@+id/button"
        android:background="@drawable/ic_launcher"
        android:layout_width="32dip"
        android:layout_height="32dip"
        android:layout_gravity="center"
        android:scaleType="fitCenter"
        android:adjustViewBounds="true" />
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Some Text"
        android:textAppearance="?android:attr/textAppearanceLarge" />


</LinearLayout>
```

As you can see in the XML file, we added an `ImageButton` component and a `TextView` component to a `LinearLayout` layout. The `onClickListener()` event of `ImageButton` is implemented in the `onCreateActionView()` method of the `Chapter1ActionProvider` class. In this event, a `Toast` message is displayed.

The `Activity` class that displays the action bar is shown the following code block:

```
public class Chapter1ActionProviderActivity extends Activity{

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // TODO Auto-generated method stub
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
  }

   @Override
    public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    switch (item.getItemId()) {

    case R.id.about:
        Toast.makeText(this, "About options menu button is
        pressed", Toast.LENGTH_LONG).show();
        return true;
      default:
        return super.onOptionsItemSelected(item);
      }
    }
}
```

In order to display a custom layout for an action bar menu item, we have to assign an `ActionProvider` class in the `menu` XML file. We assign `Chapter1ActionProvider` which was implemented as in the earlier code as `ActionProvider`. The menu XML file in our example is as follows:

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

    <item android:id="@+id/settings" android:title="Settings"
    android:showAsAction="ifRoom|withText"
```

```
    android:actionProviderClass="com.chapter1.
Chapter1ActionProvider"></item>

    <item android:id="@+id/about" android:title="About"
    android:showAsAction="ifRoom|withText"></item>

</menu>
```

As you see in the `menu` XML file, we provided an `ActionProvider` class to the `settings` menu item. The last important thing is setting the minimum SDK version to API Level 14 in the `AndroidManifest.xml` file, because `ActionProvider` is a new feature released in API Level 14. The `AndroidManifest.xml` file should look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >
<!—set minSDKversion to 11 because ActionProvider is
     available since API Level 11-->

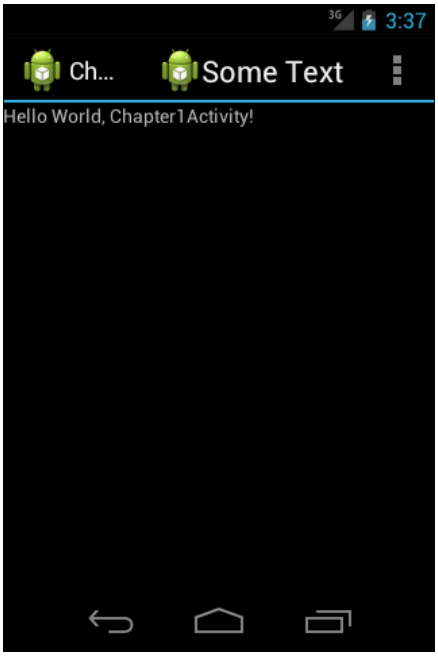
    <uses-sdk android:minSdkVersion="14"
        android:targetSdkVersion="14"  />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Chapter1ActionProviderActivity"
            android:label="@string/app_name"
            >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

When you run this application in an emulator, a user interface component with an image button and a text view will be displayed in the action bar. A toast message will be displayed if you press the image button. The screen will look like the following:



Adding submenus to the ActionProvider

It is possible to show submenus with ActionProvider. In order to add submenus, we should override the `onPrepareSubMenu(SubMenu subMenu)` and `hasSubMenu()` methods in the `Chapter1ActionProvider` class. The resulting code of the `Chapter1ActionProvider` class should look like the following code block:

```
package com.chapter1;

import android.app.Activity;
import android.content.Context;
import android.view.ActionProvider;
import android.view.LayoutInflater;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.MenuItem.OnMenuItemClickListener;
import android.view.SubMenu;
import android.view.View;
import android.widget.ImageButton;
```

```
import android.widget.Toast;

public class Chapter1ActionProvider extends ActionProvider implements
OnMenuItemClickListener {

  Context mContext;

  public Chapter1ActionProvider(Context context) {
    super(context);
    mContext = context;
  }

  @Override
  public View onCreateActionView() {
         return null;
  }

  @Override
  public boolean onPerformDefaultAction() {

    Toast.makeText(mContext, "Action Provider click",
    Toast.LENGTH_LONG).show();
    return true;
  }

  @Override
  public void onPrepareSubMenu(SubMenu subMenu) {
    //In order to add submenus, we should override this method
    // we dynamically created submenus

    subMenu.clear();
    subMenu.add("SubItem1").setOnMenuItemClickListener(this);
    subMenu.add("SubItem2").setOnMenuItemClickListener(this);
  }

  @Override
  public boolean onMenuItemClick(MenuItem item) {

    Toast.makeText(mContext, "Sub Item click",
    Toast.LENGTH_LONG).show();
    return true;
  }

  @Override
```

```
    public boolean hasSubMenu() {
         // we implemented it as returning true because we have menu
      return true;
    }

}
```

In the `onPrepareSubMenu(SubMenu subMenu)` method, we dynamically created submenus and set their `onMenuItemClickListener` events. The `onPrepareSubMenu(SubMenu subMenu)` method is called if the `hasSubMenu()` method returns true, so we implemented it as returning true.

It is also possible to create submenus from a `menu` XML file. If you want to create submenus from a `menu` XML file, `onPrepareSubMenu(SubMenu subMenu)` should look like the following code block:

```
@Override
public void onPrepareSubMenu(SubMenu subMenu) {

  MenuInflater inflater =
  ((Activity)mContext).getMenuInflater();
  inflater.inflate(R.menu.menu2, subMenu);
}
```

This code shows how we could inflate an XML file to create the submenus using the `menu` XML file `menu2`.

ShareActionProvider

ShareActionProvider provides a consistent way of sharing. It puts an action button on the action bar with a share icon. When you click that button, it lists the available applications for sharing. All you need is to declare `ShareActionProvider` in the `menu` item as shown in the following code block:

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

    <item android:id="@+id/share" android:title="Share"
    android:showAsAction="ifRoom"
    android:actionProviderClass="android.widget.
    ShareActionProvider" ></item>
    <item android:id="@+id/about" android:title="About"
    android:showAsAction="ifRoom"></item>
    <item android:id="@+id/settings" android:title="Settings"
    android:showAsAction="ifRoom"></item>

</menu>
```

The `Activity` class that uses `ShareActionProvider` should look like the following code block:

```
package com.chapter1;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.ShareActionProvider;

public class Chapter1ShareActionProviderActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

      ShareActionProvider myShareActionProvider;
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    MenuItem item = menu.findItem(R.id.share);
    myShareActionProvider =
    (ShareActionProvider)item.getActionProvider();
    myShareActionProvider.setShareHistoryFileName(
    ShareActionProvider.DEFAULT_SHARE_HISTORY_FILE_NAME);
    myShareActionProvider.setShareIntent(getShareIntent());

    return true;
    }

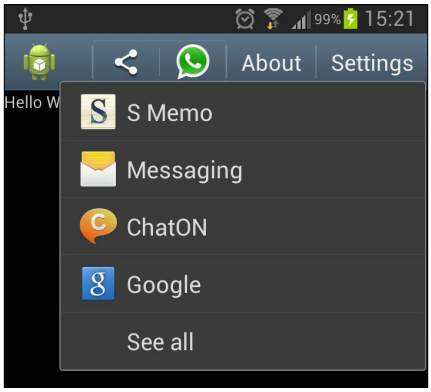
    private Intent getShareIntent() {
      Intent shareIntent = new Intent(Intent.ACTION_SEND);
      shareIntent.setType("text/plain");
      shareIntent.putExtra(Intent.EXTRA_TEXT, "www.somesite.com");
      return shareIntent;
      }

}
```

As you can see in the code, we get the `ShareActionProvider` attribute of the `menu` item in the `onCreateOptionsMenu(Menu menu)` method. Then we define the intent for sharing with the `setShareIntent` method of `ShareActionProvider`. `getShareIntent()` method creates an **intent** for sharing text. We use this method to define intent for the `ShareActionProvider` instance.

ShareActionProvider keeps the history of applications used for sharing in a file. The default file that ShareActionProvider uses is `ShareActionProvider.DEFAULT_SHARE_HISTORY_FILE_NAME`. It is possible to change this file with the `setShareHistoryFileName` method. All you need is to pass an XML file name with the .xml extension to this method. ShareActionProvider uses this file to find the most frequently used application for sharing. Then it displays the most frequently used application near the share action button as a default sharing target.

The screen of the application with ShareActionProvider looks like the following:



Since the ShareActionProvider was introduced in API Level 14, we have to set the minimum SDK to 14 in the `AndroidManifest.xml` file as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >
<!—set minSdkVersion to 14 because ShareActionProvider is available
    since API Level 14-->

    <uses-sdk android:minSdkVersion="14" />

    <application
```

```
            android:icon="@drawable/ic_launcher"
            android:label="@string/app_name" >
            <activity
                android:name=".Chapter1ShareActionProviderActivity"
                android:label="@string/app_name" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

                    <category android:name="android.intent.category.
                    LAUNCHER" />
                </intent-filter>
            </activity>
        </application>

    </manifest>
```

Adding an action view

An **action view** is a user interface component that appears in the action bar instead of an action button. This view is collapsible, that is if it is configured as collapsible, meaning it expands when the action button is pressed. If it is not configured as collapsible, it is viewed expanded by default. In the following example, we added an action view and showed its events and how to handle these events.

Firstly, add a layout for the action view that has three buttons with the text `Large`, `Medium`, and `Small` as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <Button
        android:id="@+id/buttonLarge"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Large"
        android:textSize="15dp" />

    <Button
        android:id="@+id/buttonMedium"
        android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Medium"
            android:textSize="12dp" />

        <Button
            android:id="@+id/buttonSmall"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Small"
            android:textSize="9dp" />

    </LinearLayout>
```

Then we need to bind this action view to an action bar `menu` item. The XML code of `menu` is shown in the following code bock:

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

    <item android:id="@+id/size" android:title="Size"
    android:showAsAction="ifRoom|collapseActionView"
    android:actionLayout="@layout/actionview"></item>

    <item android:id="@+id/about" android:title="About"
    android:showAsAction="ifRoom"></item>
    <item android:id="@+id/settings" android:title="Settings"
    android:showAsAction="ifRoom|withText"></item>

</menu>
```

As you can see in the `menu` XML code, we bind the action view to the `size` menu item by setting the `actionLayout` property. We also set the `showAsAction` property to `collapseActionView`. This way the action view is collapsible and it expands when the action button item is pressed. This option helps us to save space in the action bar. If this property is not set as `collapseActionView`, the action view is displayed as expanded by default.

The `Activity` class that handles action view events is shown in the following code block:

```
package com.chapter1;

import android.app.Activity;
import android.os.Bundle;
```

```
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.MenuItem.OnActionExpandListener;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

public class Chapter1ActionViewActivity extends Activity implements
OnClickListener {
  Button buttonLarge;
  Button buttonMedium;
  Button buttonSmall;
  Menu menu;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
      //you can set on click listeners of the items in Action View
     in this method

      this.menu = menu;
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    MenuItem item = menu.findItem(R.id.size);
    item.setOnActionExpandListener(new
    Chapter1ActionListener(this));
    buttonLarge =
    (Button)item.getActionView().findViewById(R.id.buttonLarge);
    buttonLarge.setOnClickListener(this);

    buttonMedium =
    (Button)item.getActionView().findViewById(R.id.buttonMedium);
    buttonMedium.setOnClickListener(this);

    buttonSmall =
    (Button)item.getActionView().findViewById(R.id.buttonSmall);
```

```
  buttonSmall.setOnClickListener(this);

  return true;
  }

  @Override
public boolean onOptionsItemSelected(MenuItem item) {
  // Handle item selection
  switch (item.getItemId()) {

  case R.id.size:
    Toast.makeText(this, "Size options menu button is
    pressed", Toast.LENGTH_LONG).show();
    return true;
  case R.id.about:
    Toast.makeText(this, "About options menu button is
    pressed", Toast.LENGTH_LONG).show();
    return true;
    case R.id.settings:
    Toast.makeText(this, "Settings options menu button is
    pressed", Toast.LENGTH_LONG).show();
    return true;
  default:
    return super.onOptionsItemSelected(item);
  }
}

@Override
public void onClick(View v) {

  if(v == buttonLarge )
  {
    Toast.makeText(this, "Large button is pressed",
    Toast.LENGTH_LONG).show();
                //Collapse the action view
    menu.findItem(R.id.size).collapseActionView();
  }
  else if(v == buttonMedium )
  {
    Toast.makeText(this, "Medium button is pressed",
    Toast.LENGTH_LONG).show();
                 //Collapse the action view
    menu.findItem(R.id.size).collapseActionView();
  }
```

```
        else if(v == buttonSmall)
        {
          Toast.makeText(this, "Small button is pressed",
          Toast.LENGTH_LONG).show();
                        //Collapse the action view
          menu.findItem(R.id.size).collapseActionView();
        }

      }

          // This class returns a callback when Action View is
          expanded or collapsed
      public static class Chapter1ActionListener implements
      OnActionExpandListener
      {
        Activity activity;

        public Chapter1ActionListener(Activity activity)
        {
          this.activity = activity;
        }

        @Override
        public boolean onMenuItemActionCollapse(MenuItem item) {

          Toast.makeText(activity, item.getTitle()+" button is
          collapsed", Toast.LENGTH_LONG).show();
          return true;
      }

       @Override
        public boolean onMenuItemActionExpand(MenuItem item) {
          Toast.makeText(activity, item.getTitle()+" button is
          expanded", Toast.LENGTH_LONG).show();
          return true;
        }

      }
    }
```

As you see in the `Chapter1ActionViewActivity`, you can set event listeners of the items in action view in the `onCreateOptionsMenu(Menu menu)` method. We set the `onClickListener` event of the buttons in the action view in the `onCreateOptionsMenu(Menu menu)` method.

It is possible to expand and collapse the action view programmatically with the `expandActionView()` and `collapseActionView()` methods. As you can see in the `onClick(View v)` method of the `Chapter1ActionViewActivity` method, we manually collapsed the action view with the `collapseActionView()` method.

You can do an action when the action view is expanded or collapsed with the `OnActionExpandListener` class. As you can see in the code, we defined the `Chapter1ActionListener` class that implements `OnActionExpandListener`. We override the `onMenuItemActionCollapse(MenuItem item)` and `onMenuItemActionExpand(MenuItem item)` methods of this class in order to show a `Toast` message. We passed `Activity` as a parameter to the constructor of `Chapter1ActionListener` because we need the `Activity` when showing the `Toast` message. We have to register the `setOnActionExpandListener()` method with the `OnActionExpandListener` class, in order to handle expand and collapse events. As you can see in the code, we registered this event in the `onCreateOptionsMenu(Menu menu)` method. We show a `Toast` message when the action view is collapsed and expanded.

Since the action view is introduced in API Level 14, we have to set the minimum SDK property to 14 or greater in the `AndroidManifest.xml` file as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter1"
    android:versionCode="1"
    android:versionName="1.0" >
<!—set minSdkVersion to 14 because Action View is
   available since API Level 14-->

    <uses-sdk android:minSdkVersion="14" />

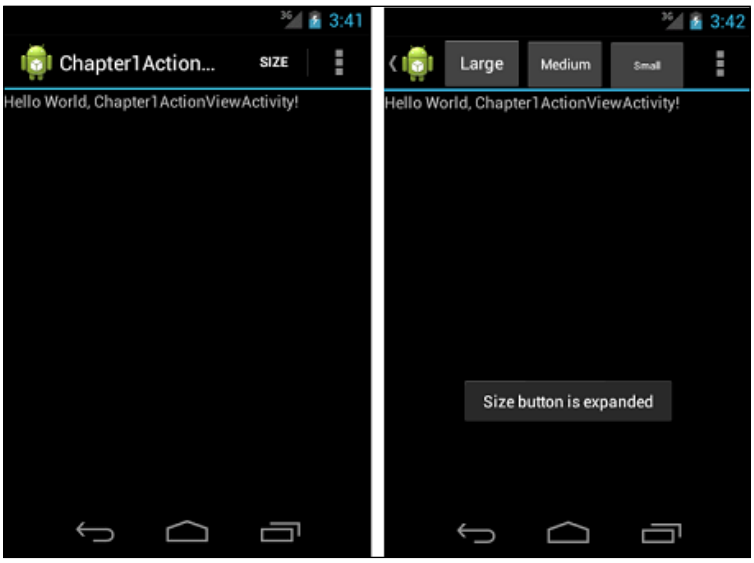
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Chapter1ActionViewActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
```

```
            </intent-filter>
        </activity>
    </application>

</manifest>
```

When you run this application on an emulator it will look like the following screenshot:



Using the action bar for navigation

Tabbed navigation could also be implemented with the `TabWidget` class. However, the action bar has some advantages. The action bar automatically adjusts itself according to the device screen size. For instance, if there is not enough space for tabs, it will display tabs in a stacked bar manner. Thus, it's better to use the action bar for tabbed navigation implementation.

Now, we are going to see how to use the action bar for tabbed navigation. Firstly, create an Android project and add two fragments: one that displays `Fragment A` and an other that displays `Fragment B`. The layout XML for fragments should look like the following code block:

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

```
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Fragment A"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_gravity="center_horizontal"/>

</LinearLayout>
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Fragment B"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_gravity="center_horizontal"/>

</LinearLayout>
```

The classes that extend the `Fragment` class for the two fragments should look like the following code block:

```
package com.chapter1;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class FragmentA extends Fragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup
  container,
```

```
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_a, container,
        false);
      return view;
    }
  }
  package com.chapter1;

  import android.app.Fragment;
  import android.os.Bundle;
  import android.view.LayoutInflater;
  import android.view.View;
  import android.view.ViewGroup;

  public class FragmentB extends Fragment {


    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container,
        Bundle savedInstanceState) {
      View view = inflater.inflate(R.layout.fragment_b, container,
      false);
      return view;
    }
  }
```

In order to use action bar for tabbed navigation, we should firstly implement the `ActionBar.TabListener` class. The class that implements `TabListener` is going to be used in the `Activity` class in adding tabs. The `Activity` class with the `TabListener` implementation should look like the following code block:

```
  package com.chapter1;

  import android.app.ActionBar;
  import android.app.ActionBar.Tab;
  import android.app.ActionBar.TabListener;
  import android.app.Activity;
  import android.app.Fragment;
  import android.app.FragmentTransaction;
  import android.os.Bundle;

  public class Chapter1ActionBarTabActivity extends Activity {

    @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    Tab tab = actionBar
        .newTab()
        .setText("First tab")
        .setTabListener(
          new
          Chapter1TabListener<FragmentA>(this, "fragmentA",
          FragmentA.class));
    actionBar.addTab(tab);

    tab = actionBar
        .newTab()
        .setText("Second Tab")
        .setTabListener(
            new Chapter1TabListener<FragmentB>
            (this, "fragmentB",FragmentB.class));
    actionBar.addTab(tab);
  }

  public static class Chapter1TabListener<T extends Fragment>
implements
      TabListener {
    private Fragment mFragment;
    private final Activity mActivity;
    private final String mTag;
    private final Class<T> mClass;

    public Chapter1TabListener(Activity activity, String tag,
    Class<T> clz) {
      mActivity = activity;
      mTag = tag;
      mClass = clz;
    }

    @Override
    public void onTabSelected(Tab tab, FragmentTransaction ft) {
                // we initialize and add the fragment to our
                  Activity if it doesn't exist
```

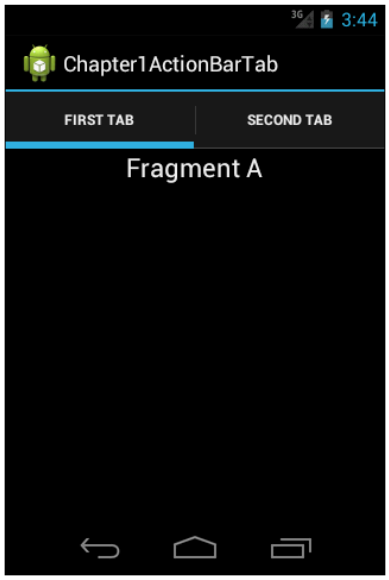
```
        if (mFragment == null) {

          mFragment = Fragment.instantiate(mActivity,
          mClass.getName());
          ft.add(android.R.id.content, mFragment, mTag);

          } else {
                              // If it exists, we simply attach it
          ft.attach(mFragment);
        }
      }

      @Override
      public void onTabUnselected(Tab tab, FragmentTransaction ft) {
                      // in this method we detach the fragment because
                      // it shouldn't be displayed
      if (mFragment != null) {
          ft.detach(mFragment);
          }
      }
      @Override
      public void onTabReselected(Tab tab, FragmentTransaction ft) {
                    // This method is called when the tab is reselected
      }
    }
  }
```

In the `Chapter1TabListener` class there are three methods that need to be overridden: `onTabReselected(Tab tab, FragmentTransaction ft)`, `onTabUnselected(Tab tab, FragmentTransaction ft)`, and `onTabSelected(Tab tab, FragmentTransaction ft)`. In the `onTabSelected(Tab tab, FragmentTransaction ft)` method, we initialize and add the fragment to our activity if it doesn't exist. If it exists, we simply attach to it. When the tab is unselected, the `onTabUnselected(Tab tab, FragmentTransaction ft)` method is called. In this method, we detach the fragment because it shouldn't be displayed. When the tab is reselected, the `onTabReselected(Tab tab, FragmentTransaction ft)` method is called. We do nothing in this method. In the `Chapter1ActionBarTabActivity` class, we create and set up the action bar. Layout for our activity has nothing but a `LinearLayout` layout and we use fragments for the user interface. Firstly, we set the navigation mode of action bar to `ActionBar.NAVIGATION_MODE_TABS` because we want tabbed navigation. Then we create two tabs, set their `TabListener` events, and add them to the `action bar` instance. When you run the application, you will see two tabs named **FIRST TAB** and **SECOND TAB**. The first tab will display **Fragment A** and the second tab will display **Fragment B**. The screen will look like the following:



It is important not to forget to set the minimum SDK level to API Level 11 or higher, because the action bar was introduced in API Level 11.

Summary

In this chapter, you learned how to use the action bar as this approach is more consistent than using the options menu. You also saw how to create custom layouts in the action bar using the ActionProvider. You learned how to use ShareActionProvider and how it is an effective way of implementing sharing in your app. You learned how to use the action view and how to make it collapsible. Finally, you learned how to use a the ction bar for tabbed navigation. It has the advantages of adapting itself to device screen size, so it is better to use the action bar than using older APIs. In the next chapter, we are going to learn about an Android layout called GridLayout and we will see how to add and configure it.

2
A New Layout – GridLayout

A new layout is introduced with Android Ice Cream Sandwich known as the **GridLayout**. This layout is an optimized layout and could be used instead of **LinearLayout** and **RelativeLayout**. This chapter shows how to use and configure GridLayout.

The topics covered in this chapter are as follows:

- Why to use GridLayout
- Adding a GridLayout
- Configuring GridLayout

GridLayout

GridLayout is a layout that divides its view space into rows, columns, and cells. GridLayout places views in it automatically, but it is also possible to define the column and row index to place a view into GridLayout. With the span property of cells, it is possible to make a view span multiple rows or columns. The following code block shows a sample layout file using a `GridLayout` layout:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/GridLayout1"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:columnCount="2"
android:orientation="horizontal" android:rowCount="2">

<TextView
android:id="@+id/textView1"
android:text="Cell 1,1"
```

```
android:textAppearance="?android:attr/textAppearanceLarge" />

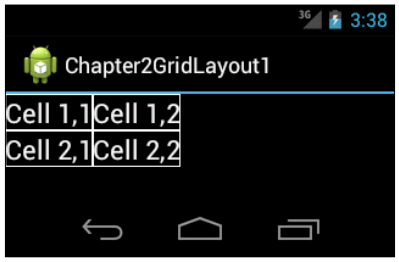
<TextView
android:id="@+id/textView2"
android:text="Cell 1,2"
android:textAppearance="?android:attr/textAppearanceLarge" />

<TextView
android:id="@+id/textView3"
android:text="Cell 2,1"
android:textAppearance="?android:attr/textAppearanceLarge" />

<TextView
android:id="@+id/textView4"
android:text="Cell 2,2"
android:textAppearance="?android:attr/textAppearanceLarge" />

</GridLayout>
```

When this layout XML is viewed in the emulator it will look like the following screenshot:



In this layout XML file, we placed four `TextView` components with the texts `Cell 1,1`, `Cell 1,2`, `Cell 2,1`, and `Cell 2,2`. With the GridLayout `orientation` set to `horizontal` and the `columnCount` and `rowCount` properties set to `2`, GridLayout firstly places items automatically to the first row, and when the number of items reaches `columnCount`; it starts placing items in the second row.

The first thing you will notice in this layout is that `TextView` components don't have `layout_width` and `layout_height` properties. These properties are not used because `GridLayout` uses the `layout_gravity` property for determining the size of cells instead of `layout_width` and `layout_height` properties. Normally the `gravity` property is used to align the content of a view, but in `GridLayout` it is used for a different purpose. The available gravity constants include `left`, `top`, `right`, `bottom`, `center_horizontal`, `center_vertical`, `center`, `fill_horizontal`, `fill_vertical`, and `fill`.

In `GridLayout`, you can explicitly define the cell that a view will be placed in by specifying the index of the column and row. If the index is not specified, `GridLayout` will automatically place the views to the first available position according to the orientation of the `GridLayout` layout.

Why to use GridLayout

LinearLayout and **RelativeLayout** are the most common layouts used in user interface design in Android. For simple user interfaces they are a good choice but when the user interface gets complicated, the use of nested LinearLayout tends to increase. Nested layouts (of any type) can hurt performance, and furthermore nested LinearLayout deeper than 10 may cause a crash in your application. Thus, you should either avoid using too many nested LinearLayout blocks or you should use RelativeLayout in order to decrease nested LinearLayout blocks. Another drawback of these layouts for complicated user interfaces is the difficulty in readability. It is difficult to maintain nested LinearLayout or RelativeLayout layouts that have many views. It is a good choice to use **GridLayout** in these cases. Too many nested LinearLayouts could be avoided by using GridLayout. Furthermore, it is much easier to maintain GridLayout. Many of the user interfaces that use LinearLayout**,** RelativeLayout, or **TableLayout** can be converted to GridLayout where GridLayout will provide performance enhancements. One of the major advantages of GridLayout over other layouts is that you can control the alignment of a view in both horizontal and vertical axes.

Adding a GridLayout

In this section we are going to migrate an Android application from **LinearLayout** to **GridLayout**. The layout's XML code of the application with `LinearLayout` is shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical" android:background="#ffffff">
<!-- we used 3 nested LinearLayout-->
<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="horizontal" >
<!—LinearLayout that contains labels-->
<LinearLayout
```

```
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:orientation="vertical" >

<TextView
android:id="@+id/textView1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Username:"
android:textAppearance="?android:attr/textAppearanceSmall"
android:textColor="#000000" android:layout_gravity="right"/>

<TextView
android:id="@+id/textView2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Password:"
android:textAppearance="?android:attr/textAppearanceLarge"
android:textColor="#000000" />
</LinearLayout>
<!—Linearlayout that contains fields-->
<LinearLayout
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="vertical" >

<EditText
android:id="@+id/editText1"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:background="@drawable/borders_bottom_right"
android:ems="10" >

</EditText>

<EditText
android:id="@+id/editText2"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_weight="1"
android:background="@drawable/borders_bottom_right"
android:ems="10" />
</LinearLayout>
```

```
</LinearLayout>

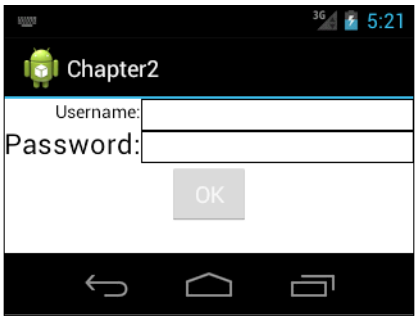
<Button
android:id="@+id/button1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="OK" android:layout_gravity="center_horizontal"/>

</LinearLayout>
```

The drawable `borders_bottom_right` background used in the preceding layout file is shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
  <layer-list
  xmlns:android="http://schemas.android.com/apk/res/android" >
  <item>
    <shape android:shape="rectangle">
    <stroke android:width="1dp" android:color="#FFFFFF" />
    <solid android:color="#000000" />
    </shape>
  </item>
  </layer-list>
```

The screen will look like the following:



As you can see in the layout XML code, we used three nested `LinearLayout` instances in order to achieve a simple login screen. If this screen was designed with `GridLayout`, the XML code of the layout would look like the following code block:

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

```
    android:background="#ffffff"
    android:columnCount="2"
    android:orientation="horizontal" >

    <TextView
      android:id="@+id/textView1"
      android:layout_gravity="right"
      android:text="Username:" android:textColor="#000000"/>

    <EditText
    android:id="@+id/editText1"
    android:ems="10" android:background="@drawable/borders"/>

    <TextView
      android:id="@+id/textView2"
      android:text="Password:"
      android:textAppearance="?android:attr/textAppearanceLarge"
      android:textColor="#000000"/>

    <EditText
      android:id="@+id/editText2"
      android:ems="10" android:background="@drawable/borders">

    </EditText>

    <Button
      android:id="@+id/button1"
      android:layout_columnSpan="2"
      android:text="Button"
      android:layout_gravity="center_horizontal"/>

</GridLayout>
```

We set the `columnCount` property to `2` because we have a `TextView` component and an `EditText` component in a row. After that we placed the views and didn't specify the row or column index. **GridLayout** placed them automatically according to `orientation` and `columnCount`. We set the `layout_columnSpan` property to `2` in order to make the button span two columns. With the `layout_gravity` property we made the button appear in the center of the row. As you can see in the XML code of the layout, it is very simple and easy to design the same screen with GridLayout. Furthermore, alignments are much easier with GridLayout and this code has better readability.

GridLayout has been available since API Level 14, so the minimum SDK property in the `AndroidManifest.xml` file should be set to `14` or greater as shown in the following code line:

```
<uses-sdkandroid:minSdkVersion="14" />
```

Configuring GridLayout

Firstly, we will write a sample `GridLayout` XML code and then we will use this code as a base for other examples. The XML code of the sample layout will look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:columnCount="3"
android:rowCount="3" >

<TextView
android:text="[1,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[1,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[1,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,3]"
```

```
android:textAppearance="?android:attr/textAppearanceLarge"/>

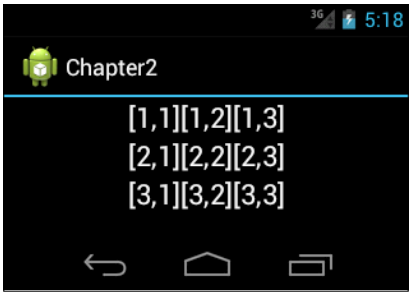
<TextView
android:text="[3,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

</GridLayout>
```

Using the preceding code block the screen will look like the following:



As you can see in the layout XML code, `TextView` components are placed without indices and they are automatically positioned in `GridLayout` according to `orientation`, `columnCount`, and `rowCount`. Now we will set the index number of `[1, 3]` to `[2, 1]`. The layout XML code should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:columnCount="3"
android:rowCount="3" >

<TextView
android:text="[1,1]"
```

```
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[1,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
<!-- set the row and column index with layout_row and layout_column
properties-->
<TextView
android:text="[1,3]"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_row="1" android:layout_column="1"/>

<TextView
android:text="[2,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

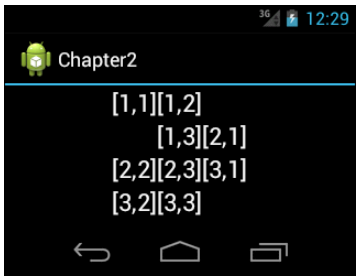
<TextView
android:text="[3,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

</GridLayout>
```

The screen should look like the following:



As you see in the layout XML code (the highlighted part), we set the row and column index with `layout_row` and `layout_column` properties. The index is zero-based, thus the `TextView` component with text `[1, 3]` is placed to the second row and second column. The interesting part here is that the `TextView` component with text `[2, 1]` is placed after `[1, 3]`. This is because `[2, 1]` doesn't have an index and **GridLayout** continues positioning after `[1, 3]`. That is **GridLayout** looks for the first available position after the last placed view. Another noteworthy thing is that after shifting indices, the row count increased to `4` although we set the row count to `3`. **GridLayout** doesn't throw exception in such cases.

In the following sample, we will swap `[1, 2]` and `[2, 2]`.The layout XML code should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:columnCount="3"
android:rowCount="3" >

<TextView
android:text="[1,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
<!-- set layout_row of [1, 2] to 1-->
<TextView
android:text="[1,2]"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_row="1"/>
<!-- set layout_row of [1, 2] to 1-->
<TextView
android:text="[1,3]"
```

```
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_row="0"/>

<TextView
android:text="[2,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
<!-- set the layout_row of [2, 2] to 0 and layout_column to 1-->
<TextView
android:text="[2,2]"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_row="0" android:layout_column="1"/>
<!-- set layout_row of [2, 3] to 1 in order to make it appear after
[1,2]'s
new position-->
<TextView
android:text="[2,3]"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_row="1"/>

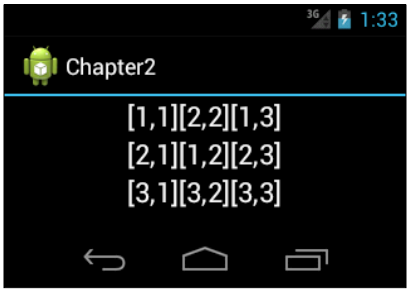
<TextView
android:text="[3,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

</GridLayout>
```

The screen should look like the following:



As you see in the layout XML code, we firstly set `layout_row` of `[1, 2]` to `1`. By this way, it will appear in place of `[2, 2]`. Then we have to set `layout_row` of `[1, 3]` to `0` and `layout_column` to `2`, because the cursor position of `GridLayout` was changed by setting the index of `[1, 2]`. If we don't change the index of `[1, 3]`, it will be placed after the `[1, 2]` index's new position. After that, in order to make `[2, 2]` appear in position of `[1, 2]`, we set the `layout_row` of `[2, 2]` to `0` and `layout_column` to `1`. Lastly, we have to set `layout_row` of `[2, 3]` to `1` in order to make it appear after `[1, 2]` index's new position. It seems a little complex to configure views in **GridLayout**, but if you try it in an emulator, you will see that it isn't that difficult.

In the following sample code, we will delete `[2, 2]` and make `[1, 2]` to span two rows. The layout XML code look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
android:columnCount="3"
android:rowCount="3" >

<TextView
android:text="[1,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
<!-- set layout_rowSpan property of [1,2] to 2. By this way [1,2] will
cover 2 rows.-->
<TextView
android:text="[1,2]"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_rowSpan="2" android:layout_gravity="fill"
android:gravity="center"/>

<TextView
android:text="[1,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
```

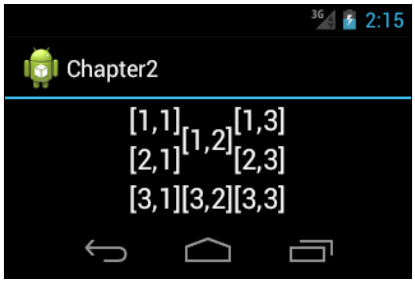
```
<TextView
android:text="[3,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

</GridLayout>
```

Using the preceding code block we get the following screen:



As you can see in the layout XML code, we deleted the cell `[2,2]` and set the `layout_rowSpan` property of `[1,2]` to `2`. By this way, `[1,2]` will cover two rows. We set the `layout_gravity` property to `fill` in order to make it fill the space of two rows. Then we set the `gravity` property to `center` in order to make the content of the `TextView` component to align to the center of space that it covers.

A new view – Space

Space is a new view introduced with Android Ice Cream Sandwich. It is used for putting spaces between views. It is very useful in **GridLayout**. In the following sample layout XML code, we removed `TextView` components with text `[2, 2]` and `[2, 3]` and then put **Space** instead of them as shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_gravity="center_horizontal"
```

```
android:columnCount="3"
android:rowCount="3" >

<TextView
android:text="[1,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[1,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[1,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[2,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>
<Space
android:layout_row="1"
android:layout_column="1"
android:layout_columnSpan="2"
android:layout_gravity="fill"/>

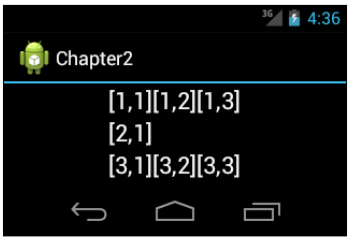
<TextView
android:text="[3,1]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,2]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

<TextView
android:text="[3,3]"
android:textAppearance="?android:attr/textAppearanceLarge"/>

</GridLayout>
```

As you can see in the layout XML code, we removed `TextView` components with text `[2,2]` and `[2,3]`. We put a `Space` view to row `1` and column `1`. We set `layout_columnSpan` to `2` in order to make it span two columns. The screen will look like the following:



Summary

LinearLayout and **RelativeLayout** are the most common layouts in Android application development. However, when designing complex user interfaces you may need to use nested LinearLayout or RelativeLayout. This is a drawback in the performance and readability of your code because these layouts increase the view hierarchy which results in unnecessary iterations on view refreshes. **GridLayout** is a new layout introduced in Android Ice Cream Sandwich that overcomes these kinds of issues. You may design user interfaces without the need for nesting layouts. If you are developing applications for API Level 14 and more, it is better to use GridLayout. In the next chapter, we are going to learn about the new Social APIs introduced with Android Ice Cream Sandwich.

3
Social APIs

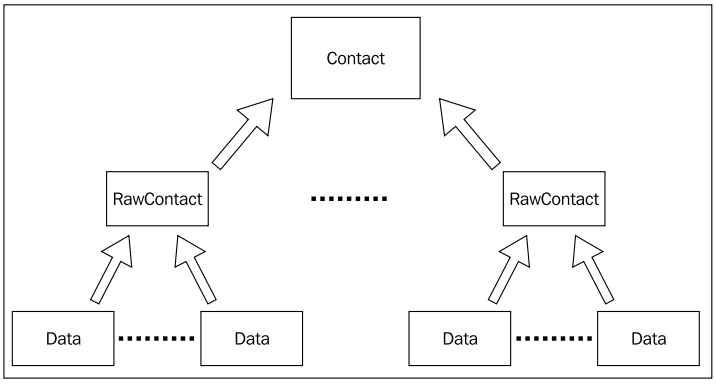
New Social APIs have been introduced with Android Ice Cream Sandwich and this API makes it easy to integrate social networks. Furthermore, high resolution photos could be used as contact photos after Android Ice Cream Sandwich was released. This chapter shows Social API usage with examples.

The topics covered in this chapter are as follows:

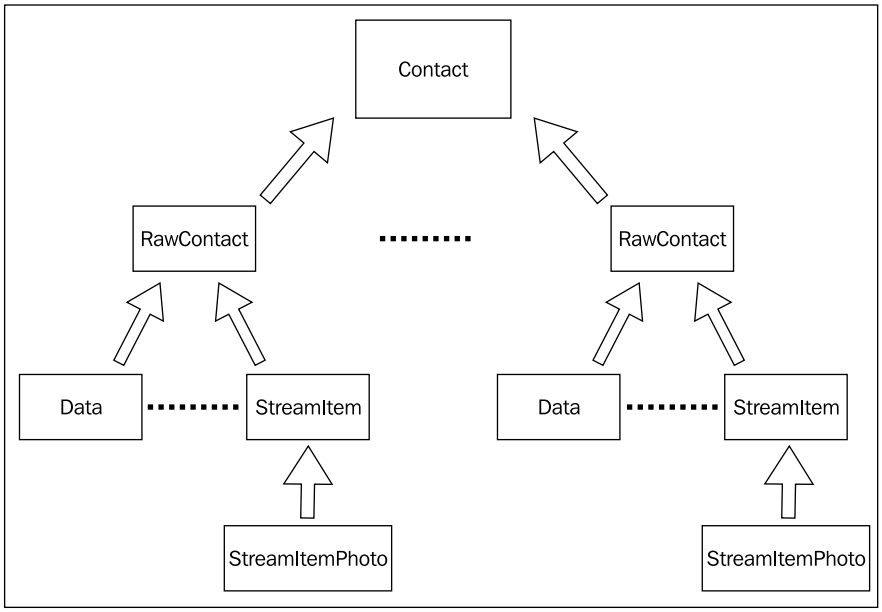
- Basics of contacts in Android
- Using the Social API

Basics of contacts in Android

A person may have multiple contact information details. In Android, these multiple contact information details are joined and displayed as one contact detail. For instance; a person may have a Google+ contact, a Skype contact, and a phone contact, and Android joins all these contacts into one contact. Each of these sources of contacts is **RawContact**. Each RawContact has one or more data row, which holds some data about the contact such as phone number, e-mail, and so on. Refer to the following block diagram for a better understanding of their relationship:



Each RawContact has support for storing social network streams—texts and photos—with Android Ice Cream Sandwich. Each RawContact is associated with **StreamItems** which contains texts, timestamp, and comments from social media updates, such as Google+, and each StreamItem is associated with **StreamItemPhotos** which contains photos (such as photos in a Google+ post). However, there is a limit for the number of StreamItems stored in RawContact. This number can be fetched with a query with `StreamItems.CONTENT_LIMIT_URI URI`. When the number exceeds the limit, the stream item with the oldest time stamp is removed. The following block diagram depicts the relationship between these blocks:



Using Social API

In the following example, we are going to show how to add **StreamItem** and then how to display added StreamItems. Firstly, we inserted two buttons into the user interface, one for triggering an insert and one for listing StreamItems. In order to display StreamItems, we put three `TextView` components in the layout. The layout XML should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

```
    <!-- we put two buttons to the user interface, one for triggering
    insert and one for listing stream items-->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <Button
            android:id="@+id/buttonInsert"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Insert" />

        <Button
            android:id="@+id/buttonList"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="List" />
    </LinearLayout>
    <!-- In order to display stream items, we put three TextViews to
    the layout-->
    <TextView
        android:id="@+id/txt1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"/>
    <TextView
        android:id="@+id/txt2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/txt3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge" />
</LinearLayout>
```

We will implement the `Activity` class step-by-step that firstly adds a contact and then adds `StreamItems` and displays them. The `Activity` class with the `onCreate()` method is shown in the following code block:

```
package com.chapter3;

import java.io.IOException;
import java.io.InputStream;
```

```
import java.util.Calendar;
import android.app.Activity;
import android.content.ContentResolver;
import android.content.ContentUris;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.provider.ContactsContract.CommonDataKinds.Email;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.provider.ContactsContract.CommonDataKinds.
StructuredName;
import android.provider.ContactsContract.Data;
import android.provider.ContactsContract.RawContacts;
import android.provider.ContactsContract.StreamItemPhotos;
import android.provider.ContactsContract.StreamItems;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class Chapter3_1Activity extends Activity implements
OnClickListener {

  Button insertButton;
  Button listButton;
  Button chooseButton;
  TextView txt1;
  TextView txt2;
  TextView txt3;
  long rawContactId;


  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
              //initialize UI components
    insertButton = (Button) this.findViewById(R.id.buttonInsert);
    insertButton.setOnClickListener(this);
    listButton = (Button) this.findViewById(R.id.buttonList);
    listButton.setOnClickListener(this);
    txt1 = (TextView) this.findViewById(R.id.txt1);
```

```
        txt2 = (TextView) this.findViewById(R.id.txt2);
        txt3 = (TextView) this.findViewById(R.id.txt3);

      }
           @Override
      public void onClick(View v) {
                 // when the insert button is clicked, addContact method
                 // is called
        if (v == insertButton)
          this.rawContactId = addContact("Murat Aydın", "9999999",
              "maydin@gmail.com", "Murat", "com.google");
        else if (v == listButton) {
          getStreams(this.rawContactId);
        }
      }

    }
```

As you can see in this code, we firstly get instances of `Button` and `TextView` in the layout in the `onCreate(Bundle savedInstanceState)` method. The `Chapter3_1Activity` class implements `OnClickListener` for the buttons. As you can see in the `onClick(View v)` method, when the **Insert** button is clicked, the `addContact()` method is called. The `addContact()` method is defined as follows:

```
    public long addContact(String name, String phone, String email,
          String accountName, String accountType) {
                 // firstly a raw contact is created with the
                 // addRawContact method
        Uri rawContactUri = addRawContact(accountName, accountType);

        if (rawContactUri != null) {
          long rawContactId = ContentUris.parseId(rawContactUri);
                      // we use the ID of the created raw contact in
                      // creating name, email, phone number and stream
                      // items
          addName(name, rawContactId);

          addPhoneNumber(phone, rawContactId);

          addEmail(email, rawContactId);

          addContactStreamItem(rawContactId, accountName,
          accountType, "Social Media Update 1");
          addContactStreamItem(rawContactId, accountName,
          accountType, "Social Media Update 2");
          addContactStreamItem(rawContactId, accountName,
```

```
        accountType, "Social Media Update 3");

        return rawContactId;
      }
      return 0;
    }
```

In the `addContact()` method, firstly a RawContact is created with the `addRawContact()` method. In the `addRawContact()` method, we use `accountName` and `accountType` to create a raw contact. The `addRawContact()` method is defined as follows:

```
public Uri addRawContact(String accountName, String accountType) {
            // we use account name and type to create a raw contact
    ContentResolver cr = getContentResolver();
    ContentValues values = new ContentValues();
    values.put(RawContacts.ACCOUNT_TYPE, accountType);
    values.put(RawContacts.ACCOUNT_NAME, accountName);
    Uri rawContactUri = cr.insert(RawContacts.CONTENT_URI,
    values);
    return rawContactUri;
  }
```

After the raw contact is created, we use the ID of the created raw contact in creating the name, e-mail, phone number, and StreamItems. `addName()`, `addEmail()`, and `addPhoneNumber()` methods are using `ContentValues` class to create the name, e-mail, and phone number data as shown in the following code block:

```
      // This method is for creating email data
  private void addEmail(String email, long rawContactId) {
    ContentResolver cr = getContentResolver();
    ContentValues values = new ContentValues();
    values.put(Email.ADDRESS, email);
    values.put(Email.TYPE, Email.TYPE_OTHER);
    values.put(Email.MIMETYPE, Email.CONTENT_ITEM_TYPE);
    values.put(Data.RAW_CONTACT_ID, rawContactId);
    cr.insert(Data.CONTENT_URI, values);
  }
      //This method is for creating phone number data
  private void addPhoneNumber(String phone, long rawContactId) {

    ContentResolver cr = getContentResolver();
    ContentValues values = new ContentValues();
    values.put(Phone.NUMBER, phone);
    values.put(Phone.TYPE, Phone.TYPE_OTHER);
```

```
    values.put(Phone.MIMETYPE, Phone.CONTENT_ITEM_TYPE);
    values.put(Data.RAW_CONTACT_ID, rawContactId);
    cr.insert(Data.CONTENT_URI, values);
  }
      //This method is for adding name data
  private void addName(String name, long rawContactId) {
    ContentValues values = new ContentValues();
    values.put(Data.RAW_CONTACT_ID, rawContactId);
    values.put(Data.MIMETYPE, StructuredName.CONTENT_ITEM_TYPE);
    values.put(StructuredName.DISPLAY_NAME, name);
    getContentResolver().insert(Data.CONTENT_URI, values);
  }
```

In the `addContactStreamItem()` method, we create the StreamItems. We provide the raw contact ID, text of the StreamItem, time stamp in milliseconds in which the StreamItem is created, account name, and type to create StreamItems. Raw contact ID, account name, and type are required fields for creating a StreamItem. The `addContactStreamItem()` method is defined as follows:

```
      //StreamItems are created in this method
  private long addContactStreamItem(long rawContactId, String
accountName,
    String accountType, String text) {
            // Raw contact ID, account name and type are required
            // fields for creating a stream item.

    ContentResolver cr = getContentResolver();
    ContentValues values = new ContentValues();
    values.put(StreamItems.RAW_CONTACT_ID, rawContactId);
    values.put(StreamItems.TEXT, text);
    values.put(StreamItems.TIMESTAMP,
    Calendar.getInstance().getTime()
      .getTime());
    Uri.Builder builder = StreamItems.CONTENT_URI.buildUpon();
    builder.appendQueryParameter(RawContacts.ACCOUNT_NAME,
    accountName);
    builder.appendQueryParameter(RawContacts.ACCOUNT_TYPE,
    accountType);
    Uri streamItemUri = cr.insert(builder.build(), values);
    long streamItemId = ContentUris.parseId(streamItemUri);

    addContactStreamPhoto(streamItemId, accountName, accountType);

    return streamItemId;
  }
```

The `addContactStreamPhoto()` method is used for creating StreamItemPhotos for a StreamItem. We have to provide a photo in binary, or `PHOTO_FILE_ID`, or `PHOTO_URI`. As you can see in the following code block, we used a drawable to create a photo in binary using `loadPhotoFromResource` and `readInputStream` methods. We also provide the StreamItem ID, sort index, account name, and type for creating a stream photo. If we don't provide a sort index, the ID column will be used for sorting. The `addContactStreamPhoto()` method is defined as follows:

```
     //This method is used for creating a stream photo for a stream
     item
private long addContactStreamPhoto(long streamItemId,String
accountName,
    String accountType) {
           // provide stream item ID, sort index, account name
           and type for creating a stream photo
  ContentValues values = new ContentValues();
  values.put(StreamItemPhotos.STREAM_ITEM_ID, streamItemId);
  values.put(StreamItemPhotos.SORT_INDEX, 1);
  values.put(StreamItemPhotos.PHOTO,
  loadPhotoFromResource(R.drawable.ic_launcher));
  Uri.Builder builder =
  StreamItems.CONTENT_PHOTO_URI.buildUpon();
  builder.appendQueryParameter(RawContacts.ACCOUNT_NAME,
  accountName);
  builder.appendQueryParameter(RawContacts.ACCOUNT_TYPE,
  accountType);
  Uri photoUri = getContentResolver().insert(builder.build(),
  values);
  long photoId = ContentUris.parseId(photoUri);
  return photoId;
}
//This method is used for creating a photo in binary
private byte[] loadPhotoFromResource(int resourceId) {
      InputStream is =
      getResources().openRawResource(resourceId);
      return readInputStream(is);
  }
private byte[] readInputStream(InputStream is) {
      try {
          byte[] buffer = new byte[is.available()];
          is.read(buffer);
          is.close();
          return buffer;
      } catch (IOException e) {
```

```
            throw new RuntimeException(e);
        }
    }
```

When the **List** button is clicked, the `getStreams()` method is called. As you can see in the following code, in the `getStream()` method, we firstly retrieve the `contactId` details of the raw contact using the `getContactId()` method. Then we use this contact ID in querying StreamItems by passing `contactId` as the search parameter. Since we query the StreamItems, `ContactsContract.StreamItems.CONTENT_URI` is used as the URI. Lastly, StreamItems are retrieved with a cursor and texts of StreamItems are displayed in TextViews. The `getStreams()` method and the `getContactId()` method are defined as follows:

```
public void getStreams(long rawContactId) {
  long contactId = getContactId(rawContactId);
  ContentResolver cr = getContentResolver();
  Cursor pCur = cr.query(ContactsContract.StreamItems.CONTENT_URI,
  null,
      ContactsContract.StreamItems.CONTACT_ID + " = ?",
      new String[] { String.valueOf(contactId) }, null);
  int i = 0;
  if (pCur.getCount() > 0) {
    while (pCur.moveToNext()) {
      String text = pCur.getString(pCur
  .getColumnIndex(ContactsContract.StreamItems.TEXT));
      if (i == 0)
        this.txt1.setText(text);
      else if (i == 1)
        this.txt2.setText(text);
      else if (i == 2)
        this.txt3.setText(text);
      i++;

    }
  }
          pCur.close();
}
public long getContactId(long rawContactId) {
  Cursor cur = null;
  try {
    cur = this.getContentResolver().query(
        ContactsContract.RawContacts.CONTENT_URI,
        new String[] {
        ContactsContract.RawContacts.CONTACT_ID },
```

```
            ContactsContract.RawContacts._ID + "=" +
            rawContactId, null, null);
        if (cur.moveToFirst()) {
          return cur
              .getLong(cur
                  .getColumnIndex(ContactsContract.
                  RawContacts.CONTACT_ID));
        }
      } catch (Exception e) {
        e.printStackTrace();
      } finally {
        if (cur != null) {
          cur.close();
        }
      }
      return -1l;
    }
```

Lastly, we need some permissions for reading and writing social streams and contacts: `READ_SOCIAL_STREAM`, `WRITE_SOCIAL_STREAM`, `READ_CONTACTS`, and `WRITE_CONTACTS`. Furthermore, we have to set the minimum SDK to the API Level 15 in order to use Social APIs. The `AndroidManifest.xml` file should look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter3"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="15" />
    <uses-permission android:name="android.permission.READ_CONTACTS"
    />
    <uses-permission android:name="android.permission.READ_SOCIAL_
    STREAM" />
    <uses-permission android:name="android.permission.WRITE_SOCIAL_
    STREAM" />
    <uses-permission android:name="android.permission.WRITE_CONTACTS"
    />

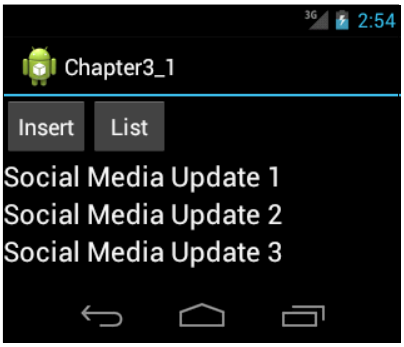
    <application
        android:icon="@drawable/ic_launcher"
```

```
            android:label="@string/app_name" >
            <activity
                android:name=".Chapter3_1Activity"
                android:label="@string/app_name" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

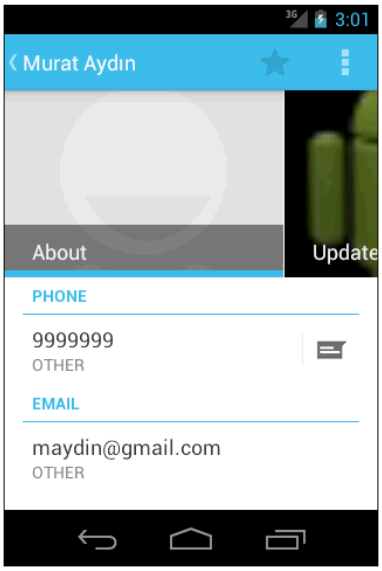
                    <category
                    android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
        </application>

    </manifest>
```

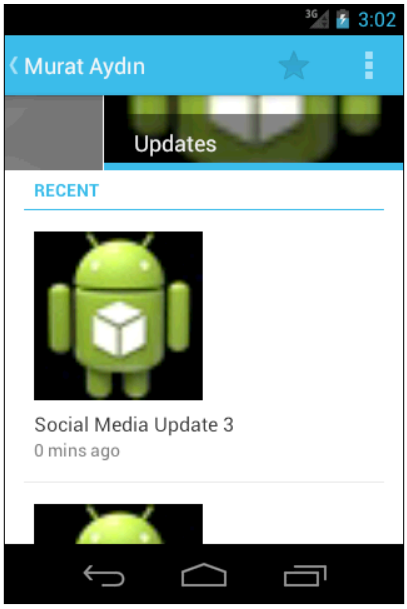
When we execute the application in the emulator, click the **Insert** button and then click the **List** button; the screen will look like the following:



When you execute the `People` app in the emulator, you will see that the contact named **Murat Aydın** is created as seen in the following screenshot:

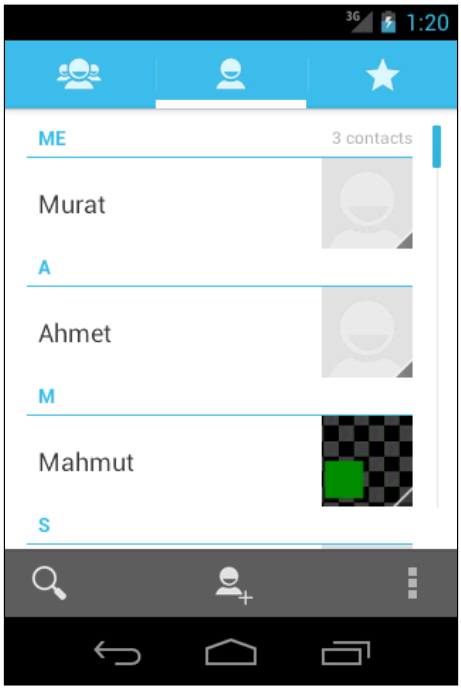


You will also see the recent social network updates with photos that we created programmatically:



Device user profile

Starting from API Level 14, Android displays the device user profile at the top of the contacts as **ME** as seen in the following screen:



`ContactsContract.Profile. CONTENT_URI and ContactsContract.Profile. CONTENT_RAW_CONTACTS_URI` URIs could be used in order to read and write the device user profile. Operations are similar to reading and writing a contact except that `READ_PROFILE` and `WRITE_PROFILE` permissions are needed in the `AndroidManifest.xml` file.

Summary

Integration of contacts and social networks became easier with the new Social APIs introduced with Android Ice Cream Sandwich. `StreamItems` and `StreamItemPhotos` classes are used for storing social network updates to be stored in contacts. In this chapter, we learned how to use these classes. Furthermore, we learned the device user profile that displays the contact information of the device user.

New APIs have been introduced with Android Ice Cream Sandwich for managing calendars. In the next chapter, we will learn the new Calendar API and how to use it.

4
Calendar APIs

New Calendar APIs have been introduced with Android Ice Cream Sandwich for managing calendars. Event, attendee, alert, and reminder databases can be managed with these APIs. These APIs allow us to easily integrate calendars with our Android applications. This chapter shows how to use Calendar APIs with examples.

The topics covered in this chapter are as follows:

- Using Calendar APIs
- Creating an event
- Adding an attendee
- Adding a reminder

Using Calendar APIs

The main class that manages the calendar data is the `CalendarContract` class. Noteworthy tables that store the calendar information are as follows:

- `CalendarContract.Calendar`: This table stores calendar specific data for each calendar
- `CalendarContract.Event`: This table stores event specific data for each event
- `CalendarContract.Attendee`: This table stores data about the attendee of an event
- `CalendarContract.Reminder`: This table stores data about the reminder for an event



In the following examples, we will execute the applications in an Android device, because in order to test the Calendar API in an emulator, an account is needed. If you want to test examples in an emulator, make sure to choose the **Google API**'s API Level 14 or higher when creating the **AVD**. The Google API allows you to add a Google account to an emulator, which is needed for the Calendar APIs. You also need to set up the Calendar to sync with Gmail. You can use `m.google.com` as the server and `your_email@gmail.com` as the Domain/Username when adding an account. After creating and syncing your account, you can run the following examples in the emulator.

Creating an event

In order to create a calendar event, we need to create a `ContentValues` instance and put event information to this instance. Then, using the `ContentResolver` class, we could insert the event information into the calendar. There are some required fields in order to insert an event in to calendar. These fields are as follows:

- Start time of the event
- End time of the event if the event is not repeating
- Recurrence rule or recurrence date of the event if the event is repeating
- Duration if the event is repeating
- Event time zone and calendar ID

The `Activity` class that inserts an event is defined as follows:

```
package com.chapter4;

import java.util.Calendar;
import java.util.TimeZone;

import android.app.Activity;
import android.content.ContentValues;
import android.database.Cursor;
import android.os.Bundle;
import android.content.ContentUris;
import android.net.Uri;
import android.provider.CalendarContract;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
```

```
public class Chapter4_1Activity extends Activity implements
OnClickListener {

  Button insertButton;
  long calendarID;
  long eventID;


    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        insertButton = (Button)this.findViewById(R.
        id.buttonInsertEvent);
        insertButton.setOnClickListener(this);
    }

  @Override
  public void onClick(View v) {

    addEvent();

  }

  private void addEvent() {
    calendarID = getCalendarID();
    ContentValues eventValues = new ContentValues ();
             // provide the required fields for creating an event to
             // ContentValues instance and insert event using
             // ContentResolver
    eventValues.put
    (CalendarContract.Events.CALENDAR_ID,calendarID);
    eventValues.put (CalendarContract.Events.TITLE,"Event 1");
    eventValues.put (CalendarContract.Events.DESCRIPTION,
    "Testing Calendar API");
    eventValues.put
(CalendarContract.Events.DTSTART,Calendar.getInstance().
getTimeInMillis());
    eventValues.put
(CalendarContract.Events.DTEND,Calendar.getInstance().
getTimeInMillis());

  eventValues.put(CalendarContract.Events.EVENT_TIMEZONE,
```

```
TimeZone.getDefault().toString());



    Uri eventUri = this.getContentResolver().insert
(CalendarContract.Events.CONTENT_URI, eventValues);
             eventID = ContentUris.parseId(eventUri);
  }
  // we use this method in order to get the ID of the calendar because
  // calendar ID is a required field in creating an event
  public long getCalendarID() {
    Cursor cur = null;
    try {
                // provide CalendarContract.Calendars.CONTENT_URI to
                // ContentResolver to query calendars
      cur = this.getContentResolver().query(
          CalendarContract.Calendars.CONTENT_URI,
          null,null,null, null);
      if (cur.moveToFirst()) {
        return cur
            .getLong(cur

  .getColumnIndex(CalendarContract.Calendars._ID));
      }
      } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (cur != null) {
      cur.close();
      }
    }
    return -1L;
  }
}
```

As you can see in this code, we use the `getCalendarID()` method in order to get the ID of the calendar because `calendarID` is a required field in creating an event. We provided `CalendarContract.Calendars.CONTENT_URI` to `ContentResolver` to query calendars.

We used a button click event in order to add an event. On the clicking of this button, we call the `addEvent()` method. In the `addEvent()` method, we provide the required fields for creating an event to the `ContentValues` instance and insert the event using the `ContentResolver`. We provide `CalendarContract.Events.CONTENT_URI` to `ContentResolver` in order to add an event.

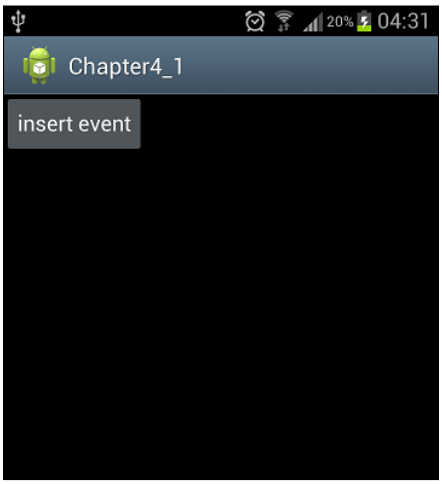
The XML code of the layout of this application is `LinearLayout` with a `Button` component as seen in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/buttonInsertEvent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="insert event" />

</LinearLayout>
```

The screen will look like the following when you execute this code:



In order to use the new Calendar APIs, the minimum SDK version in the `AndroidManifest.xml` file should be API Level 14 or more. Furthermore, `WRITE_CALENDAR` and `READ_CALENDAR` permissions are required for reading and writing to the calendar. The `AndroidManifest.xml` file should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter4"
    android:versionCode="1"
```

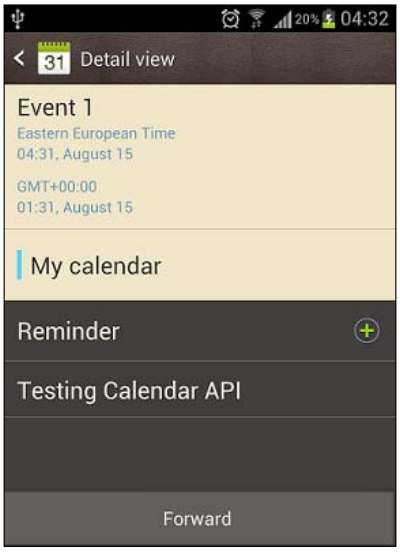
```
        android:versionName="1.0" >

        <uses-sdk android:minSdkVersion="14" />
    <uses-permission android:name="android.permission.WRITE_CALENDAR" />
    <uses-permission android:name="android.permission.READ_CALENDAR" />
        <application
            android:icon="@drawable/ic_launcher"
            android:label="@string/app_name" >
            <activity
                android:name=".Chapter4_1Activity"
                android:label="@string/app_name" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

                    <category android:name="android.intent.category.
                    LAUNCHER" />
                </intent-filter>
            </activity>
        </application>

    </manifest>
```

When the event is created, the calendar will look like the following:



Using Intents for creating events

The same event could also be created using `Intent objects`. The following method shows how to add an event using `Intent objects`:

```
private void addEventUsingIntent() {
    calendarID = getCalendarID();
    Intent intent = new Intent(Intent.ACTION_INSERT)
        .setData(CalendarContract.Events.CONTENT_URI)
        .putExtra(CalendarContract.Events.DTSTART,
Calendar.getInstance().getTimeInMillis())
        .putExtra(CalendarContract.Events.DTEND,
Calendar.getInstance().getTimeInMillis())
        .putExtra(CalendarContract.Events.TITLE,"Event 1")
        .putExtra(CalendarContract.Events.DESCRIPTION,"Testing
Calendar API");
    startActivity(intent);
  }
```

We can call this method instead of the `addEvent()` method in order to create an event using `Intent` objects. By using `Intent` objects, we don't need to create a view in order to create an event. Using `Intent` objects is a best practice for modifying and showing calendars.

Adding an attendee

Adding an attendee is similar to creating an event. We use `CalendarContract.Attendees.CONTENT_URI` as the URI for inserting an attendee. The required fields for inserting an attendee are event ID, attendee e-mail, attendee relationship, attendee status, and attendee type. We put a `Button` component in the XML layout of the application. The resulting layout is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/buttonInsertEvent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="insert event" />
```

```
        <Button
            android:id="@+id/buttonInsertAttendee"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="insert attendee" />
</LinearLayout>
```

Then we call the following method on clicking the insert attendee button:

```
    private void addAttendee()
    {
      ContentValues cv = new ContentValues();
      cv.put(Attendees.ATTENDEE_NAME, "Murat AYDIN");
      cv.put(Attendees.ATTENDEE_EMAIL, "maydin@gmail.com");
      cv.put(Attendees.EVENT_ID, eventID);
      cv.put(Attendees.ATTENDEE_RELATIONSHIP,
Attendees.RELATIONSHIP_ATTENDEE);
      cv.put(Attendees.ATTENDEE_STATUS,
Attendees.ATTENDEE_STATUS_INVITED);
      cv.put(Attendees.ATTENDEE_TYPE,Attendees.TYPE_OPTIONAL);


    this.getContentResolver().insert(CalendarContract.
    Attendees.CONTENT_URI,
cv);
    }
```

Before clicking on the insert attendee button, an event should be created because we are using an event ID when inserting an attendee.

Adding a reminder

We use `CalendarContract.Reminder.CONTENT_URI` as the URI in inserting a reminder for an event. The required fields for inserting a reminder are event ID, minutes that the reminder needs to fire before the event, and method. We put a `Button` component in the XML layout of the application. The resulting layout is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

```
    <Button
        android:id="@+id/buttonInsertEvent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="insert event" />

    <Button
        android:id="@+id/buttonInsertAttendee"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="insert attendee" />

    <Button
        android:id="@+id/buttonInsertReminder"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="insert reminder" />

</LinearLayout>
```

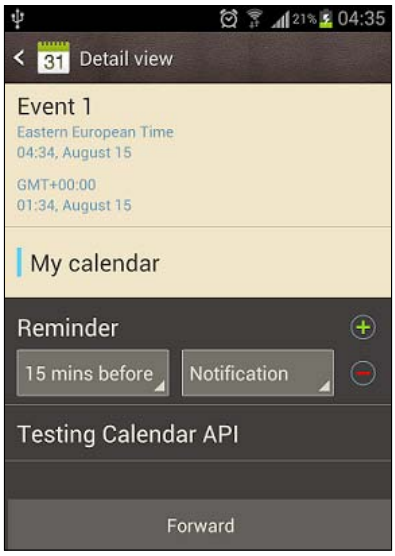
Then we call the following method on the click of the insert reminder button:

```
private void addReminder()
  {
    ContentValues values = new ContentValues();
    values.put(Reminders.MINUTES, 15);
    values.put(Reminders.EVENT_ID, eventID);
    values.put(Reminders.METHOD, Reminders.METHOD_ALERT);

  this.getContentResolver().insert(CalendarContract.
  Reminders.CONTENT_URI,
  values);
  }
```

As you see in this code, this reminder will fire 15 minutes before the event's time. Before pressing the insert reminder button, an event should be created because we are using the event ID when inserting a reminder.

When the reminder is added, the calendar will look like the following:



Summary

With the new Calendar API, integrating calendars to Android applications became easier. In this chapter, we learned how to create an event and the required fields for creating an event. Then we learned about adding an attendee and a reminder to an event. We need to set the required permissions for modifying the calendar.

Although Fragments were introduced with Android 3.0, they are now available for small screen devices with Android Ice Cream Sandwich. In the next chapter, we will cover the basics of **Fragments** and how to use them.

5
Fragments

Although **fragments** were introduced with Android 3.0, they are now available for small screen devices with Android Ice Cream Sandwich. This chapter will cover the basics of fragments and how to use them.

The topics covered in this chapter are as follows:

- Fragment basics
- Creating and managing fragments
- Types of fragments

Fragment basics

Fragment is a modular component in an activity which has its own life cycle and event handling, and it is very similar to an activity. Although fragments have their own life cycle, they are directly affected by their owner activity's life cycle. For instance, if an activity is destroyed, its fragments are also destroyed. Every fragment should have an owner activity. A fragment could be added to or removed from an activity dynamically.

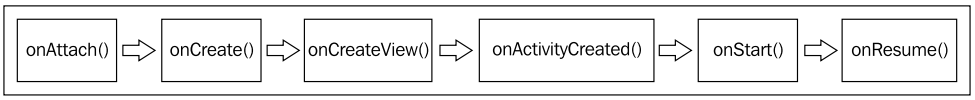
Fragments increase software reusability and provide flexibility in user interface design. A fragment could be used by more than one activity. This way you implement once and use multiple times. Furthermore, it is possible to use a fragment for different layout configurations and different screen modes. This way it provides flexibility in user interface design.



It is important to design fragments such that they could work independently, that is, they should not depend on other fragments and activities. In this way it is possible to reuse fragments independently from other fragments.

Fragment lifecycle

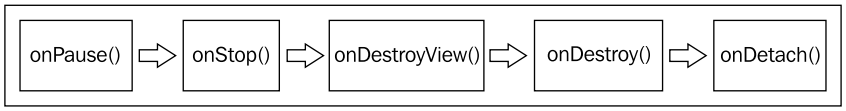
Fragments have their own lifecycle; however, they are still directly affected by their owner activity's lifecycle. The following diagram shows the creation flow of lifecycle of a fragment:



The blocks in the diagram perform the following tasks:

- `onAttach()`: When a fragment is added to an activity, the `onAttach()` method is called.
- `onCreate()`: This method is called when a fragment is created.
- `onCreateView()`: This method returns a view. This view is the user interface of the fragment. If the fragment is doing background works and doesn't have a user interface, then this method should return null.
- `onActivityCreated()`: This method is called after the owner activity is created.
- `onStart()`: After this method is called, the fragment's view becomes visible to the user.
- `onResume()`: After this method is called, the fragment becomes active and the user can interact with the fragment. This method could be called more than once, because this method is called after the application is restarted or paused.

The following diagram shows the destruction flow of the life cycle of a fragment:



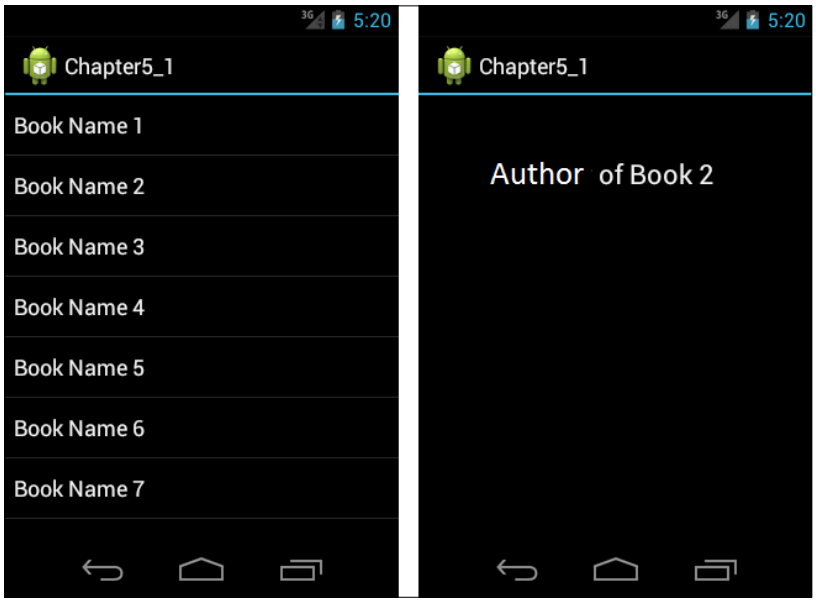
The blocks in the diagram perform the following tasks:

- `onPause()`: This method is called when the fragment is paused and no longer interacts with the user.
- `onStop()`: This method is called when the fragment is stopped. The fragment is not visible to the user after this method is called.
- `onDestroyView()`: This method is called when the view of the fragment is destroyed.

- `onDestroy()`: This method is called when the fragment is no longer in use.
- `onDetach()`: This method is called when the fragment is removed from the activity.

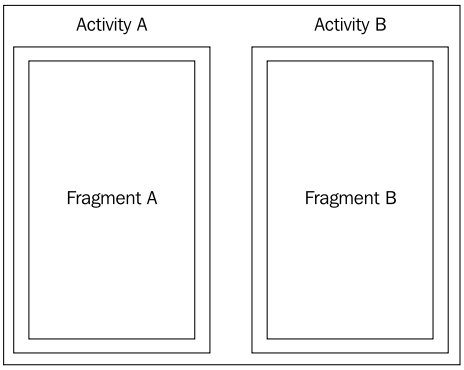
Creating and managing fragments

We are going to learn how to create and manage fragments with a sample Android application. This application is going to list book names. When a book name is clicked, the author of the book will be displayed. This application will be designed for small and large screen devices, this way we will see how to use fragments for different screen sizes. The following is the screenshot of this application for small screens. As you can see in this screenshot, the left hand side of the screen has the list of books and when a book is clicked, the right hand side of the screen will be displayed which shows the author of the clicked book:



We will firstly implement these screens, and then we will design this application for large screens.

In this application, we have two activities, one for the first screen and one for the second screen. Each activity consists of one fragment. The following diagram shows the structure of this application:



The XML code for the layout of `Fragment B` is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/textViewAuthor"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge" />

</LinearLayout>
```

As you can see in this code, it has a `LinearLayout` layout with a `TextView` component. `TextView` is for displaying the author of the book. We don't have a layout for `Fragment A`, because it is a `ListFragment` property which includes the `ListView` component.

Now we need two classes that extend the `Fragment` classes for each fragment. The following is the class for `Fragment A`:

```
package com.chapter5;

import android.app.ListFragment;
import android.content.Intent;
```

```
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;

public class Chapter5_1FragmentA extends ListFragment implements
OnItemClickListener {


  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
             super.onActivityCreated(savedInstanceState);
             //initialize the adapter and set on click events of
              items
    ArrayAdapter<String> adapter = new
    ArrayAdapter<String>(getActivity(),
        android.R.layout.simple_list_item_1, Book.BOOK_NAMES);
    this.setListAdapter(adapter);
    getListView().setOnItemClickListener(this);
  }
  @Override
  public void onItemClick(AdapterView<?> parent, View view, int
  position, long id)
  {
    //Start a new Activity in order to display author name
    String author = Book.AUTHOR_NAMES[position];
    Intent intent = new
    Intent(getActivity().getApplicationContext(),
    Chapter5_1Activity_B.class);
    intent.putExtra("author", author);
    startActivity(intent);
  }
}
```

As you can see, the `Chapter5_1FragmentA` class extends `ListFragment`, because we are listing the books in this screen. It is similar to `ListActivity` and this class has a `ListView` view. In the `onActivityCreated` method we set the `ListAdapter` property of the `ListFragment`. The source for the adapter is a class that contains the string arrays of book names and authors as shown in the following code block:

```
package com.chapter5;

public class Book {
  public static final String[] BOOK_NAMES = { "Book Name 1", "Book
  Name 2", "Book Name 3", "Book Name 4", "Book Name 5", "Book Name
  6", "Book Name 7", "Book Name 8" };
```

```
    public static final String[] AUTHOR_NAMES = { "Author of Book
    1", "Author of Book 2", "Author of Book 3", "Author of Book
    4", "Author of Book 5", "Author of Book 6", "Author of Book
    7", "Author of Book 8" };
}
```

After initializing `ListAdapter`, we set the `OnItemClickListener` event of the `ListView` view. This event is called when a `ListView` item is clicked. When an item is clicked, the `onItemClick` method is called. In this method, a new activity is started with the author of the book. As you can see in the code, we reach the owner activity of the fragment with the `getActivity()` method. We could receive the `ApplicationContext` with the `getActivity()` method. Remember that the `OnCreateView` method is called before `OnActivityCreated`, and because of that we initialized `ListAdapter` and `ListView` in the `OnActivityCreated` method, because we need the user interface components to be created before we initialize them and they are created in `OnCreateView`. We don't need to override the `OnCreateView` method of `ListFragment`, because it returns a `ListView`. You can override the `OnCreateView` method if you want to use a customized `ListView`.

The following is the class for `Fragment B`:

```
package com.chapter5;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Chapter5_1FragmentB extends Fragment {

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup
  container,
      Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.fragment_b, container,
    false);
    return view;
  }
}
```

As you can see from this code, if a fragment has a user interface, this method should be overridden and should return a view. In our sample application, we are returning a view inflated with the XML layout that we previously implemented.

Now we need two activity classes that host these fragments. The following is the `Activity` class of `Activity A` that hosts `Fragment A`:

```
package com.chapter5;

import android.app.Activity;
import android.os.Bundle;

public class Chapter5_1Activity_A extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_a);
    }
}
```

It is a simple `Activity` class that just sets the content view with a layout. The XML layout code of the `Activity A` class is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <fragment
        android:id="@+id/fragment_a"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.chapter5.Chapter5_1FragmentA" />

</LinearLayout>
```

As you can see from this code, we specified `Fragment A` with the class property `com.chapter5.Chapter5_1FragmentA`. Furthermore, we specified the `id` property. Fragments should have either an `id` or a `tag` property as an identifier because Android needs that in restoring the fragment when the activity is restarted.

The `Activity` class for `Activity B` that hosts `Fragment B` is as follows:

```
package com.chapter5;

import android.app.Activity;
import android.os.Bundle;
```

```
import android.widget.TextView;

public class Chapter5_1Activity_B extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_b);

        Bundle extras = getIntent().getExtras();
    if (extras != null) {
      String s = extras.getString("author");
      TextView view = (TextView)
      findViewById(R.id.textViewAuthor);
      view.setText(s);
    }
    }
}
```

It is a simple `Activity` class that just sets the content view with a layout. The XML layout code of `Activity B` is as follows:

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

    <fragment
        android:id="@+id/fragment_b"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.chapter5.Chapter5_1FragmentB" />

</LinearLayout>
```

As you can see from this code, we specified `Fragment B` with the class property `com.chapter5.Chapter5_1FragmentB`.

Programmatically adding a fragment

In our previous sample application, we added a fragment to an activity layout in XML layout code. You can also add a fragment to an activity programmatically. The following is the programmatically added fragment version of our previous sample application and XML layout code of the activity:

```
package com.chapter5;

import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;

public class Chapter5_1Activity_A extends Activity {
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_a);
    addFragment();
  }

  public void addFragment() {
    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction fragmentTransaction = fragmentManager
    .beginTransaction();

    Chapter5_1FragmentA fragment = new Chapter5_1FragmentA();
    fragmentTransaction.add(R.id.layout_activity_a, fragment);
    fragmentTransaction.commit();
  }
}
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:id="@+id/layout_activity_a"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

</LinearLayout>
```

As you can see from this XML code, we removed the `Fragment` tags because we are adding `Fragment A` programmatically. As you can see in the `Chapter5_1Activity_A` class, we added a method called `addFragment()`. We used the `FragmentTransaction` class in order to add `Fragment A`. The `FragmentTransaction` class is used for operations such as adding fragments, removing fragments, attaching fragments to the UI, and so on. As you can see in the `addMethod()` method, you can get an instance of `FragmentTransaction` with `FragmentManager` using the `beginTransaction()` method. Finally we have to call the `commit()` method for the changes to be applied.

`FragmentManager` is used for managing fragments. As you can see in the code, you can get an instance of `FragmentManager` by the `getFragmentManager()` method. `FragmentManager` allows you to begin a transaction by the `beginTransaction()` method, get a fragment in activity by the `findFragmentById()` or `findFragmentbyTag()` methods, and pop a fragment off the back stack by the `popBackStack()` method.

Event sharing with activity

In our example, we started an activity in the `ListFragment` class' `onItemClick` method. We can establish the same operation by creating a callback interface in `ListFragment` and make the `Activity` class implement that callback. By this way the `Fragment` class will notify the owner `Activity` class. When the owner `Activity` class is notified, it can share the notification by other fragments. This way, fragments can share an event and communicate. We can go about this operation using the following steps:

1. We create the callback interface in the `Chapter5_1FragmentA` class:

```
public interface OnBookSelectedListener
  {
    public void onBookSelected(int bookIndex);
  }
```

2. We create an instance of `OnBookSelectedListener` and assign the owner activity to that instance in the `Chapter5_1FragmentA` class:

```
OnBookSelectedListener mListener;
@Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);
    mListener = (OnBookSelectedListener) activity;
  }
```

As you can see from this code, the owner activity class of `Chapter5_1FragmentA` should implement the `onBookSelectedListener` instance or there will be a class cast exception.

3. We make the `Chapter5_1Activity_A` class implement the `onBookSelectedListener` interface:

```
public class Chapter5_1Activity_A extends Activity implements
OnBookSelectedListener {
//some code here
        @Override
        public void onBookSelected(int bookIndex) {

          String author = Book.AUTHOR_NAMES[bookIndex];
          Intent intent = new Intent(this,
          Chapter5_1Activity_B.class);
          intent.putExtra("author", author);
          startActivity(intent);
       }
//some more code here
}
```

As you can see from this code, `Chapter5_1Activity_A` receives a selected book index in the event callback and starts the activity with author data.

4. We call the `onBookSelected` method in the `onItemClick` method of the `Chapter5_1FragmentA` class:

```
@Override
public void onItemClick(AdapterView<?> parent, View view, int
position, long id) {

  mListener.onBookSelected(position);
}
```

In this way, we made the activity and fragment share an event callback.

Using multiple fragments in an activity

Our sample book listing application is designed for small screens. When you execute this application on a larger screen, it will look bad. We have to use the space efficiently in larger screen sizes. In order to achieve this, we have to create a new layout for large screens. The new layout is as follows:

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

```
        android:id="@+id/layout_small_a"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="horizontal" >

        <fragment
            android:id="@+id/fragment_a"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            class="com.chapter5.Chapter5_1FragmentA"
            android:layout_weight="1"/>
        <fragment
            android:id="@+id/fragment_b"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            class="com.chapter5.Chapter5_1FragmentB"
            android:layout_weight="1"/>

    </LinearLayout>
```

As you can see from this code, we put two fragments in a horizontal `LinearLayout` layout. In the previous sample application, there was one fragment in each activity, but in this activity there are two fragments in order to use the space efficiently. By setting the `layout_weight` property to `1`, we make the fragments consume equal spaces on the screen.

We have to put this new layout XML file under a folder named `layout-xlarge-land` under the `res` folder. In this way, the Android uses this layout file when the device screen is large and in landscape mode. Android decides which layout file to use on runtime according to layout folder names. `layout` is the default folder name for Android. If Android can't find a suitable layout folder for a device screen size and mode, it uses the layout in the `layout` folder. Some of the common qualifiers for layout are as follows:

- `small` for small screen sizes
- `normal` for normal screen sizes
- `large` for large screen sizes
- `xlarge` for extra large screen sizes
- `land` for landscape orientation
- `port` for portrait orientation

However, this layout is not enough to make our sample function correctly on large screens. To make the new layout function correctly, we have to change how the fragments are managed. Update the `onBookSelected` property in `Chapter5_1Activity_A` as follows:

```
@Override
public void onBookSelected(int bookIndex) {

  FragmentManager fragmentManager = getFragmentManager();
  Fragment fragment_b =
  fragmentManager.findFragmentById(R.id.fragment_b);
  String author = Book.AUTHOR_NAMES[bookIndex];
  if(fragment_b == null)
  {
    Intent intent = new Intent(this,
        Chapter5_1Activity_B.class);
    intent.putExtra("author", author);
    startActivity(intent);
  }
  else
  {
    TextView textViewAuthor =
    (TextView)fragment_b.getView().findViewById
    (R.id.textViewAuthor);
    textViewAuthor.setText(author);
  }
}
```

As you can see from this code, we get the `Fragment B` class by using `FragmentManager`. If the `fragment_b` is not null, we understand that this activity contains `Fragment B`, and the device has a large screen, because `Fragment B` is used in `Activity A` only when the screen is large and in landscape mode. Then using `fragment_b`, we get the `textViewAuthor` TextView component and update its text with the chosen book's author name. On the right of the screen we see the author name of the chosen book.

If `fragment_b` is null, we understand that the device has a small screen, and we start a new activity using `Intent`.

In the `AndroidManifest.xml` file, we have to set the minimum SDK version to API Level 14, because fragments have been available for small screens since API Level 14. The `AndroidManifest.xml` file should look like the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter5"
```

```
        android:versionCode="1"
        android:versionName="1.0" >

        <uses-sdk android:minSdkVersion="14" />

        <application
            android:icon="@drawable/ic_launcher"
            android:label="@string/app_name" >
            <activity
                android:name=".Chapter5_1Activity_A"
                android:label="@string/app_name" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

                    <category
                    android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
            </activity>
            <activity android:name=".Chapter5_1Activity_B"/>
        </application>

    </manifest>
```

Our sample application will look like the following on a large screen:



Types of fragments

There are four types of fragments:

- `ListFragment`
- `DialogFragment`
- `PreferenceFragment`
- `WebViewFragment`

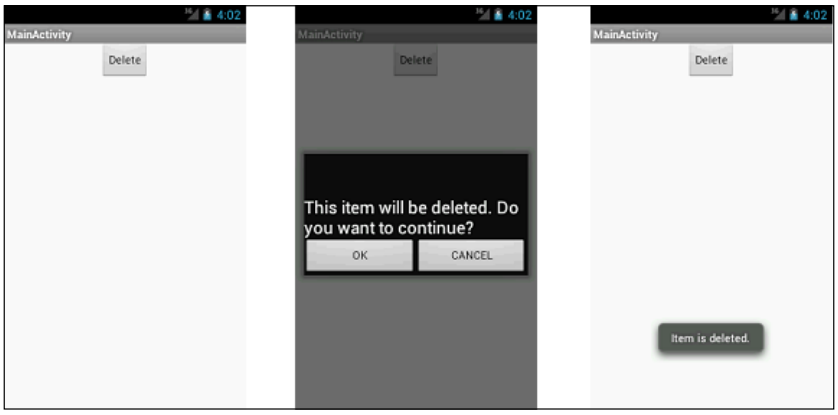
In this section, we will develop a sample application that uses these fragments. At the end of this section, the application will be completed.

ListFragment

This fragment is similar to `ListActivity` and contains a `ListView` view by default. It is used for displaying a list of items. In our previous sample code, we used `ListFragment`; see the *Creating and managing fragments* section for `ListFragment`.

DialogFragment

This fragment displays a dialog on top of its owner activity. In the following sample application, we are going to create a fragment that has a **Delete** button. When the button is clicked, a **DialogFragment** dialog box will be displayed. The **DialogFragment** dialog box will have a confirmation message and two buttons – **OK** and **Cancel** buttons. If the **OK** button is clicked, a message will be displayed and **DialogFragment** will be dismissed. The screens of the sample application will look like the following screenshot:



The layout XML code of the fragment with a **Delete** button is shown in the following code block:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical" >

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

</LinearLayout>
```

This layout is a simple layout with a `LinearLayout` layout and a `Button` component in it. The `Fragment` class of this layout is as follows:

```
package com.chapter5;

import android.app.Fragment;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.Button;

public class Chapter5_2Fragment extends Fragment implements
OnClickListener{

  Button fragmentButton;
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup
  container,
      Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.fragment, container,
    false);
```

```
        fragmentButton = (Button)view.findViewById
        (R.id.buttonFragment);
        fragmentButton.setOnClickListener(this);
        return view;
    }

    @Override
    public void onClick(View v) {
                //we need a FragmentTransaction in order to display a
                dialog
        FragmentTransaction transaction =
        getFragmentManager().beginTransaction();

        Chapter5_2DialogFragment dialogFragment = new
        Chapter5_2DialogFragment();

        dialogFragment.show(transaction, "dialog_fragment");

    }
}
```

As you can see from this code, in the `onClick` method of `Chapter5_2Fragment` class, an instance of the `Chapter5_2DialogFragment` class is created and using this instance, a dialog is displayed with its `show()` method.

The layout code of the **DialogFragment** dialog box is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:columnCount="2"
    android:orientation="horizontal"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <TextView
        android:id="@+id/textViewMessage"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_columnSpan="2"
        android:layout_gravity="fill"
        android:text="This item will be deleted. Do you want to
        continue?"
        android:textAppearance="?android:attr/textAppearanceLarge"
        />
```

```
        <!—we used a linear layout here because we need it in order to
        evenly distribute the buttons -->
       <LinearLayout
           android:layout_width="wrap_content"
           android:layout_gravity="fill_horizontal"
           android:layout_columnSpan="2" >

           <Button
               android:id="@+id/buttonOk"
               android:layout_width="fill_parent"
               android:layout_height="wrap_content"
               android:layout_weight="1"
               android:text="OK" />

       <Button
           android:id="@+id/buttonCancel"
           android:layout_width="fill_parent"
           android:layout_height="wrap_content"
           android:layout_weight="1"
           android:text="CANCEL" />

       </LinearLayout>

   </GridLayout>
```

As you can see from this previous code, we used `GridLayout` as the root layout. Then we input a `TextView` component which displays the confirmation message. Finally, two buttons are added to the layout—**OK** and **Cancel** buttons. The following is the `DialogFragment` class of this layout:

```
package com.chapter5;

import android.app.DialogFragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.Toast;

public class Chapter5_2DialogFragment extends DialogFragment
implements
OnClickListener{
```

```
    Button okButton;
    Button cancelButton;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container,
        Bundle savedInstanceState) {

      View view = inflater.inflate(R.layout.dialog_fragment,
      container, false);
      //initialize the buttons and set click events
      okButton = (Button)view.findViewById(R.id.buttonOk);
      okButton.setOnClickListener(this);

      cancelButton = (Button)view.findViewById(R.id.buttonCancel);
      cancelButton.setOnClickListener(this);

      return view;
    }

    @Override
    public void onClick(View v) {

      if(v == cancelButton)
        dismiss();
      else if( v == okButton)
      {
        Toast.makeText(this.getActivity(), "Item is deleted.",
        Toast.LENGTH_LONG).show();
        dismiss();
      }

    }
  }
```

As you the see from this code, this class extends can `DialogFragment` class. In the `onCreateView` method of the `Chapter5_2DialogFragment` class, we initialize the buttons and set their `onClick` events. In the `onClick` method of the `Chapter5_2DialogFragment` class, we handle the button click events. If the clicked button is **Cancel**, we dismiss the dialog window. If the clicked button is **OK**, we show an information message and dismiss the dialog. As you can see from the preceding code, the `dismiss()` method is used for closing the dialog.

PreferenceFragment

This fragment is similar to `PreferenceActivity`. It shows the preferences and saves them to `SharedPreferences`. In this section, we will extend the previous example code. We will put a preference about showing the confirmation message during deletion. The user could be able to choose to see or not to see confirmation message. We perform the following steps for using `PreferenceFragment`:

1. Create a source XML for the preference screen and put it under the `res/xml` folder:

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

    <CheckBoxPreference android:summary="check this in order
    to show confirmation message when deleting"
        android:title="show confirmation message"
        android:key="checkbox_preference"/>

</PreferenceScreen>
```

   As you can see from the previous code, our preference screen contains a check box preference for the confirmation message.

2. Create a class that extends `PreferenceFragment`:

```
package com.chapter5;

import android.os.Bundle;
import android.preference.PreferenceFragment;

public class Chapter5_2PereferenceFragment extends
PreferenceFragment {

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.pref);
  }

}
```

As you can see from this code, creating a preference screen is very easy; you just call the `addPreferencesFromResource` method with the XML file you created for the preferences. Now, we will in put a settings option menu item and we will open the preference screen by clicking on this menu item. In order to achieve this, we will modify the `Chapter5_2Fragment` class using the following steps:

1. We will add `setHasOptionsMenu(true)` to the `onCreateView` method of the `Chapter5_2Fragment` class:

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup
  container, Bundle savedInstanceState)
{

    View view = inflater.inflate(R.layout.fragment, container,
    false);
    fragmentButton =
    (Button)view.findViewById(R.id.buttonFragment);
    fragmentButton.setOnClickListener(this);

    setHasOptionsMenu(true);
    return view;
  }
```

2. We will add the following methods to the `Chapter5_2Fragment` class:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater
inflater) {
    inflater.inflate(R.menu.fragment_menu, menu);

  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {

    Intent intent = new
    Intent(getActivity(),Chapter5_2PreferenceActivity.class);
    startActivity(intent);
    return true;
  }
```

As you can see from this code, `onCreateOptionsMenu` is contributing to the options menu. This is the how a fragment contributes to the owner activity's menu. When the options menu item is clicked, a new activity is started with the `onOptionsItemSelected` method.

The `fragment_menu` menu XML is as follows:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/itemSettings"
    android:title="Settings"></item>
</menu>
```

`Chapter5_2PreferenceActivity` is the class that hosts `Chapter5_2PereferenceFragment` :

```
package com.chapter5;

import android.app.Activity;
import android.os.Bundle;

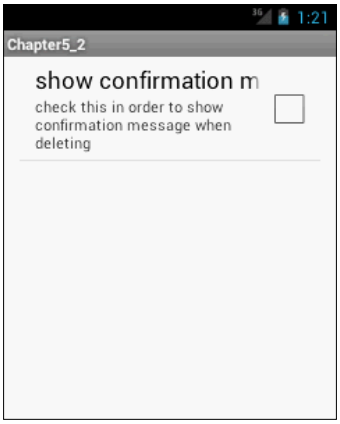
public class Chapter5_2PreferenceActivity extends Activity {

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

    getFragmentManager().beginTransaction()
        .replace(android.R.id.content, new
        Chapter5_2PereferenceFragment())
        .commit();
  }
}
```

As you can see from this code, we programmatically add `Chapter5_2PereferenceFragment` to the `Chapter5_2PreferenceActivity` class.

The preference screen should look like the following screenshot:



By adding this preference option, the user has the choice of whether or not to receive the confirmation message. (To read the setting, use the standard `SharedPreference` APIs.)

WebViewFragment

`WebViewFragment` is a premade `WebView` wrapped in a fragment. `WebView` inside this fragment is automatically paused or resumed when the fragment is paused or resumed. In this section, we will extend the previous sample codes to show the usage of `WebViewFragment`.

1. We add an **open web** button to the `Chapter5_2Fragment` class' layout XML code. The resulting layout is as follows:

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

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

        <Button
        android:id="@+id/buttonOpenWeb"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Open Web" />

</LinearLayout>
```

2. We create a class that extends `WebViewFragment` and an activity that hosts this fragment using the following code block:

```
package com.chapter5;

import android.os.Bundle;
import android.webkit.WebViewFragment;
```

```
public class Chapter5_2WebViewFragment extends WebViewFragment {

  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);

    getWebView().loadUrl("http://www.google.com");
  }
}
```

As you can see from this code, we get the `WebView` instance in the `onActivityCreated` method and load a URL that opens Google's website.

The activity that hosts this fragment is as follows:

```
package com.chapter5;

import android.app.Activity;
import android.os.Bundle;

public class Chapter5_2WebActivity extends Activity {


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

    getFragmentManager().beginTransaction()
        .replace(android.R.id.content, new
         Chapter5_2WebViewFragment())
        .commit();
  }
}
```

As you can see from this code, we programmatically add `Chapter5_2WebViewFragment` to `Chapter5_2WebViewActivity`. This sample application opens the `www.google.com` website when the **open web** button is clicked.

The final version of the `Chapter5_2Fragment` class is as follows:

```
package com.chapter5;

import android.app.Fragment;
import android.app.FragmentTransaction;
import android.content.Intent;
import android.os.Bundle;
```

```
import android.view.LayoutInflater;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.ViewGroup;
import android.widget.Button;

public class Chapter5_2Fragment extends Fragment implements
OnClickListener{

  Button fragmentButton;
  Button openWebButton;

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup
  container, Bundle savedInstanceState) {

    View view = inflater.inflate(R.layout.fragment, container,
    false);
    fragmentButton =
    (Button)view.findViewById(R.id.buttonFragment);
    fragmentButton.setOnClickListener(this);

    openWebButton = (Button)view.findViewById(R.id.buttonOpenWeb);
    openWebButton.setOnClickListener(this);

    setHasOptionsMenu(true);
    return view;
  }

  @Override
  public void onClick(View v) {
    if(v == fragmentButton)
    {
      FragmentTransaction transaction =
      getFragmentManager().beginTransaction();
      Chapter5_2DialogFragment dialogFragment = new
      Chapter5_2DialogFragment();
      dialogFragment.show(transaction, "dialog_fragment");
    }
    else if( v == openWebButton)
    {
```

```
        Intent intent = new
        Intent(getActivity(),Chapter5_2WebActivity.class);
        startActivity(intent);
      }
    }

    @Override
    public void onCreateOptionsMenu(Menu menu, MenuInflater
    inflater) {
      inflater.inflate(R.menu.fragment_menu, menu);

    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

  Intent intent = new
  Intent(getActivity(),Chapter5_2PreferenceActivity.class);
      startActivity(intent);
      return true;
    }
  }
```

The main `Activity` class for this application is as follows:

```
package com.chapter5;

import android.app.Activity;
import android.os.Bundle;

public class Chapter5_2Activity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

}
```

This `Activity` class is the owner activity of `Chapter5_2Fragment`. The layout of the preceding `Activity` is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:id="@+id/layout_small_a"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <fragment
        android:id="@+id/fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.chapter5.Chapter5_2Fragment" />

</LinearLayout>
```

The `AndroidManifest.xml` file of this sample application should look like the following:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter5"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="15" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".Chapter5_2Activity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"
                />

                <category
                android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
        android:name=".Chapter5_2PreferenceActivity"></activity>
        <activity android:name=".Chapter5_2WebActivity"></activity>
    </application>

</manifest>
```

As you can see in this code, we need Internet permission to open a website. Furthermore, we need to set the minimum SDK to API Level 14 in order to use fragments with small screens.

Summary

Fragments are available for small screen devices with Android Ice Cream Sandwich being introduced. In this chapter, we first learned the basics of fragments, and the construction and destruction life cycle of fragments. Then, we learned about creating and managing fragments with a sample application. Finally, we learned the specialized types of fragments – `ListFragment`, `DialogFragment`, `PreferenceFragment`, and `WebViewFragment`. In the next chapter, we will see some practices to develop applications that support different screen sizes.

6
Supporting Different Screen Sizes

Android 3.0 is only for large screen devices. However, Android Ice Cream Sandwich is for all small and large screen devices. Developers should create applications that support both large and small screen sizes. This chapter will show the ways of designing user interfaces that support different screen sizes.

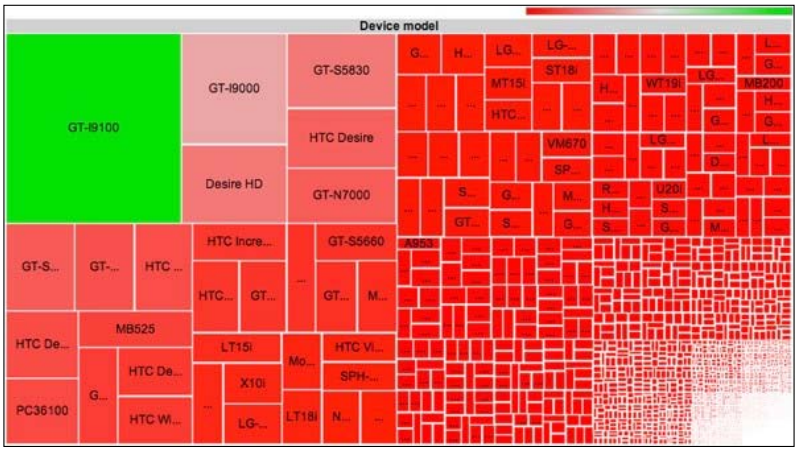
The topics covered in this chapter are as follows:

- Using `match_parent` and `wrap_content`
- Using nine-patch
- Using `dip` instead of `px`

Android 4.0 supports different screen sizes

There is a vast variety of Android devices and hence there are many different screen sizes.

The following graph (source `opensignalmaps.com`) shows the Android device fragmentation:



As you see in the graph, there is a vast variety of devices (nearly 4000 distinct devices). This means many different screen sizes and densities. Android scales and resizes the user interface of your Android applications. However, this is not enough all the time. For instance, a user interface designed for a small screen will be magnified for a large screen by the Android. This doesn't look good on large screens. The space on large screens should be used efficiently and the user interfaces of large screens should be different than the user interfaces for small screens. Android provides some APIs for designing the user interfaces that fit the different screen sizes and densities. You should use these APIs to make your application look good on different screen sizes and densities. In this way, user experience of Android applications could be increased.

The things to be considered in designing user interfaces for Android applications are as follows:

- **Screen Sizes**: This is the physical screen size of the devices. Screen sizes may range from 2.5" to 10.1" for smart phones and tablets.
- **Resolution**: This is the number of pixels that a device has in each dimension. It is usually defined as width x height such as 640 x 480.

- **Screen Density**: This is the maximum number of pixels in a physical area. High density screens have more pixels than low density screens in an area.
- **Screen Orientation**: A device could be in landscape or portrait mode. When in landscape mode its width increases.

Using match_parent and wrap_content

`match_parent` and `wrap_content` could be used to set the `layout_height` and `layout_width` properties of a view. When `match_parent` is used, Android expands the view to make its size the same as its parent. When `wrap_content` is used, Android expands the view according to its content's size. It is also possible to set the width and height using pixel values. However, it is not a good practice to use pixel values, because the number of pixels change according to the screen properties and a given pixel value is not of the same size in every screen. In the following example, we use the pixel value in order to set the width and height of the view. The layout XML code is shown in the following code block:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:layout_width="240px"
        android:layout_height="30px"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:padding="@dimen/padding_medium"
        android:text="hello world1 hello world 2 hello world 3
        hello"
        tools:context=".Chapter6_1Activity" />

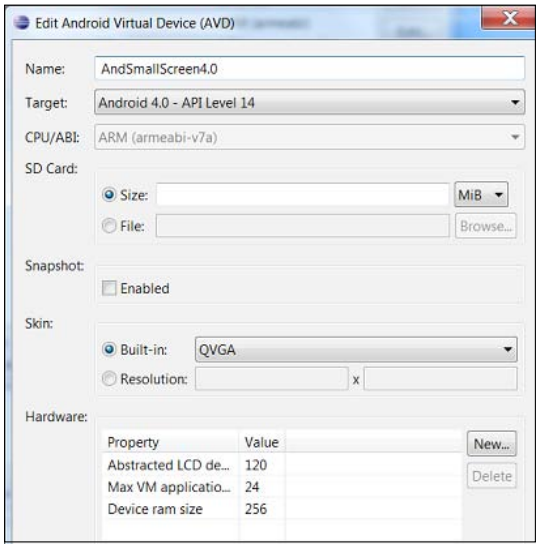
</RelativeLayout>
```



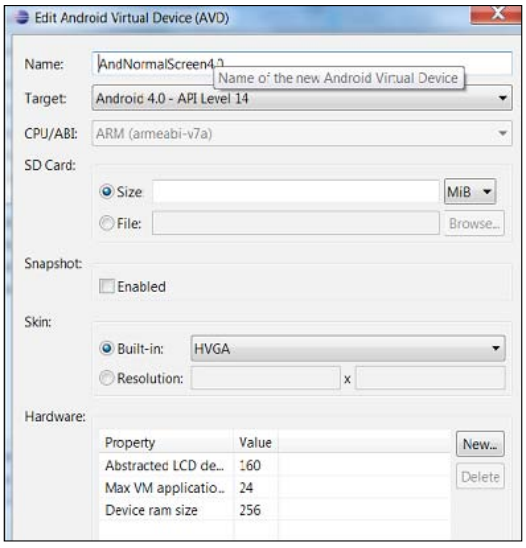
Some of the definitions such as `@dimen` used in the preceding layout XML files will be available in the source code files of this book.

As you can see in the code, we set `layout_width` to `240px` and `layout_height` to `30px`. We will execute this application in three different emulators with different screen properties. The emulator properties are as follows:

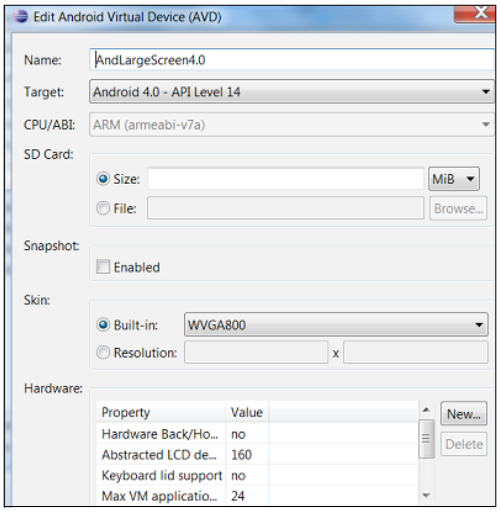
- **Small Screen Properties***:* This configuration is for small screens. These properties can be configured as shown in the following screenshot:



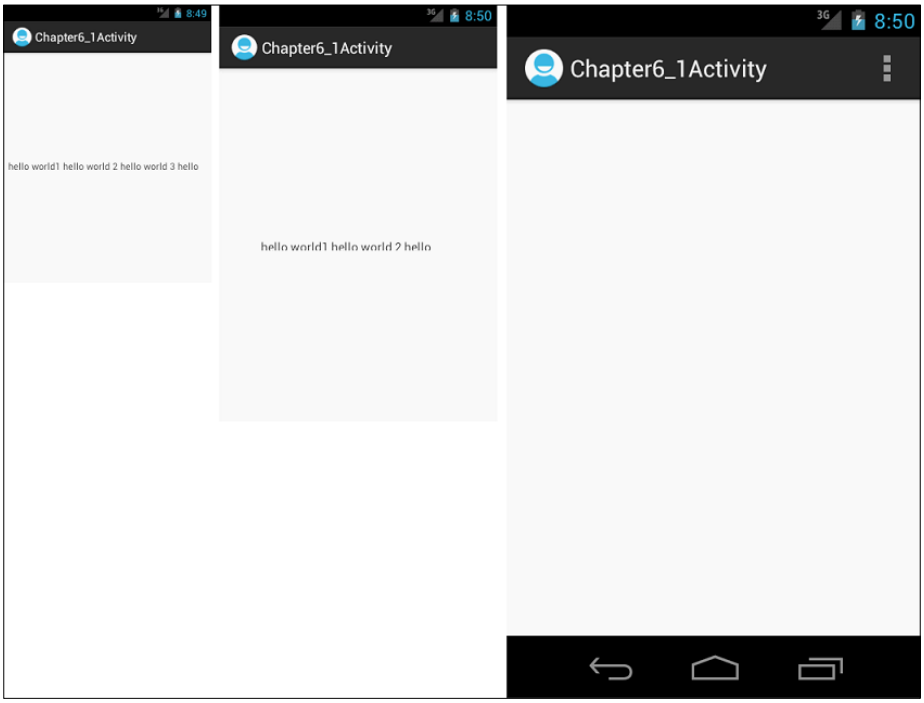
- **Normal Screen Properties:** This configuration is for normal screens. These properties can be configured as shown in the following screenshot:



- **Large Screen Properties**: This configuration is for large screens. These properties can be configured as shown in the following screenshot:



When this application is executed in the previous emulator configurations, the screens will look like the following:



As you can see in the screenshot, it looks fine on the small screen. However, on the normal screen, the text is cropped and not all the content of the `TextView` component is visible. On the large screen, nothing is visible. This sample shows that using pixel as a width and height value is not a good practice.

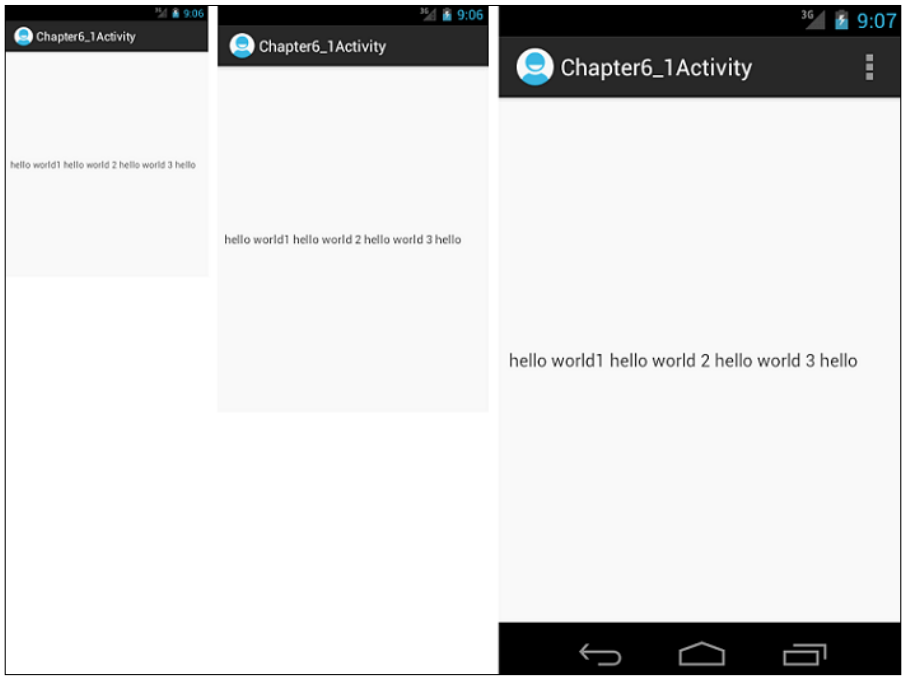
Now, we will use `wrap_content` and `match_parent` to set the height and width lengths. The layout XML code will look like the following:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:padding="@dimen/padding_medium"
        android:text="hello world1 hello world 2 hello world 3
        hello"
        tools:context=".Chapter6_1Activity" />

</RelativeLayout>
```

When the application is executed with same emulator configurations, the screen will look like the following:



As you can see in this screenshot, the application looks the same in each emulator and screen configurations, and all the content of the `TextView` component is displayed. Thus, using `wrap_content` and `match_parent` is a best practice in designing user interfaces.

Using dip instead of px

Another option for the previous sample is to use the `dip` (density independent pixels) value instead of the `px` value. In this way, the `TextView` component will look nearly the same in different screen sizes. The code will look like the following:

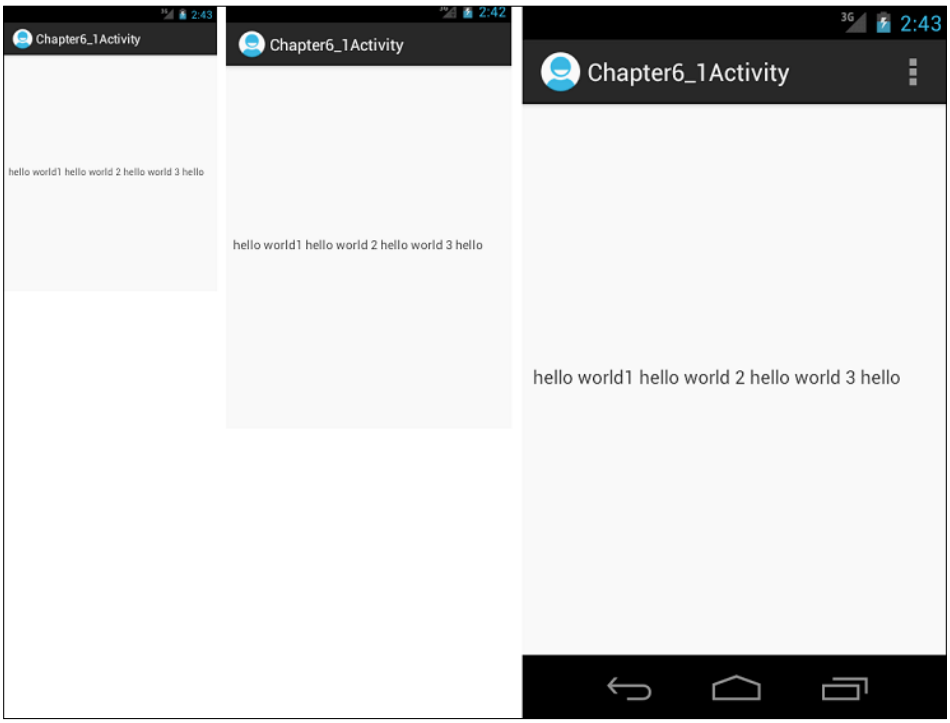
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:layout_width="350dip"
        android:layout_height="40dip"
        android:layout_centerHorizontal="true"
```

```
        android:layout_centerVertical="true"
        android:padding="@dimen/padding_medium"
        android:text="hello world1 hello world 2 hello world 3
        hello"
        tools:context=".Chapter6_1Activity" />

</RelativeLayout>
```

As you can see in this code, we used the `dip` value for the width and height. If you execute this application in the emulators defined in the previous section, it would look like the following:





For the font sizes, the sp (scale independent pixel) unit could be used instead of px.

Omit using AbsoluteLayout

AbsoluteLayout is a deprecated layout which uses fixed positions for the views in it. `AbsoluteLayout` is not a good practice in designing user interfaces because it will not look same in different screen sizes. We will see this with the following example layout:

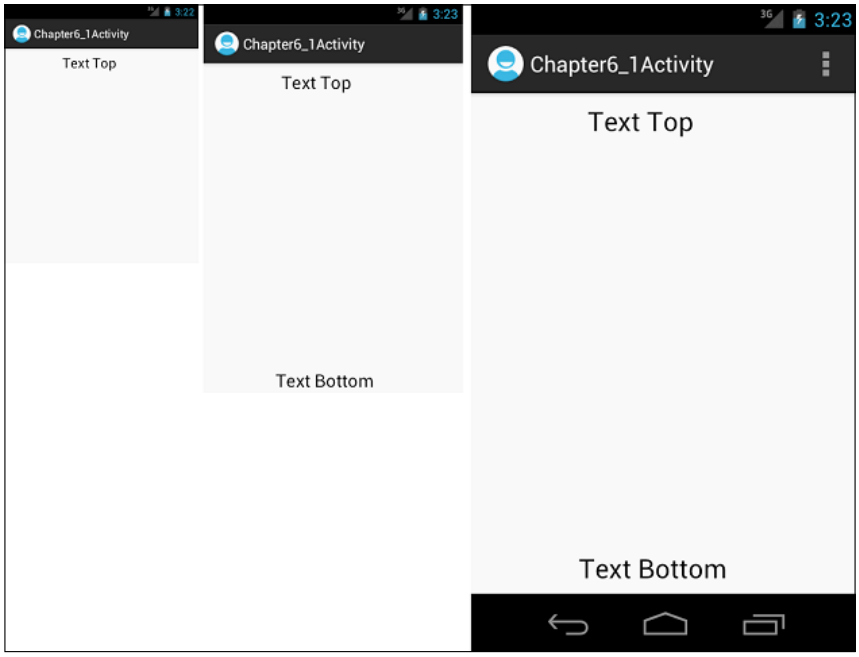
```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_x="96dp"
        android:layout_y="8dp"
        android:text="Text Top"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/textView4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_x="89dp"
        android:layout_y="376dp"
        android:text="Text Bottom"
        android:textAppearance="?android:attr/textAppearanceLarge" />

</AbsoluteLayout>
```

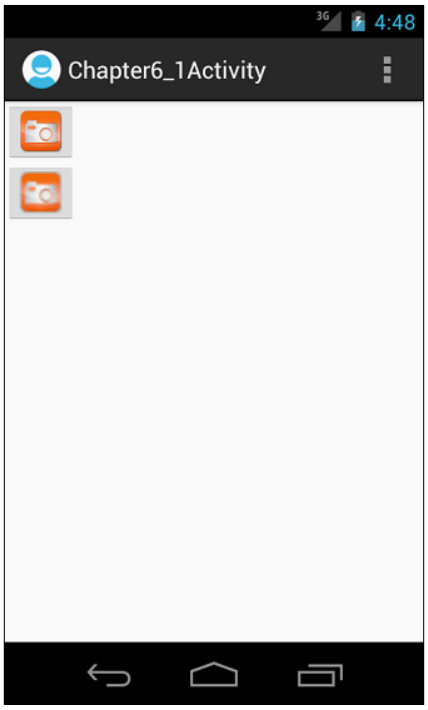
As you can see in this XML code, fixed positions are used for the views in `AbsoluteLayout`. When this application is executed in the emulators defined in previous section, it will look like the following:



As you can see in the screen shots, the text at the bottom is not visible on the small screen, but it is visible on the other screens. AbsoluteLayout is not a good practice in user interface design.

Providing different bitmap drawables for different screen densities

Android scales bitmaps according to screen density. However, images will not look good if only one bitmap is provided. The image would look blurry or corrupted. Providing different bitmap drawables for different screen densities is a good practice. In the following screenshot, two image buttons are used. Different bitmap drawables are provided for the first image button and a low density bitmap drawable is provided for the second image button. As you can see in the screenshot, the bitmap in the second image button looks blurry; however, the bitmap in the first image button looks fine.



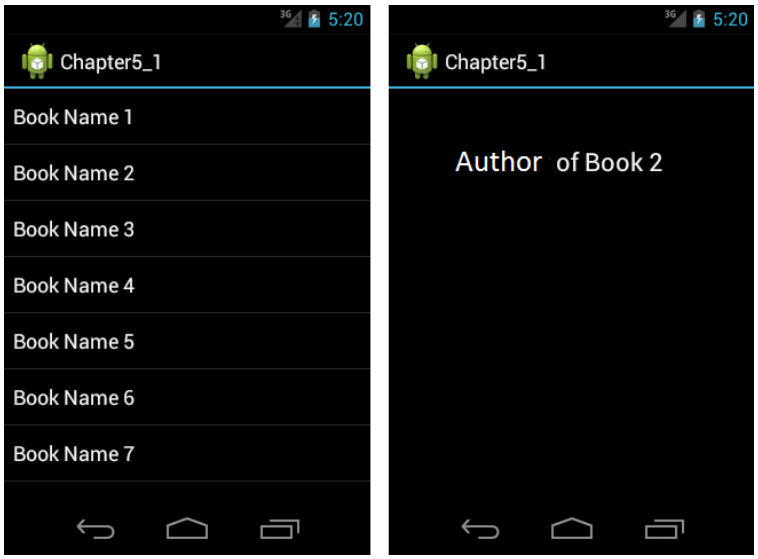


If the images are put in the `drawable-nodpi` folder they won't be scaled.

Providing different layouts for different screen sizes

Android scales the layouts to make them fit to device screens. However, this is not enough in some cases. In *Chapter 5*, *Fragments*, we developed an application that lists the books and when a book is clicked, the author of the book is displayed.

The following is the screenshot that is displayed on small screens:



For larger screens, this design is not a good alternative. The user interface will look bad. We should use the space in larger screens efficiently. We can use a different layout for larger screens that combines the two screens displayed on smaller screens. The user interface should look like the following:



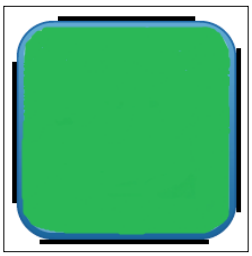
The layout files should be put in appropriate folders. For instance, the layout files for large screens should be put in the `res/layout-large` folder and for small screens should be put in the `res/layout-small` folder.

New screen size qualifiers are introduced with Android 3.2 in addition to existing ones. The new qualifiers are as follows:

- `sw<N>dp`: This qualifier defines the smallest width. For example, `res/layout-sw600dp/`.
    - When the screen width is at least `N` dp, regardless of the screen orientation, the layout file in this folder is used.
- `w<N>dp`: This qualifier defines the exact available width. For example, `res/layout-w600dp/`.
- `h<N>dp`: This qualifier defines the exact available height. For example, `res/layout-h600dp/`.

Nine-patch

The **nine-patch** feature allows the using of stretchable bitmaps as resources. Bitmaps are stretched according to defined stretchable areas. Stretchable areas are defined by a 1-pixel width black line. The following is a sample nine-patch drawable:



The image file should be put into the drawable folder with an extension of `.9.png`. The top and the left black lines define the stretchable area and the bottom and the right black lines define the stretchable area to fit content.

There is a tool called **Draw 9-patch** that is bundled with Android SDK. You can easily create nine-patch images with this editor.

Summary

In this chapter we learned some design guidelines in order to support different screen sizes and densities. We shouldn't use hardcoded pixel values to define layout width and height. Instead, we should use `wrap_content` and `match_parent` properties, or use `dip` values instead of `px` values. We should use different layouts for different screen sizes to make applications look good in all screen sizes. We also learned about using nine-patch rule to create stretchable bitmaps. After developing an application, we should test the application in various screen sizes and densities in an Android emulator to see how it looks. In this way we could detect user interface problems and bugs.

In the next chapter, we are going to learn about the Android Compatibility Package and we will see how to use it.

7
Android Compatibility Package

New Android APIs do not work in the previous versions of Android, so the Android Compatibility Package was thus introduced to allow the porting of the new APIs to the older versions of the Android platform. This chapter shows how we can use the Android Compatibility Package.

The topics covered in this chapter are as follows:

- What is and why do we use the Android Compatibility Package
- How to use the Android Compatibility Package

What is Android Compability Package

Android has some great new APIs released with Android 3.0 and its later versions. However, many users don't upgrade their devices to the latest Android platform. Google released the Android Compatibility Package that contains support for some of the new APIs released with Android 3.0 and its later versions. In this way, developers could develop applications that use new APIs and work in older Android versions. Some of the classes that are included in the Android Compatibility Package are as follows:

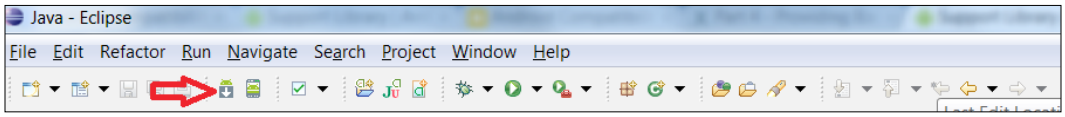
- `Fragment`
- `FragmentManager`
- `FragmentTransaction`
- `ListFragment`

- `DialogFragment`
- `LoaderManager`
- `Loader`
- `AsyncTaskLoader`
- `CursorLoader`

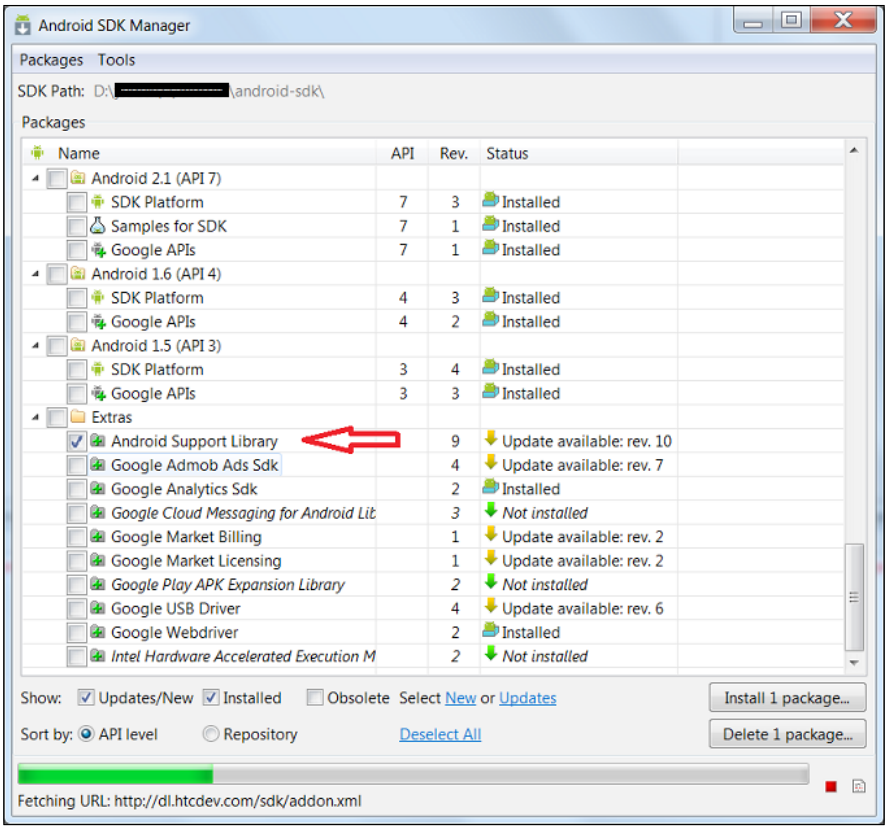
Some useful APIs such as animation classes, action bar, and FragmentMapActivity are not included in the Android Compatibility Package.

How to use the Android Compatibility Package

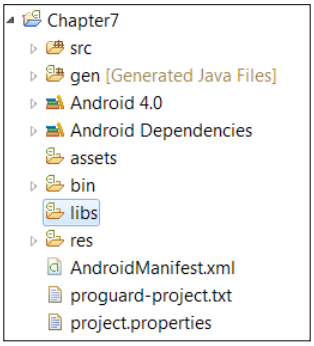
1. We need to download and install the Android Compatibility Package. In order to download the Android Compatibility Package, click on the **Android SDK Manager** button in the Eclipse menu as shown in the following screenshot:



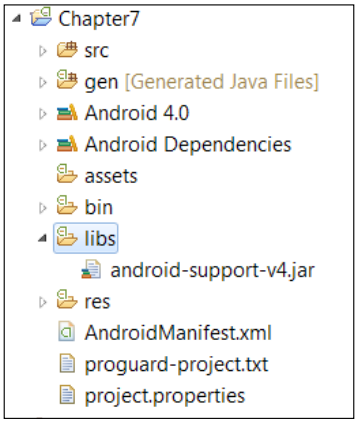
   Alternately, we can reach the Android SDK Manager by the Eclipse menu using **Window | Android SDK Manager**. After the **Android SDK Manager** window is opened, check the **Android Support Library** option as shown in the following screenshot:



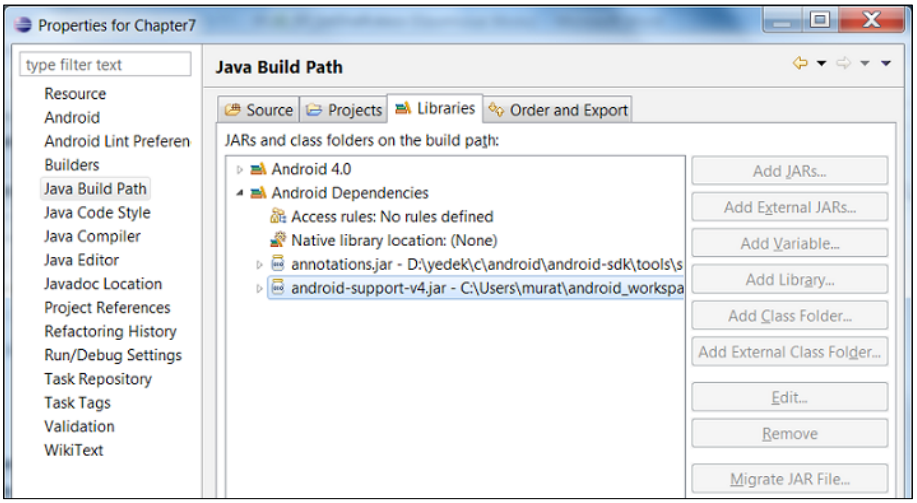
2. Then, click on the **Install** button and install the package. Now we are ready to develop an Android project that can use the Android Compatibility Package. Firstly, create an Android project in Eclipse. Then, we need to add the support library to our Android project. If it doesn't exist, create a folder named `libs` under the Android project's root folder as shown in the following screenshot:



3. Now, find and copy the `<your android sdk folder>/extras/android/support/v4/android-support-v4.jar` file to the `libs` folder. The folder structure should be as shown in the following screenshot:



4. Lastly, if the `.jar` file is not in the project build path, add the `.jar` file to the project build path as shown in the following screenshot:



Now you know how to manually add the support library. Eclipse makes this process easy with the **Add Support Library** menu option. Use the following steps:

1. Right-click on the project in the explorer.
2. Go to the **Android Tools** | **Add Support Library…** option.
3. Follow the steps to complete the wizard.

Now we can make use of the compatibility package. We are going to create an application that uses the `Fragment` class, but the `Fragment` class present in the compatibility package, to display a text using the following steps:

1. Firstly, create a layout XML for the fragment and name the XML file `fragment.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello Android Compatibility Package"
        android:textAppearance="?android:attr/textAppearanceLarge"
/>

</LinearLayout>
```

2. Then, create a layout for the activity using the following code block:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

</RelativeLayout>
```

3. Now we are going to create the `Fragment` class for the `fragment.xml` layout:

```
package com.chapter7;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
```

```
public class Chapter7Fragment extends Fragment {


  @Override
  public View onCreateView(LayoutInflater inflater,
  ViewGroup
container,
      Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment,
    container,
false);
    return view;
  }
}
```

As you can see from the preceding code, the `Fragment` class is from the `android.support.v4.app.Fragment` package. This means that we are using the Android Compatibility Package. If we don't want to use the compatibility package, then we should use the `Fragment` class from `the android.app.Fragment` package.

The `Activity` class for our application is as follows:

```
package com.chapter7;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentTransaction;
import android.view.Menu;

public class Chapter7Activity extends FragmentActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        addFragment();
    }

    public void addFragment() {
    FragmentManager fragmentManager =
    this.getSupportFragmentManager();
    FragmentTransaction fragmentTransaction =
    fragmentManager.beginTransaction();
```

```
        Chapter7Fragment fragment = new Chapter7Fragment();
        fragmentTransaction.add(R.id.main_layout,fragment);
        fragmentTransaction.commit();
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

As you can see from the preceding code block, the support library APIs follow the same naming as the standard APIs. We just need to use the correct imports and call the correct managers. We have to add `android.support.v4.app` to our import list in order to use classes that are in the compatibility package.

In order to get the `FragmentManager` instance, we call the `getSupportFragmentManager()` method of our `Activity` class. As you will have noticed, the `Activity` class extends the `FragmentActivity` class. We need to do this because it is the only way in which we can use Fragments.

The `AndroidManifest.xml` file should look like the following:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter7"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".Chapter7Activity"
            android:label="@string/title_activity_chapter7" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                LAUNCHER" />
```

```
            </intent-filter>
        </activity>
    </application>

</manifest>
```

As you can see in this code, the minimum SDK level is set to API Level 8. We can set the minimum API Level to 4 or more. In this way, we can use the new APIs in older versions of Android.

Summary

In this chapter, we learned what the Android Compatibility Package is and how we can use it. We also learned how with the help of this library, we can use the new APIs in the older versions of Android.

In the next chapter, we are going to learn using new connectivity APIs – Android Beam and Wi-Fi Direct.

8
New Connectivity APIs – Android Beam and Wi-Fi Direct

New connectivity APIs have been introduced with Android Ice Cream Sandwich – **Android Beam** which uses the NFC hardware of the device, and **Wi-Fi Direct** which allows devices to connect to each other without using a wireless access point. This chapter will teach us Android Beam and Wi-Fi Direct APIs' usage.

The topics covered in this chapter are as follows:

- Android Beam
- Beaming NdefMessages
- Sharing data with Wi-Fi Direct

Android Beam

Devices that have NFC hardware can share data by tapping them together. This could be done with the help of the Android Beam feature. It is similar to Bluetooth, as we get seamless discovery and pairing as in a Bluetooth connection. Devices connect when they are close to each other (not more than a few centimeters). Users can share pictures, videos, contacts, and so on, using the Android Beam feature.

Beaming NdefMessages

In this section, we are going to implement a simple Android Beam application. This application will send an image to another device when two devices are tapped together. There are three methods that are introduced with Android Ice Cream Sandwich that are used in sending **NdefMessages**. These methods are as follows:

- **setNdefPushMessage()**: This method takes an NdefMessage as a parameter and sends it to another device automatically when devices are tapped together. This is commonly used when the message is static and doesn't change.
- **setNdefPushMessageCallback()**: This method is used for creating dynamic NdefMessages. When two devices are tapped together, the `createNdefMessage()` method is called.
- **setOnNdefPushCompleteCallback()**: This method sets a callback which is called when the Android Beam is successful.

We are going to use the second method in our sample application.

Our sample application's user interface will contain a `TextView` component for displaying text messages and an `ImageView` component for displaying the received images sent from another device. The layout XML code will be as follows:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

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

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/textView"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="14dp"
         />

</RelativeLayout>
```

Now, we are going to implement, step-by-step, the `Activity` class of the sample application. The code of the `Activity` class with the `onCreate()` method is as follows:

```
public class Chapter9Activity extends Activity implements
    CreateNdefMessageCallback
  {

  NfcAdapter mNfcAdapter;
  TextView mInfoText;
  ImageView imageView;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    imageView = (ImageView) findViewById(R.id.imageView);
    mInfoText = (TextView) findViewById(R.id.textView);
    // Check for available NFC Adapter
       mNfcAdapter =
       NfcAdapter.getDefaultAdapter(getApplicationContext());

    if (mNfcAdapter == null)
    {
      mInfoText = (TextView) findViewById(R.id.textView);
      mInfoText.setText("NFC is not available on this device.");
      finish();
      return;
    }
    // Register callback to set NDEF message
    mNfcAdapter.setNdefPushMessageCallback(this, this);
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
  }
}
```

As you can see in this code, we can check whether the device provides an `NfcAdapter`. If it does, we get an instance of `NfcAdapter`. Then, we call the `setNdefPushMessageCallback()` method to set the callback using the `NfcAdapter` instance. We send the `Activity` class as a callback parameter because the `Activity` class implements `CreateNdefMessageCallback`.

In order to implement `CreateNdefMessageCallback`, we should override the `createNdefMessage()` method as shown in the following code block:

```
@Override
public NdefMessage createNdefMessage(NfcEvent arg0) {

  Bitmap icon =
  BitmapFactory.decodeResource(this.getResources(),
      R.drawable.ic_launcher);
  ByteArrayOutputStream stream = new ByteArrayOutputStream();
  icon.compress(Bitmap.CompressFormat.PNG, 100, stream);
  byte[] byteArray = stream.toByteArray();

  NdefMessage msg = new NdefMessage(new NdefRecord[] {
      createMimeRecord("application/com.chapter9", byteArray)
      , NdefRecord.createApplicationRecord("com.chapter9")
});
    return msg;
  }
  public NdefRecord createMimeRecord(String mimeType, byte[]
  payload) {

    byte[] mimeBytes = mimeType.getBytes(Charset.forName("US-
    ASCII"));
    NdefRecord mimeRecord = new
    NdefRecord(NdefRecord.TNF_MIME_MEDIA,
        mimeBytes, new byte[0], payload);
    return mimeRecord;
  }
```

As you can see in this code, we get a drawable, convert it to bitmap, and then to a byte array. Then we create an `NdefMessage` with two `NdefRecords`. The first record contains the mime type and the byte array. The first record is created by the `createMimeRecord()` method. The second record contains the **Android Application Record** (**AAR**). The Android Application Record was introduced with Android Ice Cream Sandwich. This record contains the package name of the application and increases the certainty that your application will start when an **NFC Tag** is scanned. That is, the system firstly tries to match the intent filter and AAR together to start the activity. If they don't match, the activity that matches the AAR is started.

When the activity is started by an Android Beam event, we need to handle the message that is sent by the Android Beam. We handle this message in the `onResume()` method of the `Activity` class as shown in the following code block:

```
@Override
public void onResume() {
```

```
    super.onResume();
    // Check to see that the Activity started due to an Android
    Beam
    if (NfcAdapter.ACTION_NDEF_DISCOVERED.
     equals(getIntent().getAction())) {
      processIntent(getIntent());
    }
  }

  @Override
  public void onNewIntent(Intent intent) {
    // onResume gets called after this to handle the intent
    setIntent(intent);
  }


  void processIntent(Intent intent) {

    Parcelable[] rawMsgs = intent

  .getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
    // only one message sent during the beam
    NdefMessage msg = (NdefMessage) rawMsgs[0];
    // record 0 contains the MIME type, record 1 is the AAR
    byte[] bytes = msg.getRecords()[0].getPayload();
    Bitmap bmp = BitmapFactory.decodeByteArray(bytes, 0,
    bytes.length);

    imageView.setImageBitmap(bmp);
  }
```

As you can see in this code, we firstly check whether the intent is `ACTION_NDEF_DISCOVERED`. This means the `Activity` class is started due to an Android Beam. If it is started due to an Android Beam, we process the intent with the `processIntent()` method. We firstly get `NdefMessage` from the intent. Then we get the first record and convert the byte array in the first record to bitmap using `BitmapFactory`. Remember that the second record is AAR, we do nothing with it. Finally, we set the bitmap of the `ImageView` component.

The `AndroidManifest.xml` file of the application should be as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter9"
    android:versionCode="1"
```

```
        android:versionName="1.0" >

        <uses-permission android:name="android.permission.NFC"/>
        <uses-feature android:name="android.hardware.nfc"
        android:required="false" />

        <uses-sdk
            android:minSdkVersion="14"
            android:targetSdkVersion="15" />

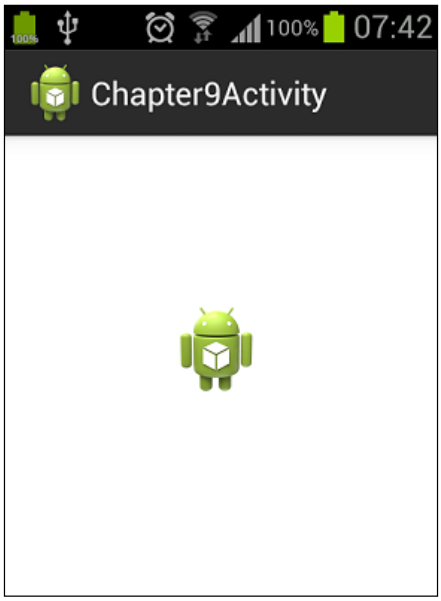
        <application
            android:icon="@drawable/ic_launcher"
            android:label="@string/app_name"
            android:theme="@style/AppTheme" >
            <activity
                android:name=".Chapter9Activity"
                android:label="@string/title_activity_chapter9" >
                <intent-filter>
                    <action android:name="android.intent.action.MAIN" />

                    <category
                    android:name="android.intent.category.LAUNCHER" />
                </intent-filter>
                 <intent-filter>
                    <action
                    android:name="android.nfc.action.NDEF_DISCOVERED" />
                    <category
                    android:name="android.intent.category.DEFAULT" />
                    <data android:mimeType="application/com.chapter9" />
                </intent-filter>
            </activity>
        </application>

    </manifest>
```

As you can see in this code, we need to set the minimum SDK to API Level 14 or more in the `AndroidManifest.xml` file because these APIs are available in API Level 14 or more. Furthermore, we need to set the permissions to use NFC. We also set the `uses` feature in `AndroidManifest.xml`. The feature is set as not required. This means that our application would be available for devices that don't have NFC support. Finally, we create an intent filter for `android.nfc.action.NDEF_DISCOVERED` with `mimeType` of `application/com.chapter9`.

When a device sends an image using our sample application, the screen will be as follows:



Wi-Fi Direct

In a conventional wireless network, devices are connected to each other through a wireless access point. With the help of **Wi-Fi Direct**, devices connect to each other without the need of a wireless access point. It's similar to Bluetooth, but it is faster and the range of Wi-Fi Direct is longer. New Wi-Fi Direct APIs are introduced with Android Ice Cream Sandwich which allows us to use Wi-Fi Direct properties of Android devices.

The main class that will help us to find and connect peers is the `WifiP2pManager` class. We are going to use the following `Listener` classes during finding and connecting to peers:

- `WifiP2pManager.ActionListener`
- `WifiP2pManager.ChannelListener`
- `WifiP2pManager.ConnectionInfoListener`
- `WifiP2pManager.PeerListListener`

Finally, the following intents will help us in a Wi-Fi Direct connection:

- `WIFI_P2P_CONNECTION_CHANGED_ACTION`
- `WIFI_P2P_PEERS_CHANGED_ACTION`
- `WIFI_P2P_STATE_CHANGED_ACTION`
- `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION`

In this section, we are going to learn how to use these new Wi-Fi Direct APIs with a sample application.

Sample Wi-Fi Direct application

In order to use Wi-Fi Direct APIs, we need to set the minimum SDK version to API Level 14 or more in `AndroidManifest.xml`. Furthermore, we need some permission to use Wi-Fi Direct APIs. The `AndroidManifest.xml` file should be as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter9"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="15" />
        <uses-permission
        android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission
    android:name="android.permission.CHANGE_WIFI_STATE" />
    <uses-permission
    android:name="android.permission.CHANGE_NETWORK_STATE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".Chapter9Activity"
            android:label="@string/title_activity_chapter9" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

```
                <category
                android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

The first class that we need is a class that extends `BroadcastReceiver` and handles the intents that we listed previously in the `onReceive()` method. The constructor of this class should be as follows:

```
package com.chapter9;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.NetworkInfo;
import android.net.wifi.p2p.WifiP2pManager;
import android.net.wifi.p2p.WifiP2pManager.Channel;
import android.net.wifi.p2p.WifiP2pManager.PeerListListener;
import android.widget.Toast;

public class Chapter9WiFiDirectBroadcastReceiver extends
BroadcastReceiver {

    private WifiP2pManager manager;
    private Channel channel;
    private Chapter9Activity activity;

    public Chapter9WiFiDirectBroadcastReceiver(WifiP2pManager manager,
Channel
channel,
        Chapter9Activity activity) {
        super();
        this.manager = manager;
        this.channel = channel;
        this.activity = activity;
    }
}
```

As you can see in this code, we passed the `Channel`, `WifiP2pManager`, and the `Activity` classes to the constructor as parameters because we will need them later in the `onReceive()` method. We need to override the `onReceive()` method of `BroadcastReceiver` as shown in the following code block:

```
@Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.
equals(action)) {


            int state =
            intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE,
            -1);


            if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
                // Wifi Direct mode is enabled
              Toast.makeText(activity, "wifi direct is
              enabled",Toast.LENGTH_LONG).show();
            } else {
              // Wifi Direct mode is disabled
              Toast.makeText(activity, "wifi direct is
              disabled",Toast.LENGTH_LONG).show();
            }

        } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.
        equals(action))
        {

            // request peers from the wifi p2p manager
            if (manager != null) {
                manager.requestPeers(channel, (PeerListListener)
                activity);
            }

        } else if
          (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.
          equals(action)) {

            if (manager == null) {
                return;
            }
```

```
            NetworkInfo networkInfo = (NetworkInfo) intent
                    .getParcelableExtra(WifiP2pManager.
                    EXTRA_NETWORK_INFO);

            if (networkInfo.isConnected()) {

                // request connection info
                manager.requestConnectionInfo(channel, activity);
            } else {
                // It's a disconnect

            }
        } else if
          (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.
          equals(action)) {

        }
    }
```

In this method, we handle the received intents. Firstly, we check whether the intent is `WIFI_P2P_STATE_CHANGED_ACTION`. This intent is received when Wi-Fi Direct is enabled or disabled. We receive the Wi-Fi Direct status from the intent and take action according to the Wi-Fi Direct status.

Secondly, we check whether the intent is `WIFI_P2P_PEERS_CHANGED_ACTION`. This intent is received when the `discoverPeers()` method of the `WifiP2pManager` class is called. We get the list of the peers from the `requestPeers()` method of the `Wifi2P2pManager` class when we receive the `WIFI_P2P_PEERS_CHANGED_ACTION` intent.

Next, we check whether the received intent is `WIFI_P2P_CONNECTION_CHANGED_ACTION`. This intent is received when the Wi-Fi connection changes. We handle connections or disconnections when we receive the `WIFI_P2P_CONNECTION_CHANGED_ACTION` intent. We firstly get `NetworkInfo` from the intent to understand whether there is a connection or disconnection. If it is a connection, we call the `requestConnectionInfo()` method of `WifiP2pManager` to connect.

Lastly, we check whether the intent is `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION`. We receive this intent when the device details have changed. We do nothing for this intent.

We have a simple user interface for this application; a layout with two buttons. The first button is to find and second button is to connect peers. The XML code of the layout is as follows:

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

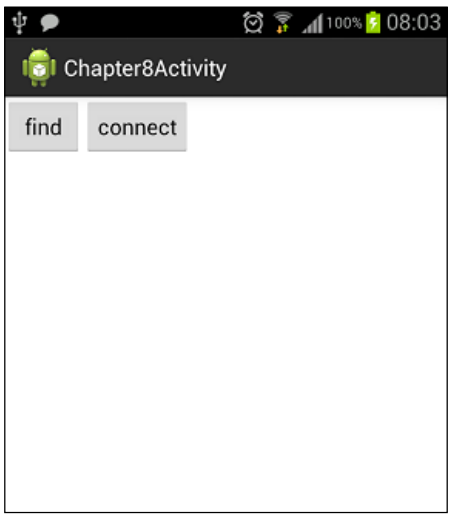
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

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

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

</LinearLayout>
```

The user interface will look like the following screenshot:



Lastly, we need to implement the `Activity` class of this application. The code of the `Activity` class should be as follows:

```
package com.chapter9;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.IntentFilter;
import android.net.wifi.p2p.WifiP2pConfig;
import android.net.wifi.p2p.WifiP2pDevice;
import android.net.wifi.p2p.WifiP2pDeviceList;
import android.net.wifi.p2p.WifiP2pInfo;
import android.net.wifi.p2p.WifiP2pManager;
import android.net.wifi.p2p.WifiP2pManager.ActionListener;
import android.net.wifi.p2p.WifiP2pManager.Channel;
import android.net.wifi.p2p.WifiP2pManager.ChannelListener;
import android.net.wifi.p2p.WifiP2pManager.ConnectionInfoListener;
import android.net.wifi.p2p.WifiP2pManager.PeerListListener;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;
public class Chapter9Activity extends Activity implements
ChannelListener,OnClickListener,PeerListListener,
ConnectionInfoListener {

    private WifiP2pManager manager;
    private final IntentFilter intentFilter = new IntentFilter();
    private Channel channel;
    private BroadcastReceiver receiver = null;
    private Button buttonFind;
    private Button buttonConnect;
    private WifiP2pDevice device;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        manager = (WifiP2pManager)
        getSystemService(Context.WIFI_P2P_SERVICE);
```

```
        channel = manager.initialize(this, getMainLooper(), null);

        intentFilter.addAction(WifiP2pManager.
         WIFI_P2P_STATE_CHANGED_ACTION);
        intentFilter.addAction(WifiP2pManager.
        WIFI_P2P_PEERS_CHANGED_ACTION);
        intentFilter.addAction(WifiP2pManager.
        WIFI_P2P_CONNECTION_CHANGED_ACTION);
        intentFilter.addAction(WifiP2pManager.
        WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);

        receiver = new
        Chapter9WiFiDirectBroadcastReceiver(manager, channel,
        this);
        registerReceiver(receiver, intentFilter);

        this.buttonConnect = (Button)
        this.findViewById(R.id.buttonConnect);
        this.buttonConnect.setOnClickListener(this);

        this.buttonFind =
        (Button)this.findViewById(R.id.buttonFind);
        this.buttonFind.setOnClickListener(this);
    }
}
```

The implementation is not complete currently. We will add the necessary methods step-by-step.

As you can see in this code, our `Activity` class implements various `Listeners` to handle the Wi-Fi Direct events. `ConnectionInfoListener` is for the callback when the connection info is available. `PeerListListener` is for the callback when the peer list is available. `ChannelListener` is for the callback when the channel is lost.

We create an intent filter and add the intents that we will check in the `onReceive()` method of the class that extends `BroadcastReceiver`.

We initialize the `WifiP2pManager` class by calling the `initialize()` method. This will register our application with the Wi-Fi network.

1. We need to override the `onChannelDisconnected()` method because we implemented `ChannelListener`, as shown in the following code block:

```
@Override
  public void onChannelDisconnected() {
    //handle the channel lost event
  }
```

2. We need to implement the `onPeersAvailable()` method because we implemented `PeerListListener`, as shown in the following code block:

```
@Override
  public void onPeersAvailable(WifiP2pDeviceList peerList) {

    for (WifiP2pDevice device : peerList.getDeviceList()) {
      this.device = device;
      break;
    }
  }
```

   We get the available `peerList` in this method. We get the first device and break the `for` loop. We need the device for connection.

3. We need to implement the `onConnectionInfoAvailable()` method because we implemented `ConnectionInfoListener`, as shown in the following code block:

```
  @Override
  public void onConnectionInfoAvailable(WifiP2pInfo info) {
    String infoname = info.groupOwnerAddress.toString();

  }
```

   This is the place where we get the connection info and connect and send data to the peer. For instance, an `AsyncTask` that transfers a file could be executed here.

4. We need to implement the `onClick()` method for the buttons:

```
  @Override
  public void onClick(View v) {
    if(v == buttonConnect)
    {
      connect(this.device);
    }
    else if(v == buttonFind)
    {
      find();
    }

  }
```

The `find()` and `connect()` methods are as follows:

```
public void connect(WifiP2pDevice device)
  {
    WifiP2pConfig config = new WifiP2pConfig();
    if(device != null)
    {
      config.deviceAddress = device.deviceAddress;
      manager.connect(channel, config, new ActionListener() {

          @Override
          public void onSuccess() {

            //success
          }

          @Override
          public void onFailure(int reason) {
            //fail
          }
      });
  }
    else
    {
      Toast.makeText(Chapter9Activity.this, "Couldn't connect,
      device is not found",

                Toast.LENGTH_SHORT).show();
    }
  }
       public void find()
  {
    manager.discoverPeers(channel, new
    WifiP2pManager.ActionListener()
       {

            @Override
            public void onSuccess() {
                Toast.makeText(Chapter9Activity.this, "Finding
                Peers",
                        Toast.LENGTH_SHORT).show();
       }
```

```
            @Override
            public void onFailure(int reasonCode)
           {
                Toast.makeText(Chapter9Activity.this, "Couldnt
                find peers ",
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
```

When the **find** button is clicked, we call the `discoverPeers()` method of `WifiP2pManager` to discover the available peers. As you will remember, calling the `discoverPeers()` method will cause `BroadcastReceiver` to receive the `WIFI_P2P_PEERS_CHANGED_ACTION` intent. Then we will request the peer list in `BroadcastReceiver`.

When the **connect** button is clicked, we call the `connect()` method of the `WifiP2pManager` using the device info. This starts a peer-to-peer connection with the specified device.

The sample application to introduce the Wi-Fi Direct APIs is complete with these methods.

Summary

In this chapter, we firstly learned the Android Beam feature of Android. With this feature, devices can send data using the NFC hardware. We implemented a sample Android Beam application and learned how to use Android Beam APIs. Secondly, we learned what Wi-Fi Direct is and how to use Wi-Fi Direct APIs.

Index

A

AAR 130
AbsoluteLayout 113, 114
Action Bar
  about 7, 8
  adding 9-13
  using, for navigation 30-35
ActionBar.TabListener class 32
actionLayout property 25
ActionProvider
  adding 14, 16, 17
  submenus, adding 19, 21
ActionProvider class 14, 17
action view 24, 25, 28, 29
Activity class 25, 55, 68, 124
addContact() method 57, 58
addContactStreamItem() method 59, 60
addEmail() method 58
addEvent() method 70, 73
addFragment() 86
addName() method 58
addPhoneNumber() method 58
addPreferencesFromResource method 97
addRawContact() method 58
Add Support Library menu option 122
Android
  Android Compatibility Package 119
  contacts 53
  device user profile 65
Android 3.2
  h<N>dp, screen size qualifiers 117
  screen size qualifiers 117
  sw<N>dp, screen size qualifiers 117
  w<N>dp, screen size qualifiers 117
Android Application Record. *See* **AAR**
Android applications
  user interface, designing
    considerations 106, 107
Android Beam
  about 127
  Activity class, implementing 129
  createMimeRecord() method 130
  NdefMessages, beaming 128
  onResume() method 130
Android Compatibility Package
  about 119
  Activity class 124
  Add Support Library menu option 122
  AndroidManifest.xml file 125
  Android SDK Manager button 120
  classes 119
  FragmentActivity class 125
  Fragment class 123
  getSupportFragmentManager() method 125
  Install button 121
  using 120, 121, 122, 125
Android Ice Cream Sandwich
  Android Beam 127
  space 49, 51
  Wi-Fi Direct 133
AndroidManifest.xml file 18, 125, 134
Android SDK Manager button 120
attendee
  adding 73, 74

B

beginTransaction() method 86
bitmap drawables
  for different densities 114, 115

borders_bottom_right background 41
Button component 74

C

Calendar APIs
about 67
attendee, adding 73, 74
CalendarContract.Attendee table 67
CalendarContract.Calendar table 67
CalendarContract.Event table 67
CalendarContract.Reminder table 67
event, creating 68-71
events creating, intents used 73
reminder, adding 74, 75
CalendarContract.Attendee table 67
CalendarContract.Calendar table 67
CalendarContract.Event table 67
CalendarContract.Reminder table 67
Chapter1ActionViewActivity method 29
Chapter1TabListener class 34
collapseActionView() method 29
collapseActionView property 25
columnCount property 38, 42
commit() method 86
connectivity APIs 127
connect() method 142, 143
contacts
about 53, 54
RawContact 53
StreamItemPhotos 54
StreamItems 54
ContentResolver class 68
ContentValues instance 70
createMimeRecord() method 130
createNdefMessage()method 130

D

density independent pixels. *See* **dip**
device
resolution 106
screen density 107
screen orientation 107
screen sizes 106
device user profile 65
DialogFragment 91, 93
dip
using, instead of px value 111
discoverPeers() method 137, 143
dismiss() method 95
Draw 9-patch tool 117

E

emulator properties
large screen properties 109
normal screen properties 108
small screen properties 108
events
creating 68, 70
creating, intents used 73
expandActionView() method 29

F

find() method 142
FragmentActivity class 125
Fragment class 123
FragmentManager 86
fragments
about 77
Activity class 83
addFragment() 86
adding, programmatically 85, 86
beginTransaction() method 86
commit() method 86
creating 79
DialogFragment 91-93
Fragment B layout 80
FragmentManager 86
getActivity() method 82
getFragmentManager() method 86
id property 83
layout_weight property 88
lifecycle 78
lifecycle, flow 78
LinearLayout layout 80
ListFragment 91
ListView component 80
managing 80-83
multiple fragments, using in activity 87-90
onActivityCreated method 81, 82
onActivityCreated() method 78

onAttach() method 78
onBookSelectedListener interface 87
onBookSelected method 87
onBookSelected property 89
onCreate() method 78
onCreateView() method 78
OnCreateView method 82
onDestroy() method 79
onDestroyView() method 78
onDetach() method 79
onItemClick method 86
onPause() method 78
onResume() method 78
onStart() method 78
onStop() method 78
popBackStack() method 86
PreferenceFragment 96, 97, 98
tag property 83
TextView component 80
types 91
WebViewFragment 99-104

G

getActivity() method 82
getCalendarID() method 70
getContactId() method 61
getFragmentManager() method 86
getStream() method 61
getSupportFragmentManager() method 125
GridLayout
about 37, 38
adding 39-42
configuring 43-49
uses 39
GridLayout layout 37

H

hasSubMenu() method 19, 21
h<N>dp, screen size qualifiers 117

I

id property 83
ImageButton component 16
ImageView component 128, 131
initialize() method 140
intents
using, to create events 73

L

layout_columnSpan property 42
layout_gravity property 38
layout_height properties 38, 107
layout_weight property 38, 88, 107
LinearLayout layout 9, 80
Listeners 140
ListFragment 91
ListView component 80

M

match_parent 110, 111

N

navigation
action bar, using 30-35
NdefMessages, beaming
setNdefPushMessageCallback() method 128
setNdefPushMessage() method 128
setOnNdefPushCompleteCallback() method 128
NFC Tag 130
nine-patch
about 117
Draw 9-patch tool 117

O

OnActionExpandListener class 29
onActivityCreated method 81, 82
onActivityCreated() method 78
onAttach() method 78
onBookSelectedListener interface 87
onBookSelected method 87
onBookSelected property 89
onChannelDisconnected() method 140
onClickListener() event 16
onClick method 95
onClick() method 141

onClick(View v) method 57
onCreateActionView() method 16
onCreateActionView() method 16
onCreate(Bundle savedInstanceState) method 57
onCreate() method 55, 78, 129
onCreateOptionsMenu(Menu menu) method 23, 28, 29
onCreateOptionsMenu method 10
onCreateView() method 78
onCreateView method 82
onDestroy() method 79
onDestroyView() method 78
onDetach() method 79
onItemClick method 86
onMenuItemClickListener events 21
onOptionsItemSelected method 10
onPause() method 78
onPeersAvailable() method 141
onPerformDefaultAction() method 16
onPrepareSubMenu(SubMenu subMenu) method 21
onReceive() method 135, 136
onResume() method 78, 130
onStart() method 78
onStop() method 78

P

PeerListListener 140
popBackStack() method 86
PreferenceFragment 96, 97, 98
processIntent() method 131
px value. *See* **dip**

R

RawContact
RawContactabout 53, 54
READ_CALENDAR permission 71
reminder
adding 74, 75
requestConnectionInfo() method 137
requestPeers() method 137
rowCount properties 38

S

screen sizes
about 105, 106
device, resolution 106
device, screen density 107
device, screen orientation 107
different screen sizes, supporting 105-107
screen sizesqualifiers, for Android 2.3 117
setNdefPushMessageCallback() method 128, 129
setNdefPushMessage() method 128
setOnActionExpandListener() method 29
setOnNdefPushCompleteCallback() method 128
setShareHistoryFileName method 23
ShareActionProvider attribute 21-24
ShareActionProvider. getShareIntent() method 23
showAsAction property 25
show() method 93
Social API 54-64
space 49-51
StreamItemPhotos 54
StreamItems 54
submenus
ActionProvider, adding 19, 21
sw<N>dp, screen size qualifiers 117

T

tag property 83
TextView component 38, 46, 110, 111, 128

U

user interface
designing considerations, for Android applications 106, 107
user interface, designing
AbsoluteLayout, use omitting 113, 114
bitmaps, for screen densities 114, 115
dip using, instead of pix 111, 112
large screen, properties 109, 110
layout, for screen sizes 115-117
nine-patch feature 117

normal screen, properties 108
small screen, properties 108
to support screen sizes 106, 107
wrap_content used 107

W

WebViewFragment 99, 102, 104
Wi-Fi Direct
about 133
onClick() method, implementing 141
onConnectionInfoAvailable() method, implementing 141
sample application 134-137
Wi-Fi Direct connection 134
WifiP2pManager class 133, 140
WIFI_P2P_PEERS_CHANGED_ACTION 137
WIFI_P2P_THIS_DEVICE_CHANGED_ACTION 137
w<N>dp, screen size qualifiers 117
wrap_content 107, 110, 111
WRITE_CALENDAR permission 71



Thank you for buying

Android 4: New features for Application Development

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



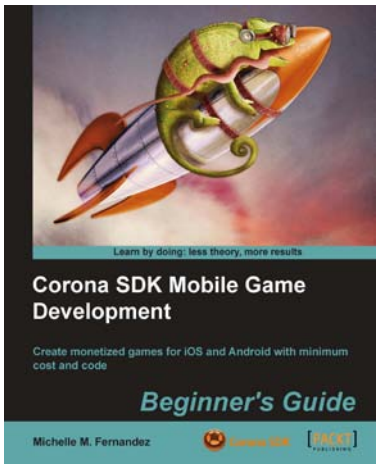
Responsive Web Design with HTML5 and CSS3

ISBN: 978-1-84969-318-9 Paperback: 324 pages

Learn responsive design using HTML5 and CSS3 to adapt websites to any browser or screen size

1. Everything needed to code websites in HTML5 and CSS3 that are responsive to every device or screen size
1. Learn the main new features of HTML5 and use CSS3's stunning new capabilities including animations, transitions and transformations
1. Real world examples show how to progressively enhance a responsive design while providing fall backs for older browsers

---



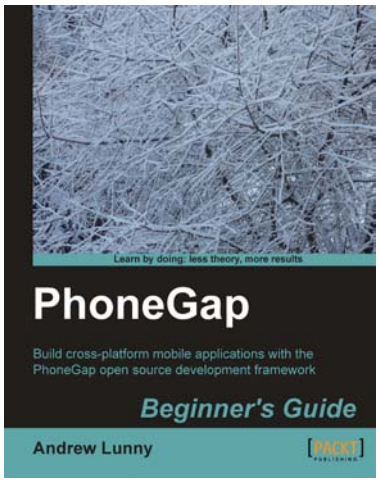
Corona SDK Mobile Game Development: Beginner's Guide

ISBN: 978-1-84969-188-8 Paperback: 408 pages

Create monetized games for iOS and Android with minimum cost and code

1. Build once and deploy your games to both iOS and Android
2. Create commercially successful games by applying several monetization techniques and tools
3. Create three fun games and integrate them with social networks such as Twitter and Facebook

[PACKT] open source
community experience distilled
PUBLISHING



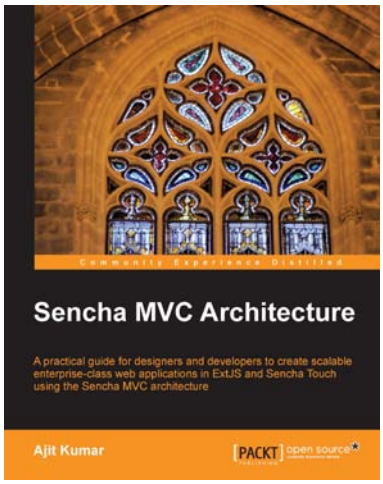
PhoneGap Beginner's Guide

ISBN: 978-1-84951-536-8 Paperback: 328 pages

Build cross-platform mobile applications with the PhoneGap open source development framework

1. Learn how to use the PhoneGap mobile application framework
2. Develop cross-platform code for iOS, Android, BlackBerry, and more
3. Write robust and extensible JavaScript code
4. Master new HTML5 and CSS3 APIs
5. Full of practical tutorials to get you writing code right away

---



Sencha MVC Architecture

ISBN: 978-1-84951-888-8 Paperback: 126 pages

A practical guide for designers and developers to create scalable enterprise-class web applications in ExtJS and Sencha Touch using the Sencha MVC architecture

1. Map general MVC architecture concept to the classes in ExtJS 4.x and Sencha Touch
2. Create a practical application in ExtJS as well as Sencha Touch using various Sencha MVC Architecture concepts and classes
3. Dive deep into the building blocks of the Sencha MVC Architecture including the class system, loader, controller, and application

Please check **www.PacktPub.com** for information on our titles