# Composing Mobile Apps

Learn | Explore | Apply

*using* Android™

Anubhav Pradhan
Anil V Deshpande

CD contains

- Step-by-step code to create a Live App
- Deployment guide
- Chapter wise code snippets

WILEY

# Composing Mobile Apps

Learn | Explore | Apply

*using* Android™

# Composing Mobile Apps

Learn | Explore | Apply

*using* Android™

Anubhav Pradhan

Anil V Deshpande

WILEY

Composing Mobile Apps

Learn | Explore | Apply *using* Android™

Other Wiley Editorial Offices:

John Wiley & Sons, Inc. 111 River Street, Hoboken, NJ 07030, USA

Wiley-VCH Verlag GmbH, Pappellaee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 1 Fusionopolis Walk #07-01 Solaris, South Tower, Singapore 138628

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada, M9W 1L1

First Edition: 2014

ISBN: 978-81-265-4660-2

ISBN: 978-81-265-8158-0 (ebk)

www.wileyindia.com

# Dedication

My respected parents, dear wife Shivani and two wonderful sons Gahan and Gatik

—Anubhav Pradhan

My beloved parents, wonderful wife Priya and charming daughter Anika

—Anil V Deshpande

# Foreword

When mobile apps were first introduced, they had limited purpose − media consumption, reading e-books and web browsing. Now, apps are much more than just shortcuts to open music playlists.

Today, mobile apps are used for everything – banking, travel itineraries, learning and education and even to watch television. It doesn't end here. Apps are used in medical science to monitor patient condition and for diagnosis. Apps are extensively used for payments/commerce. With a small attachment they allow credit cards to be read by phones. We are in an age when apps are seamlessly integrated into our lives, blurring the line between the real and virtual world.

Apps have disrupted the traditional role of technology organizations. While there is an app for everything, now anyone can develop an app. An app developer isn't just someone who works in a large multinational technology organization, it's anyone with an idea to share. Apps have given each one of us the ability to become inventors, empowering us to translate our inspiration into real useful applications.

However, for ideas to be represented accurately and for apps to be shared successfully, it is important to use the right reference to learn how to create apps. *Composing Mobile Apps,* authored by fellow Infoscions – Anubhav and Anil, serves as an excellent handbook to help one build effective apps, with pragmatic insights and guidance.

A young child in Indonesia was inspired to create education and translation apps as a phone tutor to his sister; developers were driven to create an app for personal safety when a woman was cruelly assaulted in India; and of course, three friends in an university in Finland were inspired by a stylized sketch of wingless birds to create, arguably, what is the world's largest mobile app success. Whatever your inspiration, I hope this book will help you translate it and tell the world your story.

**Sanjay Purohit**
Senior Vice President & Global Head − Products, Platforms & Solutions
Infosys Limited

# Preface

*The last thing one discovers in composing a work is what to put first.*

—T. S. Eliot

Mobility has established itself as the technology of this decade akin to how Internet revolutionized the previous. Mobile apps have acted as a catalyst to the growth of both mobile platforms and smart devices. Gartner research predicts that 309.6 billion app downloads will happen by the year 2016. The revenue generated by these apps will be in the order of several billion dollars.

Android$^{TM}$ has immensely contributed to the growth of Mobility. Since its humble debut in 2007, it has been widely accepted as the mobile platform of choice by device manufacturers, consumers, and developers alike, across the globe. Android platform is sported on a wide variety of devices including smart phones, tabs, smart watches, smart televisions, e-readers, gaming consoles, jukeboxes, smart cars and even space projects.

Behind the scenes, app stores host millions of apps for almost anything and everything, and are fuelling this growth. App stores have created a competitive ecosystem for individual developers and enterprises alike, where they can share their ideas − realized as useful applications − with their end-users. On the other hand, end-users have got multitude of options to select from, and get the best as per their individual preferences.

Though we have apps for everything, and anyone can develop an app, the app development is a niche skill that presents its unique challenges to developers and enterprises. Diverse consumer preferences, multitude of available devices and their features, security and privacy of data, and the environment in which the app is accessed are a few key ones.

This book attempts to address these challenges while presenting various approaches and technologies to build mobile apps. It takes the reader through several facets of composing mobile apps – design, development, validation, packaging and distribution of apps. Using Android as the means to expound these concepts, it presents a holistic view of mobile app development to the readers.

This book also presents the readers a unique opportunity to experience mobile app development by incrementally building a sample app, over its course. The chosen reference app is so designed that it enables them to implement most of the topics discussed in this book, thus providing them the much needed hands-on experience.

This book is intended for all mobility enthusiasts – beginners and professionals alike – who are looking forward for realizing their ideas into winning apps.

This book is categorized into four parts – *Introduction, Building Blocks, Sprucing Up* and *Moving to Market*.

The first part – *Introduction* – describes the world of mobility and discusses various approaches and technologies for app development. In particular, it describes the Android platform, its architecture and setup of the development environment required to compose Android apps. It outlines the reference app – 3CheersCable – that is going to be built incrementally over the course of this book (illustrated in *Let's Apply* sections of this book). It also familiarizes the readers with various challenges that app development may present, and proposes tenets of composing winning apps.

The second part – *Building Blocks* – takes a deep dive into the fundamentals of developing Android apps. This part focuses on designing app user interface, implementing app functionality and handling an app's data needs. It discusses several key concepts of designing user interface (UI) in Android apps such as Activity and its lifecycle, UI resources, Fragments and Action bar. It examines various ways to deal with long running tasks in an Android app by illustrating the usage of Threads, AsyncTasks and Services. It also explains several other key concepts of Android such as Intents, Broadcast Receivers, Notifications, telephony and SMS services. This part further describes persistence and access of local app data using Flat files, Shared Preferences, SQLite databases and Content Providers. It also sheds light on the consumption of enterprise data exposed using Web services.

The third part – *Sprucing Up* – guides the readers with the usage of Graphics, Animation, Multimedia, Sensors, Location and Maps capabilities in their apps. It discusses commonly used Graphics and Animation libraries of the Android framework, describes various multimedia options available for the capture and playback of audio and video, illustrates the usage of Location services and Maps to create location aware apps, and also delves into the usage of commonly available device sensors while building Android apps.

The final part – *Moving to Market* – introduces comprehensive validation landscape of mobile apps with a detailed focus on unit testing of individual Android app components using Android JUnit framework. The readers are also presented with the process and strategies to publish and distribute their apps.

The accompanying CD contains the chapter wise code snippets discussed over the course of this book, incremental builds (and the final codebase) of the reference app – 3CheersCable, and a deployment guide that assists the readers in setting up the app in their development environment.

Trust you will enjoy reading this book, as much we enjoyed bringing it to you!

Anubhav Pradhan (pradhan.anubhav@gmail.com)
Anil V Deshpande (anildesh82@gmail.com)

# Acknowledgements

During the course of bringing up this book, we have been supported by many of our friends and colleagues. In no particular order, we would like to place on record our heartfelt thanks to all of them.

To start with, we would like to acknowledge and thank Infosys Limited immensely for supporting and encouraging this project. We are highly indebted to Sanjay Purohit, SVP & Global Head – PPS (Products, Platforms & Solutions), for his encouragement and the inspirational foreword that he has been kind enough to share with the readers. We would like to specially thank Binod H. R., SVP and Global Head – TAFF (Talent Acquisition and Fulfillment Function), Nandita Gurjar, SVP and Global Head – ETA (Education, Training and Assessment), Srikantan Moorthy, SVP and Global Head – HR, Sudeep Gautam, VP and Head – Enterprise Mobility practice, Ramesh Adiga, AVP and Global Delivery Head – Enterprise Mobility practice, and Anup Uppadhayay, VP and Service Offering Head – Financial Services, for their constant support and encouragement during the course of this project. We would also like to thank Subrahmanya S. V., VP-ETA, Satheesha B. N., AVP-ETA and Acharya K.N.S., AVP-ETA for providing crucial inputs and specific directions to make this project successful. We are grateful to them for being our guides and mentors.

We are highly grateful to Anoop Kumar P, Principal Technology Architect – Enterprise Mobility practice, for his inputs and critical technical review of this book. His timely support in reviewing the manuscript and suggestions for some crucial improvements in the text is highly appreciated. Our distinctive thanks are due to our team of four young technology evangelists – Sekhar Subramanian, Akash Malik, Prashanth S. P. and Akhil Sharma, for their untiring support in coding, documenting, testing and implementing the reference application. We highly appreciate their efforts in this exercise.

We would like to thank J. A. Eswaran, Prajith Nair, Prashanth Puranik and Sivaramakrishnan Natarajan for reviewing the manuscript from a reader's perspective, and making it as close as you would like it. We would also like to thank Saumya Jain, Prashant Srivastava, Md. Akram Ulla Shariff and Nandeesha Ramesh for providing timely support in this project.

Our special thanks goes to the entire team of Wiley, India – including Paras Bansal, Executive Publisher; Meenakshi Sehrawat, Executive Editor; and Rakesh Poddar, Production Editor – for working closely with us and producing this book in time.

Last, but not the least, we would also like to thank our fellow Infoscions for inspiring us throughout this endeavor.

Anubhav Pradhan
Anil V Deshpande

# Table of Contents

# Part I
# Introduction

# 1 Mobility and Android

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Describe mobility.
- Classify mobility panorama.
- Describe mobile platforms.
- Outline mobile app development approaches.
- Describe Android platform and its architecture.

## 1.1 INTRODUCTION

From being active on social networks to while away time in playing games, from locating a place of interest to getting lost in the melodies of a favorite number, from managing money using financial apps to spending a whole lot in online shopping, from organizing the days using a planner to unwinding evenings watching movies – mobility is everywhere.

Mobility is not just about glittering mobiles or gleaming tabs, but about transforming user experience from the confines of a desk to the convenience of anytime–anywhere. The spontaneity, ubiquity, and indispensability of mobility are indeed defining it as the *Seventh Sense*. This chapter lays the foundation for the rest of the book. Readers in this chapter will get introduced to the mobility panorama, get an overview of mobile platforms, and explore the mobile app development approaches. They will also get introduced to the Android platform and its architecture.

## 1.2 MOBILITY PANORAMA

The entire mobility panorama can be broadly classified into logical landscape and physical ecosystem, as depicted in Fig. 1.1. The logical landscape describes the rationale behind mobility for different stakeholders, whereas the physical ecosystem portrays the infrastructure that enables mobility.

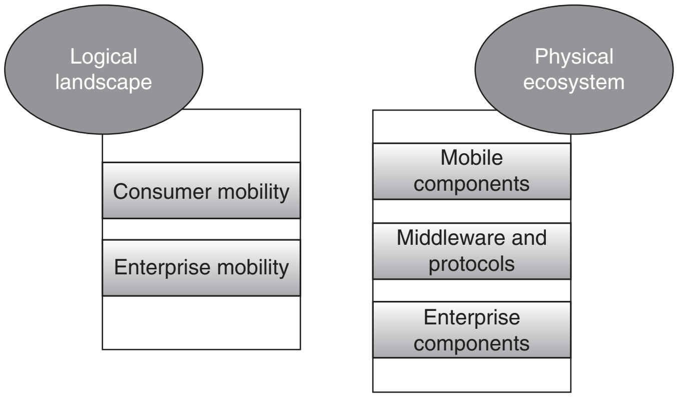


**Figure 1.1** | Mobility panorama.

Logical landscape defines two predominant logical components of mobility – consumer mobility and enterprise mobility. Consumer mobility is focused toward the end user, and typically comprises mobility solutions such as social networking, games, shopping, bidding, and utilities (see Fig. 1.2). Enterprise mobility is focused toward various stakeholders of an organization such as vendors, partners, suppliers, workforce, and their end customers. Enterprises across industry segments are leveraging the mobility channel to extend their reach and simplify business processes to cater to more and more customers, and in turn, increase profits. At the same time, mobility is enabling enterprises to increase productivity of their mobile workforce. A few representative industries and mobility solutions are illustrated in Fig. 1.2.

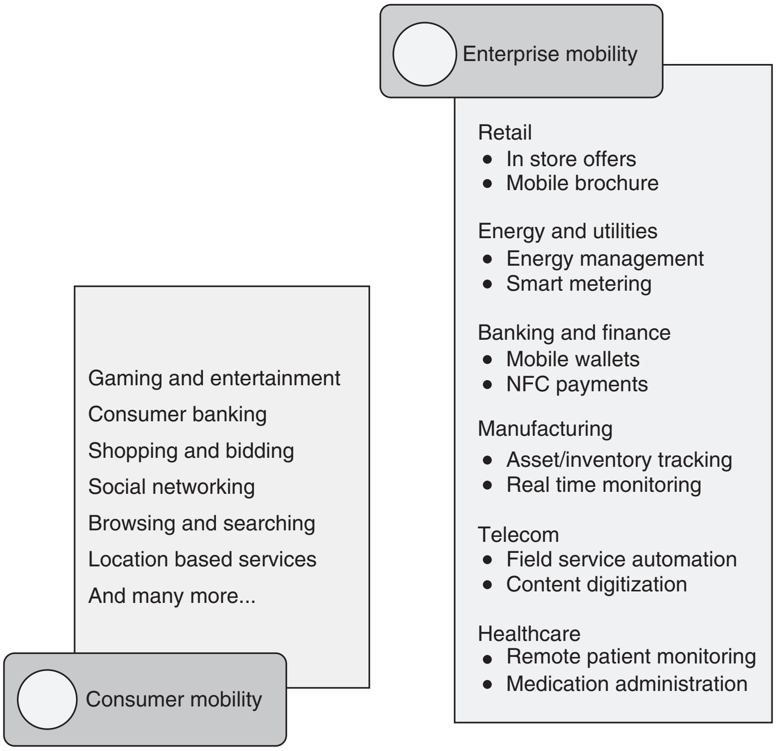


**Figure 1.2** | Logical landscape of mobility.

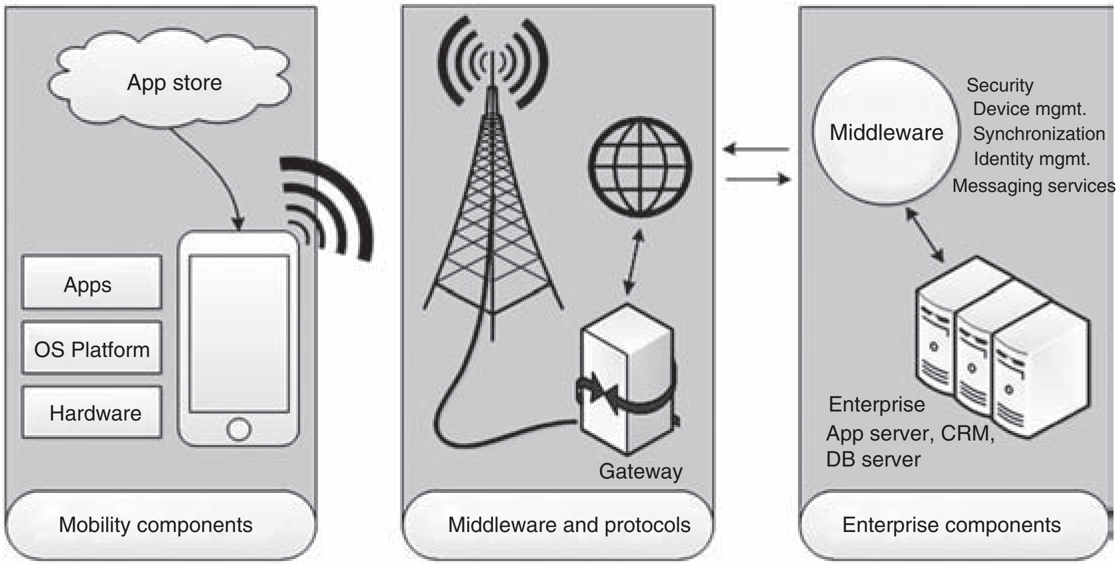


**Figure 1.3** | Physical ecosystem of mobility.

The physical ecosystem of mobility encompasses three key distinct physical components – mobility components, enterprise components, and middleware and protocols that glue the first two components (see Fig. 1.3).

The key mobility components are mobile devices, mobile platforms, and mobile app stores. Mobile devices are the centerpiece of mobility, and usually come in varieties of shapes and sizes such as smart phones, tablets, phablets, e-readers, music players, and smart watches. Mobile platforms, such as Android and Apple iOS, are software stacks that power mobile devices. We will further explore the mobile platforms in Section 1.3. Mobile app stores are online distribution systems or market places of mobile apps. Apple® App Store™ and Google Play™ are the two most popular app stores.

Enterprise components typically comprise hosts of servers, such as database servers and application servers, that cater to enterprise portion of mobility solutions. They also comprise enterprise solutions that cater to the requirements of data security, data synchronization between mobile devices and enterprise servers, and identity management of mobile users.

The component – middleware and protocols – acts as a glue between mobility and enterprise components. Access mechanisms such as Wi-Fi, Bluetooth, Code Division Multiple Access (CDMA), General Packet Radio Service (GPRS), and Global System for Mobile Communications (GSM) are some of the key components of this layer that allow mobile devices to communicate. Other key components are gateways such as Wireless Application Protocol (WAP) and Short Message Service (SMS) gateways that enables interaction between mobile devices and the Internet.

## 1.3 MOBILE PLATFORMS

Operating systems that power and manage the computing resources have come off the ages – right from mainframe operating systems to desktop and server operating systems – and now mobile operating systems, more popularly known as mobile platforms.

Technically, a mobile platform is not just an operating system, but a software stack that typically comprises an operating system, libraries, and application development framework(s). The operating system contributes to the core features of the platform such as memory management, process management, and various device drivers. The libraries furnish the much needed core functionality of the platform such as media libraries that provide codecs, libraries for native data storage, libraries for rendering screens and drawing surfaces, and libraries for graphics. The application development framework is the set of Application Programming Interfaces (APIs) that in turn interact with the underlying libraries, and are exposed to the developers for app development.

Android, Apple iOS, BlackBerry®, and Windows Phone® are among the most popular mobile platforms. As these platforms are device specific, they are therefore at times also referred to as native platforms, and the apps developed using them are referred to as native apps.

## 1.4 APP DEVELOPMENT APPROACHES

The development of a mobile app is purely driven by the following three key aspects:

1. The first key aspect is the business use case that the app is going to mobilize. A business use case can be a business-to-consumer (B2C) scenario such as delivering in-store offers on customers' devices in a retail outlet or facilitating banking on consumers' devices. The use case can also be a business-to-employee (B2E) scenario such as providing guided sales facilities on sales workforce devices or enabling remote patient monitoring through smart devices for medical practitioners. It can also

be a business-to-business (B2B) scenario such as mobilizing supply chain management of an enterprise and respective vendors/partners/suppliers.

2. The second key aspect is the profile of the user who is going to use the app. The user may be a field agent, customer, partner, executive, or a member of senior management.
3. The third key aspect is the mobile device that plays the app. The device type (sophisticated smart device/rugged device), device connectivity (offline/online), and the operating environment (office/on-the-move) play key roles in this case.

Predominantly, these three key factors determine the app development requirements with respect to user experience, device diversity, offline capabilities, security, backend integration, app life cycle management, etc.

There are three broad approaches to develop a mobile app, namely native, web, and hybrid (see Fig. 1.4).

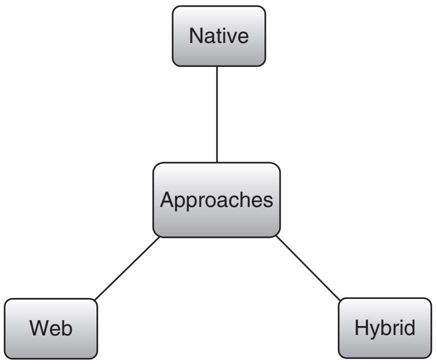


**Figure 1.4** | Mobile app development approaches.

The *native approach* yields a native app that is developed for the native platform, using platform specific APIs. The native app is distributed through online app stores, from where it can be downloaded and installed on a mobile device. This approach is used primarily when the app requires a native look and feel; and needs to leverage high-end device capabilities.

The *web approach* results in a mobile web app that is typically developed using web technologies such as HTML5, CSS3, and JavaScript. A web app is not installed on a mobile device; it rather gets rendered in a mobile browser, over the network. The web approach is used primarily when the app requires to cater to diverse devices using a single codebase. However, this approach will neither be able to provide a native look and feel, nor leverage high-end device capabilities.

The *hybrid approach* is a mixed approach that incorporates features of both native and web approaches while developing an app. Apps in hybrid approach are implemented using mobile cross platforms. Unlike mobile native platforms, these platforms do not power a mobile device. These platforms are device agnostic and facilitate multiplatform development, wherein the same codebase of a mobile app can be translated to fit into any of the supported native platforms. These platforms follow three broad philosophies to build apps using hybrid approach:

1. **Web-based philosophy**: Using web-based philosophy, a hybrid app is created in two steps. The first step creates a web app using the usual web approach, as explained earlier. The second step wraps the web app with a native wrapper. A native wrapper enables the app to leverage the underlying hardware features in a device. Frameworks such as jQuery Mobile or Sencha Touch are used to create the web app, and tools such as Adobe® PhoneGap™ are used to provide the native wrapper. The resultant hybrid app can be distributed through online app stores, from where it can be

downloaded and installed on a mobile device. At run time on the device, the app runs within a full-screen browser, giving an illusion of a native app to the user.

2. **Cross-compiler philosophy**: Using cross-compiler philosophy, a hybrid app is created using web technologies, and the resultant app can be cross compiled for the supported mobile native platforms (converted to relevant native app code). The resultant hybrid app can be distributed through online app stores, from where it can be downloaded and installed on a mobile device. At run time on the device, it runs like a native app. Tools such as Appcelerator Titanium™ are used to create these apps.
3. **Middleware philosophy**: A pure middleware philosophy enables a hybrid app to be hosted on a middleware server. Users can retrieve the app from middleware as and when required, while the middleware server will facilitate the interaction between the app and the enterprise systems at the backend. Though full-fledged commercial Mobile Application Development Platforms (MADPs) such as SAP® SMP (SAP Mobile Platform) or Kony™ are primarily used as middleware, they do support other philosophies of hybrid mobile app development as well.

In this book, we will learn, explore, and apply mobile app development using the native approach, and Android is used as the mobile native platform to implement this approach.

## 1.5 ANDROID OVERVIEW

Android is an open source mobile native platform governed by Open Handset Alliance (OHA) and led by Google. Since its debut in 2007, it has grown phenomenally and established itself as a comprehensive software platform for smart devices with varying form factors and features.

Android has revolutionized the mobile market by democratizing the platform. It has been widely embraced both by Original Equipment Manufacturers (OEMs) and by app developers due to its openness. Android is highly customizable because of which OEMs can modify and distribute their own flavors of the platform. Android has also attracted a wide community of developers because of the ease with which they can start building the apps.

The Android platform follows a layered architecture approach as depicted in Fig. 1.5. The platform is based on a Linux kernel and has native libraries, Android runtime, application framework, and applications as the layers in the software stack. As developers, we will be using the application framework layer to develop apps. These apps reside in the applications layer along with prebuilt apps that come bundled with the platform/device.

The *Linux kernel* provides the core operating system infrastructure such as memory management, process management, security model, networking, and various device drivers.

The *native libraries* lie at the heart of the Android platform, and are written in C and C++. The surface manager library is responsible for rendering windows and drawing surfaces of various apps on the screen. The media framework library provides media codecs for audio and video. The SQLite library provides support for native data storage. The OpenGL (Open Graphics Library) and SGL (Scalable Graphics Library) are the graphics libraries for 3D and 2D rendering, respectively. The FreeType library is used for rendering fonts, and the WebKit library is a browser engine for the Android browser.

The *Android runtime* is designed to run apps in a constrained environment that has limited muscle power in terms of battery, processing, and memory. It has two key components – Dalvik Virtual Machine (DVM) and core libraries. The DVM runs the executable files of an app. We will further explore DVM in Chapter 2. The core libraries are written in Java, and comprise core Java libraries such as Utility, Collections, and IO (input/output).

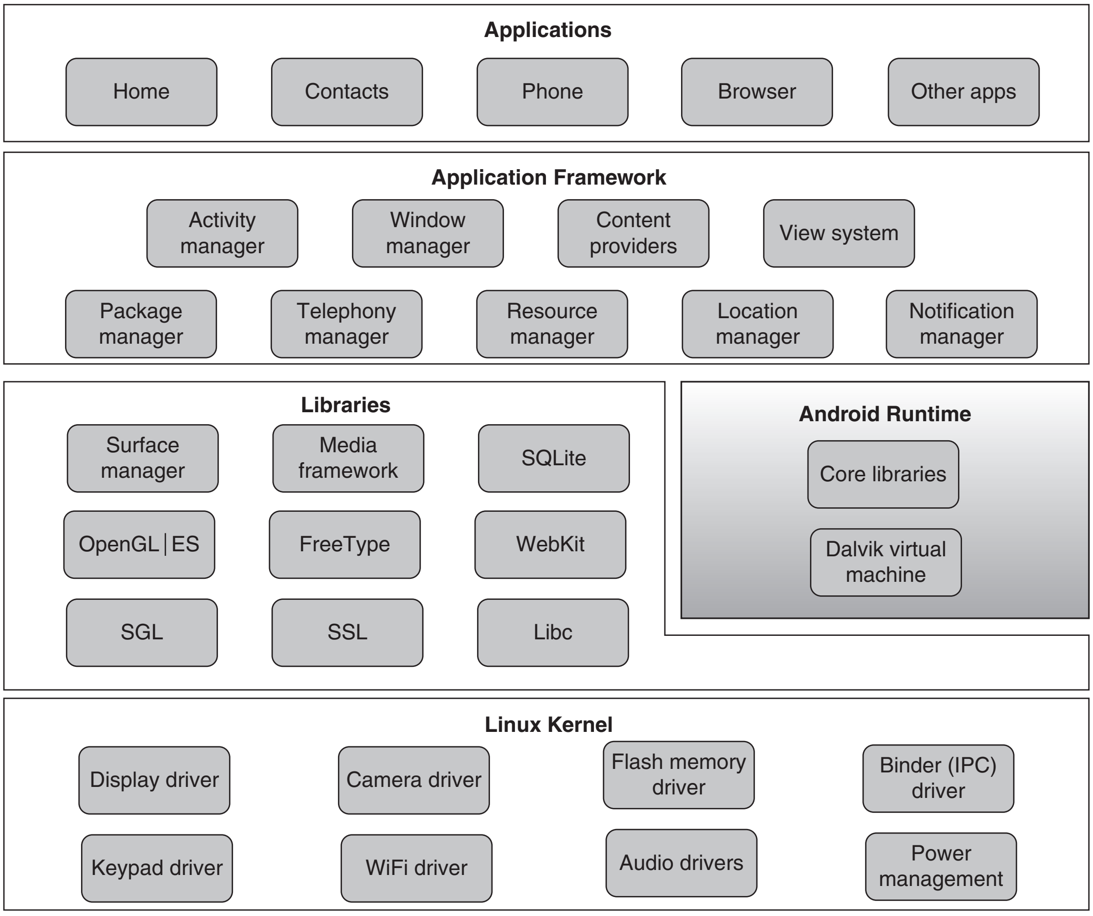


**Figure 1.5** | Android platform architecture.

The *application framework* is a collection of APIs that are very crucial for an Android developer as the framework forms the basis of the development of Android apps. This framework is made available to developers as Android Software Development Kit (SDK) that comes along with the Android Developer Tools (ADT) bundle (refer to Section 2.2 of Chapter 2). This layer is written in Java. The window manager manages windows and drawing surfaces, and is an abstraction of the surface manager library. Content providers provide the mechanism to exchange data among apps. The package manager keeps a tab on the apps installed on the device. The telephony manager enables app to leverage phone capabilities of the device. The resource manager is used to store the resources of an app such as bitmaps, strings, layouts, and other artwork. The view system contains the User Interface (UI) building blocks such as buttons, check boxes, and layouts, and also performs the event management of UI elements. The location manager deals with location awareness capabilities, and the notification manager deals with notifications on mobile devices.

The *applications* layer is the topmost layer that contains all the apps installed on the device, including those that come bundled with it. This layer uses all the layers below it for proper functioning of these mobile apps.

Since its humble beginning as an operating system for mobile devices, because of its openness, Android has been serving in varieties of areas – ranging from smart phones to tabs, televisions to watches, set-top boxes to juke boxes, e-readers to gaming consoles, cars to space projects – truly paving its way to become one of the preferred *Operating System of Things*. Android is everywhere and for everyone.

## SUMMARY

This chapter has introduced mobility and made an attempt to classify the entire mobility panorama by looking at it from two viewpoints – logical landscape and physical ecosystem. The logical landscape emphasizes on the world of consumer and enterprise mobility. The physical ecosystem depicts the nuts and bolts that enable mobility.

The chapter has elucidated three approaches to develop mobile apps – native, web, and hybrid. Scenarios where these approaches are applicable and technologies that are used to implement them are discussed to bring out the similarities and differences among them.

The chapter has also presented a brief primer on Android with a focus on understanding the design and architecture of the platform.

## REVIEW QUESTIONS

1. Define logical and physical landscape of mobility.
2. Define mobile native platforms with examples.
3. Illustrate three approaches to develop a mobile app along with the scenarios where we need to apply these approaches.
4. Illustrate the three philosophies of hybrid app development. List down the frameworks and tools that are used in these approaches.
5. Describe Android platform architecture. Discuss the various layers and their components and functions.

## FURTHER READINGS

1. Android: http://www.android.com/
2. Apple iOS: http://www.apple.com/
3. jQuery Mobile: http://jquerymobile.com/
4. Sencha Touch: http://www.sencha.com/products/touch
5. PhoneGap: http://phonegap.com/
6. Titanium: http://www.appcelerator.com/titanium/
7. SAP SMP: http://www54.sap.com/pc/tech/mobile/software/solutions/platform/app-development-earn.html
8. Kony: http://www.kony.com/

2

# Getting Started with Android

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Set up the development environment for Android.
- Write your first Android app.
- Outline a typical Android app project structure.
- Illustrate different logical components of an Android app.
- Use Android tool repository.
- Install and run an app on a mobile device.

## 2.1 INTRODUCTION

We have so far appreciated a quick primer on the world of mobility and Android; let us now get started with developing our first Android app – a plain vanilla "HelloWorld". Developing an Android app involves setting up the development environment, designing and creating the app, executing the app on an emulator, testing the app on a real device, and finally publishing it to the online app market.

This chapter familiarizes readers with the usage of development environment and related tools to create and debug Android apps. Readers will delve into the runtime environment and build proces of Android apps. The chapter also outlines the logical constituents of a typical Android app and its physical project structure. Towards the end of this chapter, readers will get familiarized with running an app on an Android mobile device.

## 2.2 SETTING UP DEVELOPMENT ENVIRONMENT

Android primarily uses Java for development; however, it also supports C, C++, and Python. Our main focus in this book is development using Java. Java Development Kit (JDK) is required for developing Java applications. The minimum version of JDK specified by Android is 1.5. In this book, the development is done using JDK 1.6, also referred to as J2SE 6.0.

Besides the JDK, following components are required to set up the development environment:

1. Android Software Development Kit (SDK).
2. Eclipse Integrated Development Environment (IDE).
3. Android Development Tools (ADT) plugin.

Android SDK is a software development kit that comprises the required libraries to build Android apps, development tools, and an emulator to run the app. In this book, development is done on Android SDK API level 18, popularly referred to as Android Jelly Bean (Android 4.3).[1]

It is recommended to use an IDE for quick, efficient, and professional development of apps. Eclipse is one of the preferred IDEs for Android apps development. In this book, Eclipse Juno is used.

The Eclipse IDE has to be adapted to Android app development with the help of ADT plugin. This provides guided features such as wizards in Eclipse IDE which helps developers in developing, debugging, and packaging mobile apps.

These three components are available for download as a single ADT bundle from the Android Developers website.[2] Assuming that the JDK1.6[3] has been installed, let us now get started with setting up the development environment. To get started, one has to just download the ADT bundle and unzip the contents as depicted in Fig. 2.1.

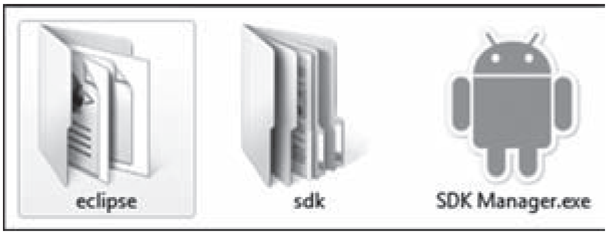


**Figure 2.1** | Contents of downloaded Android Development Tools bundle.

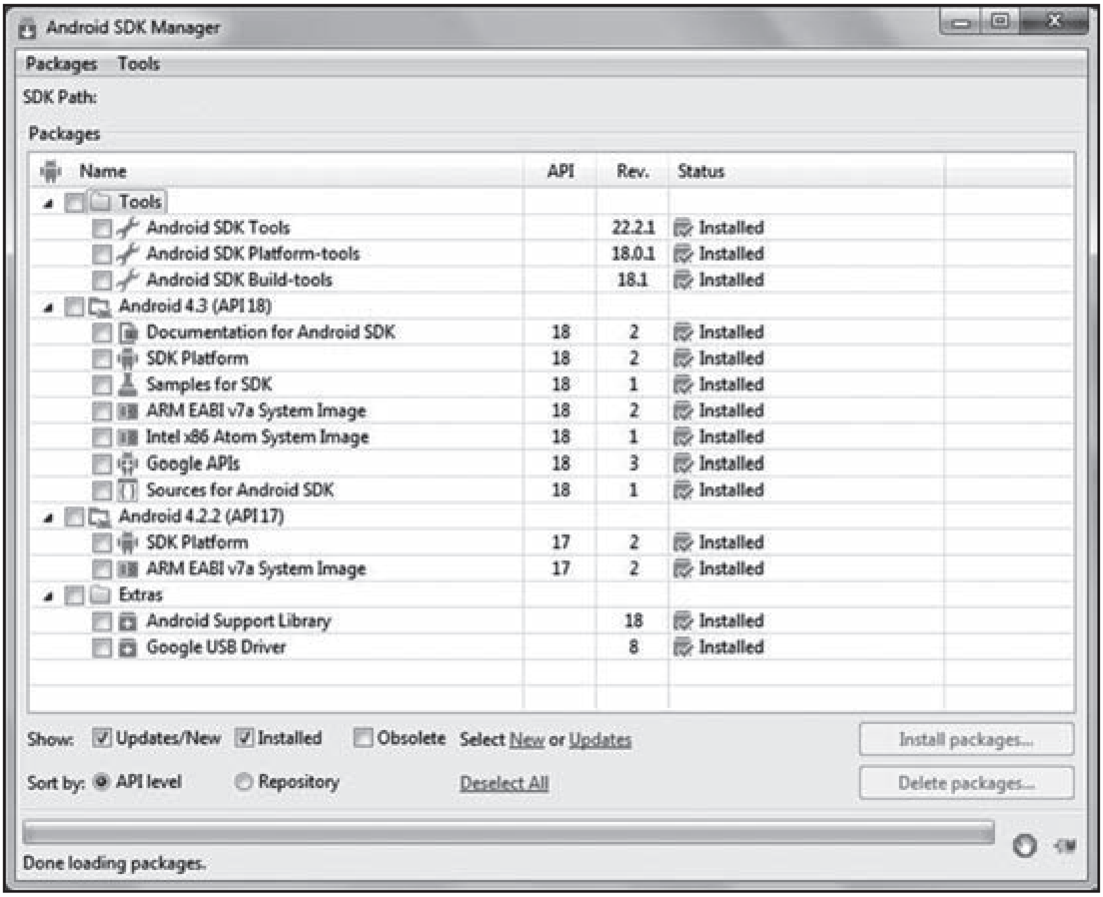
[1] At the time of writing, the latest version of Android platform was Android KitKat (Android 4.4); however, Android Jelly Bean's widespread usage among developers and device manufacturers made it the preferred one for the authors.

[2] http://developer.android.com

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html

By default, the SDK consists of only the latest version of Android, but there may be instances when one has to work on additional versions of Android. The Android SDK Manager comes handy to update the existing SDK. Figure 2.2 depicts the Android SDK Manager with two versions of Android – Android 4.3 (API 18) and Android 4.2.2 (API 17).

The development environment used in this book is set up on a Microsoft Windows® machine. An Android development environment can also be set up on an Apple Mac OS® or Linux machine.[4] There may be a few minor changes in the way the environment is set up on different operating systems.



SDK Path:

Packages

| Name | API | Rev. | Status |
|---|---|---|---|
| Tools | | | |
| Android SDK Tools | | 22.2.1 | Installed |
| Android SDK Platform-tools | | 18.0.1 | Installed |
| Android SDK Build-tools | | 18.1 | Installed |
| Android 4.3 (API 18) | | | |
| Documentation for Android SDK | 18 | 2 | Installed |
| SDK Platform | 18 | 2 | Installed |
| Samples for SDK | 18 | 1 | Installed |
| ARM EABI v7a System Image | 18 | 2 | Installed |
| Intel x86 Atom System Image | 18 | 1 | Installed |
| Google APIs | 18 | 3 | Installed |
| Sources for Android SDK | 18 | 1 | Installed |
| Android 4.2.2 (API 17) | | | |
| SDK Platform | 17 | 2 | Installed |
| ARM EABI v7a System Image | 17 | 2 | Installed |
| Extras | | | |
| Android Support Library | | 18 | Installed |
| Google USB Driver | | 8 | Installed |

Show: ✓ Updates/New ✓ Installed ☐ Obsolete Select New or Updates

Sort by: ● API level ○ Repository Deselect All

Done loading packages.

Figure 2.2 | Android SDK Manager.

## 2.3 SAYING HELLO TO ANDROID

After setting up the development environment, we are now ready to create Android apps. Figure 2.3 represents a typical flow of Android app development. The app development begins with designing app resources (layouts, navigation, artwork, etc.) and creating Android app components. Typically, these two tasks go hand in hand. It is followed by building the app source code that comprises compiling and creating an executable. Individual Android components of the app need to be unit tested before the whole app is

[4] http://developer.android.com/sdk/index.html

validated against its functional and nonfunctional requirements. Finally, the app can be distributed through various mechanisms such as app stores, websites, or even e-mails.

App resources are designed in Eclipse IDE, which facilitates both drag and drop and Extensible Markup Language (XML) editing mechanisms. Android app components can be created using Java. The Android SDK provides various tools to build and unit test the app source code. It also provides a sophisticated emulator to quickly see our app in action, even without a real device.

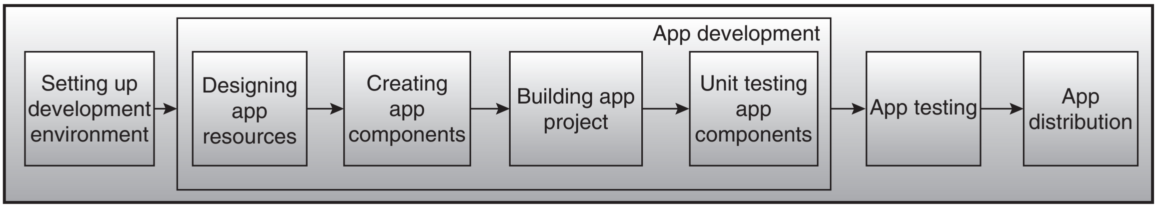


**Figure 2.3** | Android app development flow.

Let us now get started with creating our first Android app – a plain vanilla "HelloWorld" followed by setting up an emulator in which we shall launch it.

## 2.3.1 Creating the First App – Step by Step

Creating the first Android app is both simple and exciting. To begin with, start Eclipse (from ADT bundle) and choose a workspace – a folder where all projects would reside.

1. Right click on the *Package Explorer* window of the Eclipse workbench. Click on *New*. A set of options such as Java Project and Android Application Project appears to select from. Because we intend to create an Android app, select *Android Application Project*, as depicted in Fig. 2.4.

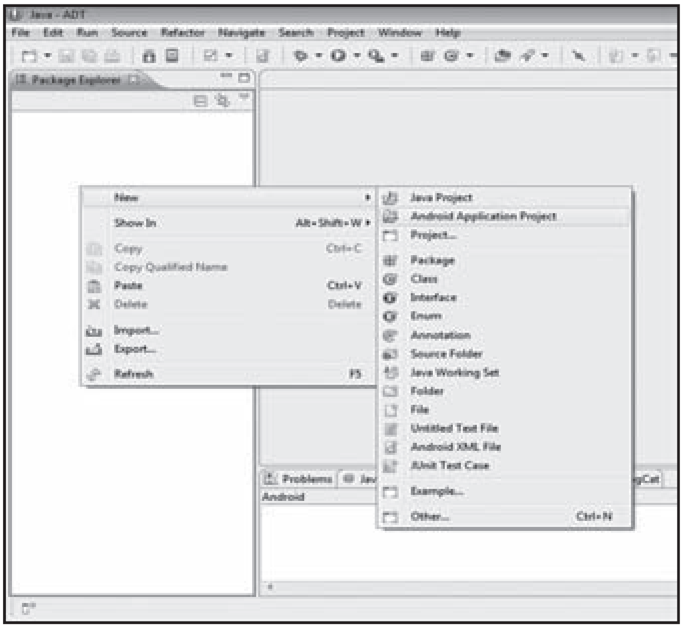
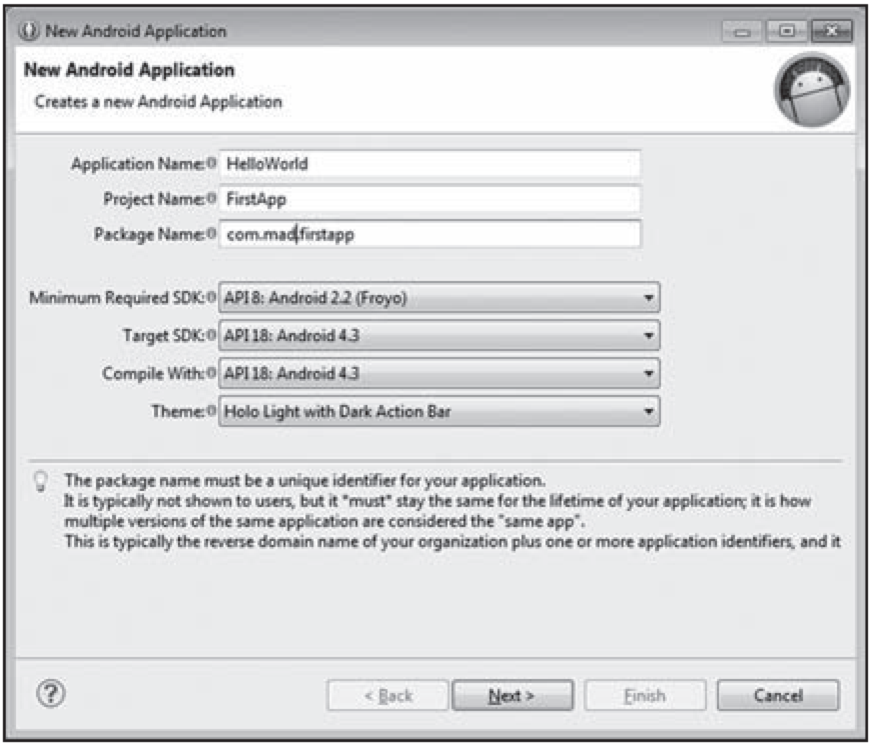


**Figure 2.4** | Creating an Android project.

2. A New Android Application wizard appears, as depicted in Fig. 2.5, to capture details such as *Application Name*, *Project Name*, and *Package Name*. Besides this, select other details such as *Minimum Required SDK*, *Target SDK*, *Compile With*, and *Theme* of the application. The purpose of each field is explained in the following list:
   - *Application Name* is the name of the mobile app from the end user's perspective.
   - *Project Name* is the name of the project from the developer's perspective.
   - *Package Name* is the logical namespace used to uniquely identify an app from the Android platform's perspective.
   - *Minimum Required SDK* is the least API level required on the device to support the app.
   - *Target SDK* is the API level for which the app is being targeted.
   - *Compile With* is the API level against which the app is compiled.
   - *Theme* is the style (look and feel) applied homogeneously to all screens of the app.
3. Click on *Next*. The Configure Project screen appears, as depicted in Fig. 2.6. Select *Create custom launcher icon* checkbox to help create an app icon that the end user will see on the device after installing the app. Select *Create activity* checkbox to create an Activity – a component that defines the user interface (UI) of an app. Select *Create Project in Workspace* checkbox to provide the default location in which the Android application project is going to reside.
4. Click on *Next*. The Configure Launcher Icon screen, as depicted in Fig. 2.7, appears that allows us to choose and configure the app icon.



**New Android Application**

Creates a new Android Application

Application Name: HelloWorld

Project Name: FirstApp

Package Name: com.mad.firstapp

Minimum Required SDK: API 8: Android 2.2 (Froyo)

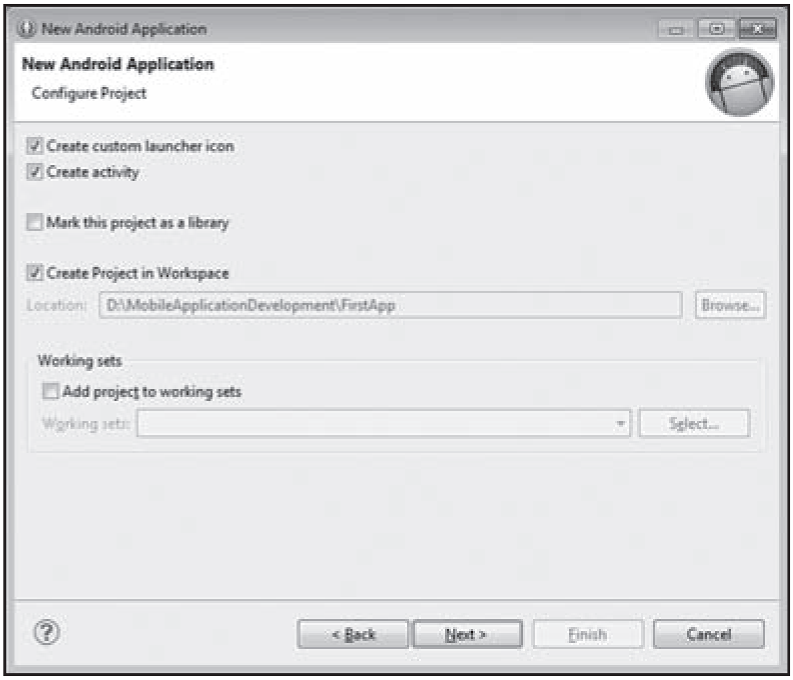
Target SDK: API 18: Android 4.3

Compile With: API 18: Android 4.3

Theme: Holo Light with Dark Action Bar

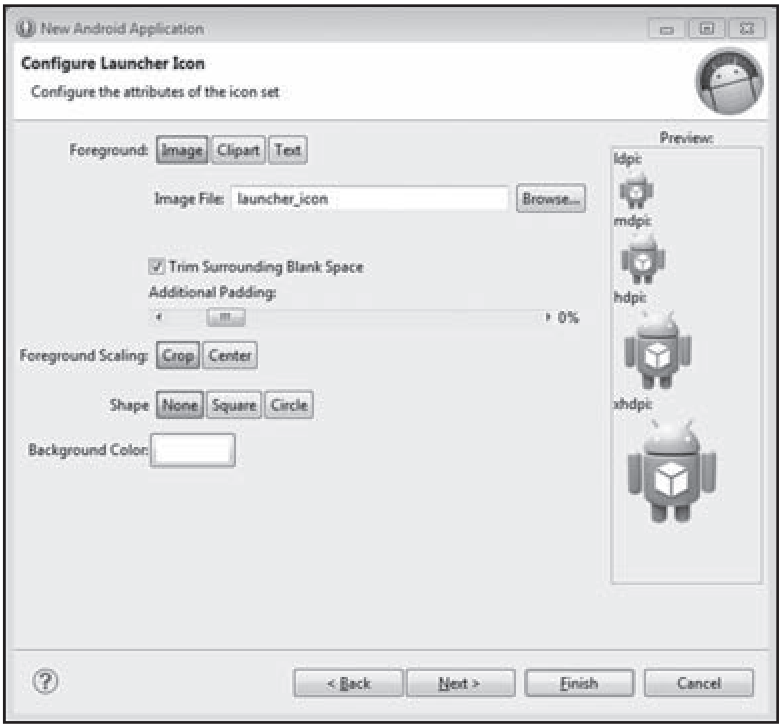
The package name must be a unique identifier for your application.
It is typically not shown to users, but it "must" stay the same for the lifetime of your application; it is how multiple versions of the same application are considered the "same app".
This is typically the reverse domain name of your organization plus one or more application identifiers, and it

**Figure 2.5** | New Android Application wizard.



☑ Create custom launcher icon
☑ Create activity
☐ Mark this project as a library
☑ Create Project in Workspace
Location: D:\MobileApplicationDevelopment\FirstApp
☐ Add project to working sets
Working sets:

**Figure 2.6** | New Android Application wizard – Configure Project.

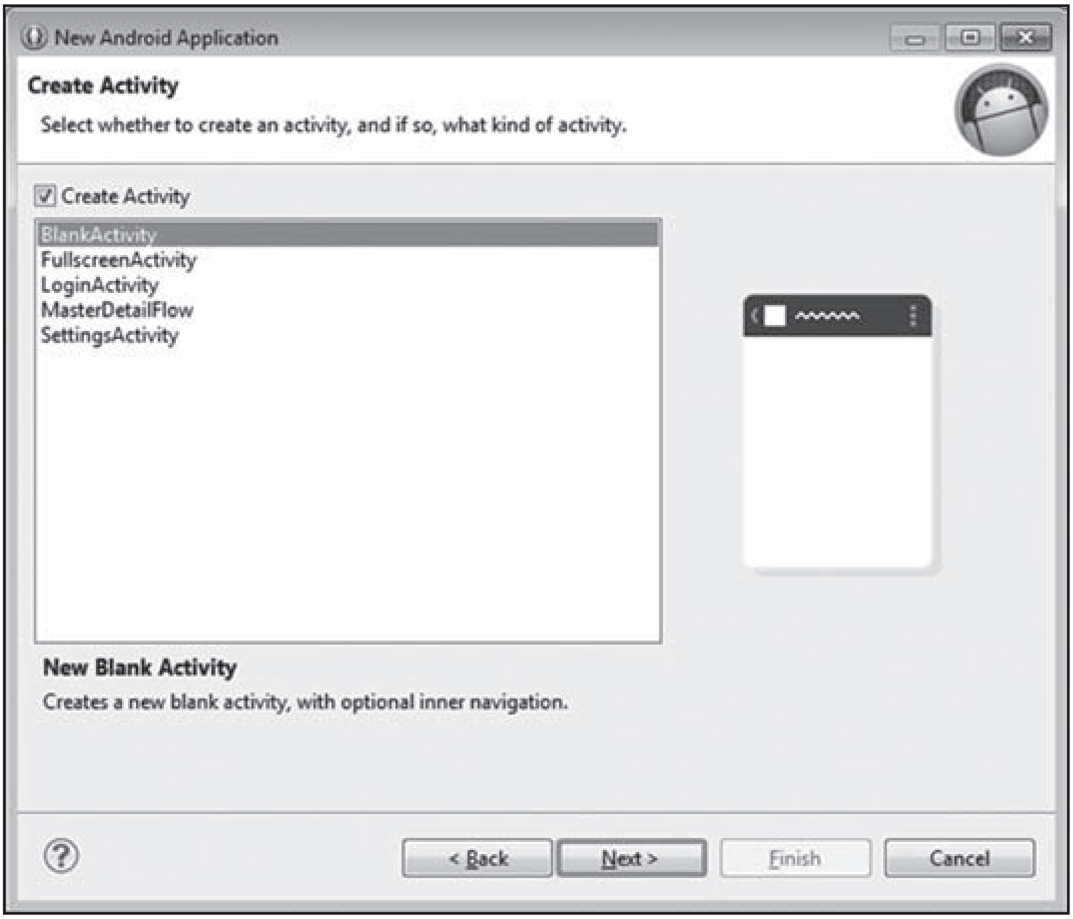


Foreground: [✓ Image] Clipart Text
Image File: launcher_icon
☑ Trim Surrounding Blank Space
Additional Padding: 0%
Foreground Scaling: [✓ Crop] Center
Shape: [✓ None] Square Circle
Background Color:

**Figure 2.7** | New Android Application wizard – Configure Launcher Icon.

**5.** Click on *Next*. The Create Activity screen appears as depicted in Fig. 2.8. This screen allows us to select the kind of launcher Activity to be created – the first screen that an end user is going to see on tapping the app icon on the device.

Creating an Activity for an app is not mandatory. An Android app can be developed without using an Activity, as the design decision is purely based on the objectives of the app. However, most of the apps typically have one or more Activities for human interaction. Let us create an Activity as part of our first app. Select the default option *BlankActivity*, and click on *Next*.


New Android Application

Create Activity

Select whether to create an activity, and if so, what kind of activity.

☑ Create Activity

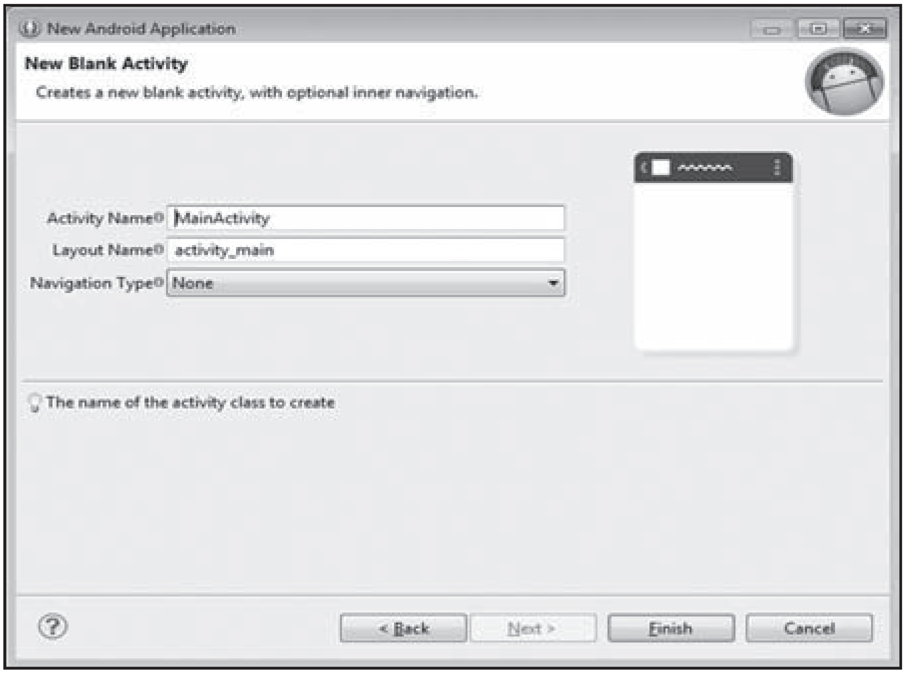
BlankActivity
FullscreenActivity
LoginActivity
MasterDetailFlow
SettingsActivity

New Blank Activity

Creates a new blank activity, with optional inner navigation.

**Figure 2.8** | New Android Application wizard – Create Activity.

**6.** The New Blank Activity screen, as depicted in Fig. 2.9, appears that allows us to provide details for creating a blank Activity. Provide the *Activity Name*, and the *Layout Name* of the layout file to be created. The layout is required to define the outline of UI elements such as buttons, sliders, and checkboxes on the screen.


New Android Application

New Blank Activity

Creates a new blank activity, with optional inner navigation.

Activity Name: MainActivity

Layout Name: activity_main

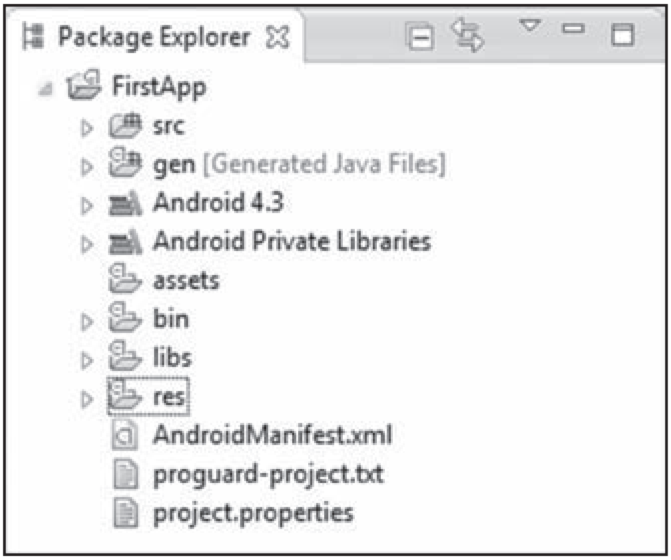
Navigation Type: None

The name of the activity class to create

**Figure 2.9** | New Android Application wizard – New Blank Activity.

**7.** Click on *Finish*, and that is it! Here is our first app!

In the Eclipse workbench, a folder structure gets created, as depicted in Fig. 2.10. This structure contains all the app artifacts that are generated for the HelloWorld app just created. We will explore this project structure in Section 2.4.


Package Explorer

```
FirstApp
  src
  gen [Generated Java Files]
  Android 4.3
  Android Private Libraries
  assets
  bin
  libs
  res
  AndroidManifest.xml
  proguard-project.txt
  project.properties
```

**Figure 2.10** | Project structure.

Let us now run the HelloWorld app.

Right click on the *FirstApp* project, and select *Run As → Android Application*, as depicted in Fig. 2.11.

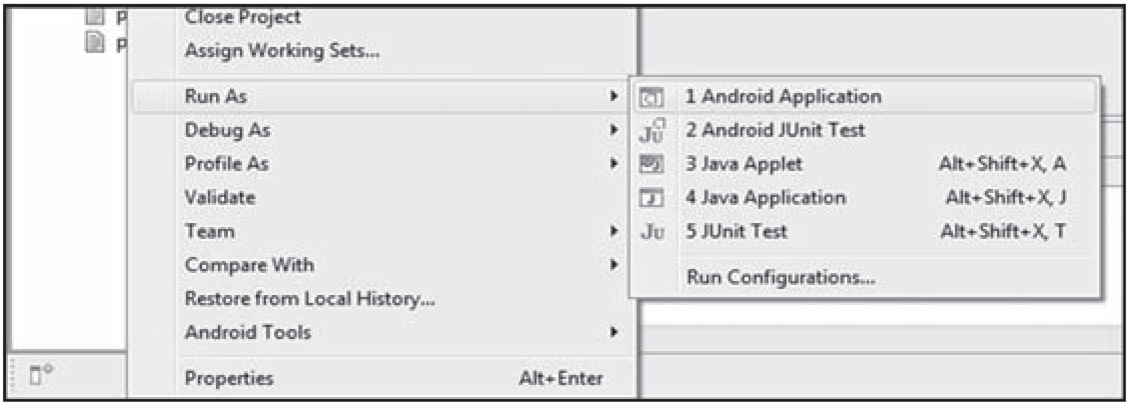
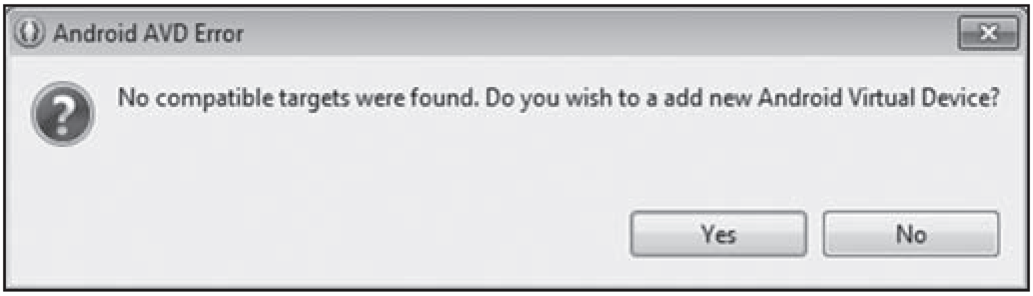


**Figure 2.11** | Running an Android app.

When we try to run the HelloWorld app, Eclipse may pop up a dialog window as shown in Fig. 2.12, with a message "No compatible targets were found. Do you wish to add a new Android Virtual Device?"



Android AVD Error

No compatible targets were found. Do you wish to a add new Android Virtual Device?

**Figure 2.12** | Android AVD Error dialog window.

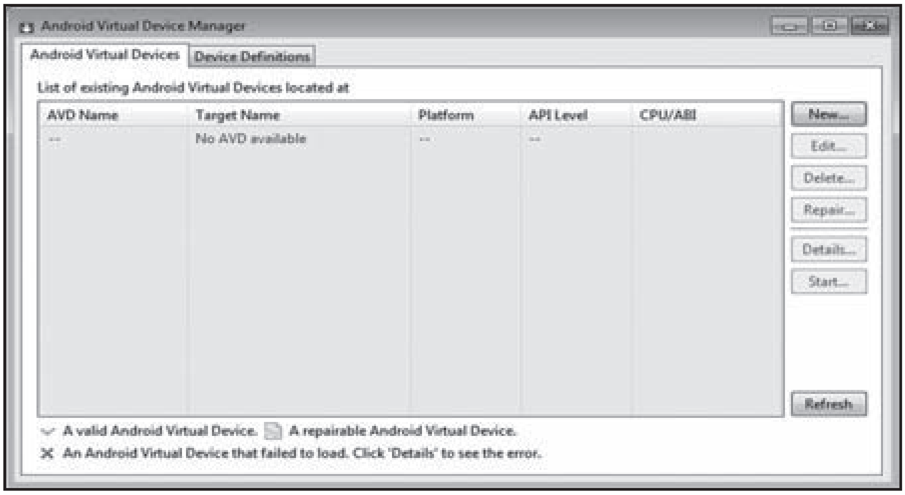
This happens because we have not yet set up an Android Virtual Device (AVD) to run the app. Let us now create an AVD, usually referred to as emulator.

## 2.3.2 Setting Up an Emulator

An emulator is typically a software that virtually reproduces the exact behavior of one computing machine on another. For example, an AVD (emulator) runs on Microsoft Windows, and reproduces an Android mobile device's behavior such as placing or receiving calls, sending SMS, and launching a mobile app.

To set up an emulator, click on *Yes* in the Android AVD error dialog window (see Fig. 2.12). It opens the AVD Manager dialog window as illustrated in Fig. 2.13.

Now, to create a new AVD, click on *New* option in the AVD Manager. This will open up the Create new AVD dialog window, as depicted in Fig. 2.14, with options to create and configure an AVD.



Android Virtual Device Manager

Android Virtual Devices | Device Definitions

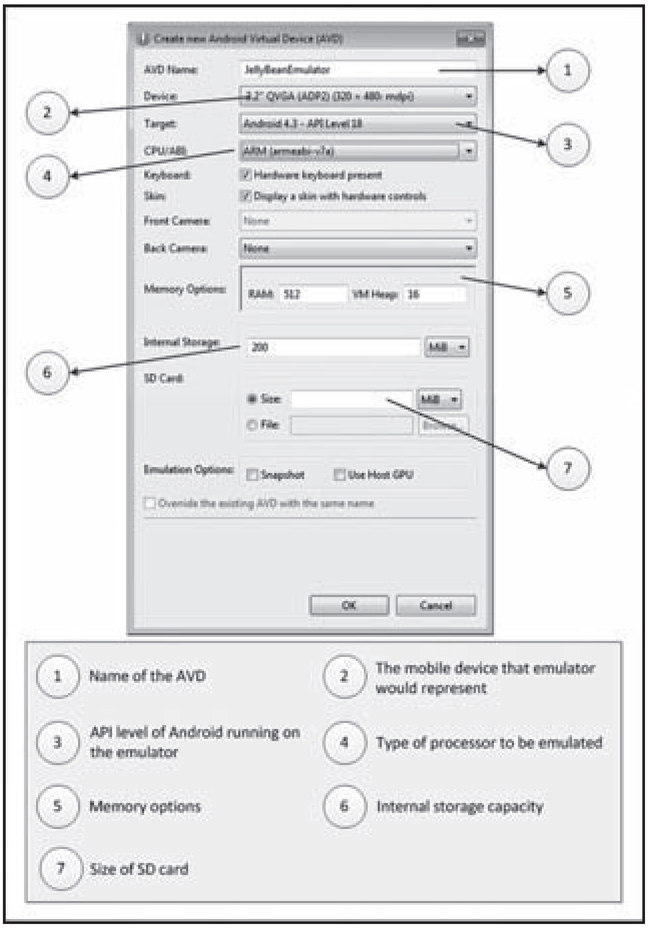
List of existing Android Virtual Devices located at

| AVD Name | Target Name | Platform | API Level | CPU/ABI |
|---|---|---|---|---|
| -- | No AVD available | -- | -- | |

✓ A valid Android Virtual Device. ▨ A repairable Android Virtual Device.

✕ An Android Virtual Device that failed to load. Click 'Details' to see the error.

**Figure 2.13** | Android Virtual Device (AVD) Manager.



Create new Android Virtual Device (AVD)

AVD Name: JellyBeanEmulator

Device: 3.2" QVGA (ADP2) (320 × 480: mdpi)

Target: Android 4.3 - API Level 18

CPU/ABI: ARM (armeabi-v7a)

Keyboard: ☑ Hardware keyboard present

Skin: ☑ Display a skin with hardware controls

Front Camera: None

Back Camera: None

Memory Options: RAM: 512 VM Heap: 16

Internal Storage: 200 MiB

SD Card: ● Size: MiB ○ File:

Emulation Options: ☐ Snapshot ☐ Use Host GPU

☐ Override the existing AVD with the same name

1 Name of the AVD

2 The mobile device that emulator would represent

3 API level of Android running on the emulator

4 Type of processor to be emulated

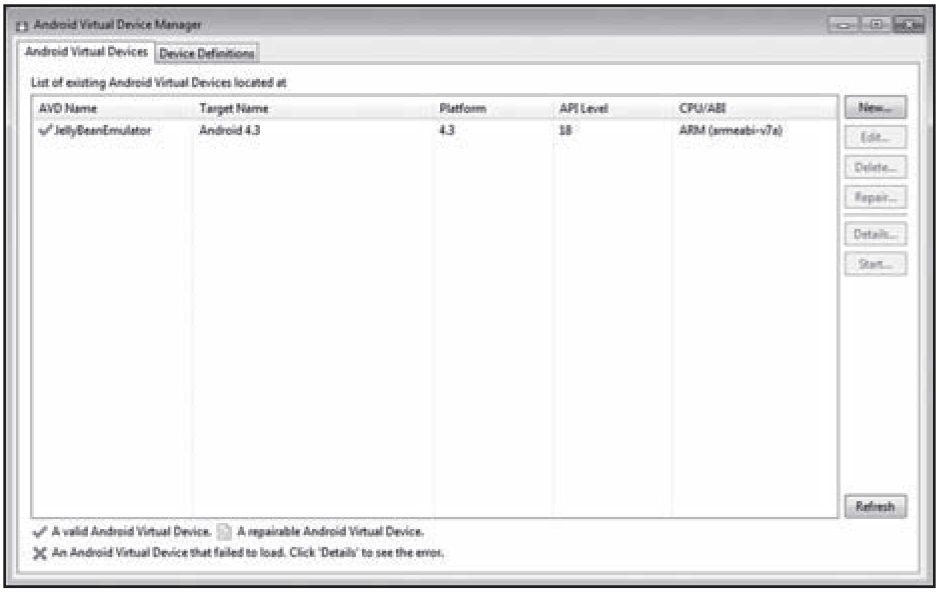
5 Memory options

6 Internal storage capacity

7 Size of SD card

**Figure 2.14** | Create new AVD dialog window.

Populate the fields as proposed in Fig. 2.14, and click on *OK*. The result is evident in Fig. 2.15, where we can locate the new AVD (Jelly Bean Emulator) in the AVD Manager.



List of existing Android Virtual Devices located at

| AVD Name | Target Name | Platform | API Level | CPU/ABI |
|---|---|---|---|---|
| ✓ JellyBeanEmulator | Android 4.3 | 4.3 | 18 | ARM (armeabi-v7a) |

✓ A valid Android Virtual Device. A repairable Android Virtual Device.
✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

**Figure 2.15** | AVD Manager with a new emulator.

We can now launch the newly created emulator by selecting it, and clicking *Start*. This will bring up a screen resembling a real mobile device. However, let us not use the AVD Manager dialog window to start the emulator.

Instead, close the AVD Manager dialog window, and run the HelloWorld app as shown in Fig. 2.11, that is, right click on the *FirstApp* project, and select *Run As → Android Application*. The emulator automatically gets started, as depicted in Fig. 2.16(a). Drag the lock icon to unlock screen, as depicted in Fig. 2.16(b).

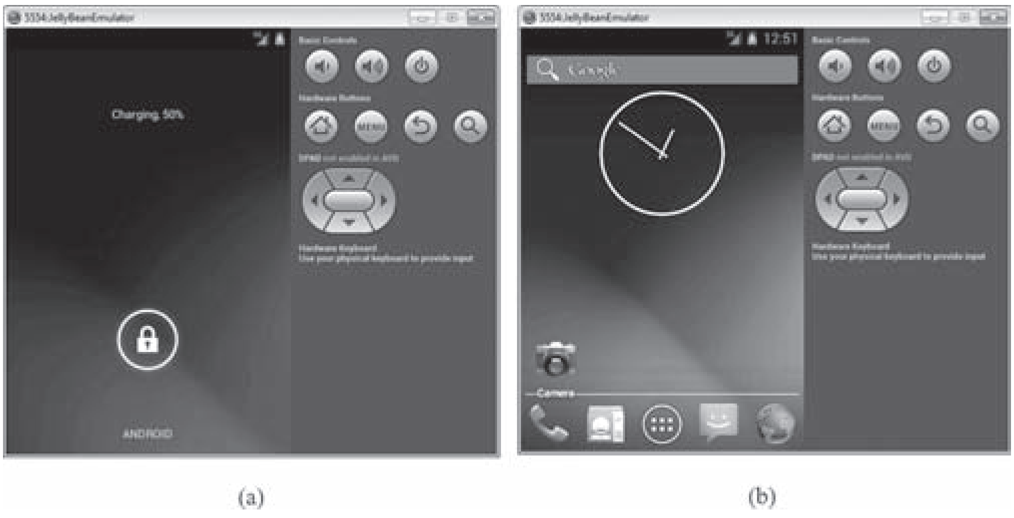


**Figure 2.16** | Emulator: (a) Screen locked and (b) screen unlocked.

In the meantime, behind the scenes, the HelloWorld app is automatically installed and started on the emulator, as depicted in Fig. 2.17.

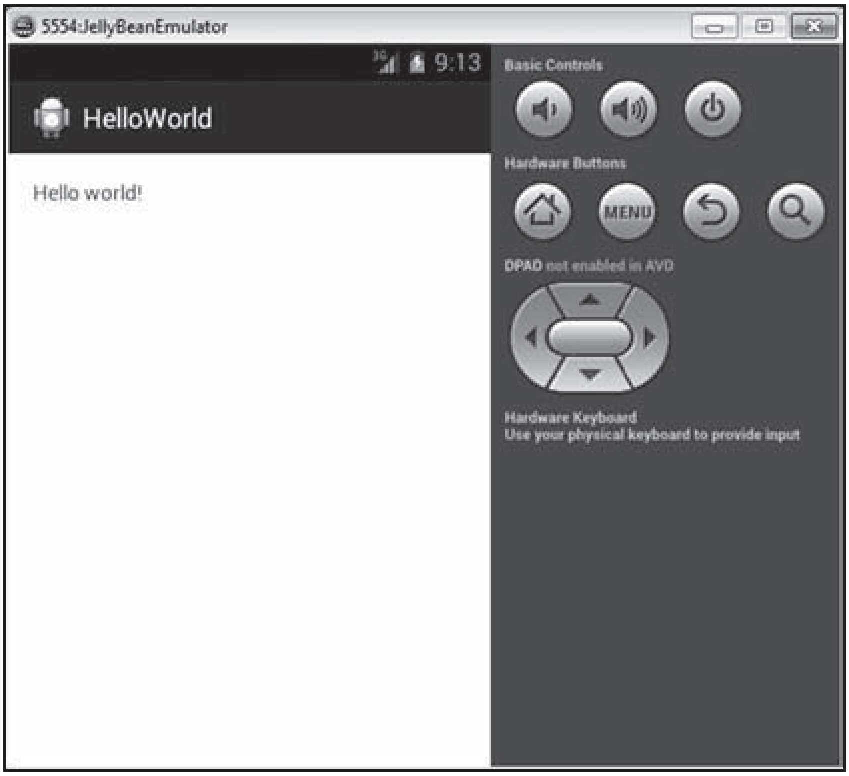


**Figure 2.17** | HelloWorld app running in the emulator.

The installation of HelloWorld app on the emulator can be validated by browsing through the installed apps on the emulator.

## 2.3.3 Behind the Scenes

The first app is automatically installed and successfully executed, even without writing a single line of code. So, what really happened behind the scenes!

When the FirstApp project is executed by selecting *Run As → Android Application*, a .apk (Android package) file gets created through a series of implicit steps, managed by the Eclipse IDE, as depicted in Fig. 2.18.

The Java source code gets compiled into ".class" files, which in turn get converted into a single ".dex" file. The dx tool, which is part of the Android SDK, performs this conversion. A .dex file is an executable file that runs inside the Dalvik Virtual Machine (DVM) when the app is launched. This file is packaged along with app resources and manifest file to yield a .apk file using Android Application Packaging Tool (aapt).

Only a signed app can run on a device/emulator to ensure its authenticity, therefore the .apk file is signed using jarsigner utility; zipalign utility also kicks in to optimize the .apk file, and makes it ready for installation.

During the runtime of an app, the Android platform initiates a process that in turn contains an instance of DVM to host the app. Each app that is running on the device/emulator has a dedicated process and a DVM in which it gets executed. This runtime architecture ensures the robustness of the platform by making sure that if one app behaves erratically, it does not affect other running apps.

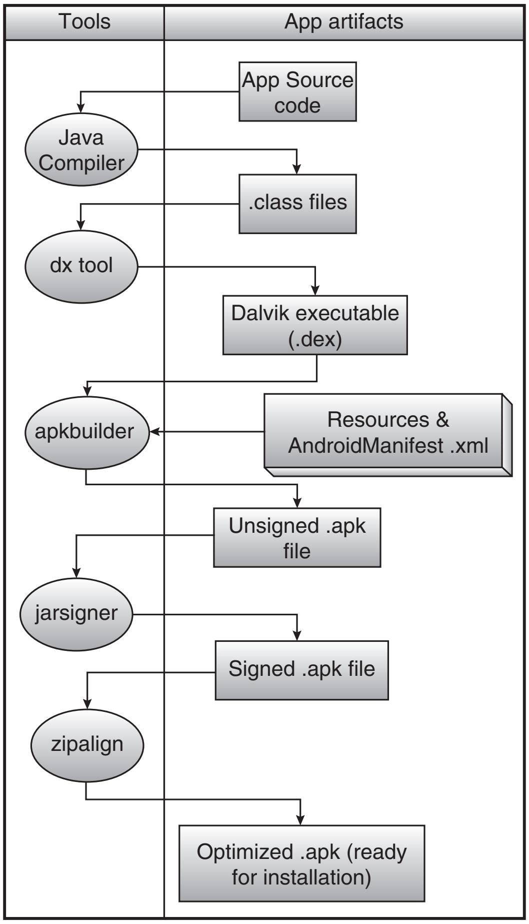


**Figure 2.18** | Behind the scenes.

## 2.4 TRAVERSING AN ANDROID APP PROJECT STRUCTURE

We have just created and launched the HelloWorld app, and also explored the nitty-gritty of what goes behind the scenes during its launch. Every Android app project is organized into a structure with varieties of artifacts such as the source code, image resources, string resources, menus, screen definitions, media files, and app configuration files. Let us now understand the HelloWorld app project structure, as depicted in Fig. 2.19:

1. FirstApp is the name of the project.
2. src is the source folder where all .java files are placed. Android mandates the creation of packages to manage the .java files. A package name must have at least two identifiers, for example, com.mad.
3. `MainActivity` is the default launcher activity of the app and its contents are represented in Snippet 2.1. It inherits from the `Activity` class and contains the `onCreate()` method. Within this method, the `setContentView()` method is called to inflate the layout of the screen (Line 9).

```
1  package com.mad.firstapp;
2  import android.os.Bundle;
```

```
import android.app.Activity;
import android.view.Menu;
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
  ...
}
```

**Snippet 2.1** | MainActivity.java.

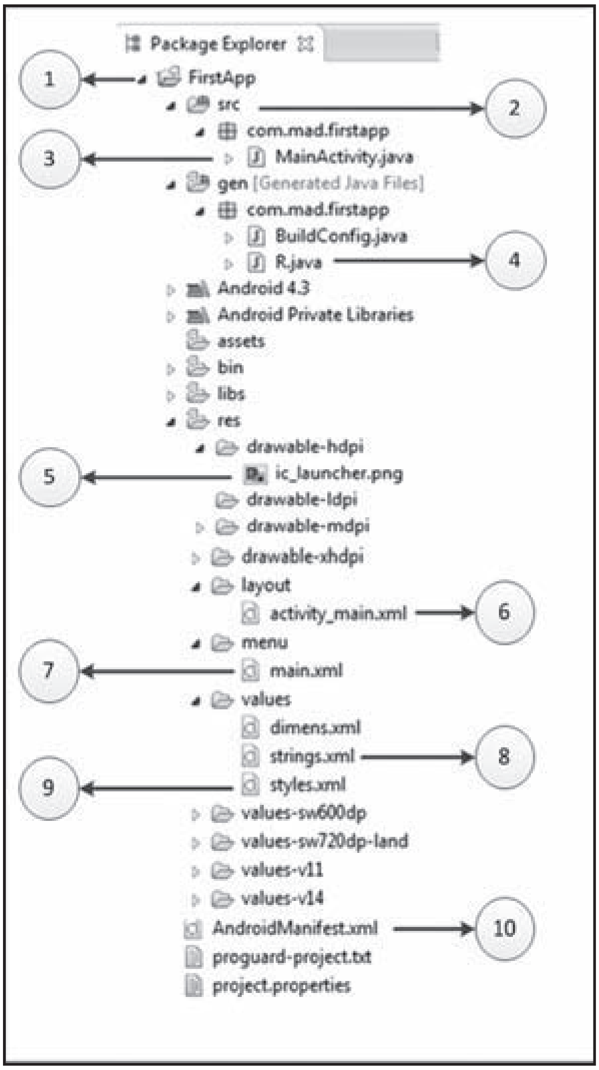


**Figure 2.19** | Android project structure.

4. R.java is an autogenerated file containing integer constants for all resources included in the project. The integer constants defined here will be used in other java files to refer to different resources using the format R.<folder_name>.<res_name>, as seen in Line 9 of Snippet 2.1.
5. ic_launcher.png is the default icon of the app. It is placed under the res\drawable-hdpi folder. The res folder contains all the app resources. The drawable subfolder in it is a container for all image resources. Several drawable subfolders (drawable-hdpi, drawable-ldpi, drawable-mdpi, and drawable-xhdpi) are included in a project to manage duplicate image resources of varying dimensions. This is done to ensure that image resources appear best across all possible screen densities.
6. activity_main.xml (Snippet 2.2) is a layout resource that defines the blueprint of various elements appearing on the screen of the app. This file can be edited to modify the look and feel using either a Graphical Layout editor (Fig. 2.20) or an XML editor.

```
<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”
    android:paddingBottom=“@dimen/activity_vertical_margin”
    android:paddingLeft=“@dimen/activity_horizontal_margin”
    android:paddingRight=“@dimen/activity_horizontal_margin”
    android:paddingTop=“@dimen/activity_vertical_margin”
    tools:context=“.MainActivity”>

         <TextView
             android:layout_width=“wrap_content”
             android:layout_height=“wrap_content”
             android:text=“@string/hello_world”/>

</RelativeLayout>
```

**Snippet 2.2** | activity_main.xml (layout file).

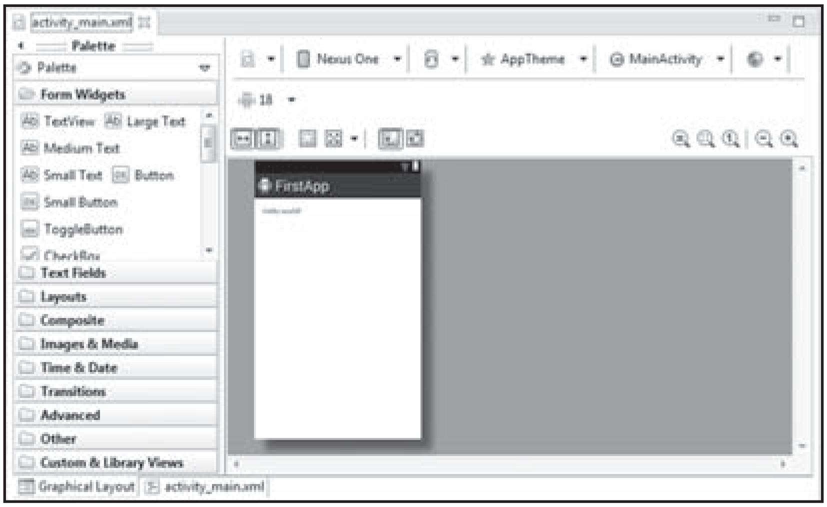
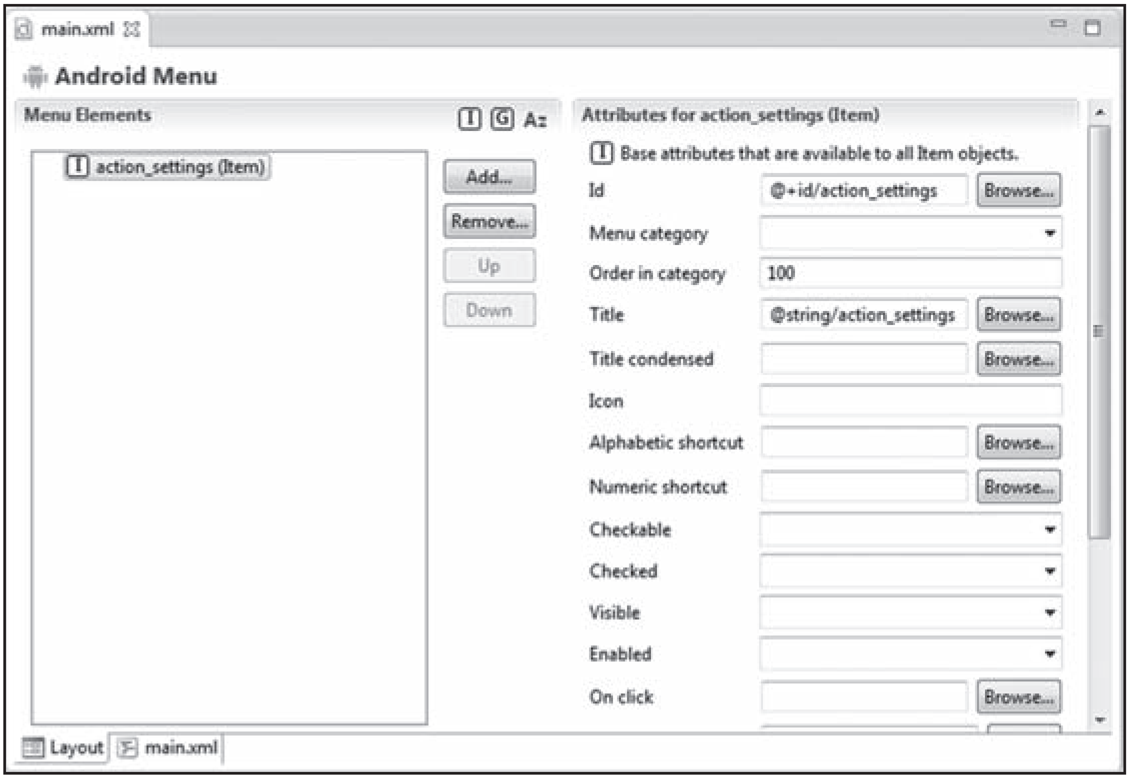


**Figure 2.20** | Graphical Layout editor.

**7.** main.xml (Snippet 2.3) under the menu folder contains the definition of a menu in the app. Menus can also be edited using either a Layout editor (Fig. 2.21) or an XML editor.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:showAsAction="never"
        android:title="@string/action_settings"/>
</menu>
```

**Snippet 2.3** | activity_main.xml (menu file).


main.xml

**Android Menu**

Menu Elements

action_settings (Item)

Attributes for action_settings (Item)

Base attributes that are available to all Item objects.

Id: @+id/action_settings
Menu category:
Order in category: 100
Title: @string/action_settings
Title condensed:
Icon:
Alphabetic shortcut:
Numeric shortcut:
Checkable:
Checked:
Visible:
Enabled:
On click:

**Figure 2.21** | Menu Layout editor.

**8**. strings.xml contains string resources in a key–value pair format as depicted in Snippet 2.4. These resources can be used across the app in various situations. In our app, the text displayed on the screen originates from the strings.xml file (Line 4). You may have observed that Line 14 of Snippet 2.2 refers to this string resource.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <string name="app_name">HelloWorld</string>
```

```
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
</resources>
```

**Snippet 2.4** | strings.xml.

9. styles.xml is used to define themes in an app. The themes defined can be used for the entire app or an individual UI element in it.
10. AndroidManifest.xml is the most important file in an Android app. It is a registry of several details such as list of the logical components, sdk requirements, and version of the app, as depicted in Snippet 2.5.

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

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="18"/>

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
              android:name="com.mad.firstapp.MainActivity"
              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>
```

**Snippet 2.5** | AndroidManifest.xml.

## 2.5 LOGICAL COMPONENTS OF AN ANDROID APP

We just created an Android app that comprises only an Activity. Activity is one of the key logical components of an Android app. Besides this, an app may include other equally important components such as

Services, Broadcast Receivers, and Content Providers. Having all or some of these logical components in an app is purely requirement driven. Let us have a brief overview of these four key logical components:

1. **Activity:** It is the basis of the UI of any Android app. The overall UI may consist of other visual elements, layout elements, resources, and many more things, besides the Activity that forms the foundation of UI.

   The terminology Activity may sometimes be misleading because it might be misunderstood as task/action that happens in an Android phone. Though it is true to some extent, at times there may be a task happening even without an active UI. The music being played in the background when the user is browsing through a gallery may be a typical example. In this scenario, the gallery, an active UI component(s), and the background music are two different components, the first being an Activity. The Activity and related UI components are discussed in detail in Chapter 4.
2. **Service**: Service is another key logical component of an Android app. It runs in the background, and does not have an UI. To interact with a Service, typically an UI (Activity) is associated with it.

   In a music player app, we need an UI to select or shuffle songs from the play list. However, a user need not be in the same Activity to continue listening to that song. He or she may navigate away from the music player UI, and start browsing the Internet while listening to the song. This is a sample scenario wherein Service (that plays music) is still running in the background as an independent component. The Service and related components are discussed in detail in Chapter 5.
3. **Broadcast Receiver:** While the mobile device is being used, there may be several announcements (broadcasts) which an app needs to capture and respond to. Announcements such as an incoming call, an incoming SMS, availability of Wi-Fi spot, or low battery are initiated by the system (Android platform). Some announcements are generated by an app itself to let the other apps know, such as about the completion of an event, for example, about a download that is complete and is available to be used.

   The component that captures and responds to such announcements is called as Broadcast Receiver. This component responds to only those announcements to which it is registered to listen to. More about the Broadcast Receiver is discussed in Chapter 5.
4. **Content Provider:** Persisting data pertaining to an app and making it accessible across apps is a crucial aspect of mobile app development. However, as a security feature, the Android platform prevents one app to access data of another app. To overcome this constraint, where an app needs to share the data with other apps, the data has to be exposed, which is done using Content Provider. For example, if an app needs to access the contacts list, the contacts app (another app) should provide a mechanism to share the contact data using the Content Provider. More about the Content Provider is discussed in Chapter 6.

In a typical Android app, these logical components are decoupled by virtue of the Android system design. This design enables a component to interact with others, not only within an app but also across apps (of course, with certain restrictions!). This is achieved using Intents – a message passing framework. More about Intents is discussed in Part II (Chapters 4 and 5).

Unlike conventional applications that manage their own life cycle, an Android app's life cycle is managed by Android runtime. It means that in case of a resource crunch, Android runtime may kill a low priority process, and eventually the app running in that process. Recall that each app is launched in an individual process that is initiated by the Android operating system. The priority of an app is equivalent to its highest priority component. Typically, the priority of an individual component is determined by its current state (foreground/background). For example, Activities interacting with the user are placed at the highest priority, and, therefore, the app (and the process hosting it) is unlikely to be terminated by runtime.

The developers have to use this design philosophy judiciously to make seamless, responsive, and harmonious apps. An Android app typically goes through the transitions of getting killed, and being resumed later; the onus lies with the developer to ensure seamlessness during these transitions. Developers should create responsive apps, by keeping track of individual components. For example, trying to update the UI of an Activity that is not being viewed currently may not only be futile but may also lead to unexpected results. Using right components enables developers to make their apps harmonious. Using Services for background tasks that do not interfere with UI, and using Broadcast Receivers to listen to system-wide broadcasts while the user is doing something else are some examples of how various app components can coexist.

## 2.6 ANDROID TOOL REPOSITORY

Because mobile apps run in resource-constrained environments, there is a dire need to ensure that they are optimally designed, responsive, and performant. The Android SDK comes bundled with a rich set of tools that not only help developers to cater to these pressing demands but also provide utilities that help them during the development of apps.

These tools are broadly classified as SDK tools and platform tools. SDK tools are Android platform version agnostic, and are present in the tools subfolder of the SDK. Platform tools are specific to Android platform version, and are present in platform-tools subfolder of the SDK. Most of the platform tools are used internally by Eclipse, and, therefore, rarely used explicitly by the developer. Some of the key SDK tools and platform tools are summarized in Tables 2.1 and 2.2, respectively.

**Table 2.1** | SDK tools

| Tool | Description |
|---|---|
| **DDMS** | Debugs apps and monitors their behavior in verbose mode |
| **Mksdcard** | Simulates an SD card in the emulator |
| **Monkey** | Stress tests apps by generating pseudo-random user events |
| **Proguard** | Obfuscates code so that the app cannot be reverse engineered |
| **sqlite3** | Manages SQLite databases in Android apps |
| **Traceview** | Profiles app performance |
| **Zipalign** | Optimizes .apk file |

**Table 2.2** | Platform tools

| Tool | Description |
|---|---|
| **adb** | Connects to emulators and devices through command prompt |
| **aapt** | Compiles app resources and generates R.java |
| **dx** | Converts .class files to a .dex file |

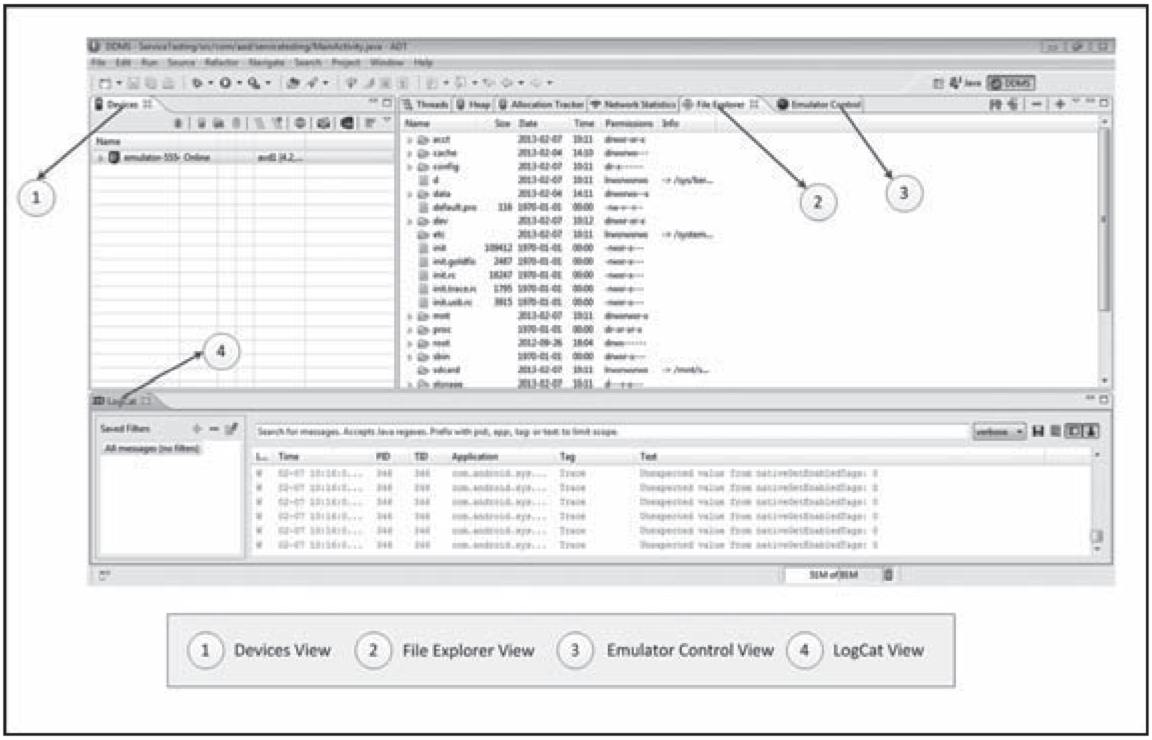
Debugging plays a crucial role during the development of apps to ensure that they are optimally designed, responsive, and performant. Failing which, the app becomes unpopular on app stores, social media and websites, leading to low retention, loss of revenue, and even tarnishing of the brand image. Developers use the Dalvik Debug Monitor Server (DDMS) and the Android Debug Bridge (adb) very frequently to debug their apps.

The DDMS is a powerful tool to debug apps and monitor their behavior in verbose mode. It is accessed as a perspective in Eclipse IDE. This perspective has four key views, as shown in Fig. 2.22. Each of these views serves different purposes when interacting with the emulator/device.

The Devices view displays the list of emulators and devices connected with the IDE. This view provides options such as capturing a snapshot of the emulator/device screen and getting data useful for memory profiling. It also displays the list processes running on any selected device.

The File Explorer view is used to access the file system of a selected emulator/device. It also facilitates file operations such as pulling out a file from the emulator/device to the computer and pushing in a file from the computer to the emulator/device.

The Emulator Control view provides controls to simulate events such as an incoming call or SMS. It also provides an interface to configure Global Positioning System (GPS) information of an emulator, which comes handy while debugging location-based apps.

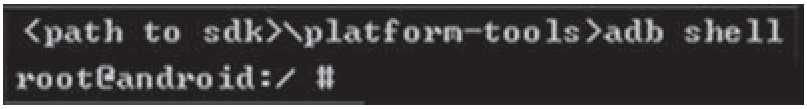


```
L... Time            PID  TID  Application          Tag    Text
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
W    02-07 10:16:0...  346  346  com.android.sys...   Trace  Unexpected value from nativeGetEnabledTags: 0
```

**Figure 2.22** | Views in DDMS perspective.

The LogCat view displays the logs made by processes running on a selected emulator/device. The logs displayed in the LogCat are categorized in various levels and contain details such as process id of the app, package name of the app, tag value used to identify the log, and the message of the log.

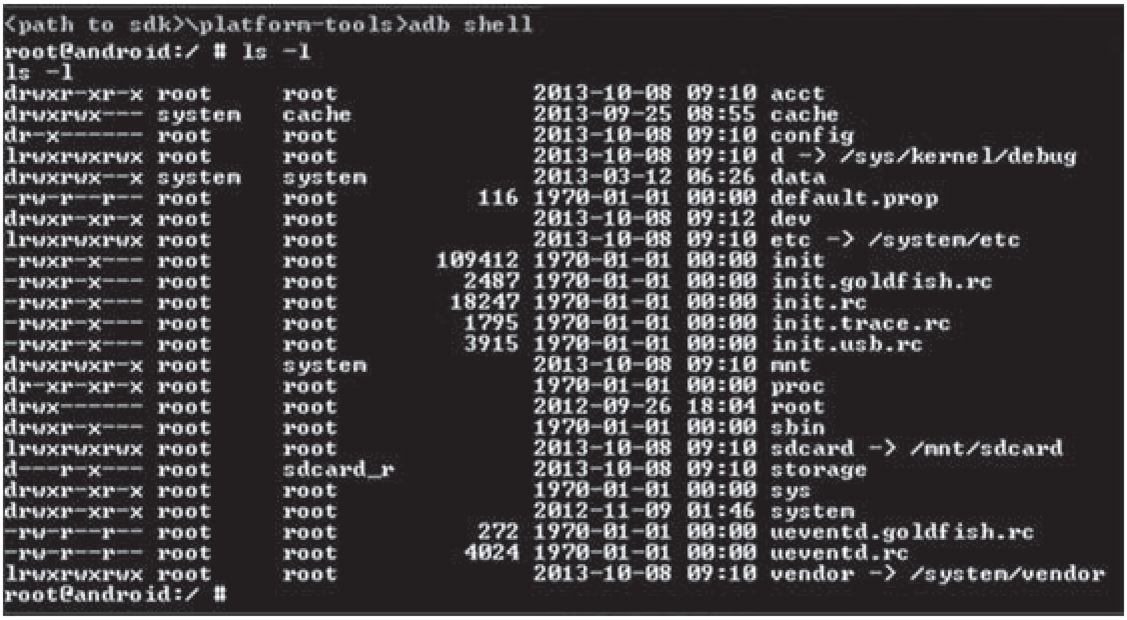
The Android Debug Bridge (adb) facilitates connection and communication with devices and emulators. The Eclipse IDE internally uses adb for this purpose. Developers can communicate with devices (or emulators) through adb from the command prompt using the "adb shell" command as shown in Fig. 2.23.



```
<path to sdk>\platform-tools>adb shell
root@android:/ #
```

**Figure 2.23** | Communicating with an emulator using adb.

Once connected to any device through adb, developers can execute commands that facilitate various operations such as installation of apps, copying of files, and reading of device logs. Figure 2.24 depicts the list of files present in the root directory of an emulator, by executing "ls –l" – a Linux command used to display a long list of files present in the file system.



```
<path to sdk>\platform-tools>adb shell
root@android:/ # ls -l
ls -l
drwxr-xr-x root     root              2013-10-08 09:10 acct
drwxrwx--- system   cache             2013-09-25 08:55 cache
dr-x------ root     root              2013-10-08 09:10 config
lrwxrwxrwx root     root              2013-10-08 09:10 d -> /sys/kernel/debug
drwxrwx--x system   system            2013-03-12 06:26 data
-rw-r--r-- root     root          116 1970-01-01 00:00 default.prop
drwxr-xr-x root     root              2013-10-08 09:12 dev
lrwxrwxrwx root     root              2013-10-08 09:10 etc -> /system/etc
-rwxr-x--- root     root       109412 1970-01-01 00:00 init
-rwxr-x--- root     root         2487 1970-01-01 00:00 init.goldfish.rc
-rwxr-x--- root     root        18247 1970-01-01 00:00 init.rc
-rwxr-x--- root     root         1795 1970-01-01 00:00 init.trace.rc
-rwxr-x--- root     root         3915 1970-01-01 00:00 init.usb.rc
drwxrwxr-x root     system            2013-10-08 09:10 mnt
dr-xr-xr-x root     root              1970-01-01 00:00 proc
drwx------ root     root              2012-09-26 18:04 root
drwxr-x--- root     root              1970-01-01 00:00 sbin
lrwxrwxrwx root     root              2013-10-08 09:10 sdcard -> /mnt/sdcard
d---r-x--- root     sdcard_r          2013-10-08 09:10 storage
drwxr-xr-x root     root              1970-01-01 00:00 sys
drwxr-xr-x root     root              2012-11-09 01:46 system
-rw-r--r-- root     root          272 1970-01-01 00:00 ueventd.goldfish.rc
-rw-r--r-- root     root         4024 1970-01-01 00:00 ueventd.rc
lrwxrwxrwx root     root              2013-10-08 09:10 vendor -> /system/vendor
root@android:/ #
```

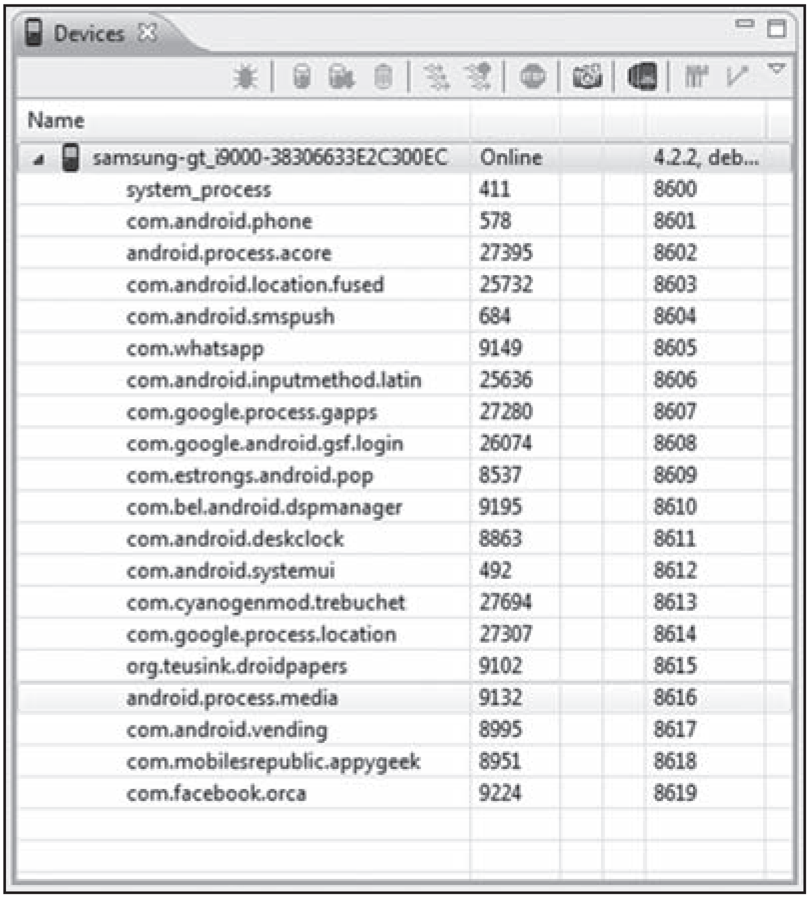
**Figure 2.24** | Executing commands on adb shell.

Other utilities such as proguard, zipalign, mksdcard, aapt, and dx tools are internally used by the Eclipse IDE during various stages of Android app development.

## 2.7 INSTALLING AND RUNNING APP DEVICES

So far, we have been executing and debugging our app on an emulator. In addition to doing this, it is always recommended to test apps on real device(s), before it moves to market. Testing on a real device puts an app in a real environment that may not be available in an emulator, such as availability of native hardware (such as camera or sensors) or behavior on a real network. This ensures that the app will always behave as desired.

The DDMS comes handy in running an app on a device. Before hooking the target device to the system that hosts the development environment, via Universal Serial Bus (USB) port, we need to ensure that USB debugging is enabled on the device, and necessary USB drivers are installed on the system. Once connected, the target device is visible in the Devices view of the DDMS, as depicted in Fig. 2.25.

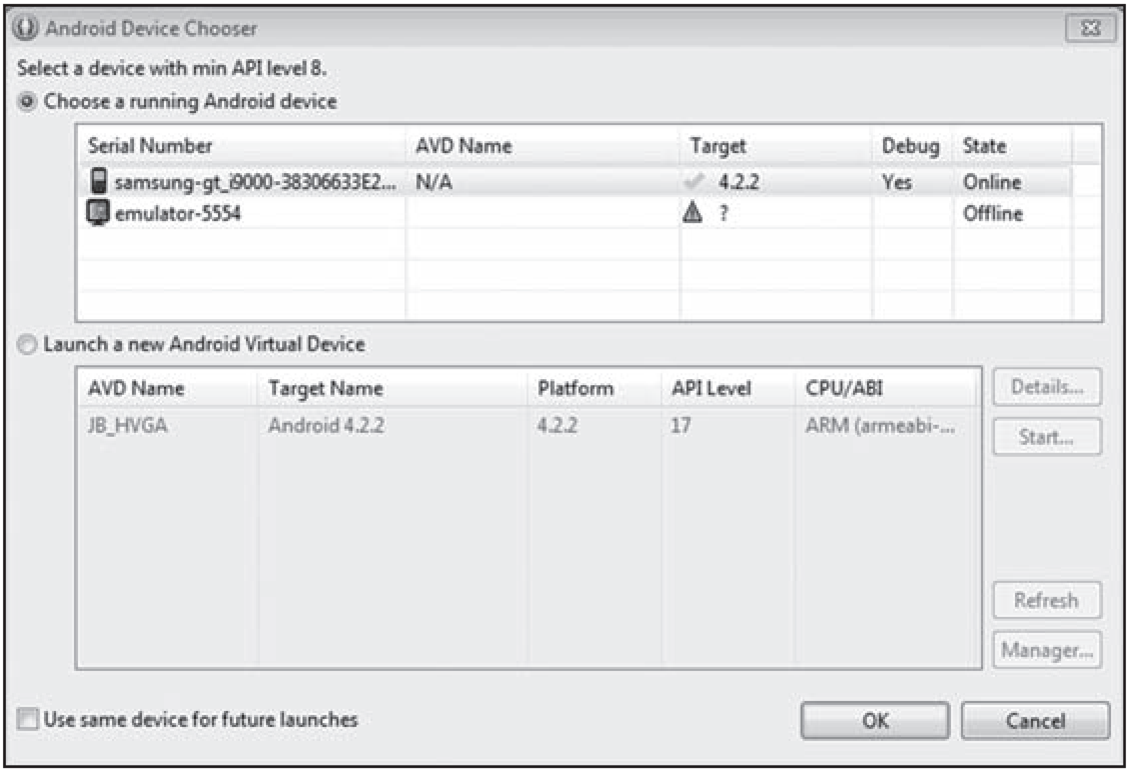


| Name | | | |
|---|---|---|---|
| samsung-gt_i9000-38306633E2C300EC | Online | | 4.2.2, deb... |
| system_process | 411 | | 8600 |
| com.android.phone | 578 | | 8601 |
| android.process.acore | 27395 | | 8602 |
| com.android.location.fused | 25732 | | 8603 |
| com.android.smspush | 684 | | 8604 |
| com.whatsapp | 9149 | | 8605 |
| com.android.inputmethod.latin | 25636 | | 8606 |
| com.google.process.gapps | 27280 | | 8607 |
| com.google.android.gsf.login | 26074 | | 8608 |
| com.estrongs.android.pop | 8537 | | 8609 |
| com.bel.android.dspmanager | 9195 | | 8610 |
| com.android.deskclock | 8863 | | 8611 |
| com.android.systemui | 492 | | 8612 |
| com.cyanogenmod.trebuchet | 27694 | | 8613 |
| com.google.process.location | 27307 | | 8614 |
| org.teusink.droidpapers | 9102 | | 8615 |
| android.process.media | 9132 | | 8616 |
| com.android.vending | 8995 | | 8617 |
| com.mobilesrepublic.appygeek | 8951 | | 8618 |
| com.facebook.orca | 9224 | | 8619 |

**Figure 2.25** | Target device in Dalvik Debug Monitor Server.

Now, to execute an app on this target device, select *Run As → Android Application*, as depicted in Fig. 2.11. This leads to Android Device Chooser dialog window where we can select the desired device to execute the app, as depicted in Fig. 2.26. The app gets installed and executed on the device automatically.

We have just learnt to set up the development environment and build our first Android app. In Chapter 3, we get a peek into the reference mobile app that we are going to build incrementally over the course of this book.


Android Device Chooser

Select a device with min API level 8.

● Choose a running Android device

| Serial Number | AVD Name | Target | Debug | State |
|---|---|---|---|---|
| samsung-gt_i9000-38306633E2... | N/A | ✓ 4.2.2 | Yes | Online |
| emulator-5554 | | ⚠ ? | | Offline |

○ Launch a new Android Virtual Device

| AVD Name | Target Name | Platform | API Level | CPU/ABI |
|---|---|---|---|---|
| JB_HVGA | Android 4.2.2 | 4.2.2 | 17 | ARM (armeabi-... |

☐ Use same device for future launches

**Figure 2.26** | Android Device Chooser dialog window.

## SUMMARY

This chapter has introduced the setup of an Android development environment on a Microsoft Windows machine. Over the course of this chapter, the first mobile app has been developed using the Android platform, and an Android emulator – used to launch and test the apps – has been set up.

The chapter has also elucidated the various steps that take place behind the scenes, which an app goes through before it is ready for installation, and has explored the project structure of a typical Android app.

It has presented a brief overview of various logical components of a typical Android app – Activity, Service, Broadcast Receiver, and Content Provider – and outlined the messaging framework that binds these components.

It has also unraveled the tool repository of Android, and discussed the tools commonly used by developers – DDMS and adb. Toward the end, it has demonstrated the installation and running of an app on an Android mobile device.

## REVIEW QUESTIONS

1. Define AVD. Differentiate between an emulator and a simulator.
2. Outline the steps that an app goes through before it is ready for installation.

3. Illustrate the purpose of res folder in the Android project structure.
4. Illustrate the purpose of AndroidManifest.xml.
5. Describe the various logical components of an Android app.
6. Outline the various views of the DDMS perspective and their purposes.

## FURTHER READINGS

1. Android development environment: http://developer.android.com/sdk/index.html
2. Android Studio: http://developer.android.com/sdk/installing/studio.html
3. Android tool repository: http://developer.android.com/tools/help/index.html
4. Running app on devices: http://developer.android.com/tools/device.html

3

# Learning with an Application – 3CheersCable

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Outline the reference app – 3CheersCable – used across the book.
- Describe the solution of the reference app.
- Identify mobile apps development challenges.
- Illustrate tenets of a winning app.

## 3.1 INTRODUCTION

This chapter outlines the reference application 3CheersCable, a mobile app that is implemented in this book. An overview of its solution, which will be built incrementally in subsequent chapters, is also described. Readers have to refer to the “Let’s Apply” sections in subsequent chapters to understand the implementation of various building blocks of this solution. Readers will also get introduced to the key mobile apps development challenges, and the tips to build a winning app.

## 3.2 3CHEERSCABLE APP

3CheersCable is a leading cable entertainment company, headquartered in Europe, with presence in more than 50 countries across continents. The enterprise has decided to bring its TV channels onto its consumers’ mobile devices to extend the entertainment experience outside the set-top box.

As phase I rollout, it has been decided to launch a native Android app – 3CheersCable app – that will cater to the following use cases:

1. Authenticate consumers before they use the app.
2. Subscribe a set of channels.
3. View program schedules for each channel.
4. Share favorite shows with friends and family.
5. Watch live TV on the go.
6. Contact customer-care executives.
7. Persist user credentials and preferences.

This reference app will enable us to explore and apply various nuances of the Android platform that are used to create typical Android apps. As we realize this app, we will first understand designing and implementing user interfaces, dealing with long-running tasks, and handling local and external data. We will also be able to leverage graphics, animations, multimedia, and sensors capabilities of the Android platform in this app. Finally, we will learn the nuances of publishing this app in an online app market.

One of the feasible solutions to design and implement the 3CheersCable app can be logically summed up into the following two layers:

1. The mobile app itself.
2. The backend enterprise application.

The left half of Fig. 3.1 depicts the logical architecture of 3CheersCable mobile app. The right half depicts the logical architecture of the backend enterprise application that caters to the remote data requirements of the mobile app such as authenticating user, fetching channel list, retrieving TV guide, and streaming live TV. The mobile app comprises of six logical building blocks:

1. App user interface is the topmost layer intended for user interaction.
2. App functionality is the “app logic” layer.
3. Native hardware access is the layer that enables mobile app to access native device hardware such as accelerometer and camera.
4. Native data access is the layer that accesses app data residing on the device.
5. Enterprise data access is the layer that enables remote data access.
6. Native data storage is the layer that physically stores the app data on the device.

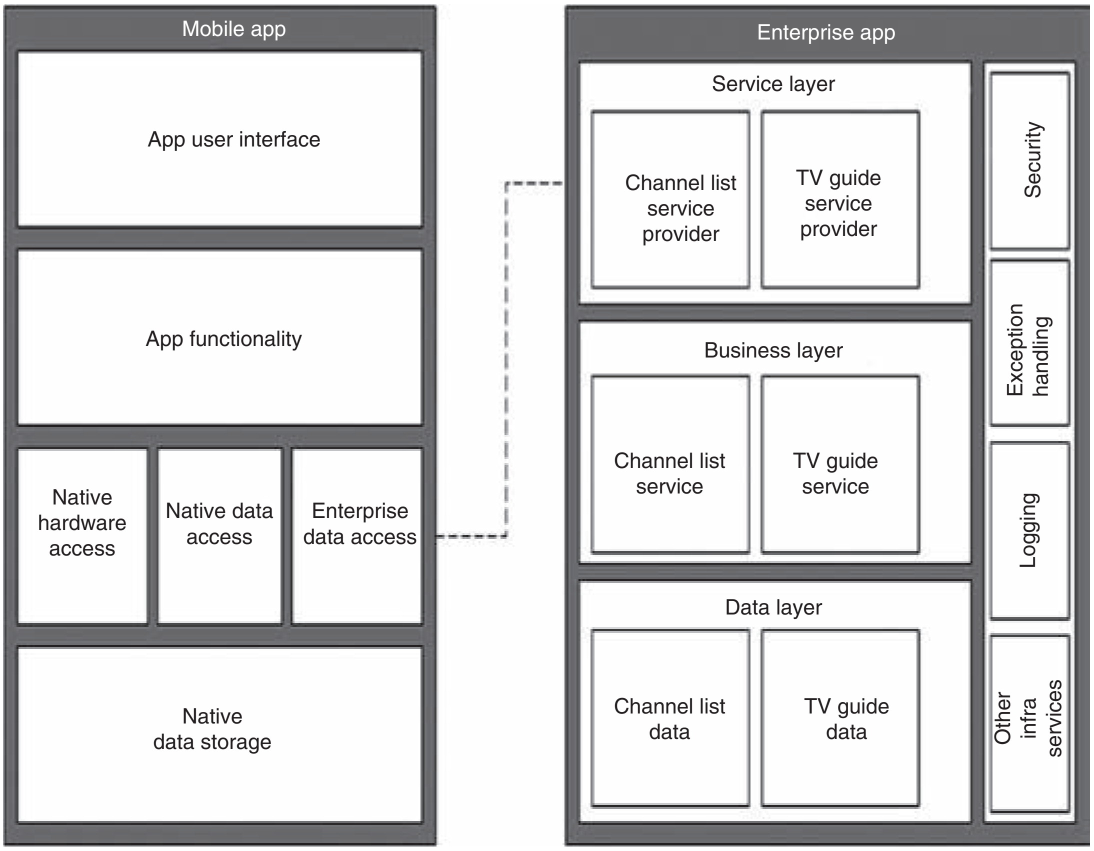


**Figure 3.1** | Logical architecture – both mobile and enterprise app.

The enterprise data access layer of the mobile app sources the required remote data from the service layer of the enterprise application, as depicted in Fig. 3.1 (represented with a dotted line). The data layer of enterprise application hosts the required data entities such as channel list and TV guide–related data. The business layer of the enterprise application actually hosts the application logic to provide channel list and TV guide, which is further exposed by service layer to be consumed by external systems (such as a mobile app in this case).

The nitty-gritty of implementing enterprise applications is out of the scope of this book. The core focus throughout the book is only on mobile app development – the 3CheersCable app. However, this book will deal with the interaction of mobile apps with an external system for remote data needs. The complete source code of enterprise application is available for reference, along with this book. Readers who are interested in learning and practicing to build enterprise applications may refer to the book *Raising Enterprise Applications*, published by Wiley India (2010), which attempts to shed light on building enterprise applications, while confirming to proven software engineering practices.

After outlining problem and solution descriptions of the reference app, let us now try to appreciate various challenges, in general, that may arise during the development of mobile apps.

## 3.3 MOBILE APP DEVELOPMENT CHALLENGES

Developing mobile apps is an interesting, yet challenging, task. It is entirely different from developing an enterprise or a desktop-based application. Challenges are multifold, and may be broadly categorized into the following four categories:

1. **User experience–specific challenges:** User experience–specific challenges primarily comprise the following four areas:
   - *Short span of user focus*: Mobile apps get very short span of time to make an impression on the user. Thus, it is important to structure the app in such a manner that the most important things are the ones that the user gets to see, as soon as the app is launched. The learning curve of using an app must not be steep. The more an app engages the user, the longer and better it would be used.
   - *Native look and feel*: Mobile users get used to the native look and feel of the platform in the long run of its usage. An app developer needs to make sure that the look and feel of the app should be consistent with that of the platform.
   - *Responsiveness*: Mobile devices provide the user the liberty to do things on the go. Hence, the user expects mobile apps to be fast and responsive. The user should be able to perform desired operations in the app within a short span of time. Complex operations should be divided into simple subtasks that gets completed in short period of time. An app that keeps a user waiting would never be appreciated. Thus, it is the developer's responsibility to structure the app and optimize the code for simple and fast operations.
   - *Personalization*: The app can be customized to suit individual users. For example, the device location may be used to provide information that is more relevant to the user location. It may also be a good idea to store username and password on first login, and keep the user logged in for subsequent access. Storing user preferences and using them to personalize the app would also delight the user.
2. **App functionality–specific challenges:** App functionality–specific challenges typically include the following two categories:
   - *Battery life and computational power*: The computational power of an average mobile device is less than a modern personal computer or laptop. Battery life is yet another hardware constraint. Smartphones and tablets tend to consume more battery owing to touchscreens, graphics, sensors, and multiple apps running together. The developer has to ascertain that app functionality does not demand excessive computational power or battery life.
   - *Network access*: The advent of 3G and 4G mobile communication standards has marked a phenomenal rise in data transfer speed limits. However, network reliability is a big problem. Apps should not make the user wait for too long while data is being transferred across the network. Additionally, they may also have an offline mode in case of unavailability of network.
3. **Data-specific challenges:** There are mainly two types of data-specific challenges:
   - *Security*: The ubiquity of mobile devices provides numerous expediencies to the consumer, but also makes him or her vulnerable to security threats. Mobile devices store valuable user information, which if compromised may result in varying degree of losses to the user. Such loss may vary from an embarrassing social networking update to transmission of confidential e-mails to unauthorized recipients or even unauthorized monetary transactions. It is the developers' responsibility to build apps that prevent unauthorized access to any information.
   - *Privacy*: User privacy should never be questioned, challenged, or subverted, rather it needs to be respected. Not every user would be comfortable sharing information such as location, contacts, browsing history, and call details. An app must inform the user about activities that it performs in

the background. It is always a good idea to take user's permission before using any of his or her personal information, and such usage should be under user agreement. It is important to develop an app that is a friend of the user rather than a pesky neighbor.

4. **Platform-specific challenges:** Platform-specific challenges primarily include the following:
   - *Platform fragmentation*: The mobile device market is highly competitive with a variety of operating systems and platforms powering these devices. The diverse offerings come as a blessing to the consumers who get a lot of options to choose from. But this also poses a huge challenge for app developers.
   - *Screen types*: Mobile devices come in a variety of screen sizes and types. While Original Equipment Manufacturers (OEMs) strive continuously to provide choices to the consumer, developers have to make sure that their app works well on all targeted devices. Mobile devices in different price segments sport different screen sizes, resolutions, pixel density, and display technologies. The onus lies on the developer to ensure that the app works consistently across different devices. Besides this, developers also face the challenge of adapting the app to change in device orientation. Some developers even go a step further, and exploit the change in orientation to provide additional functionalities.
   - *Input mechanisms*: Mobile devices in today's market come with different input mechanisms such as touch screens, QWERTY keyboard, trackballs and touchpads, and sensors to facilitate user input. Mobile devices could use one or more of these mechanisms to capture user input. The app has to be designed in such a manner that it can be used on any targeted device using any available mechanism for user input.

Beyond the app development challenges, there are other concerns that the mobile app ecosystem brings in at large: the way in which an app is distributed, the place from where the app gets installed, the environment in which the app is accessed, and the device on which it is used are some of the key ones.

## 3.4 TENETS OF A WINNING APP

Though the challenges for developing mobile apps are multifold, to create a winning app we have to design it in such a way that it always enjoys advocacy from end users. This may be achieved by following five tenets, as proposed in Fig. 3.2. The app should be intimate, interactive, immediate, intelligent, and insightful.

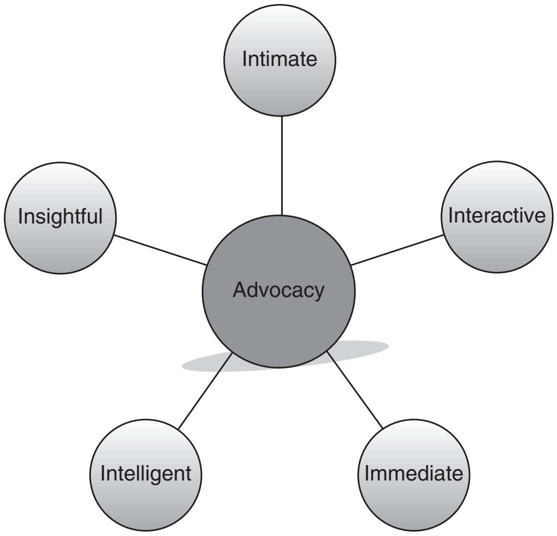


**Figure 3.2** | Five tenets of a winning app.

Intimacy element ensures the belongingness of user with the app. Interactivity element ensures cooperation of app with the user, and delights the user on each interaction. Immediateness element ensures instant gratification to the user by providing direct access to information required at that moment in time. Intelligence element ensures pertinent inferences for the user, by way of reasoning. Insightful element ensures appropriate suggestions for the user by exhibiting relevant insights and perspectives in the app.

This brings us to the end of Part I of this book. Now, we will take a deep dive into understanding the building blocks of a typical Android app using 3CheersCable app as the reference application.

## SUMMARY

This chapter has introduced the reference app – 3CheersCable – that will run through this book to illustrate mobile app development. It has highlighted the design importance of an app, and how the app gets bundled and interoperates with enterprise applications.

The chapter has also provided an overview of various challenges that a developer may have to overcome while developing mobile apps, viz. user experience–specific, app functionality–specific, data-specific, and platform-specific challenges. Finally, the chapter concludes with presenting the five tenets that are proposed for building winning mobile apps.

## REVIEW QUESTIONS

1. Elucidate user experience–specific, app functionality–specific, data-specific, and platform-specific challenges while developing a mobile app.
2. Illustrate the tenets that would help to design a winning mobile app.

## FURTHER READING

1. Anubhav Pradhan, Satheesha B. Nanjappa, Senthil K. Nallasamy, and Veerakumar Esakimuthu (2010), *Raising Enterprise Application: A Software Engineering Perspective*, Wiley India, New Delhi.

# Part II
# Building Blocks

# 4

# App User Interface

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Create basic user interface of Android app.
- Illustrate Activity life cycle.
- Devise interaction among Activities using Intents.
- Compose tablet UI using Fragments and action bar.

## 4.1 INTRODUCTION

“First impression is the last impression,” very aptly fits to the user experience of an app. User experience focuses on the overall experience users have while interacting with the app to pursue their task at hand. User experience is not limited to the user interface of an app but also includes aspects such as usability, brand value, trust worthiness, usefulness, and accessibility.

This chapter primarily focuses on user interface aspects of an app that contribute to a rich user experience. It starts with the core building block of Android UI – Activity – and further delves into layouts and other UI elements that are required for designing the screen makeup. It also drills down into the nitty-gritty of handling user interface across Android devices with varying form factors.

## 4.2 ACTIVITY

An Android app user interface (UI) typically comprises components such as screen layouts, UI controls, art work, and events that may occur during user interaction. These components can be broadly categorized into programming and nonprogramming components. The advantage with this clear distinction is reusability of components. It also brings in separation of concern that makes sure that nonprogramming components can be designed without bothering much about nuances of the app logic.

Programming components of UI mean app components that have to be programmed using Java such as an Activity and event handling. The nonprogramming components of UI typically include resources such as image files, icon files, XML files, screen layout files, and other media elements. Android app development framework not only strives to bring in this distinction but also provides Application Programming Interfaces (APIs) that enable access of nonprogramming components inside programming components.

We already had a glimpse of Activity in Chapter 2 in the HelloWorld app. Typically, an app may contain one or more Activities based on the UI requirements. There might be a possibility that an app may not have any Activity, in case UI is not needed. You may recall from Chapter 2 that an app’s Activity is created by extending the `Activity` class. The Activity of an app, being a central point to the app UI, hosts the event-handling logic and runtime interaction with the nonprogramming components.

A user typically begins interacting with an app by starting an Activity. Over the course of interaction, the state of the Activity may undergo changes such as being visible, partially visible, or hidden. Let us now explore these states, and the associated life-cycle methods that get fired behind the scenes to manage the state changes.

### 4.2.1 Activity States

When the user taps on an app icon, the designated Activity gets launched into the foreground where it has the user focus. Pressing the Back key destroys the Activity in focus (current Activity) and displays the previous Activity to the user. Pressing the Home key moves the Activity in focus to the background, navigating to the Home screen.

In a nut shell, from the end user’s perspective, the Activity is either visible or invisible at a given point in time. Though, if we put on a developer’s hat and try to analyze the behavior of an Activity, we may ponder upon the following questions: How does an Activity get started? What happens to the Activity when it is invisible? When the Activity is invisible, is it still alive? Does Activity retain its state when it is visible again? To answer these, let us begin with understanding the states of an Activity.

An Activity, at any given point in time, can be in one of the following four states:

1. **Active:** An Activity in this state means it is active and running. It is visible to the user, and the user is able to interact with it. Android runtime treats the Activity in this state with highest priority and never tries to kill it.

2. **Paused:** An Activity in this state means that the user can still see the Activity in the background such as behind a transparent window or a dialog box, but cannot interact with it lest he or she is done with the current view. It is still alive and retains its state information. Android runtime usually does not kill an Activity in this state, but may do so in an extreme case of resource crunch.
3. **Stopped:** An Activity in this state means that it is invisible but not yet destroyed. Scenarios such as a new Activity started on top of the current one or user hitting the Home key may bring the Activity to the stopped state. It still retains its state information. Android runtime may kill such an Activity in case of resource crunch.
4. **Destroyed:** An Activity in this state means that it is destroyed (eligible to go out of memory). This may happen when a user hits Back key or Android runtime decides to reclaim the memory allocated to an Activity that is in the paused or stopped state. In this case, the Activity does not retain its state.

Android runtime manages Activities in a task stack. The Activity in active state always sits on the top of the stack. As new Activities get started further, they will be pushed on top of the last active Activity. And, as this happens, the last active Activity will transition to the paused/stopped state based on the scenarios explained above. When the Activity is destroyed, it is popped from the task stack. If all Activities in an app are destroyed, the stack is empty.

An Activity does not have the control over managing its own state. It just goes through state transitions either due to user interaction or due to system-generated events. This calls for the responsibility on part of a developer to manage the app logic during state transitions, for ensuring the robust behavior of an app. Let us now explore the callback methods, also referred to as life-cycle methods, provided by Android app framework, which help developers in achieving this task.

### 4.2.2 Life-Cycle Methods

As we have noticed in the HelloWorld app in Chapter 2, the `MainActivity` has an overridden Activity life-cycle method – `onCreate()`. There are six such key life-cycle methods of an Activity: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`. These life-cycle methods get executed throughout the life cycle of an Activity based on its state transitions, as depicted in Fig. 4.1.

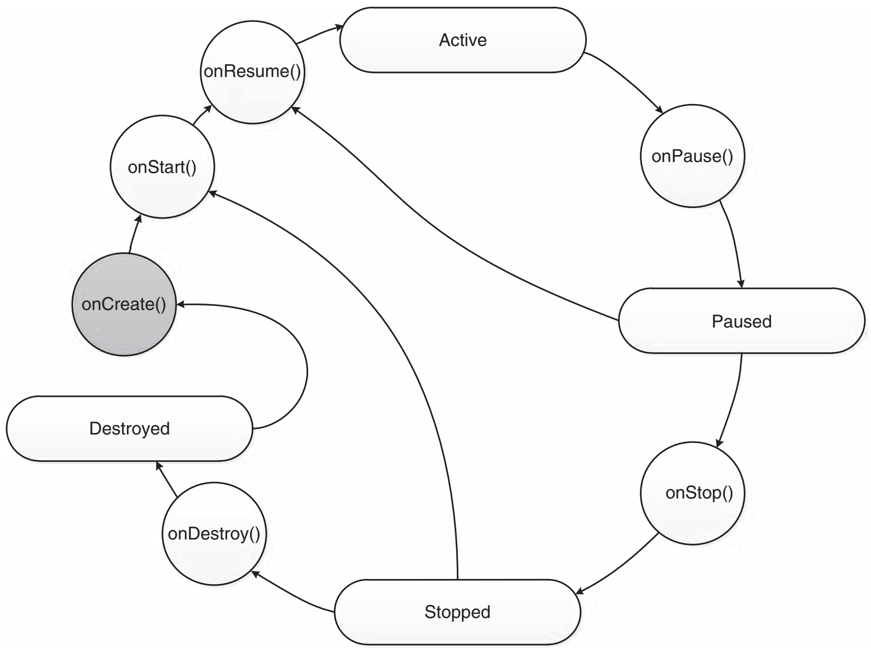


**Figure 4.1** | Activity life-cycle methods.

Let us now explore these life-cycle methods using Snippet 4.1. All six life-cycle methods are overridden, and in each method a log statement – `Log.i (String tagname, String message)` – has been placed to reflect the life-cycle method and state, as transition happens. The log messages are displayed in the LogCat utility in Dalvik Debug Monitor Server (DDMS).

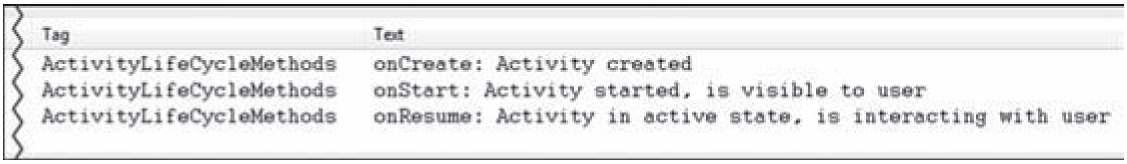
```
public class MainActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   Log.i("ActivityLifeCycleMethods", "onCreate: Activity
         created");
}
@Override
protected void onStart() {
   super.onStart();
   Log.i("ActivityLifeCycleMethods", "onStart: Activity
          started, is visible to user");
}
@Override
protected void onResume() {
   super.onResume();
   Log.i("ActivityLifeCycleMethods", "onResume: Activity in
         active state, is interacting with user");
}
@Override
protected void onPause() {
   super.onPause();
   Log.i("ActivityLifeCycleMethods", "onPause: Activity in
         paused state, is partially visible to user");
}
@Override
protected void onStop() {
   super.onStop();
   Log.i("ActivityLifeCycleMethods", "onStop: Activity in
         stopped state, is not visible to user ");
}
@Override
protected void onDestroy() {
   super.onDestroy();
   Log.i("ActivityLifeCycleMethods", "onDestroy: Activity
         destroyed");
}

}
```

**Snippet 4.1** | Activity life-cycle methods.

When we run Snippet 4.1 and observe the LogCat, we see that three life-cycle methods (`onCreate`, `onStart`, and `onResume`) get executed as soon as the app is launched, as shown in Fig. 4.2.



```
Tag                          Text
ActivityLifeCycleMethods     onCreate: Activity created
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
```

**Figure 4.2** | LogCat view depicting methods execution on app launch.

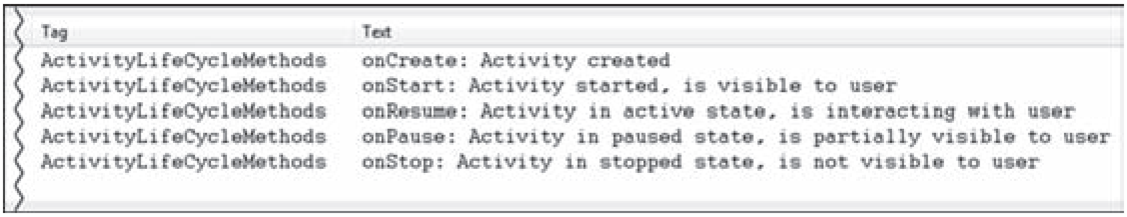
The first method to get executed is `onCreate()`. This is executed only once during the lifetime of an Activity, that is, at the beginning, when the Activity is launched. If we have any instance variables in the Activity, the initialization of those variables can be done in this method. After `onCreate()`, the `onStart()` method is executed.

During the execution of `onStart()`, the Activity is not yet rendered on screen but is about to become visible to the user. In this method, we can perform any operation related to UI components. When the Activity finally gets rendered on the screen, `onResume()` is invoked. At this point, the Activity is in the *active* state and is interacting with the user.

If the Activity loses its focus and is only partially visible to the user, it enters the *paused* state. During this transition, the `onPause()` method is invoked. In the `onPause()` method, we may commit database transactions or perform light-weight processing before the Activity goes to the background.

When the Activity comes back to focus from the paused state, `onResume()` is invoked. This transition between paused and active states is a frequent phenomenon, thus `onResume()` is called more often than any other life-cycle method of an Activity. Because of this reason, it is advisable to keep the implementation of `onResume()` as light as possible.

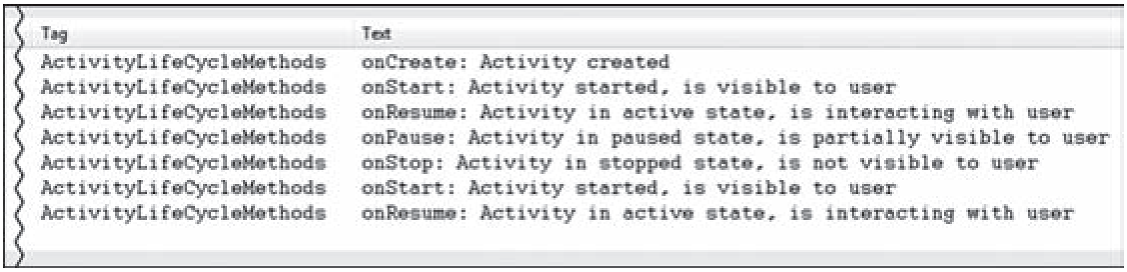
From the active state, if we hit the Home key, the Activity goes to the background and the Home screen of the device is made visible. During this event, the Activity enters the *stopped* state. Both `onPause()` and `onStop()` methods are executed, as shown in Fig. 4.3.



```
Tag                          Text
ActivityLifeCycleMethods     onCreate: Activity created
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
ActivityLifeCycleMethods     onPause: Activity in paused state, is partially visible to user
ActivityLifeCycleMethods     onStop: Activity in stopped state, is not visible to user
```

**Figure 4.3** | LogCat output when the Home key is hit.

When we reopen the app,[1] we observe that the Activity is now transitioning from the stopped state to the active state. This is characterized by invocation of only `onStart()` and `onResume()` methods (see Fig. 4.4), validating that `onCreate()` gets invoked only once in a lifetime of the Activity.

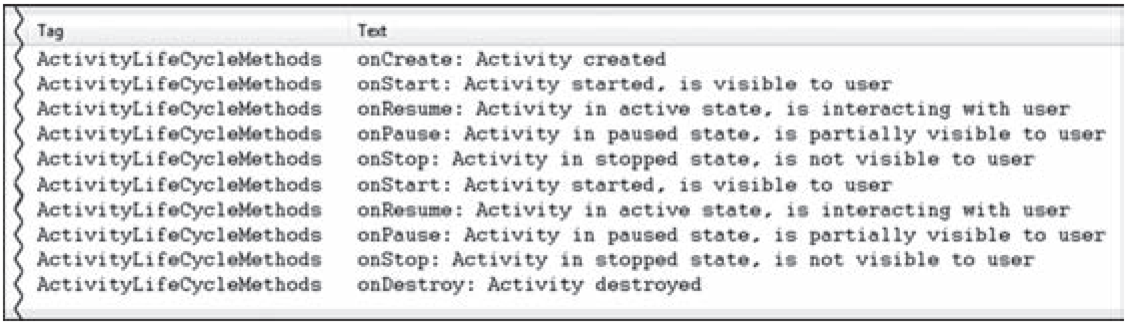


```
Tag                          Text
ActivityLifeCycleMethods     onCreate: Activity created
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
ActivityLifeCycleMethods     onPause: Activity in paused state, is partially visible to user
ActivityLifeCycleMethods     onStop: Activity in stopped state, is not visible to user
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
```

**Figure 4.4** | LogCat output when the app is reopened.

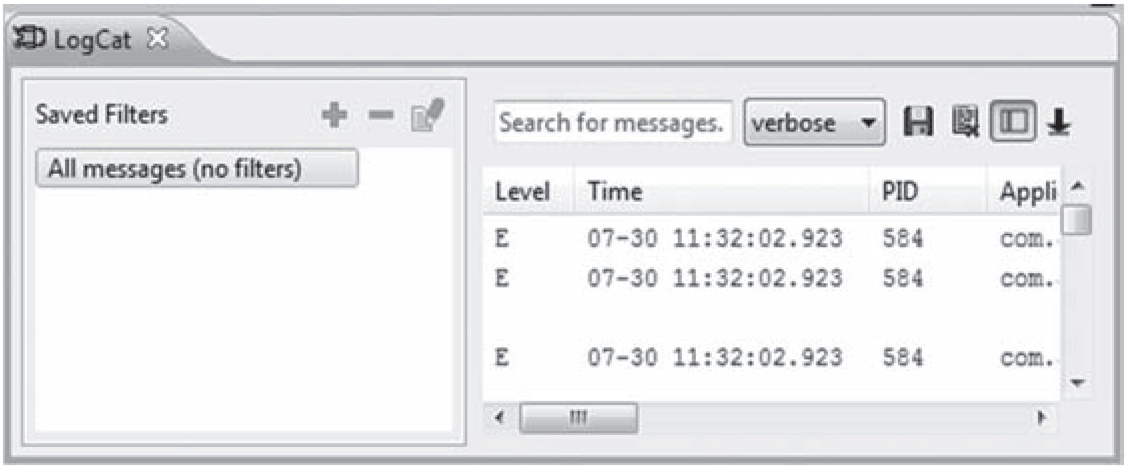
[1] Either by tapping the app icon or by long pressing of Home key and selecting the running app

To destroy the Activity on the screen, we can hit the Back key. This moves the Activity into the *destroyed* state. During this event, `onPause()`, `onStop()`, and `onDestroy()` methods are invoked, as shown in Fig. 4.5. You may recall that, in the destroyed state, the Android runtime marks the Activity to become eligible to go out of memory.



```
Tag                          Text
ActivityLifeCycleMethods     onCreate: Activity created
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
ActivityLifeCycleMethods     onPause: Activity in paused state, is partially visible to user
ActivityLifeCycleMethods     onStop: Activity in stopped state, is not visible to user
ActivityLifeCycleMethods     onStart: Activity started, is visible to user
ActivityLifeCycleMethods     onResume: Activity in active state, is interacting with user
ActivityLifeCycleMethods     onPause: Activity in paused state, is partially visible to user
ActivityLifeCycleMethods     onStop: Activity in stopped state, is not visible to user
ActivityLifeCycleMethods     onDestroy: Activity destroyed
```

**Figure 4.5** | LogCat output when Back key is hit.



```
Level   Time                  PID    Appli
E       07-30 11:32:02.923    584    com.
E       07-30 11:32:02.923    584    com.

E       07-30 11:32:02.923    584    com.
```

**Figure 4.6** | Saved Filters view of LogCat.

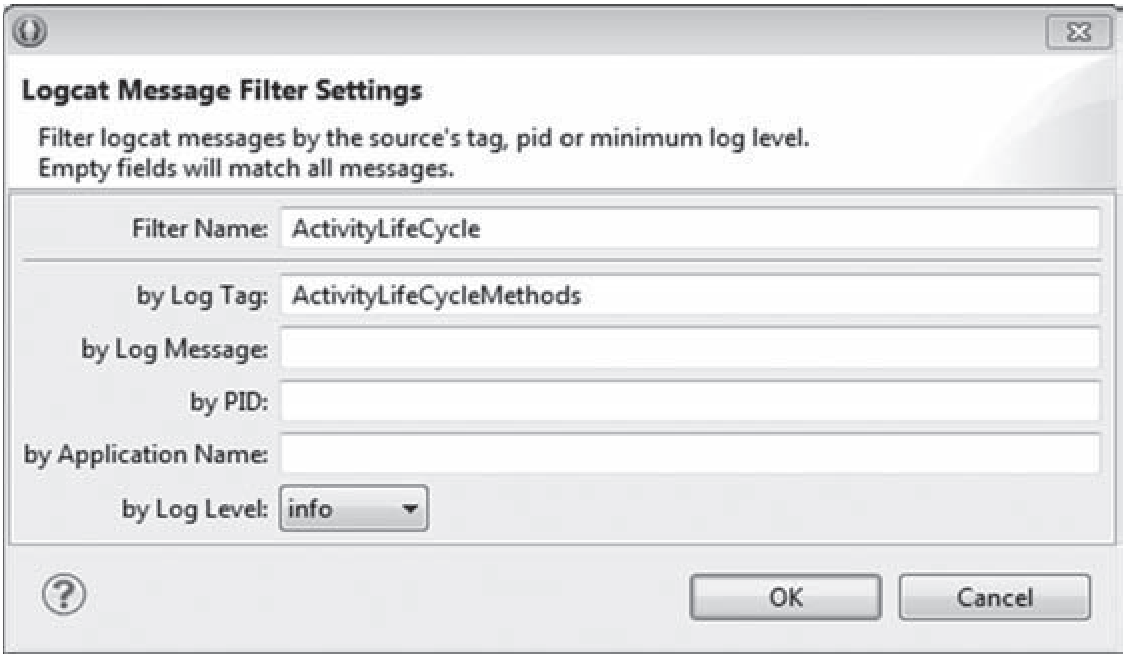
To summarize, the Activity is alive from `onCreate()` to `onStop()` methods and depending upon how many times the Activity goes to the background or loses focus and comes back to the foreground, `onStart()`, `onResume()`, `onPause()`, and `onStop()` methods get executed. When an Activity is destroyed, the `onDestroy()` method is called. Once an Activity is destroyed, it has to be recreated to return to the active state.

As there may be a flurry of messages that may get generated in LogCat, it would be convenient to filter the app-specific log messages. To add a new filter, click on + sign of the Saved Filters view, as depicted in Fig. 4.6. This will open the Logcat Message Filter Settings dialog box, as depicted in Fig. 4.7.

Enter a relevant name in the *Filter Name* field (Fig. 4.7). This name will appear in the Saved Filters view, as depicted in Fig. 4.8. The LogCat message can be filtered in various ways, but we are using filter *by Log Tag*. Enter the `tagname` parameter (refer to Snippet 4.1) of `Log.i()`[2] in the *by Log Tag* field, as depicted in Fig. 4.7. Select *info* in *by Log Level* drop-down; this corresponds to the log level selected in Snippet 4.1.

Click *OK* in the Logcat Message Filter Settings dialog box to save the filter, as depicted in Fig. 4.8.

[2] i in Log.i stands for info.



**Logcat Message Filter Settings**

Filter logcat messages by the source's tag, pid or minimum log level.
Empty fields will match all messages.

Filter Name: ActivityLifeCycle

by Log Tag: ActivityLifeCycleMethods

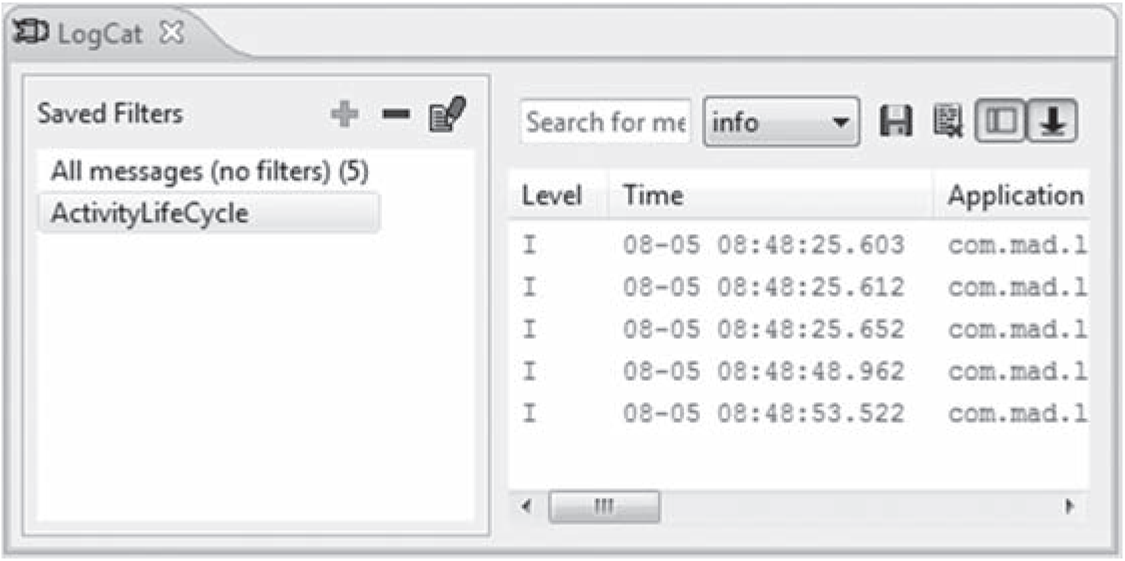
by Log Message:

by PID:

by Application Name:

by Log Level: info

**Figure 4.7** | Logcat Message Filter Settings.



All messages (no filters) (5)
ActivityLifeCycle

```
Level  Time                 Application
I      08-05 08:48:25.603   com.mad.l
I      08-05 08:48:25.612   com.mad.l
I      08-05 08:48:25.652   com.mad.l
I      08-05 08:48:48.962   com.mad.l
I      08-05 08:48:53.522   com.mad.l
```

**Figure 4.8** | Updated Saved Filters view of LogCat.

The LogCat outputs shown in Figs. 4.2–4.5 were captured after applying the filter. Having understood the core programming component of UI (Activity), its states and life-cycle methods, let us now focus on building nonprogramming components of UI in Section 4.3.

## 4.3 UI RESOURCES

User interface resources are typically the nonprogramming components of UI that help a developer in laying out the visual structure of screens, along with string constants and images that populate these screens. There are other UI resources such as menu that provide different ways of user interaction and navigation in apps.

### 4.3.1 Layout Resources

Layout resource provides a visual blueprint for an Activity and defines how the UI elements in a screen is arranged. Layout resources are XML files located in the res\layout folder. A layout file may be associated with one or more Activities. You may recall the `setContentView(R.layout.activity_main)` method that associates an Activity with a layout file (activity_main.xml).

Upon creating a layout resource, Android framework generates a unique id for it in the R.java file present in the gen folder. The R.java file contains information about all nonprogramming components, using which a developer can resolve these resources inside programming components. For example, if a layout file is named activity_main.xml, it is represented as an integer constant – `public static final int activity_main` – in the R.java file under a static class – `layout`. This is applicable to all nonprogramming components.

Snippet 4.2 depicts a sample layout file – activity_main.xml – and the resultant integer constant in the R.java file is listed in Snippet 4.3.

```
<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"
 tools:context=".MainActivity">

 <TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_centerHorizontal="true"
  android:layout_centerVertical="true"
  android:text="@string/hello_world"/>

</RelativeLayout>
```

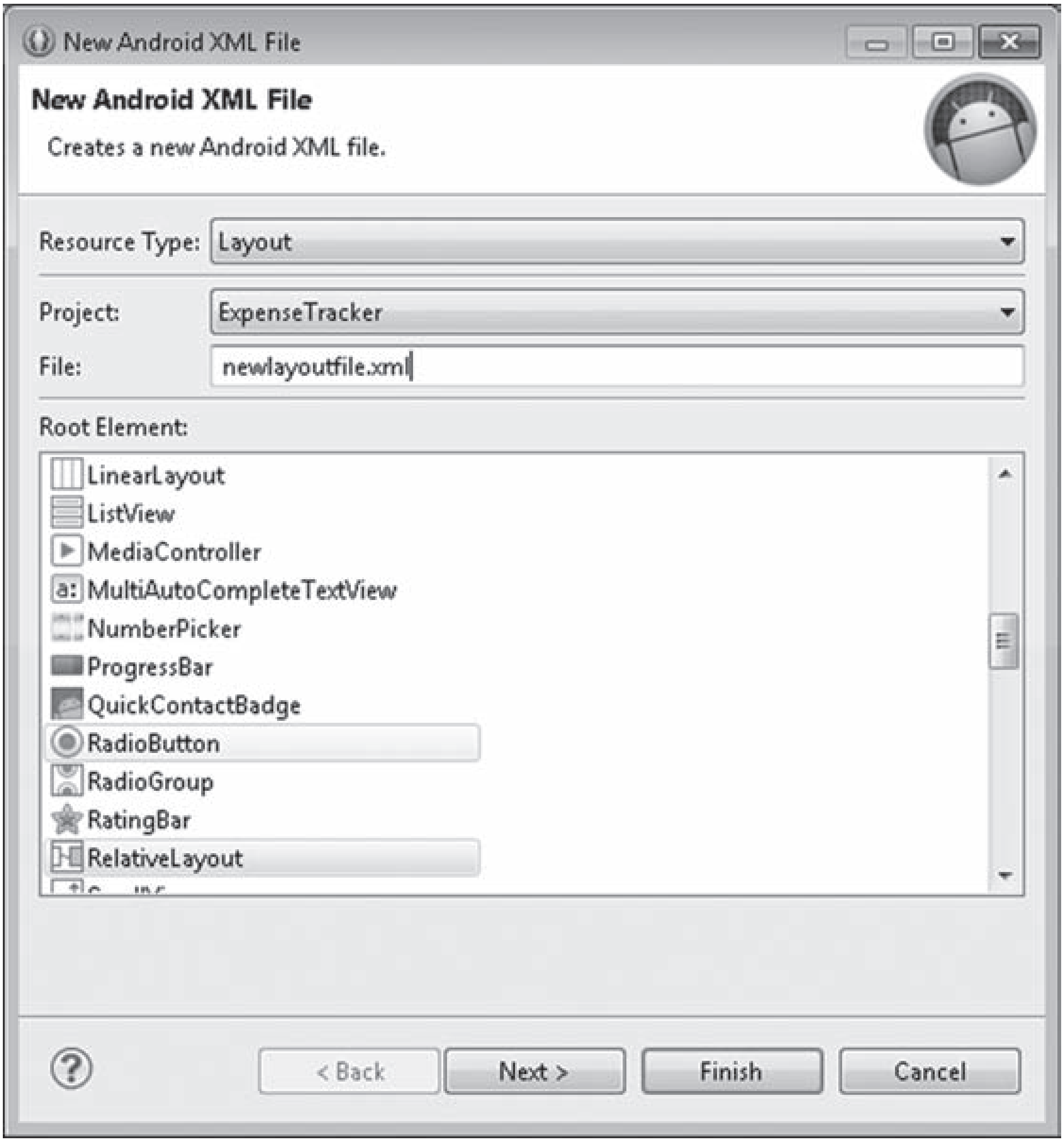
**Snippet 4.2** | Layout file – activity_main.xml.

```
public final class R {
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
}
```

**Snippet 4.3** | Integer constant in R.java for activity_main layout resource.

Android provides the Layout editor[3] to create the layouts. To create a new layout file, right click on the *layout* folder of Android app project structure, and then select *New → Android XML File*, which will pop up the New Android XML File wizard, as depicted in Fig. 4.9.

[3] DroidDraw is another popular open source to create layouts.


New Android XML File

**New Android XML File**

Creates a new Android XML file.

Resource Type: Layout

Project: ExpenseTracker

File: newlayoutfile.xml

Root Element:

**Figure 4.9** | New Android XML File wizard.

The Layout editor also provides a mechanism to drag and drop the UI components in the layout file to create the required UI. The equivalent XML layout file gets generated automatically behind the scenes.

Because apps have different requirements for arranging the UI elements (referred to as views in Android context), the Layout editor provides various ready-to-use layouts such as relative, linear, grid, and table.

Relative layout is the default layout of an Activity wherein views (UI elements) are arranged relative to other views in that layout and/or the layout itself. Because views are always placed relative to one another, rendering views happens automatically without any hassles on devices of diverse screen resolutions, sizes, or orientations.

To create a relative layout, select *RelativeLayout* as the root element from New Android XML File wizard, as already explored in Fig. 4.9. This will yield an XML file with `RelativeLayout` as the root element. When we use relative layout, there are two types of attributes that can be assigned to views – to align views relative to one another in the layout or to align views relative to the layout itself. For example, as depicted in Fig. 4.10 and Snippet 4.4, `EditText1` is placed relative to its container `(RelativeLayout)`, whereas `Button1` and `Button2` are placed relative to other views (`EditText1` and `Button1`, respectively).

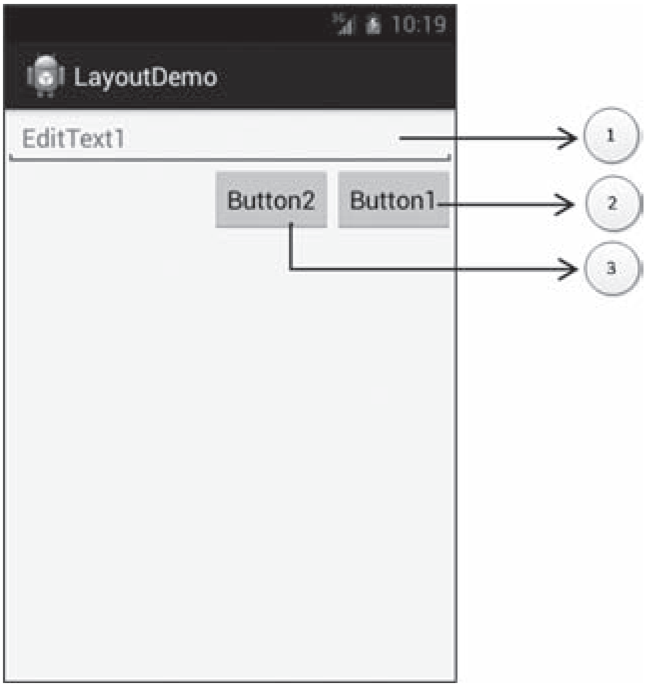


**Figure 4.10** | LayoutDemo app.

```
<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"
 tools:context=".MainActivity">

```

```
 <EditText
  android:id="@+id/editText1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_alignParentLeft="true"
  android:layout_alignParentRight="true"
  android:layout_alignParentTop="true"
  android:hint="EditText1">
  <requestFocus/>
 </EditText>
```

1

2

```
17  <Button
18   android:id="@+id/button1"
19   android:layout_width="wrap_content"
20   android:layout_height="wrap_content"
21   android:layout_alignParentRight="true"
22   android:layout_below="@+id/editText1"
23   android:text="Button1"/>
```

3

```
24  <Button
25   android:id="@+id/button2"
26   android:layout_width="wrap_content"
27   android:layout_height="wrap_content"
28   android:layout_alignBaseline="@+id/button1"
29   android:layout_alignBottom="@+id/button1"
30   android:layout_toLeftOf="@+id/button1"
31   android:text="Button2"/>
32
33 </RelativeLayout>
```

**Snippet 4.4** | Layout XML file of LayoutDemo app – with relative layout.

In Snippet 4.4, `EditText1` contains properties such as `layout_alignParentLeft`, `layout_alignParentRight`, and `layout_alignParentTop` set to `true` (Lines 11–13). This ensures that the `EditText1` is aligned to the left, right, and top of the parent (`RelativeLayout`). `Button1` is aligned to the right of its parent (`RelativeLayout`) using the `layout_alignParentRight` (Line 21) attribute and below `EditText1` using the `layout_below` attribute (Line 22). `Button2` is placed to the left of `Button1` using the `layout_toLeftOf` attribute (Line 30). The `layout_alignBaseline` and `layout_alignBottom` attributes (Lines 28 and 29) ensure that the baseline and bottom edge of `Button2` are matched with those of `Button1` so that both views are in the same line.

On the other hand, linear layout ensures that UI elements are placed either horizontally or vertically. It is defined by using the `LinearLayout` tag in the layout file. Snippet 4.5 shows how the linear layout can be used to recreate the screen of LayoutDemo app (shown in Fig. 4.10).

```
1  <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
2   android:layout_width="match_parent"
3   android:layout_height="match_parent"
4   android:orientation="vertical">
```

1

```
5   <EditText
6    android:id="@+id/editText1"
7    android:layout_width="match_parent"
8    android:layout_height="wrap_content"
9    android:ems="10">
10   <requestFocus>
11  </EditText>
12
13  <LinearLayout
14   android:layout_width="match_parent"
```

```
15   android:layout_height="wrap_content"
16    android:gravity="right">
17

18  <Button                                                    3
19   android:id="@+id/button2"
20   android:layout_width="wrap_content"
21   android:layout_height="wrap_content"
22   android:text="Button2"/>
23

24  <Button                                                    2
25   android:id="@+id/button1"
26   android:layout_width="wrap_content"
27   android:layout_height="wrap_content"
28   android:text="Button1"/>
29
30 </LinearLayout>
31
32 </LinearLayout>
```

**Snippet 4.5** | Layout XML file of LayoutDemo app – with linear layout.

The most important attribute of the linear layout is `android:orientation` (Line 4). Using this attribute, we can place the elements horizontally or vertically in the layout. Here, we have set it to `vertical`. We can also see that layouts can be nested (see Line 13). The parent `LinearLayout` now hosts a child `LinearLayout`, which in turn contains the two buttons: `Button1` and `Button2`. Another important attribute of the `LinearLayout` is `android:gravity`. Gravity refers to the alignment of child elements in a container. In Line 16, the child `LinearLayout`'s gravity is set as `right` so that all child elements of this layout would be aligned to the right.

### 4.3.2 String Resources

Android provides a mechanism to maintain string resources in XML files so that they can be reused across the app codebase. We could have hard coded the strings but at the cost of reusability. If an app developer requires to change a string, instead of fiddling with the code, he or she just needs to change appropriate string resource in the XML file.

Usually string resource XML files are kept in res\values folder. To create a string resource XML file, right click on *values* folder, select *New → Android XML File*, and provide the appropriate values in New Android XML File wizard (shown earlier in Fig. 4.9). Editing of this XML file can be done either in a visual XML file editor or a simple text-based editor.

Two most commonly used string resources are string constants and string arrays. String constant is a simple key–value pair. You may recall from Snippet 2.4 of Chapter 2 that in the HelloWorld app, string "Hello world!" is being assigned to the `hello_world` key in strings.xml resource file, under the `<resources>` tag. Similar to a layout resource, a unique id is generated for string constants in the R.java file that may be further used to programmatically access it, as depicted in Snippet 4.6.

```
Resources resources=getResources();
String message1=resources.getString(R.string.hello);
```

**Snippet 4.6** | Accessing string constant programmatically.

The method `getResources()` of `Resources` class is used to access all app resources that are present in the app project structure. The string resource is further extracted using the `getString()` method by passing the unique id of string constant (`R.string.hello`) generated in the R.java file (see Line 2 of Snippet 4.6).

String array is an array of strings assigned to a key. It is declared in the string resource file using the `<string-array>` tag, as depicted in Snippet 4.7. The individual strings in the array are declared using the `<item>` tag. The key is declared as name attribute of the `<string-array>` tag.

```
<resources>
<string-array name="nations">
     <item>India</item>
     <item>Malaysia</item>
     <item>Singapore</item>
     <item>Thailand</item>
</string-array>
</resources>
```

**Snippet 4.7** | Declaring string array resource programmatically.

The string array resource is extracted from the `Resources` object using the `getStringArray()` method by passing the unique id of string array constant (`R.array.nations`) generated in the R.java file, as depicted in Line 2 of Snippet 4.8.

```
1  Resources resources=getResources();
2  String[] nations=resources.getStringArray(R.array.nations);
```

**Snippet 4.8** | Accessing string array resource programmatically.

String resources are also useful in localizing an app in the case of multilingual rollout. This is done by placing string resources in locale-specific resource folders such as res\values-fr and res\values-hi for French and Hindi, respectively. All locale equivalents of a specific string resource will use the same key so that Android runtime automatically picks up the appropriate value based on the locale.

### 4.3.3 Image Resources

The very first place where a user interacts with an image resource is the app icon itself. There are several other places where images can be incorporated such as in image view, layout background, button, and some other UI elements to enhance the visual appeal of an app.

Image files are placed in res\drawable folders. Android provides a mechanism to access these images in app components programmatically or nonprogrammatically. Programmatically, `Resources` class is used to access an image resource using the `getDrawable()` method (see Snippet 4.9). The return type of this method is `Drawable`. The unique id of image resource (`R.drawable.filename`) has to be passed as parameter to this method. In this example, ic_launcher is the filename that represents the app icon.

```
Drawable drawable=resources.getDrawable(R.drawable.ic_launcher);
```

**Snippet 4.9** | Accessing image resource programmatically.

Nonprogrammatically, an image resource is associated with other UI resources or elements in the layout file to set the background of layout or other UI elements, or to display the image itself. Using `@drawable/filename` as the value to the properties that set the background (`android:background`) of a view, an image resource can be set as the background. An example of this is shown in Snippet 4.10.

```
<RelativeLayout
    ...
    android:background="@drawable/ic_launcher">
       …
</RelativeLayout>
```

**Snippet 4.10** | Accessing image resource nonprogrammatically.

We will take a further deep dive into image resources along with other nitty-gritties of graphics and animations in Chapter 7. After having an understanding of Activity and UI resources, let us now explore UI elements and associated events to create a functional UI.

## 4.4 UI ELEMENTS AND EVENTS

We have already noticed two of the most commonly used views (UI elements) – TextView and Button. TextView is an uneditable content holder for text. Button is another view used to trigger an event in response to a user action.

Android provides many more views to handle data and user interaction for a functional UI. The user interaction may happen in multiple forms – data entry in an editable view, display of textual/visual data, or triggering of an event based on user actions such as click of a button, drag of a list, or multitouch gestures.

Any user interaction with a view triggers an event associated with it. Such events need to be handled by a developer to provide required functionalities in views. In order to understand the event-handling paradigm in Android, let us have a quick look at Section 4.4.1, before jumping into individual views and events associated with them that a developer can leverage to make a functional UI.

### 4.4.1 Event-Handling Paradigm

Each view in Android is an *event source*. The event source (view) generates an *event object* in response to a user action. This event object is passed on to an *event listener*, provided this view has registered for that event listener. Event listener is a specialized interface designed to listen to the generated event, and respond to it through callback methods, also referred to as *event handlers*. The event-handling paradigm and its related vocabulary are summarized in Table 4.1.

**Table 4.1** | Event-handling paradigm vocabulary

| Vocabulary | Description | Example |
|---|---|---|
| Event source | The view on which user performs an action. | Button |
| Event object | The object generated by the view. | Click |
| Event listener | The interface associated with the view that listens to the event. | OnClickListener |
| Event handler | The callback method that responds to the event. | onClick() |

### 4.4.2 UI Elements

We have already seen that UI elements are declared in layout files. The Layout editor provides a mechanism to drag and drop the required UI components into the layout. The equivalent XML representing the UI element gets generated automatically in the associated layout file that can be tweaked further, if need be.

These UI elements have some commonly used attributes that you may have noticed earlier in this chapter. Before we explore the individual UI elements, let us understand some commonly used attributes, as listed in Table 4.2.

**Table 4.2** | Commonly used attributes of a view

| Attribute | Description | Sample Usage |
|---|---|---|
| `android:id` | UI element is identified using this unique id. `@+id/id_name` is used to set the id. | `android:id=“@+id/id_name”` |
| `android:layout_width` | Sets the width of an UI element. Possible values are `wrap_content`, `match_parent`, `fill_parent`, or hardcoded values (not advisable). | `android:layout_width=“wrap_content”`<br>`android:layout_width=“20dp”`<br>`android:layout_width=“20px”` |
| `android:layout_height` | Sets the height of an UI element. Possible values are `wrap_content`, `match_parent`, `fill_parent`, or hardcoded values (not advisable). | `android:layout_height=“wrap_content”`<br>`android:layout_height =“20dp”`<br>`android:layout_height =“20px”` |
| `android:gravity` | Sets the alignment of components inside an UI element. There are many possible values of which some are `center`, `left`, `right`, `center_horizontal`, and `center_vertical`. We can also compound values, e.g., `top\|left` | `android:gravity=“center”`<br>`android:gravity=“center_vertical”`<br>`android:gravity=“center_horizontal”` |

Android supports a lot more attributes apart from those mentioned in Table 4.2. The usage of attributes is typically based on the required UI design.

*Button*, as we have seen earlier, is created using the `<Button>` tag in a layout file, as also depicted in Snippet 4.11.

```
1  <Button
2   android:id=“@+id/clickButton”
3   android:layout_width=“wrap_content”
4   android:layout_height=“wrap_content”
5   android:text=“Click Me”/>
```

**Snippet 4.11** | Declaring a Button.

We can access the Button created in a layout resource using the `findViewById(int)` method. This method takes the unique id of the Button (Line 2 of Snippet 4.11) as parameter. The return type of this method is `View`, and to assign it to a `Button` reference variable, we have to typecast it to a `Button`, as depicted in Snippet 4.12.

```
Button clickButton=(Button)findViewById (R.id.clickButton);
```

**Snippet 4.12** | Accessing a Button.

A user can perform various actions on the Button. One of the most common actions is to click it. To make a Button respond to the click event, we need to set a listener on it. The listener associated with handling click events is the `OnClickListener`. To set the listener on a Button, we can either let the Activity hosting it implement the `OnClickListener` interface or pass an anonymous inner class as parameter to the `setOnClickListener()` method of the `Button` as shown in Snippet 4.13.

```
clickButton.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View arg0) {

     Log.i("Click event", "A button has been clicked");
    }
});
```

**Snippet 4.13** | Making a Button respond to a Click event.

The `onClick()` method is the event handler that is automatically executed when a user clicks the Button.

*EditText* is an Android component used to accept user input. It is created using the `<EditText>` tag in the layout file as depicted in Snippet 4.14.

```
<EditText
 android:id="@+id/editText1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
/>
```

**Snippet 4.14** | Declaring an EditText.

The `findViewById(int)` method is used to access the EditText, as shown in Snippet 4.15. As we have seen earlier that the `findViewById(int)` method returns a `View` object, therefore we need to typecast this returned object to `EditText`.

```
EditText editText1=(EditText)findViewById(R.id.editText1);
```

**Snippet 4.15** | Accessing an EditText.

We use the `getText()` method to get the text entered by the user, as shown in Snippet 4.16.

```
String enteredText=editText1.getText().toString();
```

**Snippet 4.16** | Getting the text entered in EditText.

We may also want to set listeners for various actions on EditText. One of the common actions is when the user changes the focus away from the view. The event, listener, and handler associated with this event are specified in Table 4.3, and the respective implementation is shown in Snippet 4.17.

**Table 4.3** | Event handling associated with EditText

| Event Object | Event Listener | Event Handler |
|---|---|---|
| FocusChange | OnFocusChangeListener | onFocusChange() |

```
editText1.setOnFocusChangeListener(new OnFocusChangeListener() {

     @Override
      public void onFocusChange(View arg0, boolean arg1) {

        Log.i("Focus changed event", "The focus on the edit text
              has been changed");
      }
});
```

**Snippet 4.17** | Making an EditText respond to FocusChange event.

*Checkbox* is an Android component that accepts user input choices. It is created using the `<CheckBox>` tag in the layout file as depicted in Snippet 4.18.

```
<CheckBox
 android:id="@+id/checkBox1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="CheckBox"/>
```

**Snippet 4.18** | Declaring a CheckBox.

We can access a CheckBox by using the `findViewById(int)` method and typecasting the resulting `View`, as shown in Snippet 4.19.

```
CheckBox checkBox=(CheckBox)findViewById(R.id.checkBox1);
```

**Snippet 4.19** | Accessing a CheckBox.

The most common event on a CheckBox is the CheckedChange event, that is, switching of its state from checked to unchecked or vice versa. The event, listener, and handler are described in Table 4.4.

**Table 4.4** | Event handling associated with CheckBox

| Event Object | Event Listener | Event Handler |
|---|---|---|
| CheckedChange | OnCheckedChangeListener | onCheckedChange() |

Snippet 4.20 describes how we can set a listener to the `CheckBox` and respond to changes in its state.

```
checkBox.setOnCheckedChangeListener(new
OnCheckedChangeListener() {

     @Override
      public void onCheckedChanged(CompoundButton arg0, boolean
                                  arg1) {
```

```
          Log.i("CheckedChange event","The state of the check box
                  has changed");
        }
});
```

**Snippet 4.20** | Making a CheckBox respond to a change in state.

*RadioGroup* is used to define a set of options among which only one can be chosen by the user. A RadioGroup in turn contains many *RadioButtons*, which are defined in the layout using the `<RadioGroup>` and `<RadioButton>` tags, respectively, as shown in Snippet 4.21.

```
<RadioGroup
 android:id="@+id/radioGroup1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content">

 <RadioButton
  android:id="@+id/radio0"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:checked="true"
  android:text="RadioButton"/>

 <RadioButton
  android:id="@+id/radio1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="RadioButton"/>
 </RadioGroup>
```

**Snippet 4.21** | Declaring a RadioGroup with RadioButtons.

A RadioGroup defined in the layout file is accessed using the `findViewById(int)` method. The `View` returned from this method is typecasted into `RadioGroup` so that desired operations can be performed on the obtained view. Snippet 4.22 depicts how a RadioGroup can be accessed.

```
RadioGroup radioGroup=(RadioGroup)findViewById(R.id.radioGroup1);
```

**Snippet 4.22** | Accessing a RadioGroup.

A change in the state of any RadioButton inside a RadioGroup triggers the CheckedChange event on the RadioGroup. Table 4.5 describes the event handling associated with a RadioGroup.

**Table 4.5** | Event handling associated with a RadioGroup

| Event Object | Event Listener | Event Handler |
|---|---|---|
| CheckedChange | OnCheckedChangeListener | onCheckedChange() |

The `onCheckedChange()` event handler is called with two arguments as depicted in Snippet 4.23. The second argument to this callback method indicates the id of the RadioButton that is checked.

```
radioGroup.setOnCheckedChangeListener(new
OnCheckedChangeListener() {

    @Override
    public void onCheckedChanged(RadioGroup arg0, int arg1) {

     Log.i("Checked Change event on RadioGroup", "Newly
            checked RadioButton's id is "+arg1);
    }
});
```

**Snippet 4.23** | Making a RadioGroup respond to CheckedChange event.

*ListView* is one of the most common UI elements to show a scrollable list of items on the screen wherein each item is selectable. It is created using the `<ListView>` tag in the layout file as depicted in Snippet 4.24.

```
<ListView
 android:id="@+id/listView1"
 android:layout_width="match_parent"
 android:layout_height="wrap_content">
</ListView>
```

**Snippet 4.24** | Declaring a ListView.

The ListView can be populated in two ways: either at compile time through a string array resource (refer to Section 4.3.2) or programmatically at runtime. Populating ListView at compile time with string array resource is pretty straightforward. We just need to set `android:entries` attribute to refer to a string array resource (see Line 5 of Snippet 4.25). The string array resource is depicted in Snippet 4.26.

```
1  <ListView
2   android:id="@+id/listView1"
3   android:layout_width="match_parent"
4   android:layout_height="wrap_content"
5   android:entries="@array/nations">
6  </ListView>
```

**Snippet 4.25** | Populating ListView at compile time.

```
<string-array name="nations">
 <item>India</item>
 <item>Malaysia </item>
 <item>Singapore</item>
 <item>Thailand</item>
</string-array>
```

**Snippet 4.26** | String array resource to populate ListView.

To populate a ListView at runtime we need to use adapters. An adapter decouples the data and the ListView in which the data has to be rendered. To achieve the decoupling, the adapter is first associated with the required data and then it is attached to the ListView. As data changes, the adapter gets refreshed and eventually refreshes the ListView. Snippet 4.27 depicts how to populate a ListView at runtime.

```
1  ListView listView=(ListView)findViewById(R.id.listView1);
2  String nations[]=getResources().getStringArray(R.array.nations);
3  ListAdapter adapter=new
   ArrayAdapter<String>(getApplicationContext(),
   android.R.layout.simple_list_item_1,nations);
4  listView.setAdapter(adapter);
```

**Snippet 4.27** | Associating adapter with data and ListView.

In Snippet 4.27, we first access the ListView using the `findViewById(int)` method (Line 1). Also, the string array resource used earlier (Snippet 4.26) is now accessed via the `getResources().getStringArray(int)` method and assigned to a variable called `nations`, which now acts as the data source for the ListView (Line 2). In Line 3 of Snippet 4.27, we create an `adapter` as reference to the `ListAdapter` class. This `adapter` is instantiated using an object of `ArrayAdapter`. The constructor of `ArrayAdapter` in this case takes three parameters, viz. context, layout, and array. The layout parameter specifies how the data obtained from the data source (array) is to be displayed in the ListView. The array parameter specifies the data source for this adapter. Line 4 of Snippet 4.27 shows how we can attach the adapter onto the ListView.

Once the data is populated in the ListView, the user can perform various actions on it. Typically, the user clicks on an item in the ListView. This generates an ItemClick event. The event handling related to ItemClick of a ListView is shown in Table 4.6.

**Table 4.6** | Event handling associated with ListView

| Event Object | Event Listener | Event Handler |
|---|---|---|
| ItemClick | OnItemClickListener | onItemClick() |

Snippet 4.28 demonstrates how we can set a listener on the ListView to handle the ItemClick event.

```
1  listView.setOnItemClickListener(new OnItemClickListener() {
2  @Override
3   public void onItemClick(AdapterView<?> arg0, View arg1, int
                             arg2, long arg3) {
4       Log.i("Item Click event on ListView", "The position of
                 the item clicked in the ListView is "+arg2);
5   }
6  });
```

**Snippet 4.28** Making a ListView respond to the ItemClick event.

The `onItemClick()` event handler gets invoked with four arguments: the ListView in which the item was clicked, the row inside ListView that was clicked, the position of the row that was clicked, and the id of the row that was clicked. In Line 4 of Snippet 4.28, we have logged the position of the row clicked.

*ImageView* is a container for image resources. It is created using the `<ImageView>` tag. The `src` attribute resolves the location of the image that needs to be rendered in ImageView, as depicted in Snippet 4.29.

```
1  <ImageView
2   android:id="@+id/imageView1"
3   android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:src="@drawable/ic_launcher"/>
```

**Snippet 4.29** | Declaring an ImageView.

An Image can also be rendered in the ImageView programmatically using the `setImage-Drawable(Drawable)` method, as depicted in Snippet 4.30.

```
ImageView imageView=(ImageView)findViewById(R.id.imageView1);
Drawable drawable=getResources().getDrawable(R.drawable.
ic_launcher);
imageView.setImageDrawable(drawable);
```

**Snippet 4.30** | Rendering image in ImageView programmatically.

*Dialog* is a modal window displayed on the current Activity that supports the user to perform small tasks related to the Activity in focus. There are various types of dialogs that are supported in Android such as AlertDialog, ProgressDialog, TimePickerDialog, and DatePickerDialog.

An AlertDialog is created using a `Builder` class, as depicted in Snippet 4.31.

```
1  AlertDialog.Builder builder=new
   AlertDialog.Builder(MainActivity.this);
2  builder.setTitle("Alert Dialog");
3  builder.setMessage("This is an Android alert dialog");
4  builder.setPositiveButton("Ok", new OnClickListener() {

5  @Override
6  public void onClick(DialogInterface arg0, int arg1) {
7       Toast.makeText(getApplicationContext(), "You have clicked
   on the positive button of the Alert Dialog",
   Toast.LENGTH_LONG).show();
8       }
9  });
10  builder.setNegativeButton("Cancel", new OnClickListener() {
11  @Override
12  public void onClick(DialogInterface arg0, int arg1) {
13       Toast.makeText(getApplicationContext(), "You have
         clicked on the negative button of the Alert Dialog",
         Toast.LENGTH_LONG).show();
14       }
15  });
16  AlertDialog alertDialog=builder.create();
17  alertDialog.show();
```

**Snippet 4.31** | Creating an `AlertDialog`.

The `Builder` class is an inner class of `AlertDialog` that is used to set the layout and behavior of the dialog. We can observe that the `Builder` allows us to configure the title, message, and buttons of the `AlertDialog` (Lines 2–15). An AlertDialog typically shows two buttons, positive and negative, to collect response from the user. The `Builder` provides the `setPositiveButton()` and `setNegativeButton()` methods to configure this (see Lines 4 and 10). We have to set listeners to monitor the user's action on these buttons. In Lines 7 and 13, the user is shown a message on click of any of the buttons using a `Toast`.

Toast is an Android widget used to show unobtrusive messages to the user. It is created using the `Toast` class. To create a Toast message, we have to use the `makeText()` method that returns a `Toast` object. The `makeText()` method accepts the following parameters – context, text to be displayed, and duration for which the Toast has to be displayed. Toast is displayed to the user using the `show()` method.

## 4.5 LET'S APPLY

Having learnt how to create an Activity, prepare layouts for it, and add a variety of UI components to it, let us now apply what we have learnt. In this section, let us create the `SubscribeActivity` for the 3CheersCable app. This Activity allows the user to select the channel categories of his or her choice to subscribe to. The selected channel categories are then displayed on a Confirm subscription dialog window for user confirmation (see Fig. 4.11).

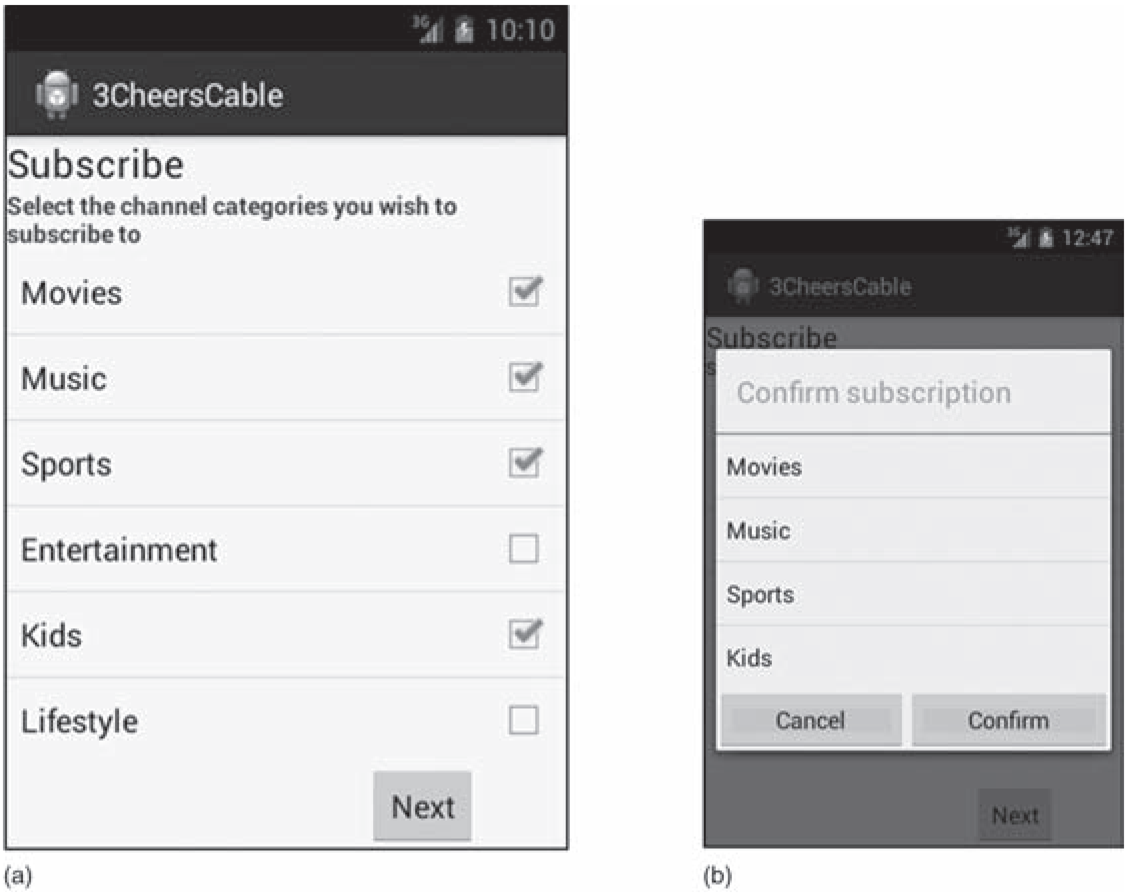


**Figure 4.11** | `SubscribeActivity`: (a) Subscribe Activity and (b) Confirm subscription dialog window.

For the sake of simplicity, let us break down the process involved in creating this Activity into following simple steps, which are easy to understand and follow:

**Step 1: Starting up:**

- Create a new Android project titled 3CheersCable.
- Create a package `com.app3c` for the subscription module.
- Select the latest version of the Android platform available.
- Select the Create activity option in the New Android Application wizard and name the Activity as `SubscribeActivity`.

**Step 2: Setting up the layout resource:** Click *Finish*. We now have an Android project with a folder structure similar to the HelloWorld app that we had created in Chapter 2. Then

- Create a layout resource activity_subscribe.xml in the res\layout folder.
- Set the argument passed to the `setContentView()` method as `R.layout.activity_subscribe` in the SubscribeActivity.java file in order to allow the Activity to use activity_subscribe.xml as its layout resource.

**Step 3: Adding UI components:** Let us now add UI components to the layout file (activity_subscribe.xml):

- Set `<RelativeLayout>` as the root element in activity_subscribe.xml.
- Add TextView to this relative layout by dragging a TextView component from the Form Widgets section of the Palette and dropping it on the Graphical Layout editor. This TextView will work as the title or the heading of this Activity. The following code is automatically generated in the activity_subscribe.xml file.

```
<TextView
   android:id="@+id/textView1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_alignParentLeft="true"
   android:layout_alignParentTop="true"
   android:text="@string/subs_heading">
</TextView>
```

  The TextView uses a string resource subs_heading whose value has to be set as "Subscribe" in strings.xml.

- Similarly, add another TextView to provide instruction – "Select the channel categories you wish to subscribe to".
- Drag and drop a ListView from the Composite section of the Palette. This will generate the following code in the activity_subscribe.xml file.

```
<ListView
    android:id="@+id/listView1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="80dp"
    android:layout_alignParentBottom="true">
</ListView>
```

  The `layout_alignParentBottom` attribute is set to `true` so that the ListView does not hide the Button below it. This ListView will display the list of channel categories that the user can subscribe from.

- Drag and drop a Button (*Next*) from the Form Widgets section onto the Graphical Layout editor.

```
<Button
   android:id="@+id/button1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_alignParentBottom="true"
   android:layout_alignRight="@id/listView1"
   android:layout_marginRight="50dp"
   android:text="@string/subs_next"/>
```

**Step 4: Setting up the ListView**: Let us now populate the ListView created in the previous step:

- Create an `ArrayAdapter` in the `onCreate()` method that will be used to set the `ListAdapter` for our ListView. The `ArrayAdapter` constructor takes the app context, the layout description of each list item, and the list of objects to display in the ListView, as parameters.

```
ArrayAdapter<String> categoryListAdapter=new
ArrayAdapter<String>(this,
android.R.layout.simple_list_item_multiple_choice,category);
```

  The layout description used here is `simple_list_item_multiple_choice`. Use this description to add a checkbox to every list item, hence allowing multiple selections.

- Create a method `getCategoryNames()` that will return a list of categories. This list is used to populate `category` (the third argument used in the constructor of `ArrayAdapter`), which will insert these objects in the ListView.

```
public List<String> getCategoryNames(){
   List<String> categoryNames=new ArrayList<String>();
   categoryNames.add("Movies");
   …
   return categoryNames;
}
```

  These values are hardcoded in this specific exercise, but we shall learn how to fetch these values dynamically through a Web service in Chapter 6.

- Insert the following lines of code in the `onCreate()` method to set the adapter to the ListView and to enable multiple options in the list to be selected.

```
ListView categoryList=(ListView) findViewById(R.id.listView1);
categoryList.setAdapter(categoryListAdapter);
categoryList.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
```

**Step 5: Handling events**: Let us handle the Click event on the Button added earlier in the layout. On click of the Button, all the selected channel categories will be displayed in the Confirm subscription dialog window.

- Create a reference for the Button in the `onCreate()` method.

```
Button next= (Button)findViewById(R.id.button1);
```

- Register an `onClickListener` for the Button:

```
next.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
    // TODO Logic for the event handler goes here
    }
});
```

- Get a list of all the selected items from `categoryList`. Use a `SparseBooleanArray` to store the position of the checked items. Add these items to an ArrayList that will be used to set up the ListView in the dialog. Add this code to the `onClick()` callback method.

```
List<String> confirmed=new ArrayList<String>();
SparseBooleanArray sp=categoryList.getCheckedItemPositions();
for(int i=0;i<category.size();i++) {
    if(sp.get(i)){
       confirmed.add(category.get(i));}     }
```

**Step 6: Creating a dialog**: Now let us create the dialog that appears on click of the *Next* button. The dialog will have a ListView containing names of the selected channel categories and two buttons to cancel or confirm the selection.

- Create a layout file subscribe_dialog in the layout folder. Add a ListView and two Buttons to the layout.
- Create a new `Dialog` object by passing the current Activity as the context and set the content view of the dialog using the subscribe_dialog layout. This code is written within the `onClick()` callback method of `SubscribeActivity`.

```
subscribe_dialog=new Dialog(SubscribeActivity.this);
subscribe_dialog.setContentView(R.layout.subscribe_dialog);
```

- Populate the ListView in the dialog using the `confirmed` ArrayList obtained from the previous step:

```
ListView lv=(ListView)subscribe_dialog.findViewById(R.id.listView2);
ArrayAdapter<String> la=new ArrayAdapter<String>(SubscribeActivity.this, android.R.layout.simple_list_item_1,confirmed);
lv.setAdapter(la);
```

  Notice that we have used `subscribe_dialog.findViewById()` to obtain the ListView. This is because the ListView here is a part of the dialog, and not of the Activity.

- Set the title for the dialog and call the `show()` method of the dialog.

```
subscribe_dialog.setTitle("Confirm subscription");
subscribe_dialog.show();
```

## 4.6 INTERACTION AMONG ACTIVITIES

So far, we have seen how to create an Activity, define its makeup in a layout file, followed by arranging required UI elements in the layout, and provide functionality to these elements to respond to user actions. This discussion was only around a single Activity.

However an app may comprise more than one Activity. For example, an e-mail app typically has an Activity to list all the messages and another one to display a selected message. In such scenarios, a user has to navigate and share data between Activities. This leads to the requirement for Activities not only to interact with each other but also to send and receive information between them. Android provides this mechanism of navigating and sharing information between Activities using Intent.

As already discussed in brief in Section 2.5 of Chapter 2, Intent is a message-passing framework that is used to carry out interaction between Activities (and also other Android building blocks such as Services and Broadcast Receivers, which we will explore later in Chapter 5).

Let us now explore both the aspects of an Intent with respect to Activities – navigation and exchange of data between Activities – in the following two subsections.

### 4.6.1 Navigating Between Activities

To navigate to an Activity from another Activity, we make use of an Intent. It is a two-step process, as follows:

1. Create an `Intent` object with two parameters, as depicted in Line 1 of Snippet 4.32. The first parameter is context and the second parameter is the second (next) Activity to navigate. `Context` is an object provided by Android runtime environment to the app. It contains the global information about the environment in which the app is running.
2. Pass this `Intent` object to `startActivity()` method, as depicted in Line 2 of Snippet 4.32.

```
1  Intent intent=new Intent
   (getApplicationContext(),SecondActivity.class);
2  startActivity (intent);
```

**Snippet 4.32** | Creating an Intent and starting another Activity.

Every Android component needs to get registered in AndroidManifest.xml after implementation. Therefore, we have to register the Activities created in an app in the manifest file, as shown in Snippet 4.33.

```
<manifest>
 ...
 <application>
   ...
  <activity android:name=“SecondActivity”></activity>
 </application>
</manifest>
```

**Snippet 4.33** | Registering Activity in AndroidManifest.xml.

### 4.6.2 Exchanging Data

We can pass data as part of the Intent that can be retrieved by the second (next) Activity. The simplest way to pass data between Activities is using a key–value pair. We can pass data of types such as boolean, int, float, char, String, arrays, and ArrayList using the key–value pairs. Line 2 of Snippet 4.34 shows adding of a boolean data to an `Intent` object, before the next Activity is started. Data is added as key–value pair using the `putExtra()` method of the `Intent` object. The `putExtra()` method accepts a String as the first parameter, which is the key and the value corresponding to it as the second parameter. In Snippet 4.34, `key1` signifies the key and `true` signifies the value `key1` holds.

```
1  Intent intent=new Intent
   (getApplicationContext(),SecondActivity.class);
2  intent.putExtra (“key1”, true);
3  startActivity (intent);
```

**Snippet 4.34** | Adding a boolean data to an `Intent` object.

The `putExtra()` method is overloaded and accepts primitive data types, arrays, and ArrayList of these data types.

Another way of passing values is by creating an object of type `Bundle`. It acts like a collection of key–value pairs that can be passed to another Activity. Line 2 of Snippet 4.35 shows creation of a `Bundle` object. Lines 3–5 show addition of a collection of key–value pairs to the `Bundle`. This `Bundle` is added to an `Intent` object, before the next Activity is started (Line 6).

```
1  Intent intent=new Intent
   (getApplicationContext(),SecondActivity.class);
2  Bundle bundle=new new Bundle();
3  bundle.putBoolean(“key1”,true);
4  bundle.putString(“key2”,“John”);
5  bundle.putInt(“key3”,18);
6  intent.putExtras(bundle);
7  startActivity (intent);
```

**Snippet 4.35** | Adding a `Bundle` to an `Intent` object.

Now, to retrieve data in the next Activity (at the receiving end), we need to get an instance of the `Intent` passed from the first Activity. This is achieved using the `getIntent()` method. From the `Intent` object, we have to retrieve the key–value pair either directly (see Snippet 4.36) or from the `Bundle` object (see Snippet 4.37).

```
1    Intent receivedIntent=getIntent();
2    boolean varBoolean=receivedIntent.getBooleanExtra(“key1”, false);
```

**Snippet 4.36** | Retrieving key–value pairs directly.

```
1    Intent receivedIntent=getIntent();
2    Bundle receivedData=receivedIntent.getExtras();
3    boolean varBoolean=receivedData.getBoolean(“key1”, false);
```

**Snippet 4.37** | Retrieving key–value pairs from the `Bundle`.

In Snippets 4.36 and 4.37, the `getIntent()` method is used to retrieve the Intent that started the component. In Line 2 of Snippet 4.36, the boolean data that was sent earlier directly using the `putExtra()` method (refer to Snippet 4.34) is retrieved using the `getBooleanExtra()` method.

The first parameter is the key that was passed while sending the data. The second parameter is the default value that must be assigned to the variable in case the key–value pair corresponding to the supplied key is unavailable in the Intent.

In Line 2 of Snippet 4.37, the Bundle that was sent earlier (refer to Snippet 4.35) is retrieved using the `getExtras()` method. One of the key–value pairs is retrieved from the `Bundle` object using the `get-Boolean()` method, which accepts the key as the first parameter and default value as the second.

## 4.7 LET'S APPLY

Now that we have learnt how interaction between Activities takes place, that is, traverse from one Activity to another as well as pass data between Activities through Intents, let us now include these functionalities in 3CheersCable app.

In this section, we will create another Activity – `MainActivity` – that will act as the launcher Activity of 3CheersCable app. The user is navigated from this Activity to the `SubscribeActivity` discussed in Section 4.5.

The screen shots of `MainActivity` in Fig. 4.12 depict the initial and final states of the `MainActivity`.

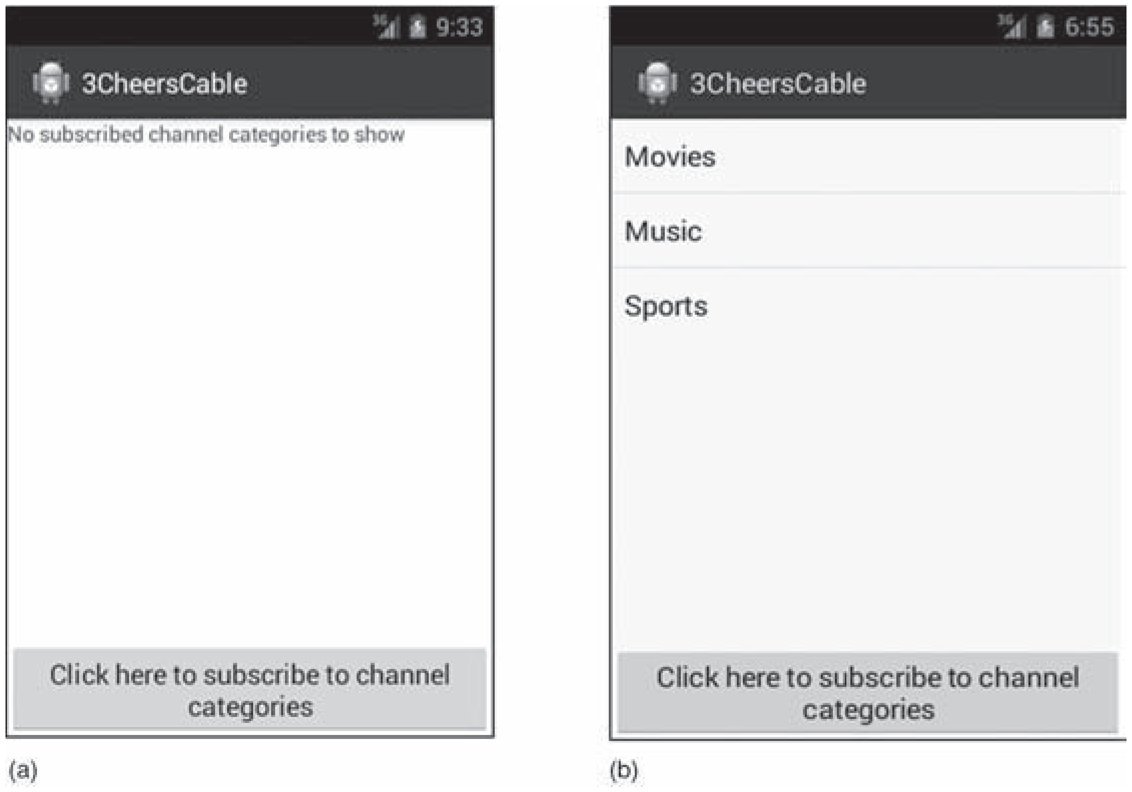

9:33
3CheersCable
No subscribed channel categories to show
Click here to subscribe to channel categories
(a)
6:55
3CheersCable
Movies
Music
Sports
Click here to subscribe to channel categories
(b)

**Figure 4.12** | `MainActivity`: (a) Initial state and (b) final state.

The step-by-step procedure of creating `MainActivity` and modifying the `SubscribeActivity` in order to enable the interaction between Activities, is as follows:

**Step 1: Creating `MainActivity`**: The first step involves the following:

- Create a new Activity called `MainActivity` within the `com.app3c` package. The layout file for this Activity is activity_main.xml, which is to be created under the res\layout folder.
- Set relative layout as the parent layout in activity_main.xml, and add a ListView component within it. This will generate the following code in the XML file.

```
<ListView
 android:id="@+id/listView1"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_alignParentLeft="true"
 android:layout_alignParentTop="true">
</ListView>
```

- Add a TextView after the ListView component.

```
<TextView
 android:id="@+id/textView1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/no_categories"
 android:visibility="invisible"/>
```

  This TextView is used to display the message "No subscribed channel categories to show" in case the ListView is empty.

- Add a Button after the TextView component.

```
<Button
 android:id="@+id/button1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_alignParentBottom="true"
 android:text="Click here to subscribe to channel categories"/>
```

- Now that we have prepared the layout file, let us set it to the `MainActivity`, and get handles to UI components using the following code written in the `onCreate()` method.

```
setContentView(R.layout.activity_main);
listView=(ListView) findViewById(R.id.listView1);
textView=(TextView) findViewById(R.id.textView1);
button=(Button) findViewById(R.id.button1);
button.setOnClickListener(this);
```

- Populate the ListView with values within the `categoryList` (an ArrayList) using an `ArrayAdapter`. If the `categoryList` is empty, set visibility of the TextView to `View.VISIBLE` so that the message "No subscribed channel categories to show" is displayed.

```
la=new ArrayAdapter<String> (this,
android.R.layout.simple_list_item_1,categoryList);
listView.setAdapter(la);
```

```
if(categoryList.size() == 0){
  textView.setVisibility(View.VISIBLE);
}
```

- Write the following code inside `onClick()` to take the user to the `SubscribeActivity` on click of the Button.

```
Intent intent=new Intent(this, SubscribeActivity.class);
startActivityForResult(intent, REQUEST_CATEGORY);
```

The `startActivityForResult()` is generally used to get the result back from an Activity. Here we have used `startActivityForResult()` to start the `SubscribeActivity`, and get the list of selected channel categories as a result back to the `MainActivity`. The second integer parameter in the method is a request code that is used in identifying this call in the next Activity `(SubscribeActivity)`.

- Store the result, that is, the list of selected channel categories in the `SubscribeActivity`, which comes back to the `MainActivity` through the `onActivityResult()` method, into the `categoryList` (an ArrayList) and display it in the ListView.

```
if(data!=null){
   categoryList=data.getStringArrayListExtra(CATEGORIES_LIST);
    la=new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_1,categoryList);
    textView.setVisibility(View.INVISIBLE);
    listView.setAdapter(la);}
```

- The above code is written within an `if(data!=null)` condition so that if no result is obtained from the `SubscribeActivity`, the "No subscribed channel categories to show" message should be displayed. To do this, write the following code:

```
else{
  categoryList.clear();
  la=new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1,categoryList);
  listView.setAdapter(la);
  textView.setVisibility(View.VISIBLE);
}
```

**Step 2: Modifying `SubscribeActivity`**: Now that we have created the `MainActivity` and its layout file, let us now go back to the `SubscribeActivity` created in Section 4.5 and modify it, as follows:

- Earlier we had a list of items displayed in the `SubscribeActivity`, and items selected in the list were displayed in a dialog for confirmation. Now, we need to implement the `onClickListener` interface on the Buttons (Cancel and Confirm) in the Confirm subscription dialog.

```
btn_ok = (Button) subscribe_dialog.findViewById(R.id.button2);
btn_ok.setOnClickListener(SubscribeActivity.this);
btn_cancel=(Button)subscribe_dialog.findViewById(R.id.button1);
btn_cancel.setOnClickListener(SubscribeActivity.this);
```

- On click of the *Confirm* Button, we pass the selected list of channel categories as the result, back to the `MainActivity` using `setResult()` with the arguments – result code and Intent.

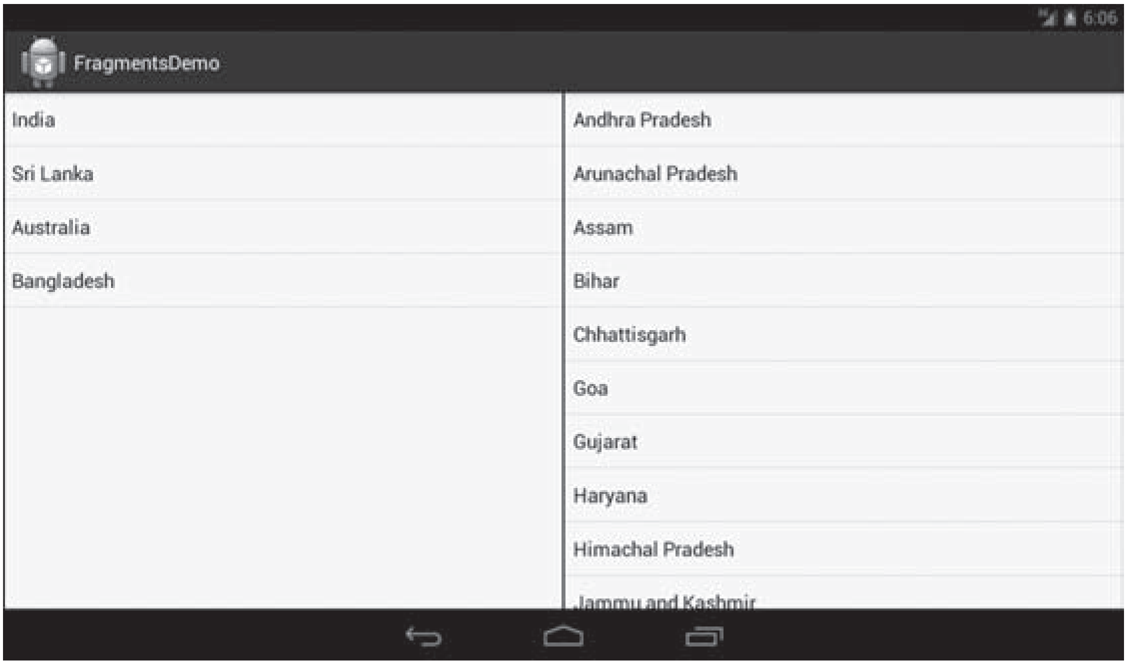
```
Intent intent = new Intent();
intent.putStringArrayListExtra(MainActivity.CATEGORIES_LIST,
 (ArrayList<String>) confirmed);
setResult(MainActivity.REQUEST_CATEGORY, intent);}
subscribe_dialog.dismiss();
this.finish();
```

The `finish()` method closes the current Activity `(SubscribeActivity)` and returns the result to the Activity `(MainActivity)` that started it.

## 4.8 FRAGMENTS

With the advent of tablets in market, developers got more real estate for an app UI. Fragments have been introduced to cater to leverage the larger UIs since the advent of Android 3.0 (Honeycomb – API level 11). Fragments are UI modules and are similar to Activities in many ways.

Fragments allow the flexibility of designing the UI such that in a single screen it is possible to accommodate more than one UI modules. In such a scenario, typically, Activity acts as a container and hosts multiple Fragments to be shown on a single screen. Figure 4.13 depicts two Fragments hosted within an Activity. Fragment on the left shows a list of nations and that on the right shows the corresponding states.



India
Sri Lanka
Australia
Bangladesh

Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
Goa
Gujarat
Haryana
Himachal Pradesh
Jammu and Kashmir

**Figure 4.13** | Activity hosting two Fragments.

### 4.8.1 Building Fragments

Building Fragments of an app can be broadly categorized into three steps: designing the Fragment layout, creating Fragment class, and associating Fragment to an Activity. Let us now try to understand these steps with an example that will help in achieving the result as depicted in Fig. 4.13.

To start with, we have to design the layout of the Fragment in the associated layout XML file. As in the example, we have two Fragments: nation Fragment and state Fragment. We need to define two layout files: nation_fragment.xml and state_fragment.xml (see Snippets 4.38 and 4.39, respectively).

```
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <ListView
  android:id="@+id/listView1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:entries="@array/nationList">
  </ListView>
 </LinearLayout>
```

**Snippet 4.38** | nation_fragment.xml.

```
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical">
 <ListView
  android:id="@+id/listView1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:entries="@array/IndiaStateList">
 </ListView>
</LinearLayout>
```

**Snippet 4.39** | state_fragment.xml.

The next step is to create a class that extends `Fragment` (or its subclasses). This class is required to inflate the Fragment layout and define the behavior of Fragment. Because we have two Fragments, we need to define two Fragment classes: `NationFragment` and `StateFragment` (see Snippets 4.40 and 4.41, respectively).

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

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

**Snippet 4.40** | NationFragment.java.

```
1  public class StateFragment extends Fragment {
2      @Override
3      public View onCreateView(LayoutInflater inflater,
       ViewGroup container,Bundle savedInstanceState) {
4
5      View view=inflater.inflate(R.layout.state_fragment,
       container,false);
6      return view;
7      }
8  }
```

**Snippet 4.41** | StateFragment.java.

In Snippets 4.40 and 4.41, we override the `onCreateView()` method. This is one of the life-cycle methods of a Fragment and is used to instantiate the UI of the Fragment. This method returns a `View` object, which is the UI of the Fragment, created using the `inflate()` method of the `LayoutInflater` class (Lines 5 and 6). The `inflate()` method inflates the Fragment's UI (defined in the Fragment layout file and passed as the first argument here), and then attaches it to the designated container in the host Activity (passed as the second argument here).

The concluding step is to associate the Fragment(s) to the anchor Activity (`MainActivity` here). This can be achieved either statically or dynamically.

To associate the Fragment to an Activity in a static manner, we need to add the `<fragment>` tag in the layout file of the host Activity (see Snippet 4.42). The `<fragment>` tag defines the designated container for Fragments, and has most attributes similar to other UI elements in Android. The most important attribute of the `<fragment>` tag is `android:name` that defines the Fragment class to be loaded in the designated container. As depicted in Lines 10 and 23, the value for `android:name` attribute is the Fragment class created earlier (`NationFragment` and `StateFragment`). No changes need to be made in the Activity class, and as soon as the Activity is rendered, both Fragments are attached to it and we get the result as shown earlier in Fig. 4.13.

```
1  <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
2   xmlns:tools="http://schemas.android.com/tools"
3   android:id="@+id/LinearLayout1"
4   android:layout_width="match_parent"
5   android:layout_height="match_parent"
6   tools:context=".MainActivity">
7
8    <fragment
9     android:id="@+id/nationFragment"
10    android:name="com.mad.fragmentsdemo.NationFragment"
11    android:layout_width="wrap_content"
12    android:layout_height="match_parent"
13    android:layout_weight="3"/>
14
15   <View
16    android:id="@+id/View1"
17    android:layout_width="2dp"
18    android:layout_height="match_parent"
```

```
   android:background=“@android:color/background_dark”/>

  <fragment
   android:id=“@+id/stateFragment”
   android:name=“com.mad.fragmentsdemo.StateFragment”
   android:layout_width=“wrap_content”
   android:layout_height=“match_parent”
   android:layout_weight=“1”/>

</LinearLayout>
```

**Snippet 4.42** | Activity layout file with Fragments attached.

Alternatively, Fragments can be dynamically associated with an Activity using placeholders for them in the layout file of the Activity. Here, this is done using `<FrameLayout>` tag as shown in Snippet 4.43.

```
<LinearLayout
  xmlns:android=“http://schemas.android.com/apk/res/android”
  android:layout_width=“match_parent”
  android:layout_height=“match_parent”
  android:orientation=“horizontal”>

  <FrameLayout
   android:id=“@+id/nationfragmentContainer”
   android:layout_width=“wrap_content”
   android:layout_height=“match_parent”
   android:layout_weight=“1”>
  </FrameLayout>

  <View
    android:id=“@+id/View1”
    android:layout_width=“2dp”
    android:layout_height=“match_parent”
    android:background=“@android:color/background_dark”/>

  <FrameLayout
    android:id=“@+id/stateFragmentContainer”
    android:layout_width=“wrap_content”
    android:layout_height=“match_parent”
    android:layout_weight=“3”>
  </FrameLayout>

</LinearLayout>
```

**Snippet 4.43** | Activity layout file with placeholders for Fragments.

In the `onResume()` method of the Activity, we attach the required Fragments at runtime using the `FragmentManager` API (See Snippet 4.44). In Line 1, an object of `FragmentTransaction` is obtained from the `beginTransaction()` method of the `FragmentManager` class. `FragmentTransaction` is used to perform transactions such as add, replace, or remove on Fragments dynamically. Lines 2 and 3 depict the addition of the Fragments to the Activity using the `add()` method of the `FragmentTransaction` class. The `add()` method accepts the following parameters – id of the placeholder for the Fragment, instance of the Fragment, and a tag with which the Fragment can be referred later, if need be. The `commit()` method used in Line 4 is the most important line in Snippet 4.44. It commits the transactions performed on the Fragments.

```
1  FragmentTransaction
   fragmentTransaction=getFragmentManager().beginTransaction();
2  fragmentTransaction.add(R.id.nationfragmentContainer,new
   NationFragment(),“tagNationFragment”);
3  fragmentTransaction.add(R.id.stateFragmentContainer,new
   StateFragment(),“tagStateFragment”);
4  fragmentTransaction.commit();
```

**Snippet 4.44** | Dynamically adding `NationFragment` and `StateFragment` to an Activity.

Till now, we have seen how to design a Fragment layout, create a Fragment class, and associate a Fragment to an Activity. Working with Fragments requires us to further understand its life cycle and how they interact with each other. Let us now explore these two critical aspects in the next two subsections.

### 4.8.2 Life Cycle of Fragments

Similar to an Activity, Fragment too has its own life cycle. Because Fragment is attached to an Activity, the life cycle of a Fragment is intertwined with the Activity. This results in interlacing of the life-cycle methods of Activity and Fragment.

The Fragment life cycle starts with a call to `onAttach()` method when a Fragment is added to an Activity. The next call is to the method `onCreate()` that is used for initializing instance variables of the Fragment. `onCreateView()` method follows `onCreate()` method to instantiate the UI of the Fragment, as seen earlier in Snippets 4.39 and 4.40. In this method, we inflate the XML layout file that describes the UI of the Fragment. This is followed by call to the `onActivityCreated()` method, which is invoked when the host Activity of this Fragment has completed execution of its own `onCreate()` method. `onStart()` is the next method that is invoked when the Fragment becomes visible. The last method that is called before the Fragment becomes active is `onResume()`, which enables the user to interact with the Fragment. When the Fragment transitions from the active to destroyed state, it goes through `onPause()`, `onStop()`, `onDestroyView()`, `onDestroy()`, and `onDetach()` callbacks.

The main difference between the life cycle of an Activity and that of a Fragment is that the Activity life cycle is managed by Android runtime using task stack (refer Section 4.2.1), whereas Fragments are managed by the host Activity using back stack mechanism. The back stack mechanism helps developer to manage multiple Fragment transactions in an Activity.

### 4.8.3 Interaction Between Fragments

Referring to Fig. 4.13, if a user selects another nation, the second Fragment has to be populated with the respective states. To achieve this, two Fragments have to interact with each other. In this example, as a single Activity hosts two Fragments, the interaction can be achieved as explained below.

To start with, the NationFragment.java (refer to Snippet 4.40) has to be modified as depicted in Lines 7–12 of Snippet 4.45.

```
1  public class NationFragment extends Fragment {
2      MainActivity mainActivity;
3       @Override
4       public View onCreateView(LayoutInflater inflater, ViewGroup
        container, Bundle savedInstanceState) {
5         View view=inflater.inflate(R.layout.nation_fragment,
          container,false);
6         mainActivity=(MainActivity)getActivity();
7         ListView nationsList =
          (ListView)view.findViewById(R.id.listView1);
8         nationsList.setOnItemClickListener(new
          OnItemClickListener() {
9            public void onItemClick(AdapterView<?> arg0, View
             arg1, int arg2, long arg3) {
10             mainActivity.onFragmentAction(getResources().
               getStringArray(R.array.nationList)[(int)arg3]);
11           }
12        });
13        return view;
14      }
15  }
```

**Snippet 4.45** | Modified NationFragment.java.

In Line 7 of Snippet 4.45, we access the ListView in the Fragment that displays the list of nations. In Line 8, we set an `OnItemClickListener` on the ListView. The logic inside the `onItemClick()` handler determines the country clicked by the user and passes that as a parameter to the `onFragmentAction()` method. The `onFragmentAction()` is a user-defined method in the `MainActivity` class that passes on the selected country to the `StateFragment` so that the `StateFragment` can modify the list it is displaying. The logic of `onFragmentAction()` method is as shown in Snippet 4.46.

```
1  public void onFragmentAction(String country) {
2      FragmentManager fragmentManager=getFragmentManager();
3      StateFragment stateFragment=
       (StateFragment)fragmentManager.findFragmentByTag
        ("tagStateFragment");
4      stateFragment.setList(country);
5  }
```

**Snippet 4.46** | `onFragmentAction()` method definition in `MainActivity`.

In the `onFragmentAction()` method, an instance of `FragmentManager` is obtained as shown in Line 2 of Snippet 4.46. The obtained instance is further used to access the `StateFragment`. The `findFragmentByTag()` method takes a String variable as parameter and returns the `Fragment` identified by the tag supplied. In Line 4, the `setList()`, a user-defined method of `StateFragment` is invoked with the country selected by the user as argument. The definition of `setList()` method is as shown below in Snippet 4.47.

```
1  public void setList(String country) {
2      Resources res=getResources();
3      ListView lv=
       (ListView)getView().findViewById(R.id.listView1);
4      String[] states=null;
5      if(country.equalsIgnoreCase(res.getStringArray(R.array.nati
          onList)[0]))
       {
6         states=res.getStringArray(R.array.IndiaStateList);
7       }
8      else
       if(country.equalsIgnoreCase(res.getStringArray(R.array.nati
          onList)[1]))
       {
9         states=res.getStringArray(R.array.SriLankaStateList);
10      }
11     ...   // Similarly for other nations
12      ArrayAdapter<String> adapter=new
        ArrayAdapter<String>(getActivity(),
        android.R.layout.simple_list_item_activated_1,states);
13      lv.setAdapter(adapter);
14  }
```

**Snippet 4.47** | `setList()` method definition in `StateFragment`.

The `setList()` method in Snippet 4.47 accepts the country selected by the user. On the basis of this selection, it populates the required states into `states` array (see Lines 5–11). The `states` array is further used in Line 12 to set the adapter of the ListView, thereby ensuring that when a nation is selected in the `NationFragment`, its corresponding states are displayed in the `StateFragment`.

## 4.9 LET'S APPLY

In this section, let us extend the functionality of 3CheersCable app to include Fragments. By now we know that we can either create static Fragments inside an Activity by including the `<fragment>` tag in the layout XML or spawn Fragments dynamically using the `FragmentManager` class. In this section, we shall be using the latter approach and create Fragments dynamically.

In this section, we will create another Activity – `SettingsActivity` – where user can view channels and do other app-related settings. The `SettingsActivity` acts as a base Activity on which we shall populate Fragments dynamically. The user is navigated to this Activity from the *SETTINGS* option in the `MainActivity`, discussed later in Section 4.11.

The screen in Fig. 4.14 depicts the `SettingsActivity` in both orientations – portrait and landscape. You may observe that in the landscape mode, this Activity contains two Fragments: `OptionsListFragment` and `CategoriesListFragment`. On clicking a channel category, the user will be shown `ChannelsDialogFragment` (Fig. 4.15). In the portrait mode, only one Fragment is visible when settings option is selected. Clicking *View Channels* here will lead to `CategoriesListFragment`, as shown obscured by the `ChannelsDialogFragment` (portrait mode).

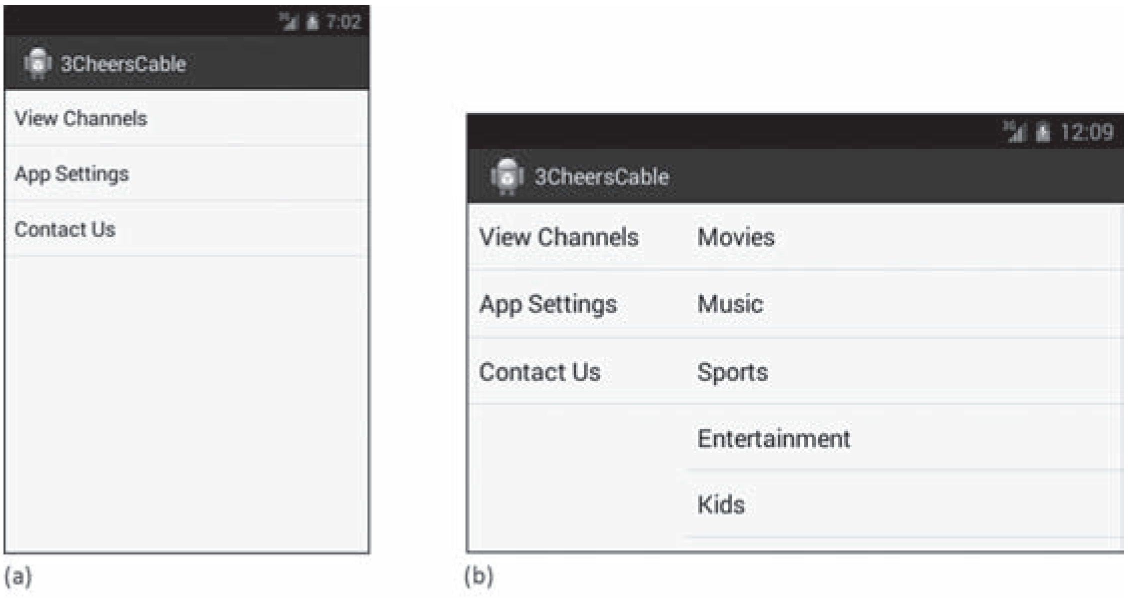


**Figure 4.14** | `SettingsActivity`: (a) Portrait and (b) landscape.

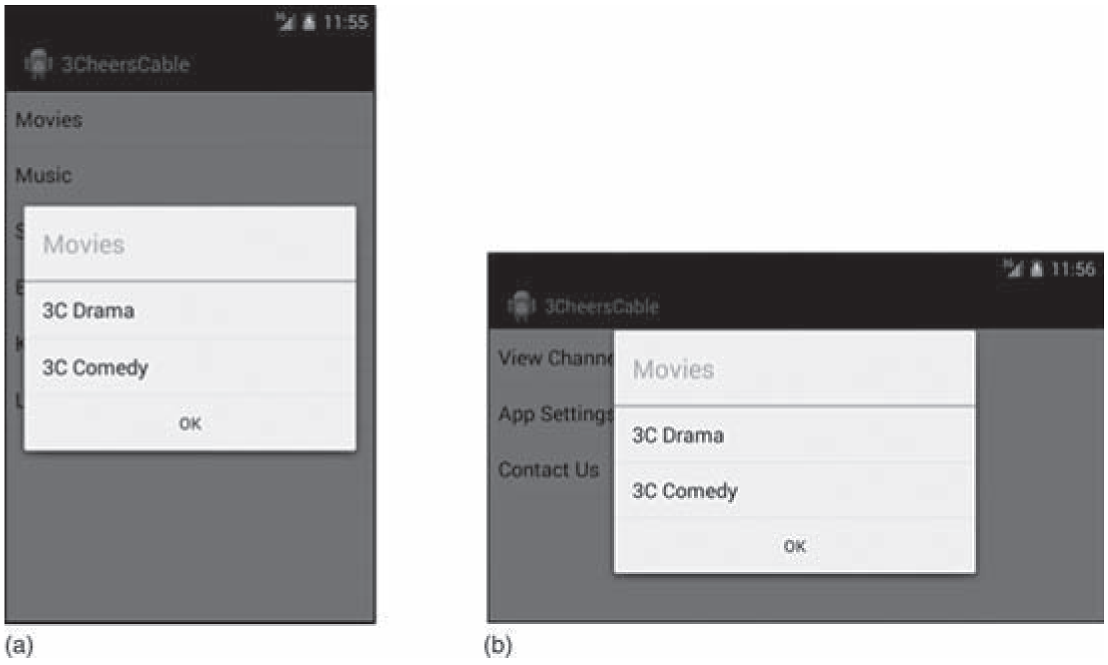
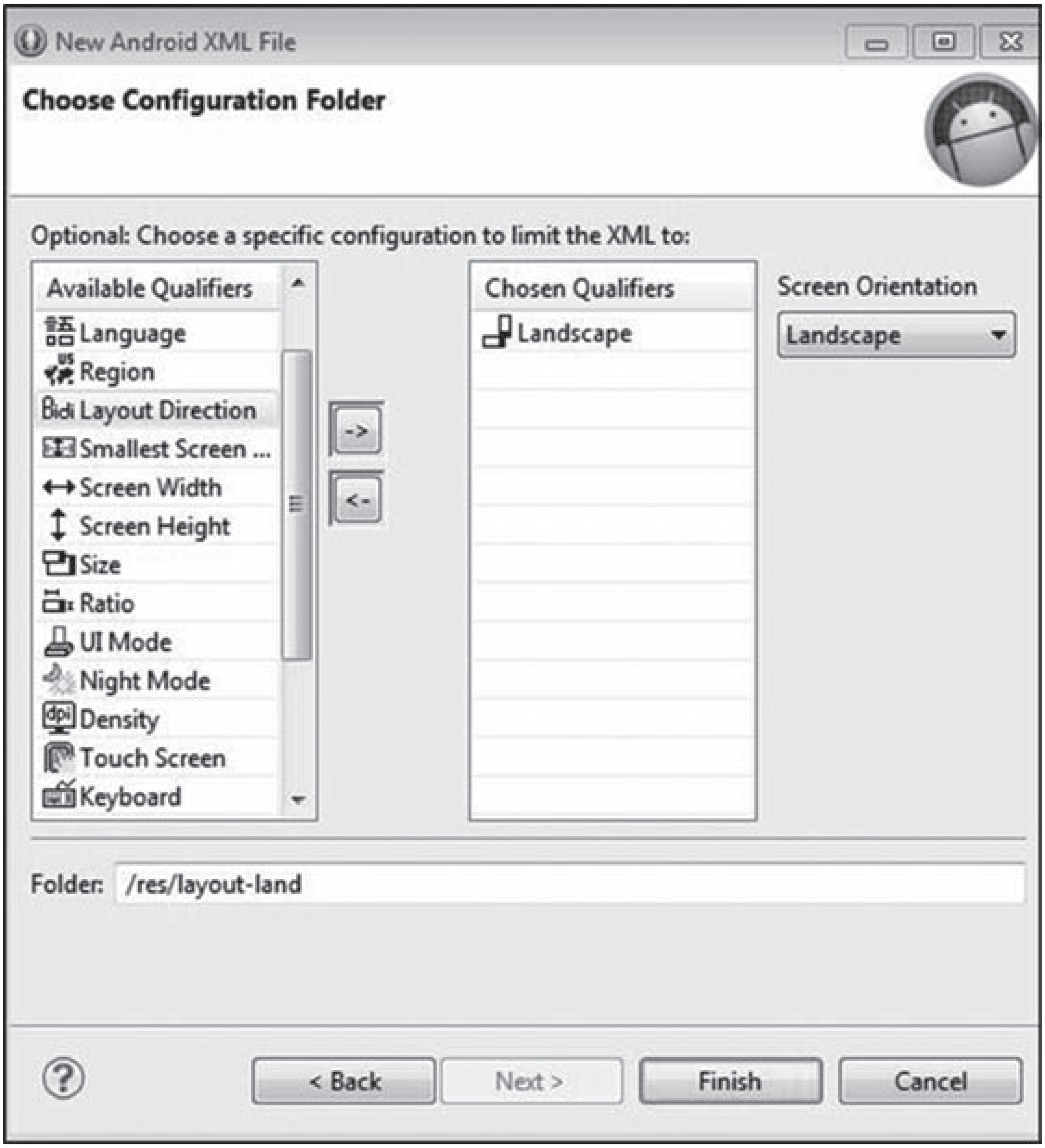


**Figure 4.15** | `ChannelsDialogFragment`: (a) Portrait and (b) landscape.

The steps for creating the `SettingsActivity` in both orientations are as follows:

**Step 1: Creating `SettingsActivity`:** While creating Activities with Fragments, we usually create two different layouts – one for the portrait mode and another for the landscape mode. We shall create a new `SettingsActivity`, which has activity_settings.xml, for the portrait mode and another layout (named activity_settings.xml too) to cater to the landscape mode. While creating the layout for the landscape mode, we have to choose *Orientation* as a *Qualifier* in the New Android XML File wizard and select *Landscape* in the *Screen Orientation* drop-down (Fig. 4.16). The layout file gets automatically created in the res\layout-land folder.



Chosen Qualifiers: Landscape

Screen Orientation: Landscape

Folder: /res/layout-land

**Figure 4.16** | New Android XML File wizard.

The activity_settings.xml for portrait mode will contain `<FrameLayout>` with id `container` as a Fragment container.

```
<FrameLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 android:id="@+id/container"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 tools:context=".SettingsActivity">
</FrameLayout>
```

The activity_settings.xml for landscape mode will contain two `<FrameLayout>` containers with id `container` and `optionsContainer`.

```
<FrameLayout
 android:layout_width="wrap_content"
 android:layout_height="match_parent"
 android:id="@+id/container"
 android:layout_weight="2">
</FrameLayout>
<FrameLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:id="@+id/optionsContainer"
 android:layout_weight="1">
</FrameLayout>
```

In the landscape mode, `OptionsListFragment` is hosted within the `<FrameLayout>` with id as `container`. The `OptionsListFragment` is a `ListFragment` that contains the options which the user can perform, such as View Channels and App Settings. On the basis of the user selection in the `OptionsListFragment`, the `CategoriesListFragment`, `SettingsFragment`, or `CustomerCareFragment` is loaded into the second `<FrameLayout>` having id as `optionsContainer`.

In the `onCreate()` method of the `SettingsActivity`, we first remove all instances of previously attached Fragments, within the two `<FrameLayout>` containers, if any. We then obtain handles for the `container` FrameLayout (portrait mode) and load the `OptionsListFragment` into it, as shown below.

```
if(getFragmentManager().findFragmentByTag("tag2")!=null)
{getFragmentManager().beginTransaction().remove(getFragmentManag
er().findFragmentByTag("tag2")).commit();}
if(getFragmentManager().findFragmentByTag("tag1")!=null)
{
getFragmentManager().beginTransaction().remove(getFragmentManage
r().findFragmentByTag("tag1")).commit();}
if(findViewById(R.id.container)!= null){
OptionsListFragment optionsFrag=new OptionsListFragment();
getFragmentManager().beginTransaction().replace(R.id.container,
optionsFrag,"tag1").commit();}
```

To monitor the selection of options in the `OptionsListFragment` and selectively load Fragments in the `optionsContainer`, the `SettingsActivity` implements `OnOptionSelectedListener` and overrides its `onOptionSelected(int option)` method.

```
public void onOptionSelected(int option) {
  Fragment fragment;
  switch(option){
    case 0: fragment=new CategoriesListFragment();
    break;
    case 1: fragment=new SettingsFragment();
    break;
    case 2: fragment=new CustomerCareFragment();
    break;
    default: fragment=new CategoriesListFragment();
    break;
  }
}
```

These Fragments have to be loaded into designated containers (`container` in portrait mode and `optionsContainer` in landscape mode) in the `SettingsActivity`.

```
if(findViewById(R.id.optionsContainer)!= null){
getFragmentManager().beginTransaction().replace(R.id.optionsCont
ainer, fragment, "tag2").commit();}
else if(findViewById(R.id.container)!= null){
getFragmentManager().beginTransaction().replace(R.id.container,
fragment, "tag2").addToBackStack(null).commit();}
```

We have used the `addToBackStack()` method here before `commit()` to add the Fragment to the back stack, so that whenever the user presses the Back key, the Fragment is loaded from the back stack.

**Step 2: Creating `OptionsListFragment` and `CategoriesListFragment`**: Let us now create `OptionsListFragment` and `CategoriesListFragment` classes. Both these classes are created by extending the `ListFragment` class. The `ListFragment` is a specialized class for dealing with Fragments containing only the ListView.
In the `onCreate()` method of the `OptionsListFragment`, an ArrayList – `optionsList` – is populated from a String array resource – `optionsList` – in strings.xml. The ListView in the Fragment is set to display these values using the `setListAdapter()` method.

```
ArrayList<String> optionsList=new ArrayList<String>();
Collections.addAll(optionsList,getResources().getStringArray(R.a
rray.optionsList));
ArrayAdapter<String> adapter=new
ArrayAdapter<String>(getActivity(),
android.R.layout.simple_list_item_1, optionsList);
setListAdapter(adapter);
```

On clicking of any option in the list, the selected option is sent back to the `SettingsActivity` so that the corresponding Fragment can be loaded. To do this, we create an interface – `OnOptionSelectedListener` – with a method – `onOptionSelected(int option)`.

```
public interface OnOptionSelectedListener{
  public void onOptionSelected(int option);
}
```

In the `onAttach()` method of the `OptionsListFragment`, an instance of the `Settings-Activity` – `optionsCallback` – is obtained.

```
try{
  optionsCallback=(OnOptionSelectedListener) activity;
}
catch (Exception e) {
  throw new ClassCastException(activity.toString() + " must
implement OnOptionSelectedListener");
}
```

On click of a list item, we send the position of the selected item to the parent Activity (`SettingsActivity`) by invoking the `onOptionSelected(int position)` method.

```
public void onListItemClick(ListView l, View v, int position,
long id) {
 optionsCallback.onOptionSelected(position);
 getListView().setItemChecked(position, true);
}
```

The ListView in the `CategoriesListFragment` is to be populated with channel categories present in the `categoriesList` String array resource.

```
optionsList=new ArrayList<String>();
Collections.addAll(optionsList,
getResources().getStringArray(R.array.categoriesList));
ArrayAdapter<String> adapter=new
ArrayAdapter<String>(getActivity(),
android.R.layout.simple_list_item_1, optionsList);
setListAdapter(adapter);
```

On selection of any item in the ListView, the selected item is passed on to the `ChannelsDialogFragment` using the `setArguments()` method.

```
public void onListItemClick(ListView l, View v, int position,
long id) {
   Bundle args=new Bundle();
   args.putString(CATEGORY_NAME, optionsList.get(position));
   ChannelsDialogFragment fragment=new ChannelsDialogFragment();
   fragment.setArguments(args);
   fragment.show(getFragmentManager(), CATEGORY_NAME);
}
```

**Step 3: Creating `ChannelsDialogFragment`:** The `ChannelsDialogFragment` extends the `Dialog Fragment` class used to create dialogs within Fragments. It shows the list of channels belonging to a selected category in a dialog. The category selected in the `CategoriesListFragment` is received here using the `getArguments()` method, and based on the selection, corresponding string array resources are displayed in an AlertDialog.

```
public Dialog onCreateDialog(Bundle savedInstanceState) {
   AlertDialog.Builder builder=new
   AlertDialog.Builder(getActivity());
   String title=
   getArguments().
   getString(CategoriesListFragment.CATEGORY_NAME);
   int itemsId;
   if(title.equalsIgnoreCase("Movies")){
     itemsId=R.array.Movies;
   }
   else if(title.equalsIgnoreCase("Music")){
     itemsId=R.array.Music;
   }
   else if(title.equalsIgnoreCase("Sports")){
     itemsId=R.array.Sports;
   }
   else if(title.equalsIgnoreCase("Lifestyle")){
     itemsId=R.array.Lifestyle;
   }
   else if(title.equalsIgnoreCase("Kids")){
     itemsId=R.array.Kids;
   }
   else{
     itemsId=R.array.Entertainment;
   }
   builder.setTitle(title).setItems(itemsId, new
   OnClickListener() {
       @Override
       public void onClick(DialogInterface arg0, int arg1) {
       }
   }).setNegativeButton("OK", new OnClickListener() {
       @Override
       public void onClick(DialogInterface arg0, int arg1) {
       }
   });
   return builder.create();
 }
```

## 4.10 ACTION BAR

An action bar is a dedicated functional strip on top of the screen that provides exclusive space for displaying varieties of global functionality required in an app such as navigation tabs, search, and user settings. It is up to the developer to populate action bar with more functionalities that deemed fit to be accessed across the app.

Figure 4.17 shows a typical action bar. Android runtime automatically manages its placement based on the device screen size and orientation. As shown in the figure, an action bar typically consists of the following elements:

1. App branding (icon and name).
2. Navigation tabs that provide a better way of navigating between the Fragments in a single Activity.
3. Action items such as zoom in and zoom out; Action views such as search in this example that provide a mechanism for the user to perform common functionalities inside an app.
4. Overflow button that holds extra menu options that are not visible by default in an action bar. The extra options surface up when the overflow button is tapped.

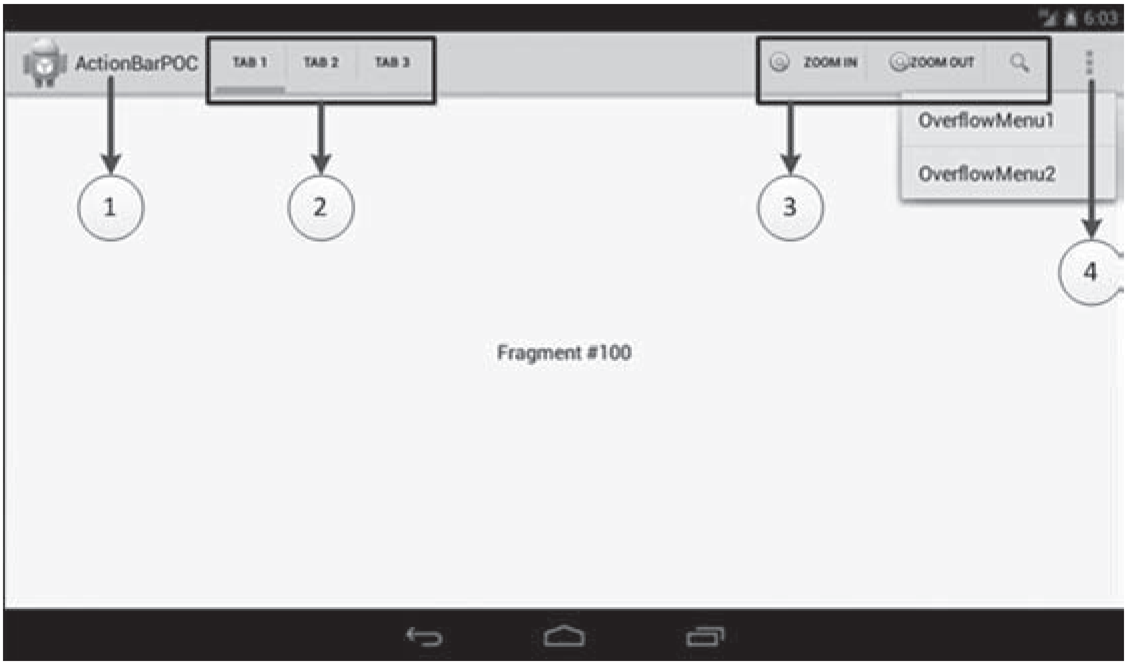


**Figure 4.17** | Action bar.

Having understood the basic layout of the action bar, let us now implement the action bar. An action bar is available by default in any app, and it contains the app icon and name. This is because `android:theme` attribute in `<application>` tag is set to `Theme.Holo`, as shown in Snippet 4.48. Action bar can be removed by setting the value of `android:theme` to `Theme.NoTitleBar`.

```
...
<application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.Holo">
...
```

**Snippet 4.48** | AndroidManifest.xml file with theme set as `Theme.Holo`.

Themes are used to maintain consistent look and feel across the app. `Theme.Holo` is the default theme in Android Jelly Bean. We can further populate the action bar with navigation tabs, action views, action items, and overflow button.

Navigation tabs provide a way of switching between multiple Fragments in the same Activity. Snippet 4.49 depicts how navigation tabs are implemented in an action bar. In Line 7 of Snippet 4.49, we get an instance of `ActionBar` using the `getActionBar()` method. To enable the action bar to display navigation tabs, we use the `setNavigationMode()` method of the `ActionBar` as shown in Line 8. The `ActionBar.NAVIGATION_MODE_TABS` constant is supplied as a parameter to the `setNaviga-tionMode()` method to make the action items appear as tabs in the action bar. In Lines 9–11, we create three tabs using the `newTab()` method. Additionally, we also set the text to be displayed on the tab (using the `setText()` method) and the listener to be triggered when a user selects the tab (using the `setTabListener()` method). When we pass `this` as a parameter to the `setTabListener()`, it means that the Activity is going to implement the event handler to handle the user action on the tab. This is another way of achieving event handling in Android unlike what we have already learnt in Section 4.4.2 of this chapter. In Lines 12–14, we use the `addTab()` method to add the tabs created in Lines 9–11 to the action bar.

```
1  public class ActionBarPOCActivity extends Activity implements
   TabListener {
2
3  @Override
4  public void onCreate(Bundle savedInstanceState) {
5    super.onCreate(savedInstanceState);
6    setContentView(R.layout.main);
7    ActionBar actionBar=getActionBar();
8    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
9    Tab tab1=actionBar.newTab().setText(“Tab 1”).
     setTabListener(this);
10   Tab tab2=actionBar.newTab().setText(“Tab 2”).
     setTabListener(this);
11   Tab tab3=actionBar.newTab().setText(“Tab 3”).
     setTabListener(this);
12   actionBar.addTab(tab1);
13   actionBar.addTab(tab2);
14   actionBar.addTab(tab3);
15  }
16  @Override
17  public void onTabSelected(Tab tab, FragmentTransaction ft) {
18   int position=tab.getPosition();
19   Fragment fragmentToBeShown;
20   switch (position) {
21   case 0:
22   fragmentToBeShown=new Fragment1();
23   break;
24   case 1:
```

```
25   fragmentToBeShown=new Fragment2();
26   break;
27   case 2:
28   fragmentToBeShown=new Fragment3();
29   break;
30   default:
31   fragmentToBeShown=null;
32   break;
33  }
34  ft.replace(R.id.fragmentContent, fragmentToBeShown);
35 }
36 }
```

**Snippet 4.49** | Adding navigation tabs to an `ActionBar`.

The layout of this Activity contains a `FrameLayout` with id `fragmentContent`. This is the container for Fragments in this Activity. In the `onTabSelected()` method (Lines 17–35), we obtain the position of the selected tab and replace the Fragment in the container based on the user selection. `Fragment1`, `Fragment2`, and `Fragment3` are three Fragments, each containing a TextView. The layouts of these Fragments have been designed, and the classes for these have been created just like how we did earlier in Section 4.8.1 of this chapter. In Line 34, the `replace()` method of `FragmentTransaction` is called to replace an existing Fragment in the container with a new one.

Action views and items provide a mechanism for the user to perform common functionalities inside an app. To create them in the action bar of an app, we first have to declare a menu resource. Menu resources are located in the res\menu folder in the project structure of the app. To create a menu resource XML file, right click on *menu* folder, select *New → Android XML File*, and provide the appropriate values in the New Android XML File wizard (shown earlier in Fig. 4.9). Editing of this XML file can be done using either a visual XML file editor or a simple text-based editor. Snippet 4.50 shows the definition of the menu XML file.

```
<menu
  xmlns:android=“http://schemas.android.com/apk/res/android”>

  <item
   android:id=“@+id/menu_zoomin”
   android:icon=“@android:drawable/btn_plus”
   android:showAsAction=“ifRoom|withText”
   android:title=“Zoom In”/>
  <item
   android:id=“@+id/menu_zoomout”
   android:icon=“@android:drawable/btn_minus”
   android:showAsAction=“ifRoom|withText”
   android:title=“Zoom Out”/>
  <item
   android:id=“@+id/menu_search”
```

```
15   android:actionViewClass="android.widget.SearchView"
16   android:icon="@android:drawable/ic_menu_search"
17   android:showAsAction="always"
18   android:title="Search"/>
19  <item
20   android:id="@+id/item1"
21   android:icon="@drawable/ic_launcher"
22   android:title="OverflowMenu1"/>
23  <item
24   android:id="@+id/item2"
25   android:title="OverflowMenu2"/>
26  </menu>
```

**Snippet 4.50** | Menu XML file.

In Snippet 4.50, the menu contains a total of five items. The first item (Lines 3–7) and second item (Lines 8–12) represent the zoom-in and zoom-out buttons (see Fig. 4.17). These items have `android:icon`, `android:title`, and `android:showAsAction` attributes among other attributes. The `android:icon` attribute is used to set an icon to the menu item when it is being displayed. The value of the `android:title` attribute is the text that is displayed in the action bar (in this case "Zoom In" and "Zoom Out"). The value of the `android:showAsAction` attribute determines when and how the item is shown in the action bar. The fourth and fifth items (Lines 19–22 and 23–25) do not specify the `android:showAsAction` attribute. This results in the appearance of the overflow button on the action bar. These menu items are displayed only when the user taps on the overflow button in the action bar. The third item (Lines 13–18) represents a specialized widget in Android called the SearchView. The SearchView is used to add querying functionality in an app. The `android:actionViewClass` attribute defines the look and feel of the widget on the action bar.

Once the menu resource is designed, we need to inflate it in the Activity class using the `onCreateOptionsMenu()` method, as shown in Snippet 4.51.

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

**Snippet 4.51** | `onCreateOptionsMenu()` method in Activity class.

In the `onCreateOptionsMenu()` method, an instance of `MenuInflater` is obtained to inflate the menu by using the `getMenuInflater()` method. The `inflate()` method of the `MenuInflater` takes in two arguments – the menu resource to be inflated and an object of `Menu` class into which the inflated object is assigned. Once the menu is inflated, user actions on the menu are captured using the `onOptionsItemSelected()` method of the Activity. The definition of this method is shown in Snippet 4.52.

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case android.R.id.home:
           //App icon click related functionality
      case R.id.item1:
         // Fourth menu item selected
      ...
      default:
          return super.onOptionsItemSelected(item);
    }
}
```

**Snippet 4.52** | `onOptionsItemSelected()` method in Activity

The `onOptionItemSelected()` method is called with the selected menu item as parameter. On the basis of this selection, various functionalities can be implemented.

Although the action bar comes handy and is sufficient to navigate around in an app and access common functionalities, at times user requires a specific set of functionalities that are only relevant in a certain context. For example, user would like to see delete/share option after selecting an image from a gallery. This can be achieved using a *contextual action bar*. The contextual action bar is used to provide action items in the context of a view (UI element) in an Activity.

Figure 4.18 shows the contextual action bar being displayed as a result of a long press on the TextView (displaying Fragment #100). The contextual action bar in this example comprises two action items: CONTEXT ITEM1 and CONTEXT ITEM2.

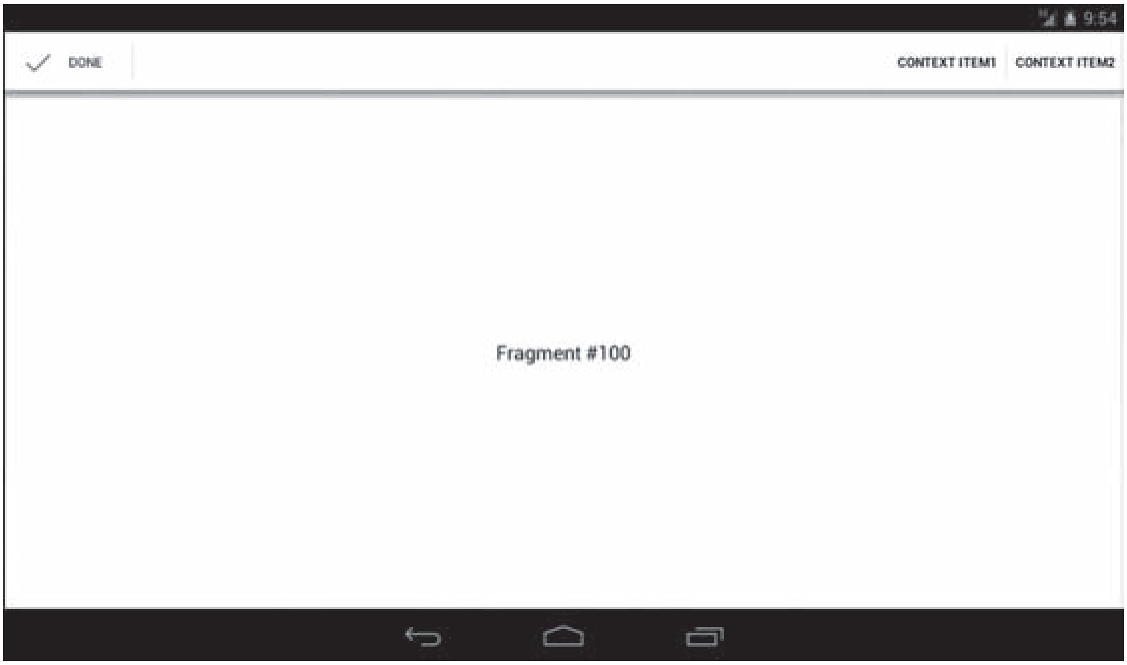


**Figure 4.18** | Contextual action bar.

Snippet 4.53 is an example of how we can create a contextual action bar.

```
public class Fragment1 extends Fragment {
ActionMode mActionMode=null;
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
container, Bundle savedInstanceState) {
    View v=inflater.inflate(R.layout.hello_world, container,
    false);
    TextView tv=(TextView) v.findViewById(R.id.text);
    tv.setOnLongClickListener(new View.OnLongClickListener() {
       public boolean onLongClick(View view) {
          if (mActionMode!= null) {
          return false;
          }

          mActionMode=getActivity().
            startActionMode(mActionModeCallback);
          view.setSelected(true);
          return true;
          }
     });
     return v;
   }

 private ActionMode.Callback mActionModeCallback=new
 ActionMode.Callback()
 {
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu
    menu) {
      MenuInflater inflater=mode.getMenuInflater();
      inflater.inflate(R.menu.context_menu, menu);
      return true;
    }
    @Override
    public boolean onActionItemClicked(ActionMode mode,
    MenuItem item) {
      switch (item.getItemId()) {
          case R.id.item1:
            //Context Menu item 1 related functionality
            return true;
          case R.id.item2:
            //Context Menu item 2 related functionality
            return true;
          default:
            return false;
39        }
40      }
41    }
42  }
```

**Snippet 4.53** | Creating a contextual action bar.

In Line 7 of Snippet 4.53, an `onLongClickListener` is set on the TextView (displaying Fragment #100). In Line 13, inside the `onLongClick()` event handler, the current Activity is fetched using `getActivity()` method, on this, an `ActionMode` is enabled using the `startActionMode()` method. ActionMode refers to the replacement of UI elements on screen in a specific context. The `startActionMode()` method accepts an object of `ActionMode.Callback`. Thus, when the `ActionMode` is enabled, the methods of `ActionMode.Callback` object are called. The definition of the `ActionMode.Callback` object supplied to the `startActionMode()` method in Line 13 is as shown from Lines 21–41. The `onCreateActionMode()` method (Line 23) is invoked when the `ActionMode` is started. In this method, we inflate a menu resource that contains CONTEXT ITEM1 and CONTEXT ITEM2 as items. As soon as the `ActionMode` is started, the contextual action bar appears on the screen (see Fig. 4.18). To capture the selection of user on the contextual action bar, the `onActionItemClicked()` method (Lines 29–39) is implemented. This method is called with the selected menu item as argument, and on the basis of the selection, various functionalities can be implemented.

## 4.11 LET'S APPLY

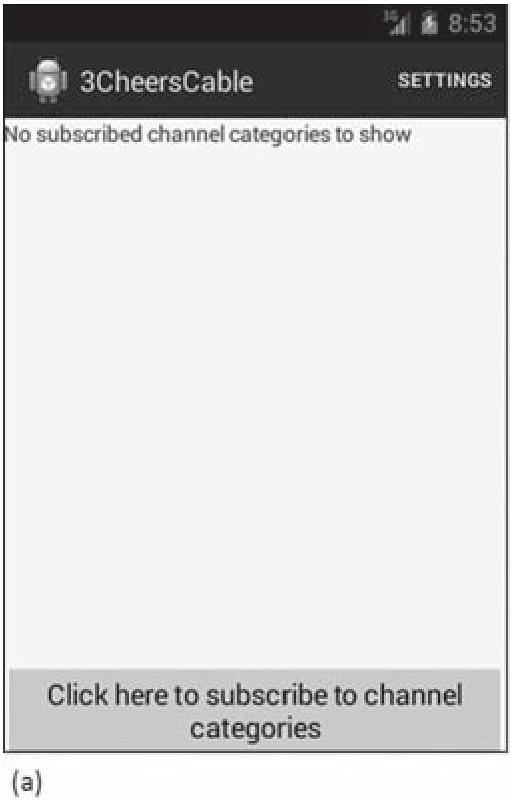
Now that we have learnt the use of action bar as an integral component in building app UI, let us add an action bar into the 3CheersCable app that we have been building in the previous sections.

Until now, we have explored the subscription facility of the 3CheersCable app where the user will be able to subscribe to different channel categories from a list, and the settings page of the app where the user will be able to traverse among various settings.

In this section, we will integrate both these functionalities in the app, using the action bar. To achieve this we have to create an action item – SETTINGS – in the `MainActivity` (see Fig. 4.19), tapping which a user is navigated to the `SettingsActivity` created in Section 4.9. The `OptionsListFragment` will now be replaced with navigation tabs in the `SettingsActivity`.

The steps for adding the SETTINGS action item in the `MainActivity`, and creating navigation tabs in the `SettingsActivity` are as follows:

**Step 1: Inflating `MainActivity` with menu resource:** Override the `onCreateOptionsMenu()` method in the `MainActivity` to inflate the `activity_main` menu resource. This ensures that the SETTINGS action item is visible in the action bar of the `MainActivity`. To enable navigation from the `MainActivity` to the `SettingsActivity` on click of the action item, the `onOptionsItemSelected()` method must be overridden. In the `onOptionsItemSelected()` method, the `startActivity()` method is invoked to launch the `SettingsActivity`.


3CheersCable
SETTINGS
No subscribed channel categories to show
Click here to subscribe to channel categories
(a)

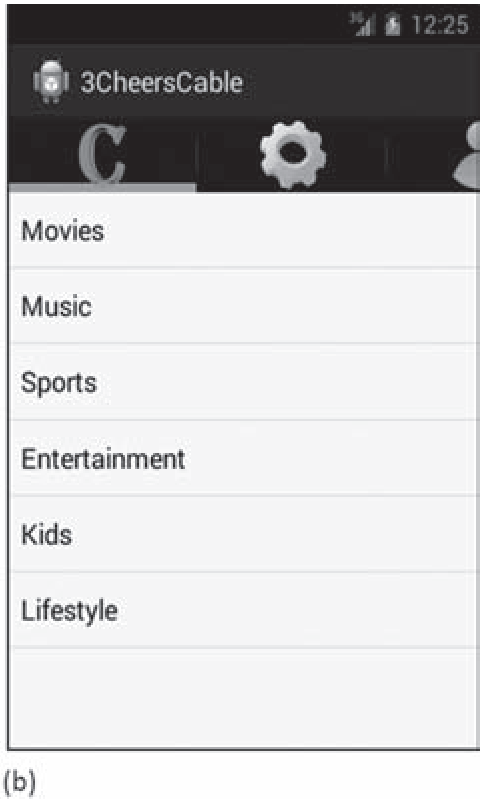


**Figure 4.19** | Settings: (a) `MainActivity` and (b) `SettingsActivity`.

```
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.activity_main, menu);
  return true;
}
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
  case R.id.menu_settings:
  Intent intent = new Intent(this, SettingsActivity.class);
  startActivity(intent);
  return true;
  default:
  return false;
 }
```

**Step 2: Modifying `SettingsActivity`:** The `SettingsActivity` will now be modified, replacing the item list from the `OptionsListFragment`, as navigation tabs in action bar.

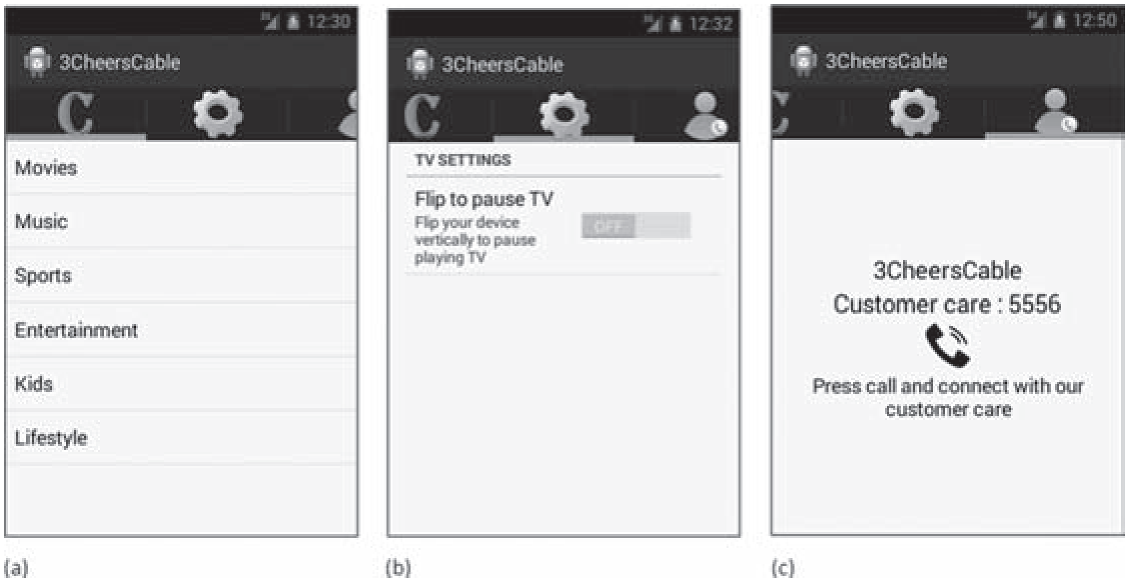
```
ActionBar actionBar=getActionBar();
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
```

```
Tab tab=actionBar.newTab().setIcon(R.drawable.category)
.setTabListener(new
TabListener<CategoriesListFragment>(this,"Categories",
CategoriesListFragment.class));
actionBar.addTab(tab);
tab=actionBar.newTab().setIcon(R.drawable.settings)
.setTabListener(new TabListener<SettingsFragment>(this,
"Settings", SettingsFragment.class));
actionBar.addTab(tab);
tab=actionBar.newTab().setIcon(R.drawable.custcare)
.setTabListener(new TabListener<CustomerCareFragment>(this,
"CustomerCare", CustomerCareFragment.class));
actionBar.addTab(tab);
    if(savedInstanceState!= null){
     actionBar.setSelectedNavigationItem
    (savedInstanceState.getInt(SELECTED_TAB, 0));
    }
```

The `TabListener` is a user-defined class that is responsible for loading the corresponding Fragment onto the `SettingsActivity` on selection of a tab. For example, when the tab corresponding to View Channels is selected, the `CategoriesListFragment` is loaded. We override the `onTabSelected()` and `onTabUnselected()` methods of `TabListner` to attach and detach the corresponding Fragments in the `SettingsActivity`.

```
public void onTabSelected(Tab arg0, FragmentTransaction arg1) {
  fragment=activity.getFragmentManager().findFragmentByTag(tag);
  if(fragment==null){
     fragment=Fragment.instantiate(activity,
     fragClass.getName());
     arg1.add(android.R.id.content,fragment, tag);}
  else{
     arg1.attach(fragment);}
}
public void onTabUnselected(Tab arg0, FragmentTransaction arg1)
{
    if(fragment!= null){
    arg1.detach(fragment);
    FragmentManager manager=activity.getFragmentManager();
    if(manager.getBackStackEntryCount()>0){
        manager.popBackStack(); }
    }
}
```

The result of selection of navigation tabs in the `SettingsActivity` is shown in Fig. 4.20. The `SettingsFragment` and the `CustomerCareFragment` are the other Fragments that need to be created.



Flip to pause TV
Flip your device vertically to pause playing TV

3CheersCable
Customer care : 5556
Press call and connect with our customer care

**Figure 4.20 (a)** `CategoriesListFragment`, (b) `SettingsFragment`, and (c) `CustomerCareFragment`.

## SUMMARY

This chapter has introduced the design and development of dynamic UI in an Android app using Activities and Fragments. It has explored the nitty-gritties of UI resources such as layouts, string, and image resources along with some commonly used UI elements. It has also dealt with event-handling paradigm that enables making the UI functional.

The chapter has further delved into interaction among Activities with a focus on navigation and data exchange between them. It has emphasized global navigation using the action bar. It has also explored UI nuances for larger smart devices such as tablets.

The chapter has also brought out the application of the concepts learnt so far by implementing the 3CheersCable reference app in "Let's Apply" sections.

## REVIEW QUESTIONS

1. Illustrate Activity life-cycle states and respective callback methods.
2. Define layouts with examples.
3. Explain event-handling paradigm with the help of an UI element.
4. Define the procedure to navigate between Activities and exchange data between them.
5. Outline the steps to build Fragments.
6. List down the elements of an action bar.

## FURTHER READINGS

1. Activity and its life cycle : http://developer.android.com/guide/components/activities.html
2. App resources: http://developer.android.com/guide/topics/resources/index.html
3. App user interface: http://developer.android.com/guide/topics/ui/index.html
4. Fragments: http://developer.android.com/guide/components/fragments.html

5

# App Functionality – Beyond UI

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Design and implement long-running tasks using Threads, AsyncTask, and Services.
- Design and implement notifications.
- Design and implement components that respond to system events.
- Utilize telephony and SMS services.

## 5.1 INTRODUCTION

In Chapter 4, we have explored the nuances of building a functional user interface (UI) of an Android app wherein event handlers play a key role. They provide a mechanism to implement UI-specific app functionality such as what should happen when a button is tapped, which Activity should trigger when an action item is selected in an action bar, or what should happen after a user has completed entering text into an edit text. All such functionalities are typically part of the event-handling logic related to an app UI.

Now, what if we have to support functionality that is beyond the app UI – functionalities such as downloading files from network, playing audio, and processing large data or complex database queries? These functionalities typically happen behind the scenes, and consume longer time (may be in minutes or hours) to execute, needing them to be dealt with differently.

The reason is quite obvious. Each app has a single thread called *main* thread on which all its components, including event callbacks, execute in a sequential manner. If a user interacts with an app while the *main* thread is already busy running the previous functionality, and if it takes more than 5s to respond, then Android runtime will pop up the dreaded Application Not Responding (ANR) dialog prompting the user to either wait or force close the app. Therefore, an app functionality that consumes more than 5s should be considered as a long-running task, and should be tackled by offloading it from the *main* thread to ensure app responsiveness.

There are several ways of offloading a long-running task from the *main* thread. The most primitive way is to spawn a new thread to carry out such tasks. Android has also provided an Application Programming Interface (API) called `AsyncTask`, which enables a cleaner way to do multithreading. Android also provides Service, one of the key building blocks that can be used to carry out long-running tasks. All these ways of dealing with functionality beyond UI find their utility in different scenarios, and have their own pros and cons. Let us now explore them in detail in this chapter, starting with threads. Besides this, we will delve into notifications, Intents, Broadcast Receivers, and telephony services in this chapter.

## 5.2 THREADS

In Android, working with threads is exactly similar to how we do it in Java. Just to recall, to spawn a thread, we create a `Runnable` (or a `Thread`) instance and override its `run()` method. This method contains the functionality that we want to offload. To start a thread, we need to invoke the `start()` method on the `Runnable` instance (or `Thread` instance).

Let us now explore a scenario where an Activity, as depicted in Fig. 5.1, allows a user to start, stop, and reset a counter. When the counter is running, its current value has to be continuously displayed in a TextView.

If we implement the increment of this counter in the `onClick()` method in a usual way and start the counter, subsequent taps on stop or reset may result in ANR. This is because, as the counter gets started, the increment logic may hog the *main* thread for more than 5 seconds.

To overcome this problem, we need to offload the counter increment logic to a separate thread, as depicted in Snippet 5.1. On click of the *Start Counter* button, a new thread is defined as shown in Lines 5–23. This new thread accepts a `Runnable` instance as argument and executes its `run()` method. The thread is started using the `start()` method at Line 23. The `run()` method is as defined in Lines 7–22. In this method, a while loop runs as long as the `keepCounting` variable holds the value `true`. At the end of each second, the `count` variable's value is incremented.

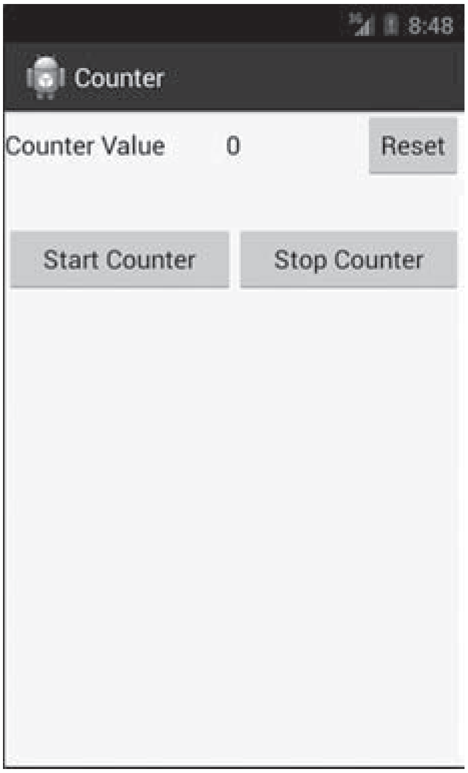


**Figure 5.1** | Counter Activity.

```
public void onClick(View arg0) {
    switch (arg0.getId()) {
    case R.id.startCounterButton:
     keepCounting = true;
     new Thread(new Runnable() {
      @Override
      public void run() {
       while (keepCounting) {
          try {
            Thread.sleep(1000);
          } catch (InterruptedException e) {
             e.printStackTrace();
          }
          count++;
          displayValue.post(new Runnable() {
          @Override
          public void run() {
            displayValue.setText(count + “”);
          }
          });
      }
22     }
23       }).start();
24     break;
25     case R.id.stopCounterButton:
26       keepCounting = false;
27     break;
28     case R.id.resetButton:
29       count = 0;
30       displayValue.setText(count + "");
31     break;
32    }
33  }
```

**Snippet 5.1** | Counter functionality in a thread.

We can now observe that after incrementing the `count` at Line 14, the `displayValue` is not updated directly on the non-UI thread that we have created, rather a `post()` method is invoked that has the `Runnable` instance as the parameter (Line 15). This method ensures that the UI update operation is executed on the *main* thread, as Android mandates that the UI of an app should be updated in the *main* thread only.

There is more to it than meets the eye. To understand this, we need to equip ourselves with the `Handler` and `Looper` concept in Android. To access the *main* thread from a non-UI thread, we first need to get hold of the *main* thread. This is accomplished by using a `Handler`. A `Handler` is an object that always refers to the thread that created it. The `Handler` of the *main* thread allows other threads to post new tasks (`Runnable` instances) to a message queue maintained by the *main* thread. This message queue is used to line up tasks, including UI updates, as depicted in Fig. 5.2. Tasks in this queue are executed on a

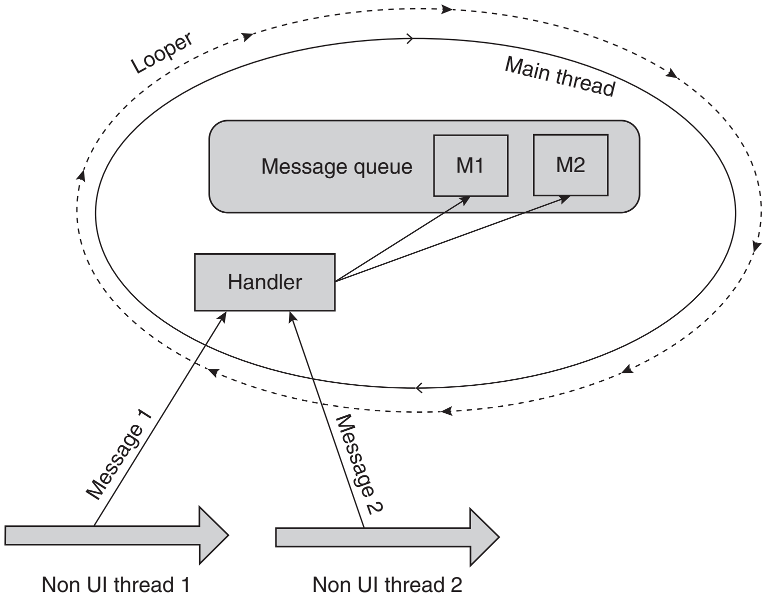


**Figure 5.2** | Handler and Looper.

first-come, first-served basis. Once, the message queue is exhausted, the *main* thread waits for a new task(s) in an endless loop. This endless looping behavior of the *main* thread is induced by a `Looper`. When the `post()` method is invoked, the non-UI thread post a task to the message queue of the *main* thread by obtaining the `Handler` of the *main* thread.

Implementing multithreading may become tedious (at times cumbersome) from a developer's perspective, especially if multiple threads are involved in updating the UI (on *main* thread). To deal with such scenarios, Android provides a cleaner mechanism called AsyncTask.

## 5.3 ASYNCTASK

An AsyncTask has to deal with two key aspects: execute the long-running task in the background (in a separate thread) and update the *main* thread as the task progresses/completes. To achieve this, the `AsyncTask` class provides two key callback methods: `doInBackground(Params params…)` and `onProgressUpdate(Progress… progress)`. The former method gets executed on a separate thread in the background, different from the *main* thread, and is an ideal place for long-running tasks. The latter method gets executed on the *main* thread, and is suited to update the progress of the background task on UI. This asynchronous communication between the `doInBackground(Params params…)` and the `onProgressUpdate(Progress… progress)` methods is achieved through the `publish–Progess(params)` method call inside `doInBackground(Params params…)` method.

Let us now implement the Counter example discussed in Section 5.2 through an `AsyncTask`, as depicted in Snippet 5.2. Typically, an `AsyncTask` is implemented as a nested class of an Activity whose UI it needs to update. An `AsyncTask`, called `CounterAsyncTask`, is created in Line 1. The different methods related to an `AsyncTask` are overridden, viz. `doInBackground()`, `onProgressUpdate()`, and `onPostExecute()`. In the `doInBackground()` method (Lines 3–15), a while loop is executed as long as the `keepCounting` variable holds the value `true`, increasing the value of `count` for every second of its execution. At the end of each second, the value of `count` is published to the `onProgressUpdate()` method using the `publishProgress()` method as depicted in Line 9.

```
1  class CounterAsyncTask extends AsyncTask<Integer, Integer,
   Integer> {
2  @Override
3  protected Integer doInBackground(Integer... arg0) {
4     count = arg0[0];
5     while (keepCounting) {
6      try {
7       Thread.sleep(1000);
8       count++;
9       publishProgress(count);
10     }catch (InterruptedException e) {
11      e.printStackTrace();
12     }
13    }
14    return count;
15  }
16  @Override
17  protected void onProgressUpdate(Integer... values) {
18     super.onProgressUpdate(values);
19     displayValue.setText(values[0] + "");
20  }
21  @Override
22  protected void onPostExecute(Integer result) {
23      super.onPostExecute(result);
24      displayValue.setText(result + "");
25  }
26  }
```

**Snippet 5.2** | Counter functionality in an AsyncTask.

In the `onProgressUpdate()` method (Lines 17–20), the text in `displayValue` (TextView displaying `count` value) is set to the value of `count` received from the `doInBackground()` method. As the `onProgressUpdate()` is called for every second the control remains inside the while loop, the TextView is periodically updated. In the `onPostExecute()` method (Lines 22–25), the `displayValue` TextView is set to the final value of `count` as the counter stops. This method gets executed on the UI thread when `doInBackground()` completes its execution. Snippet 5.3 shows the modification required in Snippet 5.1 to start the `CounterAsyncTask` (Line 3).

```
1  case R.id.startCounterButton:
2  keepCounting = true;
3  new CounterAsyncTask().execute(count);
4  break;
```

**Snippet 5.3** | Modified code of the `onClick()` method.

Let us now apply the key concepts of AsyncTask in the 3CheersCable app.

## 5.4 LET'S APPLY

In the 3CheersCable app, which we have been building as part of our previous "Let's Apply" sections in Chapter 4, we have provided the user with a subscribe functionality where he or she is able to subscribe to channel categories desired.

In Chapter 4, when we were building the app UI, we had a predefined set of channel categories and a set of channels within each category. In a fully functional app, we must realize that these values are subject to change and, therefore, always obtained from a data source. The data source may be either a local database or a remote Web service.

Retrieving such data from an external data source usually falls under the category of long-running task and hence must not be performed on the *main* thread. In our app, we shall use an `AsyncTask` called `WebServiceHitter` to fetch the channel categories from an exposed Web service.

**Step 1: Creating `WebServiceHitter`:** Create the `WebServiceHitter` class that is a subclass of `AsyncTask` with a parameterized constructor, taking in an instance of `WebServiceFinishedListener` as parameter.

```
public class WebServiceHitter extends AsyncTask<Object, Void,
WebResponse> {
  WebServiceFinishedListener webServiceFinishedListener = null;
  public WebServiceHitter(WebServiceFinishedListener
  webServiceFinishedListener){
   this.webServiceFinishedListener = webServiceFinishedListener;
}
```

The `WebServiceFinishedListener` is a user-defined interface with two methods: `onNetworkCallComplete()` and `onNetworkCallCancel()`. The methods of this interface have to be implemented in the calling Activity `(SubscribeActivity)` where the `AsyncTask` instance is created to initiate data fetch from the Web service.

In the `doInBackground()` method of the `WebServiceHitter`, a user-defined class – `WebResponse` – is used. This class contains `response` and `responseCode` as instance variables with setter and getter methods for each. We make use of the `HttpParams` and `HttpClient` classes to create a Hypertext Transfer Protocol (HTTP) connection, and return the response obtained as a `WebResponse` object. Details of parsing the response obtained from the Web service will be discussed in Chapter 6.

```
protected WebResponse doInBackground(Object... params) {
   WebResponse webResponse = new WebResponse();
   StringBuilder jsonStr = new StringBuilder();
   HttpParams httpParameters = new BasicHttpParams();
   HttpConnectionParams.setConnectionTimeout(httpParameters,
   AppConfig.CONNECTION_TIMEOUT);
   HttpConnectionParams.setSoTimeout(httpParameters,
   AppConfig.SOCKET_TIMEOUT);
   HttpClient client = new DefaultHttpClient(httpParameters);
   try {
      HttpResponse response=
      client.execute((HttpPost)params[0]);
      BufferedReader rd = new BufferedReader(new
      InputStreamReader(response.getEntity().getContent()));
      String line = "";
      while ((line = rd.readLine()) != null) {
         jsonStr = jsonStr.append(line);
      }
      webResponse.setResponse(jsonStr.toString());
      webResponse.setResponseCode(ResponseCode.SUCCESS);
      } catch (Exception e) {
         e.printStackTrace();
         webResponse.setResponse(e.getMessage());
         webResponse.setResponseCode(ResponseCode.FAILURE);
     }
       return webResponse;
    }
```

In the `onPostExecute()` method, the `onNetworkCallComplete()` and `onNetworkCallCancel()` methods of the `WebServiceFinishedListener` are invoked based on the success of the network call.

```
protected void onPostExecute(WebResponse result) {
 super.onPostExecute(result);
 if(result.getResponseCode().compareTo(ResponseCode.SUCCESS)==0)
     webServiceFinishedListener.onNetworkCallComplete(result);
 else
     webServiceFinishedListener.onNetworkCallCancel();
}
```

**Step2: Modifying `SubscribeActivity`:** Implement the `WebServiceFinishedListener` interface in the `SubscribeActivity` and override its methods.

```
asyncTask = new
WebServiceHitter(this).execute(GenerateURLs.getPostURL(Operation
.CATEGORIES));
public void onNetworkCallComplete(WebResponse object) {

categoriesBean =
 CategoriesResponseParser.parse(object.getResponse().toString());
categoriesList = new ArrayList<String>();
categoriesIdList = new ArrayList<String>();
ArrayList<Category> categories = categoriesBean.getCategories();

for(int i=0; i<categories.size(); i++){
    categoriesList.add(categories.get(i).getName());
    categoriesIdList.add(categories.get(i).getCategoryId());
}
ArrayAdapter<String> categoriesListAdapter=new ArrayAdapter<String>
(this,android.R.layout.simple_list_item_multiple_choice,categoriesList);
categoryList.setAdapter(categoriesListAdapter);
myProgressDialog.dismiss();
 }
}
```

We have created a reference of the `WebServiceHitter` and are passing the `URL` as argument. The obtained result is parsed using `CategoriesResponseParser` – a user defined class - into an object of `CategoriesBean` – a user-defined bean class – and the obtained categories are displayed on the ListView.

Similarly, for displaying a list of channels or list of shows, we make corresponding network calls and store the data obtained into their corresponding bean classes. All the bean classes used for this purpose are created in the `com.app3c.beans` package. The classes pertaining to interaction with the Web services are created in the `com.app3c.webserviceconsumer` package.

## 5.5 SERVICE

We have just learnt that AsyncTask is used when there is a need to constantly update an app UI in an asynchronous manner. But, there might be scenarios wherein long-running tasks [such as downloading a file, playing an audio, or processing a RSS (Rich Site Summary) feed] need to run in the background, and update the user once in a while, either toward the completion of the task or when user intervention is desired. To address such scenarios, Android provides Service where we can offload the long-running task that can interact with the user using notifications.

Service is an Android component that is used for running long tasks in the background. It should not be confused with a process or a thread. It gets executed by default in the *main* thread like an Activity. However, to avoid ANR, a developer has to spawn a thread from a Service to offload a long-running task. To further simplify the job of a developer, Android also provides `IntentService`, which internally takes care of pushing a long-running task onto a separate thread.

By all means, a Service provides a straightforward option to execute long-running tasks in the background, even when the user is not interacting with the app that started the Service. Before exploring further on types of Services and their usage, let us first delve into Service states and their life-cycle methods.

### 5.5.1 States and Life-Cycle Methods

The life cycle of a Service (see Fig. 5.3) begins when a call to the `startService()` method is made in another component (such as an Activity). This triggers the creation of a Service, and the `onCreate()` method is called as a result. The Service progresses to the *running* state via a call to the `onStartCommand()` method. It continues to stay in the running state during the execution of `onStartCommand()`, and remains in the same state till the component that called the Service explicitly stops it using the `stopService()` method. A Service can also stop itself by calling the `stopSelf()` method at the end of `onStartCommand()`. In either of the cases, as a Service is being stopped, it moves to the *destroyed* state via a call to the `onDestroy()` method. In case the Service continues to remain in the running state, subsequent calls to `startService()` from other components result in the execution of `onStartCommand()` for every call.

`onCreate()` is the first method that gets executed when a Service is started for the first time. It happens only once in the life cycle of a Service. This method is well suited for initializations before other callback methods get executed. `onStartCommand()` is the place where long-running tasks are implemented. This method may get executed multiple times during the life cycle of a Service. `onDestroy()` is the best place to de-allocate the resources that were held by the Service.

Once the Service gets started, it never stops until it is explicitly stopped using the `stopService()` or `stopSelf()`. Exception to this behavior occurs when there is a system resource crunch, the Android runtime kicks in to reclaim resources by killing the Service. A developer has to be wary of both these facts,

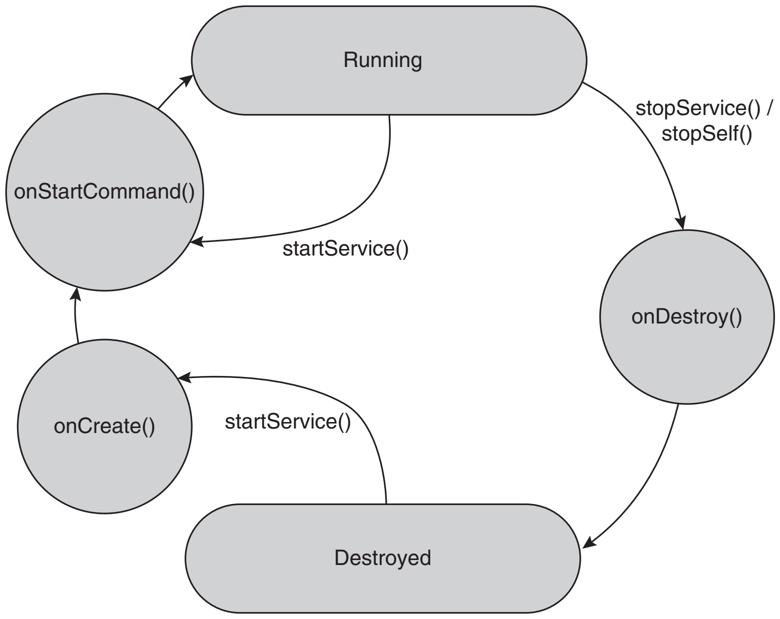


**Figure 5.3** | Service states and Life-Cycle Methods.

and should make necessary arrangements to properly stop the Service. Otherwise, it may lead to a situation where in a Service may either run forever in the background, leading to a quick battery drain, or be stopped abruptly due to resource crunch, and may never return to complete the task assigned to it.

### 5.5.2 Initiating a Service

Let us now implement a Service for the Counter scenario that we are already familiar with. The only difference here is, instead of continuously updating the counter value on UI, we are now going to log the counter values, as the counter increments in the background using Service. The Service is going to get started or stopped using an Activity, as depicted in Fig. 5.4.

The Service has a counter variable that will start incrementing as soon as it is started by tapping the *Start Counter* button. The counter will keep on incrementing till the Service is stopped by tapping the *Stop Counter* button. Because the Service runs in the background, in this scenario, it is ideal to log counter values that can be observed in the LogCat.

In Snippet 5.4, the `CounterService` class is created by extending `Service`. All life-cycle methods of the `Service` have been overridden to monitor the life cycle of the Service. In the `onStartCommand()`

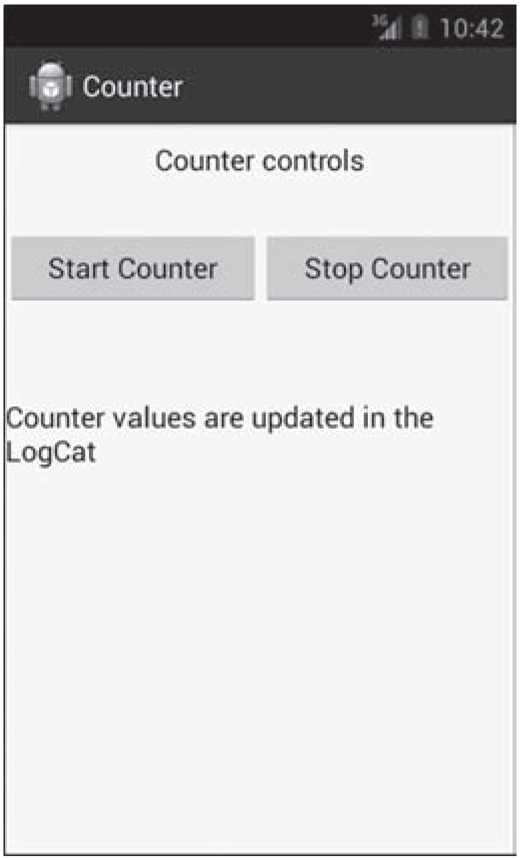


**Figure 5.4** | Activity to Interact with Service.

method (Lines 14–33), the counter logic is implemented. The Service spawns a new thread (Line 18) that keeps increasing the value of `count` for every second of its execution. The `count` value is updated in the LogCat as shown in Line 28.

```
1  public class CounterService extends Service {
2   boolean keepCounting = false;
3   int count = 0;
4   @Override
5   public IBinder onBind(Intent arg0) {
6    return null;
7   }
8   @Override
9   public void onCreate() {
10   super.onCreate();
11   Log.i(“Service Lifecycle”, “onCreate called”);
12  }
13  @Override
14  public int onStartCommand(Intent intent, int flags, int
    startId) {
15   Log.i(“Service Lifecycle”, “onStartCommand called”);
16   keepCounting = true;
17   count = 0;
18   new Thread(new Runnable() {
19    @Override
20    public void run() {
21     while (keepCounting) {
22      try {
23        Thread.sleep(1000);
24      } catch (InterruptedException e) {
25        e.printStackTrace();
26      }
27      count++;
28      Log.i(“CounterStatus,”Time elapsed since service
        started: “+ count + “ seconds”);
29    }
30    }
31    }).start();
32   return START_STICKY;
33  }
34  @Override
35   public void onDestroy() {
36    super.onDestroy();
37    Log.i(“Service Lifecycle”, “onDestroy called”);
38    count = 0;
39    keepCounting = false;
40   }
41  }
```

**Snippet 5.4** | Service implementation of the Counter.

In Line 32, an integer constant `START_STICKY` is returned by the `onStartCommand()`. This return value is crucial in determining how Android runtime treats Services (while restarting) that are terminated in case of resource crunch. `START_STICKY` ensures that the Service will always get restarted once the required resources are available, but with an empty Intent (without any data).

The other possible values are `START_NOT_STICKY` and `START_REDELIVER_INTENT`. The `START_NOT_STICKY` constant ensures that the Service will not be restarted unless it was called during the resource crunch. In this case also, it gets started with an empty Intent. The `START_REDELIVER_INTENT` constant ensures that the Service will always restart with the previous Intent.

Intent is used to start and stop the Service from other components, such as an Activity in our example. The `startService()` method accepts an `Intent` object as parameter to start the Service, as shown in Line 5 of Snippet 5.5. This `Intent` object in turn is passed as the first argument to `onStartCommand()` (Line 14, Snippet 5.4). The `stopService()` method is used to stop the Service explicitly. This in turn calls the `onDestroy()` method of the Service.

```
1  Intent intent= new Intent(this, CounterService.class);
2  public void onClick(View arg0) {
3     switch (arg0.getId()) {
4     case R.id.startCounterButton:
5           startService(intent);
6           break;
7     case R.id.stopCounterButton:
8           stopService(intent);
9           break;
10    default:
11          break;
12    }
13 }
```

**Snippet 5.5** | Modified code of the `onClick()` method to call a Service.

Every Android component needs to be registered in AndroidManifest.xml after implementation. `CounterService` is registered using `android:name` attribute of the `<service>` tag, as depicted in Snippet 5.6.

```
<manifest>
 ...
 <application>
   ...
   <service
   android:name="com.mad.servicedemo.CounterService"></service>
 </application>
</manifest>
```

**Snippet 5.6** | Registering Service in AndroidManifest.xml.

In the above example, the onus is always on the developer to spawn a thread from the Service to cater to the logic of any long-running task (incrementing the counter in this case). To take care of this mundane task, Android provides an `IntentService` API that internally pushes the long-running task onto a separate thread.

## 5.5.3 IntentService

`IntentService` is a subclass of the `Service` class. It automatically spawns a thread, usually referred to as worker thread, to cater to long-running tasks. Creating an `IntentService` is pretty straightforward. We need to create a user-defined class that extends `IntentService` and override the `onHandleIntent()` method. Implementation of other callback methods of a Service (`onBind`, `onCreate`, `onStartCommand`, and `onDestroy`) is optional.

Let us now modify the previous example of incrementing a counter in a Service, using `Intent-Service`, as depicted in Snippet 5.7. It shows the `CounterService` class as a subclass of `IntentService`. A default constructor is defined as shown in Line 4 that invokes a constructor of the parent class (`IntentService`) using the `super()` method (Line 5). A String value is passed as argument to this method, which indicates the name of the worker thread that would be spawned to handle long-running tasks.

```
1  public class CounterService extends IntentService {
2   boolean keepCounting = false;
3   int count = 0;
4    public CounterService() {
5     super("CounterService");
6    }
7    @Override
8    protected void onHandleIntent(Intent arg0) {
9     keepCounting = true;
10    count = 0;
11    while (keepCounting) {
12      try {
13       Thread.sleep(1000);
14      } catch (InterruptedException e) {
15       e.printStackTrace();
16      }
17      count++;
18      Log.i("CounterStatus", "Time elapsed since service
        started: "+ count + "seconds");
19     }
20    }
21    @Override
22    public void onDestroy() {
23     super.onDestroy();
24     Log.i("Service Lifecycle", "onDestroy called");
25     count = 0;
26     keepCounting = false;
27    }
28 }
```

**Snippet 5.7** | Counter implemented using an `IntentService`.

The `onHandleIntent()`, as described from Lines 8–20, contains the code to implement a counter that keeps increasing the value of `count` for every second of its execution. In the `onDestroy()` method, which is called when the user taps the *Stop Counter* button, the counter is stopped by changing the value of `keepCounting` variable to `false` (see Line 26).

### 5.5.4 Bound Service

So far, we have learnt that a Service is the ideal building block in the Android platform to cater to functionality related to long-running task where we need to run them in the background (frequent refreshes on UI not required). In the Counter example implemented using Service (and `IntentService`), the counter variable gets incremented and logged, until stopped by the user using the *Stop Counter* button.

However, there might be a scenario wherein an Android component (from same app or different app) may need to access the current counter value. To facilitate this, there should be a provision in the Service to expose its functionality to other Android components. This is achieved by using Bound Service. The relationship between a Service and the other component that wants to connect to the Service is that of a server and client, respectively. The Bound Service acts as a server and the other app component that intends to bind to the Service acts as a client.

In the case of Bound Services, the Service life cycle behaves differently, as represented in Fig. 5.5. It begins when a call to the `bindService()` method is made in another component (such as an Activity). This triggers the creation of a Service, and the `onCreate()` method is called. The Service progresses to the running state via a call to the `onBind()` method. When a Service is running, other components can also bind to it using the `bindService()` method. This results in multiple execution of `onBind()` for every client bound to the Service. As clients begin to discontinue using the Service by calling the `unBind-Service()` method, the `onUnbind()` method of the Service is called. After all clients have unbound from the Service, the Service can go to the destroyed state via the `onDestroy()` method.

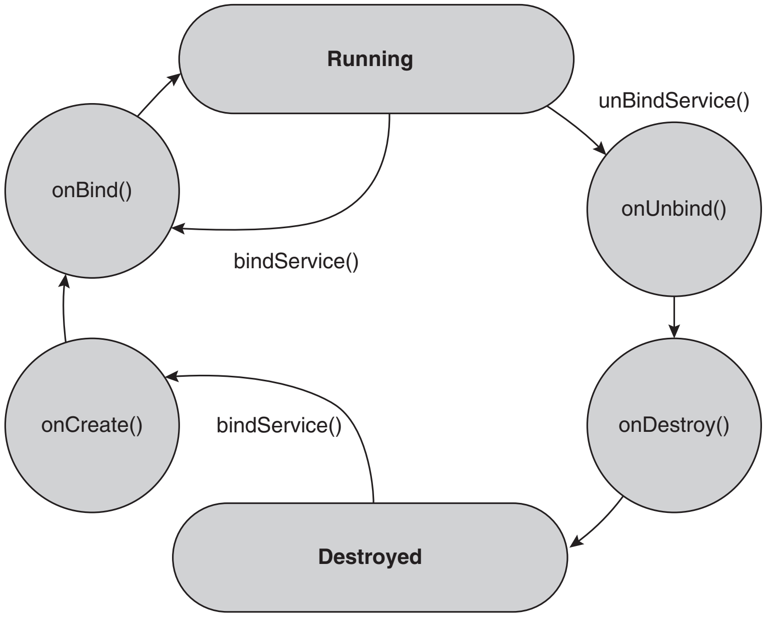


**Figure 5.5** | Bound Service States and Life-Cycle Methods.

There are two typical binding scenarios, viz., local binding and cross-app binding. Local binding is used if both the Service and the other component belong to the same app. It is implemented using a `Binder` class. Cross-app binding is used when the client component resides in a different app. It is supported either by `Messenger` API or by Android Interface Definition Language (AIDL). Let us now explore the `Binder` class and `Messenger` API to implement local and cross-app binding, respectively.

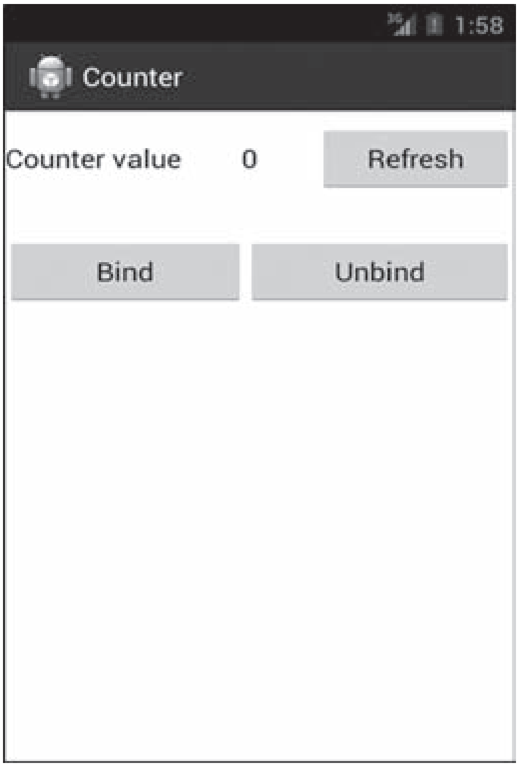


**Figure 5.6** | Client component for Bound Service.

We will now modify the previous counter example such that a client component (`CounterActivity`) binds to the `CounterService` and fetches the value of `count` as depicted in Fig. 5.6.

The `CounterService` is modified to cater to local binding as demonstrated in Snippet 5.8. Lines 5–8 depict the `onBind()` callback method that returns an `IBinder` object – `CounterServiceBinder`. A `Binder` acts as an interface between a Service and its clients. The `CounterServiceBinder` is created as an inner class of `CounterService` in Line 9. This is a subclass of the `Binder` (which in turn is a subclass of `IBinder`) and contains a user-defined `getService()` method that returns an instance of `CounterService`.

```
1  public class CounterService extends Service {
2   boolean keepCounting = false;
3   int count = 0;
4   @Override
5   public IBinder onBind(Intent intent) {
6     startCount();
7     return new CounterServiceBinder();
8   }
9   public class CounterServiceBinder extends Binder {
10    public CounterService getService() {
11         return CounterService.this;
12    }
13  }
14   private void startCount() {
15     keepCounting = true;
16     count = 0;
17     new Thread(new Runnable() {
18      @Override
19      public void run() {
20       while (keepCounting) {
21          count++;
22          try {
23            Thread.sleep(1000);
24          } catch (InterruptedException e) {
25            e.printStackTrace();
26          }
27       }
28     }
29    }).start();
30  }
31  private void stopCount() {
32    keepCounting = false;
33    count = 0;
34  }
35  public int getCount() {
36    return count;
37  }
38  @Override
39  public boolean onUnbind(Intent intent) {
40    stopCount();
41    return super.onUnbind(intent);
42  }
43  @Override
44  public void onDestroy() {
45    super.onDestroy();
46    Log.i("Service Lifecycle", "onDestroy called");
47  }
48 }
```

**Snippet 5.8** | `CounterService` modified to cater to local binding.

Furthermore, the `CounterService` class contains user-defined methods such as `startCount()`, `stopCount()`, and `getCount()`. The `startCount()` method contains the code to increment the value of `count` for every second of its execution. The `stopCount()` method is used to stop the counter and reset the `count` variable. The `getCount()` method is a public method exposed to clients to fetch the current value of `count`. The `startCount()` and `stopCount()` methods are invoked from the `onBind()` and `onUnbind()` methods, respectively (Lines 6 and 40).

When the `CounterActivity` (Snippet 5.9) binds to the `CounterService` using the `bindService()` method, as shown in Line 32, the `onBind()` method of the Service is invoked. The `bindService()` method accepts three parameters: Intent to the Service, a `ServiceConnection` object, and an integer flag.

```
public class CounterActivity extends Activity implements OnClickListener {
  Button startCounter, stopCounter, refresh;
  ...
  boolean bound = false;
  CounterService counterService = null;

  ServiceConnection counterServiceConnection = new ServiceConnection() {

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        bound = false;
    }

    @Override
    public void onServiceConnected(ComponentName arg0, IBinder arg1){
     CounterService.CounterServiceBinder counterServiceBinder=
       (CounterService.CounterServiceBinder) arg1;
     counterService = counterServiceBinder.getService();
     bound = true;
    }
   };
   @Override
   protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_counter);
    ...
    intent = new Intent(this, CounterService.class);
   }
   @Override
   public void onClick(View arg0) {
    switch (arg0.getId()) {
    case R.id.startCounterButton:
     bindService(intent, counterServiceConnection,BIND_AUTO_CREATE);
    break;
    case R.id.stopCounterButton:
     unbindService(counterServiceConnection);
    break;
    case R.id.fetchCount:
     if (bound == true) {
      displayCountValue.setText(counterService.getCount()+"");
     }
    break;
42    default:
43    break;
44     }
45   }
46  }
```

**Snippet 5.9** | Client component – `CounterActivity`.

The `ServiceConnection` object (Lines 7–20) monitors the state of the client's connection with the Service. It has two callback methods – `onServiceConnected()` and `onServiceDisconnected()` – which are triggered when the connection state changes. The `onServiceConnected()` method is called with the `IBinder` object returned by the Service from its `onBind()` method. In Lines 16 and 17, the `IBinder` object is typecasted to the `CounterServiceBinder` and used to retrieve an instance of `CounterService` (using `getService()` method).

The flags' values are used to indicate the priority that should be assigned by the Android runtime to the Service in case of resource crunch. Some of the possible values are `BIND_AUTO_CREATE` (start the Service as long as the binding exists), `BIND_ADJUST_WITH_ACTIVITY` (prioritize based on the visibility of the Activity), and `BIND_ABOVE_CLIENT` (keep the priority higher than that of the client).

The Service is unbound using the `unBindService()` method, as shown in Line 35. When the user clicks on the *Refresh* button (see Fig. 5.6), the `count` value is obtained using the `getCount()` method of `CounterService`, and is displayed in the TextView as shown in Line 39.

So far, we have seen how to bind an app component to a local Service. However, in cross-app binding scenarios, where an app component has to bind to a remote Service, a `Messenger` API is used. Let us now delve into the cross-app binding scenario using the Counter example.

In Snippet 5.10, the `CounterService` class is modified to support cross-app binding using the `Messenger` API. A `Messenger` object – `counterServiceMessenger` – is used in the `onBind()` method to return a `Binder` as shown in Line 27. This `Messenger` acts as a reference to a `Handler` object – `CounterServiceHandler`. A `Handler` is used to receive and respond to messages exchanged between components residing in different app processes, in this case between the `CounterActivity` and the `CounterService`. The `CounterServiceHandler` contains the `handleMessage()` method (Line 8), which is triggered when any message is received by the Service. Using varied integer values for the `what` variable (Line 9), different clients can send different messages to the Service.

Messages are sent using the `Message` class as shown in Lines 11, 12, and 14. In Line 11, an instance of `Message` class – `sendCount`, which is used to reply to the client component – is created using the `obtain()` method. In Line 12, the `arg1` variable of `sendCount` holds the value of `count` (obtained using `getCount()` method). In Line 14, the `sendCount` message is sent to the client using the `send()` method. The `replyTo` variable will hold a reference to the `Messenger` of the client.

```
1  public class CounterService extends Service {
2    boolean keepCounting = false;
3    int count = 0;
4    public static final int GET_COUNT=0;
5
6    class CounterServiceHandler extends Handler{
7      @Override
8      public void handleMessage(Message msg) {
9        switch(msg.what){
10       case GET_COUNT:
        Message sendCount=Message.obtain(null,GET_COUNT);
        sendCount.arg1=getCount();
        try {
           msg.replyTo.send(sendCount);
        } catch (RemoteExceptione) {
           e.printStackTrace();
        }
      }
        super.handleMessage(msg);
    }
  }
  Messenger counterServiceMessenger = new Messenger(new CounterServiceHandler());

  @Override
  public IBinder onBind(Intent intent) {
    startCount();
    return counterServiceMessenger.getBinder();
  }
  //startCount(),stopCount() and getCount() methods are retained as defined earlier

}
```

**Snippet 5.10** | `CounterService` modified to cater to cross-app binding.

Snippet 5.11 describes the `CounterActivity`, which is the client component for `CounterService`. Because the Service replies to the client with the value of `count`, the `CounterActivity` also contains a Handler – `RecieveCountHandler` – to receive the message, and Messenger – `recieveCountMessenger` – to send a message asking for `count`. In the `handleMessage()` method of `RecieveCountHandler`, the value of `count` is fetched from the incoming message's `arg1` variable and displayed in the TextView.

```
public class CounterActivity extends Activity implements OnClickListener {
  ...
  public static final int GET_COUNT=0;
  boolean bound = false;
  Messenger requestCountMessenger, recieveCountMessenger;

  class RecieveCountHandler extends Handler{
    @Override
    public void handleMessage(Message msg) {
       countValue=0;
       switch (msg.what) {
       case GET_COUNT:
         countValue=msg.arg1;
         displayCountValue.setText(countValue+ "");
       break;
       default:
       break;
```

```
        }
        super.handleMessage(msg);
      }
  }
  ServiceConnection CounterServiceConnection = new
  ServiceConnection() {
    @Override
    public void onServiceDisconnected(ComponentName arg0) {
      requestCountMessenger=null;
      recieveCountMessenger=null;
      bound = false;
    }
    @Override
    public void onServiceConnected(ComponentName arg0, IBinder
    arg1)
    {
      requestCountMessenger=new Messenger(arg1);
      recieveCountMessenger=new Messenger(new RecieveCountHandler());
      bound=true;
    }
  };
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
    intent = new Intent("com.mad.counterservice");
  }
  @Override
  public void onClick(View arg0) {
    switch (arg0.getId()) {
    // bind and unbind implemented as described earlier in
       local binding
    case R.id.fetchCount:
      if (bound == true) {
        Message requestMessage=Message.obtain(null, GET_COUNT);
        requestMessage.replyTo=recieveCountMessenger;
        try {
          requestCountMessenger.send(requestMessage);
        } catch (RemoteException e) {
          e.printStackTrace();
        }
      }
     break;
     default:
     break;
    }
   }
  }
```

**Snippet 5.11** | Client component – `CounterActivity`.

When the `CounterActivity` binds to the Service, the `onServiceConnected()` method is invoked with the `Binder` (`arg1`) returned by the Service (Line 30). This `Binder` is used to instantiate the `requestCountMessenger`, which will be used to send request messages to the Service (Line 31). When the user taps on the *Refresh* button, the request to fetch the value of `count` is made as shown in Lines 47, 48, and 50. In line 47, the `requestMessage` object of the `Message` class is created using the `obtain()` method. The `replyTo` variable of `requestMessage` is set as `recieveCountMessenger`, so that when the Service replies with the value of `count`, the `RecieveCountHandler` will handle the reply, as shown in Line 48. In Line 50, the `requestMessage` is sent to the Service using the `send()` method of `requestCountMessenger`.

In Line 39, the Intent object (`intent`) is created using a String value as argument to the constructor; this String value is referred to as `action` to Intent in Android (refer to Section 5.7). This way of creating an Intent is done because the Service class now resides in a different app. The entry in the AndroidManifest.xml file of the `CounterService` also has to be modified as shown in Snippet 5.12; also refer to Section 5.7.

```
<service android:name="CounterService">
  <intent-filter >
      <action android:name="com.mad.counterservice"/>
      <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</service>
```

**Snippet 5.12** | AndroidManifest.xml of the `CounterService`.

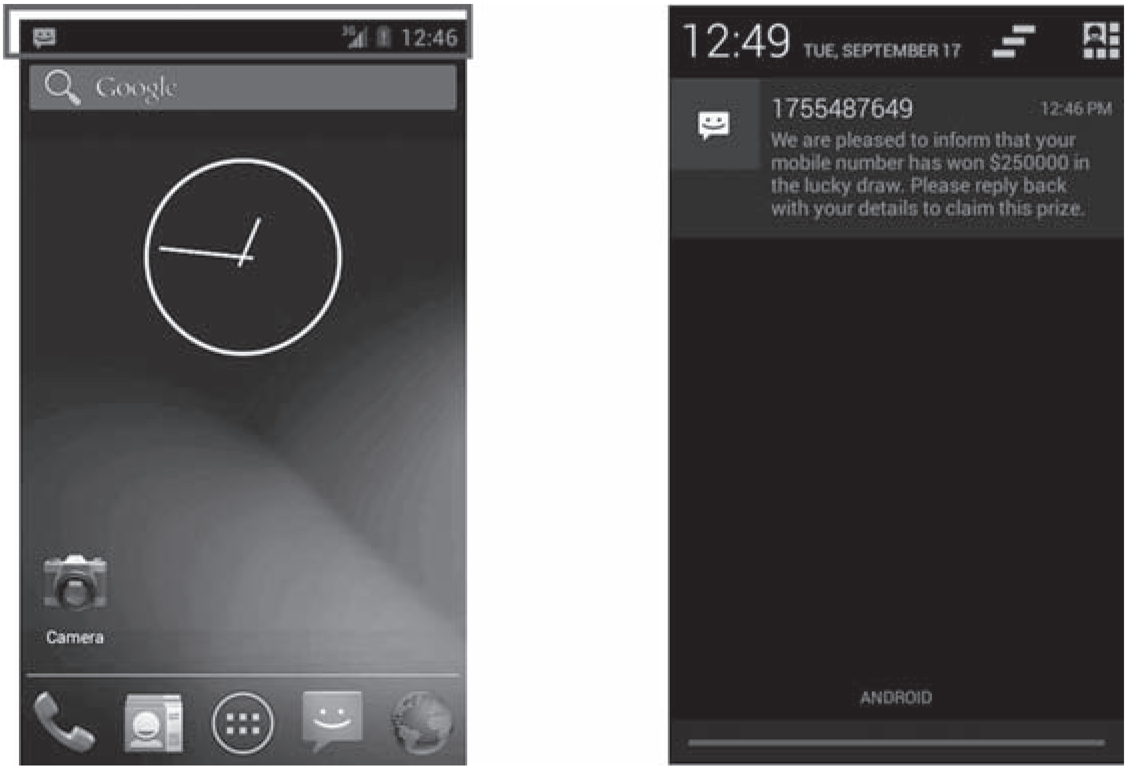
Till now we have covered a pretty good ground on the intricacies of Service. As a Service runs in the background, it is crucial to interact with the Service at its major milestones or completion. For example, once the file download is completed, the user has to be informed about it. Firing an Activity from the Service to inform the user about download completion (or any such scenario) is intrusive, and surely not a polite way of getting user's attention. Rather, this has to be done in a more subtle manner, where the user is kept informed without being interrupted. Notifications perfectly fit the bill.

## 5.6 NOTIFICATIONS

Notification is a mechanism to inform a user about the occurrence of an event. The event might be a missed call, a SMS (Short Message Service) received, completion of a download, or any other app-specific event. Notifications are broadly categorized into hardware and software notifications. Vibrating a mobile phone or flashing a light-emitting diode (LED) fall under the hardware notifications. The software notifications usually involve flashing message to a user in a nonintrusive manner.

In Android, a software notification is displayed as a small icon in the notification area, on the top of the screen as depicted in Fig. 5.7(a). The details pertaining to the notification can be seen in a notification drawer [Fig. 5.7(b)] by dragging down the notification area.

The advantage of using notification lies in the fact that it does not distract a user from the current task at hand, but still provides him or her an option to respond to notification by opening the notification drawer at anytime, whenever convenient. Added to this, a notification can be associated with a dedicated Activity where the user can perform necessary action, for example, read a received SMS, and if desired reply back.



1755487649 12:46 PM
We are pleased to inform that your mobile number has won $250000 in the lucky draw. Please reply back with your details to claim this prize.

**Figure 5.7** | (a) Notification area and (b) notification drawer.

Android provides two views for notifications – normal view and big view – that are visible in the notification drawer. Figure 5.7(b) depicts a normal view of a notification. The big view provides extra information in comparison to the normal view, as depicted in Fig. 5.8.

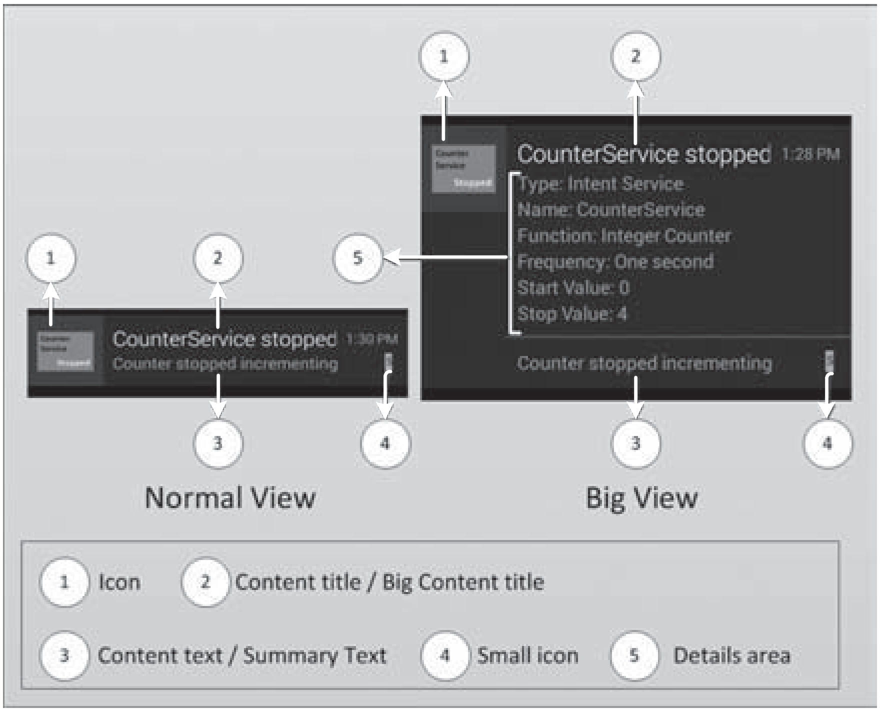
Implementing a notification starts with creating the template of notification, followed by defining the navigation from the notification, and finally triggering it. Let us now explore the nitty-gritty of implementing notifications using the Counter example that will be modified to trigger a notification when the `CounterService` is stopped.

A template of notification is a blueprint that defines icons, content text, content title, and other related things, as depicted in Fig. 5.8. To start with, we need to create a `NotificationCompat.Builder` object that helps setting the details of template, as shown in Snippet 5.13 (Line 2–6).

```
1  private void triggerNotification(boolean big) {
2    NotificationCompat.Builder builder = new Builder(this);
3    builder.setLargeIcon
       (BitmapFactory.decodeResource(getResources(),
         R.drawable.servicestopped));
4    builder.setSmallIcon(R.drawable.ic_small_servicestop);
5    builder.setContentTitle("CounterService stopped");
6    builder.setContentText("Counter stopped incrementing");
7    Intent intent = new Intent(this, ResultActivity.class);
8    intent.putExtra("Count", count);
9    PendingIntent pendingIntent=PendingIntent.getActivity(this,
     0, intent, PendingIntent.FLAG_UPDATE_CURRENT);
10   builder.setContentIntent (pendingIntent);
```

```
  if (big) {
   String[] events = new String[6];
   events[0] = “Type: Intent Service”;
   events[1] = “Name: CounterService”;
   events[2] = “Function: Integer Counter”;
   events[3] = “Frequency: One second”;
   events[4] = “Start Value:  0”;
   events[5] = “Stop Value:” + count;
   NotificationCompat.InboxStyle inboxStyle = new
    NotificationCompat.InboxStyle();
   inboxStyle.setBigContentTitle(“CounterService stopped”);
   inboxStyle.setSummaryText(“Counter stopped incrementing”);
   for (int i = 0; i < events.length; i++) {
     inboxStyle.addLine(events[i]);
   }
  builder.setStyle(inboxStyle);
 }
 NotificationManager manager=
 (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
 manager.notify(1, builder.build());
}
```

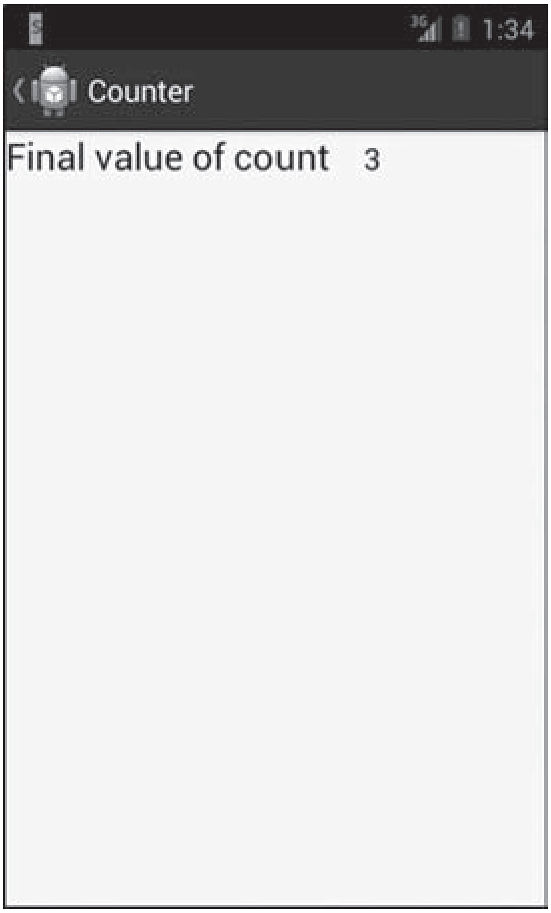
**Snippet 5.13** | Creating a notification.


1
2
5
Normal View
Big View
3
4
1 Icon
2 Content title / Big Content title
3 Content text / Summary Text
4 Small icon
5 Details area

CounterService stopped 1:30 PM
Counter stopped incrementing

CounterService stopped 1:28 PM
Type: Intent Service
Name: CounterService
Function: Integer Counter
Frequency: One second
Start Value: 0
Stop Value: 4
Counter stopped incrementing

**Figure 5.8** | Normal view and big view.



Final value of count 3

**Figure 5.9** | `ResultActivity` obtained on click of notification.

Further, to define the details in the big view, the `NotificationCompat.InboxStyle` API is used. The `addLine()` method (Line 23) of this API is used to set multiple lines of text (Lines 12–18) that will appear in the details area. In addition to this, `BigContentTitle` and `SummaryText` of the `InBoxStyle` object are also set (Lines 20–21). Once the `InBoxStyle` object is configured, we set it as the style of the notification using the `setStyle()` method of `NotificationCompat.Builder` (Line 25).

The next step is to define the navigation from the notification when a user taps on it. In our example, let us define the navigation from the notification to `ResultActivity`, as depicted in Fig. 5.9. This Activity contains a TextView that displays the final value of `count` when the Service is stopped.

When the user taps on the notification, which is a part of Android system's notification service, he or she is supposed to navigate to our app's Activity (`ResultActivity`). This scenario violates Android's security guideline that mandates that an app can only trigger its own components explicitly. To resolve this situation, we use the `PendingIntent` API, to which we pass the necessary Intent, along with the required permissions, to the Android's notification service, as shown in Lines 7–10.

The `getActivity()` method of `PendingIntent` API accepts four arguments: context, request code (currently not used), Intent, and flags. The flags passed could take `FLAG_UPDATE_CURRENT`, `FLAG_ONE_SHOT`, `FLAG_NO_CREATE`, and `FLAG_CANCEL_CURRENT` integer constants as values. `FLAG_UPDATE_CURRENT` ensures that if another `PendingIntent` is created for the same Intent, the current one is updated. `FLAG_ONE_SHOT` ensures that this `PendingIntent` can be used to trigger the component only once. `FLAG_NO_CREATE` does not create a `PendingIntent`, if it already exists.

Finally, to trigger the notification, we use the `NotificationManager` (which is Android system's notification service) object's `notify()` method as shown in Lines 27 and 28. The `notify()` method accepts two arguments: an `id` and a `Notification` object. The `build()` method of the builder creates the `Notification` object and the `id` is used to uniquely identify each notification.

The code to display the value of `count` in the TextView of `ResultActivity` is shown in Snippet 5.14.

```
public class ResultActivity extends Activity {
@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.result);
 TextView textView=(TextView) findViewById(R.id.displayCount);
 int count = getIntent().getIntExtra("Count",  0);
 textView.setText(count + "");
 }
}
```

**Snippet 5.14** | Displaying value of `count` in `ResultActivity`.

When the user clicks on the notification created, he or she is taken to the `ResultActivity` in our example. This is a special Activity, because it appears only when the user clicks on the notification, and never as a part of the regular app navigation. In such scenarios, it may not be required to list the app in the Recents view (obtained by long press of Home button) of the device. To achieve this, we have to edit the AndroidManifest.xml, as depicted in Snippet 5.15.

```
1  <activity
2   android:name="ResultActivity"
3   android:excludeFromRecents="true"
4   android:launchMode="singleTask"
5   android:taskAffinity="">
6  </activity>
```

**Snippet 5.15** | Changes to AndroidManifest.xml.

In Snippet 5.15, the `excludeFromRecents` (Line 3) attribute ensures that the app will not be visible in the Recents view, if navigated to via the `ResultActivity`. The `singleTask` value assigned to the `launchMode` attribute (Line 4) ensures that only one instance of this Activity is created at any point of time. Thereby, even if the user clicks on the notification multiple times, the Activity will be created only once. The `taskAffinity` attribute specifies the affinity of the Activity for a task. Because this Activity is a `singleTask` Activity, it ideally does not contain affinity for any task; hence an empty string is set as the value (Line 5).

It might be desirable to cancel the notification (remove from the notification area) automatically after the user has been navigated to the `ResultActivity`. To achieve this, the notification id that is used to trigger the notification (Line 28, Snippet 5.13) is passed as an extra in the `Intent` object (Line 7, Snippet 5.13), as shown in Snippet 5.16.

```
intent.putExtra(“notificationid”, 1);
```

**Snippet 5.16** | Passing notification id in the Intent.

The notification id is then retrieved in the `ResultActivity`, and the `cancel()` method of the `NotificationManager` is used to clear the notification, as shown in Snippet 5.17.

```
int notificationid=getIntent().getIntExtra(“notificationid”, 0);
NotificationManager manager=
(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
manager.cancel(notificationid);
```

**Snippet 5.17** | Cancelling the notification from `ResultActivity`.

In some cases, the Activity triggered from notification may be a part of an app’s regular workflow. For example, when a user gets a new e-mail notification and clicks it, he or she is taken to an Activity displaying the newly received e-mail. When the user taps on the Back button, while in this Activity, he or she is taken to the inbox of the e-mail app. This scenario is called as regular Activity navigation. For such scenarios, the `PendingIntent` supplied to the `NotificationManager` must contain information about the task stack of the app. To achieve this scenario in our example, we use the `TaskStackBuilder` API as shown in Snippet 5.18.

```
1  TaskStackBuilder stackBuilder=TaskStackBuilder.create(this);
2  stackBuilder.addParentStack(ResultActivity.class);
3  stackBuilder.addNextIntent(intent);
4  PendingIntent pendingIntent=stackBuilder.getPendingIntent(0,
   PendingIntent.FLAG_UPDATE_CURRENT);
```

**Snippet 5.18** | `TaskStackBuilder` used to define navigation from notification.

In Line 2 of Snippet 5.18, the `addParentStack()` method is used to define the parent task stack of the `ResultActivity` (triggered on click of the notification), and the `addNextIntent()` is used to add the `Intent` to trigger `ResultActivity` (Line 3). The `PendingIntent` to be used is obtained using the `getPendingIntent()` method of `TaskStackBuilder` object, as shown in Line 4.

Additionally, we have to specify the parent of the Activity (`ResultActivity` in this example) in the AndroidManifest.xml using the `parentActivityName` attribute as shown in Snippet 5.19.

```
<activity android:name=“ResultActivity”
  android:parentActivityName=“CounterActivity”>
</activity>
```

**Snippet 5.19** | Changes to AndroidManifest.xml.

When the user now taps on the Back button, while in the `ResultActivity`, he or she is navigated back to the `CounterActivity`, whereas in the earlier case (special Activity scenario) the user is navigated back to either the Home screen or the previously opened app (if any).

## 5.7 INTENTS AND INTENT RESOLUTION

It is quite apparent by now that Intents play a very important role as binding agents in the decoupled architecture of Android. We have already seen how Intents are used to trigger an Activity and Services. Let us now explore what happens behind the scene that makes the interaction among the Android components seamless.

Imagine we have to borrow a book from a library. If the book details are explicitly available with us, we can directly locate that book. Otherwise, with partial details such as author or subject or year of publication, the librarian can help us zero-in on the book. This analogy applies to Intents as well.

While we have been triggering another Activity or Service from the current component, we explicitly knew the name of the component to be triggered. The Intents used in such scenarios are known as explicit Intents. However, when we do not know the name of the component, we use implicit Intents. An implicit Intent contains necessary information that helps Android runtime to zero-in on the appropriate component that needs to be triggered. This process of identifying the appropriate component by Android runtime is called as Intent resolution.

Going back to the analogy of a library, if the search ends up in more than one book, the reader is offered the choice to select one among them. Similarly, if Android runtime lookup results in multiple components, the user is presented with choices to select from. For example, when we want to share a photo from the gallery, Android runtime offers multiple choices such as Bluetooth, e-mail, or social networks.

In Section 5.5.4, we have used an implicit Intent (Line 39, Snippet 5.11) to trigger the `CounterService` from the `CounterActivity`, which was part of another app. Let us now take a deep dive on how an Android component can be prepared to respond to an implicit Intent so that it can be identified by Android runtime.

Intent resolution is achieved using the `<intent-filter>` tag in AndroidManefest.xml file of the component that needs to be triggered. This tag is nested inside the component tags such as `<activity>` or `<service>`. The `<intent-filter>` tag can contain `<action>`, `<category>`, and `<data>` tags, as shown in a generic Snippet 5.20.

```
<intent-filter>
  <action android:name="<<action>>"/>
  <category android:name="android.intent.category.DEFAULT"/>
  <category android:name="<<category>>"/>
  <data android:mimeType="<<mimeType>>" />
</intent-filter>
```

**Snippet 5.20** | Blueprint of Intent filter.

The `<action>` tag contains a string constant to specify the action that can be performed by the component. For example, `ACTION_CALL` is one of Android's predefined actions used to trigger a dialer. We can also define custom actions, as shown in Line 3 of Snippet 5.12. This string is used to create the implicit Intent object, as shown in Line 39 of Snippet 5.11.

The `<category>` tag also contains a string constant to provide additional information used for Intent resolution. For example, `CATEGORY_LAUNCHER` is one of Android's predefined categories used to specify the launcher Activity of an app. `CATEGORY_DEFAULT` is a predefined category that must always be specified as mandated by Android runtime.

The `<data>` tag is an optional tag that is used to specify the type of data [MIME (Multipurpose Internet Mail Extensions) type] that can be handled by the component. For example, when action is `ACTION_CALL`, the data that can be provided is a telephone number.

Once the component that needs to be triggered is configured as above, we can then create an implicit Intent in the triggering component. We have to specify action, category, and data (optional) in this Intent object. The Android runtime will match these values against `<action>`, `<category>`, and `<data>` values specified in apps' manifest file of the component to be triggered.

Android runtime uses the following rules when matching the values passed by Intent with those present in Intent filter. An implicit Intent will be able to trigger a component, if and only if:

1. Its action matches against at least one action in the Intent filter.
2. All of its supplied category values are present in the Intent filter.
3. Values of data supplied in it match (even partially) against those in the Intent filter.

The matching always happens in a sequence – action, category, and data. If multiple components match to the criteria, the user is presented with choices by Android runtime. For example, when we want to open a URL from an e-mail app, Android runtime offers a choice of browsers to select from, if multiple browsers are installed.

Intents also contain flags, which decide the launch configuration of the component to be triggered. Some of the values of flags[1] are `FLAG_ACTIVITY_CLEAR_TASK` (clears the task stack of an Activity that is going to be started, if the Activity is already associated with any other task), `FLAG_ACTIVITY_SINGLE_TOP` (does not start the Activity, if it is already on the top of the task stack), and `FLAG_ACTIVITY_NEW_TASK` (starts the Activity in a new task stack). These flags, however, do not play any role in Intent resolution.

## 5.8 BROADCAST RECEIVERS

A plethora of events keep occurring in smart devices. These events may be broadcasted by apps or by the system itself. Events such as an incoming call, an incoming SMS, and Wi-Fi availability are some of the common ones generated by the system. All of these events are announced by Android runtime by broadcasting implicit Intents. An app can respond to these events by using Broadcast Receivers.

Broadcast Receiver is a unique building block of an Android app, as it can get triggered even when the app is not running. Each event creates a new Broadcast Receiver object. A Broadcast Receiver runs on the *main* thread of the app, and once it runs its course, it is ready for garbage collection. Android mandates a Broadcast Receiver to complete its execution within 10 s.

A Broadcast Receiver is implemented by extending the `BroadcastReceiver` class, and overriding its only callback method – `onReceive()`, as shown in Snippet 5.21. As soon as a Broadcast Receiver is triggered to respond to an event, the `onReceive()` method is executed.

```
public class MyCustomBroadcastReceiver extends BroadcastReceiver{
  @override
  public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "The BR has been triggered",
    Toast.LENGTH_SHORT).show();
  }
}
```

**Snippet 5.21** | Broadcast Receiver implementation.

[1] http://developer.android.com/reference/android/content/Intent.html

The implemented Broadcast Receiver has to be registered in the AndroidManifest.xml file. To enable it to be triggered by an implicit Intent, we need to configure its `<intent-filter>` element, as shown in Snippet 5.22.

```
1  <receiver android:name="MyCustomBroadcastReceiver">
2    <intent-filter>
3    <action
      android:name="com.mad.broadcastdemo.SIMPLE_BROADCAST"/>
4    <category android:name="android.intent.category.DEFAULT"/>
5    </intent-filter>
6  <receiver/>
```

**Snippet 5.22** | Registering Broadcast Receiver in app manifest.

The value of `action` supplied to the Intent filter in Line 3 of Snippet 5.22 is a custom value. The implicit Intent that will be used to trigger this Broadcast Receiver will contain this string as the value to its `action` parameter as shown in Snippet 5.23. We use the `sendBroadcast()` method from the triggering component (Activity in this case) to send out a broadcast to trigger the Broadcast Receiver as shown in Line 2 of Snippet 5.23. A triggering component could be an Activity, a Service, or even another Broadcast Receiver.

```
1  Intent intent=new
   Intent("com.mad.broadcastdemo.SIMPLE_BROADCAST");
2  sendBroadcast(intent);
```

**Snippet 5.23** | `sendBroadcast()` method in triggering component.

Once the Broadcast Receiver is registered, it gets triggered whenever any app broadcasts implicit Intent with the specified action string. Now, if there is a need to limit other apps from triggering this Broadcast Receiver, we can set `android:exported` attribute value to `false` in the `<receiver>` tag of the app manifest. This will make sure that only the app that contains the Broadcast Receiver can trigger it.

When a Broadcast Receiver is registered in the manifest file, it will always respond to matching broadcasts, and there is no way to disable it. However, if we wish to control enabling or disabling a Broadcast Receiver in an app, we can register and unregister it programmatically, as shown in Snippet 5.24.

```
1  protected void onResume() {
2      super.onResume();
3      registerReceiver(myCustomBroadcastReceiver, new
           IntentFilter("com.mad.broadcastdemo.SIMPLE_BROADCAST"));
4  }
5  protected void onPause() {
6     super.onPause();
7     unregisterReceiver(myCustomBroadcastReceiver);
8  }
```

**Snippet 5.24** | Dynamic registration/deregistration of Broadcast Receiver.

The scenario depicted here is a fairly simple one, where an app has sent the broadcast. However, to listen and respond to system-generated broadcasts in an app, we need to use the built-in action strings (relevant to the event) when configuring the Intent filter of Broadcast Receiver in that app. We will delve into system-generated broadcast scenarios in Section 5.9, when exploring telephony and SMS-related APIs.

## 5.9 TELEPHONY AND SMS

Android provides `TelephonyManager` and `SmsManager` APIs to implement telephony and SMS-related functionalities, respectively. Key features of the `TelephonyManager` include accessing network and device-type information, and retrieving information about phone states. The `SmsManager` provides methods to send and receive SMS.

The `TelephonyManager` is accessed through a telephony service provided by Android, as depicted in Snippet 5.25. The `getSystemService()` method, earlier used in Snippet 5.17 to access the `NotificationManager`, is used here to access the telephony service.

```
TelephonyManager telephonyManager =
 (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
```

**Snippet 5.25** | Accessing `TelephonyManager`.

Once we have access to the `TelephonyManager`, its `getNetworkType()` and `getPhoneType()` methods are used to access network and device type information, respectively, as shown in Lines 2 and 27 of Snippet 5.26. This information may be crucial to those apps that need to determine the network and device type before executing a specific task. For example, on the basis of network type, media of appropriate resolution could be streamed at any specific point in time.

```
1  String networkType;
2  switch (telephonyManager.getNetworkType()) {
3    case TelephonyManager.NETWORK_TYPE_CDMA:networkType=“CDMA”;
4    break;
5    case TelephonyManager.NETWORK_TYPE_EDGE:networkType=“EDGE”;
6    break;
7    case TelephonyManager.NETWORK_TYPE_EHRPD:networkType=“EHRPD”;
8    break;
9    case TelephonyManager.NETWORK_TYPE_HSDPA:networkType=“HSDPA”;
10   break;
11   case TelephonyManager.NETWORK_TYPE_HSPA:networkType=“HSPA”;
12   break;
13   case TelephonyManager.NETWORK_TYPE_HSPAP:networkType=“HSPAP”;
14   break;
15   case TelephonyManager.NETWORK_TYPE_HSUPA:networkType=“HSUPA”;
16   break;
17   case TelephonyManager.NETWORK_TYPE_LTE:networkType=“LTE”;
18   break;
19   case TelephonyManager.NETWORK_TYPE_UMTS:networkType=“UMTS”;
20   break;
21   case
      TelephonyManager.NETWORK_TYPE_UNKNOWN:networkType=“UNKNOWN”;
22    break;
23    default: networkType=“Unknown”; break;
24  }
25  System.out.println(“Network type detected is:“+networkType);
26  String phoneType;
27  switch (telephonyManager.getPhoneType()) {
```

```
case TelephonyManager.PHONE_TYPE_CDMA: phoneType="CDMA";break;
case TelephonyManager.PHONE_TYPE_GSM: phoneType="GSM";break;
case TelephonyManager.PHONE_TYPE_SIP:phoneType="SIP";break;
case TelephonyManager.PHONE_TYPE_NONE: phoneType="NONE";break;
default: phoneType="Unknown";
break;
}
System.out.println("Phone type detected is:"+phoneType);
```

**Snippet 5.26** | Accessing network and device-type information.

The `TelephonyManager` can also be used to determine the state of a mobile device. A change in phone state is a system-generated broadcast. This broadcast is fired with an Intent containing extras (key – `EXTRA_STATE`), as depicted in Table 5.1. These extras are part of the `TelephonyManager` that helps in retrieving information about phone states in an app that is registered to listen to these changes.

**Table 5.1** | Phone states

| Phone State | EXTRA_STATE Value |
|---|---|
| Phone disconnected | `EXTRA_STATE_IDLE` |
| Phone ringing | `EXTRA_STATE_RINGING` |
| Phone answered | `EXTRA_STATE_OFFHOOK` |

To listen to these phone state changes, a Broadcast Receiver has to be implemented in the app, as depicted in Snippet 5.27. When configuring the Intent filter of Broadcast Receiver in this app, we need to use the built-in action string `android.intent.action.PHONE_STATE`.

```
public class PhoneStateChangeReceiver extends BroadcastReceiver{
  public void onReceive(Context context, Intent intent) {
  Log.i("com.mad.telephonymanagerdemo",
         PhoneStateChangeReceiver triggered");
  String triggeredState=
       intent.getStringExtra(TelephonyManager.EXTRA_STATE);
  if(triggeredState.equals(TelephonyManager.EXTRA_STATE_IDLE)){
  Log.i("com.mad.telephonymanagerdemo", "Phone is
        idle");}
  else
  if(triggeredState.equals(TelephonyManager.EXTRA_STATE_OFFHOOK))
  {
  Log.i("com.mad.telephonymanagerdemo", "Phone is off
         the hook");}
  else if
  (triggeredState.equals(TelephonyManager.EXTRA_STATE_RINGING)){
  Log.i("com.mad.telephonymanagerdemo", "Phone is ringing");}
  }
}
```

**Snippet 5.27** | Broadcast Receiver listening to phone state changes.

The app has to request READ_PHONE_STATE permission in the AndroidManifest.xml file to read the device state change, as shown in Snippet 5.28. Failing which, the app will throw security exception and terminate.

```
<uses-permission android:name="android.permission.READ_PHONE_STATE">
```

**Snippet 5.28** | Permissions added to manifest file to read phone state.

Another key aspect of telephony is to place a call. However, we don't need the TelephonyManager to implement it. To place a call, we need to create an Intent object with action Intent.ACTION_DIAL, as shown in Snippet 5.29. The phone number to be called is set as data to the Intent object (Line 2) with scheme "tel:". This will trigger default dial pad Activity with the requested number displayed.

```
1  Intent makeCallIntent=new Intent(Intent.ACTION_DIAL);
2  makeCallIntent.setData(Uri.parse("tel:"+phoneNumber));
3  startActivity(makeCallIntent);
```

**Snippet 5.29** | Placing a call using Intent.

The SmsManager provides methods to send and receive SMS. To access the SmsManager, we have to use the getDefault() method, as shown in Snippet 5.30.

```
SmsManager manager=SmsManager.getDefault();
```

**Snippet 5.30** | Accessing SmsManager.

To send a text message, the sendTextMessage() method is used, as shown in Snippet 5.31. The first parameter of this method is the destination phone number. In this example, the destination is another emulator with port number 5556. The second parameter is SMS service center address. In our example, null is used for the default settings. The third parameter is the text message that needs to be sent. Fourth and fifth parameters are PendingIntents to be broadcasted if message is successfully sent and delivered to recipient, respectively. We also need to request SEND_SMS permission to send SMS.

```
manager.sendTextMessage("5556", null, message, null, null);
```

**Snippet 5.31** | Sending SMS.

The other aspect of the SmsManager is to receive SMS. To receive an incoming SMS, a Broadcast Receiver has to be implemented in the app, as depicted in Snippet 5.32. When configuring the Intent filter of Broadcast Receiver in this app, we need to use the built-in action string android.provider.Telephony.SMS_RECEIVED. We also need to request for RECEIVE_SMS and READ_SMS permissions for receiving and reading the SMS.

```
public class SmsReceiver extends BroadcastReceiver {
 @Override
 public void onReceive(Context arg0, Intent arg1) {
  Bundle bundle=arg1.getExtras();
5     Object[]pdus = (Object[]) bundle.get("pdus");
6     SmsMessage message=
         SmsMessage.createFromPdu((byte[])pdus[0]);
7      Log.i("com.mad.smsreceived",message.getMessageBody());
8     Log.i("com.mad.smsreceived",message.getOriginatingAddress());
9    }
10 }
```

**Snippet 5.32** | Receiving SMS.

Each SMS is received as a Packet Data Unit (PDU). If the message is too long, it is split into multiple PDUs. These PDUs can be obtained as an array of `Object` from the Intent in the `onReceive()` method of Broadcast Receiver, as shown in Lines 4 and 5. To extract the message, the `createFromPdu()` method of `SmsMessage` class is used, as shown in Line 6. Now, to extract the message body and the sender's number, the `getMessageBody()` and `getOriginatingAddress()` methods are used, respectively, as shown in Lines 7 and 8.

## 5.10 LET'S APPLY

The 3CheersCable app allows the user to subscribe to channel categories such as Movies and Music, and shows the list of subscribed categories in the `MainActivity` [Fig. 5.10(a)]. On click of each category, a TV guide that displays the channels and shows under the selected category is shown in `TVGuide` Activity [Fig. 5.10(b)]. The `TVGuide` Activity contains two Fragments: `ChannelsListFragment` (that displays the channels under a selected category) and `ShowsListFragment` (that displays the shows under a selected channel).

In the `ShowsListFragment`, against every show, an icon is used to indicate if the user has marked the corresponding show as his or her favorite. In this section, the functionality to share these favorite shows and timings when the user taps on the `sendSMS` action item shall be implemented.

The `onOptionsItemSelected()` method must be overridden to capture the user selection of the `sendSMS` menu item. The `showSendSMSDialog()` method is invoked to show a confirmation dialog box before sending the SMS [see Fig 5.11(a)].

```
public boolean onOptionsItemSelected(MenuItem item) {
switch (item.getItemId()) {
case R.id.sendSMS:
showSendSMSDialog();
return true;
default:
return false;}}
```

In the `showSendSMSDialog()` method, on click of the positive button of the alert dialog, an implicit Intent with action as `Intent.ACTION_VIEW` and data as `sms:1234` is triggered to start the Messaging app on the device. The Intent also contains the favorite shows and timings as extras. This extra is assigned to the key `sms_body` so that the Messaging app includes it as the body of the SMS to be sent.

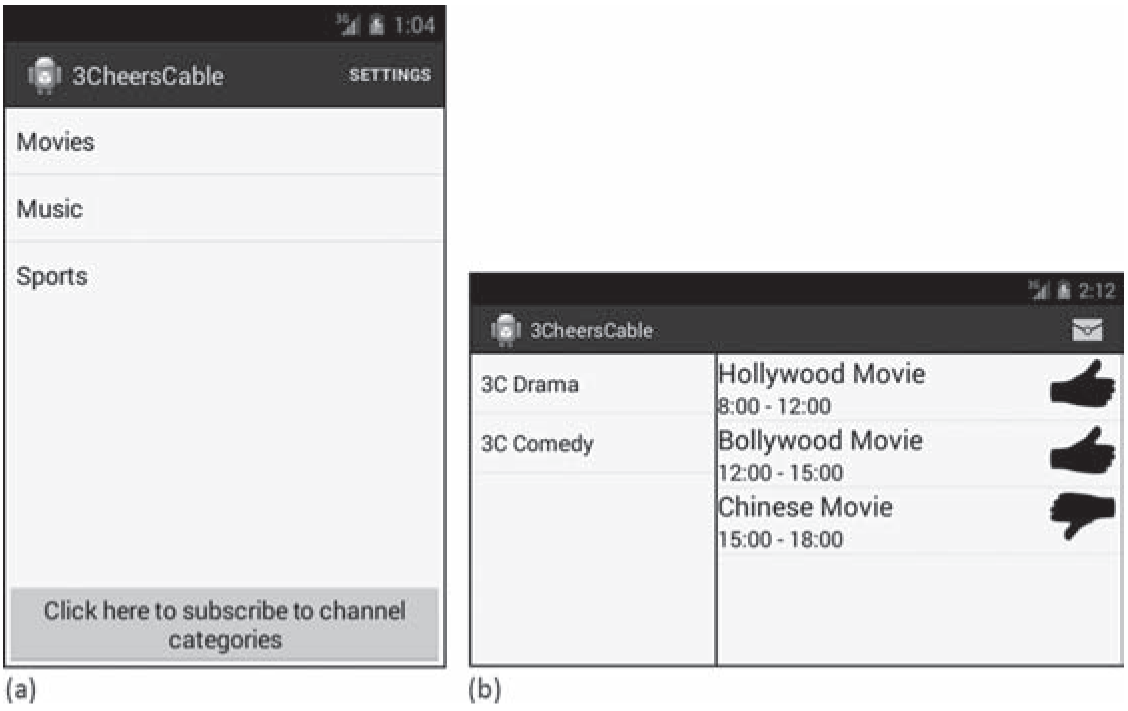
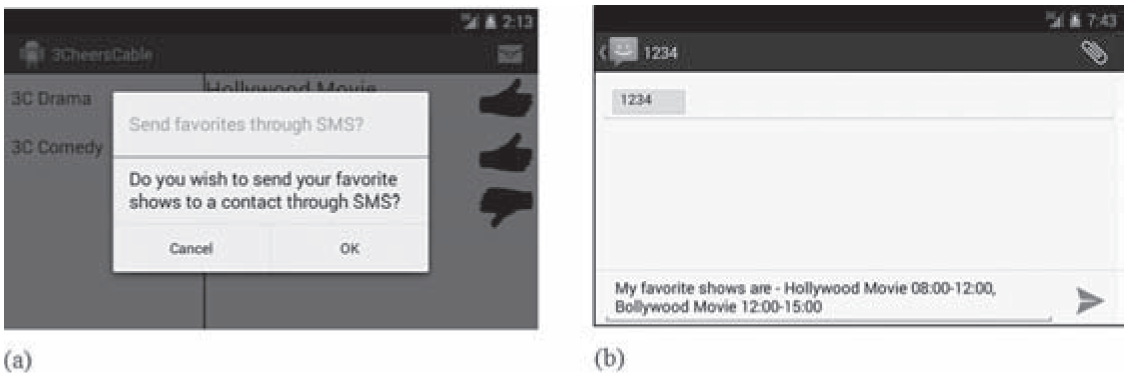


**Figure 5. 10** | (a) `MainActivity` and (b) `TVGuide`.

```
public void showSendSMSDialog() {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Send favorites through SMS?")
.setMessage("Do you wish to send your favorite shows to a contact
through SMS?")
.setPositiveButton("OK", new OnClickListener() {
  public void onClick(DialogInterface arg0, int arg1) {
    ... //code to construct the sms
    Intent sendIntent = new Intent(Intent.ACTION_VIEW);
    sendIntent.setData(Uri.parse("sms:1234"));
    sendIntent.putExtra("sms_body", sms);
    startActivity(sendIntent);}})
.setNegativeButton("Cancel", new OnClickListener() {
  public void onClick(DialogInterface arg0, int arg1) {
   }}).create().show();
}
```

This approach of sending SMS using the Messaging app is used so that the user can select the contacts he or she intends to send this SMS to (Fig. 5.11).



Send favorites through SMS?

Do you wish to send your favorite shows to a contact through SMS?

My favorite shows are - Hollywood Movie 08:00-12:00, Bollywood Movie 12:00-15:00

**Figure 5.11** | (a) Message dialog and (b) Messaging app.

## SUMMARY

This chapter has introduced the components and techniques that are ideal for implementing long-running tasks in Android apps. It has explored the Thread, AsyncTask, and Service to suggest appropriate scenarios in which they can be used for implementing long-running tasks, and has also discussed life cycle and various modes of operations of a Service.

The chapter has outlined the implementation of a nonintrusive way of communication with app user using notifications, and has drawn out a complete picture of Android messaging framework – Intents and discussed the process of Intent resolution in detail.

The chapter has also delved into one of the key components of Android – the Broadcast Receiver – which is used for responding to both system and app broadcasts. Toward the end, it has also explored the telephony and SMS-related APIs.

## REVIEW QUESTIONS

1. Illustrate the benefits of AsyncTask over Threads for implementing long-running tasks.
2. Illustrate Service life-cycle states and respective callback methods.
3. Define IntentService.
4. Explain local and remote binding concepts of Services.
5. Define the procedure to create notifications.
6. Outline the process of Intent resolution.
7. Define Broadcast Receiver.
8. Explain TelephonyManager API and its usage.

## FURTHER READINGS

1. AsyncTask: http://developer.android.com/reference/android/os/AsyncTask.html
2. Service and its life cycle : http://developer.android.com/guide/components/services.html

3. IntentService: http://developer.android.com/reference/android/app/IntentService.html
4. Bound Services: http://developer.android.com/guide/components/bound-services.html
5. Notifications: http://developer.android.com/guide/topics/ui/notifiers/notifications.html
6. Intents and Intent Filters: http://developer.android.com/guide/components/intents-filters.html
7. BroadcastReceiver: http://developer.android.com/reference/android/content/BroadcastReceiver.html
8. TelephonyManager:http://developer.android.com/reference/android/telephony/TelephonyManager.html
9. SmsManager: http://developer.android.com/reference/android/telephony/SmsManager.html

# 6
# App Data – Persistence and Access

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Identify various data persistence and access mechanisms in a mobile app.
- Implement data persistence and access using flat files and shared preferences.
- Construct and leverage relational database on devices using SQLite.
- Share app data with other apps using Content Providers.
- Perform data exchange with the backend enterprise systems.

## 6.1 INTRODUCTION

So far, you have learnt two important aspects of mobile app development – the user interface and the long-running tasks. In both these cases, data was captured through user inputs and exchanged using Intents. Such data is volatile, and gets wiped off once the app is terminated by the user or Android runtime.

However, we will come across scenarios where the app data may have to be stored permanently in order to be retrieved at a later point in time. This data can be saved either locally on the device or remotely on the enterprise systems. For local storage, apps use internal memory of the device or storage media cards such as Secure Digital (SD) card. For remote storage, apps have to connect to external enterprise systems over the network. Because of limited storage space and network bandwidth, developers have to ensure that app data needs are optimally handled.

Data could be either primitive or complex in nature, and can be stored on the device in an unstructured or structured manner. Android provides shared preferences framework to deal with primitive data. It supports storage of unstructured data in flat files. It also provides SQLite database and Application Programming Interface (API) to deal with relational data. Onus is on the developer to identify the data-handling techniques based on app requirements.

This chapter primarily focuses on data handling in Android apps. It starts with exploring storage mechanism for unstructured data using flat files. It further delves into shared preference framework as a mechanism to store primitive data such as user preferences. It also explores SQLite database and API to manage relational data, and discusses data sharing across apps using Content Provider. Towards the end, it elaborates on mechanisms to deal with remote data needs.

## 6.2 FLAT FILES

Flat files are used to persist unstructured data – primitive and complex alike – and act as an ad hoc scribble pad for an app. Android uses Java input–output (IO) API to read and write flat files.

Let us now create a simple app where a user can read and write text in a flat file on the device, as depicted in Fig. 6.1.

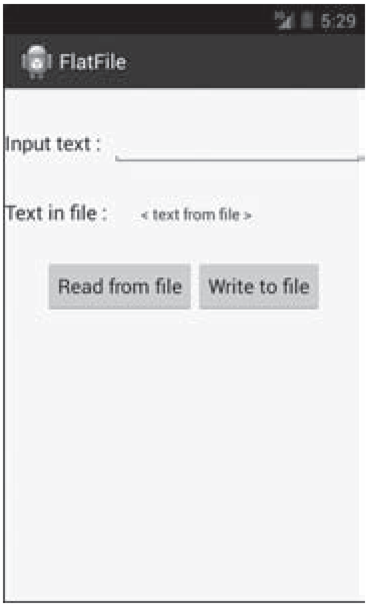


**Figure 6.1** | FlatFile app.

To implement this, two user-defined methods – `getFileContent()` and `writeToFile(String text)` – are used in the Activity. The `getFileContent()` method is invoked to fetch the content of a flat file, and populate the TextView (displaying "<text from file>" in Fig. 6.1) on click of the *Read from file* button. The `writeToFile(String text)` method is used to save the text entered in the EditText into a flat file on click of the *Write to file* button.

Snippet 6.1 describes the `writeToFile(String  text)` method. In Line 3, the `openFile-Output()` method is used to create and open a file named `userinput.txt.` The second parameter indicates the permissions on the file. It can take different values such as `MODE_APPEND` (opens file in append mode) and `MODE_PRIVATE` (erases previous data and writes afresh). Using the `write()` method of `OutputStreamWriter`, we can write data into the file, once it is created as shown in Line 4.

```
1  private void writeToFile(String text) {
2    try {
3      OutputStreamWriter outputStreamWriter = new
       OutputStreamWriter(openFileOutput("userinput.txt",
       Context.MODE_PRIVATE));
4      outputStreamWriter.write(text);
5      outputStreamWriter.close();
6    }
7    catch (IOException e) {
8      e.printStackTrace();
9    }
10 }
```

**Snippet 6.1** | Writing to a flat file.

As depicted in Fig. 6.2, the userinput.txt file gets created inside \data\data\com.mad.flatfile\files folder of device/emulator. In Android, data pertaining to any app is maintained in \data\data\<package-name> folder. This folder in turn has subfolders, such as files subfolder to store flat files, or databases subfolder to store relational database files.

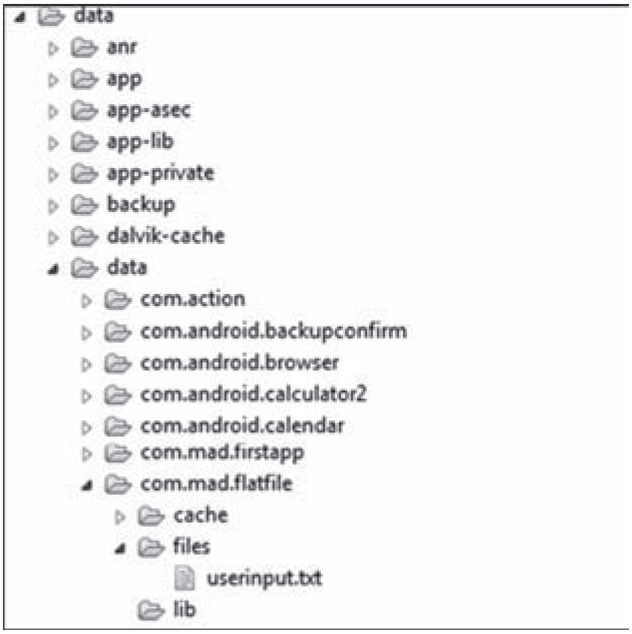


**Figure 6.2** | File Explorer view of Dalvik Debug Monitor Server.

The `getFileContent()` method is used to read data from the file and is described in Snippet 6.2. The `openFileInput()` method is used in Line 4 to open the `userinput.txt` file for reading. In Line 10, the `readLine()` method of `BufferedReader` is used iteratively to read the lines inside the file (userinput.txt). Each line that is read from the file is cumulated using the `append()` method of the `StringBuilder` class, as shown in Line 11. The complete contents of the file are hence obtained and assigned to the `fileContent` variable (Line 14), which is then returned by the `getFileContent()` method (Line 21).

```
1  private String getFileContent() {
2    String fileContent = “”;
3    try {
4     InputStream inputStream = openFileInput(“userinput.txt”);
5     if(inputStream != null) {
6      InputStreamReader inputStreamReader=new
           InputStreamReader(inputStream);
7      BufferedReader bufferedReader = new
           BufferedReader(inputStreamReader);
8      String line = “”;
9      StringBuilder stringBuilder = new StringBuilder();
10     while ((line = bufferedReader.readLine()) != null) {
11       stringBuilder.append(line);
12     }
13     inputStream.close();
14     fileContent = stringBuilder.toString();
15    }
16   } catch (FileNotFoundException e) {
17      e.printStackTrace();
18   } catch (IOException e) {
19      e.printStackTrace();
20   }
21   return fileContent;
22 }
```

**Snippet 6.2** | Reading from a flat file.

So far, we have been writing and reading flat files stored in the internal memory of the device. In many scenarios, an app may need to store large amount of data, leading to the use of a secondary (external) storage media such as an SD card. Android provides `Environment` API to deal with external storage.

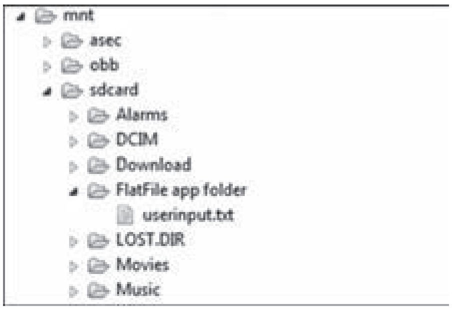
Let us now modify the FlatFile app to store the data in a flat file on an SD card. The modified `getFileContent()` and `writeToFile(String text)` methods are depicted in Snippet 6.3.

```
1  private void writeToFile(String text) {
2    try {
3     File sdCard = Environment.getExternalStorageDirectory();
4     if (sdCard.exists() && sdCard.canWrite()) {
5      File newFolder = new
          File(sdCard.getAbsolutePath()+“/FlatFile app folder”);
6      newFolder.mkdir();
7      if (newFolder.exists() && newFolder.canWrite()) {
8       File textFile = new File(newFolder.getAbsolutePath()+
                                  “/userinput.txt”);
9       textFile.createNewFile();
10      if (textFile.exists() && textFile.canWrite()) {
11        FileWriter fileWriter = new FileWriter(textFile);
12        fileWriter.write(text);
13        fileWriter.flush();
14        fileWriter.close();
15      }
16     }
17    }
18    } catch (IOException e) {
19       e.printStackTrace();
20    }
21  }
22  private String getFileContent() {
23    String fileContent = “”;
24    File sdCard = Environment.getExternalStorageDirectory();
25    if (sdCard.exists() && sdCard.canRead()) {
26     File appFolder = new File(sdCard.getAbsolutePath()+
                             “/FlatFile app folder”);
27     if(appFolder.exists() && appFolder.canRead()) {
28      File textFile = new File(appFolder.getAbsolutePath()+
                             “/userinput.txt”);
29      FileInputStream fileInputStream;
30      try {
31       fileInputStream = new FileInputStream(textFile);
32       BufferedReader bufferedReader= new BufferedReader(
                      new InputStreamReader(fileInputStream));
33       String line = “”;
34       StringBuilder stringBuilder = new StringBuilder();
35       while ((line = bufferedReader.readLine()) != null) {
36         stringBuilder.append(line);
37       }
38       bufferedReader.close();
39       fileContent = stringBuilder.toString();
40     } catch (FileNotFoundException e) {
41        e.printStackTrace();
42     } catch (IOException e) {
43        e.printStackTrace();
44     }
45
46    }
47   }
48   return fileContent;
49  }
```

**Snippet 6.3** | Reading and writing flat files in an SD card.

In Line 3 of Snippet 6.3, a `File` instance referring to the SD card is created using the path of the SD card, which is obtained by invoking the `getExternalStorageDirectory()` method of the `Environment` class. In Lines 5 and 6, an app-specific folder named `FlatFile app folder` is created, if an SD card exists and is writeable. Once the app-specific folder is created, the `userinput.txt` file is created (Lines 8 and 9). Upon creation of the file, the text entered by the user is written into it using the `FileWriter` API, as shown in Lines 11–13.

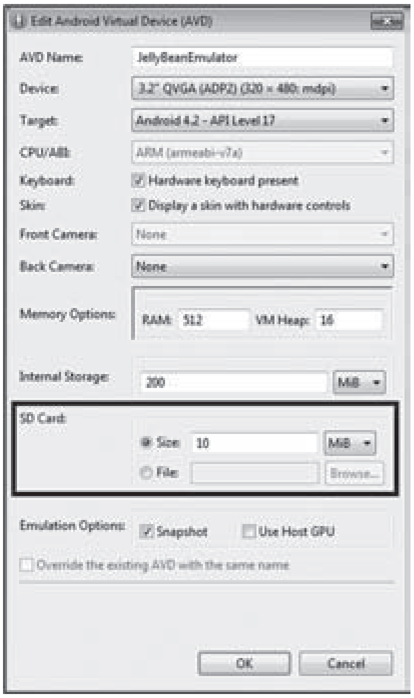
In the `getFileContent()` method (Lines 22–49), a `File` instance of the userinput.txt is created (Lines 24–28) and read from. The `FileInputStream`, `InputStreamReader`, and `BufferedReader` APIs are used to read this file as demonstrated from Lines 31–44. The variance in the use of APIs while doing file operations in the case of SD card as compared to that of internal memory is because the `openFileInput` and `openFileOutput` methods (used in case of internal memory) can be availed only to read files from the internal memory. In addition to the modification in the code, we need to request for the `WRITE_EXTERNAL_STORAGE` permission in the manifest file of the app. The userinput.txt file now gets created in the mnt\sdcard\FlatFile app folder, as shown in Fig. 6.3.



```
mnt
  asec
  obb
  sdcard
    Alarms
    DCIM
    Download
    FlatFile app folder
      userinput.txt
    LOST.DIR
    Movies
    Music
```

**Figure 6.3** | userinput.txt created in an SD card.

When working with emulators, we have to ensure that the SD card is emulated by setting a value in the *Size* field of the *SD Card* option in the Android Virtual Device (AVD) create/edit wizard, as shown in Fig. 6.4. It is advisable not to give huge values in this field because the space required is blocked from the computer's hard drive.



Edit Android Virtual Device (AVD)

AVD Name: JellyBeanEmulator
Device: 3.2" QVGA (ADP2) (320 × 480: mdpi)
Target: Android 4.2 - API Level 17
CPU/ABI: ARM (armeabi-v7a)
Keyboard: ✓ Hardware keyboard present
Skin: ✓ Display a skin with hardware controls
Front Camera: None
Back Camera: None
Memory Options: RAM: 512 VM Heap: 16
Internal Storage: 200 MiB
SD Card: ● Size: 10 MiB ○ File: Browse...
Emulation Options: ✓ Snapshot ☐ Use Host GPU
☐ Override the existing AVD with the same name

OK Cancel

**Figure 6.4** | Emulating an SD card.

## 6.3 SHARED PREFERENCES

Shared preferences is Android’s simple and hassle-free solution to store primitive data in key–value pairs. Key–value pairs are most often used to save user preferences such as ringtone, user-preferred app settings, etc. Shared preferences provides support for persisting boolean, float, int, long, and String data types.

Shared preferences stores data in an XML file in the internal memory of the device. The creation, storage, and manipulation of the XML file are internally taken care by the `SharedPreferences` API.

Let us explore the `SharedPreferences` API by creating an app – SharedPreference – that sends an SMS to the caller when the call is unattended (rejected or missed). In this app, the user can set the message to be sent to a caller with a custom signature. A mechanism to disable the sending of SMS is also provided.

The SharedPreference app will contain two components: `PreferenceSettingsActivity` (as depicted in Fig. 6.5) and `SMSSender`. The `SMSSender` class is a Broadcast Receiver that listens to changes in phone states, determines if the call was unattended, and sends the SMS. The `PreferenceSettingsActivity` enables the user to customize required settings when the *Save settings* button is clicked.

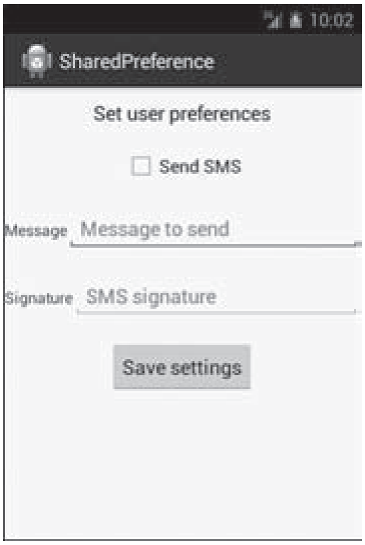


**Figure 6.5** | SharedPreference app.

Snippet 6.4 describes the saving of user settings using the `SharedPreferences` API in the `PreferenceSettingsActivity`.

```
SharedPreferences
preferences=getSharedPreferences(“SMSPreferences”,MODE_PRIVATE);
btnSave.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View arg0) {
   Editor editor=preferences.edit();
   editor.putBoolean(“SendSMS”, chkEnable.isChecked());
   editor.putString(“Message”, etMessage.getText().toString());
   editor.putString(“Signature”,
   etSignature.getText().toString());
9     editor.commit();
10   }
11 });
```

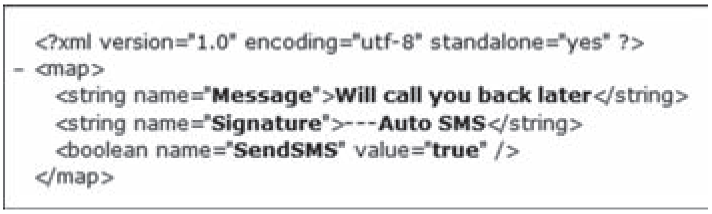
**Snippet 6.4** | Saving user settings using `SharedPreferences` API.

In Line 1, a shared preferences XML file named `SMSPreferences` is created with `MODE_PRIVATE` permission using the `getSharedPreferences()` method. In the `onClick()` event handler of *Save settings* button, an instance of `Editor` is obtained using the `edit()` method of the `SharedPreferences` object (Line 5). In Lines 6–8, the values entered by the user are saved using methods of `editor`. The `Editor` API offers various methods to save data in shared preferences as summarized in Table 6.1.

**Table 6.1** | Methods offered by `Editor` to save data in shared preferences

| **Method** | **Description** |
|---|---|
| `putBoolean(String key, boolean value)` | Saves a boolean value against a key |
| `putFloat(String key, float value)` | Saves a float value against a key |
| `putInt(String key, int value)` | Saves an integer value against a key |
| `putLong(String key, Long value)` | Saves a long value against a key |
| `putString(String key, String value)` | Saves a String value against a key |
| `putStringSet(String key, Set values)` | Saves a set of String values against a key |

Data saved using the `SharedPreferences` API gets saved in data\data\<package-name>\shared_prefs folder. The contents of SMSPreferences.xml against a sample run of the app is shown in Fig. 6.6.



```
  <?xml version="1.0" encoding="utf-8" standalone="yes" ?>
- <map>
    <string name="Message">Will call you back later</string>
    <string name="Signature">---Auto SMS</string>
    <boolean name="SendSMS" value="true" />
  </map>
```

**Figure 6.6** | Contents of SMSPreferences.xml.

Snippet 6.5 depicts the `sendSMS()` method of the `SMSSender` class that is used to retrieve the data saved by the user and send an SMS when a call is unattended.

```
1  private void sendSMS() {
2   SharedPreferences preferences=
    context.getSharedPreferences("SMSPreferences",
                                 context.MODE_PRIVATE);
3   boolean sendSms=preferences.getBoolean("SendSMS", false);
4   String message=preferences.getString("Message", "");
5   String signature=preferences.getString("Signature", "");
6   if(sendSms==true)
7   {
```

```
8    //Send the SMS to the caller
9   }
10 }
```

**Snippet 6.5** | `sendSMS()` method of `SMSSender`.

In Line 2, the shared preferences XML file is retrieved using the `getSharedPreferences()` method. In Lines 3–5, the `getBoolean()` and `getString()` methods are used to retrieve the value of preferences saved. These methods accept two parameters: the key used to store the data and a default value that will be returned in case a value corresponding to the supplied key is unavailable. The scenario of configuring and storing user preferences is quite common in mobile apps. Android has recognized it, and provides a `Preference` API that works with shared preferences to standardize the configuration and storage of user preferences and present a consistent user experience.

## 6.4 LET'S APPLY

In this section, we shall create a login module for authenticating registered users in the 3CheersCable app. We shall create an Activity – `LoginActivity` – which prompts the user to enter the login id and password, as shown in Fig. 6.7. On click of the *Login* button, the user's credentials are verified at the server, and if valid, the app proceeds to further screens in the app workflow. Additionally, we shall make the app insightful by using shared preferences to save this login data and display it to the user. The user will only need to click the *Login* button for subsequent logins, thus avoiding entering the data everytime.

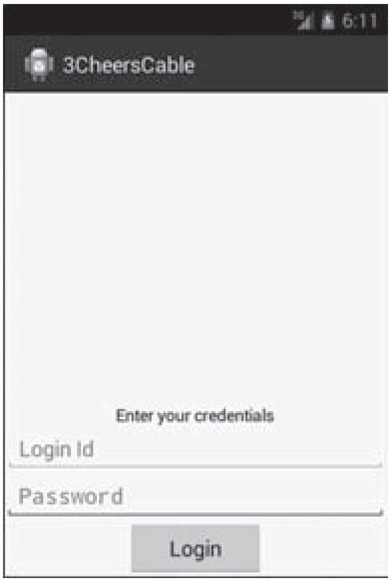


**Figure 6.7** | `LoginActivity`.

Create `LoginActivity` as shown in Fig. 6.7. Get handles for Login Id and Password EditTexts, and the Login button. In the `onCreate()` method of `LoginActivity`, read saved data from shared preferences, and populate the EditTexts with saved values of Login Id and Password.

```
prefs = PreferenceManager.getDefaultSharedPreferences(this);
if(!prefs.getString(AppConfig.USER_NAME,“”).equalsIgnoreCase(“”)
 &&!prefs.getString(AppConfig.PASSWORD,“”).equalsIgnoreCase(“”))
{
 loginIdField.setText(prefs.getString(AppConfig.USER_NAME, “”));
 passwordField.setText(prefs.getString(AppConfig.PASSWORD, “”));
}
```

In the above program, we have used the `PreferenceManager` API's `getDefaultSharedPreferences()` method to retrieve the shared preferences file. This method automatically creates a shared preferences XML file with the Activity name as the filename.

In the `onClick()` event handler of the Login button, call the `WebServiceHitter` AsyncTask with the credentials entered by the user (see Line 15 of the following program). It is advisable to first validate if the user has entered any data before making the Web service call.

```
1  loginId = loginIdField.getText().toString();
2  password = passwordField.getText().toString();
3  ...
4  if(!loginId.equalsIgnoreCase(“”)&&!password.equalsIgnoreCase(“”)
   ){
5   progDialog = ProgressDialog.show(this,    “Please    wait...”,
                                     “Authenticating...”);
6   progDialog.setCancelable(true);
7   progDialog.setOnCancelListener(new OnCancelListener(){
8   @Override
9   public void onCancel(DialogInterface dialog){
10    Toast.makeText(LoginActivity.this, “You’ve cancelled
      login operation. App will exit now.”,
      Toast.LENGTH_LONG).show();
11    asyncTask.cancel(true);
12    finish();
13  }
14 });
15 asyncTask = new
   WebServiceHitter(LoginActivity.this).execute(GenerateURLs.get-
   PostURL(Operation.LOGIN, loginId,password, null));
16 }
17 else{
18  Toast.makeText(this, “Please enter Login id & Password”,
    Toast.LENGTH_LONG).show();
19 }
```

The response from the server is verified to check if the user is authorized to access the app. Upon successful authentication, the values for Login Id and Password entered by the user are saved into shared preferences and the user is taken to `MainActivity` (see Line 11 of the following program). Appropriate error messages are displayed as Toast when authentication fails or when server connectivity could not be established (see Lines 15–21 in the following program).

```
1  public void onNetworkCallComplete(WebResponse object) {
2    if(object.getResponseCode() == ResponseCode.SUCCESS){
  LoginBean loginresponse=
   LoginResponseParser.parseLoginResponse
   (object.getResponse().toString());
  if(progDialog!=null && progDialog.isShowing())
   progDialog.dismiss();
   if(loginresponse!=null && loginresponse.getResult()==1 ){
     Editor editor = prefs.edit();
     editor.putString(AppConfig.USER_NAME,
     loginresponse.getUserName());
     editor.putString(AppConfig.PASSWORD,
     loginresponse.getPassword());
     editor.commit();
     Intent intent = new Intent(this, MainActivity.class);
     startActivity(intent);
     finish();
  }
  else{
    Toast.makeText(this,"Wrong Login id/Password",
    Toast.LENGTH_LONG).show();
    ...
  }
 }
 else{
   Toast.makeText(this, "Unable to contact server",
   Toast.LENGTH_LONG).show();
 }
}
```

## 6.5 RELATIONAL DATA

So far, we have learnt to store unstructured and primitive data on the device using flat files and shared preferences. However, at times, mobile apps may need to store data in relational format due to the features that the databases bring in such as reduced data redundancy, atomicity, consistency, isolation, durability, and querying capability. Mobile databases have different set of challenges that a developer needs to be wary of. These databases have to be really light weight as devices have low disk space, must utilize limited processing power to save system resources, and should be resilient to work in an unfriendly environment where power failure (due to unexpected battery outage) or disk failure (due to device drop) may occur. For such scenarios, Android provides SQLite database and related APIs.

SQLite is a popular, open source, relational database management system (RDBMS) that is widely used in mobile platforms such as Android, Apple iOS, BlackBerry®, and Symbian. The database is implemented in C and is very light, around 500 KiB. The SQLite database instance runs as part of the app with which it is associated, rather than in a separate process as is a typical case with enterprise databases. It implements most of the SQL standards. It has weakly typed syntax. It does not support referential integrity constraint, which can however be enforced using triggers. It does not support some form of Alter table statement. These are minor issues that mostly do not affect its usage as a light-weight database for mobile apps.

An Android app that intends to use database has to manage both the definition and manipulation aspects of data in it. It includes creation of database and tables, and CRUD (create, retrieve, update, and delete) operations. These data-related aspects have to be tightly coupled with the app functionality. It is a standard practice to organize the data definition and manipulation aspects using an Adapter that acts as a conduit between the relational database and other app components (functionalities).

Let us now explore the SQLite database and related APIs using a simple app – ExpenseTracker – that helps users to track their day-to-day expenses, as depicted in Fig. 6.8. A user can add expenses using *ADD EXPENSE* action item, modify an existing expense by long tapping on it, and delete an expense by tapping on it. The data in this app is managed using `ExpenseDBAdapter` – a user-defined Adapter class that is created to manage both the definition and manipulation aspects of expense data.

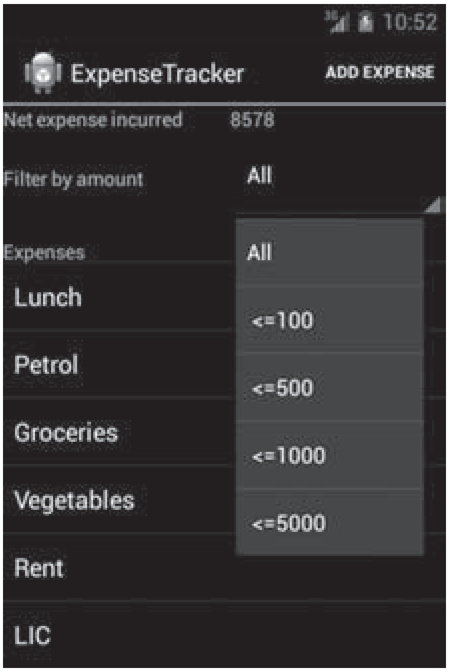


**Figure 6.8** | ExpenseTracker app.

Snippet 6.6 describes a portion of the `ExpenseDBAdapter`. In the constructor (Lines 14–18) of `ExpenseDBAdapter`, we create an object of the `MyDBHelper` class (Line 17). The `MyDBHelper` class (Lines 19–35) is a user-defined class that is a subclass of the `SQLiteOpenHelper` class, and is used to abstract data definition aspects from the Adapter class. The constructor of `MyDBHelper` accepts four parameters: context, database name, `CursorFactory` object, and database version. The `CursorFactory` instance is used by the helper to create a `Cursor` object. This `Cursor` object is used to read results from queries to the database. The database version plays an important role in identifying the version change during database upgrades.

The database with the specified name and version is created by invoking the constructor of `SQLiteOpenHelper` with required parameters (Line 23). The database created by an Android app will reside in data\data\<package_name>\databases folder. Upon creation of this database, the `onCreate()` callback method of `MyDBHelper` is invoked with a `SQLiteDatabase` object. In this method, we create the necessary tables in the database using the `execSQL()` method of the `SQLiteDatabase` API (Lines 26–28). The `execSQL()` method can be used only for data definition statements.

Database upgrades may happen when the user updates an app or when app is automatically updated on-the-air. In the case of database upgrades, the `onUpgrade()` callback method is invoked (Lines 30–34). A database upgrade is achieved by modifying the version number passed to the constructor of `MyDBHelper`. The `onUpgrade()` method can be used to serve different purposes based on app requirements. In this case, we are wiping off all the existing data from the database and creating the tables afresh.

```
1  public class ExpenseDBAdapter {
2
3    private static final String DB_NAME=“Expense_Database.db”;
4    private static final String TABLE_NAME=“Expenses_Table”;
5    private static final int DB_VERSION=1;
6    private static final String KEY_ID=”_id”;
7    private static final String COLUMN_EXPENSE=“expense”;
8    private static final String COLUMN_AMOUNT=“amount”;
9    private static final String TABLE_CREATE=“create table
     “+TABLE_NAME+”
     (“+KEY_ID+” integer primary key “+ “autoincrement
     ,”+COLUMN_EXPENSE+” text not null, “+COLUMN_AMOUNT+” integer
      not null);”;
10   private SQLiteDatabase expenseDatabase;
11   private final Context context;
12   private MyDBHelper helper;
13
14   public ExpenseDBAdapter(Context context)
15   {
16    this.context=context;
17    helper=new MyDBHelper(context, DB_NAME, null, DB_VERSION);
18   }
19   private static class MyDBHelper extends SQLiteOpenHelper
20   {
21    public MyDBHelper(Context context,String name, CursorFactory
      cursorFactory, int version)
22    {
23       super(context, name, cursorFactory, version);
24    }
25    @Override
26    public void onCreate(SQLiteDatabase db) {
27      db.execSQL(TABLE_CREATE);
28    }
29    @Override
30    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
      newVersion) {
31      Log.w(“Updation”, “Database version is being updated”);
32      db.execSQL(“DROP TABLE IF EXISTS “+TABLE_NAME);
33      onCreate(db);
34     }
35    }
36    public ExpenseDBAdapter open()
```

```
37    {
38      expenseDatabase=helper.getWritableDatabase();
39      return this;
40   }
41   public void close() {
42       expenseDatabase.close();
43   }
```

**Snippet 6.6** | `ExpenseDBAdapter` with `MyDBHelper`.

Two user-defined methods, `open()` and `close()`, are included in the `ExpenseDBAdapter` (Lines 36–40 and Lines 41–43). The `open()` method is used to instantiate the `SQLiteDatabase` instance and the `close()` method is used to dereference it. In our example, the `getWritable Database()` method of `SQLiteHelper` is used in the `open()` method to instantiate the `expenseDatabase` in writeable mode (Line 38). For cases where a database needs to be instantiated in read-only mode, the `getReadableDatabase()` method of `SQLiteOpenHelper` can be used. The `open()` and `close()` methods are invoked from app components such as an Activity, as described in Snippet 6.7.

```
1  protected void onCreate(Bundle savedInstanceState) {
2    super.onCreate(savedInstanceState);
3    setContentView(R.layout.activity_main);
4    expensesDatabase = new
     ExpenseDBAdapter(getApplicationContext());
5    expensesDatabase.open();
6    ...
7  }
8  protected void onDestroy() {
9     super.onDestroy();
10    expensesDatabase.close();
11 }
```

**Snippet 6.7** | Invoking `open()` and `close()` methods from an Activity.

Once the `SQLiteDatabase` instance is initialized, we are now ready to read and write in the database. We need to define the methods to read and write data inside `ExpenseDBAdapter`. To read the data, that is, to query database using SELECT SQL statement, the `query()` method of `SQLiteDatabase` is used. Snippet 6.8 describes two user-defined methods, `getAllExpenses()` and `getExpenses-WithinRange()`, to read data from the `Expenses_Table`. The `getAllExpenses()` returns all tuples, whereas the `getExpensesWithinRange()` returns only those tuples whose `COLUMN_AMOUNT` value is less than the amount requested by the end-user (as depicted in Fig. 6.8).

```
1  public Cursor getAllExpenses()
2  {
3    return expenseDatabase.query(TABLE_NAME, null, null, null,
   null, null, null);
4  }
5  public Cursor getExpensesWithinRange(int amount)
6  {
```

```
7    return expenseDatabase.query(TABLE_NAME, null, COLUMN_AMOUNT+”
     <= “+amount, null, null, null, null);
8  }
```

**Snippet 6.8** | Reading from database.

Lines 3 and 7 describe the `query()` method that accepts seven arguments. The first argument is the name of the table that is being queried. The second argument is the column(s) to be retrieved. It is passed as an array of String containing the column names; `null` indicates all the columns. The third argument is the selection criteria to filter data using `WHERE` SQL clause (Line 7); `null` represents no selection criteria. The fourth argument is an array of String representing the selection arguments[1] that replaces `?`s present in the `WHERE` SQL clause. The remaining three arguments refer to `GROUP BY, HAVING`, and `ORDER BY` SQL clauses.

Snippet 6.9 describes the invocation of the `getAllExpenses()` method of `ExpenseDBAdapter` from the Activity. This method returns a `Cursor` object, using which we can read the data from the result set of the query. `getExpenses` is the `Cursor` instance defined in the snippet, and it is used to populate the ListView displaying all expenses. (see Fig. 6.8)

```
1  getExpenses = expensesDatabase.getAllExpenses();
2  void refreshViews() {
3  ...
4   expenseIds = new ArrayList<Integer>();
5   expenseText = new ArrayList<String>();
6   expenseAmount=new ArrayList<Integer>();
7   while (getExpenses.moveToNext()) {
8     expenseIds.add(getExpenses.getInt(0));
9     expenseText.add(getExpenses.getString(1));
10    expenseAmount.add(getExpenses.getInt(2));
11   }
12   expensesAdapter = new
     ArrayAdapter<String>(getApplicationContext(),
     android.R.layout.simple_list_item_1, expenseText);
13   showExpenses.setAdapter(expensesAdapter);
14  }
```

**Snippet 6.9** | Usage of `Cursor` in Activity.

The `Cursor` contains methods such as `getCount()` (to get the number of rows in the result set), `get–ColumnCount()` (to get the number of columns returned in the result set), and various other methods such as `getString(), getInt(), getLong()` (to read data from a particular column in the result set, see Lines 8–10 of Snippet 6.9). Methods such as `getString(), getInt()`, and `getLong()` accept an integer as argument, indicating the index of the column from which the data has to be accessed. Column indexes begin from 0.

The `movetoNext()` method of the `Cursor` instance helps iterate over multiple rows of the result set obtained, and has to be invoked before accessing any data using the `Cursor` (Line 7). This method has to be invoked even before accessing the first row of the `Cursor`.

[1] Example of query with selection arguments: `query(TABLE_NAME, null, COLUMN_AMOUNT+” <= ? ”, new String[]{ amount }, null, null, null)`

In case we need to execute an SQL `SELECT` statement directly, the `rawQuery()` method of `SQLiteDatabase` can be used. In Snippet 6.10, we have used this method to directly execute the SQL statement to obtain the sum of all expense amounts from `Expenses_Table`.

```
public int getTotalExpense(){
 Cursor expensesSum=expenseDatabase.rawQuery(“SELECT SUM(amount)
 FROM Expenses_Table”, null);
 if(expensesSum!=null){
   expensesSum.moveToNext();
   return expensesSum.getInt(0);
 }
 else
   return 0;
 }
```

**Snippet 6.10** | Usage of `rawQuery()` method.

The `insert()` method of `SQLiteDatabase` is used to insert data into the table, as shown in Snippet 6.11. This method accepts three arguments, namely, the table name, default insert values, and an instance of `ContentValues`.

```
1  public long addExpense(String expense,int amount) {
2    ContentValues contentValues=new ContentValues();
3    contentValues.put(COLUMN_EXPENSE, expense);
4    contentValues.put(COLUMN_AMOUNT, amount);
5    return expenseDatabase.insert(TABLE_NAME,null,contentValues);
6  }
```

**Snippet 6.11** | Inserting data.

`ContentValues` is basically a key–value pair with the column name as key, and the data that needs to be inserted in the column as value. To insert a tuple in a table, we have to create a `ContentValues` instance, and add all key–value pairs in it, using the `put()` method (Lines 3 and 4).

The `delete()` and `update()` methods of `SQLiteDatabase` are used to delete and update data, respectively, as shown in Snippet 6.12. The `delete()` method (Line 2) accepts three arguments – table name, `WHERE` clause to define the criteria for deletion, and an array of String representing the selection arguments that replaces `?`s present in the `WHERE` SQL clause. In this example, we intend to delete an expense based on `KEY_ID` (primary key of `Expenses_Table`), therefore, the `KEY_ID` is matched against the `expenseId` (id corresponding to the selected expense in the ListView in Fig. 6.8). The return value of the `delete()` method is the number of tuples affected in the database.

```
1  public boolean deleteExpense(long expenseId) {
2    return expenseDatabase.delete(TABLE_NAME, KEY_ID+” =
     “+expenseId, null)>0;
3  }
4  public int updateExpense(long expenseId,String expense,int amount) {
5    ContentValues updateValues=new ContentValues();
6    updateValues.put(COLUMN_EXPENSE, expense);
7    updateValues.put(COLUMN_AMOUNT, amount);
8    return expenseDatabase.update(TABLE_NAME,updateValues,
     KEY_ID+" = "+expenseId, null);
9  }
```

**Snippet 6.12** | Deleting and updating data.

The `update()` method (Line 8)accepts four arguments: table name, `ContentValues` instance, `WHERE` clause to define the criteria for updation, and an array of String representing the selection arguments that replaces `?`s present in the `WHERE` SQL clause. The return value of the `update()` method is the number of tuples affected in the database.

## 6.6 DATA SHARING ACROSS APPS

Android apps do not share their data with each other as a security measure. However, there may be legitimate scenarios where an app has to share its data with other apps. For example, if an app needs to send an autogenerated SMS to the select few contacts, then it needs a mechanism to access the contacts stored in the Contacts app.

To achieve this, Android provides the Content Provider. It hides the implementation details of the data from other apps to provide an abstract and secure way of sharing data across apps. Using the Content Provider, we can carry out CRUD operations on data of other apps as a black box. Android provides many in-built Content Providers using which the data of in-built apps are made accessible. We can also create custom Content Providers, so that other apps can access our app's data.

### 6.6.1 In-Built Content Provider

Android provides in-built Content Providers so that a developer can conveniently access some commonly used data sets in their apps. `CalendarContract.CalendarColumns` is one such in-built Content Providers for accessing calendar-related data; `MediaStore.Audio.AlbumColumns` is another one and is used for accessing album data of an audio file.

All Content Providers expose their data sets to other apps using CONTENT_URIs. The apps accessing this data set use the `ContentResolver` API that provides a mechanism to carry out CRUD operations in an abstract manner on the Content Provider. It gives out an impression that data resides in databases, regardless of the format in which it is actually stored.

To demonstrate an in-built Content Provider, let us create an app that accesses the contacts stored in an in-built Contacts app. Figure 6.9(a) shows a snapshot of the Contacts app, and Fig. 6.9(b) shows this information being rendered in a ListView in our app. To accomplish this, we shall use the `ContactsContract.CommonDataKinds.Phone` in-built Content Provider.

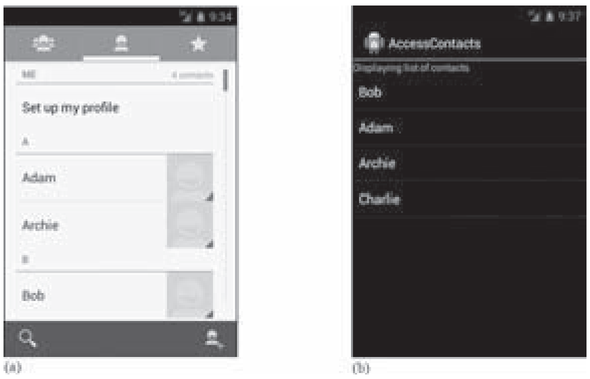


**Figure 6.9** | (a) Contacts app. (b) Contacts in our app.

The `query()` method of the `ContentResolver` class is used to query data from Content Providers (exposed using CONTENT_URI). This method returns a `Cursor` through which the data set can be accessed. Managing this `Cursor` on the *main* thread is discouraged as the retrieval of data from Content Providers is typically a long-running task. The `Loader` API is used to overcome this problem. Each Activity/Fragment can use a `Loader` to asynchronously load data and update its UI.

Snippet 6.13 describes the use of `Loader` to display the list of contacts retrieved from the `ContactsContract.CommonDataKinds.Phone` in-built Content Provider in the `MainActivity`. In Line 7, an instance of `LoaderManager` is obtained using the `getLoaderManager()` method, and is initialized using the `initLoader()` method. The `initLoader()` method accepts three arguments: an integer representing a unique identifier for each instance of `Loader`, a `Bundle` object to pass data, and the class implementing `LoaderCallbacks` using which the data set from the Content Provider is retrieved (Activity in this case). The `LoaderManager` is used to manage different loaders that can be used in an Activity/Fragment.

```
1  public class MainActivity extends Activity implements
   LoaderCallbacks<Cursor> {
2   ArrayList<String> contacts = new ArrayList<String>();
3   ListView displayContactsListView;
4   @Override
5   protected void onCreate(Bundle savedInstanceState) {
6     ...
7     getLoaderManager().initLoader(0, null, this);
8   }
9   @Override
10  public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1){
11    if (arg0 == 0) {
12      return new CursorLoader(MainActivity.this,
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
        null, null, null);
13    }
14    return null;
15   }
16   @Override
17   public void onLoadFinished(Loader<Cursor> arg0, Cursor arg1){
18     int idContactName = arg1.getColumnIndexOrThrow
       (ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME);
19     while (arg1.moveToNext()) {
20       contacts.add(arg1.getString(idContactName));
21     }
22     ListAdapter adapter = new
       ArrayAdapter<String>(getApplicationContext(),
       android.R.layout.simple_list_item_1, contacts);
23     displayContactsListView.setAdapter(adapter);
24   }
25   @Override
26   public void onLoaderReset(Loader<Cursor> arg0) {
27
28    }
29  }
```

**Snippet 6.13** | Accessing Content Provider through `Loader`.

The invocation of `initLoader()` triggers the `onCreateLoader()` callback method (Lines 10–13). This method retrieves the data from the Content Provider and returns it as a `Loader`. In this example, a `CursorLoader` (subclass of `Loader`) instance is returned from this method, as shown in Line 12. The constructor of `CursorLoader` accepts six arguments: context, `CONTENT_URI`, columns to be retrieved, SQL `WHERE` clause, selection arguments for the SQL `WHERE` clause, and the sort order. In this example, all columns and rows from the Content Provider are obtained because both column projection (second argument) and `WHERE` clause (third argument) are `null`. The resulting data set from the `onCreateLoader()` is passed on to the `onLoadFinished()` callback (Lines 17–24) as a `Cursor`.

In the `onLoadFinished()` callback method, we are fetching the index of the `DISPLAY_NAME` column in the result set (Line 18) and retrieving the display name for each contact from the `Cursor` (Line 20). The retrieved display names are used to populate the ListView, as depicted in Fig. 6.9(b). The `onLoaderReset()` callback (Line 26) gets invoked when the `Loader` is reset, and in this method, we may dereference any instances of the `Loader` used. We also need to request the `READ_CONTACTS` permission in the app manifest file so that we can access the contacts from the in-built Content Provider in our app.

### 6.6.2 Custom Content Provider

Custom Content Providers come handy when we want to expose data of our own apps to be accessed by other apps. Similar to in-built Content Providers, custom Content Providers also expose their data sets to other apps using unique Strings known as CONTENT_URIs. A typical CONTENT_URI is composed of three components – scheme, authority, and path – and is defined as scheme://authority/path. The scheme for Content Providers is always *content*. The authority is a unique string to identify the provider. The usual convention followed for defining an authority is <app_package_name>.<app_name>.<provider_name>. Path refers to the name of the underlying table of the app database, or a specific functionality that is exposed using the CONTENT_URI.

Let us now implement a custom Content Provider in the existing ExpenseTracker app (introduced in Section 6.5) to expose the expense data. We are also going to create a new app, ExpenseProviderClient, which will consume the exposed data. The custom Content Provider implementation starts with creating a class, which extends `ContentProvider`, and defining its CONTENT_URIs.

Snippet 6.14 shows a portion of the `ExpenseProvider`, our custom Content Provider, listing down various variable initializations that will be used in its different parts. `content://com.mad.expensetracker.expenseprovider/EXPENSES_TABLE` is used as a CONTENT_URI – `CONTENT_URI1` (Lines 2 and 3) – to expose the entire data of the `EXPENSES_TABLE`. `content://com.mad.expensetracker.expenseprovider/SUM` is used as another CONTENT_URI – `CONTENT_URI2` (Line 4) – to expose the functionality that aggregates all expenses.

```
1  public static final String AUTHORITY = “com.mad.expensetracker.
   expenseprovider”;
2  public static final String PATH = “EXPENSES_TABLE”;
3  public static final Uri CONTENT_URI1 = Uri.parse(“content://” +
   AUTHORITY+ “/” + PATH);
```

```
4  public static final Uri CONTENT_URI2 = Uri.parse("content://" +
   AUTHORITY+ "/SUM");
5  public static final String CONTENT_TYPE = ContentResolver.
   CURSOR_DIR_BASE_TYPE + "/" + PATH;
6  public static final String CONTENT_ITEM_TYPE = ContentResolver.
   CURSOR_ITEM_BASE_TYPE + "/" + PATH;
7  public static final int ALLEXPENSES = 1;
8  public static final int SPECIFICEXPENSE = 2;
9  public static final int SUMEXPENSES = 3;
10 private static final UriMatcher matcher =newUriMatcher
   (UriMatcher.NO_MATCH);
11 static {
12   matcher.addURI(AUTHORITY, PATH, ALLEXPENSES);
     // paths matching CONTENT_URI1
13   matcher.addURI(AUTHORITY, PATH + "/#", SPECIFICEXPENSE);
     // paths matching CONTENT_URI1+"/id"
14   matcher.addURI(AUTHORITY, "SUM", SUMEXPENSES);
     // paths matching CONTENT_URI2
15 }
16 ExpenseDBAdapter db;
```

**Snippet 6.14** | Variable initializations in `ExpenseProvider`.

`CONTENT_TYPE` and `CONTENT_ITEM_TYPE` (Lines 5 and 6) are user-defined MIME (Multipurpose Internet Mail Extensions) types that represent the underlying table and row data type, respectively. The value of `CONTENT_TYPE` is a combination of `ContentResolver.CURSOR_DIR_BASE_TYPE` constant and the path of `CONTENT_URI`. The `ContentResolver.CURSOR_DIR_BASE_TYPE` constant holds a predefined value – `vnd.android.cursor.dir`. The value of `CONTENT_ITEM_TYPE` is a combination of `ContentResolver.CURSOR_ITEM_BASE_TYPE` constant and the path of `CONTENT_URI`. However, the value of `ContentResolver.CURSOR_ITEM_BASE_TYPE` constant is `vnd.android.cursor.item`, and it refers to individual rows of the underlying table. These MIME types can be accessed from client apps to identify the kind of data being retrieved.

A Content Provider may be exposing multiple sets of data, and we need to uniquely identify them. For example, the `ExpenseProvider` exposes two different data sets – the entire table (`EXPENSES_TABLE`) and the sum of expenses. To resolve a data set uniquely, the `UriMatcher` class is used. We need to add all the CONTENT_URIs to the `UriMatcher` object that represent these different data sets, as shown in Lines 11–15. In this case, when the requested data set is the entire table (`EXPENSES_TABLE`), the `ALLEXPENSES` constant is used as a unique identifier. Similarly, the `SPECIFICEXPENSE` and `SUMEXPENSES` unique identifiers are used to identify a single row of the underlying table and sum of expenses, respectively. The `UriMatcher` object can be used at various places inside the Content Provider to identify the data set on which the data manipulation operations need to be performed.

To handle various data manipulation operations, we have to override the `query()`, `insert()`, `update()`, and `delete()` methods of `ContentProvider`, as shown in Snippet 6.15. In the `query()` method, the functionality to query the total expense is implemented (Lines 2–11). It is invoked in response to the `query()` method of `ContentResolver` from the client app. The `Uri` argument (`arg0`) is matched using the `UriMatcher` object to uniquely identify the data set on which `SQL SELECT` operation has to be performed (Line 3). In this example, sum of expenses is retrieved from the `EXPENSES_TABLE`

(Line 7). Using the `insert()` method (Lines 13–27), we implement insert functionality on a data set of the `ExpenseProvider`. It is invoked in response to the `insert()` method of `ContentResolver` from the client app. In this example, expense data is getting inserted into `EXPENSES_TABLE` (Line 19). Here, the Content Provider is not providing the update and the delete functionalities to the client app(s). In such scenarios, it is recommended that methods intended for such functionalities [`update()` and `delete()` methods in this case] should throw `UnsupportedOperationException`, as shown in Lines 30 and 34.

In addition to these methods, the `onCreate()` and `getType()` methods are also implemented. The `onCreate()` method gets fired at app launch, and is used to initialize the underlying files or databases that the Content Provider is using (Lines 39–42). The `getType()` method returns the MIME type of the requested data set (Lines 44–55). It is invoked in response to the `getType()` method of `ContentResolver` from the client app.

```
1   @Override
2   public Cursor query(Uri arg0, String[] arg1, String arg2,
                        String[] arg3,String arg4) {
3    int uriType = matcher.match(arg0);
4    switch (uriType) {
5    case SUMEXPENSES:
6     db.open();
7     return db.getTotalExpenseCursor();
8   default:
9    throw new UnsupportedOperationException();
10  }
11 }
12  @Override
13  public Uri insert(Uri arg0, ContentValues arg1) {
14    int uriType = matcher.match(arg0);
15    long id = 0;
16    switch (uriType) {
17    case ALLEXPENSES:
18     db.open();
19     id = db.addExpense(
       arg1.get(ExpenseDBAdapter.COLUMN_EXPENSE).toString(),
       Integer.parseInt(arg1.get(ExpenseDBAdapter.COLUMN_AMOUNT).
       toString()));
20     db.close();
21     break;
22   default:
23    throw new UnsupportedOperationException();
24   }
25   getContext().getContentResolver().notifyChange(arg0, null);
26   return Uri.parse(PATH + "/" + id);
27  }
28
29  @Override
30  public int update(Uri arg0, ContentValues arg1, String arg2,
   String[] arg3)throws UnsupportedOperationException {
     return 0;
   }
   @Override
   public int delete(Uri arg0,String arg1,String[] arg2)throws
   UnsupportedOperationException {
     return 0;
   }

   @Override
   public boolean onCreate() {
     db = new ExpenseDBAdapter(getContext());
     return true;
   }
   @Override
   public String getType(Uri arg0) {
     switch (matcher.match(arg0)) {
     case ALLEXPENSES:
      return CONTENT_TYPE;
     case SPECIFICEXPENSE:
      return CONTENT_ITEM_TYPE;
     case SUMEXPENSES:
      return CONTENT_ITEM_TYPE;
     default:
      return null;
     }
   }
```

**Snippet 6.15** | Various methods in `ExpenseProvider`.

Once the Content Provider is implemented, it needs to be registered with attributes – `name`, `authorities`, and `exported` – in the app manifest file, as shown in Snippet 6.16. The `authorities` attribute of the `<provider>` tag has to be assigned with all authorities used in the exposed CONTENT_URIs of a Content Provider. The `exported` attribute is used to control the access of a Content Provider. If the value is set to `true`, then the Content Provider is made accessible to other apps. The default value of this attribute is `false`.

```
<provider
  android:name="ExpenseProvider"
  android:authorities="com.mad.expensetracker.expenseprovider"
  android:exported="true">
</provider>
```

**Snippet 6.16** | Registering `ExpenseProvider` in manifest.

Figure 6.10 depicts the client app – ExpenseProviderClient – that accesses the expense data from the Expense Tracker app, exposed through the `ExpenseProvider`. This app accesses both the exposed functionalities of the `ExpenseProvider`, viz., inserting an expense and querying sum of all expenses.

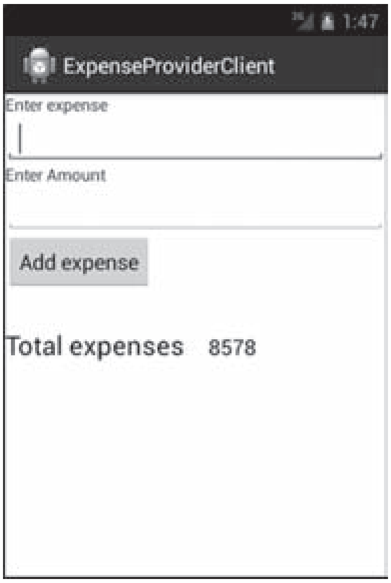


**Figure 6.10** | ExpenseProviderClient app.

To access the exposed data of the ExpenseTracker app, the client app uses the `ContentResolver` in conjunction with `Loader`. Snippet 6.17 demonstrates the implementation of add expense functionality. In the `onClick()` event handler of the *Add expense* button, the `insert()` method of the `ContentResolver` is used to insert the entered expenses into the `ExpenseProvider` (Line 22). The `allExpenseUri` used here (Line 4) holds the value `content://com.mad.expensetracker.expenseprovider/EXPENSES_TABLE` (`CONTENT_URI1` in Snippet 6.14) through which the `ExpenseProvider` exposes the add expense functionality.

```
1  public class MainActivity extends Activity implements
   LoaderCallbacks<Cursor>{
2  ...
3  public static final int QUERY_CODE=1;
4  Uri allExpenseUri=Uri.parse(
   "content://com.mad.expensetracker.expenseprovider/
    EXPENSES_TABLE");
5  Uri sumOfExpenseUri=Uri.parse(
   "content://com.mad.expensetracker.expenseprovider/SUM");
6  @Override
7  protected void onCreate(Bundle savedInstanceState) {
8   super.onCreate(savedInstanceState);
9   setContentView(R.layout.activity_main);
10  addExpense = (Button) findViewById(R.id.button1);
11  expense = (EditText) findViewById(R.id.editText1);
12  amount = (EditText) findViewById(R.id.editText2);
13  textView = (TextView) findViewById(R.id.showTotalExpense);
14  contentResolver = getContentResolver();
15  getLoaderManager().initLoader(QUERY_CODE, null, this);
16   addExpense.setOnClickListener(new OnClickListener() {
17    @Override
18    public void onClick(View arg0) {
19     ContentValues values = new ContentValues();
20     values.put("expense", expense.getText().toString());
21     values.put("amount", amount.getText().toString());
22     contentResolver.insert(allExpenseUri, values);
23     getLoaderManager().restartLoader(QUERY_CODE,null,
       MainActivity.this);
24    }
25   });
26  }
```

**Snippet 6.17** | Adding expenses from the client app.

The sum of expenses is obtained using the `Loader` API (see Snippet 6.18). The `onCreateLoader()` method is invoked as a result of the `initLoader()` method called at Line 15 of Snippet 6.17. The `sumOfExpenseUri` variable (used in Line 5 of Snippet 6.17) holds the value `content://com.mad.expensetracker.expenseprovider/SUM` (`CONTENT_URI2` in Snippet 6.14), through which the sum of expenses is queried. `updateTextView()` is a user-defined method that reads the returned data set and updates the TextView to display the sum of expenses.

```
1  @Override
2  public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
3    switch (arg0) {
4    case QUERY_CODE:
5    return new CursorLoader(MainActivity.this, sumofExpenseUri,
     null,null,null,null);
6    default:
7     return null;
8    }
9   }
10  @Override
11  public void onLoadFinished(Loader<Cursor> arg0, Cursor arg1) {
12    updateTextView(arg1);
13  }
14  private void updateTextView(Cursor cursor) {
15    while (cursor.moveToNext()) {
16     textView.setText(cursor.getInt(0) + "");
17    }
18  }
```

**Snippet 6.18** | Getting sum of expenses in the client app.

## 6.7 ENTERPRISE DATA

In the earlier sections of this chapter, we have explored how to deal with persisting and accessing native data – be it using files, shared preferences, device databases, or Content Providers. However the use cases that require device to exchange data over the network to communicate with enterprise systems need to handle data differently. For example, our 3CheersCable app has to retrieve TV guide, current shows, etc., over the network from the backend enterprise systems.

The enterprise systems expose specific functionalities, and in turn related underlying data, to serve the client apps. These functionalities are typically exposed using Web services; RESTful[2] Web services are popular for mobile clients. Sheer simplicity, light-weight approach, and support for simple CRUD operations have resulted in the popularity of RESTful Web services. The data between the mobile app and the enterprise app can be exchanged in several formats; JSON[3] (JavaScript Object Notation) is a popular format for exchanging small chunks of data in key–value pairs. Recall that the enterprise data access layer of the 3CheersCable mobile app sources the required remote data that is exposed in the service layer of the 3CheersCable enterprise app, as depicted in Fig. 3.1 of Chapter 3.

Let us now re-create the ExpenseTracker app in which the expense data is going to reside on an enterprise app, instead of being persisted on the device, as in the earlier scenario. The enterprise app will expose CRUD operations on expense data over a RESTful Web service. The expense data will be exchanged in JSON format over the network between both the parties.

Accessing data over the network requires an app to request `android.permission.INTERNET` permission. The app also needs to request `android.permission.ACCESS_NETWORK_STATE` permission to check network connectivity by accessing network state of the device.

Snippet 6.19 enlists the mechanism to check the network connectivity using a user-defined method `checkNetworkAccess()`. A `ConnectivityManager` instance is obtained by requesting the `CONNECTIVITY_SERVICE` (Line 2). Its `getActiveNetworkInfo()` method (Line 3) provides network information, using which network connectivity can be determined (Line 4).

```
1  private boolean checkNetworkAccess() {
2   ConnectivityManager connectivityManager = (ConnectivityManager)
    getSystemService(CONNECTIVITY_SERVICE);
3   NetworkInfo info = connectivityManager.getActiveNetworkInfo();
4   if (info != null && info.isConnected()) {
5     return true;
6   } else {
7     Toast.makeText(MainActivity.this, "No network access, network
      resource not accessible", Toast.LENGTH_SHORT).show();
8     return false;
9   }
10 }
```

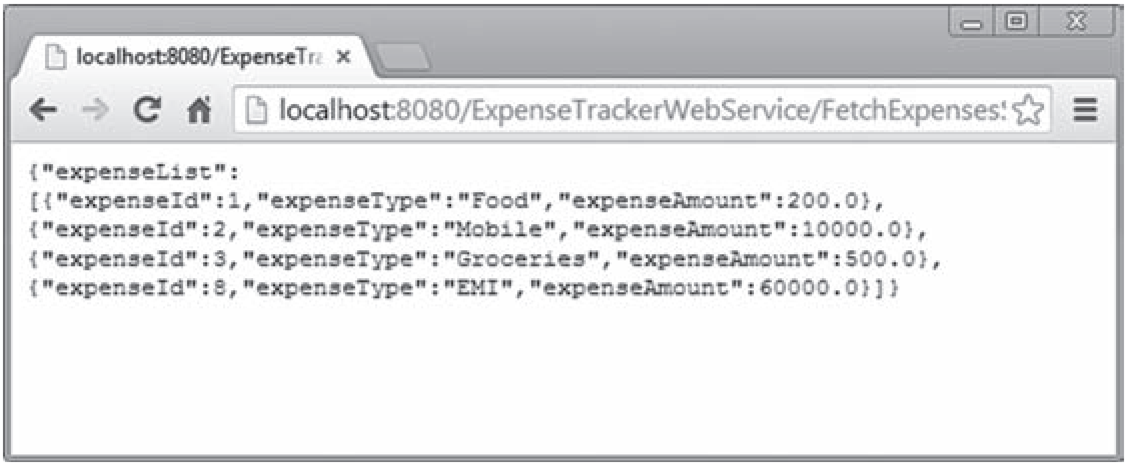
**Snippet 6.19** | Determining network connectivity.

Once the network connectivity is determined, the app needs to initiate an HTTP (Hypertext Transfer Protocol) request to exchange data with RESTful Web service. Android recommends an `HttpURLConnection` API to initiate HTTP requests. The `HttpURLConnection` API facilitates CRUD operations using `PUT`, `GET`, `POST`, and `DELETE`, HTTP methods.

In the ExpenseTracker app, the expense data is fetched using the `GET` HTTP method from a RESTful Web service. The Web service exposes the expense data at http://localhost:8080/ExpenseTrackerWebService/FetchExpensesServlet, and the data is exchanged using JSON format. The Web service response can be quickly validated by hitting the given URL, as shown in Fig. 6.11. The given URL may be different, based on the server configuration while hosting the Web service.

[2] Web Services Architecture: http://www.w3.org/TR/ws-arch/

[3] Introducing JSON: http://www.json.org/



```
{"expenseList":
[{"expenseId":1,"expenseType":"Food","expenseAmount":200.0},
{"expenseId":2,"expenseType":"Mobile","expenseAmount":10000.0},
{"expenseId":3,"expenseType":"Groceries","expenseAmount":500.0},
{"expenseId":8,"expenseType":"EMI","expenseAmount":60000.0}]}
```

**Figure 6.11** | The Web service response.

Snippet 6.20 demonstrates fetching expense data using the `GET` HTTP method from the RESTful Web service. An `HttpURLConnection` instance is created using the RESTful Web service URL (Lines 1–4), and its connection parameters are configured (Lines 5–7). To refer to the localhost from an Android emulator, IP address 10.0.2.2 is used (Line 3). The `setRequestMethod()` method sets the HTTP request method to `GET` (Line 7). The `setReadTimeout()` method is used to set the maximum time that a client can take to read the response from the Web service, and is set to 2s in this case (Line 5). The `setConnectTimeout()` method is used to set the maximum time within which a client has to establish the connection, and is set to 4s (Line 6). The `connect()` method is used to establish the connection, and make the HTTP request to fetch expense data (Line 8).

```
1  HttpURLConnection connection = null;
2  try {
3  URL url = new URL
    ("http://10.0.2.2:8080/ExpenseTrackerWebService/FetchExpensesSer
       vlet");
4  connection = (HttpURLConnection) url.openConnection();
5  connection.setReadTimeout(2000);
6  connection.setConnectTimeout(4000);
7  connection.setRequestMethod("GET");
8  connection.connect();
9  int responseCode = connection.getResponseCode();
10 if (responseCode == 200) {
11   InputStream inputStream = connection.getInputStream();
12   BufferedReader bufferedReader = new BufferedReader(
     new InputStreamReader(inputStream));
13   StringBuilder builder = new StringBuilder();
14   String line;
15   while ((line = bufferedReader.readLine()) != null) {
16     builder.append(line);
17   }
18   response = builder.toString();
19  }
```

```
20  else {
21   response = “Response was not successful”;
22  }
```

**Snippet 6.20** | Fetching expense data using HTTP `GET`.

Once the connection is established, and response is fetched successfully (HTTP status code: 200), the response is retrieved using the `getInputStream()` method (Line 11). This input stream is read through, and converted into a String object (Lines 12–19). This String object will contain the value in JSON format, as depicted earlier in Fig. 6.11. Now the app has to parse this response to fetch the individual expense data items, and display them in the ListView in the Activity, as depicted earlier in Fig. 6.8. It is advisable to use an AsyncTask to connect to the Web service, and parse the response, as these operations might take considerable amount of time depending on the network connectivity and the size of response obtained. Therefore, Snippet 6.20 should be implemented inside the `doInBackground()` method of the `AsyncTask`.

Now, the response is retrieved in the `onPostExecute()` method of the `AsyncTask` for parsing (see Snippet 6.21). In Line 5, the `readJsonStream()` method of a user-defined class `ExpenseParser` is used to obtain the list of individual expense data items. These expense data items are populated into the ListView, using `populateViews()`, a user-defined method in the Activity.

```
1  protected void onPostExecute(String result) {
2  super.onPostExecute(result);
3  List<Expense> expenses = new ArrayList<Expense>();
4  if(operationCode.equalsIgnoreCase(“1”)) {
5    expenses = new ExpenseParser().readJsonStream(result);
6    populateViews(expenses);
7   }
8  . . . //if statements for other operations(INSERT,UPDATE,
           DELETE)
9  }
```

**Snippet 6.21** | Parsing result in `onPostExecute()` method.

The `ExpenseParser` class is designed to take care of parsing the expense data items received in JSON format from the Web service. A typical “expense data items” response, constructed in JSON format, is shown below. It comprises an array of objects, each representing a row of individual expense in the underlying expenses table. The “`expenseList`” is the key to the entire array of objects. Each object in this array, in turn, comprises “`expenseId`,” “`expenseType`,” and “`expenseAmount`” keys along with respective values.

```
{“expenseList”:
[
 {“expenseId”:6,“expenseType”:“Rent”,“expenseAmount”:5000.0},
 {“expenseId”:7,“expenseType”:“Groceries”,“expenseAmount”:2000.0},
     . . . . .
]}
```

The `JsonReader` API is used to read JSON-encoded values (key–value pairs) in `ExpensePasser` class. While parsing the response, the `beginObject()`, `beginArray()`, `hasNext()`, and `nextName()` methods of the `JsonReader` class come handy. The `beginObject()` method is used to start traversing the entire JSON object, which can be stopped by using the `endObject()` method. The `beginArray()` method is used to start traversing the array of objects within the JSON object (identified with “`expenseList`” key in this case). The traversal can be stopped using the `endArray()` method. The `hasNext()` method is to identify if the end of response is reached. The `nextName()` method returns the key inside an object.

The `readJsonStream()` method of the `ExpenseParser` class accepts the "expense data items" response as parameter. It uses the `JsonReader` API to traverse the entire response using the `beginObject()` method, as shown in Line 7 of Snippet 6.22. The array of objects within the JSON response is passed to `readExpensesArray()` (Line 13), a user-defined method in the `ExpenseParser` class that returns the list of expenses.

```
1  public List<Expense> readJsonStream(String jsonExpense){
2  List<Expense> expenses=new ArrayList<Expense>();
3  byte[] stringBytes = jsonExpense.getBytes();
4  InputStream reader = new ByteArrayInputStream(stringBytes);
5  JsonReader jsonReader = new JsonReader(new
   InputStreamReader(reader));
6  try {
7    jsonReader.beginObject();
8    if(jsonReader.hasNext())
9    {
10     if(jsonReader.nextName().equals("expenseList"))
11     {
12      jsonReader.beginArray();
13      expenses=readExpensesArray(jsonReader);
14      jsonReader.endArray();
15     }
16   }
17   jsonReader.endObject();
18   jsonReader.close();
19  }catch (IOException e) {    e.printStackTrace();}
20   return expenses;}
```

**Snippet 6.22** | `readJsonStream()` method.

The `readExpensesArray()` also uses the `JsonReader` API, as shown in Snippet 6.23, to parse the individual JSON objects in the array of objects retrieved from JSON response by calling the `parse-Expense()` method (Line 6). The parsed JSON objects will eventually get added to an `Expense` list (Line 6) that gets returned from this method.

```
1  private List<Expense> readExpensesArray(JsonReader jsonReader)
   {
2   List<Expense> expenses=new ArrayList<Expense>();
3   try {
4     while(jsonReader.hasNext())
5     {
6       expenses.add(parseExpense(jsonReader));
7     }}
8    catch (IOException e) {
9     e.printStackTrace();
10   }
11   return expenses;
12  }
```

**Snippet 6.23** | `readExpensesArray()` method.

The `parseExpense()` method is a user-defined method in the `ExpenseParser` class that actually parses the individual JSON objects (the String representing the individual expense), as shown in Snippet 6.24. It traverses through each key–value (key is also referred to as token) pair, and assigns the corresponding values into an `Expense` bean object (Lines 4–15), using `nextInt()`, `nextString()`, `nextFloat()`, or `nextDouble()` methods appropriately. The `skipValue()` method of `JsonReader` is used to skip any inappropriate token, if encountered while parsing the individual JSON objects (Line 13).

```
1  private Expense parseExpense(JsonReader jsonReader) throws
   IOException {
2   Expense expense = new Expense();
3   jsonReader.beginObject();
4   while (jsonReader.hasNext()) {
5      String token = jsonReader.nextName();
6      if (token.equals(“expenseId”)) {
7       expense.setExpenseId(jsonReader.nextInt());
8      } else if (token.equals(“expenseType”)) {
9       expense.setExpenseType(jsonReader.nextString());
10     } else if (token.equals(“expenseAmount”)) {
11      expense.setExpenseAmount((float)jsonReader.nextDouble());
12     } else {
13      jsonReader.skipValue();
14     }
15  }
16  jsonReader.endObject();
17  return expense;}
```

**Snippet 6.24** | `parseExpense()` method.

The `parseExpense()` method returns an `Expense` instance after parsing individual JSON objects in the array of objects. The `Expense` bean class is enlisted in Snippet 6.25.

```
public class Expense {
 private int expenseId;
 private String expenseType;
 private float expenseAmount;
 public int getExpenseId() {
 return expenseId;
 }
//setter and getter methods
...
}
```

**Snippet 6.25** | `Expense` bean class.

Once the `readJsonStream()` method finishes parsing the JSON response obtained from the Web service, the parsed list of expenses are populated onto the ListView of the Activity, using `populateViews()`, a user-defined method in the Activity class.

The Web service also exposes the functionality to add an expense at http://10.0.2.2:8080/ExpenseTrackerWebService/InsertExpenseServlet URL. The HTTP `PUT` method is used to perform this operation, as shown in Line 5 of Snippet 6.26. The expense data to be inserted is sent as part of the HTTP `PUT` request using `DataOutputStream` of the `HttpURLConnection` object (Lines 12–14).

The length and data type of the data being sent is specified using the `setRequestProperty()` method of the `HttpURLConnection` (Lines 6 and 8, respectively). The response obtained after inserting data must be displayed in the ListView after parsing, as discussed earlier.

```
1  HttpURLConnection connection = null;
2  try {
3  URL url = new URL(
   “http://10.0.2.2:8080/ExpenseTrackerWebService/InsertExpenseSer
   vlet”);
4  connection = (HttpURLConnection) url.openConnection();
5  connection.setRequestMethod(“PUT”);
6  connection.setRequestProperty(“charset”, “utf-8”);
7  String urlParams = “expenseType=” + expenseType+
   “&expenseAmount=” +  expenseAmount;
8  connection.setRequestProperty(“Content-Length”, “”+
   Integer.toString(urlParams.getBytes().length));
9  connection.setUseCaches(false);
10 connection.setReadTimeout(2000);
11 connection.setConnectTimeout(4000);
12 DataOutputStream outputStream = new
   DataOutputStream (connection.getOutputStream());
13 outputStream.writeBytes(urlParams);
14 outputStream.flush();
15 connection.connect();
16 int responseCode = connection.getResponseCode();
17 if (responseCode == 200) {
18  // Proceed to parse response and update UI after successful
    insertion
19 }
20 else {
21  response = “Response was not successful”;
22 }
```

**Snippet 6.26** | Inserting expense data using HTTP `PUT`.

The Web service also exposes the functionality to update and delete an expense at http://10.0.2.2:8080/ExpenseTrackerWebService/UpdateExpenseServlet and http://10.0.2.2:8080/ExpenseTrackerWebService/DeleteExpenseServlet URLs, respectively. The HTTP `POST` and `DELETE` methods are used to perform these operations, as shown in Snippet 6.27 and 6.28, respectively. Performing update operation requires the same request parameter as that of insert operation, as seen earlier. In case of delete operation, the data requested to be deleted (expense id) is sent as part of the HTTP request header. This is accomplished using the `addRequestProperty()` method of `HttpURLConnection`. There is no need to construct the data as URL parameters, and write it to connection stream using the `DataOutputStream`, as we have done earlier in the case of insert and update operations.

```
1  URL url = new URL(
   “http://10.0.2.2:8080/ExpenseTrackerWebService/UpdateExpenseServ
   let”);
2  connection.setRequestMethod(“POST”);
```

```
. . . .
. . . .
String urlParams = “expenseId=”+expenseId+”&expenseType=” +
expenseType+ “&expenseAmount=” + expenseAmount;
```

**Snippet 6.27** | Updating expense data using HTTP `POST`.

```
URL url = new URL(
“http://10.0.2.2:8080/ExpenseTrackerWebService/DeleteExpenseServ
let”);
connection.setRequestMethod(“DELETE”);
...
connection.addRequestProperty(“expenseId”, expenseId);
```

**Snippet 6.28** | Deleting expense data using HTTP `DELETE`.

We have just now explored exchanging JSON data over RESTful Web services. There may be scenarios wherein the data format as well as the data transport mode may be different (from JSON and RESTful), and need to be handled differently. To deal with such scenarios, Android supports both in-house and third-party libraries to exchange data with backend enterprise systems, on a case-to-case basis.

## 6.8 LET’S APPLY

The 3CheersCable mobile app exchanges data with the enterprise app in several scenarios such as verifying user credentials during login, fetching list of channel categories, fetching list of channels belonging to a specific category, fetching list of shows in a specific channel, and storing user’s favorite shows.

So far, we have obtained these responses from the server in an AsyncTask. In this section, we shall now further parse the response that is obtained while fetching the list of channel categories in the `SubscribeActivity` (refer Section 5.4 of Chapter 5). The format of response returned by the Web service containing the channel categories is shown below.

```
{“result”: <<Number of categories returned>>,
 “categories”:[
   {“categoryId”:“10001”, “name”:“Movies”, “serviceProvider”:“Three
     Cheers Cable”, “imgurl”:“/icons/blazer”, “genre”:“English”,
     “quality”:“Entertainment”},
   {“categoryId”:“10002”, “name”:“Music”, “serviceProvider”:“Three
     Cheers Cable”, “imgurl”:“/icons/mavrik”, “genre”:“English”,
     “quality”:“Entertainment”},
   ...
]}
```

This response shall be parsed in a user-defined class, `CategoriesResponseParser`, using `JsonReader` API.

The `onNetworkCallComplete()` of `SubscribeActivity` is invoked when the response is successfully fetched from the server. In this method, the `parse()` method of `CategoriesResponse-Parser` is invoked to parse the obtained response. The `CategoriesResponseParser` is defined as follows:

```
public class CategoriesResponseParser {

```

```
public static CategoriesBean parse(String resultString) throws
NullPointerException {

  byte[] stringByte = resultString.getBytes();
  InputStream stream = new ByteArrayInputStream(stringByte);
  JsonReader jsonReader = new JsonReader(new
  InputStreamReader(stream));
  CategoriesBean categories = new CategoriesBean();
  try{
   jsonReader.beginObject();
   if(jsonReader.hasNext()){
     String token = jsonReader.nextName();
     if(token.equals(“result”))
       categories.setResult(jsonReader.nextInt());
       token = jsonReader.nextName();
       if(token.equals(“categories”)){
           jsonReader.beginArray();
       categories.setCategories(readCategoryArray(jsonReader));
          jsonReader.endArray();
       }
     }
     jsonReader.endObject();
     jsonReader.close();
   }
   catch (Exception e) {
     e.printStackTrace();
   }
   return categories;
  }

  public static ArrayList<Category> readCategoryArray(JsonReader
  jsonReader) {
      ArrayList<Category> categoryList = new
     ArrayList<Category>();
     try{
       while(jsonReader.hasNext()){
        categoryList.add(parseCategory(jsonReader));
       }

     }
     catch (Exception e) {
       e.printStackTrace();
     }
     return categoryList;
 }

  public static Category parseCategory(JsonReader jsonReader)
  throws IOException {
      Category category = new Category();
      jsonReader.beginObject();
      while(jsonReader.hasNext()){
        String token = jsonReader.nextName();
        if(token.equals("categoryId"))
          category.setCategoryId(jsonReader.nextString());
        else if(token.equals("name"))
         category.setName(jsonReader.nextString());
        else if(token.equals("serviceProvider"))
         category.setServiceProvider(jsonReader.nextString());
        else if(token.equals("imgurl"))
         category.setImgurl(jsonReader.nextString());
        else if(token.equals("language"))
         category.setLanguage(jsonReader.nextString());
        else if(token.equals("genre"))
         category.setGenre(jsonReader.nextString());
        else if(token.equals("quality"))
         category.setQuality(jsonReader.nextString());
        else{
          jsonReader.skipValue();
       }
       jsonReader.endObject();
         return category;
  }
 }
```

The `parse()` method of `CategoriesResponseParser` reads through the entire response obtained and invokes the `readCategoryArray()` method to parse the array of categories obtained in the response. The `readCategoryArray()` method invokes the `parseCategory()` method, which returns a `CategoriesBean` object after parsing the individual categories obtained in the response. The `CategoriesBean` class is defined as follows:

```
public class CategoriesBean {
  int result;
  ArrayList<Category> categories;
  ...
  //setter and getter methods for result and categories
  public static class Category {
   private String categoryId, name, serviceProvider, imgurl,
   language, genre, quality;
   ...
   //setter and getter methods for categoryId, name,
     serviceProvider, imgurl, language, genre, and quality
  }
 }
```

Similarly, for other functionalities such as login, fetching list of channels belonging to a specific category, fetching list of shows being played within a specific channel, and storing shows selected as "Favorites" within the database to be retrieved later, we have to create corresponding classes (not created as part of this exercise).

## SUMMARY

This chapter has introduced various data persistence and access mechanisms in a mobile app to cater to app data requirements. It has discussed the flat file approach to deal with unstructured data in a mobile app, and the shared preferences approach to deal with primitive data in a key–value format.

The chapter has defined the construction of relational databases on devices using SQLite. It has also discussed the data manipulation operations while dealing with relational data; SQLiteDatabase and SQLiteOpenHelper APIs have been shown to achieve this.

The chapter has explored in detail one of the key building blocks of the Android platform – the Content Provider. It has also demonstrated sharing app data with other apps using Content Providers. Toward the end, it has delved into dealing with exchange of data between mobile apps and backend enterprise systems. The scenarios are explained using RESTful Web services as the transport mode, and JSON as the data format.

## REVIEW QUESTIONS

1. Enlist various mechanisms of data persistence and access in mobile apps. Highlight the pros and cons of each mechanism.
2. Illustrate data persistence and access using shared preference.
3. Explain the steps to create a SQLite database in an app.
4. Name some commonly used in-built Content Providers.
5. Explain the make-up of CONTENT_URI.
6. Define RESTful Web services, highlighting their wide adoption while dealing with mobile apps.
7. Explain the various methods of JsonReader API used for parsing JSON data.

## FURTHER READINGS

1. Storage options in Android: http://developer.android.com/guide/topics/data/data-storage.html
2. SharedPreferences: http://developer.android.com/reference/android/content/SharedPreferences.html
3. Preference API: http://developer.android.com/guide/topics/ui/settings.html
4. Content Providers: http:// developer.android.com/guide/topics/providers/content-providers.html
5. Web Services Architecture: http://www.w3.org/TR/ws-arch/
6. Introducing JSON: http://www.json.org/
7. JsonReader API: http://developer.android.com/reference/android/util/JsonReader.html

# Part III
# Sprucing Up

7

# Graphics and Animation

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Design apps that adapt to multiple screen densities and sizes.
- Discover two-dimensional (2D) graphics types and capabilities.
- Illustrate animation types and capabilities.
- Apply graphics and animation capabilities to an app.

## 7.1 INTRODUCTION

The most popular mobile apps undoubtedly have one thing in common – awesome user experience – both on creative and functional fronts. Graphics and animation are two key ingredients that bring this to fruition by making an app lively and keeping the user engaged.

On one hand, from an end user's perspective, graphics enhances the visual quality of an app that results in the *wow* effect. On the other hand, from a developer's standpoint, graphics is all about understanding and dealing with the screen size, screen resolution, screen orientation, colors, typography, and image formats to create a simple, clean, and unique user interface (UI) with sound aesthetics.

Animations, from an end-user's perspective, further add zing to the app experience by augmenting tiny delights at multiple occasions. From a developer's standpoint, animation is all about keeping the user engaged while something is being done in the background, for example, buffering a song; or when some state change occurs, such as, when a new tweet arrives; or when there is a transition from one part of an app to another; and many more.

This chapter introduces powerful UI capabilities for an app developer using graphics and animation APIs of Android. It also explores these capabilities and the best practices to make an app lively and interactive.

## 7.2 ANDROID GRAPHICS

The key graphics capabilities of an Android platform can be broadly categorized into the following:

1. **Drawables and Canvas**: Drawables are 2D graphics components of Android. Android also allows building custom 2D graphics using Canvas.
2. **Hardware acceleration**: Hardware acceleration means using device's Graphical Processing Unit (GPU) to render an app's UI. This can mean a major performance boost for graphics-intensive apps, especially gaming.
3. **OpenGL**: OpenGL ES is an open source 3D framework that is used avidly by mobile platforms, including Android, to render 3D graphics.

Hardware acceleration and OpenGL are beyond the scope of this book. We will explore Drawables and Canvas in the following sections.

### 7.2.1 Supporting Multiple Screens

Before jumping into the nitty-gritty of Android graphics capabilities, let us understand two key concepts that predominantly impact the app UI – screen density and screen size. As the spectrum of devices sporting Android varies a lot in terms of screen densities and screen sizes, it is of paramount importance for a developer to make sure that his or her app UI adapts to these different devices.

Screen density is defined as the number of pixels per unit area of a screen, usually referred to as dots per inch (dpi). As depicted in Fig. 7.1, the leftmost screen has the lowest density because it has the least number of pixels in the same area versus other two screens that are arranged in increasing order of density.

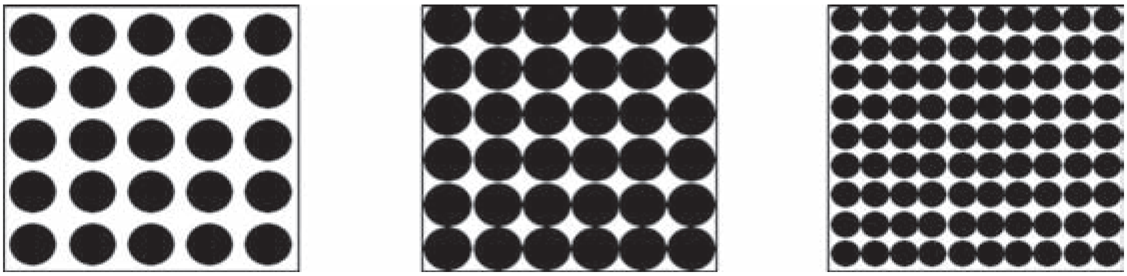


**Figure 7.1** | Screens with different densities.

Let us now see how different screen densities affect app UI. As depicted in Fig. 7.2, we have a simple app running on three different density screens. You can now easily make out how the app is being rendered on devices with different screen densities (increasing from left to right).

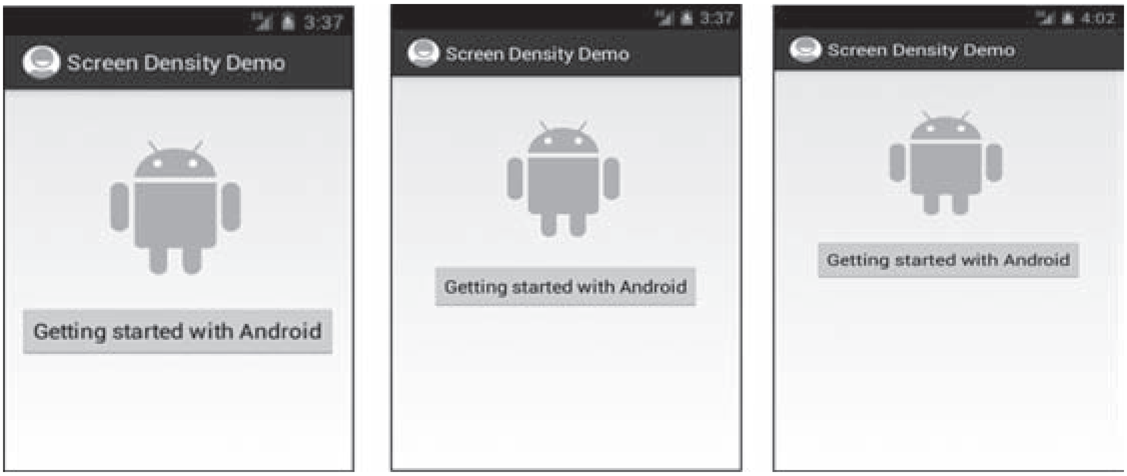

3:37
Screen Density Demo
Getting started with Android
3:37
Screen Density Demo
Getting started with Android
4:02
Screen Density Demo
Getting started with Android

**Figure 7.2** | Same app running on different density screens.

In the example given in Fig. 7.2, there is an image and a button, and they get rendered in different sizes based on the screen density. But the aim is to achieve same image size across different density screens so that the app UI looks homogeneous across devices (of the same screen size). To realize this, a developer needs to handle images and views appropriately so that the app UI can adapt to multiple target devices.

To deal with images, Android has an in-built mechanism to automatically pick the appropriate image for a given screen density. To achieve this, a developer needs to prepare multiple versions (resolution) of the original image and place them in the res\drawable (-xxx) folders. The image name should be same in all the folders. Android broadly categorizes the screen density into four categories – ldpi (low), mdpi (medium), hdpi (high), and xhdpi (very high), as depicted in Fig. 7.3 – and provides a drawable folder for each category.

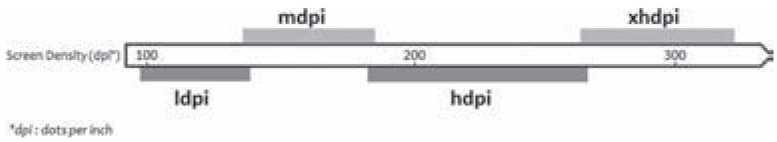


**Figure 7.3** | Screen density categorization.

Low-resolution images go in the ldpi folder, and further higher resolution images are placed in increasing order of the screen density – mdpi, hdpi, and xhdpi. In case there is only one image, it can be placed in any one of the drawable folders.

To deal with views across multiple screen densities, it is recommended to use density-independent pixels (dp) as the dimension unit instead of using pixels (px). Using dp, Android ensures that views get appropriately scaled up/down based on the screen densities. For example, a button of 50 dp might take 70 px on a high-density device and only 30 px on a low-density device. This is to ensure that the button size remains same across devices of different densities (of the same screen size).

Having understood how to make an app UI consistent across devices of same screen size, but different densities, let us now look at managing UI for devices with different screen sizes. This is crucial in the case of Android because Android devices vary immensely when it comes to sizes, from a tiny 3-inch watch to a gigantic 50-inch television. Android provides three configuration qualifiers – sw<N>dp, w<N>dp, and h<N>dp – for supporting different screen sizes. Here, <N>dp represents screen dimension (either width or height), measured in dp. Recall that the screen dimension is being measured in dp because it is independent of the screen density, and so devices with same screen size have same dimensions in dp, even if their densities are not equal.

The sw<N>dp qualifier is used for a device with smallest dimension (either height or width) of <N>dp. This qualifier is independent of the orientation of the device. The w<N>dp and h<N>dp qualifiers identify a device on the basis of the current width and height, respectively. These two qualifiers are dependent on the two orientation of the device, and with a change in the orientation, the applicability of the qualifier might vary. These qualifiers are used to provide alternate layout resources by creating multiple sub-folders in the res folder. For example, a layout resource in the res\layout-sw500dp\ folder caters to devices having smallest screen dimension of 500 dp. Similarly, a layout resource in the res\layout-w600dp-h300dp\ folder caters to devices having minimum current screen width of 600 dp and height 300 dp.

Android runtime automatically picks the appropriate layout resource based on the screen size and orientation of the device. You may recall that Android also provides Fragments to fold an app UI to deal with multiple screen sizes. In a small/normal screen, typically one Fragment is used in a layout, whereas for larger screens a layout can accommodate multiple Fragments.

### 7.2.2 Drawables

A drawable is a visual resource that is typically referred to as something that can be drawn such as a 2D image or a shape. In its simplest form, where dynamic graphics is not required, a drawable is typically drawn onto a view that can be later programmatically retrieved by using graphics APIs.

Android supports various types of drawables. Bitmap, preferably a png image, is one of the simplest drawables that finds its use in creating an app icon, or a background image, or in simple animations. We already had a glimpse of it as an image resource in Section 4.3.3 of Chapter 4. Shape drawable is another simple drawable that finds its use where geometric shapes are required to be drawn. In this case, an XML file is typically used, instead of an image file, to define the attributes of the geometric shape.

Besides the simple drawables mentioned above, several compound drawables such as layer drawable that is used when there is a requirement to manage an array of drawables to be drawn in a layered fashion (one over the other), or state list drawable that is used when there is a requirement to use different drawables based on view state (e.g., to use different images for a toggle button based on its state), are also supported.

Let us now take an example of how we can customize the look of an EditText using a shape drawable. Accomplishing this involves two steps. The first step is about preparing an XML describing the shape drawable, and in the next step we attach this drawable to the EditText.

Snippet 7.1 depicts an XML describing a rounded-rectangle shaped drawable. This XML file is saved as shape_draw.xml in the res\drawable folder.

```
<shape android:shape="rectangle">
  <gradient android:startColor=" #4A4A4A"
     android:endColor="#AAAAAA" android:angle="90"/>
  <padding android:left="7dp" android:top="7dp"
     android:right="7dp" android:bottom="7dp"/>
  <corners android:radius="8dp"/>
</shape>
```

**Snippet 7.1** | Shape drawable XML.

Snippet 7.2 depicts how we can apply this drawable as a background of an EditText. The `android:background` attribute of `<EditText>` in Line 3 is used to refer to the shape drawable (shape_draw).

```
1   <EditText
2        android:id="@+id/edit_text01"
3        android:background="@drawable/shape_draw"
4        android:layout_height="wrap_content"
5        android:layout_width="fill_parent"
6        android:text="Shape Drawable"
7      />
```

**Snippet 7.2** | EditText customized using a shape drawable.

Figure 7.4 depicts the resulting customized EditText rendered on a device.

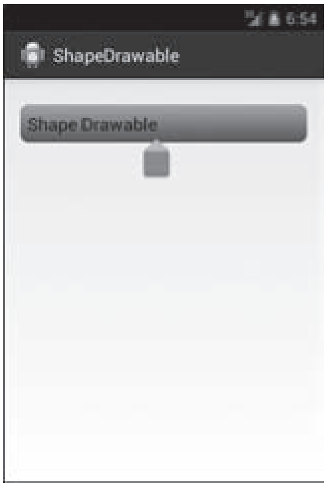


**Figure 7.4** | Customized EditText.

### 7.2.3 Custom View and Canvas

We have learnt in Section 7.2.2 that a drawable is typically drawn onto an existing view, and hence its usage is limited to modifying an existing view. What if we need to build a view from scratch, or modify the behavior of existing views? In such situations, we need to create our own custom views by extending either the `View` class or existing views such as `EditText` and `Button`.

Custom views are provided with `Canvas` object to draw its UI . A `Canvas` is like a conduit to the actual drawing surface. In order to draw anything on the screen, we need to call methods provided by the `Canvas`.

Snippet 7.3 lists the creation of a simple custom view that uses a `Canvas` to draw an image on the screen. The custom view `MyView` (Lines 8–30) extends the `View` class. Line 5 sets `MyView` as the content view of the `CanvasDemoActivity`. In the constructor of `MyView` (Lines 12–18), we have loaded the image to be drawn from the resources folder (res\drawable-xxx) and set up a `Paint` object. `Paint` is used to set drawing attributes such as color, text size, fill type, etc. The actual draw call is made in the overriden `onDraw()` method of the parent `View` class. This method provides `Canvas` as an argument. Line 21 lists the actual draw call being made to draw the image (my_image) on the screen. The image is drawn at the top left corner of the screen.

```
1  public class CanvasDemoActivity extends Activity {
2  @Override
3  protected void onCreate(Bundle savedInstanceState) {
4    super.onCreate(savedInstanceState);
5    setContentView(new MyView(this, null));
6    Toast.makeText(this, "Touch anywhere to change the location
       of image",Toast.LENGTH_LONG).show();
7  }
8  class MyView extends View {
9    Bitmap image;
10   Paint paint;
11   float left=0, top=0;
12   public MyView(Context context, AttributeSet attrs) {
13     super(context, attrs);
14     image = BitmapFactory.decodeResource(getResources(),
                R.drawable.my_image);
15     paint = new Paint();
16     paint.setDither(true);
17     paint.setAntiAlias(true);
18    }
19   @Override
20   protected void onDraw(Canvas canvas) {
21     canvas.drawBitmap(image, left, top, paint);
22   }
23   @Override
24   public boolean onTouchEvent(MotionEvent event) {
25     left = event.getX();
26      top = event.getY();
27      invalidate();
28      return true;
29    }
30  }
31  }
```

**Snippet 7.3** | A simple custom view.

This view is touch enabled; whenever a user touches the screen, `MyView` extracts the coordinates of the touch and redraws the image at these coordinates. This functionality is brought about by overriding the `onTouchEvent()` method of the `View` class (Lines 24–29). This method receives touch events as and when they occur. `MyView` maintains two variables, `top` and `left`, and keeps updating them to contain the coordinates of the last place where the user touched the screen. These variables determine the position to which the image is moved on the screen. Another key step involved in this is invalidating the view. As soon as the user touches the screen, we need to redraw the image at the touched location. This is accomplished by calling the `invalidate()` method (Line 27), which in turn causes a view refresh and subsequently calls the `onDraw()` method.

## 7.3 ANDROID ANIMATION

The Android platform provides a range of capabilities to cater from the simplest to the most sophisticated animation requirements of an app. Android animations may be broadly categorized into two types: frame-by-frame and transition.

Frame-by-frame animation, also referred to as drawable animation, is one of the simplest types of animation system that shows multiple frames in quick succession to create an animation effect such as in a film roll of a movie.

Transition animation is a more sophisticated animation system where varying any property of an object over time, in turn, delivers an animated effect to the user. Android has two types of animation systems to implement this: view animation and property animation. View animation is restricted only to animate Android views, and is relatively older. However, property animation is relatively new, more powerful, and can animate any property of any object. These objects may or may not be `View`s.

### 7.3.1 Drawable Animation

Drawable animation is about showing drawable resources (e.g., images) in quick succession one after the other to produce an animated effect. To create a drawable animation, we need to follow three simple steps.

It starts with a prework step that typically includes making/collecting the drawable resources (images), determining the sequence in which these drawables have to be played, and determining their individual duration.

The next step is to compose the animation, which involves preparing an XML that lists the drawable resources for each frame, their order and duration. The tag `<animation-list>` is the root node of this XML file, wherein each `<item>` node defines a frame and its duration, as depicted in Snippet 7.4.

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
 xmlns:android="http://schemas.android.com/apk/res/android">
 <item android:drawable="@drawable/img1"
  android:duration="250" />
 <item android:drawable="@drawable/img2"
  android:duration="250" />
 <item android:drawable="@drawable/img3"
  android:duration="250" />
 <item android:drawable="@drawable/img4"
  android:duration="250" />
</animation-list>
```

**Snippet 7.4** | Drawable animation XML.

This animation loads four images (img1–img4) one after the other in quick succession, each staying on the screen for 250 ms. This animation XML file is saved as drawable_anim.xml in the res\drawable folder.

The last step is realizing animation on the UI that involves associating the animation to a view and then playing it. As depicted in Snippet 7.5, the animation is set as the background resource of an `ImageView` (Line 12) and is played on a Button tap (Lines 16 and 17).

```
1  public class DrawableAnimationDemo extends Activity implements
   OnClickListener {
2   private Animation transAnim;
3   private ImageView image;
4
5   @Override
6   public void onCreate(Bundle savedInstanceState) {
7      super.onCreate(savedInstanceState);
8      setContentView(R.layout.draw_anim);
9      Button button = (Button) findViewById(R.id.button1);
10     button.setOnClickListener(this);
11     image = (ImageView) findViewById(R.id.ImageView1);
12     image.setBackgroundResource(R.drawable.drawable_anim);
13  }
14
15  public void onClick(View v) {
16     AnimationDrawable frameAnimation = (AnimationDrawable)
       image.getBackground();
17     frameAnimation.start();
18  }
19 }
```

**Snippet 7.5** | Applying drawable animation to a view.

The drawable animation defined in Snippet 7.4 is retrieved as `AnimationDrawable` (Line 16, Snippet 7.5). In order to play the animation, a call to the `start()` method of `AnimationDrawable` is made (Line 17, Snippet 7.5).

### 7.3.2 View Animation

A view animation is used to perform tweened animation on a view. A tweened animation helps achieving simple transformations of a view, based on four aspects – position, rotation, size, and transparency.

It starts with a prework step that includes identifying the view to be animated, and the number, order, and duration of transformations to be carried out on that view. Let us apply an animation on an image that lasts for 500 ms, wherein the image shrinks and loses opacity simultaneously. This means that we need to apply two transformations in parallel on an ImageView, namely, size and transparency.

Once we have completed the prework, the next step is to compose the animation XML that has the animation instructions to define the type of transformation, timing, and duration. On the basis of the type of transformation required in the animation, we can use one of the following tags as the root tag: `<alpha>`, `<rotate>`, `<scale>`, or `<translate>` for transparency, rotation, size, and position, respectively. We can also use `<set>` as the root tag where we need to group the transformation types so that they can be applied together.

Snippet 7.6 lists the implementation of the example animation described above. As we need to perform two transformations, we have used a `<set>` tag with `<scale>` and `<alpha>` as its child nodes. Size transformation (Lines 6–10) shrinks the image from 100% to 10% about its center. And the transparency transformation changes its opacity from completely opaque to completely transparent.

```
1  <?xml version=“1.0” encoding=“utf-8”?>
2  <set
    xmlns:android=“http://schemas.android.com/apk/res/android”
3   android:duration=“500”
4   android:fillAfter=“true”>
5
6   <scale
7    android:fromXScale=“1”  android:toXScale=“0.1”
8    android:fromYScale=“1”  android:toYScale=“0.1”
9    android:pivotX=“50%”    android:pivotY=“50%”
10   />
11
12   <alpha
13    android:fromAlpha=“1”
14    android:toAlpha=“0”/>
15
16  </set>
```

**Snippet 7.6** | View animation XML.

Because, both size and transparency animations need to run for the same duration of 500 ms, the `android:duration` has been applied as an attribute of `<set>` tag (Line 3). The `android:fillAfter` attribute (Line 4), when set to `true`, makes a view retain the state it is in, at the end of an animation. For example, if it has been made transparent toward the completion of animation, it will remain transparent after the animation completes. If `android:fillAfter` attribute is not specified, the view returns back to its original state once the animation is over.

View animation XML resources are stored in the res\anim folder. Assuming that we have stored the XML listed in Snippet 7.6 as hide_anim.xml, let us see how we can apply this animation onto a view, as the last step.

```
1  public class ViewAnimationDemo extends Activity implements
   OnClickListener {
2
3    private Button hideButton;
4    private ImageView greenBotImageView;
5
6    @Override
7    protected void onCreate(Bundle savedInstanceState) {
8      super.onCreate(savedInstanceState);
9      setContentView(R.layout.activity_view_animation_demo);
10     hideButton = (Button) findViewById(R.id.button1);
11     hideButton.setOnClickListener(this);
12     greenBotImageView = (ImageView)
       findViewById(R.id.ImageView1);
13   }
14
15   @Override
16   public void onClick(View v) {
17      Animation hideAnim =AnimationUtils.loadAnimation(this,
        R.anim.hide_anim);
18      greenBotImageView.startAnimation(hideAnim);
19   }
20
21  }
```

**Snippet 7.7** | Applying view animation to an image.

Snippet 7.7 lists a setup similar to the one we have seen in the case of drawable animation (Snippet 7.5). We have an ImageView and a Button as part of the Activity layout, and on tap of the Button we need to apply the animation (Snippet 7.6). This is being done in the `onClick()` method (Lines 16–19) in two steps. The first step is to load the animation from resources by using the `loadAnimation()` static method (Line 17) of the `AnimationUtils` class, which is a utility class for view animations. Once we have the animation, the next step is to apply this onto a view and start it. This is done by calling the `startAnimation()` method available to all views and passing the `Animation` instance as an argument (Line 18).

### 7.3.3 Property Animation

Property animation is a sophisticated animation API available in Android that allows animating any property of an object (not necessarily a view). While view animation is limited to performing only four predefined transformations – position, rotation, size, and transparency – there is no such constraint with property animation. For example, animating the background color of a layout is not possible using view animation, but can be accomplished using property animation.

Snippet 7.8 lists a simple Activity with a Button, on click of which the background color of the Activity layout is animated from red to blue. This is accomplished by using the `ObjectAnimator` class (Line 12). An `ObjectAnimator` can animate any property of an object. Depending on the type of property being animated, the `ObjectAnimator` class provides different factory methods such as `ofInt()`, `ofFloat()`, `ofObject()`, etc. In this example, we are animating over colors, which are stored as integers in Android, hence we are using the `ofInt()` factory method. Line 13 sets the duration of the animation to 1s.

```
1  public class PropertyAnimationDemo extends Activity implements
   OnClickListener {
2
3    private RelativeLayout parent;
4    private ObjectAnimator colorAnimator;
5
6    @Override
7    protected void onCreate(Bundle savedInstanceState) {
8      super.onCreate(savedInstanceState);
9      setContentView(R.layout.activity_property_animation_demo);
10     findViewById(R.id.button1).setOnClickListener(this);
11     parent = (RelativeLayout)findViewById(R.id.parent);
12     colorAnimator = ObjectAnimator.ofInt(parent,
                       “backgroundColor”,Color.RED, Color.BLUE);
13     colorAnimator.setDuration(1000);
14   }
15
16   @Override
17   public void onClick(View v) {
18     colorAnimator.start();
19   }
20  }
```

**Snippet 7.8**|Animating background color using property animation

The rest of the code is fairly straightforward. We have set up an `OnClickListener` on the Button. On tap of the Button, we make a call to the `start()` method of the `ObjectAnimator` that kick starts the color animation (Line 18).

The `ObjectAnimator` works only for those scenarios wherein the object property that is being animated has its setter/getter methods available. For example, in Snippet 7.8, we are able to animate the `backgroundColor` property of the `RelativeLayout` object using the `ObjectAnimator`, because `RelativeLayout` provides the `setBackgroundColor()` and `getBackground-Color()` methods.

For scenarios where setter/getter methods are not available, the property animation API provides a generic animator called `ValueAnimator`. To cater to scenarios where we need to animate multiple properties of a view, the property animation API provides a specific animator called `ViewAnimator`.

Behind the scenes, in all the property animators mentioned above, two important concepts play key roles – time interpolator and type evaluator. Time interpolator modulates the rate of animation. For example, a linear interpolator keeps the rate of animation constant, whereas accelerate interpolator accelerates it gradually. Android provides multiple types of interpolators such as `AccelerateInterpolator`, `DecelerateInterpolator`, `AccelerateDecelerateInterpolator`, etc. All animators provide the `setInterpolator()` method to set an interpolator of an animation. By default, the `ObjectAnimator` uses `AccelerateDecelerateInterpolator` as its interpolator. We can even create our own custom interpolator in case none of the existing interpolators suit our needs. The other key concept worth mentioning is type evaluator. Type evaluator is the component that determines the value of the animated property at any instance of time during the animation. Android provides some pre-existing type evaluators that can determine animated property values as long as the property is of type integer, float, or color. In case the property is not of the types mentioned, then we need to implement our own type evaluator.

The interpolator being used by the `ObjectAnimator` in Snippet 7.8 is `AccelerateDecelerateInterpolator`. To figure out the type evaluator being used, we can look for the factory method being used while creating the animator. In this case, we are using the `ofInt()` factory method (Line 12), so the type evaluator being used behind the scenes is `IntEvaluator`.

We have just finished exploring two types of transition animations: view and property. Both of them have their strengths and weaknesses. View animations are simple but confined only to view elements, whereas property animations have a broader reach, but might end up being cumbersome in case of complex animations. Another drawback of the view animation is that the animation occurs only at the UI level, but the actual position of the views remains unchanged. For example, if we apply view animation to change the position of a button, even though the button appears to have moved, the tap functionality does not correspond to the new position. On the basis of these details, developers decide between view animation or property animation, based on the app requirements.

## 7.4 LET'S APPLY

In this section, let us include an animation in the 3CheersCable app. We shall create a *swipe* animation such that when the user navigates from the `MainActivity` to the `SubscribeActivity` and vice versa, a *swipe* transition occurs between the screens.

To create this animation, let us create four XML files – l_to_r_enter_anim.xml, l_to_r_exit_anim.xml, r_to_l_enter_anim.xml, and r_to_l_exit_anim.xml – in the res\anim folder.

To bring in the *swipe* transition while navigating to the called Activity (`SubscribeActivity`), first we need a translate transformation that brings in the new Activity from extreme right of the screen (`fromXDelta="100%"`) to the center of the screen (`toXDelta="0%"`), which is implemented in the r_to_l_enter_anim.xml, as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<translate
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromXDelta="100%"
  android:toXDelta="0%"
  android:interpolator="@android:anim/accelerate_decelerate_inte
  rpolator"
  android:duration="500">
</translate>
```

Next, we need a translate transformation that takes out the calling Activity (`MainActivity`) from the center of the screen (`fromXDelta="0%"`) to the extreme left of the screen (`toXDelta="-100%"`), which is implemented in the r_to_l_exit_anim.xml, as shown below.

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

```
 android:fromXDelta="0%"
 android:toXDelta="-100%"
  android:interpolator="@android:anim/accelerate_decelerate_inter
  polator"
 android:duration="500">
</translate>
```

Both these transitions will last for 500 ms, and they use `AccelerateDecelerateInterpolator` as the interpolator type.

Similarly, to bring in the *swipe* transition while navigating back to the calling Activity, a translate transformation that moves the calling Activity from extreme left of the screen (`fromXDelta="-100%"`) to the center of the screen (`toXDelta="0%"`) is implemented in the l_to_r_enter_anim.xml, as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<translate
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:fromXDelta="-100%"
 android:toXDelta="0%"
 android:interpolator="@android:anim/accelerate_decelerate_inter
 polator"
 android:duration="500">
</translate>
```

And, to take out the called Activity from the center of the screen (`fromXDelta="0%"`) to the extreme right of the screen (`toXDelta="100%"`), we have a translate transformation implemented in the l_to_r_exit_anim.xml, as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<translate
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:fromXDelta="0%"
 android:toXDelta="100%"
 android:interpolator="@android:anim/accelerate_decelerate_inter
 polator"
 android:duration="500">
</translate>
```

Once animation XMLs are composed, corresponding XML files are applied on the `MainActivity` and `SubscribeActivity`, respectively. This transition occurs on click of a Button in the `MainActivity` that results in the launch of `SubscribeActivity`, as shown below.

```
public void onClick(View arg0) {
  Intent intent = new Intent(this, SubscribeActivity.class);
  startActivity(intent);
  overridePendingTransition(R.anim.r_to_l_enter_anim,
  R.anim.r_to_l_exit_anim); }
```

The animation XMLs are passed to the `overridePendingTransition()` method to specify an explicit transition animation to occur during navigation.

We also need the *swipe* animation to occur when we go back to the `MainActivity` from the `SubscribeActivity`; hence, we override the `onBackPressed()` method within the `SubscribeActivity`, which is called when the user presses the Back key of the device.

```
public void onBackPressed() {
  super.onBackPressed();
  finish();
  overridePendingTransition(R.anim.l_to_r_enter_anim,
  R.anim.l_to_r_exit_anim);}
```

## SUMMARY

This chapter has introduced the need for an app to adapt to multiple screen densities and screen sizes, and also described a few ways to achieve it. This becomes important due to the sheer variety of Android devices available in the market.

The chapter has elucidated two important concepts of Android 2D graphics: Drawables and Canvas. It has also discussed different scenarios where these two concepts can be used. Drawables are typically set as Android view backgrounds, whereas Canvas is used where there is a requirement to create/modify Android views.

Next, the chapter has delved into Android animation and has explored three different animation APIs, namely, drawable, view, and property. Drawable animation is the most simplistic of the three, and is used when frame-by-frame animation is to be performed. The view and property animations are transition animations wherein the start and end states are defined by the developer and the intermediate states are fashioned by Android.

## REVIEW QUESTIONS

1. Define strategies to deal with multiple screen densities and sizes in an app.
2. List types of drawables available in Android, and state their utility.
3. Define the types of animations supported by Android.
4. Differentiate between view and property animations.

## FURTHER READINGS

1. Supporting multiple screens: http://developer.android.com/guide/practices/screens_support.html
2. Drawable resources: http://developer.android.com/guide/topics/resources/drawable-resource.html
3. Property animation: http://developer.android.com/guide/topics/graphics/prop-animation.html

# 8 Multimedia

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Illustrate media containers and codecs.
- Design and implement media playback.
- Design and implement media capture and storage.

## 8.1 INTRODUCTION

While we are listening to music, accessing a video on demand, learning online, networking on social platforms, or playing games on smart devices, it is most likely that multimedia is being used. Multimedia surely has its place in providing an engaging and enriching experience to an app user. Multimedia, as the name speaks for itself, is all about the content in different forms such as audio, video, images, or animation.

This chapter introduces the powerful multimedia capabilities that Android provides to a developer via its multimedia APIs. These APIs help a developer to build apps with features that require interacting, recording, playing, and storing audio, video, and image elements. But, before getting into the nitty-gritty of these APIs and their capabilities, let us first explore some enablers related to multimedia that a developer needs to take into account while implementing apps.

## 8.2 AUDIO, VIDEO, AND IMAGES

An end user typically refers to video/audio using the respective file format (though loosely) such as an MP4 video or an Audio Video Interleave (AVI) video, but a developer needs to understand what this really means behind the scene, to implement multimedia features at its best.

An MP4 video (or any other video file such as AVI) is just like a zip file that can package files within it. In multimedia parlance, this packaging format is referred to as *container* format. A container may need to package video elements, audio elements, and metadata that provide information about the title or cover art of the video. Keeping audio and video elements together is usually referred to as multiplexing. MPEG-4 (.mp4), Flash (.flv), and Ogg (.ogv) are a few popular container formats that specify how to store various multimedia elements.

Audio/video, in its raw form, is typically very large that makes it unsuitable for storage or transport; therefore, before packaging it in a container format, it needs to be compressed (or encoded) using compression algorithms. These algorithms, which help in reducing the size of raw audio/video by compressing it, are referred to as *codecs* (compressor–decompressor). H.264 AVC (Advanced Video Coding), Theora, and VP8 are a few popular video codecs. MP3, AAC (Advanced Audio Coding), and Vorbis are a few popular audio codecs.

The Android platform supports several media codecs and container formats[1] to provide capabilities to play and record a wide variety of audio, video, and images. Figure 8.1 illustrates what we have learnt till now, through an example of an MPEG-4 container that has H.264 AVC-encoded video track, AAC LC-encoded audio track(s), and metadata (support data). It is important to note that all codecs may not be compatible with the container selected to package. For example, a Vorbis audio codec will not go with an MPEG-4 container.

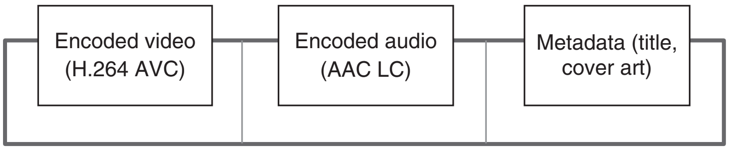


**Figure 8.1** | An MPEG-4 container with encoded audio and video elements.

[1] http://developer.android.com/guide/appendix/media-formats.html

Once media is available in a container, it is ready to play. To play it, first audio and video elements get separated (de-multiplexed). The individual elements are then decompressed back to the raw format using respective codec (the one that was used to compress it), and finally processed by the smart device in order to play. Decompressing the video elements results in displaying a series of images on the screen, and decompressing the audio elements results in a sound stream on the speakers of the device. Both these elements are synchronized with the help of metadata (such as language of an audio stream or aspect ratio of video) carried by individual media elements.

Several image types are supported by Android platforms such as jpeg, gif, bmp, and png. We have already explored images in Chapters 4 and 7. Let us now take a deep dive into multimedia APIs to play, record, and store media elements.

## 8.3 PLAYBACK

The multimedia framework in Android provides a rich set of APIs that support playing audio or video in mobile apps. The media may be stored on the device itself as an app resource, or streamed over a network. A developer may simply use the built-in app of Android to playback the media, but to get more flexibility and control on playback, in-app mechanism comes handy. The in-app mechanism is implemented using `MediaPlayer` API, which is pivotal to audio and media playback. Let us first delve into the built-in mechanism for playback and then take a deep dive into the `MediaPlayer` API, its usage, and implementation to explore the in-app mechanism of media playback.

### 8.3.1 Built-in App Mechanism

A built-in app mechanism is all about requesting the pre-existing Android app (that comes with the Android stack) to playback a media element from an app that requires the multimedia features. This mechanism is pretty straightforward from a developer's perspective as it only requires sending an Intent to play the required media using the built-in Android app. Rest is taken care by the built-in app itself.

To play an audio using a built-in app mechanism, we have to use an implicit Intent to start the intended music player app, as listed in Snippet 8.1. The data source (audio file music.mp3) that has to be played is hosted at http://www.mobmusicstore.com (just an example URL). The `setDataAndType()` method is used to set the data source and the type of audio (`audio/*`, in Line 3, * means any audio format) to be played.

```
1  Intent i = new Intent(Intent.ACTION_VIEW);
2  Uri uri = Uri.parse ("http://www.mobmusicstore.com/music.mp3");
3  i.setDataAndType(uri,"audio/*");
4  startActivity(i);
```

**Snippet 8.1** | Playing audio using an Intent.

Playing a video using a built-in mechanism is similar to playing an audio, and is listed in Snippet 8.2.

```
1  Intent i = new Intent(Intent.ACTION_VIEW);
2  Uri uri = Uri.parse ("http://www.mobmusicstore.com/video.mp4");
3  i.setDataAndType(uri,"video/*");
4  startActivity(i);
```

**Snippet 8.2** | Playing video using an Intent.

### 8.3.2 In-app Mechanism

An in-app mechanism gives more control and flexibility to a developer to handle the media playback. Unlike the built-in mechanism, an in-app mechanism does not request another app to play the desired media, rather plays it within the app. The `MediaPlayer` API is the underlying engine to achieve it. It supports to retrieve, decode, and playback the media elements (audio/video), as required.

A `MediaPlayer` object behaves as a state machine and has its own life cycle and states. Before a media stream can be played using a `MediaPlayer` object, it goes through the *idle*, *initialized*, *prepared*, and *started* states. The idle state is the first state that a new `MediaPlayer` object gets into, as soon as it is created. The next step is to set the data source – a file path or an HTTP URL of the media stream we want to play. The `setDataSource()` method is used to achieve this, and it takes the `MediaPlayer` object to the initialized state. Now, the media stream needs to be fetched and decoded so that it can be prepared for playback. The best practice to achieve this is by using `prepareAsync()` – an asynchronous method that prepares the `MediaPlayer` object without blocking the *main* thread. Once the `MediaPlayer` object is prepared, it can be started using `start()`, followed by `play()` to playback the media stream. The media stream can now be *paused* or *stopped* by calling `pause()` or `stop()`. Always remember to call `release()`, once the `MediaPlayer` object is not required anymore, which takes it to the *end* state.

The key life-cycle methods and states of the `MediaPlayer` object are listed for a quick reference in Table 8.1.

**Table 8.1** | `MediaPlayer` object's life-cycle methods and states

| Methods | States |
|---|---|
| `setDataSource()` | Initialized |
| `prepareAsync()` | Prepared |
| `start()` | Started |
| `pause()` | Paused |
| `stop()` | Stopped |
| `release()` | End |

A developer has to be very cautious of the `MediaPlayer` object's state while making a call to these methods. An invalid method call from any state may result in errors and exceptions. For example, calling the `pause()` method in started state will successfully pause the media and take the `MediaPlayer` object into the paused state. However, if the `pause()` method is called in the prepared state, it is considered as an invalid move, and takes the `MediaPlayer` object into an *error* state. For more information on valid and invalid states of a `MediaPlayer` object, readers are encouraged to refer to the Android Developers' resources.[2]

Let us now use the `MediaPlayer` object to play an audio, as shown in Snippet 8.3. The `create()` convenience method (Line 1) is used here to instantiate, initialize, and prepare the `MediaPlayer` object,

[2] http://developer.android.com/reference/android/media/MediaPlayer.html#Valid_and_Invalid_States

instead of using the `setDataSource()` and `prepareAsync()` methods explicitly. At Line 6, media playback is started using the `start()` method of the `MediaPlayer`.

```
1  MediaPlayer mediaPlayer =
   MediaPlayer.create(this,Uri.parse
   (“http://www.mobmusicstore.com/music.mp3”));
2  if(mediaPlayer==null){
3   Toast.makeText(this, “Unable to create Media player”,
     4000).show();}
4  else
5   Toast.makeText(this, “Playing song”, Toast.LENGTH_LONG).show();
6  mediaPlayer.start();
```

**Snippet 8.3** | Playing audio using `MediaPlayer`.

In our example (Snippet 8.3), the audio is sourced over the network, therefore the app requires an `INTERNET` permission, as shown in Line 1 of Snippet 8.4. In addition to this, we also need to keep the device on, to prevent any interference with playback. To ensure this, we need to provide a `WAKE_LOCK` permission (Line 2).

```
1  <uses-permission android:name=“android.permission.INTERNET”/>
2  <uses-permission android:name=“android.permission.WAKE_LOCK”/>
```

**Snippet 8.4** | AndroidManifest.xml entries for `MediaPlayer` permissions.

In addition to providing appropriate permissions as mentioned in Snippet 8.4, a developer also has to make sure that the CPU and Wi-Fi (if data source is accessed over Wi-Fi) hardware are kept awake for the duration of playback. This is achieved by holding the partial wake lock (Line 1) and Wi-Fi lock (Line 2), respectively, as shown in Snippet 8.5.

```
1  mediaPlayer.setWakeMode (this, PowerManager.PARTIAL_WAKE_LOCK);
2  WifiLock wifiLock=
      ((WifiManager)getSystemService(Context.WIFI_SERVICE)).
      createWifiLock(WifiManager.WIFI_MODE_FULL, “wifilock”);
3  wifiLock.acquire();
4  ...
5  wifiLock.release();
```

**Snippet 8.5** | Holding partial wake lock and Wi-Fi lock.

The partial wake lock is acquired by the `MediaPlayer` while playing, and is released once it is paused or stopped. However, the Wi-Fi lock has to be acquired by calling the `acquire()` method (Line 3). It is important to release the Wi-Fi lock when it is no longer required, to save the battery life, by calling the `release()` method (Line 5).

We have just explored how a `MediaPlayer` can play an audio that is sourced over the network. A `MediaPlayer` can also play an audio that is placed as a raw resource in an app, or stored in the local file system of the device, or accessed from a local Content Provider. To play an audio placed as a raw resource in an app or in the local file system of the device, we need to provide a resource identifier to the `create()` convenience method, as shown in Table 8.2.

**Table 8.2** | Playing an audio from different sources

| Audio Resource Placed As | create() |
|---|---|
| A raw resource in an app | `MediaPlayer mp = MediaPlayer.create(this,R.raw.music);` |
| A file in the local file system of the device | `MediaPlayer mp = MediaPlayer.create(this,Uri.parse("file:///sdcard/music.mp3"));` |

Playing video is a little bit more engaging than playing audio. Audio does not require any UI element for display; however, video needs a surface to render itself along with the audio. To playback a video, Android provides a convenient widget – `VideoView` – that encapsulates the underlying `MediaPlayer` to play the audio, and a surface to render the visuals. Snippet 8.6 demonstrates a video playback using the `VideoView`. The `setKeepScreenOn()` method (Line 4) ensures that the screen does not dim during playback.

```
1  VideoView vv = (VideoView) this.findViewById(R.id.videoView1);
2  Uri videoUri =
   Uri.parse("http://www.mobmusicstore.com/video.mp4");
3  vv.setVideoURI(videoUri);
4  vv.setKeepScreenOn(true);
5  vv.start();
```

**Snippet 8.6** | Playing a video using `VideoView`.

If a video is sourced over the network, we need to request the `INTERNET` permission, as discussed earlier in the case of audio playback, and shown earlier in Snippet 8.4 (Line 1). To play a video placed as a raw resource in an app or in the local file system of the device, we need to provide a resource identifier to the `setVideoURI()` method, as shown in Table 8.3.

**Table 8.3** | Playing a video from different sources

| Video Resource Placed As | setVideoURI() |
|---|---|
| A raw resource in an app | `vv.setVideoURI(Uri.parse("android.resource://" + getPackageName() + "/raw/" +"video"));` |
| A file in the local file system of the device | `vv.setVideoURI(Uri.parse(Environment.getExternalStorageDirectory().getPath()+"/video.mp4"));` |

We can also add media controls such as play, pause, rewind, and progress slider to the `MediaPlayer` through a `MediaController`. The media controls will appear in a window floating above our app and disappeas, if left idle for 3s. The `MediaController` also takes care of synchronizing the media controls as per the state of the `MediaPlayer`. Its usage along with `VideoView` is shown in a self-explanatory Snippet 8.7.

```
MediaController mc=new MediaController(this);
vv.setMediaController(mc);
@Override
 public boolean onTouchEvent(MotionEvent event) {
     super.onTouchEvent(event);
     mc.show();
     return true;
 }
```

**Snippet 8.7** | Media controls for `VideoView`.

## 8.4 LET'S APPLY

In this section, let us add the functionality to view a show that is currently being played on a channel. For demonstration purposes, only a sample video is streamed from the enterprise's backend server. The user can view the current show by clicking on the *Stream current show* Button in the `ShowsListFragment`, as shown in Fig. 8.2.

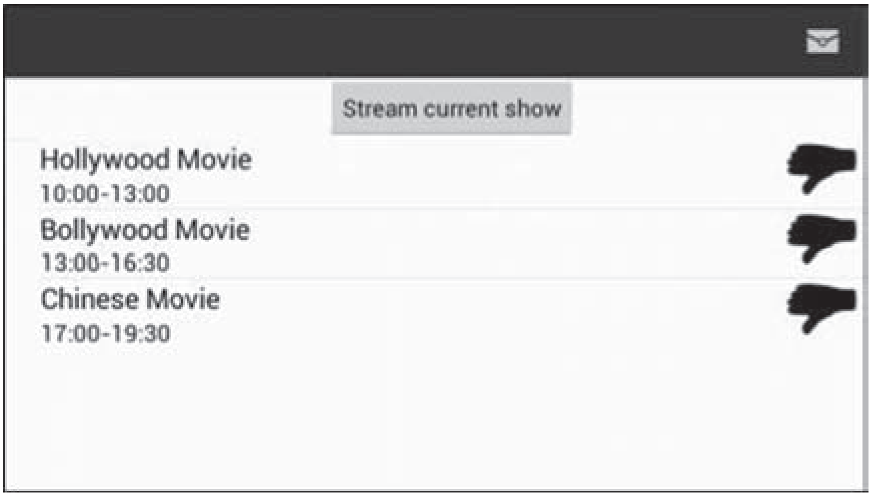


**Figure 8.2** | `ShowsListFragment` with Stream current show option.

On click of this button, the `playMedia()` method described below is called. This method creates a `MediaFragment` instance that is used to view the streamed video. This code is implemented in `TVGuide` Activity, which is hosting the `ShowsListFragment`.

```
public void playMedia(View v){
   FragmentTransaction ft =
   getFragmentManager().beginTransaction();
   Fragment prev =
   getFragmentManager().findFragmentByTag("dialog");
   if (prev != null) {
       ft.remove(prev);}
   ft.addToBackStack(null);
   DialogFragment newFragment =
   MediaFragment.newInstance(channelName);
   newFragment.show(ft, "dialog");
}
```

The `MediaFragment` class is a user-defined class, as described below. It extends the `DialogFragment` class – a specialized class to display a Fragment in a dialog window.

```
public static class MediaFragment extends DialogFragment {
   String showName=null;
   static MediaFragment newInstance(String name) {
      MediaFragment f = new MediaFragment();
      Bundle args = new Bundle();
```

```
        args.putString("channel_name", name);
        f.setArguments(args);
        return f;}
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        channelName = getArguments().getString("channel_name");
        setStyle(DialogFragment.STYLE_NO_TITLE,
        android.R.style.Theme_Holo);}
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
    container,Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.mediafragment_layout,
        container, false);
        VideoView videoView = (VideoView)
        v.findViewById(R.id.videoView1);
        MediaController mController = new
        MediaController(getActivity());
        videoView.setMediaController(mController);
        videoView.setVideoURI(Uri.parse(AppConfig.ServerAddress+Ap
        pConfig.MediaRelativeAddress));
        videoView.start();
        mController.show(3000);
        return v;}
}
```

The video is played back in a `VideoView` container. Figure 8.3 depicts a sample video being played in the app.

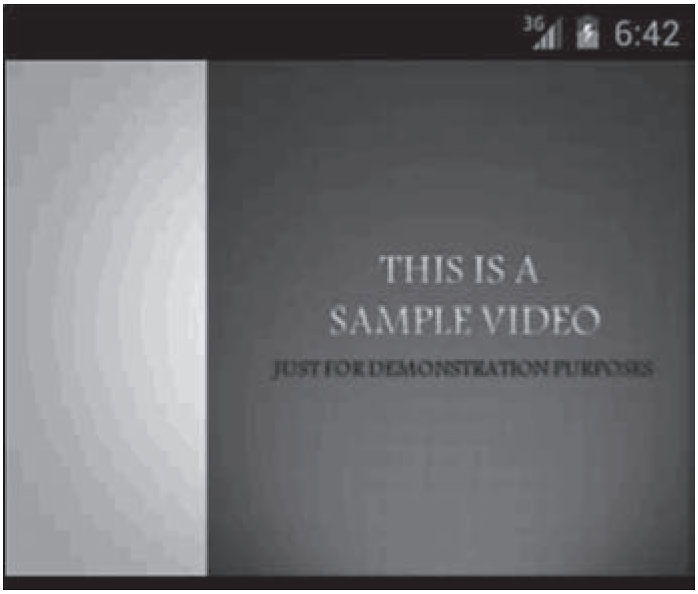


**Figure 8.3** | Snapshot of Sample video.

## 8.5 CAPTURE AND STORAGE

The multimedia framework in Android also provides a rich set of APIs that support the recording of audio and video in mobile apps. Similar to playback, a developer has two choices for recording – either to simply launch the built-in app using Intent to record the media, or to use the in-app mechanism to get more flexibility and fine-grained control on recording. The in-app mechanism is implemented using the `MediaRecorder` API, which is pivotal to audio and media capture. Let us first delve into the built-in mechanism for recording, and then take a deep dive into the `MediaRecorder` API, its usage, and implementation to explore the in-app mechanism of media capture.

### 8.5.1 Built-in App Mechanism

The simplest way to start an audio recording is using the `RECORD_SOUND_ACTION` static constant of `MediaStore` in an Intent passed to the `startActivityForResult()`, as shown in Snippet 8.8. This will launch the native audio recorder that allows user to start, stop, and preview the captured audio.

```
Intent intent = new Intent(
MediaStore.Audio.Media.RECORD_SOUND_ACTION);
startActivityForResult(intent, REQUEST_CODE);
```

**Snippet 8.8** | Recording audio using an Intent.

Similarly, the easiest technique to capture a video is by using the `ACTION_VIDEO_CAPTURE` static constant of `MediaStore` in an Intent passed to the `startActivityForResult()`, as shown in Snippet 8.9. This will launch the native video camera that allows a user to start, stop, and preview the captured video. We can also specify the video quality by using `EXTRA_VIDEO_QUALITY` extra, supported by the video capture action. The value `1` signifies high-resolution video (Line 2), and the value `0` signifies a low-resolution video.

```
1  Intent intent = new Intent( MediaStore.ACTION_VIDEO_CAPTURE);
2  intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY,1);
3  startActivityForResult(intent,REQUEST_CODE);
```

**Snippet 8.9** | Recording video using an Intent.

On similar lines, to capture an image, the `ACTION_IMAGE_CAPTURE` static constant of `MediaStore` is used in an Intent passed to the `startActivityForResult()`, as shown in Snippet 8.10. This will launch the native camera app that allows a user to capture the image. After successful capture, a small-size bitmap is returned as an extra in the Intent. For a full-size image, we have to specify a Uri value in the `EXTRA_OUPUT` extra (Line 5) supported by the image capture action. The full-size image will be stored at this Uri.

```
1  Intent intent = new Intent( MediaStore.ACTION_IMAGE_CAPTURE);
2  String filepath= Environment.getExternalStorageDirectory()+
   "/mypicture.jpg";
3  File myImage= new File(filepath);
4  Uri imageUri = Uri.fromFile(myImage);
5  intent.putExtra(MediaStore.EXTRA_OUTPUT,imageUri);
6  startActivityForResult(intent, REQUEST_CODE);
```

**Snippet 8.10** | Capturing images using an Intent.

## 8.5.2 In-app Mechanism

The in-app mechanism gives more control and flexibility to a developer to handle media capture and storage. Unlike built-in mechanism, the in-app mechanism does not request another app to capture the media, rather does it within the app under consideration. The `MediaRecorder` API is the underlying engine to achieve this. It allows us to specify the media (audio/video) source, encoders to use while recording, and the output format.

A `MediaRecorder` object also behaves as a state machine, and has its own life cycle and states. Therefore, the order in which we configure and manage the `MediaRecorder` is very important. We start with creating a `MediaRecorder` object that takes it to the *initial* state. From here, the `MediaRecorder` object moves to the *initialized* state, once it is assigned with input sources to record from, by using the `setAudioSource()/setVideoSource()` method. For example, a camera is typically an input source for capturing video. The next step is to define the output format by using the `setOutputFormat()` method that will take the `MediaRecorder` object to the *data source configured* state. In this state, the audio/video encoder, frame rate, and output size are specified, along with the path to store the captured media. The `setAudioEncoder()/setVideoEncoder()` method is used to specify the media encoder for recording. The `setVideoFrameRate()` and `setVideoSize()` methods are used to specify the frame rate and output size, respectively. The `setOutputFile()` method is used to specify the path of the output file. From here, the `MediaRecorder` object moves to the *prepared* state by calling the `prepare()` method. Once the `MediaRecorder` object is prepared, it can move to the *recording* state using the `start()` method to capture the media stream. The media stream can now be *stopped* anytime by calling the `stop()` method, which will once again take the `MediaRecorder` object to the *initial* state. Always remember to call the `release()` method once the `MediaRecorder` object is not required anymore, which will take it to the *released* state.

The key life-cycle methods and states of the `MediaRecorder` object are listed for a quick reference in Table 8.4.

**Table 8.4** | `MediaRecorder` object's life-cycle methods and states

| Methods | States |
|---|---|
| `setAudioSource()/setVideoSource()` | Initialized |
| `setOutputFormat()` | Data source configured |
| `prepare()` | Prepared |
| `start()` | Recording |
| `stop()` | Initial |
| `release()` | Released |

Similar to the `MediaPlayer` object, a developer has to be cautious of the `MediaRecorder` object's state, while making call to its methods. An invalid method call from any state may result in errors and exceptions. For more information on the `MediaRecorder` state machine, readers are encouraged to refer to the Android Developers' resources.[3]

Let us now use the `MediaRecorder` and `Camera` objects to capture a video, as shown in the following snippets. The first step is to get a `Camera` handle to access the underlying camera hardware and set up a `SurfaceView` for previewing the video being captured. The next step is to prepare a `MediaRecorder`

[3] http://developer.android.com/reference/android/media/MediaRecorder.html

for capturing the video. Once the `MediaRecorder` object is prepared, we can start recording. Once the recording is over, we should release both the `MediaRecorder` and `Camera` objects.

For an app to access the underlying camera hardware, it needs to request the `CAMERA` permission, as shown in Line 1 of Snippet 8.11. We also need to declare that our app requires camera features, and this is achieved using the `<uses-feature>` tag in the manifest file of the app, as shown in Line 2 of Snippet 8.11. This tag ensures that the app is not visible on Google Play to those devices that do not have the camera feature.

```
1  <uses-permission android:name=“android.permission.CAMERA”></uses-
   permission>
2  <uses-feature android:required=“true”
   android:name=“android.hardware.camera”/>
```

**Snippet 8.11** | Camera access permission and feature declaration in AndroidManifest.xml.

Once camera-related permissions are set, we need to get a handle of `Camera` object. This is achieved by calling the `open()` method (Line 4, Snippet 8.12) of the `Camera` class. This class is a client for the camera service that manages the actual camera hardware. We always need to check for exceptions (Line 6–8) while trying to access the camera as it may be unavailable or locked by another app. The `getCameraInstance()` is a user-defined static method, as listed in Snippet 8.12, to achieve the said functionality and should be part of our Activity.

```
1  public static Camera getCameraInstance(){
2    Camera c = null;
3    try {
4       c = Camera.open();
5    }
6    catch (Exception e){
7       Log.e(TAG, “CAMERA ERROR!!!”+e);
8    }
9    return c;
10 }
```

**Snippet 8.12** | Getting a `Camera` handle.

The next step is to set up a `SurfaceView` (used as a preview surface) and then associate it with the camera. `SurfaceHolder` is the underlying surface on which Android renders a video. We use the `getHolder()` method of the `SurfaceView` to access the `SurfaceHolder` (Line 11, Snippet 8.13). Now, we need to implement the `SurfaceHolder.Callback` to listen to surface-related changes (surface being created/destroyed). The `setPreviewDisplay()` method is used to associate the surface with the camera, as shown in Line 19.

```
1  public class CameraPreview extends SurfaceView implements
   SurfaceHolder.Callback {
2
3    private SurfaceHolder holder;
4    private Camera camera;
5    private String TAG = “CAMERA_DEMO”;
6
7    public CameraPreview(Context context, Camera camera) {
8       super(context);
9
```

```
    this.camera =camera;
    holder = getHolder();
    holder.setKeepScreenOn(true);
    holder.addCallback(this);
    . . .
}
@Override
public void surfaceCreated(SurfaceHolder holder) {
 try {
    camera.setPreviewDisplay(holder);
    camera.startPreview();
 } catch (IOException e) {
    Log.d(TAG, "Error setting camera preview: " +
    e.getMessage());
}}}
```

**Snippet 8.13** | Setting up a `SurfaceView`.

Once the `Camera` and `SurfaceView` are prepared, we can now acquire the `Camera` (Line 5, Snippet 8.14) and prepare a `CameraPreview` (Lines 13 and 14).

```
1 public void onCreate(Bundle savedInstanceState) {
2   super.onCreate(savedInstanceState);
3   setContentView(R.layout.main);
4
5   c = getCameraInstance();
6   if(c==null)
7   {
8      Log.d(TAG, "Error accessing camera!!!");
9      Toast.makeText(this, "Error accessing camera!!!",
        5000).show();
10     finish();
11  }
12
13  cameraPreview = new CameraPreview(this, c);
14  ((FrameLayout)findViewById(R.id.camera_preview)).addView(camera
    Preview);. . .
15 }
```

**Snippet 8.14** | Preparing a `CameraPreview`.

The next step is to specify a location (the directory path and filename) to store the video. We have created a `getVideoUri()` method to achieve this, as shown in Snippet 8.15. To store video on a Secure Digital (SD) card, we need to first check if the SD card is mounted, as shown in Line 2. Lines 7–13 enlist the steps to create a directory in shared location to store the media file. Finally, in Line 16, we are creating a file that is to be used as the output file by the `MediaRecorder`.

```
1  private Uri getVideoUri() {
2    if(!Environment.getExternalStorageState().equalsIgnoreCase(
         Environment.MEDIA_MOUNTED)) {
3
```

```
        return null;

    }
    File mediaStorageDir = new
    File(Environment.getExternalStoragePublicDirectory(Environment.
    DIRECTORY_PICTURES), TAG);

    if (! mediaStorageDir.exists()){
      if (! mediaStorageDir.mkdirs()){
        Log.d(TAG, "failed to create directory");
          return null;
      } }

    String timeStamp = new
    SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
    File mediaFile=new File(mediaStorageDir.getPath() +
    File.separator + "VID_"+ timeStamp + ".mp4");
    return Uri.fromFile(mediaFile);   }
```

**Snippet 8.15** | Preparing storage.

To enable writing on an SD card, the app needs to request the `WRITE_EXTERNAL_STORAGE` permission. It also needs to request the `RECORD_AUDIO` permission to enable audio recording. Snippet 8.16 shows the preparation of a `MediaRecorder`, as explained earlier in this section.

```
boolean prepareVideoRecorder(){
  recorder = new MediaRecorder();


  // Step 1: Unlock and set camera to MediaRecorder
  c.unlock();
  recorder.setCamera(c);

  // Step 2: Set sources
  recorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
  recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

  // Step 3: Check for CamcorderProfile and use it, if available
  recorder.setProfile(CamcorderProfile.get(
   CamcorderProfile.QUALITY_HIGH));

  // Step 4: Set output file
  videoUri = getVideoUri();
  recorder.setOutputFile(videoUri.getPath());
  // Step 5: Set the preview output
  recorder.setPreviewDisplay(cameraPreview.getHolder().
  getSurface());

  // Step 6: Prepare configured MediaRecorder
  try {
     recorder.prepare();
  } catch (IllegalStateException e) {
     Log.e(TAG, "IllegalStateException preparing MediaRecorder:
     " + e.getMessage());
     releaseMediaRecorder();
     return false;
  } catch (IOException e) {
     Log.e(TAG, "IOException preparing MediaRecorder: " +
      e.getMessage());
     releaseMediaRecorder();
     return false;
  }
 return true;    }
```

**Snippet 8.16** | Preparing a `MediaRecorder`.

Snippet 8.17 demonstrates how to start and stop recording.

```
public void onClick(View v) {
 if (v == stopRecording) {
   isRecording = false;
   recorder.stop();
   releaseMediaRecorder();
   c.unlock();
   ...

 } else if (v == startRecording) {

     if (prepareVideoRecorder()) {
       recorder.start();
       isRecording = true;
       ...
     } else {
      releaseMediaRecorder();
     statusTextView.setText("Error while trying to record");}}}
```

**Snippet 8.17** | Starting/stopping recording.

We must explicitly release the `MediaRecorder` and `Camera` when we have finished recording. If camera is not released explicitly, it will remain locked, and subsequent attempts to access it will fail. Snippet 8.18 enlists the steps to achieve this.

```
private void releaseMediaRecorder(){
  if (recorder != null)  {
     recorder.reset();   // clear recorder configuration
     recorder.release(); // release the recorder object
     recorder = null;
     c.lock();           // lock camera for later use
  }
}

```

```
private void releaseCamera(){
  if (c != null){
     c.release();  // release the camera for other apps
     c = null;
  }
}
@Override
protected void onPause() {
  super.onPause();
  releaseMediaRecorder();
  releaseCamera();
}
```

**Snippet 8.18** | Releasing `MediaRecorder` and `Camera`.

We have just explored recording a video. Recording an audio is much simpler as it does not require a camera or a preview surface. We just need to identify the storage location of an audio (default storage directory is `DIRECTORY_MUSIC`), prepare the `MediaRecorder`, and start audio recording. Requesting appropriate permissions and releasing the `MediaRecorder` should also be taken care of. Interested readers may take it up as an exercise to implement audio recording.

## SUMMARY

This chapter has introduced the multimedia concepts such as audio, video, images, containers, and codecs. It has also discussed the role played by containers and codecs while playing and capturing the media elements.

The chapter has presented two approaches for playback – built-in app and in-app mechanisms. The built-in app mechanism is implemented using Intents, whereas the in-app mechanism is implemented with the help of MediaPlayer API. The chapter has also introduced the MediaPlayer behavior as a state machine, and configuration and implementation of its object to achieve playback of media elements sourced from raw resource, native file system, over the network, etc.

The chapter has also presented two approaches for capture and storage of multimedia elements – built-in app and in-app mechanisms. Again, the built-in app mechanism is achieved using Intents, whereas the in-app mechanism is implemented with the help of MediaRecorder API. It has also discussed MediaRecorder behavior as a state machine, and its configuration and implementation to achieve media capture from various sources.

## REVIEW QUESTIONS

1. Define media container and codec with relevant examples.
2. List popular media formats supported by Android.
3. Illustrate the states and relevant methods of MediaPlayer API. What permissions are required to do media playback over a Wi-Fi network?
4. Define the role of MediaController.
5. Explain the Intent object that is used to capture a video using the built-in app mechanism.
6. Illustrate the states and relevant methods of MediaRecorder API.

## FURTHER READINGS

1. Supported media formats in Android: http://developer.android.com/guide/appendix/media-formats.html
2. Valid and invalid states of a MediaPlayer object: http://developer.android.com/reference/android/media/MediaPlayer.html#Valid_and_Invalid_States
3. MediaRecorder state machine: http://developer.android.com/reference/android/media/MediaRecorder.html

# 9 Location Services and Maps

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Illustrate Google Play services.
- Set up a development environment to leverage Google Play services.
- Comprehend Location services and Google Maps.
- Create location-aware apps.
- Add maps to an app and make it interactive.

## 9.1 INTRODUCTION

An app on your smart device automatically silencing the ringer as you walk into your office, or warning you when you are driving too fast, or providing turn-by-turn driving directions, or helping you find nearby stores and restaurants suggests that it is most likely using location services or maps, or both.

The power of determining location and plotting it on map brings in a lot of useful scenarios for app users, both in consumer and enterprise mobility space. Apps can simply utilize the ability of a smart device to find out its current location, and then apply this information to provide various services to the end user.

In this chapter, we are going to start with exploring Google Play services that enable Android apps to leverage the power of Location services and Maps. Following which, we will take a deep dive into various features, applications, and implementation details of Location services and Maps.

## 9.2 GOOGLE PLAY SERVICES

Google Play services is a platform that enables an app to access Google services such as Google Maps, Google+, and Location services. These services are not included in the Android platform. Rather, they work in a client–server model, where in the server component of these services is hosted by Google servers, and the client libraries are provided for each such service, using which a developer implements the client-side functionality to interface with these services.

From an end-user's perspective, it is just an app (Google Play services) that resides on the user's device. This app runs in the background to connect to Google services, either to leverage the functionality of individual services in an app, or to receive the latest updates of these services, via Google Play Store app. The benefit of receiving the latest updates on the user's device is that this app is always up-to-date with the most recent release of Google Play services, irrespective of the Android version of the device. It is this app that glues together the client and server components of Google Play services.

As Google Play services are not bundled with the Android platform, a developer has to use the Google Play services Software Development Kit (SDK) to build apps that utilize these services. The Android SDK manager is used to download and install the Google Play services SDK that eventually sits in the extras subfolder of the Android SDK environment folder. Further, we need to copy the Google Play services library project (at <android-sdk>/extras/google/google_play_services/libproject/google-play-services_lib/) that comes with the Google Play services SDK into our workspace folder, and refer this library project in our Android app's project.[1] Any Android project that leverages Google Play services must be built using Google APIs. If these APIs are not already installed while setting up the development environment, we need to install them as well, using the Android SDK Manager.

A mobile app has to make sure that Google Play services is available on the device before starting using its features, as depicted in Snippet 9.1. The `isGooglePlayServicesAvailable()` method of the `GooglePlayServicesUtil` (Line 2) is used to make sure that the Google Play services app is installed and working on the device, and there is compatibility between this app and the client libraries that the developer has used to build the mobile app. The `isConnectedToLocationServices()` is a user-defined method that invokes the `isGooglePlayServicesAvailable()` method, and is called in the `onResume()` method of an Activity.

[1] Set up Google Play services SDK: http://developer.android.com/google/play-services/setup.html

```
1  private boolean isConnectedToLocationServices() {
2    int resultCode= GooglePlayServicesUtil
         .isGooglePlayServicesAvailable(this);
3    if (resultCode == ConnectionResult.SUCCESS) {
4      return true;
5    } else {
6      Dialog dialog=
       GooglePlayServicesUtil.getErrorDialog(resultCode,this,0);
7      dialog.show();
8    }
9    return false;
10 }
```

**Snippet 9.1** | Checking availability of Google Play services.

If the Google Play services app is up-to-date, the `isGooglePlayServicesAvailable()` method returns `SUCCESS` (Line 3) and we can further proceed with using the required service, otherwise the `getErrorDialog()` method is called (Line 6). This method displays an error message to the user and in turn allows the user to either download the required Google Play services app or enable it in the device settings, as the case may be.

After familiarizing ourselves with the motivation behind Google Play services, the components involved in orchestrating them, and the setup required for implementing their features in an app, let us now explore the two widely used Google Play services, Location services and Maps, in the following sections.

## 9.3 LOCATION SERVICES

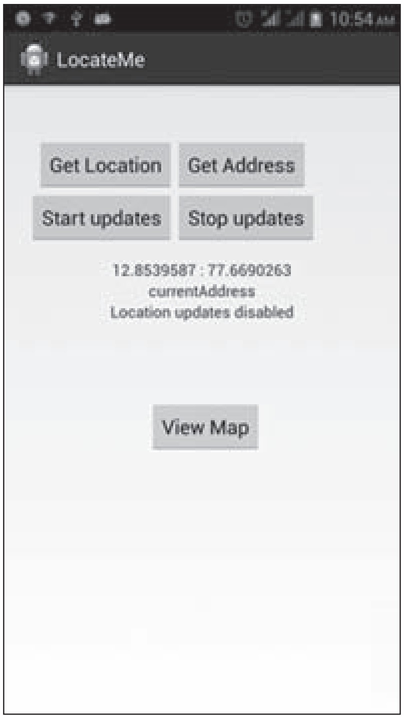
The capability of a smart device to locate itself at a given point of time, and track its current location, as it moves along with the user, provides location awareness to the apps. This awareness of current location helps an app to capitalize the current context of the user and can be leveraged in varieties of use cases across industries. For example, retail outlets can provide an exciting in-store experience by providing in-store offers or mobile brochures. Banks can guide the user to the nearest Automated Teller Machine (ATM). Hospitality businesses can help in navigating a user from his or her current location to their nearby food outlets.

Location services is a very powerful suite of functionalities ranging from capturing and tracking the user location, or retrieving the user's current address based on the location coordinates, or even updating an user when he or she is in the proximity of a location of his or her interest.

### 9.3.1 Locating the User

The most primitive task of the Location services is to provide the current location of a user. Mobile devices use several techniques to determine their location. Unique cell id of the nearest cellular tower can be used to approximately determine the current location. If the device is receiving signals from two or more towers simultaneously, cell triangulation technique can be used for better accuracy. The IP address of Wi-Fi network to which the device is connected, can also be used to determine the location. All these techniques do not require a dedicated hardware in the device, and vary in accuracy. However, location can be accurately determined using GPS (Global Positioning System), a dedicated hardware in the device. From a developer's standpoint, this location has to be retrieved in an app, as and when required, from Location services.

In this section, let us explore getting the current location from Location services, using a simple app – LocateMe – as shown in Fig 9.1.



12.8539587 : 77.6690263
currentAddress
Location updates disabled

**Figure 9.1** | LocateMe app displaying current location

The `getCurrentLocation()` method (Lines 15–22) of Snippet 9.2 is invoked on click of the *Get Location* Button (see Fig. 9.1). In this method, an instance of the `LocationClient` – `mClient` – is used to retrieve the location of the user. The `getLastLocation()` method of the `LocationClient` returns a `Location` object containing the last known coordinates. The `LocationClient` constructor used to instantiate `mClient` accepts three parameters – context, class implementing `ConnectionCallbacks` interface, and class implementing `onConnectionFailedListener` interface – as shown in Line 13.

```
1  public class MainActivity extends FragmentActivity implements
   GooglePlayServicesClient.ConnectionCallbacks,
   GooglePlayServicesClient.OnConnectionFailedListener {
2
3    private LocationClient mClient;
4    ...
5    @Override
6    protected void onCreate(Bundle savedInstanceState) {
7      super.onCreate(savedInstanceState);
8      setContentView(R.layout.activity_main);
9      ...
10     setUpLocationParams();
11   }
12   private void setUpLocationParams() {
13     mClient = new LocationClient(this, this, this);
14   }
```

```
 public void getCurrentLocation(View v) {
   if (isConnectedToLocationServices()) {
    if (mClient.isConnected()&&mClient.getLastLocation()!=null)
     {
      Location currentLocation=new
        Location(mClient.getLastLocation());
      coordinates.setText(currentLocation.getLatitude()+“:”+
     currentLocation.getLongitude());
    }
   }
 }
 @Override
 protected void onStart() {
   super.onStart();
   mClient.connect();
 }
 @Override
 public void onStop() {
    super.onStop();
    mClient.disconnect();
 }
 @Override
 public void onConnectionFailed(ConnectionResult
  connectionResult) {

 if (connectionResult.hasResolution()) {
   try {
      // Use startResolutionForResult() to start the Activity
      connectionResult.startResolutionForResult(this,
        ACTIVITY_REQUEST_CODE);
   } catch (IntentSender.SendIntentException e) {
      e.printStackTrace();
   }
 } else {
     Dialog errorDialog = GooglePlayServicesUtil.getErrorDialog(
          connectionResult.getErrorCode(), this,
          ACTIVITY_REQUEST_CODE);

     if (errorDialog != null) {
       // Create a new DialogFragment
      ErrorDialogFragment errorFragment = new
        ErrorDialogFragment();
      errorFragment.setDialog(errorDialog);
      errorFragment.show(getSupportFragmentManager(),“LOCATION
                              DEMO”);
     }
 }
}
@Override
public void onConnected(Bundle arg0) {
```

```
56
57  }
58  @Override
59  public void onDisconnected() {
60    updates_status.setText("Disconnected...");
61  }
62 }
```

**Snippet 9.2** | Retrieving current location from Android Location services.

In the `onStart()` method of the `MainActivity`, the `connect()` method of `mClient` is invoked (Line 26) to connect to Location services. This triggers the `onConnected()` callback (Line 55) of `ConnectionCallbacks`. Once the `onConnected()` method is invoked, the app is free to access the various capabilities provided by Location services. Similarly, in the `onStop()` method of the `MainActivity`, the `disconnect()` method of `mClient` is invoked (Line 31). This triggers the `onDisconnected()` method (Line 59) of `ConnectionCallbacks`, after which the app stops using Location services.

The `onConnectionFailed()` method (Lines 34–53) of `onConnectionFailedListener` gets invoked with a `ConnectionResult` instance, in case the attempt to connect to Location services from the app fails. In this method, the `hasResolution()` method of the `ConnectionResult` instance is used to determine if the connection error can be resolved by Google Play services. If the error can be resolved, as in cases such as when the user has to sign in, the `hasResolution()` method returns `true`, and we launch an Intent to facilitate such interactions using the `startResolutionForResult()` method. However, if the error cannot be resolved, we use the `getErrorDialog()` method of the `GooglePlayServiceUtil` to display an error to the user (Line 44).

Whenever the `startResolutionForResult()` method is invoked to resolve the error, Google Play services starts a new Android component for resolving the error. Once this error is resolved, the control passes to the `onActivityResult()` of the Activity that invoked the `startResolutionFor-Result()` method. In this method, we check if the error was resolved by Google Play services using the `resultCode` argument, as shown in Snippet 9.3.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
  switch (requestCode) {
  case ACTIVITY_REQUEST_CODE :
    switch (resultCode) {
      case Activity.RESULT_OK:
        updates_status.setText("Result is OK...");
        break;
      default:
        updates_status.setText("Can't resolve error...");
        break;
    }
  default:
  break;
  }
}
```

**Snippet 9.3** | `onActivityResult()` method of the `MainActivity`.

On the basis of the desired accuracy in an app, it has to request either the `ACCESS_FINE_LOCATION` permission or the `ACCESS_COARSE_LOCATION` permission in the app manifest. The `ACCESS_FINE_LOCATION` permission in turn enables the `ACCESS_COARSE_LOCATION` permission as well. Just to recall, checking the availability of Google Play services (Section 9.2, Snippet 9.1) is a prerequisite to access Location services.

## 9.3.2 Tracking User Location

Location services not only determine location when the device is stationary, as seen in the previous example, but also have the capabilities to do the same while it is moving. This will come handy in those scenarios where an app needs to process the continuous location updates such as computing the speed of the user while running or the distance covered while cycling.

Let us now modify the LocateMe app to include the capability of tracking user location by capturing the continuous location updates at regular intervals of time, as shown in Snippet 9.4. To start with, we need to implement the `LocationChanged` listener in the `MainActivity`, and override the `onLocationChanged()` method (Lines 13–17), where we display the location coordinates every time the location changes.

```
1  public static final long UPDATE_INTERVAL=5000;
2  public static final long FASTEST_UPDATE_INTERVAL=2000;
3  private LocationRequest mLocationRequest;
4
5  private void setUpLocationParams() {
6    mLocationRequest = LocationRequest.create();
7    mLocationRequest.setInterval(UPDATE_INTERVAL);
8    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCU
     RACY);
9    mLocationRequest.setFastestInterval(FASTEST_UPDATE_INTERVAL);
10   mClient = new LocationClient(this, this, this);
11 }
12 @Override
13 public void onLocationChanged(Location arg0) {
14   if(arg0!=null){
15     coordinates.setText(arg0.getLatitude()
                             +“:”+arg0.getLongitude());
16   }
17 }
18 @Override
19 public void onConnected(Bundle arg0) {
20   if (mUpdatesRequested) {
21     mClient.requestLocationUpdates(mLocationRequest, this);
22   }
23 }
24 @Override
25 public void onStop() {
26   super.onStop();
27   if (mClient.isConnected()) {
28     mClient.removeLocationUpdates(this);
29   }
30   mClient.disconnect();
31 }
```

**Snippet 9.4** | Location tracking feature of LocateMe app.

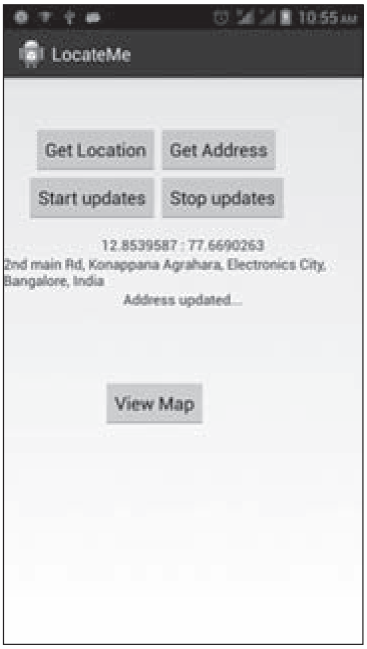
We have also defined a method `setUpLocationParams()` to create and configure the `LocationRequest` object for location updates. Here we are configuring the location update interval using the `setInterval()` method, and setting its value as 5000 ms (Line 7). In case another app on the same device is also using Location services and has a shorter update interval, our app will get updated more frequently than desired. In such cases, the `setFastestInterval()` method comes handy to restrict our app to not receive the location updates below the limit set in this method, which is 2000 ms in our case (Line 9).

Now, to start getting the location updates, we need to invoke the `requestLocationUpdates()` with the `LocationRequest` object, and the class implementing the `LocationChanged` listener as arguments, in the `onConnected()` callback. Always remember to stop receiving the location updates when not required, by using the `removeLocationUpdates()` method.

### 9.3.3 Retrieving Location Address

Location-aware apps, typically, perform various tasks after determining the current location coordinates. For example, an app that helps a user to share the current location address with his or her family and friends on a social platform has to further determine the street address based on the current location coordinates captured. This process of determining the street address from the location coordinates is referred to as reverse geocoding. Android facilitates this process using the `Geocoder` API.

Let us now further modify the LocateMe app to include the capability of reverse geocoding, as shown in Fig. 9.2.



12.8539587 : 77.6690263
2nd main Rd, Konappana Agrahara, Electronics City, Bangalore, India
Address updated...

**Figure 9.2** | LocateMe app displaying the current address.

The `getCurrentAddress()` method (Snippet 9.5) is called on tap of the *Get Address* Button. In this method, we first ensure that the `Geocoder` is available (Line 2). Once this is ensured, we can use the `getFromLo–cation()` method of the `Geocoder` to retrieve the location address based on the latitude and longitude (Line 23). There might be several street addresses that are mapped to a specific latitude–longitude; therefore, the `getFromLocation()` method may return multiple street addresses. This can be restricted by using the third parameter of this method. We have requested only one street address here (Line 23). Address lookup may take a long time to get resolved; therefore, it is advisable to execute this method in an AsyncTask (Line 12).

```
1  public void getCurrentAddress(View v){
2    if(!Geocoder.isPresent()){
3      updates_status.setText(“Geocoder not present!!!”);
4      return;
5    }
6    if(isConnectedToLocationServices()){
7      Location currentLocation = mClient.getLastLocation();
8      updates_status.setText(“Getting address. PLease wait...”);
9     (new MainActivity.GetAddressTask(this)).
       execute(currentLocation);
10   }
11 }
12 protected class GetAddressTask extends AsyncTask<Location, Void,
   String> {
13   Context context;
14   public GetAddressTask(Context context) {
15     super();
16     this.context = context;
17   }
18   @Override
19   protected String doInBackground(Location... params) {
20    Geocoder geocoder = new Geocoder(context,Locale.getDefault());
21    List <Address> addresses = null;
22    try {
23      addresses= geocoder.getFromLocation(params[0].getLatitude(),
                 params[0].getLongitude(), 1);
24    } catch (IOException exception1) {
25      updates_status.setText(“IOException while getting
                                address...”);
26      return “IOException while getting address...”;
27    // Catch incorrect latitude or longitude values
28    } catch (IllegalArgumentException exception2) {
29      updates_status.setText(“Invalid coordinates...”);
30      return “Invalid coordinates...”;
31    }
32    if (addresses != null && addresses.size() > 0) {
33      // Get the first address
34      Address address = addresses.get(0);
35
36      String addressText = String.format(“%s, %s, %s”,
        address.getMaxAddressLineIndex() > 0 ?
        address.getAddressLine(0) :“”,
        address.getLocality(),address.getCountryName());
37      return addressText;
38    } else {
```

```
39      return "No address found...";
40    }
41  }
42  @Override
43  protected void onPostExecute(String address) {
44    updates_status.setText("Address updated...");
45    address_from_coords.setText(address);
46  }
47 }
```

**Snippet 9.5** | Reverse geocoding feature of LocateMe app.

Street address(es) is retrieved as a list of `Address` object(s) wherein each address is a set of Strings. We can retrieve various address-related information from these set of Strings such as address line, country code, country name, locale, locality, and postal code. In Line 36, we have retrieved the address line, locality, and country name using the `getAddressLine()`, `getLocality()`, and `getCountryName()` methods, respectively.

Besides locating the user, tracking user location, and retrieving location address, Location services provide much more sophisticated capabilities that can be leveraged by Android apps. Apps can create and monitor the areas of interest of a user to notify the user when he or she is in or around those areas. Such areas of interest are usually referred to as geofences, and are typically used in scenarios such as reminding the user to buy household items or notifying in-store offers, once he or she is in the proximity of a supermarket. Location services also provide the capability to determine the user's current activity such as walking, jogging, or driving. This capability can be leveraged by apps to strategize the location-aware content based on the current activity. For example, different paths can be served to the user based on whether he or she is jogging, bicycling, or driving a car. For more information on implementing these capabilities, readers are encouraged to refer to the Android Developers' resources.[2]

## 9.4 MAPS

As they say, with a smartphone you can never be lost. Thanks to the collective power of Location services and Maps! While Location services determine the geographical coordinates of a location, Maps help plotting these coordinates in a much more comprehensible visual medium. Though end users typically leverage Location services and Maps to navigate to a desired location, enterprises have figured out their usage in various other scenarios. For example, the collective power of Maps and Location services is being used by enterprises to plot their customers' demographic patterns based on their locations to improve their customer relationship management services; or locate their field service agents, and guide them to their prospective service areas; or track their fleet, and instruct them in real time – just to name a few.

Android apps leverage the Google Maps Android API to incorporate Maps and related functionalities. Using this API, we can include maps in our apps, mark places, and draw routes on it. Similar to Location services, Maps and related functionalities are also accessed using Google Play services in Android apps.

### 9.4.1 Setting Up the App

To build apps with Maps and related capabilities, we have to first ensure that the Google Maps Android API is available in the app development environment. This can be done by installing the Google Play services

[2] http://developer.android.com/training/location/index.html

SDK (refer Section 9.2). Once this is done, we have to obtain a map key from Google APIs console,[3] so that the app can access Google Maps servers through the Google Maps Android API.

In order to get a map key, we need to create a project in Google APIs console (requires a Google account). Once the project is created, we need to enable the *Google Maps Android API* (present in the *APIs* subsection of *APIs & auth* section) in it. Following this, we need to create a new Android key by providing, *package name*, and *Secure Hash Algorithm 1* (SHA1) fingerprint of our app in the *Credentials* sub-section of *APIs & auth* section.

The SHA1 fingerprint is a 40-digit hexadecimal number that represents the shorthand for RSA key required to authenticate our app, before publishing it on an app store. It is generated using the keytool utility [a Java Development Kit (JDK) tool], and get stored inside a keystore – a database to store cryptographic keys and digital certificates, as shown here.

```
keytool -genkey -v -keystore <<fully_qualified_filename>>.keystore -
alias <<aliasname>> -keyalg RSA -keysize 2048 -validity 10000
```

To list the SHA1 fingerprint, we need to execute the following command:

```
keytool -list -v -keystore <<fully_qualified_filename>>.keystore -
alias <<aliasname>> -storepass <<password>> -keypass <<password>>
```

The outcome of the app registration is a map key displayed on Google APIs console. Once we get the map key, we have to add it to the app manifest file, as shown in Snippet 9.6.

```
<application>
...
<meta-data android:name="com.google.android.maps.v2.API_KEY"
   android:value="<<Obtained map key>>"/>
...
</application>
```

**Snippet 9.6** | Adding map key to AndroidManifest.xml.

The app also needs to request the `com.google.android.providers.gsf.permission.READ_GSERVICES`, `ACCESS_NETWORK_STATE`, `INTERNET`, and `WRITE_EXTERNAL_STORAGE` permissions. The `READ_GSERVICES` permission is required to access Google Maps servers. The `ACCESS_NETWORK_STATE` permission is required to determine the Internet connection status; the `INTERNET` permission is required to download maps from Google Maps servers; and the `WRITE_EXTERNAL_STORAGE` permission is required to persist map data on the device's external storage [Secure Digital (SD) card].

For more information on getting a map key and other related information, readers are encouraged to refer to the Google Developers' resources.[4]

## 9.4.2 Adding Maps

Once the app is set up, maps can be added to it. This can be achieved by either using a `MapFragment` – a specialized Fragment designed to display maps – or including a `MapView` – a UI element (View) in the layout file of an Activity.

Let us now modify the LocateMe app to display a map with the current location at its center on tap of the *View Map* Button (see Fig. 9.2), as shown in Fig. 9.3.

[3] https://code.google.com/apis/console

[4] https://developers.google.com/maps/documentation/android/start

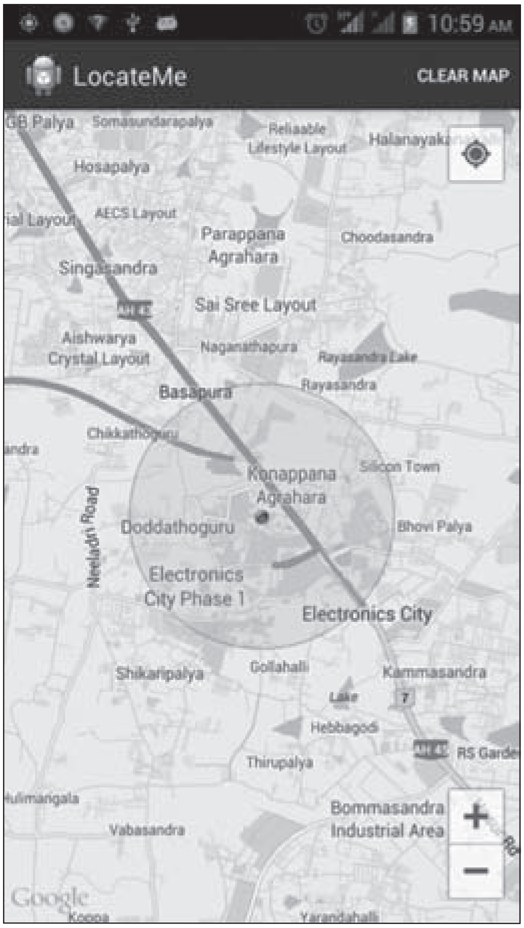


**Figure 9.3** | Map added in LocateMe app.

Snippet 9-7 shows the `<fragment>` tag used in the layout file of the Activity to display the map. The `class` attribute of the `<fragment>` tag refers to the `MapFragment` class which will display a map in the Activity upon inflation (Line 5).

```
1  <fragment
2   android:id="@+id/mapLayout"
3   android:layout_width="match_parent"
4   android:layout_height="match_parent"
5   class="com.google.android.gms.maps.MapFragment" />
```

**Snippet 9.7** | Adding `MapFragment` to layout file.

Snippet 9.8 describes the `initializeMap()`, `onStart()`, `onStop()` and `onConnected()` methods of the Activity. In the `initializeMap()` method, we obtain the `MapFragment` included in the layout file and assign it to `myMapFragment` – a `MapFragment` instance (Line 3). In Line 4, the displayed map is fetched into an instance of `GoogleMap` – `map` – by using the `getMap()` method of the `MapFragment` class. In Line 7, the `moveCamera()` method of `GoogleMap` is used to set the center of the map as the current location coordinates of the user with a zoom level (13 here). The `setMyLocationEnabled()` method of `map` is used to enable the *My Location* Button displayed on the map. When the user taps on this button, the map aligns its center to the location of the user.

```
1  private void initializeMap() {
2    coordinates = new LatLng(currentLocation.getLatitude(),
      currentLocation.getLongitude());
3     myMapFragment=(MapFragment)getFragmentManager().
      findFragmentById(R.id.mapLayout);
4     map = myMapFragment.getMap();
5     if (map != null) {
6       map.setMyLocationEnabled(true);
7       map.moveCamera(CameraUpdateFactory.newLatLngZoom(new
       LatLng(coordinates.latitude, coordinates.longitude), 13));
8     }
9  }@Override
10 protected void onStart() {
11   super.onStart();
12   mClient.connect();
13 }
14 @Override
15 public void onStop() {
16   mClient.disconnect();
17   super.onStop();
18 }
19 @Override
20 public void onConnected(Bundle arg0) {
21       initializeMap();
22 }
```

**Snippet 9.8** | Configuring the map in Activity.

Because Maps work in conjunction with Location services, the `initializeMap()` method is invoked in the `onConnected()` method of the `ConnectionCallbacks` (Line 21). The map added in this example is described as normal-type map. We can also change the type of map to terrain type or satellite type by invoking the `setMapType()` method of the `GoogleMap` object.

### 9.4.3 Making Maps Interactive

Once a map is added to an app, we can further use the Google Maps Android API to customize the user interactions with the map. These customizations can be in terms of dealing with the default UI controls that come with the map such as zoom controls, my location button, and compass; or might be tackling with various default map gestures such as zoom, scroll, rotate, or tilt gestures; or even reacting to the map events such as click or long click.

In addition to this, the Google Maps Android API also provides a mechanism to draw on a map. We can draw markers to pinpoint a location on the map, draw directions between two locations, or even draw shapes to mark an area on the map.

Let us now further modify the LocateMe app so that it can respond to a long-click map event based on which it either adds a marker on the selected location (see Fig. 9.4) or plots direction to this point from the user's current location.

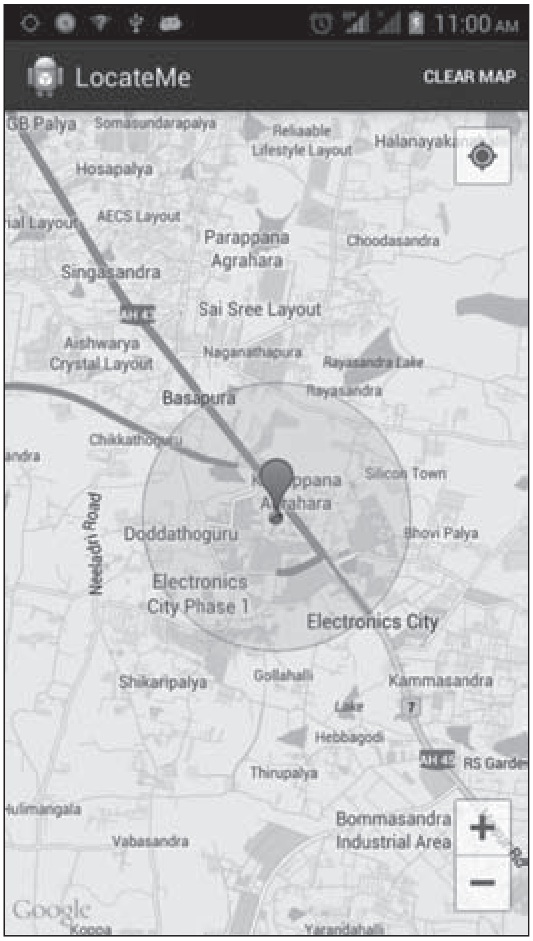


**Figure 9.4** | Marker added to map in LocateMe app.

In Line 3 of Snippet 9.9, a long-click listener is added to the map. When the user long clicks on the map, the app presents an AlertDialog to either add a marker or get direction to the selected point (Lines 5 and 7). To add a marker on the map at the selected location, the `addMarker()` method of the `GoogleMap` object is used (Line 18). The marker is customized to add the current position (Line 14), to disable the marker movement (Line 15), to add an info window with title (Line 16), and to add a description in the info window (Line 17).

```
1 private void activateLongClickOnMap() {
2   map.setOnMapLongClickListener(new OnMapLongClickListener() {
3   @Override
4   public void onMapLongClick(final LatLng arg0) {
5    View dialogLayout= getLayoutInflater().
     inflate(R.layout.dialog_options, null);
6    AlertDialog.Builder builder=new Builder(LocateMeMap.this);
7    builder.setView(dialogLayout);
8    Button addMarker = (Button)dialogLayout.
     findViewById(R.id.addMarker);
9     Button getDirections = (Button)dialogLayout.
      findViewById(R.id.getDirections);
10    addMarker.setOnClickListener(new OnClickListener() {
11    @Override
12    public void onClick(View v) {
13      MarkerOptions markerOptions = new MarkerOptions();
14      markerOptions.position(arg0);
15      markerOptions.draggable(false);
16      markerOptions.title(“Custom location”);
17      markerOptions.snippet(“Latitude”+arg0.latitude+
        “Longitude”+ arg0.longitude);
18      map.addMarker(markerOptions);
19      alertDialog.dismiss();
20   }
21   });
22   getDirections.setOnClickListener(new OnClickListener() {
23   public void onClick(View v) {
24       new GetDirectionsTask().execute(new String[] {arg0.latitude
         +““,arg0.longitude+”“,currentLocation.getLatitude()+”“,curr
         entLocation.getLongitude() + ”” });
25       alertDialog.dismiss();
26    }
27   });
28   alertDialog = builder.create();
29   alertDialog.show();
30  }
31  });
32 }
```

**Snippet 9.9** | Interacting with map in LocateMe app.

However, to get direction to the selected point, the `GetDirectionsTask` – a user-defined AsyncTask – is invoked (Line 24) to fetch directions from the Google Directions API.[5] Once the response is obtained, it is parsed using another user-defined class – `DirectionParser` – to obtain the points along a route to the selected destination from the current location.[6] Once the points are obtained, the `GetDirectionsTask` invokes the `drawDirections()` – a user-defined method to present the directions to the user by plotting these points on the map – as depicted in Snippet 9.10. The points are connected by adding polylines to the `GoogleMap` object, as shown in Line 7. The `PolylineOptions` instance is used to define the width, color, and the points of the polyline (Lines 2–6).

```
1  private void drawDirections(List<LatLng> points) {
2    PolylineOptions options=new PolylineOptions().width(5).
     color(Color.BLACK);
3    for(LatLng point: points)
4    {
```

[5] https://developers.google.com/maps/documentation/directions/

[6] The details of obtaining and parsing the response are omitted in the text. Readers are encouraged to explore it using the resources provided along with the book.

```
        options.add(point);
    }
    map.addPolyline(options);
}
```

**Snippet 9.10** | `drawDirection()`, a user-defined method.

We have just explored how drawing capabilities (markers and directions) and customized interaction capabilities (map events) of the Google Maps Android API are used to implement a useful scenario in apps that leverage Maps. In addition to this, the `UiSettings` class of the Google Maps Android API can be used to toggle the behavior of map UI controls. For example, zoom controls or compass can be disabled or enabled on the map. It can also be used to disable default map gestures such as scroll, tilt or rotate, in order to retain the state of the map. Various permutations and combinations of drawing and customized interaction features of the Google Maps Android API yield in building sophisticated Map-based apps.

## SUMMARY

This chapter has introduced the role of Google Play services while building apps with Location services and Google Maps. It has brought out the importance of necessary components involved in orchestrating Google Play services, and dealt with the setup required for implementing their features in an app.

The chapter has introduced the Location services with demonstrations on locating a user using latitude and longitude coordinates, tracking user's location based on the change in location coordinates, and retrieving the address of user's location from the location coordinates. It has also discussed other capabilities of Location services, and scenarios where they can be leveraged.

The chapter has also explored the inclusion of Google Maps in an app, along with nuances of customizing the interaction and drawing on maps. It has explored features such as drawing markers and directions along with customization of map events.

## REVIEW QUESTIONS

1. Describe Google Play services and its significance in Android app development.
2. Outline the features of Location services.
3. Define geocoding and reverse geocoding.
4. Explain the process to set up an app to use Google Maps in an Android app.
5. Illustrate various drawing and user interaction customization capabilities on maps.

## FURTHER READINGS

1. Set up Google Play services SDK: http://developer.android.com/google/play-services/setup.html
2. Making your app location-aware: http://developer.android.com/training/location/index.html
3. Google Maps: https://developers.google.com/maps/documentation/android/start
4. The Google Directions API: https://developers.google.com/maps/documentation/directions/

# 10 Sensors

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Identify sensors available in a smart device.
- Classify sensors in Android.
- Monitor different sensors.
- Leverage motion, position, and environment sensors.
- Implement sensor framework capabilities of Android.

## 10.1 INTRODUCTION

Keeping a track of your heart beat while jogging, or pointing the phone camera toward the night sky to know the name of the constellation in focus, or tilting the device to control the car wheel in a racing game, or checking the current altitude while hiking, all these suggest that most likely the mobile apps here are using sensors to provide you with a seventh sense.

It is surely not an exaggeration that sensors are the ones that make a device smarter. Sensors are those components of a device that are capable of detecting and measuring changes in and around the device. Global Positioning System (GPS), gyroscope, proximity sensor, Near Field Communication (NFC), gravity sensor, and electronic compass are some of the most common smartphone sensors.

In this chapter, we are going to look at the various sensors supported by the Android platform, and explore the Android sensor framework that lets us access these sensors. We are also going to apply this learning to build a few simple apps that leverage the power of these sensors.

## 10.2 SENSORS IN ANDROID

The Android platform classifies sensors into motion, position, and environment sensor categories, as depicted in Fig. 10.1. Motion sensors detect device movement by measuring acceleration and rotational forces along the three dimensions. Position sensors measure the physical position of the device, whereas environment sensors detect changes in the environmental parameters such as temperature, humidity, and pressure.

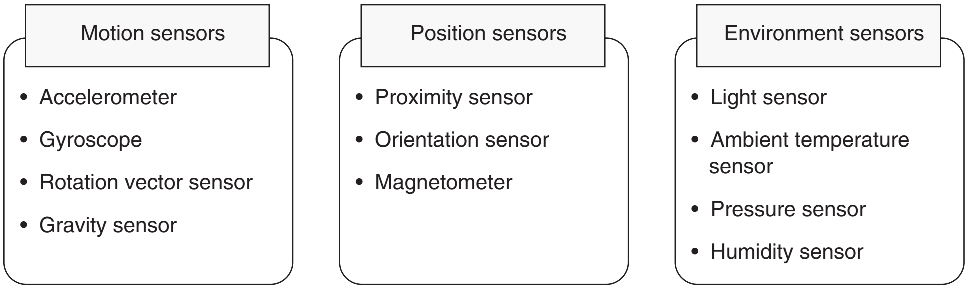


**Figure 10.1** | Classification of sensors.

Sensors such as gyroscope and temperature have a physical component plugged in the device circuitry, and are hence known as hardware sensors. On the other hand, there are software sensors such as gravity and orientation that in turn rely on hardware sensors for raw data, and process this data to provide some unique sensory information.

Let us briefly discuss common device sensors:

1 **Accelerometer:** An accelerometer is a motion sensor that is typically used to measure acceleration in any of the three dimensions. It can be used to detect events such as shake and move.
2 **Gyroscope:** While an accelerometer measures linear acceleration, a gyroscope that is also a motion sensor can measure angular acceleration. It can be used to detect events such as tilt and rotate. Therefore, it is more useful for gaming apps where there is a need for high-precision tracking of rotation, roll, or yaw.

3 **Proximity sensor:** A proximity sensor is a position sensor that is used to determine how close the device is to user's face, and is most commonly used to turn off the screen when user answers a call. This not only ensures longer battery life but also prevents any accidental inputs.
4 **Magnetometer:** A magnetometer is also a position sensor that can detect device orientation with respect to the earth's magnetic field, or in simple terms, it can detect which direction is north.
5 **Light sensor:** A light sensor is an environment sensor that is used to provide appropriate screen brightness by optimizing it to the external environment. This not only saves power but also provides a soothing experience to the user's eye.

Ambient temperature and humidity sensors are environment sensors that can report heat and humidity, respectively, at user's location.

## 10.3 ANDROID SENSOR FRAMEWORK

As smart devices are outsmarting each other day after day, it seems that more and more sensors are going to be packaged with smart devices in the times to come. But currently, not all smart devices have all the sensors, therefore it is imperative for an app developer to first confirm the presence of a desired sensor before using it.

Sensor capabilities are made available to a developer by the Android sensor framework. Let us now explore how a developer can identify the sensors present in a device and put them to use in an app.

### 10.3.1 Identifying Sensors

The first step in using sensors is to identify whether a particular sensor is available or not. This is achieved by using the `SENSOR_SERVICE`, as shown in Line 15, Snippet 10.1. The `getSensorList()` method (Line 16) of the `SensorManager` class is used to retrieve the list of sensors in a device. It takes a constant as a parameter to determine the scope of the retrieved list.

```
1  public class MainActivity extends Activity implements
   OnClickListener {
2    private static final String LOG_TAG = "list_all_sensors";
3    private Button button;
4    private SensorManager sensorManager;
5    @Override
6    protected void onCreate(Bundle savedInstanceState) {
7      super.onCreate(savedInstanceState);
8      setContentView(R.layout.activity_main);
9      button = (Button) findViewById(R.id.button1);
10     button.setOnClickListener(this);
11   }
12   @Override
13   public void onClick(View arg0) {
14     Toast.makeText(this, "Discovering device sensors..",
       Toast.LENGTH_SHORT).show();
15     sensorManager = (SensorManager)
       getSystemService(Context.SENSOR_SERVICE);
16     List<Sensor> sensorList =
       sensorManager.getSensorList(Sensor.TYPE_ALL);
17    for(Sensor sensor : sensorList){
18      Log.i(LOG_TAG, “Sensor Name: “+sensor.getName()+”| Sensor
         Type: ”+sensor.getType());
19    }
20    Toast.makeText(this, “Finished searching. Check logs for
      details.”, Toast.LENGTH_LONG).show();
21   }
22  }
```

**Snippet 10.1** | Identifying sensors available in a device.

The `TYPE_ALL` constant (Line 16) is used to get the entire list of sensors available in the device. If we want to get a specific sensor(s), we can use constants such as `TYPE_ACCELEROMETER`, `TYPE_PROXIMITY`, or `TYPE_LIGHT`.

This list helps in enabling or disabling app features that rely on sensors. There might be a possibility that a device may have multiple sensors to support a specific feature. In such cases, retrieving this list also helps in narrowing down the choices, but the onus always remains on the developer to choose the appropriate sensor.

Besides discovering the available list of sensors, the `SensorManager` class also allows us to determine the capabilities and requirements of each sensor. For example, the `getPower()` method can be used to determine the power requirements of a sensor.

### 10.3.2 Monitoring Sensors

Once availability of the required sensor is determined, the next step is to enable the sensor and start monitoring it. This is achieved by registering the sensor using the `SensorManager` API, and implementing the `SensorEventListener` interface.

Snippet 10.2 enlists a code for monitoring the accelerometer sensor and displaying linear acceleration along all the three coordinate axes.

```
1  public class MainActivity extends Activity implements
   SensorEventListener, OnClickListener {
2
3    private SensorManager sensorManager;
4    private Sensor accl;
5    private TextView xValTV, yValTV, zValTV;
6    private boolean monitoring = false;
7    private Button button;
8
9    @Override
10   protected void onCreate(Bundle savedInstanceState) {
11     super.onCreate(savedInstanceState);
12     setContentView(R.layout.activity_main);
13     // Assuming we have already confirmed that accelerometer is
       available
14     sensorManager = (SensorManager)
       getSystemService(Context.SENSOR_SERVICE);
15     accl=sensorManager.getDefaultSensor
       (Sensor.TYPE_ACCELEROMETER );
```

```
  xValTV = (TextView) findViewById(R.id.xVal);
  yValTV = (TextView) findViewById(R.id.yVal);
  zValTV = (TextView) findViewById(R.id.zVal);
  button = (Button) findViewById(R.id.button1);
  button.setOnClickListener(this);
}

@Override
public final void onAccuracyChanged(Sensor sensor, int
accuracy) {
    // Do something on sensor accuracy change.
}

@Override
public final void onSensorChanged(SensorEvent event) {
   if (monitoring) {
     float[] values = event.values;
     // Movement
     float acclX = values[0];
     float acclY = values[1];
     float acclZ = values[2];
     xValTV.setText(String.valueOf(acclX));
     yValTV.setText(String.valueOf(acclY));
     zValTV.setText(String.valueOf(acclZ));
   }
}

@Override
protected void onResume() {
   super.onResume();
   sensorManager.registerListener(this, accl,
        SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
   super.onPause();
   sensorManager.unregisterListener(this);
}

@Override
public void onClick(View arg0) {
   Toast.makeText(this, "Displaying Accelerometer data..",
   Toast.LENGTH_SHORT).show();
   monitoring = true;
}
}
```

**Snippet 10.2** | Monitoring sensors (accelerometer).

The accelerometer is registered in the app using the `registerListener()` method of the `SensorManager` (Line 45). This method accepts an implementation of the `SensorEventListener` interface. This interface declares two methods: `onAccuracyChanged()` and `onSensorChanged()`. The `onAccuracyChanged()` method gets invoked whenever the sensor in use reports a change in the accuracy value. But most of the time we would be making use of the `onSensorChanged()` method that keeps receiving the sensor data from the sensor. The frequency of invocation of the `onSensorChanged()` method is determined using the last argument supplied to the `registerListener()` method. In our snippet, we have requested for a normal frequency of sensor data updates.

Sensor data is passed to the app by encapsulating it in the `SensorEvent` class (Line 29). Depending on the sensor, the type and amount of data may vary. For example, while an accelerometer provides three discrete values of linear acceleration corresponding to the three axes in $m/s^2$, a light sensor provides only one value that corresponds to the ambient light intensity in lux unit. These values can be retrieved by accessing the `values` member of the `SensorEvent` object (Line 31). Once the sensor values are retrieved, it is up to the developer to put them to use in whichever way the app requires. Snippet 10.2 simply updates three TextViews with current linear acceleration values. One thing to keep in mind when implementing the `onSensorChanged()` method is to keep it light. This method is called very frequently, and a heavy implementation might have a negative impact on the app performance.

Line 51 lists deregistering the sensor in the `onPause()` method of the Activity. This is a best practice, because failing to deregister a sensor, as soon as it is not required, can lead to unwarranted battery drain. Android runtime delegates the responsibility of disabling sensors to the app, and does not disable them when the screen is turned off.

Now that we have figured out how to determine and monitor a sensor, let us look at each of the three sensor categories in detail.

## 10.4 MOTION SENSORS

Sensors that capture movement-related information are categorized as motion sensors. Following are some of the motion sensors available in Android phones: accelerometer, gyroscope, rotation vector sensor, and gravity sensor. While accelerometer and gyroscope are hardware sensors, the latter two can be implemented as either hardware or software sensors. These two sensors combine data from accelerometer/gyroscope and magnetometer to provide information that is more meaningful to an app, such as the orientation or rotation of a device.

In this section, let us explore the use of accelerometer using a simple app – RingSilencer. This app mutes the ringtone if the phone is flipped while it is ringing. This app uses a Broadcast Receiver that listens to the phone states and monitors the phone movement using the accelerometer, as shown in Snippet 10.3.

```
public class Silencer extends BroadcastReceiver implements
SensorEventListener {
  private SensorManager sm;
  private static final String Mode_Number="RingSilencerPrefs";
  private AudioManager aman;
  private static float prevZValue;
  private static boolean phoneFlipped = false;

  @Override
  public void onReceive(Context context, Intent intent) {
     aman=(AudioManager)context.getSystemService
     (Context.AUDIO_SERVICE);
     sm=(SensorManager)context.getSystemService
     (Context.SENSOR_SERVICE);
     if(intent.getAction().equals
     ("android.intent.action.PHONE_STATE")){
      if(intent.getStringExtra(TelephonyManager.EXTRA_STATE).
      toString().equals(TelephonyManager.EXTRA_STATE_RINGING)) {
        int prevMode;
        SharedPreferences modeSettings=
        context.getSharedPreferences(Mode_Number, 0);
        Editor editor = modeSettings.edit();
        prevMode = aman.getRingerMode();
        editor.putInt("silentMode", prevMode);
        editor.commit();
        startListening();
      }
     else
     if(intent.getStringExtra(TelephonyManager.EXTRA_STATE)
     .toString().equals(TelephonyManager.EXTRA_STATE_OFFHOOK)) {
        sm.unregisterListener(this);
    }
     else
     if(intent.getStringExtra(TelephonyManager.EXTRA_STATE).
     toString().equals(TelephonyManager.EXTRA_STATE_IDLE)) {
           SharedPreferences modeSettings=
           context.getSharedPreferences(Mode_Number, 0);
           int prevMode = modeSettings.getInt("silentMode",
           AudioManager.RINGER_MODE_VIBRATE);
           aman.setRingerMode(prevMode);
           sm.unregisterListener(this);
           Editor editMode = modeSettings.edit();
           editMode.clear();
           editMode.commit();
           phoneFlipped = false;
   }
  }
 }

 public void startListening() {
   Sensor accelSensor =
   sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
   sm.registerListener(this,
   accelSensor,SensorManager.SENSOR_DELAY_FASTEST);
 }

 public void onAccuracyChanged(Sensor sensor, int accuracy) {
 }

```

```
46  public void onSensorChanged(SensorEvent event) {
47   Sensor sensor = event.sensor;
48   if (sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
49    if (event.values[2] < 0 && prevZValue > 0) {
50     phoneFlipped = true;
51    }
52   }
53   prevZValue = event.values[2];
54   if (phoneFlipped == true) {
55    aman.setRingerMode(AudioManager.RINGER_MODE_SILENT);
56    sm.unregisterListener(this);
57   }
58  }
59  }
```

**Snippet 10.3** | RingSilencer app using accelerometer.

This snippet makes use of the `AudioManager` class (Line 4) to change the ringtone mode. The Broadcast Receiver detects the phone states and performs appropriate actions. If the state is ringing (Line 13), it retrieves the ringtone mode (Line 17) and persists it as a `SharedPreferences` (Lines 15–19) for later use. After persisting the ringtone mode, it starts monitoring the accelerometer sensor (Line 20) to find out whether the phone is flipped or not. By flipping we mean that the user rotates the phone, which was initially facing upward, to make it face downward.

If a phone is lying on a flat surface facing upward, the accelerometer records a positive acceleration in the *z*-direction due to the force of gravity. When flipped, this acceleration becomes negative. This shift from positive to negative acceleration along the *z*-axis forms the basis for verifying a flip action. The `onSensorChanged()` method implementation (Lines 46–57) observes change in acceleration along the *z*-axis by comparing the current and previous acceleration values (Line 49). If a shift is perceived in acceleration values, the ringtone mode is set to silent (Line 55) and the sensor is deregistered (Line 56).

The Broadcast Receiver gets invoked not only if a call is received but also if it is answered or when it is ended. If a user chooses to answer the call (Line 22), we simply proceed to deregister the sensor (Line 23). Once the phone state returns to idle (Line 25), either after the call is finished or if it is missed, we proceed to restore the ringtone mode to its earlier state using the value persisted in the `SharedPreferences` (Lines 26–32).

## 10.5 POSITION SENSORS

The magnetometer and proximity sensors are the most commonly used position sensors in Android devices. The magnetometer measures the magnetic field along the three axes. The proximity sensor reports the distance of nearby objects from the phone. This comes in handy, for example, to turn off the screen when the user picks a call and brings the phone near his or her face. Another position sensor – orientation sensor – provides information about the device position and its inclination in three-dimensional space. However, this sensor was deprecated in Android 2.2, and the recommended way to find device orientation is either to use rotation vector sensor, if available, or to combine data from magnetometer and accelerometer.

In this section, let us explore the use of proximity sensor using a simple app – BrightnessManager. This app regulates the screen brightness, as shown in Snippet 10.4.

```
public class BrightnessManagerActivity extends Activity
implements OnCheckedChangeListener, SensorEventListener
{
 private CheckBox enableCB;
 private SensorManager sensorManager;
 private Sensor proximitySensor;
 public enum BRIGHTNESS_LEVEL {
  LOW, MEDIUM, HIGH;
 }
 public BRIGHTNESS_LEVEL currentBrightness=BRIGHTNESS_LEVEL.LOW;

 @Override
 protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   enableCB = (CheckBox) findViewById(R.id.checkBox1);
   enableCB.setOnCheckedChangeListener(this);
 }

 @Override
 protected void onResume() {
   super.onResume();
   if (enableCB.isChecked()) {
     initiateBrightnessManager();
   }
  }

 @Override
 protected void onPause() {
   super.onPause();
   terminateBrightnessManager();
  }

 @Override
 public void onCheckedChanged(CompoundButton buttonView,
 boolean isChecked) {
  if (isChecked) {
    log("Initiating Brightness Manager");
    initiateBrightnessManager();
  } else {
    terminateBrightnessManager();
   }
 }

 private void terminateBrightnessManager() {
  if (sensorManager != null) {
   log("Deregistering proximity sensor listener.");
   sensorManager.unregisterListener(this);
47    }
48   }
49
50  private void initiateBrightnessManager() {
51    sensorManager=(SensorManager)getSystemService
      (Context.SENSOR_SERVICE);
52    proximitySensor=sensorManager.getDefaultSensor
      (Sensor.TYPE_PROXIMITY);
53    if (proximitySensor != null) {
54     sensorManager.registerListener(this, proximitySensor,
       SensorManager.SENSOR_DELAY_NORMAL);
55    }
56   }
57
58  private static final String LOG_TAG = “BrightnessManager”;
59  public void log(String msg) {
60   Log.i(LOG_TAG, msg);
61   }
62
63  @Override
64  public void onAccuracyChanged(Sensor arg0, int arg1) {
65
66  }
67
68  @Override
69  public void onSensorChanged(SensorEvent event) {
70       float distance = event.values[0];
71       log(“Distance = ” + distance);
72       if (distance == 0) {
73       int nextBrigtnessLevelNumber=
         (currentBrightness.ordinal()+1)%3;
74       currentBrightness=BRIGHTNESS_LEVEL.values()
         [nextBrigtnes sLevelNumber];
75       WindowManager.LayoutParams params =
         getWindow().getAttributes();
76       params.screenBrightness = nextBrigtnessLevelNumber-1;
77       getWindow().setAttributes(params);
78     }
79     }
80  }
```

**Snippet 10.4** | BrightnessManager app using proximity sensor.

The layout of the `BrightnessManagerActivity` has a CheckBox (Lines 15 and 16) that activates/deactivates the brightness modulation functionality (Lines 34–41). The proximity sensor provides the distance value in centimeters. In some devices, this value is just indicative of the fact whether or not there is any object in the near vicinity of the device. For example, such devices may report just two possible values – say 0 cm or 5 cm – depending on whether there is something near the sensor or not, respectively. But all devices report distance value as 0 cm whenever there is something near the sensor. This fact is being leveraged in Line 72.

This app defines three brightness levels: `LOW`, `MEDIUM`, and `HIGH` (Lines 6–8). These levels are used in cyclic order to set the screen brightness level (Lines 73 and 74). Every time the user taps on the proximity sensor, the brightness is changed to the next level. The `WindowManager` API provides the `screenBrightness` parameter using which the brightness is modulated (Lines 76–77).

## 10.6 ENVIRONMENT SENSORS

Environment sensors give information about the surroundings in which the device is placed. Android supports temperature, light, pressure, and humidity sensors. The light sensor is the most commonly available among the lot. It helps detect the lighting conditions around a phone and automatically adjusts the screen brightness. Other sensors are a bit rare; currently only the high-end phones sport them. But with time, these sensors too might become mainstream.

In this section, we will try to discover all the environment sensors available in a device using the EnvironmentSensors app. This app (Snippet 10.5) first ascertains the environment sensors available on the device, and then proceeds to monitor those which are available.

```
public class EnvironmentSensorsActivity extends Activity
implements
SensorEventListener {

  private Sensor ambientTemperatureSensor;
  private Sensor lightSensor;
  private Sensor pressureSensor;
  private Sensor relativeHumiditySensor;
  private Sensor temperatureSensor;

  private SensorManager sensorManager;

  private TextView atTV;
  private TextView lightTV;
  private TextView pressureTV;
  private TextView rhTV;
  private TextView temperatureTV;

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

    // Setting up UI references
    atTV = (TextView) findViewById(R.id.at);
    lightTV = (TextView) findViewById(R.id.light);
    pressureTV = (TextView) findViewById(R.id.pressure);
    rhTV = (TextView) findViewById(R.id.rh);
    temperatureTV = (TextView) findViewById(R.id.temperature);

    // Discover sensors
    sensorManager=(SensorManager)getSystemService
    (Context.SENSOR_SERVICE);
    ambientTemperatureSensor=sensorManager.getDefaultSensor
    (Sensor.TYPE_AMBIENT_TEMPERATURE);
    lightSensor=sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    pressureSensor=sensorManager.getDefaultSensor
    (Sensor.TYPE_PRESSURE);
   relativeHumiditySensor = sensorManager
              .getDefaultSensor(Sensor.TYPE_RELATIVE_HUMIDITY);
   temperatureSensor = sensorManager
              .getDefaultSensor(Sensor.TYPE_TEMPERATURE);
  }

  @Override
  protected void onResume() {
   super.onResume();
   monitorSensors(ambientTemperatureSensor,lightSensor,
   pressureSensor,relativeHumiditySensor, temperatureSensor);
  }

  @Override
  protected void onPause() {
   super.onPause();
   deregisterSensors(ambientTemperatureSensor, lightSensor,
    pressureSensor, relativeHumiditySensor, temperatureSensor);
  }

  private void monitorSensors(Sensor... sensors) {
   for (Sensor s : sensors) {
    if (s != null) {
     sensorManager.registerListener(this, s,
          SensorManager.SENSOR_DELAY_NORMAL);
    }
   }
  }

  private void deregisterSensors(Sensor... sensors) {
   for (Sensor s : sensors) {
    if (s != null) {
     sensorManager.unregisterListener(this, s);
    }
   }
  }

```

```
66   @Override
67   public void onAccuracyChanged(Sensor sensor, int accuracy) {
68
69   }
70
71   @Override
72   public void onSensorChanged(SensorEvent event) {
73    int sensorType = event.sensor.getType();
74    float sensorValue = event.values[0];
75    switch (sensorType) {
76     case Sensor.TYPE_AMBIENT_TEMPERATURE:
77      atTV.setText(String.valueOf(sensorValue));
78      break;
79     case Sensor.TYPE_LIGHT:
80      lightTV.setText(String.valueOf(sensorValue));
81      break;
82     case Sensor.TYPE_PRESSURE:
83      pressureTV.setText(String.valueOf(sensorValue));
84      break;
85     case Sensor.TYPE_RELATIVE_HUMIDITY:
86      rhTV.setText(String.valueOf(sensorValue));
87      break;
88     case Sensor.TYPE_TEMPERATURE:
89      temperatureTV.setText(String.valueOf(sensorValue));
90      break;
91    }
92   }
93  }
```

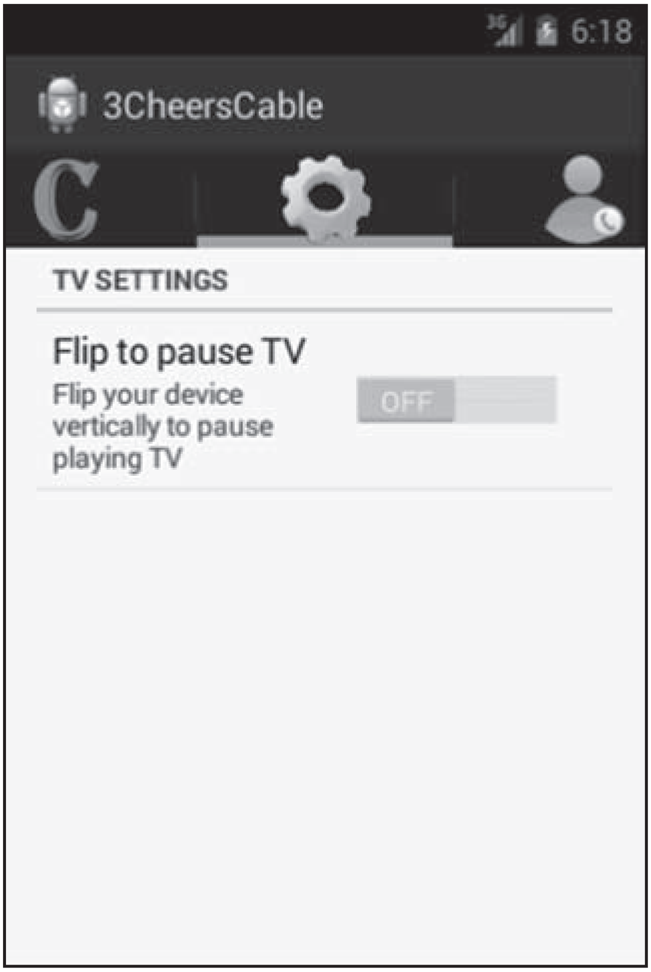
**Snippet 10.5** | EnvironmentSensors app.

The first step after the initial UI (User Interface) setup (Lines 23–27) is to find out the environment sensors available on the device. This, as you might recall from our discussion in Section 10.3.1, is done simply by performing a null check on the returned sensor. The `getDefaultSensor()` method of the `SensorManager` API returns `null` if a sensor is not available. Hence, a null check is made both at the time of registering and deregistering the sensors (Lines 52 and 60).

The current Activity, `EnvironmentSensorsActivity`, is the sensor listener for all the sensors. The `onSensorChanged()` determines the source sensor type by first obtaining the sensor from the `SensorEvent`, and then establishing its type (Line 73). On the basis of the type thus obtained, appropriate TextView is updated on the screen (Lines 75–91).

## 10.7 LET'S APPLY

In this section, let us leverage the device accelerometer capabilities in the 3CheersCable app, to include "pause live TV" feature that pauses streaming of video, when a user flips the device. This feature can be enabled or disabled by toggling the *Flip to pause TV* button in the user-defined `SettingsFragment`, as shown in Fig. 10.2.



TV SETTINGS

Flip to pause TV

Flip your device vertically to pause playing TV

**Figure 10.2** | Flip to pause TV button in `SettingsFragment`.

The `MediaFragment` class (defined earlier in Section 8.4) now needs to implement a `SensorEventListener` in order to listen to device sensor events, as shown in the following program:

```
public static class MediaFragment extends DialogFragment
implements SensorEventListener {
  String showName=null;
  SensorManager sensorManager;
  private boolean phoneFlipped;
  private static float prevZValue;
  VideoView videoView;
  static MediaFragment newInstance(String name) {
    MediaFragment f = new MediaFragment();
    Bundle args = new Bundle();
    args.putString("show_name", name);
```

```
         f.setArguments(args);
         return f;}
         @Override
         public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         showName = getArguments().getString("show_name");
         setStyle(DialogFragment.STYLE_NO_TITLE,android.R.style.
         Theme_Holo);
         //Obtaining the instance of shared preferences to check
         whether Flip to pause TV button is on or off
          SharedPreferences pref =
          PreferenceManager.getDefaultSharedPreferences(getActiv
          ity().getApplicationContext());
          if(pref.getBoolean("flip", true)){
            sensorManager = (SensorManager)
            getActivity().getSystemService(Context.SENSOR_SERVICE);
            sensorManager.registerListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
            ,SensorManager.SENSOR_DELAY_FASTEST);}}
```

Here we check the value of the *Flip to pause TV* button that is saved in the `SharedPreferences`. If the value obtained is `true`, we register for listening to changes in the values of the accelerometer.

The `onSensorChanged()` method implements the logic to determine the change in the values of the accelerometer to identify a flip and pause the video, as shown in the following program:

```
@Override
public void onSensorChanged(SensorEvent event) {
  Sensor sensor=event.sensor;
  if(sensor.getType() == Sensor.TYPE_ACCELEROMETER){
     if( event.values[2] < 0 && prevZValue > 0)
     phoneFlipped = true;}
  prevZValue = event.values[2];
  if(phoneFlipped == true ){
     videoView.pause();
     sensorManager.unregisterListener(this);}}
```

## SUMMARY

This chapter has introduced the sensors that may be compared to the sensory organs of a human being. Sensors make a device smarter, and in a way becomes our Seventh Sense via apps leveraging these sensors.

The chapter has introduced different types of sensors available in Android devices. It also delves into the Android sensor framework, enabling the developer to leverage the real power of smart devices. The first step to work with sensors is to make sure of their availability, and then start monitoring them to retrieve the continuous stream of sensory data that can be put to a specific use in a mobile app.

The chapter has further delved into individual categories of motion, position, and environment sensors available in Android. It has also explored sample apps of each category to provide the developer a hands-on feel of how to leverage various types of sensors in his or her app.

## REVIEW QUESTIONS

1. Define the categories of sensors available in Android devices.
2. Outline the generic steps involved in working with sensors in an app.
3. Explain the purpose of motion, position, and environment sensors with one real-life use case in each.
4. List down the best practices while using sensors in an app.

## FURTHER READINGS

1. Sensors overview: http://developer.android.com/guide/topics/sensors/sensors_overview.html
2. Motion sensors: http://developer.android.com/guide/topics/sensors/sensors_motion.html
3. Position sensors: http://developer.android.com/guide/topics/sensors/sensors_position.html
4. Environment sensors: http://developer.android.com/guide/topics/sensors/sensors_environment.html

# Part IV
# Moving To Market

# 11 Testing Android Apps

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Identify the app features that need to be tested.
- Apply Android JUnit framework for unit testing Android app components.
- Define the landscape of mobile app validation.

## 11.1 INTRODUCTION

So far, you have learnt how to orchestrate Android app components to create mobile apps. However, before we distribute them to the consumers, their conformance with the functional and nonfunctional requirements have to be validated. Besides validating the implemented use cases against an app's functional requirements, testing confirms the optimal usage of resources such as available memory, battery life, and processing power; support on diverse set of devices; performance on different carrier networks; security of data at rest or in motion; and usability of the app. Nonconformance of an app to these critical aspects may result in low retention of the app, or loss of revenue streams, or even loss of brand itself.

The testing of individual app components precedes the comprehensive validation of an app against its functional and nonfunctional requirements. The onus is on the developer to perform unit testing to test these individual components.

In this chapter, we are going to look at both the validation aspects – unit testing of individual components of an app and an overview of the comprehensive validation of its functional and nonfunctional requirements.

## 11.2 TESTING ANDROID APP COMPONENTS

Android components of an app must be individually tested to make sure that each component is functionally compliant to the requirements. This can be accomplished only with the intricate knowledge of the component's source code. Therefore, it is the responsibility of the developers to perform unit testing of these components to validate their requisite behavior and outcome.

Android JUnit framework is used to perform unit testing of individual Android app components. This framework is built on top of the JUnit framework – a very popular framework to unit test Java components.

To test individual unit(s), we need to create a *test project* akin to an Android app project. The requisite behavior and outcome to be tested are arranged as a collection of *test cases*, inside *test suites*, as depicted in Fig. 11.1. Each test suite is organized as test cases related to a specific Android app component, such as an Activity. All test suites in a test project belong to a *test package* that is similar to an

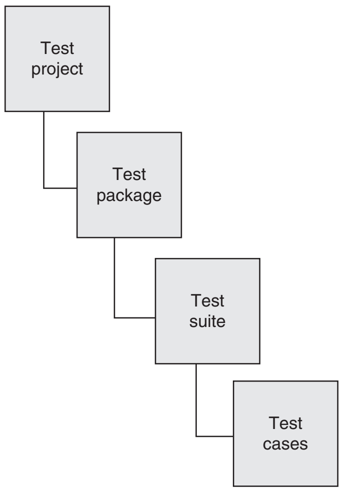


**Figure 11.1** | A test project hierarchy.

app package. A test package runs in the same process as that of the app (under test), thereby enabling the test cases to directly access different functionalities implemented in the app.

Test cases define the test criteria and mechanism using which a unit of code will be tested. The test criteria define the success/failure conditions for a specific test case. The test mechanism describes how a test has to be performed such as to click a specific Button or enter a specific value in an EditText control. JUnit and Instrumentation APIs are used to build the test cases. While JUnit APIs offer the default features to test a typical Java component, Instrumentation APIs bring in extended features to test Android-specific functionalities such as providing handles in a test case to refer to UI (User Interface) elements in an Activity, or specifying methods in a test case to directly invoke the callback methods of an Android app component, as and when required.

The `InstrumentationTestRunner` API (Application Programming Interface) is responsible for running these test cases in the same process as that of the app under test. This API can run only those test packages that are configured in an Android test project, and have a dependency established between the test and the app package. The test project's manifest file defines the required configuration, and Eclipse IDE's (Integrated Development Environment) built-in wizards help establish the required dependency.

Let us now delve into unit testing an Activity, a Service, and a ContentProvider using the Android JUnit framework.

## 11.2.1 Activity

In this section, let us explore unit testing of an Activity, using the Activity (Fig. 6.1) of FlatFile app, created earlier in Section 6.2 of Chapter 6. As mentioned before, to test Android app components, we need to start with creating a test project.A test project is created by selecting the Android Test Project option at

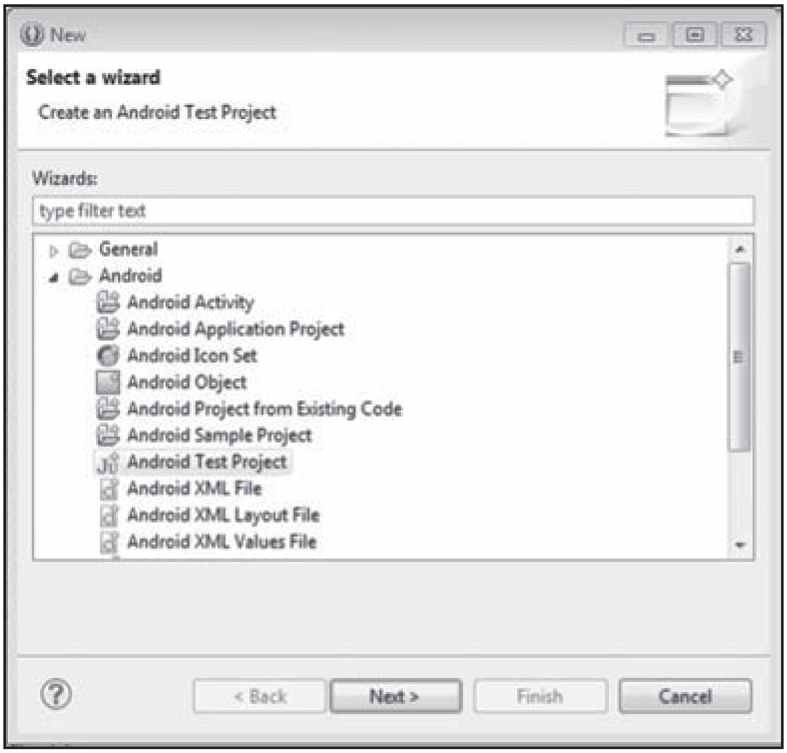
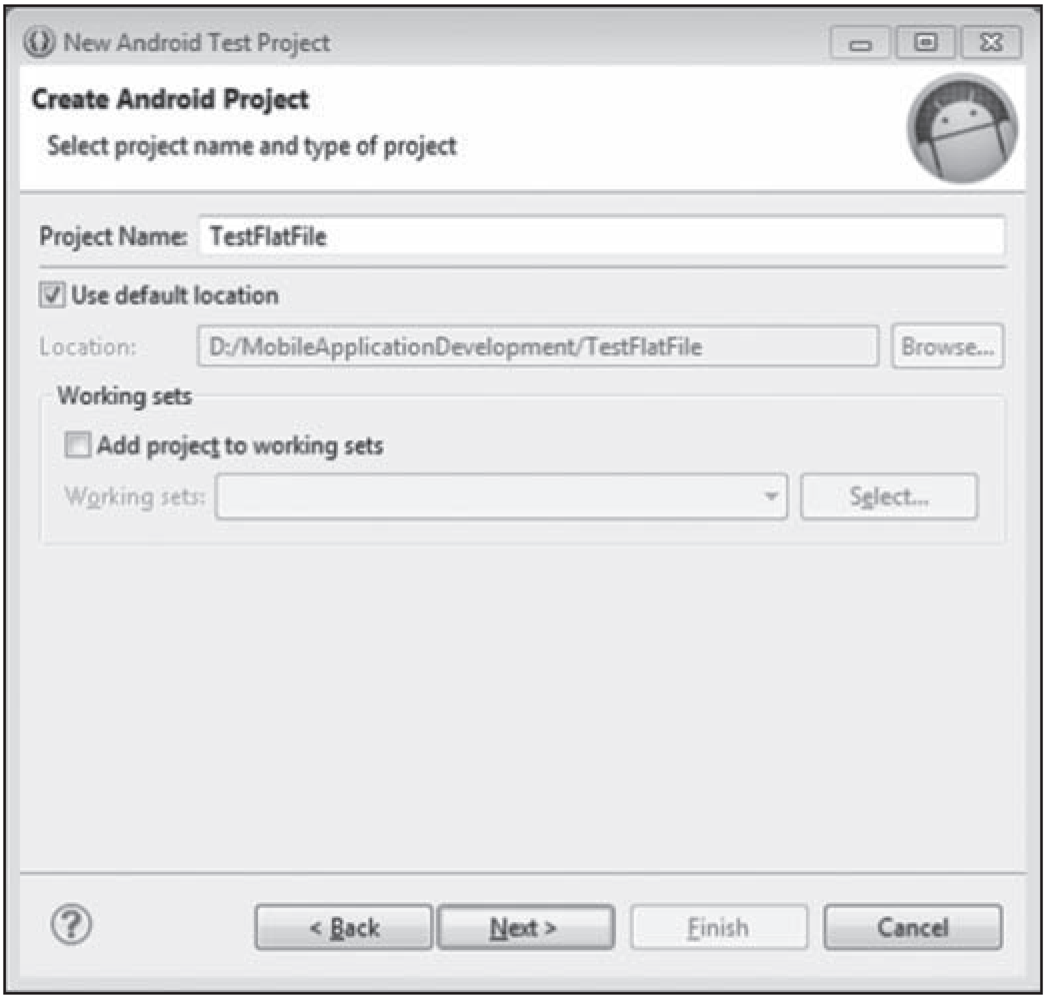


**Figure 11.2** | Creating Android Test Project.

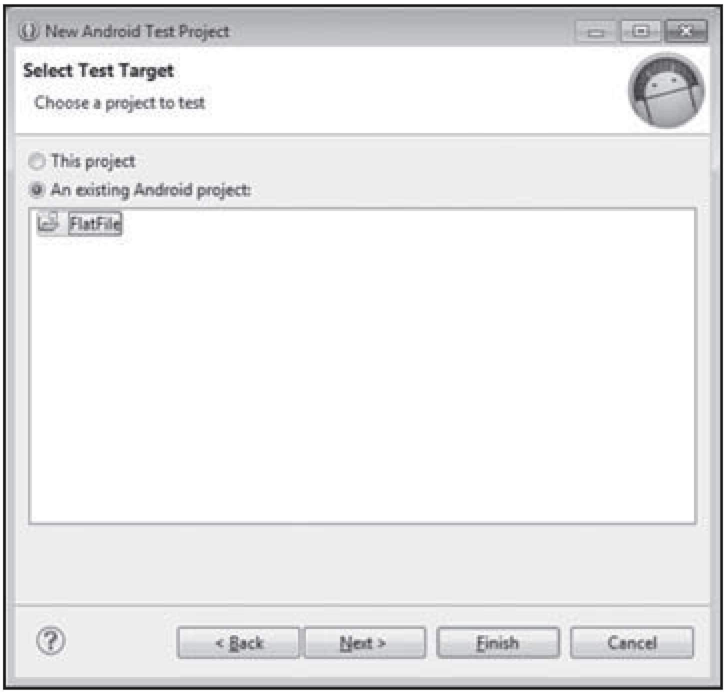
*File → New → Other → Android → Android Test Project*, as depicted in Fig. 11.2. Figure 11.3 depicts the New Android Test Project wizard that pops up on clicking *Next*. This wizard prompts us to provide the name of the test project. The convention of naming a test project is to prefix "Test" to the project name of the app under test. For example, here we have entered the name of the test project as TestFlatFile, as the name of the app project under test is FlatFile.



Project Name: TestFlatFile
[✓] Use default location
Location: D:/MobileApplicationDevelopment/TestFlatFile

**Figure 11.3** | New Android Test Project wizard.

Figure 11.4 depicts the next screen – Select Test Target – of New Android Test Project wizard, wherein we have to select the app project (FlatFile) that will be tested by the test project. This step establishes the dependencies between the test and the app packages. On clicking *Next,* Select Build Target screen of New Android Test Project wizard appears, as depicted in Fig. 11.5, where we need to select the build target for the test project. This is typically same as API level of app project under test.



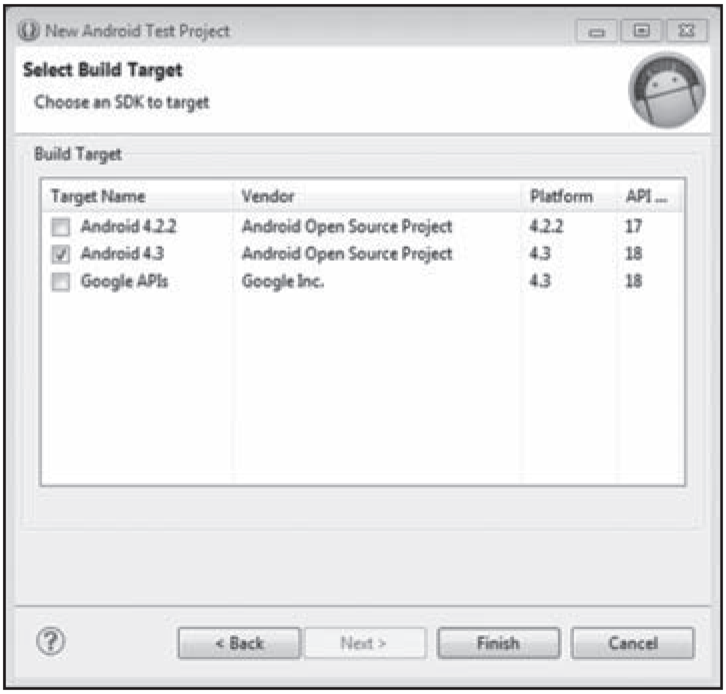
Select Test Target

Choose a project to test

( ) This project
(●) An existing Android project:

FlatFile

Figure 11.4 | Selecting Test Target.



Select Build Target

Choose an SDK to target

Build Target

| Target Name | Vendor | Platform | API ... |
|---|---|---|---|
| ☐ Android 4.2.2 | Android Open Source Project | 4.2.2 | 17 |
| ✓ Android 4.3 | Android Open Source Project | 4.3 | 18 |
| ☐ Google APIs | Google Inc. | 4.3 | 18 |

Figure 11.5 | Choosing Target SDK.

Finally, clicking *Finish* results in the creation of the required test project, as depicted in Fig. 11.6.

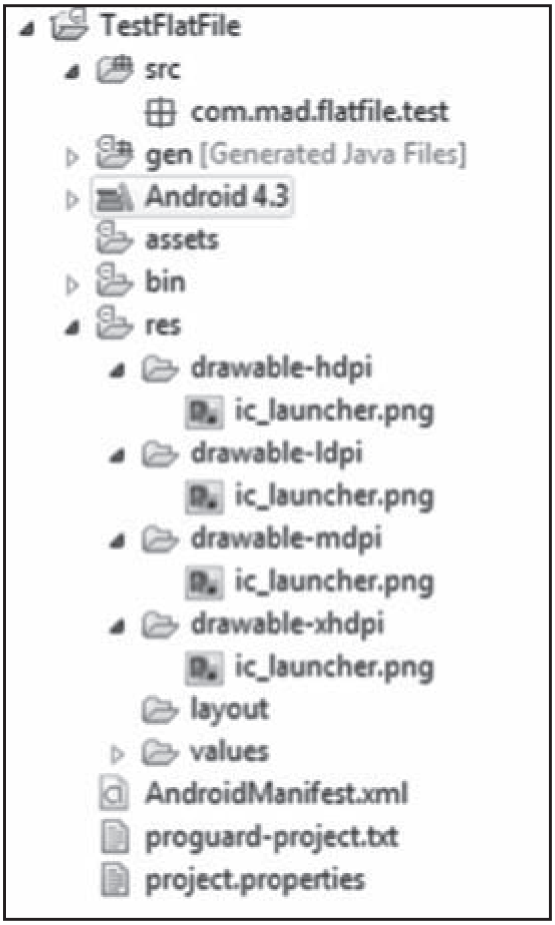


**Figure 11.6** | Android Test Project structure.

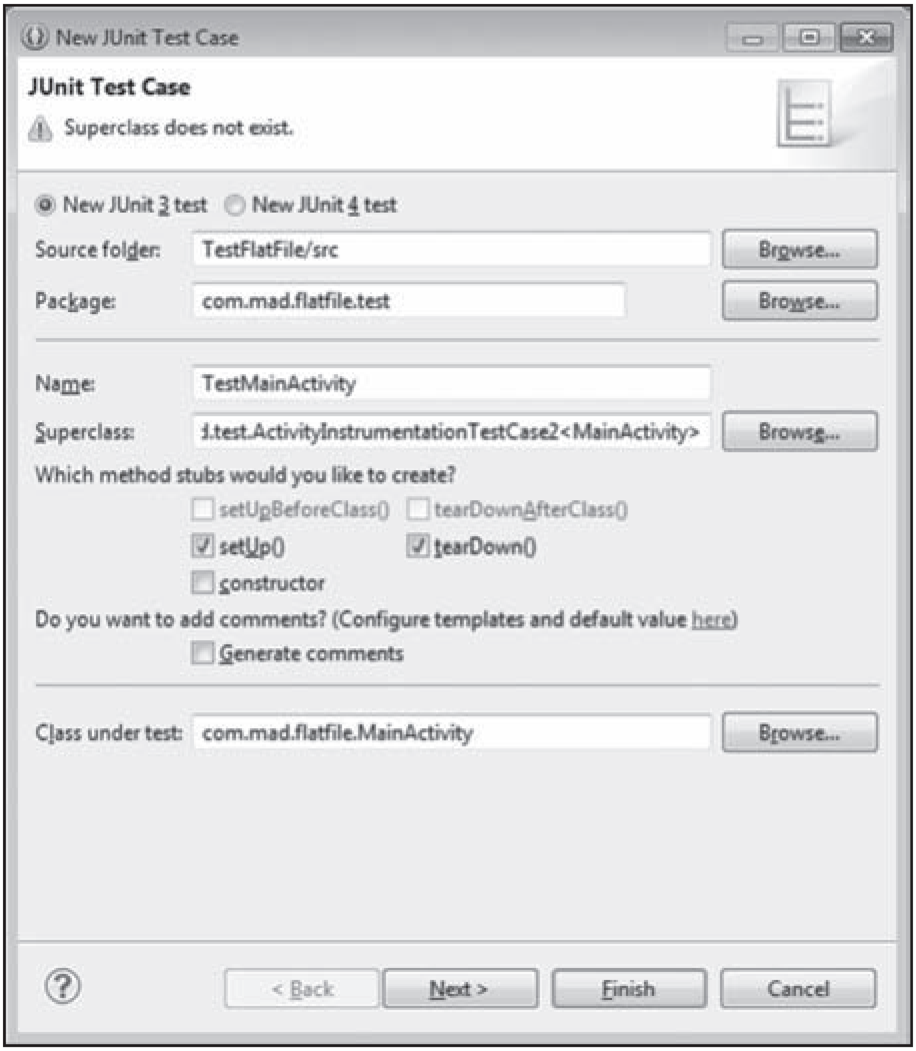
Although the test project structure resembles a regular Android app project structure, the contents of its autogenerated AndroidManifest.xml file differ significantly, as depicted in Snippet 11.1.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.mad.flatfile.test"
android:versionCode="1"
android:versionName="1.0" >
<uses-sdk android:minSdkVersion="8" />
<instrumentation
   android:name="android.test.InstrumentationTestRunner"
   android:targetPackage="com.mad.flatfile" />
<application
   android:icon="@drawable/ic_launcher"
   android:label="@string/app_name" >
```

```
10   <uses-library android:name="android.test.runner" />
11 </application>
12 </manifest>
```

**Snippet 11.1** | Test project manifest file.

The manifest file contains the `<instrumentation>` tag (Line 4), which indicates the app package to be tested using `android:targetPackage` attribute (Line 6). The `android:name` attribute is assigned the value `android.test.InstrumentationTestRunner` (Line 5) to indicate the `InstrumentationTestRunner` API that is responsible for running test cases. The `InstrumentationTestRunner` is part of the `android.test.runner` library. This library is included in the test project using the `<uses-library>` tag (Line 10).


New JUnit Test Case

**JUnit Test Case**

Superclass does not exist.

[✓ New JUnit 3 test] New JUnit 4 test

Source folder: TestFlatFile/src

Package: com.mad.flatfile.test

Name: TestMainActivity

Superclass: d.test.ActivityInstrumentationTestCase2<MainActivity>

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☑ setUp() ☑ tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: com.mad.flatfile.MainActivity

**Figure 11.7** | New JUnit Test Case wizard.

Once the test project is ready, we proceed to write the test cases to test various Android app components. To create a new test case, we can use the New JUnit Test Case wizard (as depicted in Fig. 11.7), obtained by right clicking on the test package `(com.mad.flatfile.test)`, and selecting *New → JUnit Test Case*. We need to provide the class name (`TestMainActivity` here), which will implement the test cases, in the *Name* field of this wizard. We need to specify the `ActivityInstrumentationTestCase2`, one of the pre-built test classes of the Instrumentation API to test the Activity, as the *Superclass* in this wizard. We also need to select *setUp()* and *tearDown()* checkboxes to ensure that these method stubs are included in the resulting test case class. The `setUp()` method is used to perform necessary preparations required to run the test, before every test case method is executed; the `tearDown()` method is used to windup after completion of each test case method execution. Towards the end, we also need to specify the *Class under test* in this wizard, pointing to the actual Activity in the app project that needs to be tested (`com.mad.flatfile.MainActivity` here).

Snippet 11.2 lists the resultant test case class with the `setUp()` and `tearDown()` method stubs included (Lines 5–10). The class constructor (Lines 2–4) is added later, inside which the constructor of the `ActivityInstrumentationTestCase2` class is invoked (Line 3) with the package and Activity class name of the app under test.

```
1  public class TestMainActivity extends
   ActivityInstrumentationTestCase2<MainActivity> {
2     public TestMainActivity(){
3       super(“com.mad.flatfile”, MainActivity.class);
4     }
5     protected void setUp() throws Exception {
6       super.setUp();
7     }
8     protected void tearDown() throws Exception {
9       super.tearDown();
10   }
11 }
```

**Snippet 11.2** | `TestMainActivity` class.

From here on, based on the Android app component under unit test, we need to design the relevant test case methods in the test case class. For example, while testing an Activity, we may need to design test cases to test the result of interactions with UI elements, or check input validation, or behavior of the Activity itself when its life-cycle methods are fired.

In most of these scenarios, while unit testing an Activity, we have to access its UI elements inside the test case methods. Therefore, we need to initialize them and get appropriate handles in the `setUp()` method before execution of each test case method. After each test case method completes its execution, the `tearDown()` method performs the necessary windup such as cleaning up the variables and handles acquired in the `setUp()` method. Snippet 11.3 depicts the `setUp()` and `tearDown()` methods in the updated code snippet of the `TestMainActivity`. In Line 10, the `getActivity()` method of the `ActivityInstrumentationTestCase2` is invoked to start the Activity under test. In Lines 11 and 12, we are acquiring handles for TextViews in the Activity under test. In a similar manner, handles for all required views need to be obtained. In the `tearDown()` method, these handles, and the Activity itself, are dereferenced (Lines 17 and 18).

```
1  public class TestMainActivity extends
   ActivityInstrumentationTestCase2<MainActivity> {
2
3    private MainActivity activity;
```

```
  private TextView textView1, textView2, readTextView;
  private EditText inputEditText;
  private Button buttonRead, buttonWrite;

  protected void setUp() throws Exception {
    super.setUp();
    activity=getActivity();
    textView1=(TextView)activity.findViewById
     (com.mad.flatfile.R.id.textView1);
    textView2=(TextView)activity.findViewById
     (com.mad.flatfile.R.id.textView2);
    . . .
  }
  protected void tearDown() throws Exception {
    super.tearDown();
    activity=null;
    textView1=null;
    . . .
  }
}
```

**Snippet 11.3** | Implementing the `setUp()` and `tearDown()` methods.

Let us now design two test case methods – `testonPauseResumeImpactOnApp()` and `testSuc-cessfulFileSave()`. The first method will test whether UI elements retain their values during the Activity transition between active and paused states. The second method will test whether the input String is saved in the file. You may have noticed that both test case methods begin with "test", failing which the Android JUnit framework will not consider them as test case methods.

Snippet 11.4 enlists the `testonPauseResumeImpactOnApp()` method. Line 2 simulates a tap on the `inputEditText` using the `tapView()` method of the `TouchUtils` class[1] – a class that provides a mechanism to simulate user interactions on Android views (UI elements), while testing. Line 4 enters a String value in the `inputEditText` using the `sendStringSync()` method of `Instrumentation`. The `getInstrumentation()` method is used to obtain an instance of `Instrumentation`. Lines 5 and 6 invoke the `onPause()` and `onResume()` callback methods of the Activity using the `callActivityOnPause()` and `callActivityOnResume()` methods of `Instrumentation`. You may have noticed that although an Activity's callback methods are managed by Android runtime, for testing purposes we can manage them explicitly in test case methods. Lines 2–6 form the mechanism of the test case.

The test criteria of the test case is defined by comparing its expected and actual outcome. This comparison can be done using the `assertEquals()` method of the `Assert` class. Line 7 compares if the value entered in the `inputEditText` remains the same after the Activity transitions from paused to active state. If this comparison fails, resulting in failure of the test case, the failure message specified in the first argument of the `assertEquals()` will be displayed.

```
1  public void testonPauseResumeImpactOnApp() {
2    TouchUtils.tapView(this, inputEditText);
3    String message="This is a test string to be saved in file";
4    getInstrumentation().sendStringSync(message);
5    getInstrumentation().callActivityOnPause(activity);
6    getInstrumentation().callActivityOnResume(activity);
```

[1] TouchUtils: http://developer.android.com/reference/android/test/TouchUtils.html

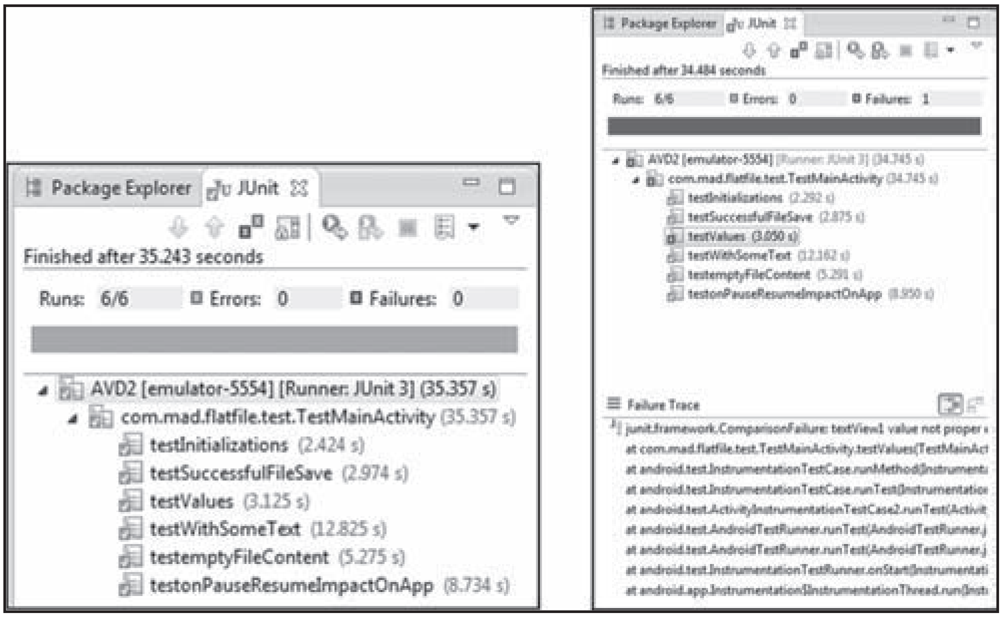
```
7    assertEquals("Test case failed: value changed during transition",
     message, inputEditText.getText().toString());
8  }
```

**Snippet 11.4** | `testonPauseResumeImpactOnApp()` method.

Snippet 11.5 enlists the second test case method – `testSuccessfulFileSave()`. In Line 2, we invoke the `runOnMainSync()` method so that we can directly manipulate the views in the Activity. This is a convenient way of accessing the UI elements against the usage of the `TouchUtils` class. The `runOnMainSync()` method accepts a `Runnable` instance as argument and executes the code inside its `run()` method on the *main* thread of the app. Inside the `run()` method, we are entering a text in the `inputEditText` (Line 6), simulating a tap on `buttonWrite` to save the entered data in a file (Line 7), and simulating a tap on `buttonRead` to retrieve the saved data from the file (Line 8).

```
1  public void testSuccessfulFileSave() {
2  getInstrumentation().runOnMainSync(new Runnable() {
3     @Override
4     public void run() {
5     String message="This is a test string to be saved in file";
6     inputEditText.setText(message);
7     buttonWrite.callOnClick();
8     buttonRead.callOnClick();
9     assertEquals(message, readTextView.getText().toString());
10    }
11 });
12 }
```

**Snippet 11.5** | `testSuccessfulFileSave()` method.



```
Finished after 35.243 seconds
Runs: 6/6    Errors: 0    Failures: 0
AVD2 [emulator-5554] [Runner: JUnit 3] (35.357 s)
  com.mad.flatfile.test.TestMainActivity (35.357 s)
    testInitializations (2.424 s)
    testSuccessfulFileSave (2.974 s)
    testValues (3.125 s)
    testWithSomeText (12.825 s)
    testemptyFileContent (5.275 s)
    testonPauseResumeImpactOnApp (8.734 s)
```

```
Finished after 34.484 seconds
Runs: 6/6    Errors: 0    Failures: 1
AVD2 [emulator-5554] [Runner: JUnit 3] (34.745 s)
  com.mad.flatfile.test.TestMainActivity (34.745 s)
    testInitializations (2.292 s)
    testSuccessfulFileSave (2.875 s)
    testValues (3.050 s)
    testWithSomeText (12.162 s)
    testemptyFileContent (5.291 s)
    testonPauseResumeImpactOnApp (8.950 s)
Failure Trace
junit.framework.ComparisonFailure: textView1 value not proper
  at com.mad.flatfile.test.TestMainActivity.testValues(TestMainAct
  at android.test.InstrumentationTestCase.runMethod(Instrument
  at android.test.InstrumentationTestCase.runTest(Instrumentatio
  at android.test.ActivityInstrumentationTestCase2.runTest(Activit
  at android.test.AndroidTestRunner.runTest(AndroidTestRunner.j
  at android.test.AndroidTestRunner.runTest(AndroidTestRunner.j
  at android.test.InstrumentationTestRunner.onStart(Instrumentati
  at android.app.Instrumentation$InstrumentationThread.run(Inst
```

**Figure 11.8** | JUnit window of Eclipse IDE depicting test results.

After defining the test mechanism, we are specifying the test criteria in Line 9 using the `assertEquals()` method, where we are comparing the result read from the file against the entered text.

Once all test case methods are designed, we can execute them by right clicking on the test project and selecting *Run As → Android JUnit Test*. The test project will be packaged as a .apk file and get installed on the emulator. Upon installation, the `InstrumentationTestRunner` runs the test case methods against the app under test.

As the test case methods get executed, their results will be displayed in the JUnit window of the Eclipse IDE, as shown in Fig. 11.8. This window displays the number of test case methods executed, errors occurred during their execution (if any), and number of test case methods that failed the assertion (if any). In case any test case method fails, the failure message can be seen in the Failure Trace section of the JUnit window.

### 11.2.2 Service

As Service is used for executing long-running tasks in the background, the approach to test it differs from that of an Activity. While testing a Service, we may need to design test cases either to test the business logic implemented in the long-running task or to ensure the execution of appropriate callback methods of a Service in response to request (to start/bind or stop/unbind the Service) from other components.

In this section, let us explore unit testing of a Service, using the `CounterService` class of the Counter app, created earlier in Section 5.5.3 of Chapter 5. We shall create a test project (TestCounterApp) with a test case class `TestCounterService` in it, to test the `CounterService`, as explained earlier in the Section 11.2.1. Here, we are going to test if the appropriate callback methods of a Service are getting executed in response to request from other components. To test such a scenario, we need to prepare the Service for testing. One of the ways to accomplish this is by introducing a few variables, in the original Service class, which keep track of Service states, as listed in Snippet 11.6 (Lines 4 and 5). The `mServiceCreated`, `mServiceStarted`, `mServiceBound`, and `onStartCommand` variables are flags to indicate if the `onCreate()`, `onStart()`, `onBind()`, and `onStartCommand()` callback methods have been called. The `mServiceCreationCount` variable is a flag indicating how many times the `onStartCommand()` has been executed. These flag variables need to be set with appropriate values inside different life-cycle methods of the `CounterService`.

```
1  public class CounterService extends IntentService {
2  boolean keepCounting = false;
3  int count = 0;
4    public boolean mServiceCreated, mServiceStarted,
     mServiceBound,onStartCommand;
5    public int mServiceCreationCount;
6  . . . }
```

**Snippet 11.6** | Preparing `CounterService` for unit testing.

The `TestCounterService` test case class is created by extending the `ServiceTestCase` class of the Instrumentation API, and overriding the `setUp()` and `tearDown()` methods, as enlisted in Snippet 11.7. Here, the constructor of the `ServiceTestCase` class is invoked with the Service under test (Line 5).

```
1  public class TestCounterService extends
   ServiceTestCase<CounterService> {
2    private Intent intent;
3    private CounterService service;
4    public TestCounterService() {
5         super(CounterService.class);
6
7    }
```

```
 protected void setUp() throws Exception {
      super.setUp();
      intent=new Intent(getContext(), CounterService.class);
 }
 protected void tearDown() throws Exception {
      super.tearDown();
      intent=null;
      service=null;
 }
. . .}
```

**Snippet 11.7** | `setUp()` and `tearDown()` methods in the `TestCounterService`.

Let us now design three test case methods – `testServiceStarts()`, `testServiceStops()`, and `testMutipleStarts()` – as depicted in Snippet 11.8. The `testServiceStarts()` method validates if the `onCreate()` and `onStart()` callbacks are called based on the values of `mService-Created` and `mServiceStarted` flag variables (Lines 4 and 5). The Service is started using the `start-Service()` method (Line 2), and its instance is retrieved using the `getService()` method (Line 3). The `testServiceStops()` method validates if the Service is stopped/unbound based on the values of `mServiceCreated`, `mServiceStarted`, and `mServiceBound` flag variables (Lines 11–13). The Service is stopped using the `shutdownService()` method of the `Instrumentation` (Line 9).

```
1   public void testServiceStarts() {
2     startService(intent);
3     service=getService();
4     assertEquals(true, service.mServiceCreated);
5     assertEquals(true, service.mServiceStarted);
6   }
7   public void testServiceStops() {
8     startService(intent);
9     shutdownService();
10    service=getService();
11    assertEquals(false, service.mServiceCreated);
12    assertEquals(false, service.mServiceStarted);
13    assertEquals(false, service.mServiceBound);
14  }
15  public void testMutipleStarts() {
16    startService(intent);
17    startService(intent);
18    startService(intent);
19    bindService(intent);
20    bindService(intent);
21    service=getService();
22    assertEquals(1, service.mServiceCreationCount);
23  }
```

**Snippet 11.8** | `TestCounterService` test case methods.

The `testMutipleStarts()` method validates that irrespective of the number of calls to `startService()` or `bindService()`, the Service is created only once by assesing the `mServiceCreationCount` variable (Line 22).

These test case methods need to be executed, as explained in the Section 11.2.1, to get the test results.

### 11.2.3 Content Provider

Testing a Content Provider, usually includes test cases to validate appropriate resolution of URIs (Uniform Resource Identifier), definition of MIME (Multipurpose Internet Mail Extensions) types, and result of CRUD (create, retrieve, update, and delete) operations. While testing a Content Provider, we have to ensure that the test data does not affect the actual data underlying in the data source. To deal with this, the Instrumentation API provides mock objects that work in an isolated environment to keep the original data intact.

In this section, let us explore unit testing of a Content Provider, using `ExpenseProvider` class of the ExpenseTracker app, created earlier in Section 6.6.2 of Chapter 6. We shall create a test project (TestExpenseTrackerApp) with a test case class `TestExpenseProvider` in it, to test the `ExpenseProvider`, as explained earlier in Section 11.2.1. Here we are going to test if appropriate MIME types are defined, and the results of CRUD operations.

The `TestExpenseProvider` test case class is created by extending the `ProviderTestCase2` class of the Instrumentation API, and overriding the `setUp()` and `tearDown()` methods, as enlisted in Snippet 11.9. Here, the constructor of the `ProviderTestCase2` class is invoked with the Content Provider under test and its authority (Line 5). In the `setUp()` method, an instance of the `MockContentResolver` (a mock object) and `Uri`s are initialized (Lines 9–11). Remember that the `MockContentResolver` works in an isolated manner, not affecting the actual data exposed by the Content Provider. Both these instances are dereferenced in the `tearDown()` method (Lines 13–17).

```
1   public class TestExpenseProvider extends
    ProviderTestCase2<ExpenseProvider> {
2   private MockContentResolver contentResolver;
3   private Uri sumuri, expenseuri;
4   public TestExpenseProvider(){
5      super(ExpenseProvider.class,
       “com.mad.expensetracker.expenseprovider”);
6   }
7   protected void setUp() throws Exception {
8      super.setUp();
9      contentResolver=this.getMockContentResolver();
10     sumuri=Uri.parse(“content://com.mad.expensetracker
       .expenseprovider/SUM”);
11     expenseuri=Uri.parse(“content://com.mad.expensetracker
       .expenseprovider/EXPENSES_TABLE”);
12  }
13  protected void tearDown() throws Exception {
14     super.tearDown();
15     contentResolver=null;
16  ...
17  }
18  . . . }
```

**Snippet 11.9** | `setUp()` and `tearDown()` methods in the `TestExpenseProvider`.

Let us now design two test case methods – `testPreConditions()` and `testQueryMethod()` – as depicted in Snippet 11.10. The `testPreConditions()` method validates if appropriate MIME type is returned from the `getType()` method of the Content Provider under test (Line 4). It also validates if the `cursor` object returned by querying the Content Provider, using the `MockContentResolver`, is not null and has a column and a row (as it is supposed to return the sum of expenses) (Lines 6–8).

```
1  public void testPreConditions()
2  {
3     String MIMEType=contentResolver.getType(sumuri);
4     assertEquals(“vnd.android.cursor.item/EXPENSES_TABLE”,
      MIMEType);
5     Cursor cursor=contentResolver.query(sumuri, null, null,
      null, null);
6     assertNotNull(cursor);
7     assertEquals(1, cursor.getCount());
8     assertEquals(1, cursor.getColumnCount());
9  }
10 public void testQueryMethod() {
11   String [] expenses=new
     String[]{“Clothes”,“Groceries”,“Snacks”};
12   String [] amounts=new String []{“1000”,“2000”,“100”};
13
14   for ( int i=0;i<3;i++)
15   {
16        ContentValues contentValues=new ContentValues();
17        contentValues.put(“expense”, expenses[i]);
18        contentValues.put(“amount”, amounts[i]);
19        contentResolver.insert(expenseuri, contentValues);
20   }
21   Cursor cursor=contentResolver.query(sumuri, null, null,
     null, null);
22   cursor.moveToNext();
23   assertEquals(3100, cursor.getInt(0));
24 }
```

**Snippet 11.10** | `TestExpenseProvider` test case methods.

In the `testQueryMethod()` method, the expense test data is inserted into the Content Provider using the `insert()` method of the `MockContentResolver` (Line 19). In Lines 21–23, the sum of the entered expense test data is validated against the value retrieved by querying the `sumuri`.

Finally, these test case methods need to be executed, as explained in Section 11.2.1, to get the test results. It is advisable to execute the test cases on real devices as well to ensure the conformance of a component in an actual environment, to avoid last minute surprises.

## 11.3 APP TESTING LANDSCAPE OVERVIEW

We have learnt that unit testing is a white-box testing of individual components of an app. However, to ensure that an app is comprehensively conforming to its specified functional and nonfunctional requirements at large, we need to validate it from several perspectives, including functionality, usability, performance, network, and security, as depicted in Fig. 11.9.

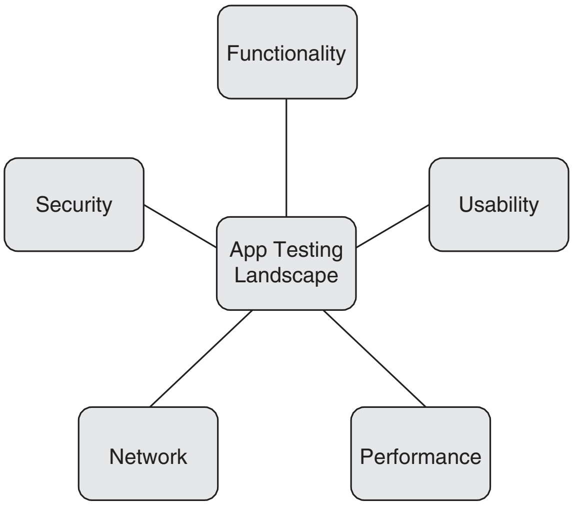


**Figure 11.9** | App Testing Landscape.

Before we explore the entire testing landscape to understand types of mobile testing and their respective focus areas of validation, we should be aware of one of the biggest challenges faced during app validation, that is, device diversity. It would be practically impossible to test an app on all available devices. This can be solved by using device sampling – a process to narrow down the number of devices on which testing should be carried out.

There is no set pattern to perform device sampling. One of the popular ways is to select a few devices from extreme ends of device spectrum based on processing power, memory, graphics capabilities, screen density, and screen size. Another popular method is to conduct extensive market research and analysis to identify the devices most commonly used by the targeted user group of the app. Ideally, a combination of both these approaches should be employed to get the target devices list for validating an app against functional and nonfunctional requirements.

Because it is impractical to procure all the targeted devices just for testing purposes, it is advisable to leverage cloud-based device sourcing that exposes adequate number of real devices for testing in a cost-effective manner. Keynote DeviceAnywhere is one such popular testing platform.

Functional requirements of an app are validated using *functionality testing* – a black-box testing methodology to validate end-to-end functionality of an app against the specified functional requirements. While performing functionality testing, we need to validate not only app-specific functionalities but also device-specific interventions that may influence the app functionality. Validating app-specific functionalities focuses on testing all the use cases that are implemented in the app. Device-specific interventions affect app functionalities in various ways due to variance in screen size, screen density, availability of sensors, and graphic capabilities on the targeted devices. Device-specific validation is necessary to ensure that an app adapts to these variations in a graceful manner. Functionality testing can be performed manually; however, it is advisable to automate it using tools such as MonkeyTalk.

*Usability testing* of an app is a black-box testing to ensure that the app is easy to understand and use. This testing has to be performed from an end-user's perspective by considering elements that help win an end-user's mind share. This may include, but not limited to, appropriate usage of the screen real estate, immediate availability of information that really matters to the end user, availability of features such as autocomplete to reduce typing efforts, and use of familiar icons, screen transitions, and typography.

We have to remember that usability is never an afterthought that is just getting validated towards the end of an app development life cycle. Rather, it should be established in the early phases of the app development itself, by creating app prototypes, and seeking feedback from the stakeholders.

*Performance testing* of an app is also a black-box testing that ensures responsiveness of the app under varying conditions of available device hardware capabilities (memory, processing power, battery life), and load on backend servers (used for remote data or functionality). Both these parameters considerably impact the performance of an app, and need to be tested thoroughly. HP LoadRunner is one of the popular tools to carry out performance testing.

Backend servers need to be tested for their ability to adapt to required load – both in terms of number of requests and volume of data – at a given point of time as well as over a period of time. This is ensured by various types of server-side performance testing where backend server is subjected to load testing, volume testing, stress testing, and endurance testing.

*Network testing* assesses the impact of network bandwidth on the performance of an app. As variation in bandwidths result in time differences to download data or retrieve functionality, we need to make sure that the user experience of an app does not suffer, even at lower bandwidth. This testing is typically done either by creating a controlled network environment to simulate various network bandwidths or on a real network itself.

*Security testing* ensures that the app is fortified from common application security vulnerabilities. This testing ensures that an app has appropriate authentication, authorization, and access controls in place. It also ensures that app data is secure – both at rest and in motion. HP Fortify is one of the popular tools to perform security testing.

This sums up the bird's eye view of landscape of mobile app validation. This learning area is quite vast, and requires exclusive attention. As the primary focus of this book is building mobile apps, this area is not being dealt in detail. We encourage interested readers to explore it further using pointers provided in the "Further Readings" section.

## SUMMARY

This chapter has introduced the need for mobile app validation and the consequences of nonconformance to an app's functional and nonfunctional requirements namely low retention of the app, or loss of revenue streams, or even loss of brand itself.

The chapter has also introduced unit testing of Android app components using the Android JUnit framework. It has elucidated the test project structure and its dependencies on the app project structure. Further, it has demonstrated unit testing of individual Android app components using simple scenarios from previous chapters.

The chapter has also presented a brief primer on mobile app testing landscape, and has discussed functional testing, usability testing, performance testing, network testing, and security testing fundamentals, along with typical scenarios/elements to be tested in an app to validate these individual focus areas.

## REVIEW QUESTIONS

1. Define a test project structure.
2. Outline the typical classes of an Instrumentation API that are used for unit testing of Android app components.
3. Explain the purpose of different types of testing for a mobile app.

## FURTHER READINGS

1. Testing fundamentals: http://developer.android.com/tools/testing/testing_android.html
2. MonkeyTalk: http://www.gorillalogic.com/monkeytalk
3. Keynote DeviceAnywhere: http://www.deviceanywhere.com/
4. HP LoadRunner:
   http://www8.hp.com/in/en/software-solutions/software.html?compURI=1171440
5. HP Fortify Software Security Center Server:
   http://www8.hp.com/in/en/software-solutions/software.html?compURI=1337262

# 12
# Publishing Apps

## LEARNING OBJECTIVES

*After completing this chapter, you will be able to:*

- Prepare an app for publishing.
- Configure the app version and API requirements.
- Package, sign, and optimize the app.
- Distribute the app on online app stores.

## 12.1 INTRODUCTION

App publication is the fruition phase of a mobile app development. This final stage of an app development is meant to distribute it to consumers through mechanisms such as app stores, websites, or even e-mails. The app publication process has to be dealt with utmost care because an improperly published app may fail to reach the targeted audience, and even if it does, it could create issues during app installation on the device. This may finally lead to loss of visibility of the app in the app store, loss of revenue, or even damage to the brand image.

This chapter delves into the process of an app publication. It enlists the necessary groundwork that is required to prepare the app for publishing, followed by specifying essential configurations such as an app version. This chapter also illustrates the signing and optimizing of an app package, and various mechanisms to distribute it.

## 12.2 GROUNDWORK

Preparing an app to get published requires a lot of sundry steps that need to be followed:

1. Sanitize the app.
2. Cross-check app resources.
3. Revisit app manifest.
4. Verify the remote server readiness, if any.
5. Decide pricing model.
6. Finalize legal agreement.
7. Create promotional material.

Sanitizing the app essentially removes the development-time debris such as unused resource files, Java source files used for testing and debugging purpose, log statements, and unused libraries. This, in turn, helps in reducing the memory foot print of the app.

App resources should be cross-checked for appropriate placement in the app project structure. This includes, but is not limited to, verifying if image files of appropriate resolution are placed in the designated drawable folders, and if appropriate locale-specific string resources are included in relevant values folders.

The app manifest has to be revisited to ensure that the chosen app package name is unique, because apps with the same app package name will not get installed on a device. App name also must be chosen carefully as this name will be displayed on the app stores and devices. We also have to ensure that only the necessary permissions are sought for in the manifest, as unnecessary permissions may lead to rejection of the app from the consumer. The value of `android:debuggable` attribute in the manifest file needs to be set to `false` to disable debugging when the app is running on a device in user mode (not in developer mode). The map key registered in the manifest (if any) should also be cross-checked with its entry in the Google APIs (Application Programming Interfaces) console.

Remote servers that the app depends on for data/functionality have to be verified to ensure that they are production ready. This is to ensure that these backend servers are secure and prepared for the anticipated load in terms of users and data.

The pricing model of the app must be decided before it is published on app stores. An app may be published with either a free pricing model, or a paid model. In case, where the app is intended to be a paid app, its pricing in different locations/currencies has to be decided.

Legal agreements such as End User License Agreement (EULA) and related artifacts may need to be created before uploading the app on various app stores to protect the app against legal and IP violations.

Promotional artifacts such as app description, screen shots, promotional videos have to be designed. These artifacts come handy while uploading the app on various app stores. The description text, screen shots, and promotional video have to be designed in a manner that captures consumer attention and improves the brand image. Description text must be well-written and contain all the important features of the app. The promotional video could be an informative video showcasing the overall app workflow. Screen shots should be selected carefully as they will help the user to visualize the app, and improve its chance of getting more downloads. Additionally, a high-quality app icon should be designed to improve the brand recognition of the app. Some app stores even mandate the creation of such high-quality icons.

The app configuration and packaging follows once the necessary groundwork is completed.

## 12.3 CONFIGURING

Configuring an app requires setting the app version and specifying the API requirements of the app. Setting the app version information is important for both users and app stores, as it may be used for upgrading to a newer version, categorizing multiple releases on an app store, or tracking changes across multiple releases. Specifying the API requirements of an app is essential to deal with the various platform versions on which it can get installed.

An Android app version is specified in its AndroidManifest file using the `versionName` and `versionCode` attributes, as depicted in Snippet 12.1. The `versionName` is a string that signifies the app version visible to the end user on app stores and devices. Line 4 demonstrates the use of `android:`–`versionName` to specify the first version of the 3CheersCable app. However, a developer uses the `versionCode` to track the app version for internal purposes, which is visible to the development team. The `versionCode` is also used by app update services to send on-the-air (OTA) updates, as and when a newer version of the app is made available. The end user is never exposed to the `versionCode` of an app.

```
1  <manifest
   xmlns:android=“http://schemas.android.com/apk/res/android”
2    package=“com.app3c”
3    android:versionCode=“1”
4    android:versionName=“v1.0”>
```

**Snippet 12.1** | First version of the 3CheersCable app.

For the later releases, the `versionCode` and `versionName` of an app would get incremented. The updated version of the 3CheersCable app is shown in Lines 3 and 4 of Snippet 12.2. Here, the minor changes in the next release are reflected in the `versionName` with a value `v1.1`. However, from a developer/app store's perspective, the `versionCode` has to be incremented in whole numbers.

```
1  <manifest
   xmlns:android=“http://schemas.android.com/apk/res/android”
2    package=“com.app3c”
3    android:versionCode=“2”
4    android:versionName=“v1.1” >
```

**Snippet 12.2** | Updated version of the 3CheersCable app.

Specifying the API requirements of an app resolves its compatibility with various Android™ platform versions. The `<uses-sdk>` tag of an app's manifest file is used to specify these requirements using the `minSdkVersion` and `targetSdkVersion` attributes, as depicted in Snippet 12.3. Recall that the values to these attributes were entered while creating an app using the New Android Application wizard in Chapter 2 (Section 2.3.1).

```
<uses-sdk
  android:minSdkVersion="8"
  android:targetSdkVersion="18"/>
```

**Snippet 12.3** | API requirements of the 3CheersCable app.

The `minSdkVersion` attribute specifies the lowest platform version of Android on which the app can be installed. The value of this attribute has to be chosen judiciously because keeping a higher value limits the range of Android platforms that can run the app. However, choosing a lower value may result in not being able to use the latest features as the developer has to ensure backward compatibility to cater to the specified version. It is advisable to choose the `minSdkVersion` as the most widely used Android platform version. In this example, we have selected Android Froyo (API level 8) as the `minSdkVersion`.

The `targetSdkVersion` attribute specifies the API version on which the app was implemented and tested. When an app is installed on a device that is running a platform whose version is lesser than or equal to that of the `targetSdkVersion`, then Android runtime will not pose any compatibility related issues. However, if the app is installed on a device that is running a platform whose version is greater than that of the `targetSdkVersion`, then Android runtime may run the app in compatibility mode. To avoid this, it is advisable to increment the `targetSdkVersion` to the latest API level (of course, after making necessary code changes to accommodate newer/updated platform features related to the app) while configuring the app, before packaging it.

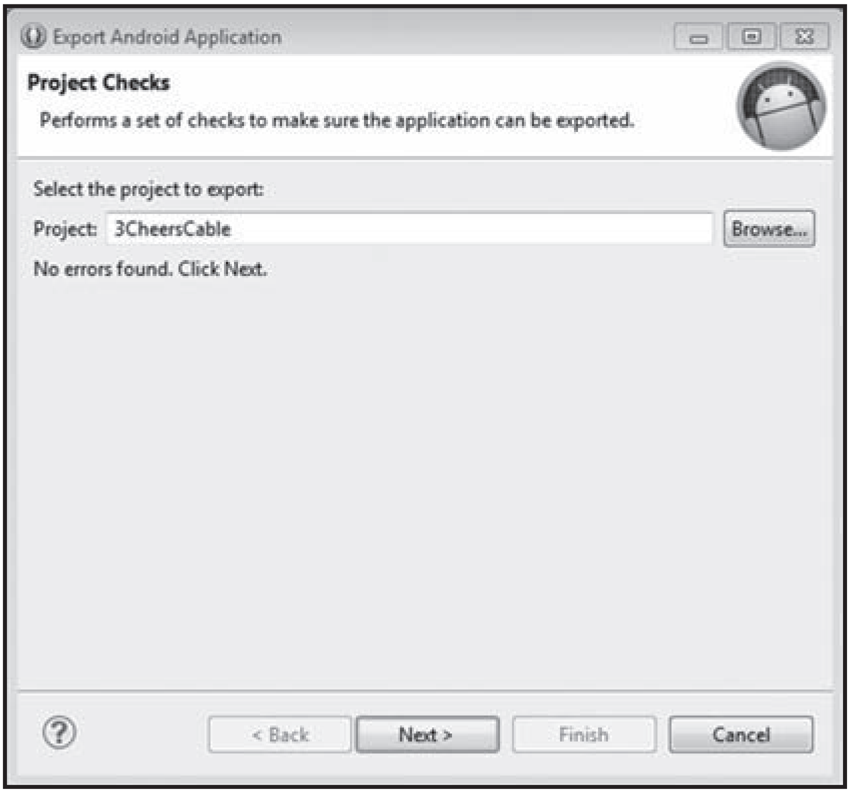
## 12.4 PACKAGING

Before distributing an Android app, it needs to be packaged as a signed .apk (Android Application Package) file containing a .dex file, the app manifest file, and resource files such as images, icons, assets, layouts, and menu (refer Section 2.3.3 of Chapter 2 for more information on .apk file). As seen earlier, during development and testing of an app, the Integrated Development Environment (IDE) automatically creates a .apk file and signs it using a default debug key, so that the app can run on emulators or devices in debug mode. However, this app is not yet ready to be distributed online.

To distribute an app online, it has to be signed by the developer/organization to ensure its authenticity. The .apk file is signed with an RSA key (generated using keytool utility, as discussed earlier in Section 9.4.1 of Chapter 9), using the jarsigner utility [a Java Development Kit (JDK) tool].

The IDE also provides a wizard that, in turn, invokes keytool and jarsigner utilities to create a private RSA key and sign the app with it, respectively. To sign the 3CheersCable app, right click on the project folder and select *Android Tools → Export Signed Application Package…*, to start the Export Android Application wizard, as shown in Fig. 12.1.

On clicking *Next*, the Keystore selection screen pops up that prompts to either select an existing keystore or create a new one, as depicted in Fig. 12.2. While selecting an existing keystore, we need



Project Checks

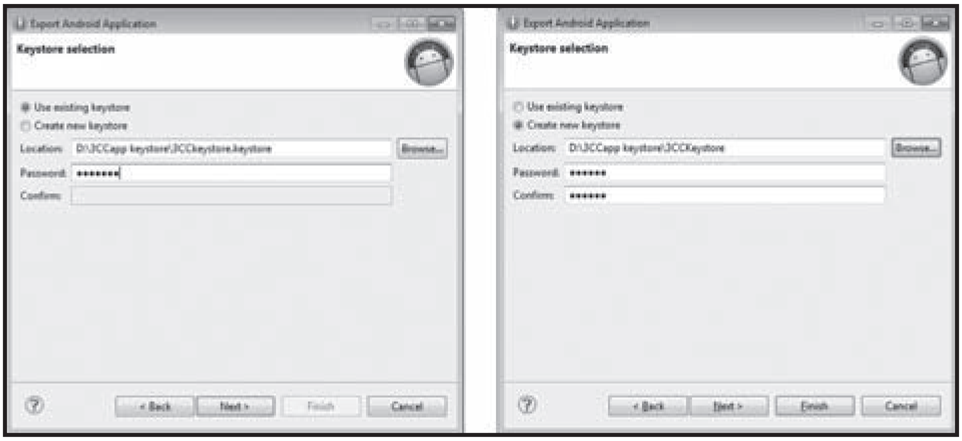
Performs a set of checks to make sure the application can be exported.

Select the project to export:

Project: 3CheersCable

No errors found. Click Next.

Figure 12.1 | Export Android Application Wizard.



Keystore selection

● Use existing keystore
○ Create new keystore

Location: D:\3CCapp keystore\3CCkeystore.keystore

Password: ••••••••

Confirm:

Keystore selection

○ Use existing keystore
● Create new keystore

Location: D:\3CCapp keystore\3CCKeystore

Password: •••••••

Confirm: •••••••

Figure 12.2 | Keystore Selection Screen.

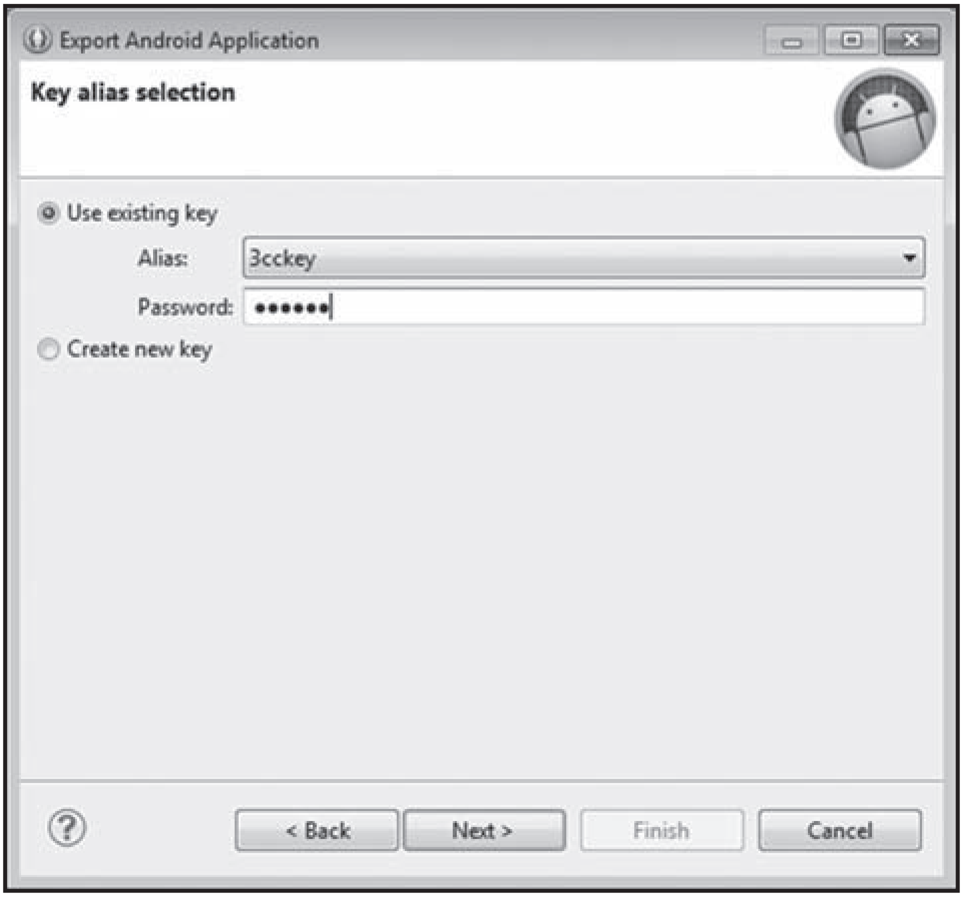
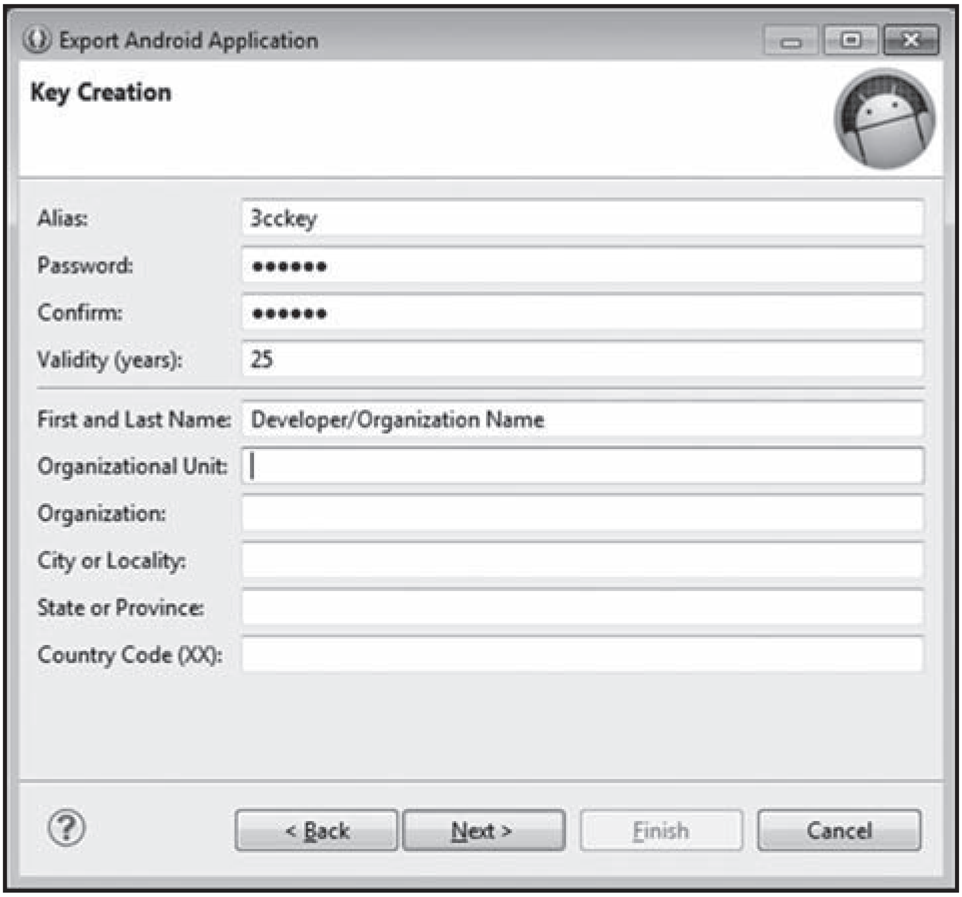


**Figure 12.3** | Key alias selection screen.

to point to the physical location of the keystore and enter the existing keystore password. This will lead to Key alias selection screen, as depicted in Fig. 12.3. While creating a new keystore, we need to provide the new keystore location and new keystore password. This will lead to the Key Creation screen, as depicted in Fig. 12.4.

Key alias selection screen, as depicted in Fig. 12.3, provides an option to use an existing key or create a new one. To use an existing key, we need to provide the existing key alias and the required key alias password. Creating a new key will also lead to Key Creation screen, as depicted in Fig. 12.4.

While creating a new key in the Key Creation screen, it is mandatory to specify an alias and a password to the key. Validity of the key is also mandatory, and recommended to be at least 25 years. This validity is checked by the Android runtime only during the installation of the app; therefore, an app continues to run on a user's device even after the certificate expires. Other fields in the Key Creation screen are self-explanatory, the First and Last Name field being the only mandatory one.



| | |
|---|---|
| Alias: | 3cckey |
| Password: | •••••• |
| Confirm: | •••••• |
| Validity (years): | 25 |
| First and Last Name: | Developer/Organization Name |
| Organizational Unit: | |
| Organization: | |
| City or Locality: | |
| State or Province: | |
| Country Code (XX): | |

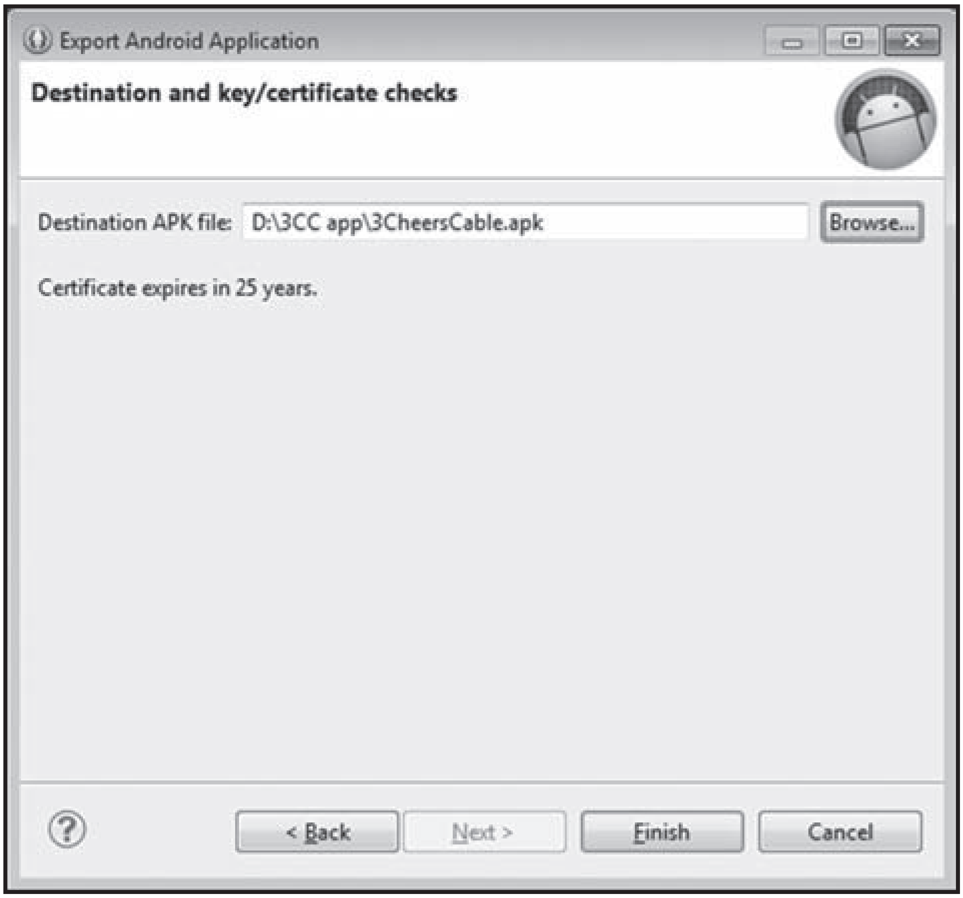
**Figure 12.4** | Key Creation screen.

Finally, we have to select the location of the signed .apk file in the Destination and key/certificate checks screen, as depicted in Fig. 12.5.

After the signing process, the obtained .apk file may contain uncompressed data such as images and other media. This uncompressed data in the file has to be aligned in order to optimize the packaging that in turn improves the performance of the app. This is done automatically by the IDE using the zipalign utility. The usage of this tool reduces the memory (Random Access Memory) consumed by the system while running the app.

Figure 12.6 depicts the keystore and the signed .apk file that gets generated as a result of the Export Android Application wizard. The generated .apk file is ready for distribution. The generated keystore file can be used later to sign more apps by the developer/organization.

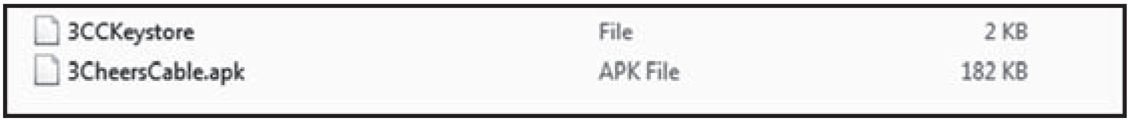
The signing process is a crucial aspect of app publishing from both the developer/organization and Android runtime perspective. From the developer/organization perspective, the key used forms the identity on online app markets and associates the app with the developer/organization. From the Android runtime


Export Android Application
Destination and key/certificate checks
Browse...
< Back
Next >
Finish
Cancel

Destination APK file: D:\3CC app\3CheersCable.apk

Certificate expires in 25 years.

**Figure 12.5** | Destination and Key/Certificate Checks Screen.



| | | |
|---|---|---|
| 3CCKeystore | File | 2 KB |
| 3CheersCable.apk | APK File | 182 KB |

**Figure 12.6** | Generated keystore and APK files.

perspective, apps signed using the same key are allowed to run in a single process. This proves essential when these apps exchange data with each other, wherein the security remains uncompromised and the interaction becomes seamless. The usage of the combination of same key and same app package name by the developer/organization while posting a newer version of the app on the market is considered by the Android runtime as an upgrade, and the upgrade happens seamlessly in such cases.

## 12.5 DISTRIBUTING

The final step in moving to market is distributing the app. Android apps may be distributed in several ways – distributing the app .apk file by e-mail, or hosting it on a website, or publishing the app on online app stores such as Google Play™ and Amazon Appstore for Android.

Distributing an app through e-mail would be a useful method when a limited number of end users are targeted. The .apk file is sent as an attachment, which can be downloaded and installed on an Android device.

Distributing an app by hosting its .apk file on a website or a blog is an easy method to reach out to a relatively larger audience. The end user can directly download and install the app on the device via the mobile browser. Although, this mechanism helps to distribute the app to relatively larger audience, there is not much control over the app from the perspective of upgradation, maintenance, or other life-cycle requirements, once it is installed.

Both the above methods are not foolproof mechanisms to take care of app distribution; therefore, publishing on online app stores is the most widely accepted mechanism. Typically, these online app stores charge a nominal fee for publishing apps, and provide various services to developers such as monitoring the number and frequency of app downloads, monitoring app usage patterns, establishing connect with service providers for localization and internationalization, tracking app performance in the market, accessing end-user's feedback, or viewing app crash reports. They also cater to distributing apps for specific set of regions, carrier networks, and device configurations. They also offer on-the-air update service for installed apps, as and when a newer version of the app is published on the store.

A developer may choose to publish an app with a paid pricing model or for free on an app store. In case of paid pricing model, the online app stores may also charge percentage cut of the app selling price. App stores support both direct and indirect monetization of apps in paid pricing model. Direct monetization is implemented via in-app purchases and subscriptions, where digital content can be sold through the app. For example, advanced app features are sold typically as in-app purchases, and content is sold as subscription, especially for content-based apps. Indirect monetization is implemented via hosting advertisements in an app (in-app ads).

Most of the developers prefer publishing apps through app stores due to its global presence. The confidence of end users regarding the authenticity of apps published on app stores is much higher, thereby yielding higher benefits to developers in terms of app downloads and revenue.

## SUMMARY

This chapter has introduced the sundry list of items that need to be performed by an app developer to prepare the app before its distribution. This typically includes sanitizing the app, cross-checking the app resources, revisiting the app manifest, verifying the remote server readiness, deciding a pricing model, finalizing the legal agreement, and creating promotional material.

The chapter has also introduced app configuration in terms of setting the app version and specifying its API requirements. It has also described the process of packaging, signing, and optimizing Android apps.

Toward the end, the chapter has outlined various mechanisms to distribute an app – e-mail, websites, and online app stores. It has also brought out the benefits of distributing apps through app stores.

## REVIEW QUESTIONS

1. Define the prework that is required to publish an app.
2. Describe the importance of signing an app before publishing.
3. Explain the various mechanisms to distribute a mobile app.

## FURTHER READINGS

1. Launch checklist: http://developer.android.com/distribute/googleplay/publish/preparing.html
2. Promoting your apps: http://developer.android.com/distribute/googleplay/promote/index.html
3. zipalign: http://developer.android.com/tools/help/zipalign.html
4. Google Play policies and guidelines: http://developer.android.com/distribute/googleplay/policies/index.html
5. Distribution control: http://developer.android.com/distribute/googleplay/about/distribution.html

# References

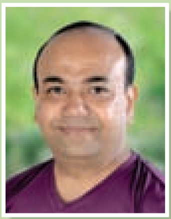
- Android: http://www.android.com/
- Apple iOS: http://www.apple.com/
- jQuery Mobile: http://jquerymobile.com/
- Sencha Touch: http://www.sencha.com/products/touch
- PhoneGap: http://phonegap.com/
- Titanium: http://www.appcelerator.com/titanium/
- SAP SMP: http://www54.sap.com/pc/tech/mobile/software/solutions/platform/app-development-earn.html
- Kony: http://www.kony.com/
- Java SE – Downloads: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Android developers' website: http://developer.android.com
- Android development environment: http://developer.android.com/sdk/index.html
- Android studio: http://developer.android.com/sdk/installing/studio.html
- Android tool repository: http://developer.android.com/tools/help/index.html
- Running app on devices: http://developer.android.com/tools/device.html
- Anubhav Pradhan, Satheesha B. Nanjappa, Senthil K. Nallasamy, and Veerakumar Esakimuthu (2010), *Raising Enterprise Application: A Software Engineering Perspective*, Wiley India, New Delhi.
- Activity and its life cycle: http://developer.android.com/guide/components/activities.html
- App resources: http://developer.android.com/guide/topics/resources/index.html
- App user interface: http://developer.android.com/guide/topics/ui/index.html
- Fragments: http://developer.android.com/guide/components/fragments.html
- Intent: http://developer.android.com/reference/android/content/Intent.html
- AsyncTask: http://developer.android.com/reference/android/os/AsyncTask.html
- Service and its life cycle : http://developer.android.com/guide/components/services.html
- IntentService: http://developer.android.com/reference/android/app/IntentService.html
- Bound Services: http://developer.android.com/guide/components/bound-services.html
- Notifications: http://developer.android.com/guide/topics/ui/notifiers/notifications.html
- Intents and intent filters: http://developer.android.com/guide/components/intents-filters.html
- BroadcastReceiver: http://developer.android.com/reference/android/content/BroadcastReceiver.html
- TelephonyManager:http://developer.android.com/reference/android/telephony/TelephonyManager.html
- SmsManager: http://developer.android.com/reference/android/telephony/SmsManager.html
- Web Services Architecture: http://www.w3.org/TR/ws-arch/
- Introducing JSON: http://www.json.org/
- Storage options in android: http://developer.android.com/guide/topics/data/data-storage.html
- SharedPreferences: http://developer.android.com/reference/android/content/SharedPreferences.html
- Preference API: http://developer.android.com/guide/topics/ui/settings.html
- Content Providers: http:// developer.android.com/guide/topics/providers/content-providers.html
- JsonReader API: http://developer.android.com/reference/android/util/JsonReader.html

- Supporting multiple screens: http://developer.android.com/guide/practices/screens_support.html
- Drawable resources: http://developer.android.com/guide/topics/resources/drawable-resource.html
- Property animation: http://developer.android.com/guide/topics/graphics/prop-animation.html
- Supported media formats in Android: http://developer.android.com/guide/appendix/media-formats.html
- MediaPlayer states: http://developer.android.com/reference/android/media/MediaPlayer.html#Valid_and_Invalid_States
- MediaRecorder: http://developer.android.com/reference/android/media/MediaRecorder.html
- Google location awareness: http://developer.android.com/training/location/index.html
- Google API Console: https://code.google.com/apis/console
- Google map key information: https://developers.google.com/maps/documentation/android/start
- Set up Google Play Services SDK: http://developer.android.com/google/play-services/setup.html
- Making your app location-aware: http://developer.android.com/training/location/index.html
- Google maps: https://developers.google.com/maps/documentation/android/start
- The Google Directions API: https://developers.google.com/maps/documentation/directions/
- Sensors overview: http://developer.android.com/guide/topics/sensors/sensors_overview.html
- Motion sensors: http://developer.android.com/guide/topics/sensors/sensors_motion.html
- Position sensors: http://developer.android.com/guide/topics/sensors/sensors_position.html
- Environment sensors: http://developer.android.com/guide/topics/sensors/sensors_environment.html
- Testing fundamentals: http://developer.android.com/tools/testing/testing_android.html
- MonkeyTalk: http://www.gorillalogic.com/monkeytalk
- Keynote DeviceAnywhere: http://www.deviceanywhere.com/
- HP LoadRunner: http://www8.hp.com/in/en/software-solutions/software.html?compURI=1171440
- HP Fortify Software Security Center Server: http://www8.hp.com/in/en/software-solutions/software.html?compURI=1337262
- Launch checklist: http://developer.android.com/distribute/googleplay/publish/preparing.html
- Promoting your apps: http://developer.android.com/distribute/googleplay/promote/index.html
- zipalign: http://developer.android.com/tools/help/zipalign.html
- Google play policies and guidelines: http://developer.android.com/distribute/googleplay/policies/index.html
- Distribution control: http://developer.android.com/distribute/googleplay/about/distribution.html

# Index

## About the Book

Composing Mobile Apps attempts to present various mobile app development approaches and technologies, along with required hands-on knowledge to deal with the nitty-gritties of designing, developing, validating, packaging and publishing them. A live mobile app has been built incrementally throughout the book using Android – one of the most popular mobile platforms – in order to bring out necessary nuances and techniques of app development, and provides the much needed practical exposure. It also provides the essential guidance on how to add that extra zing to mobile apps by leveraging sensors, graphics, animation, multimedia and location awareness capabilities of smart devices. This book is a 'do-it-yourself' guide for all mobility enthusiasts who are looking forward to develop winning mobile apps.



**ANUBHAV PRADHAN** is a technology evangelist and learning specialist with more than 17 years of IT industry experience in learning/education services. He currently heads the mobility academy at Infosys Limited that caters to diversified competency development requirements across a wide spectrum of mobile technologies – native apps development, responsive web design, mobile crossplatforms, mobile middleware, mobile validation, and mobile application management.

He has designed and delivered several large-scale competency development programs on a wide range of technologies across the globe. He has several publications to his credit that include the book 'Raising Enterprise Applications' (Wiley, 2010). He holds a master's degree in Mathematics.

**ANIL V DESHPANDE** is a technology evangelist with 7.5 years of IT industry and academia experience in education services and mobile app development. He currently works as a senior member of the mobility academy at Infosys Limited and has designed and developed many mobile competency development initiatives. He primarily evangelizes native app development platforms – Android and Blackberry, and has helped software delivery teams in developing and delivering mobile app modules. He also works in mobile validation area, and provides consultancy to app testing teams. Prior to this, he has worked quite extensively in Java enterprise technologies. He holds a master's degree in Computer Science and Engineering.

## Salient Features

- Classifies mobility space and outlines mobile app development approaches and technologies.
- Identifies mobile app development challenges and illustrates tenets of a winning app.
- Implements a live mobile app demonstrating app development from inception to publishing.
- Delves into designing app user interface that can adapt to multiple screen densities and form factors.
- Describes design and implementation of long running tasks in an app.
- Defines mechanisms to respond to device events in an app.
- Illustrates management of native and enterprise data in an app.
- Explores usage of graphics and animation capabilities in an app.
- Explains design and implementation of media playback, capture and storage in an app.
- Outlines techniques to incorporate location awareness in an app.
- Demonstrates usage of motion, position and environment sensors in an app.
- Unfolds app testing landscape.
- Discusses strategies and best practices for publishing an app.

**Wiley India Pvt. Ltd.**
4435-36/7, Ansari Road, Daryaganj
New Delhi-110 002
Customer Care +91 11 43630000
Fax +91 11 23275895
csupport@wiley.com
www.wileyindia.com
This edition is authorized for sale in the Indian sub-continent only.

ISBN: 978-81-265-4660-2

WILEY