



Community Experience Distilled

Learning Android Intents

Explore and apply the power of intents in Android application development

Muhammad Usama bin Aftab
Wajahat Karim

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Learning Android Intents

Explore and apply the power of intents in Android application development

Muhammad Usama bin Aftab

Wajahat Karim

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Learning Android Intents

Copyright © 2014 Packt Publishing

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

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

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

First published: January 2014

Production Reference: 1160114

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

ISBN 978-1-78328-963-9

www.packtpub.com

Cover Image by Prashant Timappa Shetty (sparkling.spectrum.123@gmail.com)

Credits

Authors

Muhammad Usama bin Aftab
Wajahat Karim

Reviewers

K. B. Vinay Kumar
Sharafat Ibn Mollah Mosharraf
Thiago Rocha
Sheeraz Shaikh
Imran Tanveer

Acquisition Editors

Amarabha Banerjee
Usha Iyer
Meeta Rajani

Lead Technical Editor

Susmita Panda

Technical Editors

Vrinda Amberkar Bhosale
Menza Mathew
Aman Preet Singh
Pratish Soman

Copy Editors

Janbal Dharmaraj
Deepa Nambiar
Karuna Narayanan
Lavina Pereira

Project Coordinator

Priyanka Goel

Proofreaders

Simran Bhogal
Linda Morris
Lindsey Thomas

Indexer

Rekha Nair

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

About the Authors

Muhammad Usama bin Aftab is a telecommunications engineer with a flair for programming. He has been working in the IT industry for the last two years, in which he worked on Android Development, AndEngine GLES 1 and 2, Starling, Adobe Air, and Unity 3D. He also has a total of two years of Android experience consisting of professional and freelance work that he has done. In June 2011, he started his career from a silicon-valley-based company named Folio3 Pvt. Ltd. Folio3 guided him a lot. This helped him discover various technologies with highly qualified professionals.

I would like to thank my parents for everything they did for my education and took me to the next step in my career; my sister, Goya Khan for always teasing me; and my friend/teacher/mentor, Farzeen Qureshi who always held me tightly in difficult times of my life. I thank Folio3 for being the first step in my professional life, for teaching me different platforms, and keeping me motivated. I give my special thanks to M. Aamir Ibrahim, Ayesha Ibrahim, and Wahaj Sherwani for teaching me Android/Game Development and helping me stay updated with the current technologies. I thank all of my peers, co-author, and Packt Publishing for their trust in me. Last but not least; I thank Allah (the Almighty) because without his help, I would just be a particle in sand.

Wajahat Karim is a software engineer and has a high interest in game development for mobile and Facebook platforms. He completed his graduation from NUST School of Electrical Engineering & Computer Sciences (SEECS), Islamabad, Pakistan. He has been working on games since he was in the third year of his graduation. He is skilled in many platforms including Android SDK, AndEngine GLES 1 and 2, Adobe Flash, Adobe Flex, Adobe AIR, Unity3D, and Game Maker. He is also skilled, not only in programming and coding, but also in computer graphics tools, such as Adobe Photoshop CS5, Adobe Illustrator, Adobe Flash, 3D Studio Max, and Autodesk Maya 2012. After working on a Facebook game in WhiteRabbit Studios until September 2012, he joined a silicon valley-based company, Folio3 Pvt. Ltd, where he provides his services in mobile games using Unity3D, Adobe Flash, and AndEngine. He also runs his own mobile app/game startup called AppSoul Studio (Pvt.) Ltd. in his part time

First of all, I would like to thank Allah (the Almighty) for everything and this life. Then I would like to thank my lovely sisters, Navera Karim and Sumera Aijaz, who always have been proud of me and would be shocked and surprised when they'll get this book in their hands. Then, I would like to thank my parents, Ammi and Abu, and my aunty, for all their prayers and support, motivation, and hopes for me, and my cousins, Fayaz Ahmed Memon, Ayaz Ahmed Memon, and Sheeraz Ahmed Memon, who are more than cousins and brothers to me. I would like to thank my best teachers, Shahid Razzaq, Shamyil Bin Mansoor, and Qasim Rajpoot, who taught me everything that I have written in this book directly or indirectly. I would like to thank my fiancée, Gul Sanober, for her understanding and support throughout the process of writing this book and for her ongoing support in allowing me to do what I truly enjoy in life. I would like to thank my students for keeping me motivated, especially Asad Hussain, Saifullah, Muhammad Saad, and Sara Ali for their support throughout the whole writing process by asking almost every day about the progress of the book. Thanks to my best friends, Arslan Ahmed Abro, Mubashir Hassan, and Ali Hussain, for making my life more funny and enjoyable. Last but not least; I would like to say a BIG thanks to my co-author Usama Aftab and Packt Publishing for their support and help in setting this whole project in motion and publishing my first ever book.

About the Reviewers

K B Vinay Kumar completed his graduation in 2011 from JNTU Hyderabad; right after his graduation, he picked up an opportunity in startup software solution as an Android application developer. Apart from application development, he is quite interested in exploring new things in the programming world; at the start of his two years of experience with the Android framework, he worked on cross-platform implementation using Xamarin.

Sharafat Ibn Mollah Mosharraf graduated from the University of Dhaka in Computer Science and Engineering. He is currently working as an associate senior software engineer at Therap Services, LLC. He has expertise and experience in architecting, and designing and developing enterprise applications in Java, PHP, and Android. He loves researching, as well as training people on state-of-the-art technologies for designing, developing, securing, and maintaining Web and mobile applications. He also provides coaching for various teams participating in national application development contests. His areas of interest include user experience, application security, application performance, and designing scalable applications. He loves spending his free time with his family and friends.

I'd like to thank the author for writing such a wonderful book on Android Intents. It had been difficult for me to train people to master this topic due to a lack of elaborated and organized resources. I'd also like to thank Priyanka Goel, the project coordinator of the book. It was a pleasure to work with you. And last but not least, I thank my wife, Sadaf Ishaq, for bearing with me while I was busy reviewing the book. It's been always great to have you by my side!

Thiago Rocha, also known as Kimo, is interested in the intersection of technology and entertainment. He is a Bachelor in Computer Science, graduated from Pontifícia Universidade Católica de Minas Gerais, Brazil. He started working professionally with Android development since Android Cupcake, and nowadays, he experiments with web development using Ruby. When he's not working, Thiago dedicates his time to practice some serious table tennis. In 2013, he got his third title of Absolute Champion of Minas Gerais', which is the most important table tennis title of Minas Gerais state. Thiago is part of Codelogic, a team of passionate developers (and also friends).

First, I would like to thank God, without Him, nothing would be possible. Priyanka Goel, for providing me with this amazing experience of reviewing a book. My family, that took care of me and helped me become the person that I am today. Last, but not least, my friends (from Codelogic), who supported me to accept this challenge.

Imran Tanveer is a self-learner and an Android geek who is passionate about application development. He graduated from the National University of Science and Technology in 2011 with an award-winning Android application as his final year project and has been working in the industry ever since. In his short time as a professional, he has worked for various organizations and has developed a number of Android applications. Many of his applications are available on Google Play, and have received tremendous rankings from users from all over the world. His work has also been appreciated by Ericsson and was selected twice in the top six applications from Pakistan in the Ericsson-PTA Mobile Excellence Awards, for one of which, his application was awarded the runner-up prize and a nomination in the mBillionth South Asia Awards.

Imran's vast knowledge and fervor for Android application development transcends the career boundaries of most developers and has him finding his place in education. He has delivered lectures for Android application development workshops in various universities, and has gone far enough to extend his help to students working in the domain as their external advisor. Apart from development and instruction, he also writes training material on Android application development for Android ATC, a Texas-based company.

Simply put, Imran Tanveer lives and breathes the Android domain and is willing to go the extra mile in every direction, every day.

I would like to thank the Almighty Allah and my family for always supporting and helping me achieve whatever I have achieved today. I would also like to thank my friends, Adeena, Neelofar, Arshad, and Ikram for always motivating and encouraging me whenever I got distracted.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Understanding Android	7
Introducing Android	7
Exploring the different versions of Android	8
Google Play – the official app store for Android	10
Understanding the whys and whens of Android	11
The evolution of Android OS	12
Official IDE from Google – the Android Studio	14
Features of Android Studio	15
Limitations of Android Studio	16
Building blocks of an Android application	16
Coding components	18
Media components	18
The assets folder	18
The res folder	19
XML components	19
The layout folder	20
The menu folder	20
The values folder	21
AndroidManifest.xml	22
Referencing components	23
Library components	23
Android Activity lifecycle	24
Fundamental states of an activity	26
The callback methods of the Activity lifecycle	26
The activity lifecycle flow	30
Summary	32

Chapter 2: Introduction to Android Intents	33
Role of intents in an Android Application	34
Role of intents in Android Activities	35
Role of intents in data transfer between activities	35
Role of intents in Wi-Fi and Bluetooth transfer	36
Role of intents in Android Camera	36
Role of intents in GPS Sensor	36
Role of intents in sending SMS/MMS	36
Role of intents in Mobile Calls	36
Role of intents in e-mail and social network posts	37
Role of intents in Android Services	38
Role of intent in Broadcast Receiver	38
Role of intent in time zones	39
Role of intent in Status Bar	39
Intent – a technical overview	40
The Coding component	40
The XML component	41
Implementation of Android Intents for Activity Navigation	42
Understanding the flow	46
Part one – MainActivity.java	46
Part two – MySecondActivity.java	47
Part three – activity_main.xml	47
Part four – activity_two_layout.xml	47
Part five – AndroidManifest.xml	48
Other constructors of the android.content.Intent class	49
Intent()	49
Intent(intent o)	49
Intent(Context c, Class<?> cls)	49
Intent(String action)	49
Intent(String action, URI uri)	50
Getting results from Android Intents	50
Understanding with an example	51
Going deep into the example	51
Explaining the code	56
Structure of an intent	58
Component	58
Actions	59
Data	59
Extras	60
Summary	60

Chapter 3: Intent and Its Categorization	61
Explicit intents	62
Using explicit intents in an Android application	63
Starting an activity through an explicit intent	63
Starting a service through an explicit intent	72
Implicit intents	77
Using implicit intents in an Android application	78
Sharing content using implicit intents	78
Selecting an image through an implicit intent	88
Intents and Android late binding	93
Summary	93
Chapter 4: Intents for Mobile Components	95
Common mobile components	96
The Wi-Fi component	96
The Bluetooth component	96
The Cellular component	97
Global Positioning System (GPS) and geo-location	97
The Geomagnetic field component	97
Sensor components	97
Motion sensors	98
Position sensors	98
Environmental sensors	98
Components and intents	98
Communication components	99
Using Bluetooth through intents	99
Using Wi-Fi through intents	111
Media components	120
Using intents to take pictures	120
Using intents to record video	127
Speech recognition using intents	130
Role of intents in text-to-speech conversion	134
Motion components	137
Intents and proximity alerts	137
Role of intents in proximity alerts	139
Summary	141
Chapter 5: Data Transfer Using Intents	143
Finding the need to transfer data	143
Taking a simple example	144
Data transfer between activities – an INTENTed way	145
Data transfer in explicit intents	146

Methods of data transfer between activities	146
Data transfer using putExtras()	146
Implementation of putExtras()	147
Extras supported data types	157
The concept of Android Bundles	158
Data transfer using Parcelable	160
Implementation of Parcelable	160
Data transfer using Serializable	172
What is Serializable?	172
An example of Serializable	173
Implementation of Serializable	175
Data and the implicit intents	186
Viewing a map	187
Opening a webpage	188
Sending an e-mail	189
Making a call	190
Miscellaneous scenarios	190
Summary	190
Chapter 6: Accessing Android Features Using Intents	191
Features of Android OS	192
Android features versus components	193
Common Android features	193
Layouts and display	194
Data storage and retrieval	195
Connectivity and communication	196
Accessibility and multitouch	197
Extensive content and media support	198
Hardware support	199
Background services and multitasking	199
Enhanced home screen	200
Other Android features	200
Android features and intents	201
The <uses-feature> and <uses-permission> tags	201
Hardware features	203
Software features	204
Sharing using the SEND action	207
Telephony and making calls using intents	213
Making phone calls using intents	214
SMS/MMS using intents	217
Sending SMS using intents	218
Sending MMS using intents	221

Confirming message delivery using intents	222
Receiving SMS messages using intents	225
The SmsManager class	225
The SmsMessage object	225
Protocol Data Unit (PDU)	225
Notification using intents	229
Notification forms	230
The NotificationManager class	230
The Notification class	230
The Notification layout	230
Summary	236
Chapter 7: Intent Filters	237
<hr/>	
Intent object and its categorization	237
Component name	238
Intent resolution	238
Action	239
Data	240
Use of data in ACTION_EDIT	240
Use of data in ACTION_CALL	240
Use of data in ACTION_VIEW	240
Category	240
Extras	241
Intent filters	241
Handling multiple intent filters	243
Test components of an intent filter	243
Action test	244
Writing conventions for <action>	244
Category test	246
Setting up the launcher activity	247
Data test	248
Typical representation of the <data> tag	249
Summary	250
Chapter 8: Broadcasting Intents	251
<hr/>	
Broadcasting in the Android OS	252
The broadcast intents	252
Built-in broadcasts in Android systems	253
Detecting the low-battery state of a device	255
The BatteryLowReceiver.java file	256
The BatteryLowActivity.java class	257
The AndroidManifest.xml file	258
Detecting the screen on/off state of a phone	260
The ScreenOnOffReceiver.java file	260

The AndroidManifest.xml file	262
Detecting the cell phone's reboot-completed state	263
The PhoneRebootCompletedReceiver.java file	264
The TempService.java file	264
The AndroidManifest.xml file	266
Sending and receiving custom broadcasts	267
The activity_main.xml layout file	267
The MainActivity.java file	268
The CustomReceiver.java file	269
The AndroidManifest.xml file	270
Summary	271
Chapter 9: Intent Service and Pending Intents	273
Intent Service	273
Comparison of four fundamentals	274
Best case to use	274
Triggers	275
Usage and implementation of Intent Service	275
Generating a fake notification from Intent Service	276
Taking another example	282
Pending Intents	285
How to make Pending Intents work?	285
Summary	288
Index	289

Preface

Android is an emerging technology with lots of apps on the Google Play market. Till date, it is the biggest marvel in smartphone technology, propelling a larger number of developers toward Android development. Intent is an essential part of any Android application, and no Android application is complete without using them. Features such as listening for broadcasts, sending messages, sharing via social networks, sending notifications, and accessing hardware components such as camera, sensors, and Wi-Fi, can be easily be carried out in your Android applications using intents.

Learning Android Intents focuses on using intents to make the best use of various features of Android platforms. It is ideal for developers who want to understand the backbone and the domain of Android intents, its power, and the need of it inside an Android application. Practical, in-depth examples are used throughout the book to help understand the key concepts of using intents.

The book starts by introducing the very basic concepts of Android and its various facts and figures, such as the different Android versions, their release dates, and evolution of Android devices. While covering the basic technical concepts, it proceeds from the easiest route of introducing Android intents toward the more practical view of Android intents in terms of components and features.

In this book, you will learn how to use different components and features such as transferring data between activities, invoke various features and components of Android, execute different built-in and custom-made services, access the hardware and software components of an Android device, and send notifications and alarms. You will gain theoretical knowledge of what is running behind the concepts of Android intents and practical knowledge of the mobile-efficient ways to perform a certain task using Android intents.

Toward the end, you will have a clear vision and practical grip on Android intents and their power.

What this book covers

Chapter 1, Understanding Android, covers the basic knowledge and key concepts of the Android system, its versions, a brief history of the Android OS, Google Play Market, and Android Studio. This chapter also covers topics from the development perspective including the building blocks of Android application, Activity lifecycle, and its callback methods.

Chapter 2, Introduction to Android Intents, covers the introduction of intents, basic key concepts of intent, the role of intents in Android applications, a technical overview of intents, use of objects in the `android.content.Intent` class, and its structure. Further, this chapter also explains two practical examples of how to use intents to navigate from one activity to another.

Chapter 3, Intents and Its Categorization, covers more details about intents and expands on their categories such as explicit intents and implicit intents. This chapter also provides practical implementation examples of using intents, such as sharing data with other apps, getting shared data from other Android apps, picking images from a gallery, and starting an activity or services through intents.

Chapter 4, Intents for Mobile Components, covers the basic knowledge about most common hardware components found in every Android device such as Wi-Fi, Bluetooth, cellular, Global Positioning System (GPS), geomagnetic fields, and motion and position sensors. After that, this chapter provides details on the role of intents with these hardware components along with practical examples for using intents, including turning Bluetooth on/off, making a device discoverable, turning the Wi-Fi on/off, opening Wi-Fi settings, taking pictures, recording videos, and carrying out speech recognition and text-to-speech conversion.

Chapter 5, Data Transfer Using Intents, covers the in-depth details of data transfer using intents. This chapter discusses transferring data between activities through different methods, simple data transfer using the `putExtra()` method of the `Intent` class, sending custom data objects by the `Parcelable` and `Serializable` class objects, and some scenarios of data transfer in the Android system.

Chapter 6, Accessing Android Features Using Intents, covers the most common software features such as layouts, display, connectivity, communication, accessibility, touch, and hardware support found in the Android OS. The chapter contains a discussion on two important `AndroidManifest` tags, `<uses-feature>` and `<uses-permission>`, their use, and comparison of mobile hardware components with Android OS features relating to these tags. This chapter provides practical example implementations of using intents in Android applications such as making calls, sending SMS/MMS messages, confirming message delivery, receiving messages, and sending notifications with custom layouts.

Chapter 7, Intent Filters, covers the details about intent and intent filters and how they provide the Android OS with information about the activities present inside the application. This chapter also covers details about filter tests such as action test, data test, category test, and how these tests come in handy when using intents.

Chapter 8, Broadcasting Intents, covers broadcasting in Android and the broadcast intents. This chapter provides a discussion on the Android OS's System Broadcast Intents such as battery low, power connected/disconnected, booting completed and head-set plugged in/out along with some practical example implementations of those intents. Also, this chapter covers custom broadcast intents and their use in various cases with practical examples.

Chapter 9, Intent Service and Pending Intents, covers the most advanced topics for intents such as using `IntentService` objects in contrast to the common methods such as `Thread`, `Service`, or `AsyncTask`. This chapter covers the `PendingIntent` objects and their use in practical example implementation.

What you need for this book

The software required in order to execute the various examples in the book include any IDE for Android development, preferably the Eclipse IDE with the latest Android SDK or Android Studio (which is in preview release at time of writing this book).

Who this book is for

Learning Android Intents is geared toward novice or intermediate developers who want to expand their knowledge of Android Intents. Readers are expected to have a basic understanding of Android development, how to use different Android IDEs, and how to develop applications using native Android SDK APIs.

This book is useful for every Android application developer. Starting with the first few chapters, the reader will begin to work with the basics of intent, and even intermediate developers will find useful tips throughout this book. As the reader progresses through the chapters, topics that are more difficult will be covered; so, it is important that beginners do not skip ahead.

A fundamental understanding of the Java programming language and Android development is suggested.

Conventions

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

Code words in the text are shown as follows: "To create an activity, we will extend our class from the `Activity` class and override the `onCreate()` method."

A block of code is set as follows:


```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == TAKE_VIDEO_CODE)
    {
        if (resultCode == -1)
        {
            Uri videoUri = data.getData();
            video.setVideoURI(videoUri);
            video.start();
        }
    }
}
```


Also, it can be in the following format:

```
public class Activity1 extends Activity {

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

New terms and **important words** are shown in bold. Words that you see on the screen in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen".

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Understanding Android

This chapter provides you with a strong theoretical concept of Android. It is obvious that the term is not alien even for any novice technology user. Because of the popularity of this great operating system, many developers started to shift from web development and other platforms. This huge migration has brought a significant change in the market of Android apps and has opened new, unlimited doors for new mobile application developers. Android is a strong opponent of iOS which is an operating system by Apple Inc. However, as statistics suggest, Android is catching up with the iOS market in terms of revenue as Google Play is the fastest growing app market in terms of total number of downloads.

This chapter includes the following topics:

- Introducing Android
- Understanding the whys and whens of Android
- Official Google IDE for Android Developers - the Android Studio
- Structure of an Android application
- Presenting the Android Activity lifecycle

Introducing Android

Android is a Linux-based operating system which makes it an open source software. Google distributed its license under the Apache License Agreement. The availability of Android code makes it an easily-modifiable operating system, which can be customized by the vendor as well. Due to a highly flexible design, some critics call it unsecure, which was right at a certain period of time, but now, Android is a mature operating system with a high-level secure architecture. It is said that the newest version of Android (that is, Jelly Bean) is the most secure operating system that Google has ever produced. Let's move forward with an overview of the different versions of the Android OS.

Exploring the different versions of Android

Since the beginning, Android has been transforming itself with the release of different versions. Not just UI but many features were added, modified, and enhanced in each upcoming version. The first version to officially use the name of a dessert was Android Cupcake 1.5, which was based on Linux 2.6.27. Every new Android version comes with a new set of API levels, which basically revises the previous API with some modification, obsolescence, and addition of new controls.

Releasing new versions of Android brings some obsolescence in the previous methods/functions from a developer's point of view. However, this will bring warnings but not errors; you can still use previous method calls in new API Levels as well.

The following table shows the different Android versions with their API Levels and major highlights:

Android version	Version name	Main features	API level	Release date
Android 4.1/4.2/4.3	Jelly Bean	Google Now Voice-to-search Lock screen widgets Speed enhancements Gesture typing in keyboard Secure USB debugging (for developers only) OpenGL ES 3.0 support Improved camera user interface Right-to-left languages support	16, 17, and 18	July 9, 2012, November 13, 2012, and July 24, 2013
Android 4.0	Ice Cream Sandwich	Major UI changes Enhanced lock screen actions Screen orientation animation Email app with EAS v14 Facial unlock Enhanced web browser Support of tablet and cell phones	14 and 15	October 19, 2011

Android version	Version name	Main features	API level	Release date
Android 3.x	Honeycomb	First OS for tablets Addition of system bar and action bar Quick access to camera and its features Two pane email UI view Multi-core support	11, 12, and 13	February 22, 2011
Android 2.3	GingerBread	Enhanced UI Native VoIP/SIP support Google Talk and Google Wallet Video call support	9 and 10	December 6, 2010
Android 2.2	Froyo	Speed improvements USB tethering JIT implementation	8	May 20, 2010
Android 2.0/2.1	Eclair	Updated UI Live wallpaper Bluetooth 2.1	5, 6, and 7	January 12, 2010
Android 1.6	Donut	Gesture recognition	4	September 15, 2009
Android 1.5	Cupcake	Text prediction in keyboard Record and watch videos	3	April 30, 2009



It is an interesting fact that the versions of Android are in alphabetical order. Starting off from Apple Pie 1.0 and then Banana Bread 1.1, it made its way towards Jelly Bean with a complete coherence of alphabetical sequence, and by maintaining the legacy; the next version expected will be Key Lime Pie.

As it is mentioned earlier that Android is open for modifications by the vendor due to its open-sourced nature, many famous mobile manufacturers put their own customized versions of Android in their phones. For example, Samsung made a custom touch interface over Android and calls it TouchWiz (Samsung Galaxy S4 comes with TouchWiz Nature UX 2.0). Similarly, HTC and Sony Xperia came up with their own custom user interface and called it HTC Sense and TimeScape respectively.



Google Play – the official app store for Android

Just like any other famous mobile operating systems, Android has its app store known as Google Play. Previously, the app store was called Android Market, which, at the start of the year 2012, became Google Play with a new-and-improved user experience. The update unified the whole entertainment world under the umbrella of Google Play. Music, apps, books, and movies, all became easily accessible to the users just like Apple's famous App Store (iTunes). You can find detailed information about the Android store at <http://play.google.com/about/>.



Google Movies & TV, Google Music, Google Books, and Google Magazines are only available in limited countries.

Google Play provides a wide range of applications, movies, e-books, and music. Recently, they also introduced the Google Play TV facility under the same app store. Talking about the application side, Google Play provides different categories in which a user can select applications. It ranges from games to comics and social apps. Users can enjoy many paid applications and can unlock many features by in-app billing services provided by Google Play.

There are different vendor specific app stores as well, such as Kindle's Amazon App Store, Nook Store, and many others that provide many applications under their own terms and conditions.

Understanding the whys and whens of Android

Android is a Linux-based open source operating system, primarily targeted for touch screen mobiles and tablets. Andy Rubin, Rich Miner, Nick Sears, and Chris White founded the operating system in October 2003. The basic intention behind the idea of Android was to develop an operating system for digital content. This was because, at that time, mobiles were using Symbian and Windows Mobile as their operating systems.



iPhone was released in June 2007 by Apple Inc. Android was released in November 2007 by Google Inc.



However, when they realized that there is not much of a market for devices such as cameras, they diverted their attention to mobile phones against Symbian and Windows Mobile. iPhone was not on the market then. Android Inc., a top brand for smart phone operating systems covering 75 percent of market share as of today in smartphones, was running secretly at that time. They revealed nothing to the market except that they were working on software for mobile phones. That same year, Rubin, the co-founder of Android, ran out of money, and his close friend, Steve Perlman, brought him \$10,000 cash in an envelope.

In August 2005, Google Inc. acquired Android Inc., making it a subsidiary of Google Inc. The primary employees of Android stayed in Android Inc. after acquisition. Andy Rubin developed a mobile device platform powered by Linux Kernel. Handset makers and carriers were being promised a flexible and upgradeable operating system by Google. As Google was not releasing any news about Android in the media, rumors started to spread around. Speculations spreading around included Google is developing Google branded handsets and Google is defining cell phone prototypes and technical specifications. These speculations and rumors continued until December 2006.

Later, in November 2007, Open Handset Alliance revealed that their goal was to develop an open standard for mobile devices. Android was released as its first product; a mobile device platform built on Linux Kernel Version 2.6. Open Handset Alliance is a consortium of 65 companies involved in mobile space advocating open source standards for the mobile industry.

In October 2008, the very first commercially available phone deploying Android operating system was released by HTC, called HTC Dream. The following image shows HTC Dream. Since then Android is being upgraded. Google launched its nexus series in 2010.




HTC Dream, the First Android phone using Android Activity back stack

The evolution of Android OS

After the first appearance of Android OS in HTC Dream, it gained rapid popularity among consumers. Android is continuously being upgraded by Google. Each major release includes bug fixes from the last release and new features.

Android released its first version in September 2008 in the device HTC Hero. Android 1.1 was an update tweaking bugs and issues, with no major release. After Android 1.1, Android 1.5 named Cupcake, was released with features such as video uploading, text prediction, and so on. Android 1.6 Donut and Android 2.0/2.1 Éclair released at the end of 2009, followed by 2.1 in January 2010, introduced major updates such as Google Maps, enhanced photo video capabilities, Bluetooth, multi-touch support, live wallpapers, and more. In May 2010, Android 2.2 named as Frozen Yogurt, or Froyo, was the major release, adding support for Wi-Fi hotspot connectivity.

This version became very popular among developers, and is used to be the minimum API level for android apps. Android 2.3 Gingerbread, released in May 2010 introduced the Near Field Communication (NFC) capability, which allowed users to perform tasks such as mobile payments and data exchange. This version of Android became the most popular version among developers. Android 3.0/3.1 Honeycomb, was specially optimized for tablet devices, and more UI control for developers was a big plus. Android 4.0 Ice Cream Sandwich was released in October 2011. Since Android 3.0/3.1 was only for tablets, the Ice Cream Sandwich release overhauled the gap, and was supported by both mobile phones and tablets. The latest release of Android, Android 4.2 Jelly Bean further polished the UI, refined the software, among other improvements.

[ Google started naming Android versions after sugar treats, in alphabetical order, after Android 1.1 version.]

The following image shows all the versions in a visual format:



The following screenshot shows the current distribution (March 2013) of Android versions. It is clear from the screenshot that Android 2.3 Gingerbread is the most popular version, followed by Android Ice Cream 4.0:

Android SDK version	Current market share	Change in the last 30 days
2.1 (Eclair)	3.1 %	↑ 12 %
2.2 (Froyo)	4.2 %	↓ 25 %
2.3 (Gingerbread)	23.6 %	↓ 9 %
3.0-3.2 (Honeycomb)	0.7 %	↑ 2 %
4.0.x (ICS)	14.6 %	↓ 4 %
4.1-4.3 (Jelly Bean)	50.9 %	↑ 3 %
4.4 (KitKat)	2.9 %	Breakout

Current distributions of Android versions

Official IDE from Google – the Android Studio

Before Google I/O 2013, Android was officially using Eclipse as an IDE for its development. Official Android Support clearly mentioned about the use of this IDE along with the **Android Development Tools (ADT)** and **Android Software Development Kit (SDK)** with its documentation.



Loading Screen for Android Studio (Windows 7)

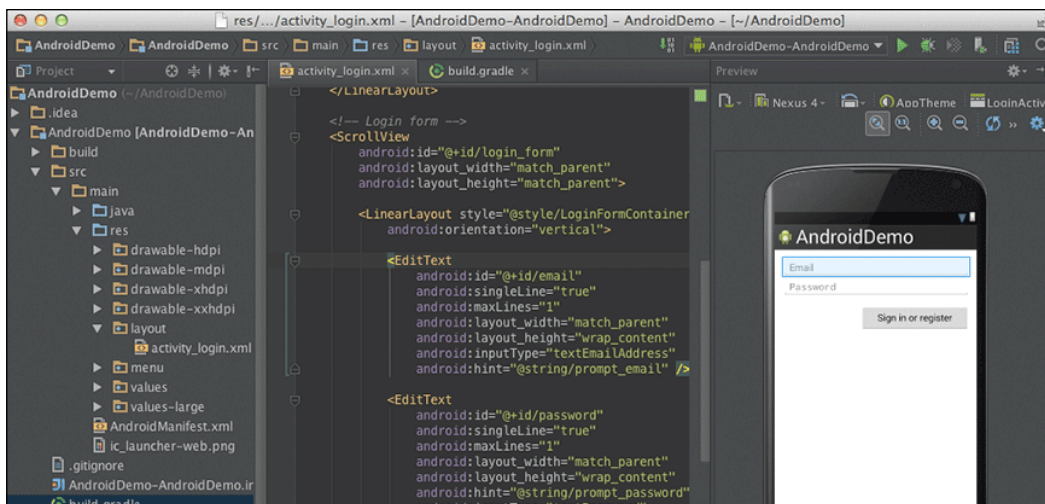
In Google I/O 2013, Google came up with a new IDE that is specially designed for the development of Android Apps. The IDE is called Android Studio, which is an IntelliJ-based software that provides promising features to the developers.


Features of Android Studio

Android Studio gives various features on top of an IntelliJ-based IDE. The list of features that are introduced in Android Studio is as follows:

- Android Studio comes with built-in Android Development Tools
- Android Studio gives Gradle-based support for the build
- Flexible controls for building an Android UI and simultaneous views on different screen sizes
- Android refactoring, quick fixes, and tips and tricks
- Advance UI maker for Android apps with drag-and-drop functionality

The following screenshot shows the Android Studio multi-screen viewer with UI maker:



 The current version of Android Studio is v0.1.1.

Apart from that, there are various other features that are offered by Android Studio. Google mentioned in the launch that the version (v0.1) is unstable and needs various fixes before it can be used with its 100 percent accuracy.

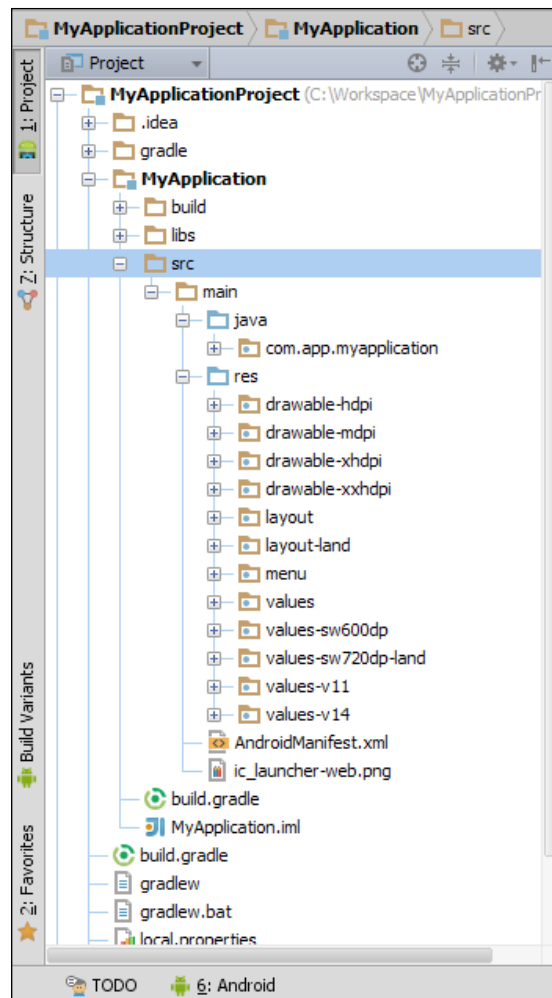
Limitations of Android Studio

Android Studio is in the early phase, which makes it an immature software with limitations. According to Google, they are working on the updates of the software and soon will rectify the issues. As per Version 0.1.1, the limitations faced by the developers are as follows:

- Android Studio can only be compiled with Android 4.2 Jelly Bean
- The user interface can only be made with Android 4.2 Jelly Bean UIs and widgets
- An Eclipse project cannot be directly imported on Android Studio (refer to <http://developers.android.com/>)
- Bugs in importing library projects

Building blocks of an Android application

An Android application consists of various building blocks that help developers to keep things organized. It gives flexibility to maintain assets, pictures, animations, movie clips, and implement the localization functionality. Moreover, there are some components that contain the information regarding the minimum and maximum versions of Android that your application supports. Similarly, menus are separately handled in Android application projects.



Various components of an Android application as shown in Android Studio


Just like Eclipse IDE, Android Studio gives various handy functionalities to play with these features. Looking forward to the building blocks of the Android application, we can classify the components into the following parts:

- Coding components
- Media components
- XML components
- Referencing components
- Library components

Coding components

Breaking into components brings an easy understanding of the structure of an Android application. Coding components are those that directly relate to the source code of an Android project. In order to write an application, a developer needs to write some lines of code that will respond in the way the user wants.

In coding components, the main folder that holds all of the developer's code is `src`. The folder consists of one or more Java packages in which developers classify their code in accordance with the type of work done. The default way to write a package name is dot separated (for example, `com.app.myapplicationproject`), which can easily distinguish it from any other package of any other project.

 The Android application's package name is used to identify it uniquely on Google Play.

Inside the packages there are `.java` files that are present for the developer to reference from the Android library and proceed to the desirable output. These Java classes may be or may not be inherited from the Android API. We can also use most of the Java functions in writing our code.

Media components

Due to highly configured hardware, users need applications with good graphics, animations, sounds, and video files. Hence, you can easily introduce any of them but it should be made sure that none of them should affect the quality of the app as there are thousands of different types of Android devices available. Android provides a flexible method that you can use to place your media files within the project. By classification, there are two ways of maintaining your media files inside an application project:

- Assets folder
- Res folder

The assets folder

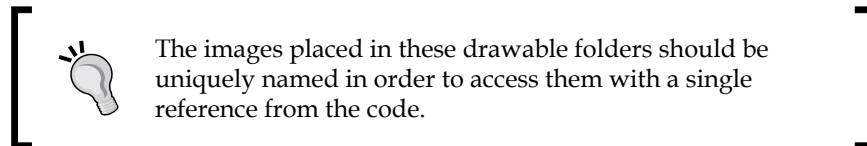
An Android project contains a folder named `assets`. This folder is responsible for holding all of the media files, including music, images, and so on. The developer can directly access the folder from the code by writing the `getAssets()` function within the inherited `Activity` class. This function returns the `AssetManager` that can easily be used to access the subfolders and files inside the main `assets` folder.

The main advantage of the `assets` folder is that there is no need to keep references for the files placed, which is very handy in the situation where the developer needs to do a test and make a runtime change. Though it does not have any reference, it may introduce errors due to typing mistakes. Another advantage of using `assets` is that the developer can arrange folders according to his or her will; similarly, the naming conventions for these folders can easily be chosen according to the ease of the developer.

The res folder

The `res` folder is used to manage an application's resources such as media files, images, user interface layouts, menus, animation, colors, and strings (text) in an Android application; or in other words, you can say that this is the most intelligent way of handling the media files. It consists of many subfolders including `drawable`, `drawable-ldpi`, `drawable-mdpi`, `drawable-hdpi`, `drawable-xhdpi`, `drawable-xxhdpi`, `raw`, `layout`, `anim`, `menu`, and `values`.

Drawable is directly related to the images that are used in the Android project. It is an intelligent way of keeping images in the project. As we know that there are various types of devices present in the market that support Android OS. In order to differentiate between these devices, the low resolution images are placed in the `ldpi` folder for the devices with less resolution. Similarly, the `mdpi` folder is for the device with medium screen density, `hdpi` for high density, `xhdpi` for extra high density, and so on.



Similarly, for placing music and sound contents, we use the `raw` folder in order to access them from the code. Any other file apart from the music and sound can also be placed in the `raw` folder (for example, the JSON file). The same goes with `anim`, `values`, `menus`, and `layout` folders for placing the animations, values, custom menus, and different types of layouts respectively.

XML components

In Android, a developer needs to use XML in order to make the user interface. Layouts, Menus, Sub Menus, and many other things are defined in the form of different Android tags based on XML. Apart from layouts, you can also store strings, color codes, and many other things in the form of XML files. The component supports the maintenance of the hierarchy of the application and makes it easy to understand for all developers.

Let's take a look at some of the most important XML files that are used as the backbone of any Android application.

The layout folder

Inside the `res` folder, there is a folder called `layout` that contains all the layouts of activities. It is to be noted that there are some extensions of this folder, just like the drawable folders. The `layout-land` and `layout-port` methods are specifically used for keeping the layout well organized in landscape and portrait mode respectively.



XML can also be used for making custom drawables that can be used as images in different scenarios. For example, the image of the custom button can be made with XML, which gives a different UI behavior on clicked and non-clicked states.

The preceding screenshot is of Android Studio where you can see an `activity_main.xml` file that is used to describe the layout of an activity. There are some Android-defined XML tags for `RelativeLayout` and `TextView` (read the following information box). Similarly, there are some other tags as well that are available for the developer to include different kinds of widgets in the layout.



`RelativeLayout` is a layout in which children are placed to the relative positions. This layout is often used by Android mobile developers.

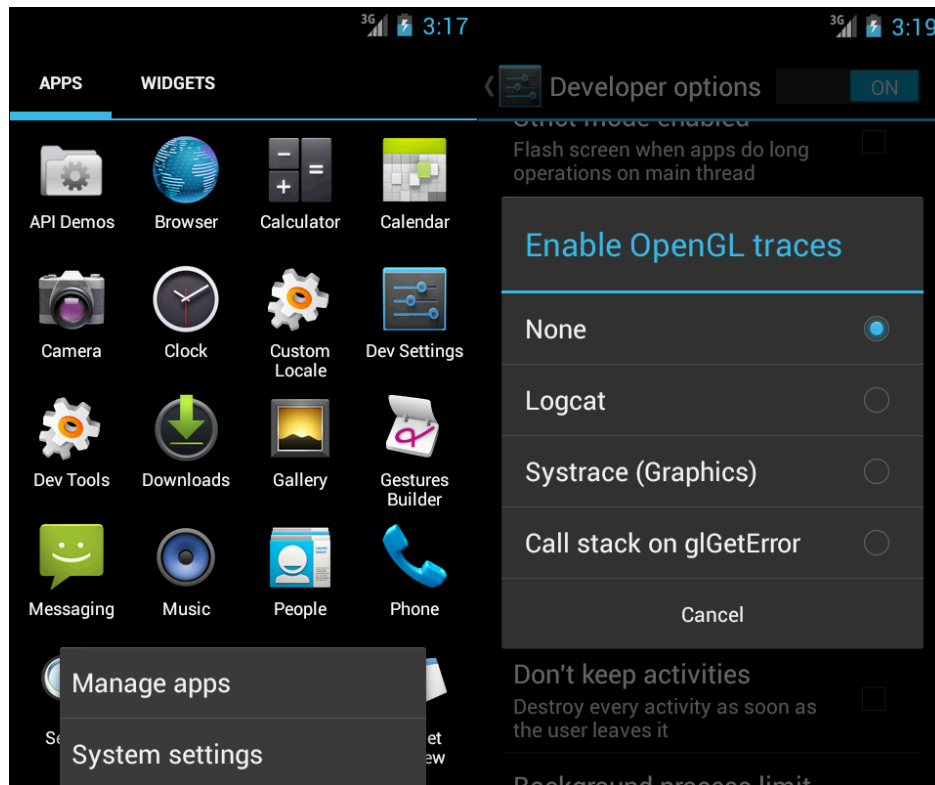
`TextView` is one of the views that is used to display any kind of text including numbers, strings, and editables.

The menu folder

Android comes with different kind of menus that can be used in order to give quick access to the prominent functionalities that are used within an activity. The different menus available are as follows:

- Context menus
- Options menus (with an action bar)
- Pop-up menus
- Custom menus

Due to the limited focus of this chapter, we cannot completely elaborate on the functionality and give examples of the different types of menus. However, all types of menus are based on XML files in which Android-defined tags such as `<menu>`, `<item>`, and `<group>` are used to introduce menus in the application. See the following screenshot for reference:



The Android ICS Options menu is on left and the Custom Pop Up menu is on the right

The values folder

The `values` folder consists of various XML files that can be used by the developer in many scenarios. The most common files for this folder are `styles.xml` and `strings.xml`. The `style` file consists of all the tags that are related to the style of any UI. Similarly, the `strings.xml` file consists of all the strings that are used in the source code of any Android project. Apart from that, the `strings.xml` file also contains the `<color>` tagged hash-coding, which is used to identify many colors inside the source code of an Android application.

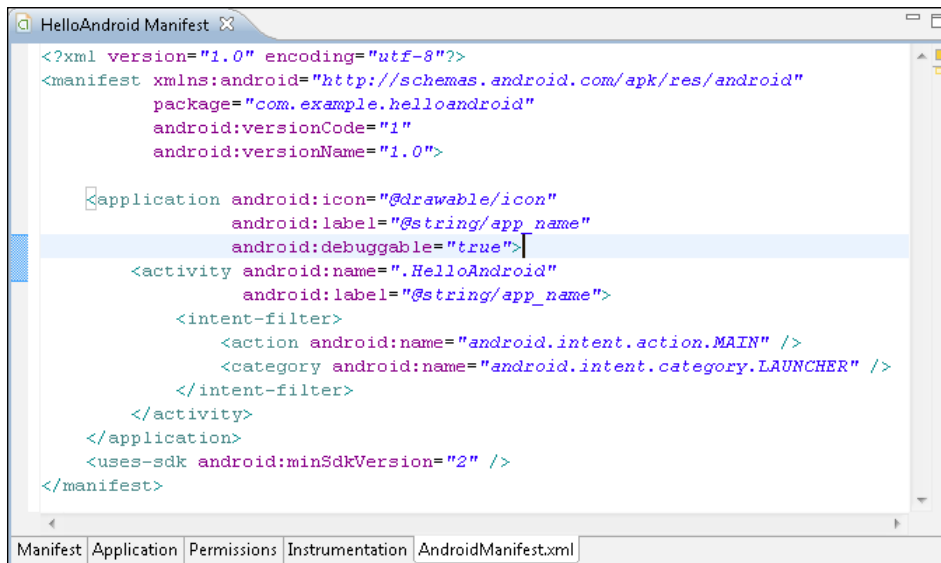
AndroidManifest.xml

Unlike the previously mentioned folders, `AndroidManifest.xml` is a file that contains important information about the Android application. The manifest file consists of various tags such as `<application>`, `<uses-sdk>`, `<activity>`, `<intent-filter>`, `<service>`, and many other tags that are enclosed within the main tag of `<manifest>`.

Just like the tags suggest, this XML file contains all the information about activities, services, SDK versions, and everything that is related to the application. There are various errors that may arise if you don't enter the correct information or miss anything in the `AndroidManifest.xml` file.

Another major advantage of the `AndroidManifest.xml` file is that it is the best way to track the structure of any Android application. The total number of activities, services, and receivers can be seen easily by this file. Apart from that, we can change the styles, fonts, SDK constraints, screen-size restrictions, and many other features just by tweaking the `AndroidManifest.xml` file.

At the time of signing the `.apk` build, we mention the package name, version name, and version code, which are uniquely identified by the Google Play in order to put the application on the market. The application will then be identified by this package name and further releases are based on changing the version codes and version name described inside the `AndroidManifest.xml` file.



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

    <application android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:debuggable="true">
        <activity android:name=".HelloAndroid"
            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>
    <uses-sdk android:minSdkVersion="2" />
</manifest>
```

Referencing components

Another basic component of an Android application is the referencing component. Put simply this component helps XML-based files to interact with the Java code. In Android Studio, the file `R.java` is placed under the source folder, which is the child of the build folder in the project hierarchy. The `R.java` file consists of all the references that are used in the XML files for layout, menus, drawables, anim, and so on. This file is then exposed to the activity files to get the references and obtain the objects to perform various functions and parameters.

Mostly, this `R.java` file is obtained as a part of the project import and used as `R.layout.main`. In this example, it clearly means that we need to obtain a layout that is a part of the `res` layout folder and the name of the layout is `main`. As a result, it will return a resource ID, which is hidden from the developer and directly referenced to the particular layout inside the `res` folder.



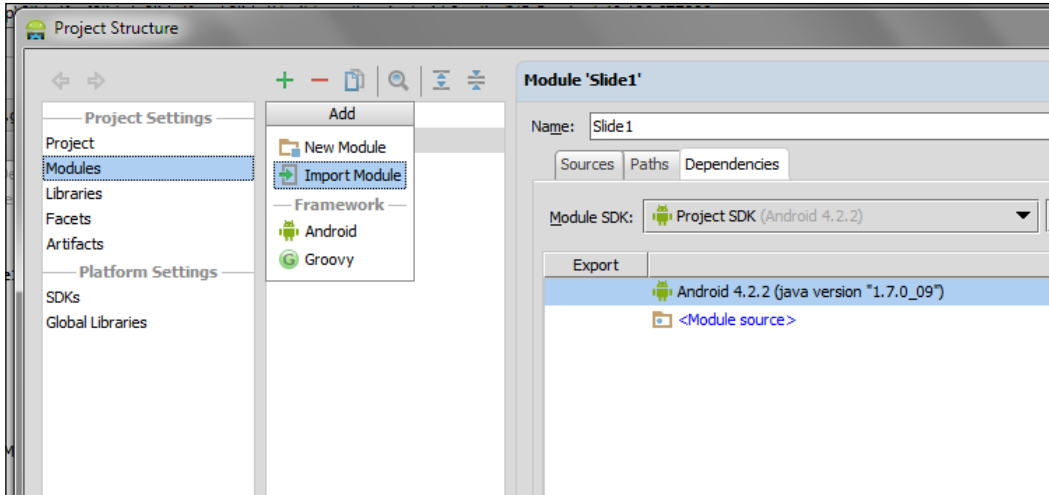
The `R.java` file is automatically generated while building the project. Hence, it should not be pushed into the repository. Ensure the content of the `R.java` file is not modified manually. The `R.java` file that exists in the `gen` folder of your project is defined by Android at the time of project making or compiling.

Library components

Libraries are pre-built Java files/projects that can be used by anyone to perform certain tasks inside this application. There are various third-party paid/unpaid libraries available that give various functionalities to the developer. Library components are not libraries themselves; rather, they are the project folders in which the libraries are kept.

In an Android project, a folder named `libs` is present inside the main application folder (Android Studio), which is used as a library component. Any `.jar` library file can be put under this folder in order to reference it from the code. While using those libraries inside the Java code, you need to import the corresponding package name that is present inside the `.jar` file in order to use the functions of that particular class.

Similarly, you can use any other Android project as a library by making it a module and importing it inside your project. This functionality was previously called as Library Project in Eclipse, imported by **Project Properties | Android | Library Reference**.



The Android Studio module importing window

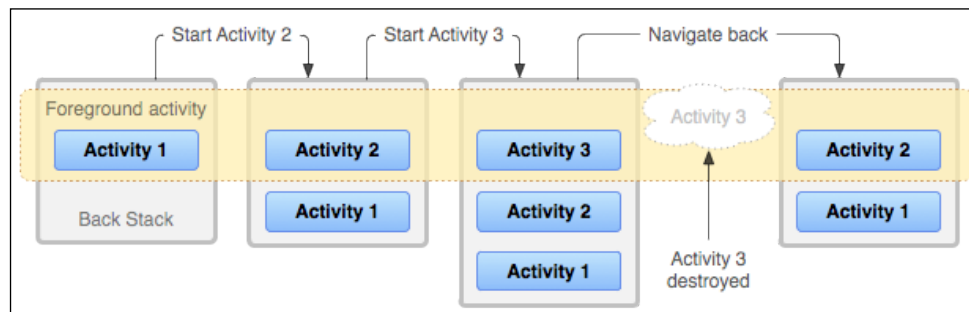
Android Activity lifecycle

An Android application consists of one or more activities. These activities are visual representations of an application in transitioning flow while performing the task, taking user inputs, and showing results to the user. Each activity presents the user with a visual representation on the screen for user interaction. Android keeps all the activities in a back stack following the last in, first out rule. Whenever a new activity is started, the current activity is pushed in the back stack. Thus, Android gives focus focuses on the new activity. The activity can take up the whole screen of the device, or it can also take part of the screen, or it can be dragged as well. Whether it is an activity taking the whole area of a screen or a small part of screen, only one activity is focused at a time in Android. When, any existing activity is stopped, it is pushed into the back stack, which in turn results the next top activity being focused.



Android 4.x versions introduced fragments. Fragments can be referred to as sub-activities, which are embedded in an activity to perform different tasks in a single activity at the same time, unlike activities.

Usually, an Android application consists of more than one activity. These activities are loosely bounded with each other. It is a good practice to create each activity for a specific task to be performed. For example, in a simple phone dialing application, there might be one activity to show all contacts, one to show full contact details of any specific contact, one for dialing a number, and so on. In all the applications, there is a main activity that behaves as the starting point of the application. This activity starts when the application is launched. Then this activity starts some other activity, which starts another, and so on. Android manages all the activities in a back stack.



Android Activity back stack

The previous figure shows a simple representation of how back stack works. The area highlighting top activities in a stack represents foreground activity, sometimes called focused activity or running activity. When a new activity is created, it is pushed in the stack, and when any existing activity is destroyed, it is pulled out of the stack. This process of being pushed in the stack and pulled out of the stack is managed by the activity lifecycle in Android. This lifecycle is called Activity lifecycle. The lifecycle manages the activities in the stack and notifies about the changes in the state in the activities through the callback methods of the cycle. An activity receives different types of states such as activity created, activity destroyed, and so on, due to change in the state. A developer overrides these callback methods to perform the necessary steps for respective change of state. For example, when an activity is started, the necessary resources should be loaded, or when an activity is destroyed, those resources should be unloaded for better performance of the app. All these callback methods play a crucial role in managing the Activity lifecycle. It is the developer's choice to override none, some, or all methods.

Fundamental states of an activity

Basically, an activity remains in three states: Resumed, Paused, and Stopped. When an activity is resumed, it is shown on the screen and gets the focus of the user. This activity remains in the foreground section of the back stack. When another activity is started and it becomes visible on the screen, then this activity is paused. This activity still remains on the foreground task, and it is still alive, but it has not gotten any user focus. It is also possible that the new activity partially covers the screen. In that case, the part of the paused activity will be visible on the screen. The activity comes in the Stopped state when it becomes completely invisible from the screen, and is replaced by another activity in the foreground. In this stopped state, the activity is still alive, but it is in the background section of the back stack. The difference between the paused and stopped states is that, in the paused state, the activity is attached to the window manager, but in the stopped state, it is not attached to the window manager.

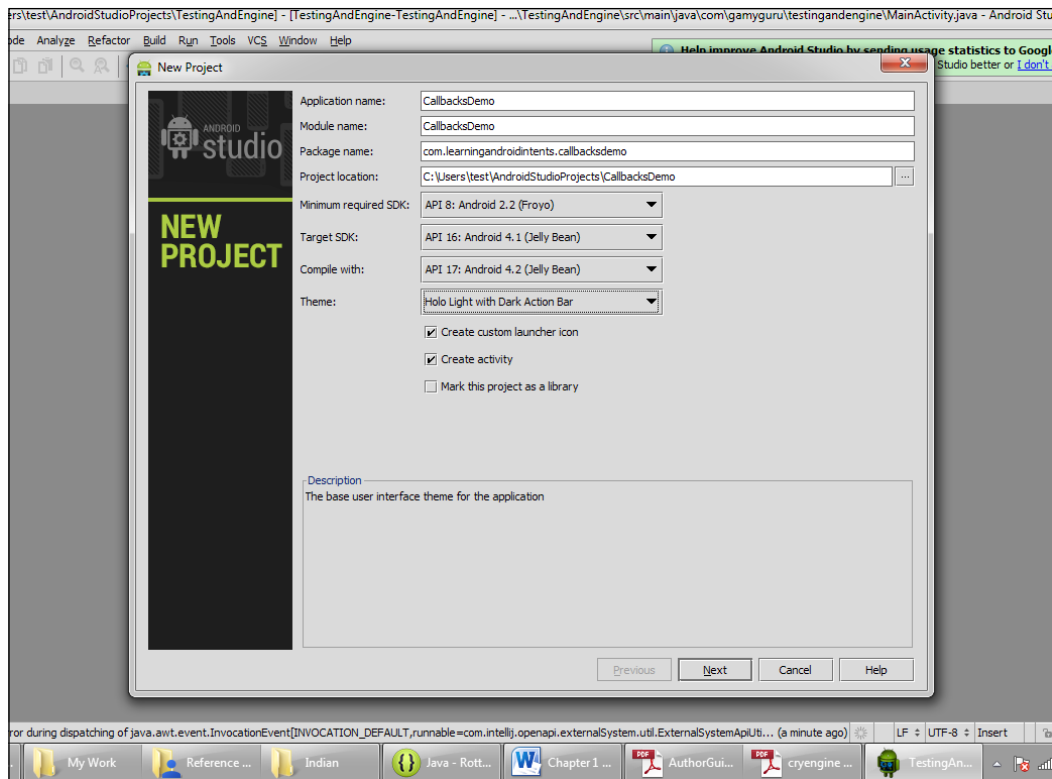


In an extremely low memory situation, an Android system can kill any paused or stopped activity by asking to finish it, or without asking by killing the process. To avoid this problem, the developer should store all the necessary data in a pause and stop callback, and should retrieve this data in the resume callback.

The callback methods of the Activity lifecycle

There are various callback methods that are called when the state of any activity is changed. Developers perform the necessary tasks and actions in these methods for better performance of the app. To show the Activity lifecycle in action, we are creating a small Android application in this section. Here is the step-by-step approach:

1. Start **Android Studio**.
2. Create an empty project with the details as shown in the following screenshot:



New Project Dialog in Android Studio

3. Add the following code in the MainActivity.java file of the project:

```
package com.learningandroidintents.callbacksdemo;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.Toast;
public class MainActivity extends Activity {

    @Override
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Toast.makeText( this, "Activity Created!",
            Toast.LENGTH_SHORT
        ).show();
    }
}
```

```
@Override
    protected void onStart ()
    {
        super.onStart ();
        Toast.makeText(this, "Activity Started!",
            Toast.LENGTH_SHORT
        ).show();
    }

@Override
    protected void onResume()
    {
        super.onResume ();
        Toast.makeText(this, "Activity Resumed!",
            Toast.LENGTH_SHORT
        ).show();
    }

@Override
    protected void onPause()
    {
        super.onPause ();
        Toast.makeText(this, "Activity Paused!",
            Toast.LENGTH_SHORT
        ).show();
    }

@Override
    protected void onStop()
    {
        super.onStop ();
        Toast.makeText(this, "Activity Stopped!",
            Toast.LENGTH_SHORT
        ).show();
    }

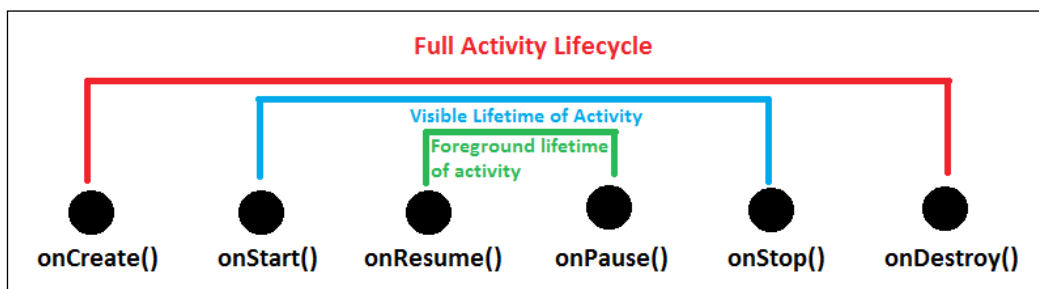
@Override
    protected void onDestroy()
    {
```

```
        super.onDestroy();
        Toast.makeText(this, "Activity Destroyed!",
            Toast.LENGTH_SHORT
        ).show();
    }
}
```

4. Run the project in the emulator, and you will see toasts being printed on screen in the following order:
 - Activity Created
 - Activity Started
 - Activity Resumed
 - Activity Paused
 - Activity Stopped
 - Activity Destroyed

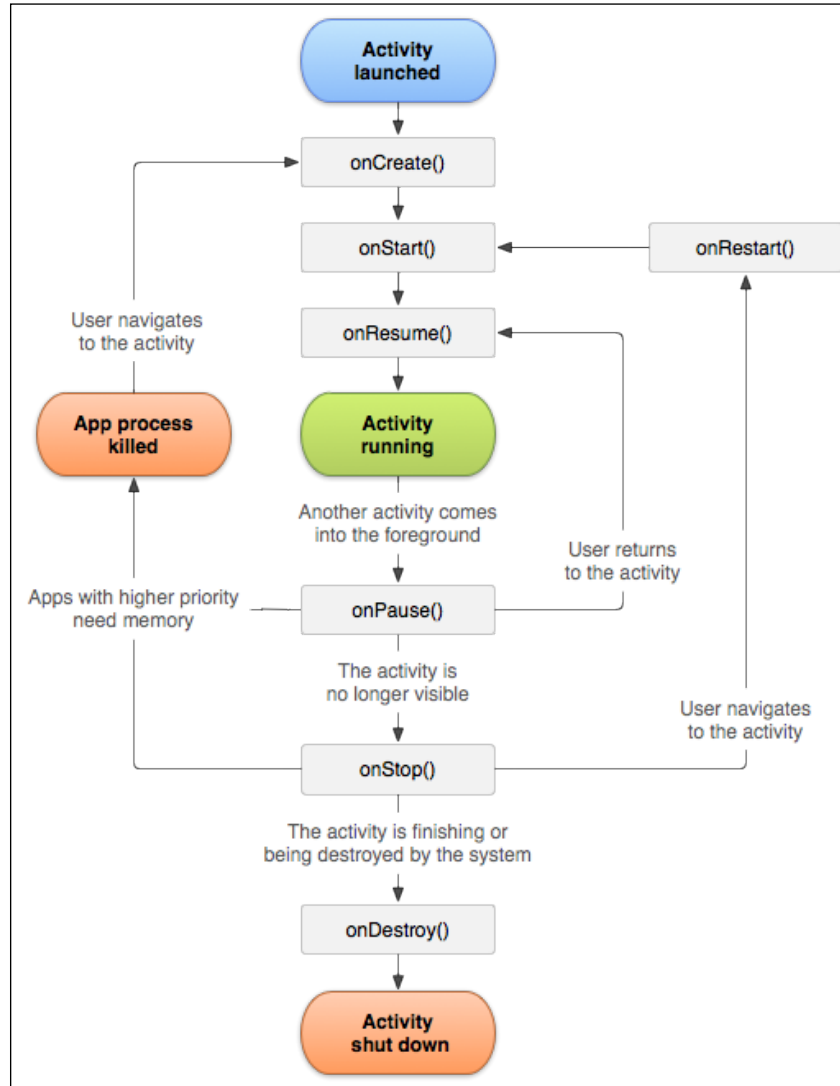
Let us see the working of the previously mentioned code.

When you run the project, the emulator will display all the toasts in the previously given order on the screen. At the start of project, an activity is created, and then the activity is started. After starting the activity, it is displayed on the screen and emulator prints **Resumed**. Now, we go back by pressing the back key, and the Android system prepares to finish the activity. So, the activity is first paused, then it is stopped, and finally it is destroyed. All these callbacks together are called the Activity lifecycle. Activity lifecycle starts from the `onCreate()` method and it stops at the `onStop()` method. The activity is visible from the `onStart()` method to the `onStop()` method, and the activity remains in foreground from the `onResume()` method to the `onPause()` method. The following figure shows this cycle distribution:



The activity lifecycle flow


Until now, we have discussed the lifecycle callback methods used, their states, and their purpose. Now, we will look into the callback method's flow. In Android, when one activity is started, the already opened activity is stopped, and this change of activity happens in a flow. The following figure shows the visual flowchart of the Activity lifecycle:



Callback methods are shown with rectangles. The very first step in the Activity lifecycle is to create an activity. Android creates an activity, if no instance of that activity is running in the same task. The `noHistory` tag does not allow multiple activities; rather it will determine whether an activity will have historical trace or not (refer to <http://developer.android.com/guide/topics/manifest/activity-element.html#nohist>), where you can determine multiple instances by the `android:launchmode` flag tag. Making this tag's value `true` means only one instance of the activity will be created in the stack, and whenever an activity intent is called, the same instance is pushed on top of the stack to show the activity on screen.

After the `onCreate()` method, the `onStart()` method is called. This method is responsible for the initial settings, but it is best practice configure these in the `onResume()` method, which is called after the `onStart()` method. Remember, the foreground phase is started from the `onResume()` method. Say a user gets a call on his or her phone, then this activity will be paused through the `onPause()` method. So, all the steps involved in storing the necessary data when the activity is paused should be done here. This method can be very beneficial in critical memory situations because in these situations, Android can stop the paused activities, which in turn can show unexpected behavior in the app. If the activity is killed due to a critical memory situation, the `onCreate()` method is called instead of the `onResume()` method, resulting in the creation of a new instance of the activity.

But, if everything goes right, then the activity returns to its same state through the `onResume()` method. In this method, the user can do all the work of reloading the stored data in the `onPause()` method, and can get the activity back to life. On turning off the activity after `onResume()` is launched the `onStop()` method is called. This triggers either the `onRestart()` method, or the `onDestroy()` method depending on user action. In a nutshell, the developer can control the Activity lifecycle using callback methods. It is a good practice to use the `onPause()` and `onResume()` methods for data management, whether the activity remains foreground or not, and `onCreate()` and `onDestroy()` should be used for only initial data management and cleaning up the resources respectively.

 All callback methods except the `onCreate()` method take no parameter or argument. In case of a critical memory situation, if an activity is destroyed, then that instance state is passed in the `onCreate()` method at the time of creation of that activity.

It is not necessary to override all the methods. The user can override any number of methods as there is no such restriction on it. The user should set a view in the `onCreate()` method. If you don't set any view for the content, a blank screen will show up. In each callback, first of all, the same callback method of the superclass should be called before doing anything. This super callback method operates the Activity lifecycle through standard flow developed by Android systems.

Summary

In this chapter, we explored the key concepts of Android. We discussed about Android, its versions that are named after sugar treats, covered a brief history of Android, and how its founders released Android with Google. We also discussed Google Play, an official store for Android apps; Android Studio, an official IDE from Google, and its features and limitations. Then we moved our discussion to a development perspective, and we discussed the building blocks for any Android application. We also discussed the Activity lifecycle, which plays a very important role in any Android application, its flow, its callback methods, and and looked at an example of it.

In the next chapter, we will discuss the intents, role played by the intents in Android, a technical overview, structure, and its uses in Android.

2

Introduction to Android Intents

Revising the previous lesson – Android Activity is the visual representation of controls, widgets and many other things with which the user interacts. An Android application is a combination of many activities which interact with each other in order to perform a single or multiple tasks for which the application is dedicated to. Mostly, there is only a single activity that is shown on the screen at a particular time. Some actions (like button tap or a gesture) may result in navigating from the current activity to a new one on the top of the Activity Stack.

Android Intents help developers to perform interaction between two activities, yet this is not the only thing that an intent does. This interaction includes moving from one activity to another, pushing data from one activity to another and bringing results after the closure of any particular activity. In short, it can be said that intent is an abstract term in android referring to any task that is to be performed. There are various other things which will be explored with the passage of time as you read through this book.

This chapter includes the following topics:

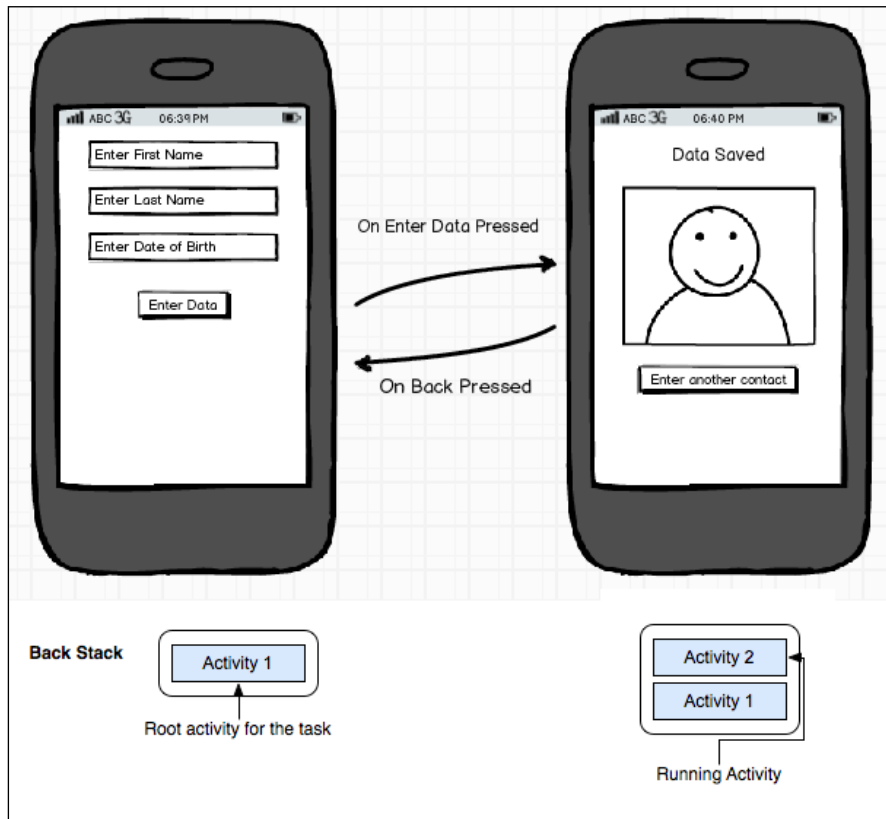
- Role of intents in an Android Application
- Intent - a technical overview
- Structure of an intent



The concepts of Android Activity, Activity Lifecycle, and Activity Stack, as discussed in the previous chapter, are the prerequisites for understanding this chapter and the chapters ahead. If you don't have the basic concept of these things, we would recommend that you read *Chapter 1, Understanding Android*, in order to move forward.

Role of intents in an Android Application

In this section, we will see what the scope of Android Intents is. Till now, we have a complete view of why the activities are required and why it is necessary to maintain and trace the flow from one activity to another.



Navigation between different activities on button tap (using intents) and its representation by an activity stack

It can be said that this portion of the book is the summary of the benefits that we can take from the Android Intents. The scope lies in the various factors of Android Activities, services, data transfer, and many other things. We will see this in the following list:

- Activity transition from one activity to another
- Data transfer from one activity to another
- Making connection with Wi-Fi and Bluetooth
- Accessing Android Camera

- GPS sensor to get current location
- Sending SMS and MMS
- Customizing mobile calls
- Sending e-mails and social media posts
- Starting and controlling Android Services
- Handling broadcast messages
- Changing time zones
- Notification bar alerts
- And many more

We will now take a look at each of the key roles of Android Intents. A short description on these main features of Android Intents is given in the following sections.

Role of intents in Android Activities

The most important and extensive use of intents can be seen in Android Activities. The Android Application consists of many activities and to transit between those activities, we need to use Android Intents. In the previous figure, you can see that in **Activity 1** (on the left-hand side) when the content is filled and the user taps the **Enter Data** button, Android will use an intent to navigate to the **Activity 2** (on the right-hand side).

Apart from the previously mentioned role of intent, it can also be used to call other applications such as a browser (with a certain website from your activity) and an e-mail client (such as Gmail or any other with proper subject and e-mail body from the activity, by sending it in a bundle).

Role of intents in data transfer between activities

It is now clear that we use intents to navigate from one activity to another. But as we all know the huge role of data in an Android Application, where the user needs to fetch, manipulate, and show data in order to perform a certain task. The handling of that data and its secure transfer from one activity to another is yet another purpose of Android Intents.

In the previous figure, once the user has filled the form in Activity 1, on tapping the **Enter Data** button, intents will perform two tasks. One task is to take the user from one activity to another and second task is to transfer the filled data to the next activity in order to show/calculate the result.

Role of intents in Wi-Fi and Bluetooth transfer

While inside the application, if you want to implement a feature which gives a facility to change the current Wi-Fi/Bluetooth connection, you need to use Android Intents. With Android Intents, you can easily provide the internal interface which will let the user switch between the Wi-Fi and Bluetooth connection while remaining inside the application.

Role of intents in Android Camera

Android hardware can be of a huge importance when you talk about Android Application. The use of these components can be a fundamental part of your Android Application. Suppose the example of a 1D or 2D bar code reader, the application needs to scan the bar code and decode it in order to extract the information. This action can only be performed by opening the camera from inside the application. The opening of camera is also handled by Android Intents.

Role of intents in GPS Sensor

The Android Application market is doing marvel in many categories. Various kinds of Android Applications are present in the market today, this includes location-based applications which perform various tasks by tracking the location of the user. Through intents a developer can easily extract the current location of the user as required for calculations.

Role of intents in sending SMS/MMS

Android Intents can be used in order to enable your application for sending SMS/MMS. This task can be done by setting the SMS/MMS body from your activity and set it in bundle to call the native built-in application for sending SMS/MMS of the mobile. This SMS/MMS sending functionality can then be enhanced by implementing a Broadcast Receiver, which enables your activity to know when the message was sent or when it was delivered.

Role of intents in Mobile Calls

Any condition that will initiate a mobile call in an application can be fulfilled by Android Intents. The Android Application will use the built-in application to start the call to any particular number that will be provided in the form of data in an intent.

Role of intents in e-mail and social network posts

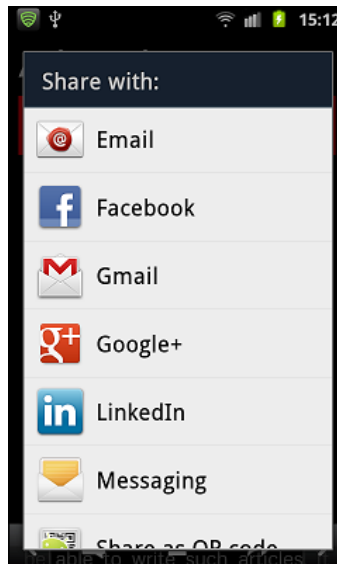
Accessing Gmail to send e-mail from your application is managed by Android Intents. There is an intent call in which we put the recipient's e-mail, attachments, body, and subject of the e-mail and start the intent on the activity. This will open the native Gmail application with those parameters filled and the user can send this to the desired recipient.

Similarly, we can send various social network updates such as Facebook, Twitter, and Google+ through an intent. This task is done using Share Intent. We put the content in the form of text or bundle and send it via Android Intent. Android then opens all the available sharing applications (as shown in the previous figure) and gives access to the user in order to pick the best application for sharing. The only condition is that there should be a preinstalled application through which the Android Intent interacts and sends the desired post is given as follows:

```
Intent sharingIntent =
    new Intent(android.content.Intent.ACTION_SEND);
```

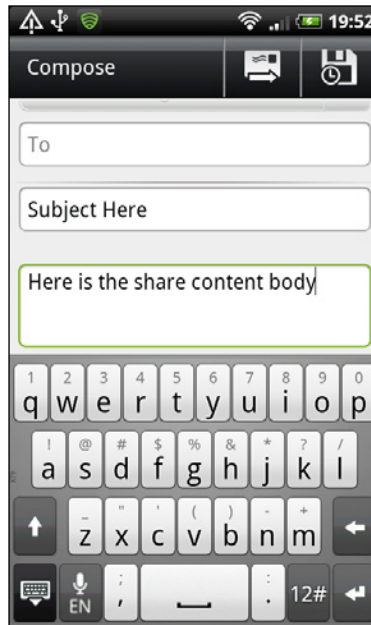


Android Intent does not disclose any other feature of social networking apart from posting on the wall or tweeting it to the timeline. To observe the complete set of features there are many third party APIs present on the Internet as free-license software or paid.



Using intent to share posts on various platforms using Share Intents

The next screenshot shows what happens when you give a functionality to the user and for example, the user wants to share it via Gmail. In this case, the screen appears with your content in it and will look as shown in the following screenshot:



Gmail interface after passing the data from Share Intent

Role of intents in Android Services

Same as Android Activity, intents are used to initiate **Android Services**. Android Services are basically the long-running tasks done by the Android Application without affecting the user interface. This background task can continue running even after the user switches the application. In other words, services can or cannot be bound from the activity. The services can work independently in order to perform a task. However in each case, an intent is used to initiate the services.

Role of intent in Broadcast Receiver

Intent has a wide usage with Broadcast Receiver. Broadcast Receiver is used to respond to the broadcast messages initiated by any other application or even the system. In this context, we catch the Android message and extract the data in order to show it in our application. For example, `Intent.ACTION_BOOT_COMPLETED` is used when we need to receive a signal when the system boot is completed. Similarly, many other intent values and intent objects can be used at different points of application in order to perform various tasks related to Broadcast Receiver.

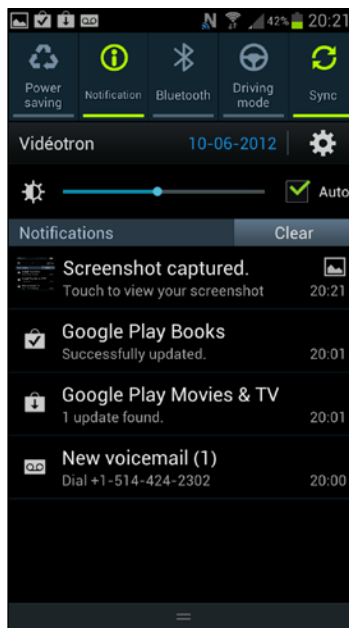
Another example can be sending the SMS/MMS where you can make a Broadcast Receiver in order to see whether the sending is completed or the message is delivered or not.

Role of intent in time zones

It is possible that your application might do something that is related to the time zone. Once the time zone has changed while you are traveling, you want your application to behave differently. In such cases, we can use Broadcast Receiver to detect the change of the time zone, get data from the intent in order to access the current time zone, and perform a certain task. This is a very handy way to maintain your application structure and data according to your time zone.

Role of intent in Status Bar

Android Status Bar is used to provide instant notification to a user without occupying too much space on the screen. The notification panel that slides from top to bottom has many features in it, such as some quick access items like wireless connection manager (currently available in few mobile phones only) and other things. We can put a notification in that bar in order to inform the user about anything. Android Intents are used in order to place the content and provide a status bar notification in it.



Android Notification Panel and Notifications

Intent – a technical overview


We have been through the theoretical overview of Android Intents in the past couple of topics. Let's take a deeper look at the technicalities of this feature of Android. In this portion, we will see the bigger picture of Android Intents which includes the example code and its explanation.

Technically, Android Intent consists of two components and both of them work independently. These two components are as follows:

- The Coding component
- The XML component

The Coding component

Android Intents are implemented in the Java code while writing a class. Normally in an Android project, we have a separate package for handling the activities. As it was mentioned earlier, in order to put a complete trace of application, there is an `AndroidManifest.xml` file in which the record of every activity, service, permission, and other things should be included.

 While implementing the code, we need to take care that all of the activities should be declared in the `AndroidManifest.xml` file in order to access them from the code. Otherwise, Android will throw an error of `ActivityNotFoundException`.

The implementation of Android Intents for activities, services, and Broadcast Receivers is the same. While implementing Android Intents in an activity, we need to take care of the following things:

- Importing the `android.content` package before implementing the intent (this is the parent package of Android in which the intent class is present).
- The intent constructor should be under the context of Android Activity. If not, it should have the object of context in order to determine on which activity the intent is called.
- The destination activity class should be imported (if it is under any other package of the source activity).
- You can only call the `startActivity()` method if the intent is in the context of Android Activity or if the context of that source activity is present in the context object.

The XML component

The second and the most important component on which the intent depends is in the `AndroidManifest.xml` file. Giving a recap about what this file includes – `AndroidManifest` is the file which contains all the information regarding the application. It contains all names of activities, services, permissions, version codes, sdk versions, and many other things.

Similarly, there are Intent Filters that are also mentioned inside this file. At this time, we just want to cover the main use of Intent Filters. The brief introduction of an Intent Filter can be found in the following list:

- Intent Filters have some conditions that have to be fulfilled in order to process the Android Intents
- Intent Filters have additional information regarding the data and category of the Android Intent

A shorter definition may be that an Intent Filter describes how the Android System will identify what behavior to adopt on a certain Android Intent.

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

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



You can include many Intent Filters inside a single activity in `AndroidManifest.xml`.

As shown in the previous code, it is clearly mentioned that the tags of `<intent-filter>` contain the information regarding the category and the action. These tags are an essential part of the activity in `AndroidManifest` when the application tries to do a task which is unknown to the system.

To understand why `<category>` appears in the previous code of an Intent Filter, take an example: we make an application in which there are two activities. If we do not mention to the system which is the first activity to start the application with; the system will get confused and show the error `No Launcher Activity Found` and it will not start the application. So, in order to achieve this task, we need to put the category of any of the one activity as `android.intent.category.LAUNCHER`. This will help the system to recognize the base activity from which the application starts and the flow continues.

Implementation of Android Intents for Activity Navigation

In this section, we will take a look at the implementation of the Android Intent. Let's get started.

In order to start this example, you need to build an Android project. You can use Android Studio or Eclipse (as per your convenience) but make sure that if you are using Eclipse, you should have correctly installed JDK, ADT, and the Android SDK, along with their compatibility packages. If you don't know the difference between these IDEs, you can refer to *Chapter 1, Understanding Android*, of this book.

Creating a project in Android Studio was covered in the previous chapter. Repeating those steps will help you to create a complete Android project with some predefined files and folders.

In order to get started with the implementation of Android Intents, you need to perform the following steps:

1. Create a new Android project or choose any existing project in which you want to implement Android Intents.
2. Open the source activity in which you want to implement the intent.



Just a reminder that intents are called when there is an event-call taking place in an activity. For example, on tapping the button, the next activity should appear. So the button-tap is the event.

3. Implement the following code in order to achieve this result:

```
//-----  
-----  
Part One - MainActivity Class
```

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (Button) findViewById(R.id.button1);
        button.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                Intent myIntent = new Intent(MainActivity.this,
                    MySecondActivity.class);
                startActivity(myIntent);
            }
        });
    }
}

//-----
-----
Part Two - MySecondActivity Class

public class MySecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_two_layout);
        Toast.makeText(this, "The Intent has been called...",
            Toast.LENGTH_LONG).show();
    }
}

//-----
-----
Part Three - activity_main.xml File

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

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

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Press to navigate"
/>
```

```
</LinearLayout>
```

```
//-----
-----
```

Part Four - activity_two_layout.xml File

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

```
</LinearLayout>
```

```
//-----
-----
```

Part Five - AndroidManifest.xml File

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

```
<uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="16" />
```

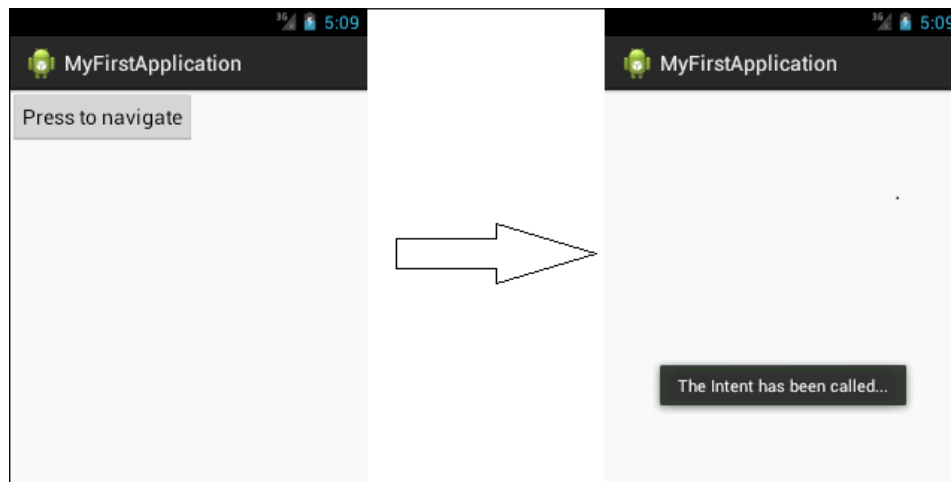
```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
```

```
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
<activity
    android:name=
        "com.app.fragmenttestingapplication.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>
<activity
    android:name=
        "com.app.fragmenttestingapplication.MySecondActivity"
    android:label="@string/app_name" >
</activity>
</application>

</manifest>
```

4. Run the project and a button will appear. Tap the button in order to navigate to the next activity through intents.



Indicates the flow of application from the first activity to the second activity

Understanding the flow

The previous code is described in five parts; we will describe them one by one. Keep in mind that these parts are referring to five different files in the Android project. Knowing the fact that you will use this code inside a predefined project, we will describe it from the scenario of a newly created project in order to make it elaborative and clear.

Part one – MainActivity.java

`MainActivity.java` is the first class that is made while creating a project. Since it is an Android Activity, it comes with an `onCreate` method which is called once the activity has been created (as discussed in *Chapter 1, Understanding Android, Android Activity Life Cycle*). The layout attached with this Activity is named as `activity_main.xml`. Hence in the `onCreate()` method, the line `setContentView(R.layout.activity_main)` refers to that XML and is used to set the view of that activity in accordance with the layout present in `activity_main.xml`.

Now, in the second step, the button present in the `activity_main.xml` layout with the ID of `button1` is fetched in the code by using the `findViewById(int id)` method present in the `Activity` class. It will return the object of the `View` class, so we can easily cast it over the button in order to have a button object.


Once the button object is extracted, we implement the `setOnClickListener()` method on it. The `setOnClickListener()` method belongs to the `View` class which takes an argument of `View.OnClickListener` (an interface). This interface requires us to override the `onClick()` functionality to be implemented. This event is triggered whenever the button is tapped in the UI.

Inside this `onClick()` method, the real implementation of intent will take place. Since we want to call our intent on tapping the button, we will do all of the stuff related to the intent in this method. Declare the intent object with the constructor taking the argument of context and the destination class. The `MySecondActivity.class` file is the destination activity on which we want to navigate while we are accessing the context of the current activity by taking `MainActivity` (or you can use the `getContext()` method which basically returns the same context of the activity). This is because we are currently in the context of `OnClickListener`.

At this moment we have the object of intent. We could do much more with this object but at the moment, our task is to only navigate it to the next activity. This is why we call the method `startActivity()` and pass this intent as an argument. This will navigate the application to the next activity which will appear from the top in the activity stack, while the previous activity will go underneath it.


Part two – MySecondActivity.java

The activity `MySecondActivity.java` is the destination activity for the intent. This is the simple activity which contains an `onCreate()` method which is setting the content view on the screen. Just after the creation of the layout, we are showing a toast message on the screen through the `show()` method in order to recognize that the second activity has been loaded and is displaying the message `The intent has been called`.


 A toast message is a simple message which appears in a box for a few seconds and goes away. By default, this message contains text, but one can easily make a custom toast in order to include pictures and many other things.

Part three – activity_main.xml

This XML file is the layout for the `MainActivity.java` class. As you can see, we tried to bring the reference of the button from the layout. That button is declared inside this `activity_main` file with the ID used to extract it from the XML.

 The references of all the layouts declared in the XML files are put inside the R file, which is used by the Java code/classes to get the object in the code.

Describing the `activity_main.xml` file, there is a Linear Layout with certain parameters regarding its length, width, and orientation. Inside the tags of Linear Layout, you can see that the button which is brought into the code of Java is declared. This tag also comes with certain parameters about height, width, ID, and text that will appear in the layout.

 Take precautionary measures in order to give the ID to any view. All of these IDs are present in the R file which handles everything inside a single class. Try to customize your ID in order to avoid confusing the view of one activity with that of another.

Part four – activity_two_layout.xml

This is a simple layout file containing the parent Linear Layout in order to make a simple blank activity. This layout file is assigned to the second activity on which the intent brings the user after tapping the button and on which the toast message is shown.

Part five – AndroidManifest.xml

No project is complete without the `AndroidManifest.xml` file because it contains all the information regarding the application. In this XML file, there is a parent tag of `manifest` which contains the three most important properties regarding the project:

- **Package Name:** This is the name of the project from which the application will go to Google Play (or any other market). This name should be uniquely defined for the whole application and it should not match any other package name in the market.
- **Version Code:** This is an integer value which represents the version number compared to the previous version of the application.
- **Version Name:** This is a string value which is used to display for the user. It is the release version name of the application.

Inside the `Manifest` tag, there is a tag named `<uses-sdk>` which, in its attributes, defines the minimum and maximum API version of Android on which the application is accessible. After this there is an `application` tag in which the information related to application is stored which contains icon, label, and theme for the application.

In the `main` tag, which describes the activities in an application, there are the `<activity>` tags. It should be equal to the number of activities used in the application. As you can see the XML component of intent is present in the first activity, namely `<intent-filter>`, which is telling the system that `MainActivity.java` is the class that should be used as a Launcher Activity. Unlike the first activity, the second activity does not contain the tag for Intent Filter. You can analyze that there is no need for the second activity to contain those tags because it is in the flow with the first activity.

From the first activity, the second activity should come on the foreground intent. That is why there is no need for it to contain the tags `<category>` and `<action>`. The application will work fine without it as well.

Future considerations

We will see in detail what more we can do with intents in this book. Till now we have only covered the basic intent functionality with its flow in the activity stack. You will encounter a lot of content on the way.

Other constructors of the android.content.

Intent class

The Intent class comes with a variety of constructors which help developers in various scenarios. In the previous section, we used only one type of constructor. Other polymorphic forms of the constructor are also available on Google.

Various kinds of constructors are explained in the following sections.

Intent()

This is the default constructor which returns an empty intent object. However, by empty it does not mean that it is a null object.

Intent(Intent o)

This constructor is used to make a clone of an existing intent. We pass the intent object which we want to clone and put it in the argument of the constructor. In this condition, the intent returned is the copy of the original one. It will also map each and every value (extras) put in the original one and returns a copy of that intent.

Intent(Context c, Class<?> cls)

This is the constructor that we used in our example before. It basically takes two arguments: a source context and the destination class. The source context is the context of the activity you are currently using, while the second parameter is the class to which you want to navigate to.

Intent(String action)

The constructor with the action is used to make an intent object in which the action is written. The proper use and definition of the action, as used in intents, will be covered in the upcoming chapters. Also keep in mind that this constructor is used to broadcast actions.

Intent(String action, URI uri)

This constructor is used to create an intent with the desired action as well as some data URL. For example, we pass the argument `new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.com"))`; . The constructor clearly means that the intent will facilitate the action that is used for viewing and with the other parameter, we are parsing the URI of the URL `http://www.google.com`. This will open the browser in which Google's website will be loaded. Take another example of this constructor: `new Intent(Intent.ACTION_DIAL, Uri.parse("tel: (+1) 123456789"))`; . By writing this statement, we are clearly mentioning that we want to make an intent which is used to activate the phone dialer and pass the value of URI in order to perform the call function through intents.



This form of constructor is mostly used for calling implicit intents. More information about the code can be found at <http://developer.android.com/reference/android/content/Intent.html>.

Getting results from Android Intents

As we have seen previously, Android Intents were used to navigate from one activity to another but in real scenarios there are many things that are required with this navigation. One of the most important features of Android Intents is going to be discussed here. What we will see here is the response that the source activity will get once the destination activity is closed.

To further explain the previous statement, we have a scenario in which:

- There is a source activity (from which the navigation will start)
- A destination activity (to which the navigation will be done)
- On the closure of the destination activity, it will return a result back to the source activity

This is a very handy option in Android Intents for bringing back the result from an activity. We will discuss this feature in a great detail with an example code.

Understanding with an example

In this example, we will look at the scenario in which there are two activities. Activity one will be used as the launcher activity. Activity two will be used as the destination activity which will also return some result back to the first activity. The first activity will catch the result and on the basis of that code, it will decide what kind of task was finished or failed in the second activity. In the end some dialog message will be shown on the first activity in accordance with the result that is given back.

Going deep into the example

Again, the following code is in five parts. It is the modification of the last example in which the normal use of intents was described. In order to implement this example, perform the following steps:

1. Create a new project or open any existing project in which you want to make the changes.
2. Open the source activity in which you want to implement the intent.
3. Implement the following code:

```
//-----  
//Part One - MainActivity Class  
  
public class MainActivity extends Activity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        Button button = (Button) findViewById(R.id.button1);  
        button.setOnClickListener(new OnClickListener() {  
  
            @Override  
            public void onClick(View v) {  
                // TODO Auto-generated method stub  
                Intent myIntent = new Intent  
                    (MainActivity.this, MySecondActivity.class);  
                startActivityForResult(myIntent, 1);  
            }  
        }  
    });  
}
```

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {

    if(requestCode == 1){
        if(resultCode == RESULT_OK){
            Toast.makeText(this,
                "The Processing is succesfully done..."
                , Toast.LENGTH_LONG).show();
        }
        if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "The Processing failed,
                try again later..", Toast.LENGTH_LONG).show();
        }
    }
}
}
```

```
//-----
//Part Two - MySecondActivity Class
```

```
public class MySecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_two_layout);
        Toast.makeText(this, "The Intent has been called..."
            , Toast.LENGTH_LONG).show();

        Intent returnIntent = new Intent();

        CheckBox yesCheckBox = (CheckBox)
            findViewById(R.id.checkBox1);
        CheckBox noCheckBox = (CheckBox)
            findViewById(R.id.checkBox2);
        Button button = (Button) findViewById(R.id.button2);
        button.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                if(yesCheckBox != null && yesCheckBox.isChecked()){
                    setResult(RESULT_OK, returnIntent);
                }
            }
        });
    }
}
```

```

        finish();
    }else if(noCheckBox != null &&
noCheckBox.isChecked()){
        setResult(RESULT_CANCELED, returnIntent);
        finish();
    }
    }
    });
}
}
}

```

```

//-----
-----

```

Part Three - activity_main.xml File

```

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Press to navigate"
    />

</LinearLayout>

```

```

//-----
-----

```

Part Four - activity_two_layout.xml File

```

<?xml version="1.0" encoding="utf-8"?>
<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" >
    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
    />

```



```
    android:layout_below="@+id/checkbox1"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="24dp"
    android:text="@string/return_string" />
```

```
<CheckBox
    android:id="@+id/checkbox2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/checkbox1"
    android:layout_alignBottom="@+id/checkbox1"
    android:layout_toRightOf="@+id/button1"
    android:text="@string/no_string" />
```

```
<CheckBox
    android:id="@+id/checkbox1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_marginTop="50dp"
    android:layout_toLeftOf="@+id/button1"
    android:text="@string/yes_string" />
```

```
</RelativeLayout>
```

```
//-----
```

Part Five - AndroidManifest.xml File

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

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
```

```

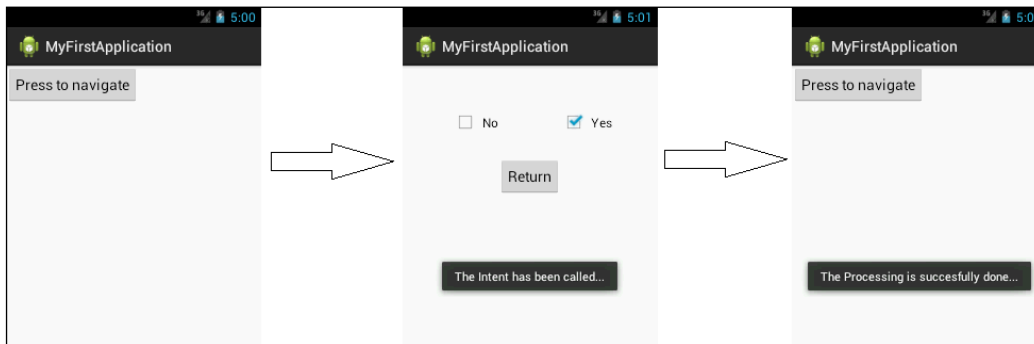
        android:name=
            "com.app.fragmenttestingapplication.MainActivity"
        android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name=
                "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity
        android:name=
            "com.app.fragmenttestingapplication.MySecondActivity"
        android:label="@string/app_name" >
        </activity>
    </application>

</manifest>

```

4. Run the project and a button will appear. Tap the button in order to navigate to the next activity through intents.
5. After successful execution of the intent, the new activity will appear. Click on the back key of the emulator or Android phone on which you are testing and the system will bring you back to the first activity.
6. After coming back to first activity it will show the toast in order to communicate the result from the destination activity.



Indicates the flow of application from the first activity to the second activity and getting the result back to the first activity

Explaining the code

In this example, as mentioned before, we will understand how to get back the result in order to verify (on the first activity) whether the task performed in the second activity is successfully done or not.

Before going further, keep in mind that this is an extension of the previously defined example; so the things that we went through previously will not be explained in this example. Kindly read the previous example if you find anything difficult in here.

Let's get started, step by step.

Part one – MainActivity.java

Similar to before, this is the first activity from which the navigation will start on tapping the button. Most part of this activity is same as that of the previous example, but if you take a deeper look, you will notice that the `startActivityForResult()` method is used in order to start the activity. That is because the activity that we are starting will return a result after the second activity is closed.

The `startActivityForResult()` method has two arguments. First is the intent which is similar to that of the `startActivity()` method described before. Let's check out what the second argument is. Take a scenario where the first activity invokes more than one activity which will return some result. Now at the time of catching that result, we need to be sure which activity it is coming from. So this is why we assign a request code which will return the result when the intent gets back the user to the first activity. We will then put a check to know which activity the result is coming from.

Moving on, if you look carefully in the code, there is an overridden method named `onActivityResult()`. This method will be called when the closure of the second activity occurs. It has three arguments as you can see in the code. The first is `requestCode` which will be in accordance to the activity which is sending the result back. Apart from that we have `resultCode` which will tell whether the task in the second activity was performed successfully or not. Thirdly, we have an intent which is basically the object which is causing the activity to move back to the first one. We can also send some data back to the first activity through this intent.

Part two – MySecondActivity.java

This is same as the first example; rather, it carries two objects of check boxes in the layout file and these are used to make a result in the second activity. We find the view using the ID and on the basis of the selected check box, we see which one is enabled. When the button is pressed, it will see which check box is enabled and on the basis of the enabled priority, it calls the `setResult()` function. This function sets a result to the intent which will help the activity to move back to the first activity.

After setting the result, we will finish the second activity and it will move back to the first activity with the result. Just after the activity has finished, the `onActivityResult()` method from the first activity will be executed in order to see the result which is sent by the second activity.

Part three – activity_main.xml

This part of the code is similar to the first example; kindly refer to the first example for an explanation.

Part four – activity_two_layout.xml


This layout file refers to the second activity. We have put two checkboxes, on the basis of which it will be decided which result is to be sent back to the first activity. There are some attributes that are the same as the ones in the previous example and hence, that example can be referred to.

Part five – AndroidManifest.xml

Again, this file is untouched and is same as the first example. Kindly refer to the first simple example of intent.


Future considerations

There are two scenarios which we can focus on as a future consideration. The first is how two or more activities can be handled while all of them are sending the result back. This is of course with the help of the `requestCode` argument. The second most important thing is, in the scenario where we are sending back only the response code, there might be a need to also send back some strings, int values, or some custom objects as a result. In that case, we will need some other methods to send those objects back to the first one via intents.

 In Chapter 5, *Data Transfer Using Intents*, we will have a complete look into how we can transfer different kinds of data inside an intent.

Structure of an intent

In this section, we will study the structure of an intent object as used in Android. An intent is the bundle of information in which there are multiple things to facilitate. It has information about the actions that should be taken while the intent is executed. Similarly, it also has information about the category which is going to handle the intent. Data plays a vital role in intents. So an intent has information of the data in the form of a URI that has to be executed.

 Due to limited space here, we cannot explain every constant in every component. In order to get a complete list of constants that Android uses, you can take a look here:
<http://developer.android.com/reference/android/content/Intent.html>

We will now go through each one of the components in order to see what they really mean.

Component

This will explain which component will get affected or handle the execution of a particular intent. For example, there is an intent that is responsible for making an action which is related to the activity in which it is called. Similarly, there is an intent which is handled by Broadcast Receiver and reports when a certain system-related task is performed. If the component name is not set, it will self-recognize the component with other information. The following table shows the different kinds of components that are used in Android Intents:

Constants	Description
CATEGORY_BROWSABLE	Describes that the intent can be safely executed by the browser to display the data.
CATEGORY_LAUNCHER	Tells that the activity should be executed as the launcher while the application starts.
CATEGORY_GADGET	The activity can be put inside another activity that starts from any gadget.

Actions

Actions basically state what action this intent will cause. For example, if we initiate an intent object with the action named `ACTION_CALL`, this intent will start the call functionality with the data string passed with the `ACTION_CALL` action in the form of URI. Taking another example of `ACTION_BATTERY_LOW`, which is related to the Broadcast Receiver component. By placing this action in the Intent Filter, it will fire the event (or in short a low battery pop-up) if the battery goes lower than that of the threshold value.

There are various kinds of actions present in Android. The following table shows some of the intent actions and their description:

Constants	Component	Description
<code>ACTION_CALL</code>	Activity	Start the phone call
<code>ACTION_EDIT</code>	Activity	Display the data of the user to edit
<code>ACTION_MAIN</code>	Activity	Start as the initial activity with no data
<code>ACTION_SYNC</code>	Activity	Sync the data present on the server with the mobile device
<code>ACTION_BATTERY_LOW</code>	Broadcast Receiver	Shows the battery low warning
<code>ACTION_HEADSET_PLUG</code>	Broadcast Receiver	Shows an alert when the headset is plugged in or unplugged
<code>ACTION_SCREEN_ON</code>	Broadcast Receiver	Triggered when the screen is turned on
<code>ACTION_TIMEZONE_CHANGED</code>	Broadcast Receiver	When the setting of the time zone is changed

Data

This should not be considered as the separate component; rather, it is used to facilitate the action component. As described previously, there are certain components which require some data to be passed. For example, the `ACTION_CALL` function requires a data value by which it recognizes on which telephone number calling should be performed. In this particular scenario, we need the `tel : xxxxxxxxxxxx` URI to be put in the data and forwarded to the action. Similarly, when the `ACTION_EDIT` or `ACTION_VIEW` actions are performed, they need to be provided with a document or a HTTP URL in order to complete the action. The data is given to the intent in the form of **URI (Universal Resource Identifier)**.

Extras

These are basically the key-value pairs of the additional data that is required by the Android Intent. We can take these values from the code (when we make an object of the intent) and transfer that data to the next activity. Talking with respect to the actions, there are some actions which require additional data to fulfill the task. For example, the `ACTION_TIMEZONE_CHANGED` action needs an extra time zone which describes the new time zone on the basis of which further tasks can be performed.

Summary

In this chapter, we discussed the introduction of intents with its role, technical overview, the basic implementation in an Android Application, and the structure of intent based on which, we will further explore different kinds of tasks that can be performed. The chapter also provides two very important implementations of Android Intents in which the navigation from one activity to another took place, while in the second one, the result corresponding to any particular activity is sent back to the first activity. The concept discussed in this chapter is the key tool to understand the advanced concepts of Android Intent which will be discussed later in this book. In the next chapter, we will learn about the categorization of Android Intents and its theory and implementation in the light of various handy examples which can be implemented easily in your Eclipse environment.

3

Intent and Its Categorization

Intents are asynchronous messages used to activate one Android component using another. These intents are used to trigger the Android OS when some event has occurred, and some action should be taken. The Android OS, on the basis of the data received, determines the receiver for the intent and triggers it.

Generally, there are two types of intents: **explicit** and **implicit**. As their names suggest, explicit intents trigger specific components of the Android OS specified explicitly by the developers. However, implicit intents trigger the general components of any category of the Android OS. It's left to the Android OS to decide which component needs to be activated. If there is more than one general component, the user is asked to select a component from a list of all the components. This feature of the intents in the Android OS makes an application more interactive because the other applications and developers can also access it. For example, you are developing a picture-editing Android application to edit any image, apply filters to it, and so on. So, if the application receives images from any source of the Android system, such as an e-mail attachment, gallery images, any other image tools, and so on, the application will become more interactive and responsive as compared to the application that loads images from the app itself. This interaction of the application, whether it is sending an image via e-mail or receiving an image in an editing application, is implemented through implicit intents.

In this chapter, you will learn about the following topics:

- Types of intents
- Explicit intents
- Using explicit intents in an Android application
- Implicit intents
- Using implicit intents in an Android application
- Intents and Android late binding



The concept of intents and their structure, as discussed in the previous chapter, are the prerequisites for understanding this chapter and the chapters that follow. If you don't have a basic understanding of these things, read *Chapter 2, Introduction to Android Intents*, in order to move forward.

These two types of intents, namely, explicit and implicit intents, are very different in terms of functionality. Let's start with the simplest type of intent first.

Explicit intents

The simplest type of intent is the explicit intent. When a developer knows which component to use and doesn't want to provide free access to the user, explicit intent is the best choice. Here, the developer explicitly specifies the component to be triggered in the declaration of the intent. This component can be any activity, service, or broadcast receiver. For example, an Android application usually consists of more than one activity for a corresponding functionality. In order to navigate from one activity to another, an explicit intent is used. The following code snippet shows a simple declaration of such an explicit intent that targets activity B from activity A:

```
// Set the target of the intent from ActivityA to ActivityB
Intent intent = new Intent(ActivityA.this, ActivityB.class);
```

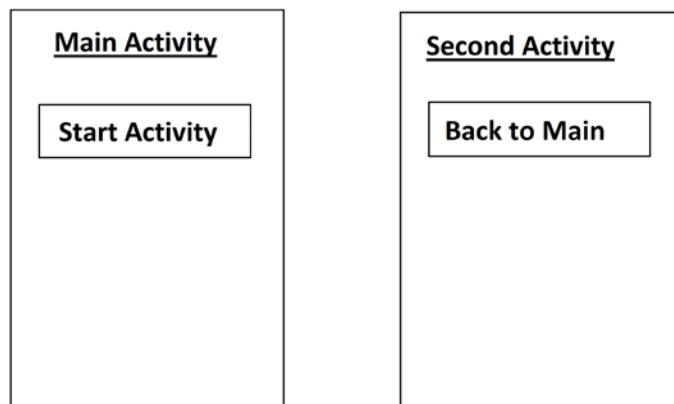
Intents are instances of the `android.content.Intent` class. The explicit components are specified as Java class identifiers, and they are used by the Android OS to activate them whenever a developer sends the intent. As explained in the earlier chapters, an object of the intent takes two parameters: **context** and **class**. The context parameter receives the source context from which the intent is triggered to activate other components. The class parameter takes the class object of a specific class that is used to specify the target component. In the preceding code snippet, an intent object takes the `ActivityA` class as the source component and `ActivityB` as the target component to be activated. Now, this intent object declared in the code snippet with the specific component as the target can be used anywhere. For example, it can be used for starting `ActivityB` with `ActivityA` as the parent activity. The following section describes the uses of explicit intents to trigger various components such as activities, services, and so on.

Using explicit intents in an Android application

In this section, we will discuss the various uses of explicit intents in an Android application. As discussed earlier, explicit intents can be used in activating other components such as activities, services, and broadcast receivers. We will talk about two examples of explicit intents; the first one to start one activity from another activity, and the other to start a service in the background.

Starting an activity through an explicit intent

Any Android application contains at least one or multiple activities. So, in the case of multiple activities, navigating between them becomes important for a developer. In this section, we will develop an example of two activities in which we will start one activity from the other and go back to it after stopping/finishing the currently opened activity. The following figure shows a simple prototype of the example we are about to develop:



As it is clear from the preceding figure, we have two activities: **Main Activity** and **Second Activity**. Both the activities have a single button to navigate and a title showing the name of the activity. Now, let's get started with the development of our first example. But, in order to start this example, you need to build an Android project. You can use Android Studio or Eclipse (as per your convenience), but in the case of Eclipse, make sure that you have correctly installed the JDK, ADT, and Android SDK along with their compatibility. If you don't know the difference between these IDEs, refer to the first chapter of this book. Creating a project in Android Studio has been explained in the previous chapter. Repeating those steps will give you a complete Android project with some predefined files and folders.

After creating an empty Android project, we are going to implement the use of explicit intents. In the example, we will write or modify many different files of various types, such as Java, XML, and so on. Each file has its own purpose and performs its own functionality in the example. Let's explore these files one by one now.

The MainActivity.java class

The main file of the project is `MainActivity.java`. The following is the code snippet to be implemented in this file:

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (Button) findViewById(R.id.button1);
        button.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                Intent myIntent = new Intent(MainActivity.this, SecondActivity.class);
                startActivity(myIntent);
            }
        });
    }
}
```

The applications having more than one activity must have a main activity. This main activity defines the starting point of the application. In our example, the `MainActivity.java` class is the main activity of the project. To provide a layout file to the activity for visual representation, we will call `setContentView(R.layout.activity_main)` within the `onCreate()` method of the activity. The `activity_main.xml` file is the layout file contained in the `layout` folder of the project directory and represents the main screen of the app. After setting the View of the activity, we get all the components that will be used in the activity by getting their Views from the layout file. To get the View of any component, we will use the `findViewById()` method that takes an ID of the View and returns a View object that is the type casted as per our requirements. In this file, we've got the View object of the button having the ID, `button1`, in the layout file, and we have the `Button` type casted it to and reference it to our button. Any button should have listeners for user interaction with the View and customizing the behavior of the View. In our file, we only set `View.OnClickListener` for the button to get the clicks/taps of the button.



There are two `OnClickListener` classes in the Android SDK. One is in the `View` class. It is used for Views such as buttons, text fields, and so on. The other is in the `DialogInterface` class and is used for the detection of clicks and taps in dialogs. Developers should be careful when importing classes and their packages.

We have set the `OnClickListener` object of the button using the `setOnClickListener()` method of the `Button` object. We referenced an anonymous listener in the parameter of the method, and have overridden the `onClick()` method. In the `onClick()` method, we have to provide the behavior that we want to show when a button is tapped on.



Anonymous objects are the objects that have no object name specified by the developer. And because of this reason, developers can't access the object directly in the code. You can also set the `OnClickListener` object of the `View` by creating the object of the interface and passing it in the `setOnClickListener()` method.

We have created an intent object with `MainActivity` as the source context, and the `SecondActivity` class is the target component to be activated. One thing to be noted here is that instead of passing the object of `SecondActivity`, we have explicitly passed the Java class representation of `SecondActivity`. This is how we have declared an explicit intent object. Now, through this intent object, we can do many different tasks depending on our requirements, but in our case, the options are limited. We have created an explicit intent that contains very specific information; so, for that reason, this intent can be used for limited purposes only. In this example, the intent is used to start another activity on top of a back stack by calling the `startActivity()` method. This method takes the parameter of an explicit intent object having the information of the source context and target activity.



To declare the intent in a class other than running activities, we can use the application context to pass in the context parameter of the intent constructor. This can be accessed by the `getApplicationContext()` method.

So, summarizing the functionality of the file here, the file represents the starting point of the application that shows a button. On tapping the button, the application navigates to another activity. This navigation between activities is implemented through an explicit intent object. After implementing the `MainActivity` file, let's implement our second activity in the `SecondActivity.java` file.

The SecondActivity.java class


The `SecondActivity.java` class is the destination activity for our explicit intent. The following is the code snippet implemented in this file:

```
public class SecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_two_layout);
        Button button = (Button) findViewById(R.id.button1);
        button.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                finish();
            }
        });
    }
}
```

Again, this class is extended from the `Activity` class, and follows the activity lifecycle. It should also override the lifecycle-callback methods as well. In the `onCreate()` method, we set the `View` of the activity by calling the `setContentView()` method, and this time, we have passed the `activity_main2.xml` file's reference using `R.layout.activity_main2`. This XML file is placed in the `layout` folder under `res`. We again get the `View` component of the button from the layout file by calling the `findViewById()` method, and we typecast this to `Button`. We then set `OnClickListener()` of the button to an anonymous listener, and override the `onClick()` method to define the behavior on a button tap. This time, we call the `Activity.finish()` method in the `onClick()` method. This method simply pulls out the activity on top from the back stack.

As we started `SecondActivity` from `MainActivity`, when we finish `SecondActivity`, we will see `MainActivity` again.

 The back button in Android devices simply calls the `finish()` method for activities or the `dismiss()` method for dialogs.

We can also create an intent object with the context of `SecondActivity` and the target class of `MainActivity.java`. But, this will create a new instance of `MainActivity` in the back stack, and will push this on top of `SecondActivity`. In that case, we will have two instances of `MainActivity` and one instance of `SecondActivity` placed in the middle of these activities in the back stack.



If we set the `android:noHistory` attribute to `true` in the `<activity>` tag of `MainActivity` in the `AndroidManifest.xml` file, starting a new instance of `MainActivity` will result in an already-created instance to be placed on top of the back stack, thus avoiding the creation of a new object. Developers should be more careful at the time of creating the app flow and navigation control because this type of flow can make loops in the app. This can cause a never-ending app problem.

We should note that both activity files, `MainActivity.java` and `SecondActivity.java`, contain almost the same code except the button listener of `MainActivity` that starts a new activity using explicit intent, and the button listener of `SecondActivity` that just pulls back the application by auto-pressing the back button of an Android phone.

Until now, we have learned how an explicit intent can be used to navigate to another activity. But, it should be remembered here that both activities were using different layout files for visual representation containing the action buttons that performed both tasks. It should be noted that the layout files don't play any role in the navigation of explicit intents. These files just show the visual content of activities to make the user interaction easy. Now, let's focus on those layout files.

The activity_main.xml file

The `activity_main.xml` file is the layout file of `MainActivity.java` and is written in XML. The following code shows the implementation of the layout file:

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Activity"
    />

</LinearLayout>
```

From the layout file, we have `LinearLayout` and `<Button>` Views within the layout. Remember that this button was extracted by the `MainActivity` file after setting the View of the activity to `OnClickListener`.



The references of all the layouts and Views declared in the XML files are autogenerated inside the `R.java` file. In Java, we can use the `R` class and access components in a static manner. For example, we could use `R.layout.layout_name` for layout. However, in XML, `R` can be accessed by placing `@`. For example, for accessing a color, we can use `android:name="@color/my_custom_color"`.

Describing the `activity_main.xml` file, there is `<LinearLayout>` with certain parameters regarding height, width, and orientation. As you can see, inside the tags of `<LinearLayout>`, the `<Button>` tag that is brought into the code of Java through the `findViewById()` method in the activity is declared. This tag also comes with certain parameters such as `id`, `width`, `height`, and `text` that will appear in the layout.

The activity_main2.xml file

The `activity_main2.xml` file layout is a visual representation of the `SecondActivity` class. The following code shows this file:

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Back to Main"
    />

</LinearLayout>
```

This layout is the same as the `activity_main` layout except that the text values of buttons are different. This is used in the `SecondActivity.java` class. Again, a button is referenced from XML to Java through the `findViewById()` method in the activity, and `OnClickListener` is set to define the custom action to be performed when a button is tapped on.

No Android application is complete without the `AndroidManifest` file. We have already implemented the visual layouts of our example app and also their functionality of navigating from one activity to the other, using explicit intents. However, when any application has multiple activities, the Android OS must be informed about all those activities and their attributes. In order to inform the Android OS about how many activity classes are used, the `AndroidManifest` file is used. Let's see how this file informs the Android OS about the activities.

The AndroidManifest.xml file

The `AndroidManifest.xml` file contains all the settings and preferences of the application. When working on multiple activities, developers should be careful while declaring activities. Only one activity that defines the starting point of the application should have an intent filter of the launcher. All activities should be saved in here. The information about activities can include the label of the activity, theme of the activity, orientation, and so on. For activities, the `AndroidManifest.xml` file is like a registration center where all the components in the app, such as `Activity` classes, `Service` classes, and so on, should be registered. If any activity is not registered in the `AndroidManifest.xml` file, Android OS will throw the `ActivityNotFoundException` exception on calling that activity.

About our explicit intent app, both the activities, MainActivity and SecondActivity, are registered in the AndroidManifest.xml file. It should be noted here that MainActivity has a subtag of <intent-filter> that declares MainActivity as the starting or launcher activity of the whole application. The following is the code implemented in the AndroidManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter3.explicitintents1"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.chapter3.explicitintents1.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>
        <activity
            android:name="com.chapter3.explicitintents1.SecondActivity"
            android:label="@string/app_name" >
        </activity>
    </application>

</manifest>
```

With the `AndroidManifest.xml` file, our explicit intent example is completed. In this example, we defined two activities; one of them was the main activity. The main activity used an explicit intent to start the other activity by declaring its name explicitly. The example contained two layout files for a visual representation of both the activities and the manifest file to register all the activities and basic settings for the application. When you run the project, you should see the screen transition as shown in following figure:



In the next section, we will have a look at another use of explicit intents in services. We will learn how a service is started explicitly using intents from an activity.

Starting a service through an explicit intent

Unlike activities, services perform specific tasks and actions in the background. Services don't have any visual layout or UI. It must be noted that services run on the main thread of the app; so, when Android has a need of memory, it will stop the services that are not running or are in a paused state. In the next example, we will start a service using an explicit intent, and we will also stop it using the same explicit intent. So, in order to start this example, create an empty project using any Android IDE such as Eclipse with the ADT plugin or Android Studio. In this project, we will implement the use of explicit intents for starting/stopping services. As our main focus is on explicit intents, we will create a very simple service that will display toasts on being started or stopped by the activity. We have modified/implemented the four section in the example application. So, let's see what these files do one by one.

The ServiceExample.java class

Since our example app includes a service, this file is a representation of a service class used in our app. The following code shows the implementation of the service:

```
public class ServiceExample extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        Toast.makeText(this, "Service Created", 300);
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
        Toast.makeText(this, "Service start", 300);
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {

        Toast.makeText(this, "task perform in service", 300);
        return super.onStartCommand(intent, flags, startId);
    }

}
```

In Android development, we have to extend our class from the `Service` class and override and implement the required methods according to our customizations to create any service. At first, the `onBind()` method is to be implemented by the developer, and this is a mandatory one. This method is an abstract method of `Service`. This method is used to bind the running services at runtime. Then, `onCreate()` and `onStart()` methods are the same as in the activity class. Developers do the necessary initial setup in these methods. In our example, we are just going to display toasts to notify the user about the method calls. The `onStartCommand()` method is a very important method, and this is where we do all the background work. It should be noted that services run in the main thread. So, if you want to do heavy processing tasks, you should create a separate thread in this method and start it. This is how background processing is done in the standard way.



We can also create threads in the main thread, and perform our heavy processing in the background; then why do we need services to create a thread? In Android OS, services are the standard method for doing background processing. When the Android OS becomes short on memory, it can stop the idle services to get their memory. But it can't stop the threads; so, using services is a better way of continuously performing tasks in the background.

We are not doing anything in the `onStartCommand()` method except displaying a toast like other methods of the service class. There isn't any special point to mention about the service class. Like the previous example, the most important part in this example app is the main activity. Let's see the main activity class in detail now.

The ServiceDemoActivity.java class

The ServiceDemoActivity.java class is the main activity of our app and defines the starting point of the application. The following code shows the implementation of the class:

```
public class ServiceDemoActivity extends Activity implements OnClickListener {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        findViewById(R.id.start).setOnClickListener(this);
        findViewById(R.id.stop).setOnClickListener(this);
    }

    private Intent inetnt;
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.start:
                inetnt=new Intent(this,ServiceExample.class);
                startService(inetnt);
                break;
            case R.id.stop:

                inetnt=new Intent(this,ServiceExample.class);
                stopService(inetnt);
                break;
        }
    }
}
```

This class is extended from the Activity class, and we have overridden some of its methods. In the onCreate() method, we set the Content View layout of the activity by referencing the layout stored in the res/layout folder of the app. Then, we extracted buttons in the layout and set OnClickListener for those buttons. In an earlier example, we used anonymous objects of the OnClickListener interface for buttons. In this example, we are providing the Activity class as our OnClickListener by passing this in the parameter of the setOnClickListener() method. The object passed in the parameter must implement the OnClickListener interface and override the onClick() method. So, our activity class implements this interface along with the extending activity. We have also overridden the onClick() method in this class. As we have two buttons this time and only one OnClickListener interface for both, we have to first find which button has been pressed and then take action accordingly.

One way to do this is to get the ID of the pressed View and compare it with the IDs in the resources. So, the `getId()` method returns the ID of the View to us; we pass the ID in the switch block and compare it with our IDs of the buttons. In both cases, we are creating an explicit intent that passes the context of the activity and our service class name as the target component to be activated, as we did in our activity example. How the service is being started by the `startService()` method and being stopped by the `stopService()` method should be noted. These methods take explicit intents that include information about which service has to be started or stopped. This class showed us how easy is it to use explicit intents to start or stop any service from any activity. As usual, this main activity is using two buttons, `Start` and `Stop`, which are extracted from a layout placed in the resources folder of the Android project directory. Let's see what this layout file contains.

The activity_main.xml file

The `activity_main.xml` file is the layout file of `ServiceDemoActivity` and is written in XML. The following code shows the implementation of the file:

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

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="StartService"
        android:id="@+id/start"/>

    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="StopService"
        android:id="@+id/stop" />

</LinearLayout>
```

We have a `<LinearLayout>` element and two button Views in the layout. Remember that these buttons were extracted by the `ServiceDemoActivity` file to set `OnClickListener` for both buttons after setting the View of the activity.



We can create layouts in XML as well as Java. Android recommends creating all the layouts in XML because Java is used for processing in Android. If we create layouts in Java, the layout creation will also be processed; this can make the app more battery consumptive. Use Java only for dynamic layouts such as the ones used in games.

When describing the `activity_main.xml` file, there is a `<LinearLayout>` element with certain parameters regarding height, width, and orientation. As you can see, inside the `<LinearLayout>` tag, there are buttons declared that are brought into the code of Java. This tag also comes with certain parameters about height, width, id, and text that will appear in the layout. Last but not least, the Android manifest file is for the application settings. Like activities, a developer has to register all the services implemented in the app in the manifest file. Let's have a look at the file and see how we have registered our service in the manifest file of the example app.

The AndroidManifest.xml file

To register a service, we have to provide the code inside the `<application>` tag. The following code shows the full implementation of the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.chapter3.explicitintents2"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.chapter3.explicitintents2.ServiceDemoActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

        <service android:name=".ServiceExample"/>
    </application>
</manifest>
```

It can be noted that after `<activity>` tags, we have placed the `<service>` tag with the attribute name to define which service we are registering. If we don't register our service in the `AndroidManifest.xml` file, we will get a `ServiceNotFoundException` exception thrown, and we will get error log such as "Unable to start service (service package with name): not found".



LogCat is found in the DDMS View in Android Studio. LogCat logs all the activity being done in the connected device or emulator. Any Force Close crash exception that is thrown is logged in LogCat, and a developer can find the cause of the crash from it and solve it.

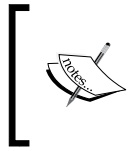
Until now, we have focused on explicit intents, and we created two simple apps that use explicit intents. Now, let's move to another type of intent called implicit intent.

Implicit intents

Unlike explicit intents, when a developer doesn't know which component to use, implicit intent is a good choice. Implicit intents do not directly specify the Android components to be activated as in explicit intents, but they only specify which actions need to be performed. The Android OS will choose the components to be triggered. If there are multiple components which can be triggered, the Android OS provides options to the user to select one component. For example, we want to open a web link in a browser. If we use explicit intents, one option that we have is to develop our own custom browser and trigger it explicitly from our app to view the web link. The other way, which is more preferable, is to use implicit intents to open any already-installed browser in the phone. If there is more than one browser installed in the phone, the user will be given the choice to select one for performing the action, that is, view the web link. This feature of implicit intents works as a general form to perform any action in the Android OS. We specify the data and action in the intent, and Android chooses a suitable component according to that data. The following code snippet shows a simple declaration of an implicit intent that provides a link to be browsed by Android in the best suitable component or browser in this case:

```
Intent intent = new Intent(ACTION_VIEW, Uri.parse("http://www.packtpub.com/"));
```

Like explicit intents, two parameters are passed in the constructor of the implicit intent. You can read more about constructors of intents in *Chapter 2, Introduction to Android Intents*. The first parameter is the action to be performed by the Android OS; we specified it here as `ACTION_VIEW`. This will tell the Android system that something is about to be viewed. The other parameter is the data, mostly defined in the URI format. We have used the PacktPub website as an example. The Android OS will open this web address in the default browser of the phone. If more than one browser is found, the user will be given a list of all the available browsers to choose one to view the address in.



Any URI can be passed in here, not only a web address. For example, the URI of any contact in the phone or any image in the gallery can also be passed, and the Android OS will take the most suitable action for the data passed in the URI.

The general behavior of this intent proves to be a very important feature in Android development. Developers save a lot of time by making generalized apps. This becomes beneficial for the developers as they just have to send information. The rest is defined by the Android OS, and the users get a choice to perform the action as they wish. Developers can not only provide the user with the feature to choose other apps for performing actions in implicit intents, but they can also develop their own custom apps to be added in the choose list. For example, we develop an image-editing app. We want the app to function in such a way that when a user selects an image from any other app, our app appears in the options list so that the user can easily navigate to our app for editing images from anywhere in his or her phone. This can be done through implicit intents. But the difference this time is that we will not send implicit intents; instead, we will receive implicit intents from other apps. We can do this by registering the intent filter in our `AndroidManifest.xml` file, where we will have to define the action for which our app will perform. This feature makes an app more interactive with other apps, and the integration between different apps and the Android OS becomes very easy for developers. In the following sections, we will develop two examples of implicit intents, and see what we can do with these examples.

Using implicit intents in an Android application

This section will discuss the various uses of implicit intents in an Android application. As discussed earlier, implicit intents can be used in communicating with other Android components in a general form unlike explicit intents. Let's look at the implicit intents in action with the following two examples: one uses implicit intents for sharing content, and the other uses implicit intents to get content from other Android apps.

Sharing content using implicit intents

Today, social networking is what makes any application viral and promotes it to other users. Due to a large variety of social networks, it becomes difficult to put all of the sharing features in the apps. Mostly, developers add Facebook, Twitter, and Instagram in their apps, but adding those SDKs in the app sometimes becomes troublesome for the developers as well as the users. For example, multiple SDKs add some size in the build file; an app becomes complex due to a lot of features.

Fortunately, we can solve this problem using a few lines of code through implicit intents. Let's see how this can become possible by creating a simple content sharing example. First create an empty project using any Android IDE such as Eclipse with the ADT plugin or Android Studio, or open any existing project in which you want to add the share feature. We will now implement the use of implicit intents in sharing any data on social networks.

We have implemented a simple, one line-sharing application that asks the user to choose the sharing method and shares the line on that network. There are two main files that are modified in any empty project. Let's explore both the files one by one.

The activity_main.xml file

The activity_main.xml file is the visual layout of our simple, line-sharing app. The following code snippet shows the implementation of this file:

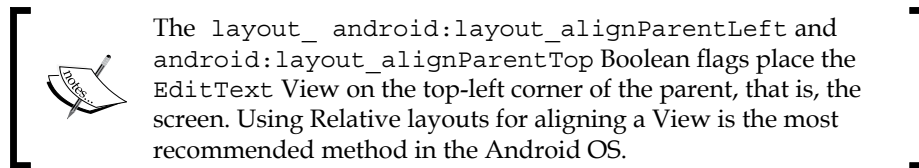
```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:ems="10" >
        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/editText1"
        android:layout_below="@+id/editText1"
        android:text="Share" />

</RelativeLayout>
```

We have a relative layout in which we have placed two Views: a button with text as Share and an <EditText> tag to get an input line from the user. In the previous examples, we used <LinearLayout> to align the Views on the screen. This time, we used <RelativeLayout> to add the Views. In relative layouts, we place our Views relative to other Views. For example, `android:layout_alignLeft` takes the ID of any View and puts this View on the left-hand side of the main View. Similarly, we also used the `android:layout_below` attribute in the Share button to place it below the text field. About the text field, this is the first View on the screen; so, this can be placed relative to the parent View.



This was the visual representation of our one line-sharing app. Now, let's see how implicit intents are used in the main activity file.

The MainActivity.java class

The MainActivity.java class file is the main activity of the line-sharing app. The following code shows the implementation of the file:

```
public class MainActivity extends Activity {

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

        Button shareBtn = (Button) findViewById(R.id.button1);
        shareBtn.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                // TODO Auto-generated method stub
                String msg = ((EditText) findViewById(
                    R.id.editText1)).getText().toString();

                Intent in = new Intent(Intent.ACTION_SEND);
                in.putExtra(Intent.EXTRA_TEXT, msg);
                in.setType("text/html");
                startActivity(Intent.createChooser(in, "Share via..."));
            }
        });
    }
}
```

The class is extended from the `Activity` class in order to make it an activity. As usual, we override the `onCreate()` method, set the Content View of the activity by the `setContentView()` method, and reference a layout that is our `activity_main.xml` file placed in the `res/layout` directory. After setting the layout, we get the button reference from the layout by the `findViewById()` method and set its `View.OnClickListener` interface to an anonymous object.

After setting the Views, let's set the listeners and define the functionalities on the touchable Views. For this purpose, we have overridden the `onClick()` method of the listener and placed our main functionality code in this method as we want to share the line on a button tap. We have first got the message text from the text field by getting its reference and then the text in the field. We created an `Intent` object and passed `ACTION_SEND` as its category into the constructor. The `ACTION_SEND` intent delivers data to the Android OS.

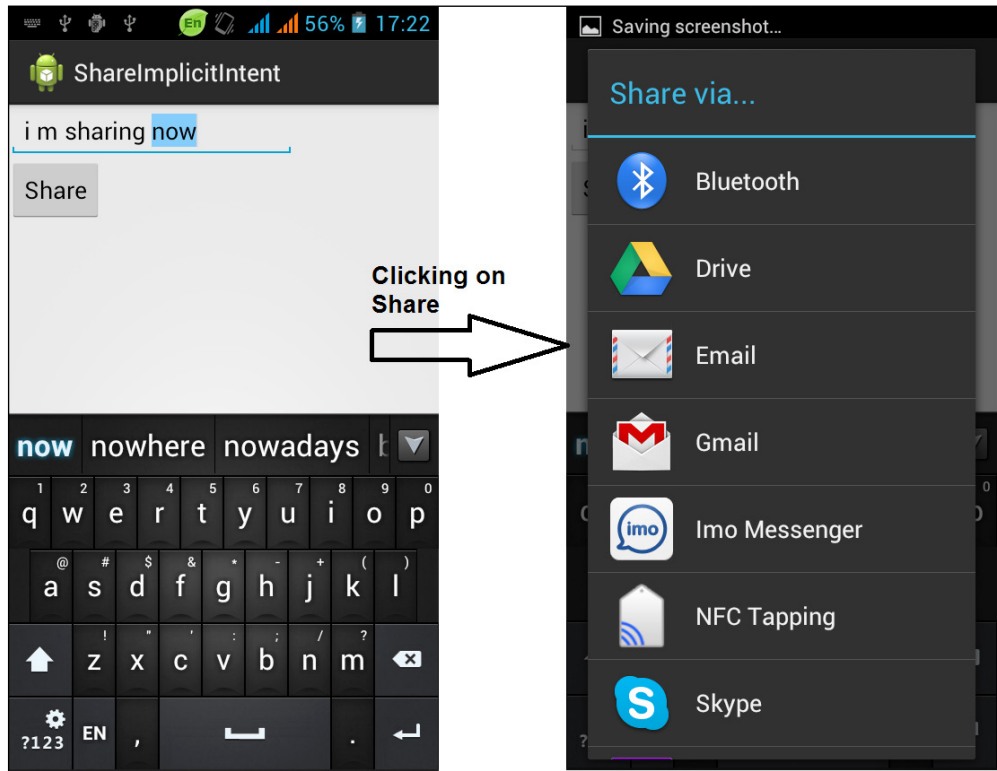
Remember that we have not specified the receiver of the data unlike in explicit intents, where we used to specify the target explicitly. It is up to the receiver of this action to ask the user where the data should be sent, and this happens through the chooser dialog.

After creating an instance of the intent, we add the extra data of the message line in the intent. We have chosen the `EXTRA_TEXT` key to add our data in. For example, if we want to send an SMS, we have to insert the data into the SMS body, and if we want to send an e-mail, we have to put the data into the e-mail body. So, we have chosen a general type of the text, and the Android OS will detect a suitable place to put the data.

Until now, we have set the category and data, but we have to set the type of the data as well. The Android OS will put the apps in our chooser dialog on the basis of the type of the data. For example, we have set the type to `text/html`; this will put all the apps that support the text or HTML format of the data, such as e-mail, Facebook, Twitter, and so on, in the chooser. If we set this to `image/png`, all the apps, such as image editors, gallery, and so on, will be put in the chooser list. Also, we can define general types of supported images by putting a slash followed by a star. For example, `image/*` will put all the image apps, not limited to only PNG, in the chooser list.

Finally, we start the activity with this intent by calling the `startActivity()` method. You should be careful to pass the intent in the `startActivity()` method or in the `Intent.chooser()` method. If we pass the intent in `startActivity()`, we will get the `ActivityNotFoundException` exception. We don't specifically know which activity to start in order to share the app; so, in that case, we will create a chooser list, and the user will decide which activity to start. This is how we provide the user with the option to choose his or her favorite method to share the content.

Now, run the project and you will see a screen that will have a **Share** button and a text field to write something to share. On pressing the **Share** button, you will see a dialog asking you to choose the method of sharing. This dialog will include all those apps, such as text messaging, SMS, MMS, e-mail, and Facebook, which are used to share the text. The following figure shows the app screens:



In this example, we used implicit intents to communicate with other apps of the Android OS. In the next example, we will see how other apps can communicate with our app.

Registering your app for the share intent

In the previous example, we triggered other sharing apps from our app. In this example, we will register our app for the share intent so that the other app developers and users can communicate with our app using implicit intents. Both examples use implicit intents, but the method of use is different in both cases.

In the previous example, we used implicit intents in Java files and triggered the intent on a button tap in `OnClickListener`. In this example, to register our app for any intent such as the share intent in this case, we have to put our code in the XML file in the `AndroidManifest.xml` file. Just to revise a little, the `AndroidManifest.xml` file manages all the settings of our app. Let's learn how to implement the app now.

In order to start this example, create an empty project using any Android IDE such as Eclipse with the ADT plugin or Android Studio, or open any existing project in which you want to receive the send intent. In this project, we will explore the use of implicit intents in getting the data shared by other apps in our app. We first register our app as a text-sharing app. Then, all the apps that share any `text/html` type file in Android can trigger our app if a user chooses it. To perform this task, we have modified the code in three files: a layout file, an activity file, and the manifest file. Let's look at those files one by one now.

The `activity_main.xml` file

The `activity_main.xml` file is the visual layout of our sharing app. The following code shows the implementation of this file:

```
<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"
        android:id="@+id/dataTextView" />

</RelativeLayout>
```

We have a Relative layout with a text View component in it. We have set the initial text value of Text View to "Hello World"; remember that the data we get from other apps while sharing will be printed in this Text View content. After the visual representation, it always comes to the processing and coding logic of app. Let's look at the next activity file which performs the main task and implements the logic of using implicit intents.

The MainActivity.java class

The MainActivity.java class file is the activity that displays the data which is shared from any other app. The following code shows the implementation of this file:

```
public class MainActivity extends Activity {

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

        // Get intent, action and MIME type
        Intent intent = getIntent();
        String action = intent.getAction();
        String type = intent.getType();

        if (Intent.ACTION_SEND.equals(action) && type != null)
            if ("text/html".equals(type)) {
                updateTextView(intent);
            }
    }

    public void updateTextView(Intent intent) {

        String sharedText = intent.getStringExtra(Intent.EXTRA_TEXT);
        if (sharedText != null)
        {
            TextView tv = (TextView) findViewById(R.id.dataTextView);
            tv.setText(sharedText);
        }
    }
}
```

When a user chooses our app from the **Share** dialog, this activity will be opened. We also have set this activity to our main activity. Any activity in the app can be registered for getting the share intent. It is not necessary that the main activity has to be registered for getting the shared data. We have set the layout of the activity, and after that, we get the intent data. This is the data that will be thrown from any other sharing app. We first get the intent by calling the `getIntent()` method. There are many types of intents; we have to make sure that our activity will work only for the intent type that we have registered in the `AndroidManifest.xml` file. To detect whether this intent was sent for sharing or not, we have to check the intent action. So, we have obtained the action of intent by calling the `Intent.getAction()` method. We can get the type of intent using the `Intent.getType()` method. Then, we check the type and action.

If the intent action is `Intent.ACTION_SEND` and the type is `text/html`, it means that we can display that type of data in our app. If both conditions are `true`, we set the Text View content to the data that we got from the intent. We can get the data from the intent through the `Intent.getStringExtra()` method. This method takes the data type as an input argument or parameter. In this example, we get the `Intent.EXTRA_TEXT` data type that represents any text or message data in the intent that is normally used for the body text in e-mails, Facebook posts, or SMS messages.

By understanding the `MainActivity` class, we saw that we have only got the intent and checked it. But there is also a problem regarding how the other apps are going to recognize our app and how they can know that our app can display the `text/html` data. If we open this activity explicitly from another activity of the same app, the same intent will be received and the same conditions will be checked. But this time, no condition will be `true`, so no text will be changed in the layout. To make the app visible to other apps, we have to register an intent filter. This is done in the `AndroidManifest.xml` file.

The AndroidManifest.xml file

To make our app visible for sharing content, we have to register the receiving activity of the app with an intent filter in this file. The following code shows the implementation of this file:

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

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

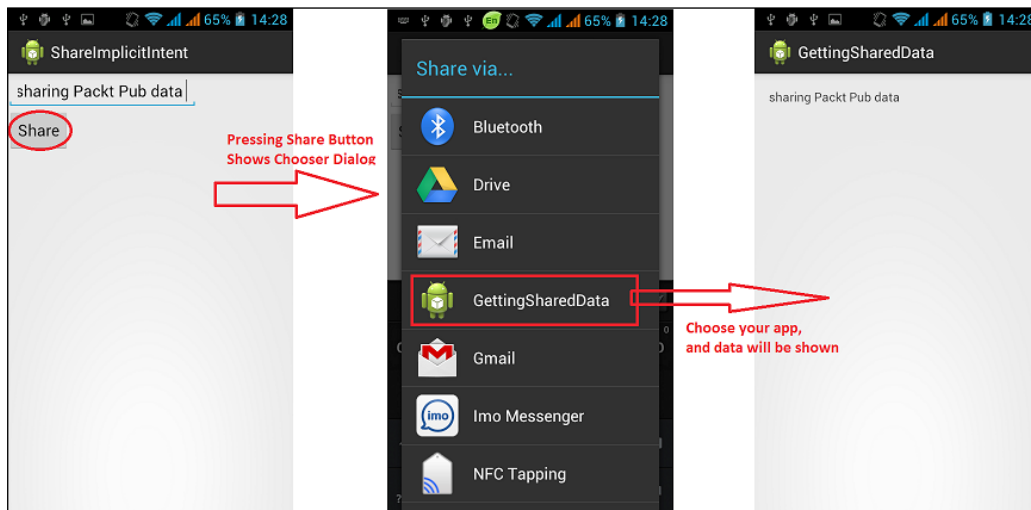
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.chapter3.gettingshareddata.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.SEND" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="text/html" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

In the `<activity>` tag of `MainActivity`, we have inserted two intent filters. The first intent filter is to make this activity a main activity and the launcher of the app. The second intent filter is a piece of code that performs the core functions of the app. We have inserted an intent filter with the action as `android.intent.action.SEND` and `mimeType` as `text/html`. This tells the Android OS that whenever any intent with the `Send` action is triggered and it contains data with `text/html` type, the app can process this intent. This is how our app is shown in the chooser dialog of the app.

Now, run the project and you will see the **Hello World** screen. Close the app and run our previous example app, **ShareImplicitIntent**. Write something in the text field and tap the **Share** button. In the chooser dialog, you will see our app, **GettingSharedData**, in the list. Choosing this app will open the activity, and this time, instead of **Hello World**, you will see the data shared from another app in the text field. The following screenshot shows a demo of the app:



So far, we have seen two examples of implicit intents. In one example, we shared some data with other apps such as e-mail, SMS, Facebook, and so on. In the other example, other apps shared their content with our app, and we received that data and displayed it. But, implicit intents are not limited to sharing content alone. There are lots of options and choices that can be performed using implicit intents, including making calls, sending texts, showing maps, searching for anything, taking pictures, showing and editing contacts, and so on.

In our next example, we will learn how we can pick any image from the gallery and display it in our activity.

Selecting an image through an implicit intent

In this project, we will implement the use of implicit intents to pick any image from the gallery. We will put an Image View that will display an image in our app. This image will be chosen by the user from the gallery. Let's implement it now! In order to start this example, create an empty project using any Android IDE such as Eclipse with the ADT plugin or Android Studio, or open any existing project in which you want to add the image picking feature.

We will register our app as an image-sharing app, and then all the apps which share any image in Android OS can trigger our app if the user chooses it. We have modified the code in three files: a layout file, an activity file, and the manifest file. Let's see what these files do.

The activity_main.xml file

Just like in all the Android apps, the activity_main.xml file represents the layout file of the main activity. The following code shows the implementation of this file:

```
<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" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="pickImage"
        android:text="Button" >
    </Button>

    <ImageView
        android:id="@+id/result"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/ic_launcher" >
    </ImageView>

</RelativeLayout>
```

We have placed two View components; a button View to open the gallery on a tap and an Image View to display the image. Unlike other apps, we have set the button listener in our layout file. Revising the last method, we set our click listener by calling the `button.setOnClickListener()` method in the activity class file. In this example, we have used the `android:onClick` attribute in the `<Button>` tag, and provided the name of the listener on the other side of the attribute. We have to provide a method name that should be defined in the activity file using this layout.



The Android OS recommends that you set listeners in the XML layout file. But, if the layout is used by more than one activity, developers should be careful as an attribute value is a method name and should be defined in the activity file. That means either all activities using the layout file should define that method, or all activities should set the listeners in Java files instead of XML.

The other View component in our layout file is Image View. This Image View will show the image picked from the gallery or other image sharing apps. We have set Image View's source to `launcher-icon` image as the default image.

After developing the layout of the app, let's focus on the logic of the app. The `MainActivity` file shows how the app gets the image from other apps and displays it.

The MainActivity.java class

The MainActivity.java class is the main activity Java file that performs all the functionalities in the app. The following code snippet is the implementation of this file:

```
public class MainActivity extends Activity {

    private static final int REQUEST_CODE = 1;
    private Bitmap bitmap;
    private ImageView imageView;

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

        imageView = (ImageView) findViewById(R.id.result);
    }

    public void pickImage(View View) {
        Intent intent = new Intent();
        intent.setType("image/*");
        intent.setAction(Intent.ACTION_GET_CONTENT);
        intent.addCategory(Intent.CATEGORY_OPENABLE);
        startActivityForResult(intent, REQUEST_CODE);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        InputStream stream = null;
        if (requestCode == REQUEST_CODE && resultCode == Activity.RESULT_OK)
            try {
                // We need to recycle unused bitmaps
                if (bitmap != null) {
                    bitmap.recycle();
                }
                stream = getContentResolver().openInputStream(data.getData());
                bitmap = BitmapFactory.decodeStream(stream);

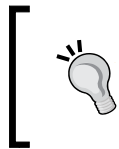
                imageView.setImageBitmap(bitmap);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } finally {
                if (stream != null)
                    try {
                        stream.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
            }
    }
}
```

We start from the `onCreate()` method where we first set the Content View of the activity to our layout file. We have created three private fields in our class. The first is the `REQUEST_CODE` constant that is an integer value. This constant is used as the request code to get any data from any other Android app. As we are picking an image from the gallery, we need a request code to identify the correct results and data. The second field is the `bitmap`. This `bitmap` is used to store the picked image in the `bitmap` format. The third and last field of the activity class is `ImageView`. This is used to reference `ImageView` in the XML file.

The `pickImage()` method is the button listener that was set in the XML layout file in the `<Button>` tag. This method should take the `view` parameter. This parameter contains the View that was tapped at runtime. As per our app requirements, we want to open the gallery on a button tap; so, to open the gallery, an implicit intent will be triggered in this method. We create an empty intent object with a no-argument constructor. Then, we set its type to any image format using `image/*`. After that, we set its intent action to `Intent.ACTION_GET_CONTENT`. This tells the Android OS to show all those apps that share the content.

Now, we already told the Android OS that we need only the image content by setting the type; so, the Android OS will only show those apps, such as gallery, that shares images. We set the category to `Intent.CATEGORY_OPENABLE`. This is used to indicate that the `GET_CONTENT` intent only wants the URIs that can be opened with `ContentResolver.openInputStream`.

Finally, we start the activity by calling the `startActivityForResult()` method. Remember that we used the `startActivity()` method in our previous apps.

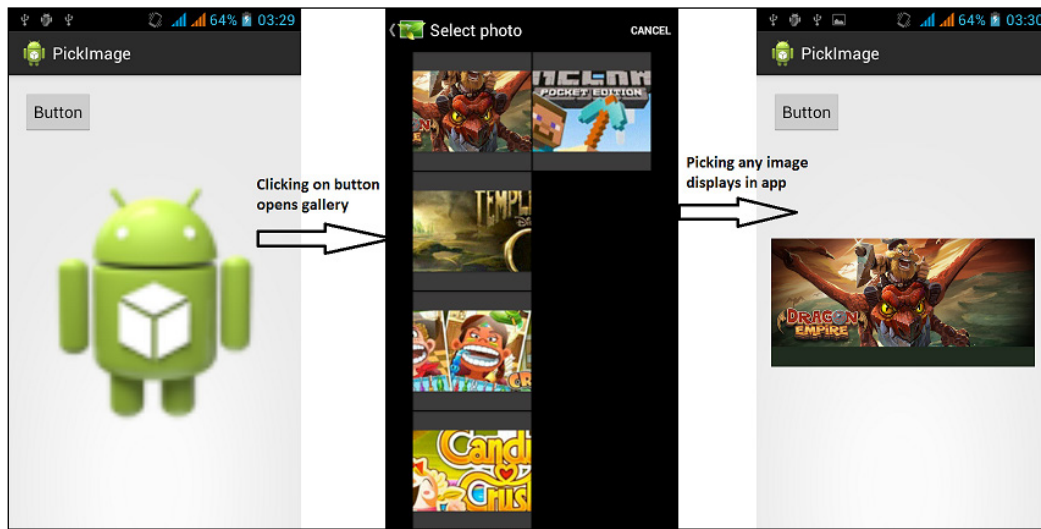


The difference between the `startActivity()` and `startActivityForResult()` methods is that the `startActivityForResult()` method returns some result to the parent activity after being stopped while `startActivity()` doesn't return anything.

As we need to get any image from gallery, the gallery app will return the URI of the image that we will use in our app to display it. To get the result in our activity, we need to override the `onActivityResult()` method in our class. This method takes three parameters. The first is a request-code that is an integer value. This value defines the request ID that we used to start an activity. We have used a constant private field `REQUEST_CODE` in our class for this value; so, in our `onActivityResult()` method, we will compare the request code to this constant value for confirmation. The second parameter, `RESULT_CODE`, is an integer value. This value tells us whether the result we have got is correct and okay to use or not. The third and last parameter is the intent; this contains the resulting data that we will use in our app.

In the `onActivityResult()` method, we created an `InputStream` object and then we compared our request code and result code to confirm whether we should process the intent data or not. If everything goes fine, we get the URI of the picked image by calling the `Intent.getData()` method and pass it to `openInputStream()` of the content resolver of this activity. The content resolver of any activity can be retrieved by calling the `Activity.getContentResolver()` method. After getting the stream of the URI, we decode it to bitmap by calling the `BitmapFactory.decodeStream()` method, and set the output bitmap to our activity-bitmap field. Then, we set the bitmap in our Image View. And in the final section of the `try/catch` block, we close our stream.

Now, run the project and you will see the screens as shown in following screenshot. The user taps on the button, and the gallery will be shown. Then, the user chooses his favorite photo to be displayed, and the app displays it on the app screen:



Summarizing the whole section of implicit intents, we implemented three examples. In the first example, we learned how to share our data with other apps. In the second example, we learned how other apps can share their data with our app. And finally, in the third and last example, we learned how to get an image from the gallery and use it in our app. In the next and last section of the chapter, we will discuss Android late binding.

Intents and Android late binding

As we all know, three of the most core components of an application in Android are activities, services, and broadcast receivers. These components communicate and are triggered via messaging. This messaging is done through intents. Intent messaging is a facility for late-runtime binding (late binding) between components in the same or different applications. In each case, an Android system finds the right component, such as an activity, service, or receiver to be triggered, and instants them if necessary. There is no overlapping within these intents. For example, broadcast receiver intents are only sent to broadcast receivers and never sent to any activity or service. Another example is that an intent passed in the `startActivity()` or `startActivityForResult()` method is never sent to any component such as a service or receiver, but only to an activity.

In the examples used in this chapter, implicit intents always performed actions in which the developer was not sure about how these actions will be performed and with what apps. This runtime behavior of assigning actions to components is called Android late-runtime binding, and this can be done easily via implicit intents.

Summary

In this chapter, we discussed the categories of intents, implicit intents, explicit intents, and late binding. The chapter also provided some important implementation of Android intents in which we shared our data with other apps, the other apps shared data with our app, picked any image from gallery, started an activity or service through explicit intents, and so on.

In the next chapter, we will learn how mobile components such as a camera, can be triggered by intents, and how they are used in our apps.

4

Intents for Mobile Components

In the last chapter, we discussed the categories of intents and how different categories are used. We also discussed the pros and cons of categories such as implicit intents and explicit intents. But besides the theory regarding the intents that we have been discussing until now, it's time to discuss some applications of intents with a more practical approach. In this chapter, we will discuss the mobile components that are commonly found in all Android phones. And we will see how those mobile components can be accessed and used very easily via intents. Android provides a vast collection of libraries and features through which a developer can utilize mobile components. This is as easy as a walk in the park. This chapter mainly includes four different categories of components: visual components such as camera, communication components such as Wi-Fi and Bluetooth, media components such as video and audio recording, speech recognition and text-to-speech conversion, and finally, motion components such as proximity sensor. The following topics will be discussed in this chapter:

- Common mobile components
- Components and intents
- Communication components
- Using Bluetooth through intents
- Using Wi-Fi through intents
- Media components
- Taking pictures and recording video through intents
- Speech recognition using intents
- Role of intents in text-to-speech conversion
- Motion components
- Proximity alerts through intents

The concepts and structures of intents, as discussed in the previous chapters, are the prerequisites for understanding this chapter and the later chapters. If you don't have a basic understanding of these things, we would recommend you to read *Chapter 2, Introduction to Android Intents* and *Chapter 3, Intents and Its Categorization* in order to move forward.

Common mobile components

Due to the open source nature of the Android operating system, many different companies such as HTC and Samsung ported the Android OS on their devices with many different functionalities and styles. Each Android phone is unique in some way or the other and possesses many unique features and components different from other brands and phones. But there are some components that are found to be common in all the Android phones.



We are using two key terms here: components and features. **Component** is the hardware part of an Android phone, such as camera, Bluetooth and so on. And **Feature** is the software part of an Android phone, such as the SMS feature, E-mail feature, and so on. This chapter is all about hardware components, their access, and their use through intents.

These common components can be generally used and implemented independently of any mobile phone or model. And there is no doubt that intents are the best asynchronous messages to activate these Android components. These intents are used to trigger the Android OS when some event occurs and some action should be taken. Android, on the basis of the data received, determines the receiver for the intent and triggers it. Here are a few common components found in each Android phone:

The Wi-Fi component

Each Android phone comes with a complete support of the Wi-Fi connectivity component. The new Android phones having Android Version 4.1 and above support the Wi-Fi Direct feature as well. This allows the user to connect to nearby devices without the need to connect with a hotspot or network access point.

The Bluetooth component

An Android phone includes Bluetooth network support that allows the users of Android phones to exchange data wirelessly in low range with other devices. The Android application framework provides developers with the access to Bluetooth functionality through Android Bluetooth APIs.

The Cellular component

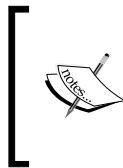
No mobile phone is complete without a cellular component. Each Android phone has a cellular component for mobile communication through SMS, calls, and so on. The Android system provides very high, flexible APIs to utilize telephony and cellular components to create very interesting and innovative apps.

Global Positioning System (GPS) and geo-location

GPS is a very useful but battery-consuming component in any Android phone. It is used for developing location-based apps for Android users. Google Maps is the best feature related to GPS and geo-location. Developers have provided so many innovative apps and games utilizing Google Maps and GPS components in Android.

The Geomagnetic field component

Geomagnetic field component is found in most Android phones. This component is used to estimate the magnetic field of an Android phone at a given point on the Earth and, in particular, to compute magnetic declination from the North.



The geomagnetic field component uses the **World Magnetic Model** produced by United States National Geospatial-Intelligence Agency. The current model that is being used for the geomagnetic field is valid until 2015. Newer Android phones will have the newer version of the geomagnetic field.

Sensor components

Most Android devices have built-in sensors that measure motion, orientation, environment conditions, and so on. These sensors sometimes act as the brains of the app. For example, they take actions on the basis of the mobile's surrounding (weather) and allow users to have an automatic interaction with the app. These sensors provide raw data with high precision and accuracy for measuring the respective sensor values. For example, gravity sensor can be used to track gestures and motions, such as tilt, shake, and so on, in any app or game. Similarly, a temperature sensor can be used to detect the mobile temperature, or a geomagnetic sensor (as introduced in the previous section) can be used in any travel application to track the compass bearing. Broadly, there are three categories of sensors in Android: motion, position, and environmental sensors. The following subsections discuss these types of sensors briefly.

Motion sensors

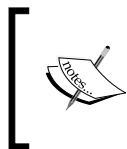
Motion sensors let the Android user monitor the motion of the device. There are both hardware-based sensors such as accelerometer, gyroscope, and software-based sensors such as gravity, linear acceleration, and rotation vector sensors. Motion sensors are used to detect a device's motion including tilt effect, shake effect, rotation, swing, and so on. If used properly, these effects can make any app or game very interesting and flexible, and can prove to provide a great user experience.

Position sensors

The two position sensors, geomagnetic sensor and orientation sensor, are used to determine the position of the mobile device. Another sensor, the proximity sensor, lets the user determine how close the face of a device is to an object. For example, when we get any call on an Android phone, placing the phone on the ear shuts off the screen, and when we hold the phone back in our hands, the screen display appears automatically. This simple application uses the proximity sensor to detect the ear (object) with the face of the device (the screen).

Environmental sensors

These sensors are not used much in Android apps, but used widely by the Android system to detect a lot of little things. For example, the temperature sensor is used to detect the temperature of the phone, and can be used in saving the battery and mobile life.



At the time of writing this book, the Samsung Galaxy S4 Android phone has been launched. The phone has shown a great use of environmental gestures by allowing users to perform actions such as making calls by no-touch gestures such as moving your hand or face in front of the phone.

Components and intents

Android phones contain a large number of components and features. This becomes beneficial to both Android developers and users. Android developers can use these mobile components and features to customize the user experience. For most components, developers get two options; either they extend the components and customize those according to their application requirements, or they use the built-in interfaces provided by the Android system. We won't read about the first choice of extending components as it is beyond the scope of this book. However, we will study the other option of using built-in interfaces for mobile components.

Generally, to use any mobile component from our Android app, the developers send intents to the Android system and then Android takes the action accordingly to call the respective component. As we have discussed earlier, intents are asynchronous messages sent to the Android OS to perform any functionality. Most of the mobile components can be triggered by intents just by using a few lines of code and can be utilized fully by developers in their apps. In the following sections of this chapter, we will see few components and how they are used and triggered by intents with practical examples. We have divided the components in three ways: communication components, media components, and motion components. Now, let's discuss these components in the following sections.

Communication components

Any mobile phone's core purpose is communication. Android phones provide a lot of features other than communication features. Android phones contain SMS/MMS, Wi-Fi, and Bluetooth for communication purposes. This chapter focuses on the hardware components; so, we will discuss only Wi-Fi and Bluetooth in this chapter. The Android system provides built-in APIs to manage and use Bluetooth devices, settings, discoverability, and much more. It offers full network APIs not only for Bluetooth but also for Wi-Fi, hotspots, configuring settings, Internet connectivity, and much more. More importantly, these APIs and components can be used very easily by writing few lines of code through intents. We will start by discussing Bluetooth, and how we can use Bluetooth through intents in the next section.

Using Bluetooth through intents

Bluetooth is a communication protocol that is designed for short-range, low-bandwidth, peer-to-peer communication. In this section, we will discuss how to interact and communicate with local Bluetooth devices and how we can communicate with the nearby, remote devices using Bluetooth. Bluetooth is a very low-range protocol, but it can be used to transmit and receive data such as files, media, and so on. As of Android 2.1, only paired devices can communicate with each other via Bluetooth devices due to encryption of the data.



Bluetooth APIs and libraries became available from Android 2.0 Version (SDK API Level 5). It should also be noted that not all Android phones will necessarily include the Bluetooth hardware.

The Bluetooth API provided by the Android system is used to perform a lot of actions related to Bluetooth that includes turning the Bluetooth on/off, pairing with nearby devices, communicating with other Bluetooth devices, and much more. But, not all of these actions can be performed through intents. We will discuss only those actions that can be performed through intents. These actions include setting the Bluetooth On/Off from our Android app, tracking the Bluetooth adapter state, and making our device discoverable for a small time. The actions that can't be performed through intents include sending data and files to other Bluetooth devices, pairing with other devices, and so on. Now, let's explain these actions one by one in the following sections.

Some Bluetooth API classes

In this section, we will discuss some classes from the Android Bluetooth API that are used in all Android apps using Bluetooth. Understanding these classes will help the developers understand the following examples more easily.

BluetoothDevice

This class represents each remote device with which the user is communicating. This class is a thin wrapper for the Bluetooth hardware of the phone. To perform the operations on the object of this class, developers have to use the `BluetoothAdapter` class. The objects of this class are immutable. We can get `BluetoothDevice` by calling `BluetoothAdapter.getRemoteDevice(String macAddress)` and passing the MAC address of any device. Some important methods of this class are:

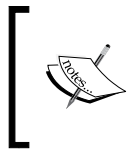
- `BluetoothDevice.getAddress()`: It returns the MAC address of the current device.
- `BluetoothDevice.getBondState()`: It returns the bonding state of the current device, such as not bonded, bonding, or bonded.



The **MAC address** is a string of 12 characters represented in the form of `xx:xx:xx:xx:xx:xx`. For example, `00:11:22:AA:BB:CC`.

BluetoothAdapter

This class represents the current device on which our Android app is running. It should be noted that the `BluetoothAdapter` class represents the current device, and the `BluetoothDevice` class represents the other devices that can or cannot be bonded with our device. This class is a singleton class and cannot be instantiated. To get the object of this class, we can use the `BluetoothAdapter.getDefaultAdapter()` method. To perform any action related to Bluetooth communication, this class is the main starting point for it. Some of the methods of this class include `BluetoothAdapter.getBondedDevices()`, which returns all paired devices, `BluetoothAdapter.startDiscovery()`, which searches for all discoverable devices nearby, and so on. There is a method called `startLeScan(BluetoothAdapter.LeScanCallback callback)` that is used to receive a callback whenever a device is discovered. This method was introduced in API Level 18.



Some of the methods in the `BluetoothAdapter` and `BluetoothDevice` classes require the `BLUETOOTH` permission, and some require the `BLUETOOTH_ADMIN` permission as well. So, when using these classes in your app, don't forget to add these permissions in your Android manifest file.

So far, we have discussed some Bluetooth classes in the Android OS along with some of the methods in those classes. In the next section, we will develop our first Android app that will ask the user to turn on the Bluetooth.

Turning on the Bluetooth app

To perform any Bluetooth action, Bluetooth must first be turned on. So, in this section, we will develop an Android app that will ask the user to turn on the Bluetooth device if it is not already on. The user can accept it and the Bluetooth will be turned on, or the user can also reject it. In the latter case, the application will continue and the Bluetooth will remain in the off state. It would be great to say that this action can be performed very easily using intents. Let's see how we can do this by looking at the code.

First, create an empty Android project in your favourite IDE. We have developed it in Android Studio. At the time of writing this book, the project is in the Preview Mode, and its beta launch is expected soon. Now, we will modify a few files from the project to make our Android Bluetooth app. We will modify two files. Let's see those files in the following sections.

The MainActivity.java file

This class represents the main activity of our Android app. The following code is implemented in this class:

```
public class MainActivity extends Activity {

    final int BLUETOOTH_REQUEST_CODE = 0;

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

        String enableBT = BluetoothAdapter.ACTION_REQUEST_ENABLE;
        Intent bluetoothIntent = new Intent(enableBT);
        startActivityForResult(bluetoothIntent, BLUETOOTH_REQUEST_CODE);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data)
    {
        super.onActivityResult(requestCode, resultCode, data);

        if (resultCode == RESULT_OK)
        {
            if (requestCode == BLUETOOTH_REQUEST_CODE)
            {
                Toast.makeText(this, "Turned On", Toast.LENGTH_SHORT).show();
            }
        }
        else if (resultCode == RESULT_CANCELED)
        {
            Toast.makeText(this, "Didn't Turn On", Toast.LENGTH_SHORT).show();
        }
    }
}
```

In our activity, we have declared a constant value with the name `BLUETOOTH_REQUEST_CODE`. This constant is used as a request code or request unique identifier in the communication between our app and the Android system. When we request the Android OS to perform some action, we pass any request code. Then, the Android system performs the action and returns the same request code back to us. After comparing our request code with Android's request code, we get to know about the action that has been performed. If the code doesn't match, it means that this action is for some other request. It is not our request. In the `onCreate()` method, we set the layout of the activity by calling the `setContentView()` method. And then, we perform our real task in the next few lines.

We create a string `enableBT` that gets the value of the `ACTION_REQUEST_ENABLE` method that pertains to the `BluetoothAdapter` class. This string is passed in the intent constructor to tell the intent that it is meant to enable the Bluetooth device. Like the Bluetooth-enable request string, the Android OS also contains many other requests for various actions such as Wi-Fi, Sensors, Camera, and more. In this chapter, we will learn about a few request strings. After creating the request string, we create our intent and pass the request string to it. And then, we start our intent by passing it in the `startActivityForResult()` method.

One thing to note here is that in the previous chapters, we used the `startActivity()` method instead of the `startActivityForResult()` method. Basically, the `startActivity()` method just starts any activity that is passed through intents, but the `startActivityForResult()` method starts any activity, and after performing some action, it returns to the original activity and presents the results of the action. So, in this example, we called the activity that requests the Android system to enable the Bluetooth device. The Android system performs the action and asks the user whether it should enable the device or not. Then, the Android system returns the result to the original activity that started the intent earlier. To get any result from other activities to our activity, we override the `onActivityResult()` method. This method is called after returning from other activities. The method contains three parameters: `requestCode`, `resultCode`, and `dataIntent`. The `requestCode` parameter is an integer value and contains the request code value of the request provided by the developer. The `resultCode` parameter is the result of the action. It tells the developer whether the action has been performed successfully with a positive response or with a negative response. The `dataIntent` object contains the original calling-intent data, such as which activity started the intent and all the related information. Now, let's see our overridden method in detail. We have first checked whether `requestCode`, our request code, is `BLUETOOTH_REQUEST_CODE`, or not. If both are the same, we have compared the result code to check whether our result is okay or not. If it is okay, it means that Bluetooth has been enabled; so, we display a toast notifying the user about it, and if the result is not okay, that means Bluetooth has not been enabled. Here also we notify the user by displaying a toast.

This was the activity class that performs the core functionality of our Bluetooth-enabling app. Now, let's see the Android manifest file in the following section.

The AndroidManifest.xml file

The AndroidManifest.xml file contains all the necessary settings and preferences for the app. The following is the code contained in this manifest file:

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

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

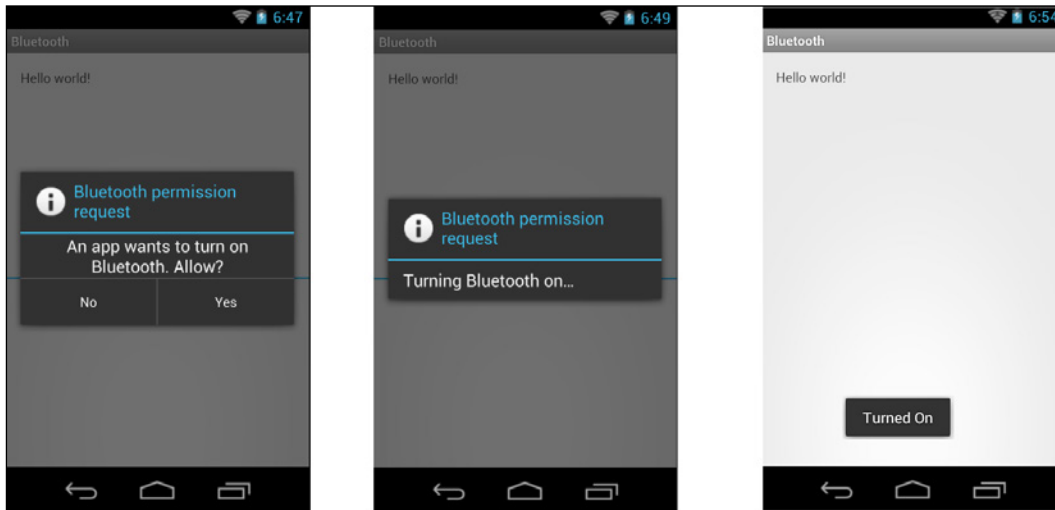
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>

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

Any Android application that uses the Bluetooth device must have the permission of Bluetooth usage. So, to provide the permission to the user, the developer declares the `<uses-permission>` tag in the Android manifest file and writes the necessary permissions. As shown in the code, we have provided two permissions: `android.permission.BLUETOOTH` and `android.permission.BLUETOOTH_ADMIN`. For most Bluetooth-enabled apps, only the `BLUETOOTH` permission does most of the work. The `BLUETOOTH_ADMIN` permission is only for those apps that use Bluetooth admin settings such as making the device discoverable, searching for other devices, pairing, and so on. When the user first installs the application, he is provided with details about which permissions are needed for the app. If the user accepts and grants the permissions to the app, the app gets installed; otherwise, the user can't install the app. The rest of the file is the same as in the other examples in the book.

After discussing the Android manifest and activity files, we would test our project by compiling and running it. When we run the project, we should see the screens as shown in the following screenshots:



Enabling Bluetooth App

As the app starts, the user is presented with a dialog to enable or disable the Bluetooth device. If the user chooses **Yes**, the Bluetooth is turned on, and a toast updates the status by displaying the status of the Bluetooth.

Tracking the Bluetooth adapter state

In the previous example, we saw how we can turn on the Bluetooth device just by passing the intent of the Bluetooth request to the Android system in just a few lines. But enabling and disabling the Bluetooth are time-consuming and asynchronous operations. So, instead of polling the state of the Bluetooth adapter, we can use a broadcast receiver for the state change. In this example, we will see how we can track the Bluetooth state using intents in a broadcast receiver.

This example is the extension of the previous example, and we will use the same code and add a new code to it. Let's look at the code now. We have three files, `MainActivity.java`, `BluetoothStateReceiver.java`, and `AndroidManifest.xml`. Let's discuss these files one by one.

The MainActivity.java file

This class represents the main activity of our Android app. The following code is implemented in this class:

```
public class MainActivity extends Activity {
    final int BLUETOOTH_REQUEST_CODE = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        registerReceiver(new BluetoothStateReceiver(), new IntentFilter(
            BluetoothAdapter.ACTION_STATE_CHANGED));
        String enableBT = BluetoothAdapter.ACTION_REQUEST_ENABLE;
        Intent bluetoothIntent = new Intent(enableBT);
        startActivityForResult(bluetoothIntent,
            BLUETOOTH_REQUEST_CODE);
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode,
        Intent data) {

        // TODO Auto-generated method stub
        super.onActivityResult(requestCode, resultCode, data);
        if (resultCode == RESULT_OK) {
            if (requestCode == BLUETOOTH_REQUEST_CODE) {
                Toast.makeText(this, "Turned On", Toast.LENGTH_SHORT).show();
            }
        }
        else if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this, "Didn't Turn On", Toast.LENGTH_SHORT).
show();
        }
    }
}
```

From the code, it is clear that the code is almost the same as in the previous example. The only difference is that we have added one line after setting the content view of the activity. We called the `registerReceiver()` method that registers any broadcast receiver with the Android system programmatically. We can also register the receivers via XML by declaring them in the Android manifest file. A broadcast receiver is used to receive the broadcasts sent from the Android system.

While performing general actions such as turning the Bluetooth on, turning the Wi-Fi on/off and so on, the Android system sends broadcast notifications that can be used by developers to detect the state changes in the mobile. There are two types of broadcasts. Normal broadcasts are completely asynchronous. The receivers of these broadcasts run in a disorderly manner, and multiple receivers can receive broadcasts at the same time. These broadcasts are more efficient as compared to the other type of broadcasts that are ordered broadcasts. Ordered broadcasts are sent to one receiver at a time. As each receiver receives the results, it passes the results to the next receiver or completely aborts the broadcast. In this case, other receivers don't receive the broadcast.

Although the `Intent` class is used for sending and receiving broadcasts, the intent broadcast is a completely different mechanism and is separate from the intents used in the `startActivity()` method. There is no way for the broadcast receiver to see or capture the intents used with the `startActivity()` method. The main difference between these two intent mechanisms is that the intents used in the `startActivity()` method perform the foreground operation that the user is currently engaged in. However, the intent used with the broadcast receivers performs some background operations that the user is not aware of.

In our activity code, we used the `registerReceiver()` method to register an object of our customized broadcast receiver defined in the `BluetoothStateReceiver` class, and we passed an intent filter `BluetoothAdapter.ACTION_STATE_CHANGED` according to the type of the receiver. This state tells the intent filter that our object of the broadcast receiver is used in detecting the Bluetooth state change in the app. After the Register receiver, we created an intent passing `BluetoothAdapter.ACTION_REQUEST_ENABLE`, telling the app to turn on the Bluetooth. Finally, we start our action by calling `startActivityForResult()`, and we compare the results in the `onActivityResult()` method to see whether the Bluetooth is turned on or not. You can read about these processes in the previous example of this chapter.



When you register the receiver in the `onCreate()` or `onResume()` method of the activity, you should unregister it in the `onPause()` or `onDestroy()` method. The advantage of this approach is that you won't receive any broadcasts when the app is paused or closed, and this can decrease Android's unnecessary overhead operations resulting in a better battery life.

Now, let's see the code of our customized broadcast receiver class.

The BluetoothStateReceiver.java file

This class represents our customized broadcast receiver that tracks the state change in the Bluetooth device. The following code shows the implementation of the file:

```
public class BluetoothStateReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub

        String stateExtra = BluetoothAdapter.EXTRA_STATE;
        int state = intent.getIntExtra(stateExtra, -1);
        int prevState = intent.getIntExtra(prevStateExtra, -1);

        String statusText = "";
        if (state == BluetoothAdapter.STATE_TURNING_ON)
            statusText = "Turning On Bluetooth";
        else if (state == BluetoothAdapter.STATE_ON)
            statusText = "Bluetooth is ON";
        else if (state == BluetoothAdapter.STATE_TURNING_OFF)
            statusText = "Turning Off Bluetooth";
        else if (state == BluetoothAdapter.STATE_OFF)
            statusText = "Bluetooth is OFF";

        Toast.makeText(context, statusText, Toast.LENGTH_SHORT).show();
    }
}
```

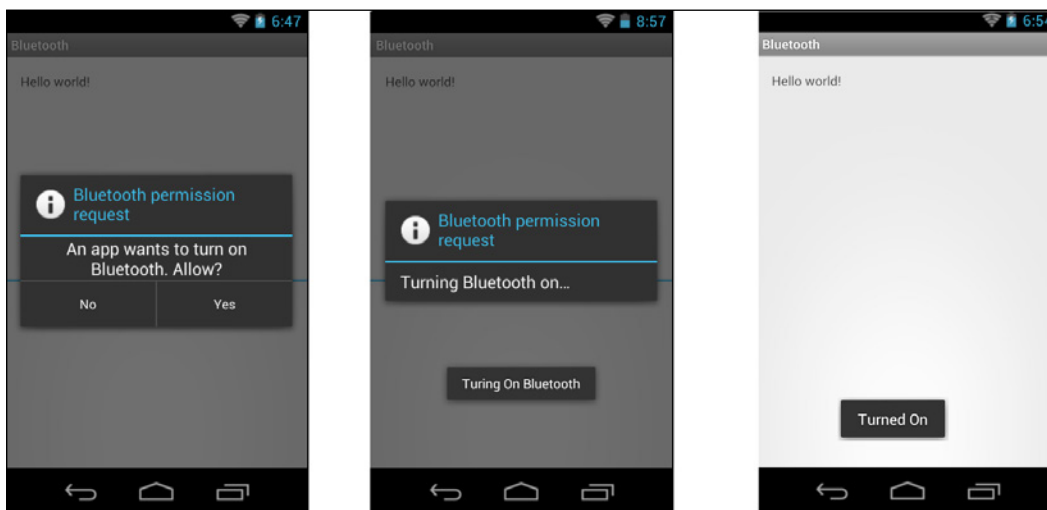
Just as we did for activities and services, to create a custom broadcast receiver, we extend from the `BroadcastReceiver` class and override methods to declare the custom behavior. We have overridden the `onReceive()` method and performed the main functionality of tracking the Bluetooth device status in this method. First, we will create a string variable to store the string value of the current state. To retrieve the string value, we have used `BluetoothAdapter.EXTRA_STATE`. Now, we can pass this value in the `get()` method of the intent to get our required data. As our states are integers and also extras, we have called `Intent.getIntExtra()` and passed our required string in it along with its default value as `-1`. Now, as we have got the current state code, we can compare these codes with the pre-defined codes in `BluetoothAdapter` to see the state of the Bluetooth device. There are four predefined states.

- `STATE_TURNING_ON`: This state notifies the user that the Bluetooth turn-on operation is in progress.
- `STATE_ON`: This state notifies the user that Bluetooth has already been turned on.

- `STATE_TURNING_OFF`: This state notifies the user that the Bluetooth device is being turned off.
- `STATE_OFF`: This state notifies the user that the Bluetooth has been turned off.

We compare our state with these constants, and display a toast according to the result we get. The Android manifest file is the same as in the previous example.

Thus, in a nutshell, we discussed how we can enable the Bluetooth device and ask the user to turn it on or off through intents. We also saw how to track the state of the Bluetooth operations using intents in the broadcast receiver and displaying the toasts. The following screenshots show the application demo:



Enabling the Bluetooth app

Being discoverable

So far, we have only been interacting with Bluetooth by turning it on or off. But, to start communication via Bluetooth, one's device must be discoverable to start pairing. We will not create any example for this application of intents, but we will only explain how this can be done via intents. To turn on Bluetooth, we used the `BluetoothAdapter.ACTION_REQUEST_ENABLE` intent. We passed the intent in the `startActivityForResult()` method and checked the result in the `onActivityResult()` method. Now, to make the device discoverable, we can pass the `BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE` string in the intent. And then, we pass this intent in the `startActivityForResult()` method, and track the result in the `onActivityResult()` method to compare the results.

The following code snippet shows the intent-creation process for making a device discoverable:

```
public final int DISCOVERY_REQUEST_CODE = 0;
String aDiscoverable = BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE;
startActivityForResult(new Intent(aDiscoverable), DISCOVERY_REQUEST_CODE);
```

In the code, you can see that there is nothing new that hasn't been discussed earlier. Only the intent action string type has been changed, and the rest is the same. This is the power of intents; you can do almost anything with just a few lines of code in a matter of minutes.

Monitoring the discoverability modes

As we tracked the state changes of Bluetooth, we can also monitor the discoverability mode using exactly the same method explained earlier in this chapter. We have to create a customized broadcast receiver by extending the `BroadcastReceiver` class. In the `onReceive()` method, we will get two extra strings: `BluetoothAdapter.EXTRA_PREVIOUS_SCAN_MODE`, and `BluetoothAdapter.EXTRA_SCAN_MODE`. Then, we pass those strings in the `Intent.getIntExtra()` method to get the integer values for the mode, and then we compare these integers with the predefined modes to detect our mode. The following code snippet shows the code sample:

```
registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        String prevScanMode = BluetoothAdapter.EXTRA_PREVIOUS_SCAN_MODE;
        String scanMode = BluetoothAdapter.EXTRA_SCAN_MODE;
        int scanMode = intent.getIntExtra(scanMode, -1);
        int prevMode = intent.getIntExtra(prevScanMode, -1);
    }
},
new IntentFilter(BluetoothAdapter.ACTION_SCAN_MODE_CHANGED));
```

Communication via Bluetooth

The Bluetooth communication APIs are just wrappers around the standard **RFCOMM**, the standard Bluetooth radio frequency communications protocol. To communicate with other Bluetooth devices, they must be paired with each other. We can carry out a bidirectional communication via Bluetooth using the `BluetoothServerSocket` class that is used to establish a listening socket for initiating a link between devices and `BluetoothSocket` that is used to create a new client socket to listen to the Bluetooth server socket. This new client socket is returned by the server socket once a connection is established. We will not discuss how Bluetooth is used in communication because it is beyond the scope of this book.

Using Wi-Fi through intents

Today, the era of the Internet and its vast usage in mobile phones have made worldwide information available on the go. Almost every Android phone user expects an optimal use of the Internet from all apps. It becomes the developer's responsibility to add Internet access in the app. For example, when users use your apps, they would like to share the use and their activities performed in your app, such as completing any level of a game or reading any article from any news app, with their friends on various social networks, or by sending messages and so on. So, if users don't get connected through your app to the Internet, social platforms, or worldwide information, then the app becomes too limited and maybe boring.

To perform any activity that uses the Internet, we first have to deal with Internet connectivity itself, such as whether the phone has any active connection. In this section, we will see how we can access Internet connectivity through our core topic — the intents. Like Bluetooth, we can do much work through intents related to Internet connectivity. We will implement three main examples: to check the Internet status of a phone, to pick any available Wi-Fi network, and to open the Wi-Fi settings. Let's start our first example of checking the Internet connectivity status of a phone using intents.

Checking the Internet connectivity status

Before we start coding our example, we need to know some important things. Any Android phone connected to the Internet can have any type of connection. Mobile phones can be connected using data connection to the Internet or it can be any open or secured Wi-Fi. Data connection is called mobile connection, and is connected via the mobile network provided by the SIM and service providers. In this example, we will detect whether the mobile phone is connected to any network or not, and if it is connected, which type of network it is connected to. Let's implement the code now.

There are two main files that perform the functionality of the app: `NetworkStatusReceiver.java` and `AndroidManifest.xml`. You might be wondering about the `MainActivity.java` file. In the following example, this file is not used because of the requirements of the app. What we are going to do in this example is that whenever the Internet connectivity status of a phone is changed, such as the Wi-Fi is turned on or off, this app will display a toast showing the status. The app will be performing its work in the background; so, activity and layouts are not needed in this app. Now, let's explain these files one by one:

The NetworkStatusReceiver.java file

This class represents our customized broadcast receiver that tracks the state change in the network connectivity of the device. The following code shows the implementation of the file:

```
public class NetworkStatusReceiver extends BroadcastReceiver {

    public void onReceive(Context context, Intent intent) {

        Toast.makeText(context, "Network Connectivity Status Changed!", Toast.LENGTH_SHORT).show();

        Bundle extras = intent.getExtras();
        boolean noNetwork = intent.getBooleanExtra(ConnectivityManager.EXTRA_NO_CONNECTIVITY, false);

        if(extras != null)
        {
            String networkInfoString = ConnectivityManager.EXTRA_NETWORK_INFO;
            NetworkInfo netInfo = (NetworkInfo) extras.get(networkInfoString);
            if(netInfo != null && netInfo.getState() == NetworkInfo.State.CONNECTED)
            {
                Toast.makeText(context, "Network " + netInfo.getTypeName() + " is Connected!",
                    Toast.LENGTH_SHORT).show();
            }
            else if(noNetwork)
            {
                Toast.makeText(context, "There is not any network connected!", Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

Just as we did for activities and services, to create a custom broadcast receiver, we extend from the `BroadcastReceiver` class and override methods to declare the custom behavior. We have overridden the `onReceive()` method, and we are performing the main functionality of tracking the Wi-Fi device status in this method. We have registered this receiver in the Android manifest file as a network status change, and we will discuss that file in the next section. This `onReceive()` method is called only when the network status is changed. So, we first display a toast stating that the network connectivity status has changed.



It must be noted that any broadcast receiver cannot be passed using `this` in the context parameter of `Toast` as we used to do in the `Activity` class because the `BroadcastReceiver` class doesn't extend the `Context` class like the `Activity` class.

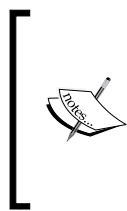
We have already notified the user about the network status changes, but we still have not notified the user about which change has occurred. So, at this point, our intent object becomes handy. It contains all the information and data of the network in the form of `extra` objects. Recalling from the previous chapters, `extra` is an object of the `Bundle` class. We create a local `Bundle` reference and store the intent `extra` objects in it by calling the `getExtras()` method. Along with it, we also store the no connectivity `extra` object in a `boolean` variable. `EXTRA_NO_CONNECTIVITY` is the lookup key for a `boolean` variable that indicates whether there is a complete lack of network connectivity, that is, whether any network is available, or not. If this value is `true`, it means that there is no network available.

After storing our required `extra` objects, we need to check whether the `extra` objects are available or not. So, we have checked the `extra` objects with `null`, and if the `extra` objects are available, we extract more network information from these `extra` objects. In the Android system, the developer is told about the data of interest in the form of constant strings. So, we first get our constant string of network information, which is `EXTRA_NETWORK_INFO`. We store it in a string variable, and then we use it as a key value parameter in the `get()` method of the `extra` objects. The `Bundle.get()` method returns an `Object` type of the object, and we need to typecast it to our required class. We are looking for network information; so, we are using the `NetworkInfo` class object.



The `Intent.EXTRA_NETWORK_INFO` string was deprecated in API Level 14. Since `NetworkInfo` can vary based on the **User ID (UID)**, the application should always obtain the network information through the `getActiveNetworkInfo()` or `getAllNetworkInfo()` method.

We have got all our values and data of interest; now, we will compare and check the data to find the connectivity status. We check whether this `NetworkInfo` data is `null` or not. If it is not `null`, we check whether the network is connected by checking the value from the `getState()` method of `NetworkInfo`. The `NetworkInfo.State` state that represents the coarse-grained network state is an enum. If the `NetworkInfo.State` enum is equal to `NetworkInfo.State.CONNECTED`, it means that the phone is connected to any network. Remember that we still don't know which type of network we are connected to. We can find the type of network by calling the `NetworkInfo.getTypeName()` method. This method will return `Mobile` or `Wi-Fi` in the respective cases.



Coarse-grained network state is mostly used in apps rather than `DetailedState`. The difference between these two states' mapping is that the coarse-grained network only shows four states: `CONNECTING`, `CONNECTED`, `DISCONNECTING`, and `DISCONNECTED`. However, `DetailedState` shows other states for more details, such as `IDLE`, `SCANNING`, `AUTHENTICATING`, `UNAVAILABLE`, `FAILED`, and the other four coarse-grained states.

The rest is an `if-else` block checking the state of the network and showing the relative toasts of status on the screen. Overall, we first extracted our `extra` objects from intent, stored them in local variables, extracted network info from extras, checked the state, and finally displayed the info in the form of toasts. Now, we will discuss the Android manifest file in the next section.

The AndroidManifest.xml file

As we have used a broadcast receiver in our application to detect the network connectivity status, it is necessary to register and unregister the broadcast receiver in the app. In our manifest file, we have performed two main tasks. First, we have added the permissions of accessing the network state. We have used `android.permissions.ACCESS_NETWORK_STATE`. Second, we have registered our receiver in the app using the receiver tag and added the name of the class.

Also, we have added the intent filters. These intent filters define the purpose of the receiver, such as what type of data should be received from the system. We have used the `android.net.conn.CONNECTIVITY_CHANGE` filter action for detecting the network connectivity change broadcast. There is nothing new in this file other than these two, and the rest of the code is the same as we have discussed in the previous chapters. The following is the code implementation of the file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.gamyguru.wifiexample"
    android:versionCode="1" android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="15" />

    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

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

        <receiver android:name=".NetworkStatusReceiver">
            <intent-filter>
                <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
            </intent-filter>
        </receiver>

    </application>
</manifest>
```

Summarizing the details of the preceding app, we created a customized broadcast receiver, and defined our custom behavior of network change, that is, displaying toasts, and then we registered our receiver in the manifest file along with the declarations of the required permissions. The following screenshots show a simple demo of the app when turning the Wi-Fi on in the phone:



The Network Change Status app

In the previous screenshot, we can see that when we turn the Wi-Fi on, the app displays a toast saying that the network status has changed. And after that toast, it displays the change; in our case, the Wi-Fi is connected. You might be wondering about the role of intents in this app. This app was not possible without using intents. The first use of intents was in registering the receiver in the manifest file to filter it for network status change. The other use of intents was in the receiver when we have received the update and we want to know the change. So, we used the intents and extracted the data from it in the form of `extra` objects and used it for our purpose. We didn't create our own intents in this example; instead, we only used the provided intents. In our next example, we will create our own intents and use them to open the Wi-Fi settings from our app.

Opening the Wi-Fi Settings app

Until now, we have only used intents for network and Wi-Fi purposes. In this example, we are going to create intent objects and use it in our app. In the previous app example, we detected the network change status of the phone and displayed it on the screen. In this example, we will add a button in the same app. On clicking on or tapping the button, the app will open the Wi-Fi settings. And the user can turn the Wi-Fi on or off from there. As the user performs any action, the app will display the network status change on the screen. For the network status, we used the `NetworkStatusReceiver.java` and `AndroidManifest.xml` files. Now, let's open the same project and change our `MainActivity.java` and `layout_main.xml` files to add a button and its functionality to them. Let's see these two files one by one:

The `activity_main.xml` file

This file is a visual layout of our main activity file. We will add a button view in this XML file. The code implementation of the file is as follows:

```
<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" >

    <Button
        android:id="@+id/btnWifiSettings"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Wi-Fi Settings" />

</RelativeLayout>
```


We have added a button in the layout with the view ID of `btnWifiSettings`. We will use this ID to get the button View in the layout file. We have already discussed the layouts in the previous chapters. Let's now see our main activity file that will use this layout as the visual content.

The MainActivity.java file

This file represents the main activity file as a launcher point of the app. We will implement our button's core functionality in this file. The code implementation of the file is as follows:

```
public class MainActivity extends Activity {

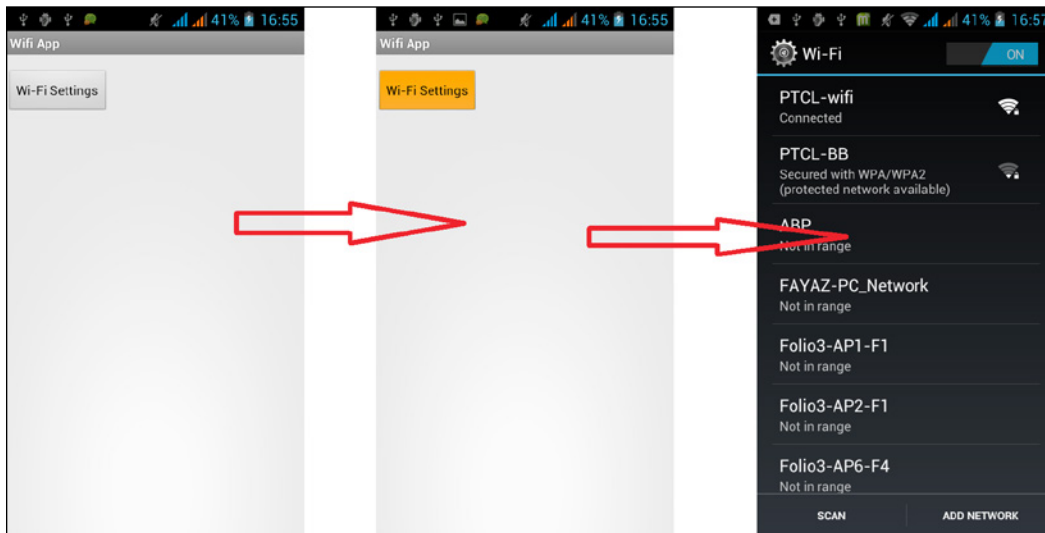
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button wifiSettings = (Button) findViewById(R.id.btnWifiSettings);
        wifiSettings.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                Intent in = new Intent(Settings.ACTION_WIFI_SETTINGS);
                startActivity(in);
            }
        });
    }
}
```

As discussed many times, we have extended our class from the Activity class, and we have overridden the `onCreate()` method of the class. After calling the super method, we have first referenced our layout file (explained in the previous section) using the `setContentView()` method and passed the layout ID as the parameter. After getting the layout file, we have extracted our Wi-Fi settings button from the layout by calling the `findViewById()` method. Remember, we set the button View's ID to `btnWifiSettings`; so, we will pass this ID in the method as an argument. We stored the referenced file of our button in a local `Button` object reference object. Now, we will set `View.OnClickListener` of the local button to perform our tasks on a button click. We have passed an anonymous object of `OnClickListener` in the `button.setOnClickListener()` method, and overridden the `onClick()` method of the anonymous object.

Until now, we have only performed some initial steps to create a setup for our app. Now, let's focus on opening the Wi-Fi settings task. We will create an `Intent` object, and we have to pass a constant string ID to tell the intent about what to start. We will use the `Settings.ACTION_WIFI_SETTINGS` constant that shows the settings to allow the configuration of the Wi-Fi. After creating the `Intent` object, we will pass it in the `startActivity()` method to open the activity containing the Wi-Fi settings. It is that simple with no rocket science at all. When we run the app, we will have something similar to the following screenshots:



Opening the Wi-Fi Settings app

As seen from the preceding screenshot, when we click or tap the Wi-Fi Settings button, it will open the Wi-Fi settings screen of the Android phone. On changing the settings, such as turning on the Wi-Fi, it will display the toasts to show the updated changes and network status.

We have finished discussing the communication components using intents, in which we used Bluetooth and Wi-Fi via intents and saw how these can be used in various examples and applications. Now, we will discuss how the media components can be used via intents and what we can do for media components in the following sections.

Media components

The preceding section was all about communication components. But the difference between old phones and the new smartphones is the media capability, such as high-definition audio-video features. And the multimedia capabilities of mobile phones have become a more significant consideration to many consumers. Fortunately, the Android system provides multimedia API's for many features such as playing and recording a wide range of image, audio, and video formats both locally and streamed. If we describe the media components in simple words, this topic can only be covered in a fully dedicated chapter, and it is beyond the scope of this book. We will only discuss those media components that can be triggered, used, and accessed through intents. The components to be discussed in this section include using intents to take pictures, using intents to record video, speech recognition using intents, and the role of intents in text-to-speech conversion. The first three topics use intents to perform the actions; but the last topic of text-to-speech conversion doesn't use intents on a complete basis. We will also develop a sample application to see the intents in action. Let's discuss these topics one by one in the following subsections.

Using intents to take pictures

Today, almost every phone has a digital camera component. The popularity of digital cameras embedded within mobile phones has caused their prices to drop along with their size. Android phones also include digital cameras varying from 3.2 megapixels to 32 megapixels. From the development perspective, pictures can be taken via many different methods. The Android system has also provided the APIs for camera control and pictures, but we will only be focusing on one method that uses intents in it. This is the easiest way to take pictures in Android development, and contains no more than few lines of code.

We will first create a layout with the image View and button. Then, in the `Activity` class, we will get the references of our views from the layout file, and set the click listener of the button. On clicking the button, we will create the intent of the capture image, and start another activity as a child class. After getting the result, we will display that captured image in our image View.

So, with the basic empty Hello World project ready, we will change three files and add our code to it. The files are `activity_main.xml`, `MainActivity.java`, and `AndroidManifest.xml`. Let's explain the changes in each file one by one:

The activity_main.xml file

This file represents the visual layout for the file. We will add an `ImageView` tag to show the captured image and a `Button` tag to take a picture and trigger the camera. The code implementation of the file is as follows:

```
<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" >

    <ImageView
        android:id="@+id/imageView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:src="@drawable/ic_launcher" />

    <Button
        android:id="@+id/btnTakePicture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/imageView1"
        android:layout_centerHorizontal="true"
        android:text="Take Picture" />

</RelativeLayout>
```

As you can see in the code, we have placed an `ImageView` tag in the relative layout with the ID of `imageView1`. This ID will be used in the main activity file to extract the view from the layout to use in the Java file. We have placed the view in the horizontal centre of the layout by assigning the value `true` in the `android:layout_centerHorizontal` tag. Initially, we have set a default image of our app's launcher icon to our image View. Below the image View, we have placed a button View. On tapping the button, the camera will be started. The button's ID is set to `btnTakePicture` by the `android:layout_below` tag below the image View layout. This relativity is the main advantage of the relative layouts as compared to linear layouts. So now, let's have a look at the activity of the app that performs the main functionality and uses this layout as a visual part as well.

The MainActivity.java file

This file represents the main launching activity of the app. This file uses the `layout_main.xml` file as the visual part, and it is extended from the `Activity` class. The code implementation of the file is as follows:

```
public class MainActivity extends Activity implements OnClickListener
{
    ImageView takenImage;
    Button imageButton;
    final int TAKE_IMAGE_CODE = 1;

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

        imageButton = (Button) findViewById(R.id.btnTakePicture);
        imageButton.setOnClickListener(this);
        takenImage = (ImageView) findViewById(R.id.imageView1);
    }

    @Override
    public void onClick(View v) {
        Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        startActivityForResult(intent, TAKE_IMAGE_CODE);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == TAKE_IMAGE_CODE)
        {
            if (resultCode == RESULT_OK)
            {
                Bitmap pic = (Bitmap) data.getExtras().get("data");
                takenImage.setImageBitmap(pic);
            }
        }
    }
}
```

We start our class by overriding the `onCreate()` method of the activity. We set the visual layout of the activity to the `activity_main.xml` layout by calling the `setContentView()` method. Now, as the layout is set, we can get references to the views in the layout file.

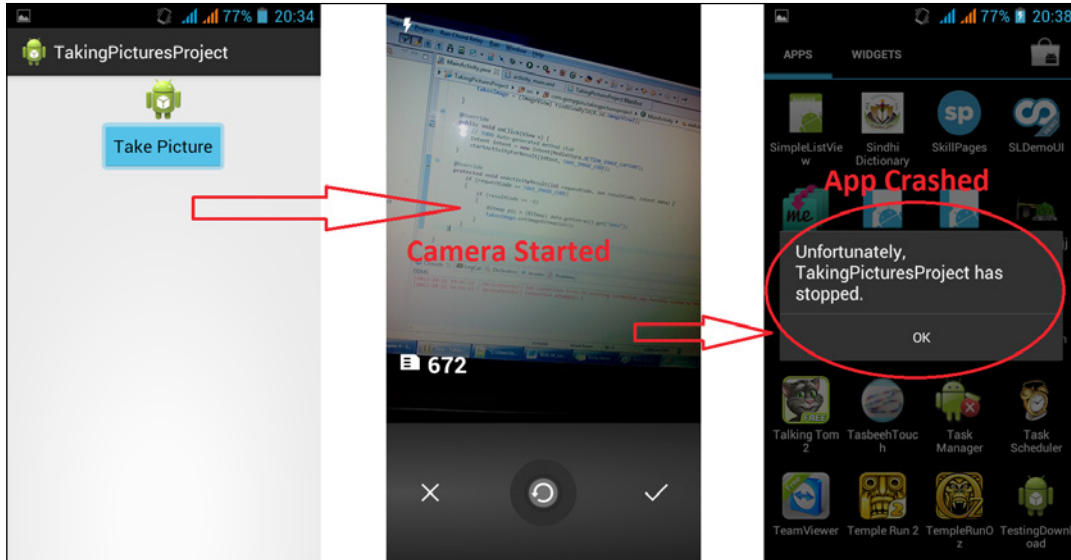
We create two fields in the class; `takenImage` of the `ImageView` class to be used to show the captured image and `imageButton` of the `Button` class to be used to trigger the camera by clicking on it. The `onClick()` method will be called when the button is tapped/clicked. So, we will define our camera-triggering code in this method. So, in this method, we are creating an instance of the `Intent` class, and we are passing the `MediaStore.ACTION_IMAGE_CAPTURE` constant in the constructor. This constant will tell Android that the intent is for the purpose of image capture, and Android will start the camera on starting this intent. If a user has installed more than one camera app, Android will present a list of all valid camera apps, and the user can choose any to take the image.

After creating an intent instance, we pass this intent object in the `startActivityForResult()` method. In our picture-capturing app, clicking on the button will start another activity of the camera. And when we close the camera activity, it will come back to the original activity of our app and give us some result of the captured picture. So, to get the result in any activity, we have to override the `onActivityResult()` method. This method is called when the parent activity is started after the child activity is completed. When this method is called, it means that we have used the camera and are now back to our parent activity. If the result is successful, we can display the captured image in the image View.

First, we can learn whether this method is called after the camera or if another action has happened. For this purpose, we have to compare the `requestCode` parameter of the method. Remember, when calling the `startActivityForResult()` method, we passed the `TAKE_IMAGE_CODE` constant as the other parameter. This is the request code to be compared to.

After that, to check the result, we can see the `resultCode` parameter of the method. As we used this code for the camera picture intent, we will compare our `resultCode` with the `RESULT_OK` constant. After the success of both conditions, we can conclude that we have received our image. So, we use the intent to get our image data by calling the `getExtras().get()` method. This will give us the `Object` type of data. We further typecast it to `Bitmap` to prepare it for `ImageView`.

Finally, we call the `setImageBitmap` method to set the new bitmap to our image View. If you run the code, you will see an icon image and a button. After clicking on the button, the camera will be started. When you take the picture, the app will crash and shut down. You can see it in the following screenshots:



The app crashed after taking a picture

You might be wondering why the crash occurred. We forgot to mention one thing; whenever any app uses the camera, we have to add the `uses-feature` tag in our manifest file to tell the app that it will use the camera feature. Let's see our Android manifest file to understand the `uses-feature` tag.

The AndroidManifest.xml file

This file defines all the settings and features to be used in our app. There is only one new thing that we haven't seen. The code implementation of the file is as follows:

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

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

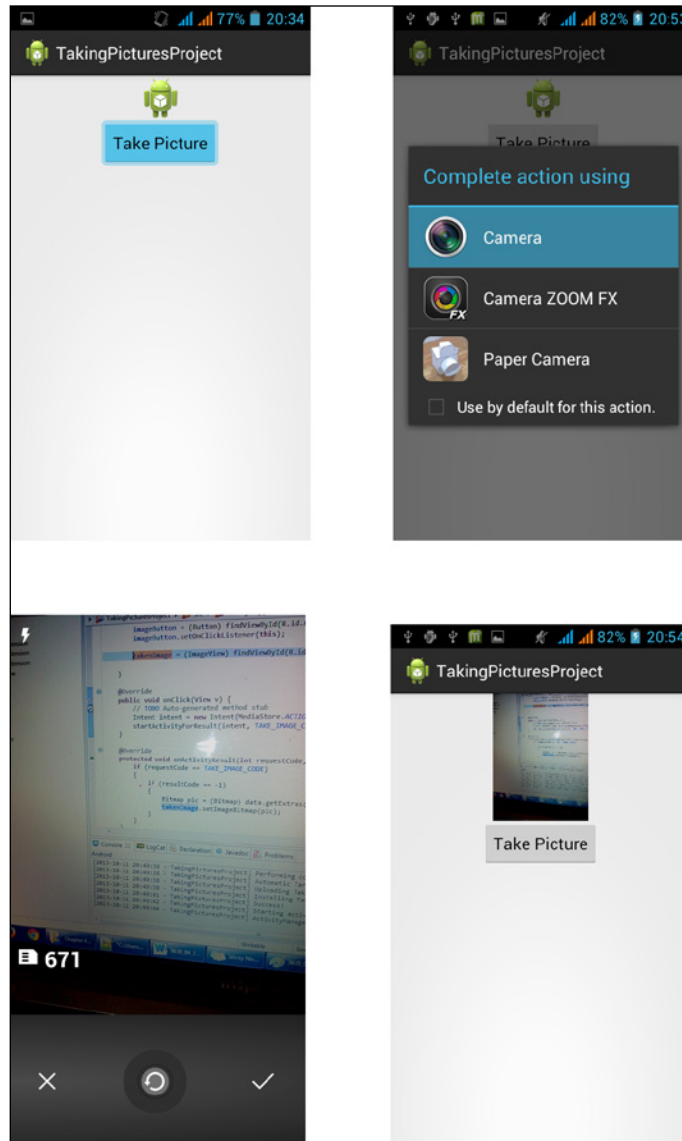
    <uses-feature android:name="android.hardware.camera" />

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

You can see that we have added the `uses-feature` tag, and we have assigned `android.hardware.camera` in the `android:name` property. This tag tells the app about the camera usage in it, and the Android OS gives our app the permission to use the external camera.

After adding this line in the manifest file and running the code, you will see something similar to the following screenshot if you have more than one camera in your phone:



Taking pictures through the intents app

In the screenshot, you can see that the user is asked to choose the camera, and when a picture is taken, the image is shown in the app.

When we summarized the code, we first created a layout with image View and button. Then, in the Activity class, we got the references of our views from the layout file, and set the click listener of the button. After clicking on the button, we created the intent of capture image, and started another activity as the child activity. After getting the result, we displayed that captured image in our image View. It was as easy as a walk in the park. In the next section, we will see how we can record video using intents.

Using intents to record video

Until now, we have already seen how to take pictures using intents. In this section, we will see how we can record video using intents. We will not discuss the whole project in this section. The procedure to record videos using intents is almost the same as taking pictures with few minor changes. We will only discuss those changes in this section. Now, let's see how the app works to record video.

The first change that we have done is in our layout file. We removed the image View section, and have placed the `videoView` tag. The following code implementation shows that tag:

```
<VideoView
    android:id="@+id/videoView1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
/>
```

You can see that everything is the same as it was in `ImageView`. Now, as we have changed image view to video view in our layout, we have to change that in our activity as well. Just as we did for `ImageView`, we will create a field object of `VideoView`, and get the reference in our `onCreate()` method of the activity. The following code sample shows the field object of `VideoView` line:

```
VideoView video;
video = (VideoView) findViewById(R.id.videoView1);
```

Everything is the same, and we have already discussed it. Now, in our `onClick()` method, we will see how we send the intent that triggers the video recording. The code implementation to be put on the `onClick()` method to send an intent is as follows:

```
Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1);
startActivityForResult(intent, TAKE_VIDEO_CODE);
```

You can see that we have created an intent object, and instead of passing `MediaStore.ACTION_IMAGE_CAPTURE`, we have passed `MediaStore.ACTION_VIDEO_CAPTURE` in the constructor of the intent. Also, we have put an extra object in the intent by calling the `putExtra()` method. We have put the extra object defining the video quality as high by assigning the `MediaStore.EXTRA_VIDEO_QUALITY` value to 1. Then, we pass the intent in the `startActivityForResult()` method again to start the camera activity.

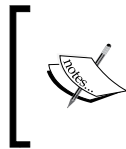
The next change is in the `onActivityResult()` method when we get the video from the intent. The following code shows some sample code to get the video and pass it in the `VideoView` tag and play it:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == TAKE_VIDEO_CODE)
    {
        if (resultCode == RESULT_OK)
        {
            Uri videoUri = data.getData();
            video.setVideoURI(videoUri);
            video.start();
        }
    }
}
```

In the case of taking a picture, we restored raw data from the intent, typecasted it to `Bitmap`, and then set our `ImageView` to `Bitmap`. But here, in case of recording a video, we are only getting the URI of the video. The `Uri` object declares the reference of data in the mobile phone. We get the URI of the video, and set it in our `VideoView` using the `setVideoURI()` method. Finally, we play the video by calling the `VideoView.start()` method.

From these sections, you can see how easy it is to use the intents to capture images or record videos. Through intents, we are using the already built-in camera or camera apps. If we want our own custom camera to capture images and videos, we have to use the Camera APIs of Android.

We can use the `MediaPlayer` class to play video, audio, and so on. The `MediaPlayer` class contains methods like `start()`, `stop()`, `seekTo()`, `isLooping()`, `setVolume()`, and much more. To record a video, we can use the `MediaRecorder` class. This class contains methods including `start()`, `stop()`, `release()`, `setAudioSource()`, `setVideoSource()`, `setOutputFormat()`, `setAudioEncoder()`, `setVideoEncoder()`, `setOutputFile()`, and much more.



When you are using `MediaRecorder` APIs in your app, don't forget to add the permissions of `android.permission.RECORD_AUDIO` and `android.permission.RECORD_VIDEO` in your manifest file.

To take pictures without using intents, we can use the `Camera` class. This class includes the methods `open()`, `release()`, `startPreview()`, `stopPreview()`, `takePicture()`, and much more.

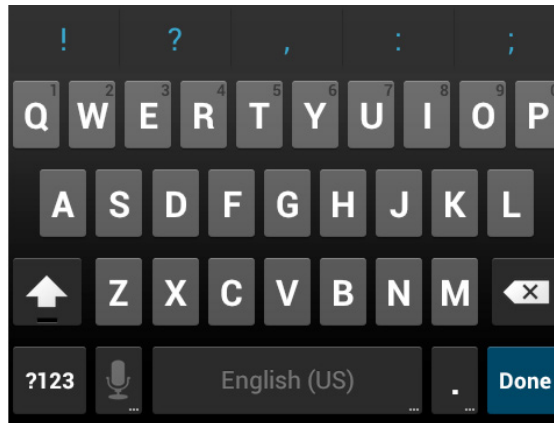


When you are using `Camera` APIs in your app, don't forget to add the permissions of `android.permission.CAMERA` in your manifest file.

Until now, we have used visual media components for videos and pictures using intents. In the next section, we will use audio components of a phone using intents. We will see how we can use the speech recognition and text-to-speech supports using intents in the next sections.

Speech recognition using intents

Smartphones introduced voice recognition that became a very big achievement for disabled people. Android introduced speech recognition in API Level 3 in Version 1.5. Android supports voice input and speech recognition using the `RecognizerIntent` class. Android's default keyboard contains a button with a microphone icon on it. This allows the user to speak instead of typing a text. It uses the speech-recognition API for this purpose. The following screenshot shows the keyboard with the microphone button on it:



Android's default keyboard with the microphone button

In this section, we will create a sample application that will have a button and text field. After clicking on the button, Android's standard voice-input dialog will be displayed, and the user will be asked to speak something. The app will try to recognize whatever the user speaks and type it in the text field. We will start by creating an empty project in the Android Studio or any other IDE, and we will modify two files in it. Let's start with our layout file in the next section.

The activity_main.xml file

This file represents the visual content of the app. We will add the text field and button view in this file. The code implementation of the file is as follows:

```
<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_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:ems="10"
        android:inputType="textMultiLine" >
    </EditText>

    <Button
        android:id="@+id/btnRecognize"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText1"
        android:layout_centerHorizontal="true"
        android:text="Recognize" />

</RelativeLayout>
```

As you can see, we have placed an `EditText` field. We have set `android:inputType` to `textMultiLine` to type the text in multiple lines. Below the text field, we have added a `Button` view with an ID of `btnRecognize`. This button will be used to start the speech-recognition activity when it is tapped or clicked on. Now, let's discuss the main activity file.

The MainActivity.java file

This file represents the main activity of the project. The code implementation of the file is as follows:

```
public class RecognizeText extends Activity implements OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button recognizeButton = (Button) findViewById(R.id.btnRecognize);
        recognizeButton.setOnClickListener(this);
    }


    @Override
    public void onClick(View v) {
        Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL, RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "Hey, are you speaking!!!");
        intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 1);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, Locale.ENGLISH);
        startActivityForResult(intent, 1);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == 1 && resultCode == RESULT_OK)
        {
            ArrayList<String> results;
            results = data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
            EditText speechText = (EditText) findViewById(R.id.editText1);
            speechText.setText(results.toString());
        }
    }
}
```


As usual, we override the `onCreate()` method, and get our button reference from the layout set by the `setContentView()` method. We set the button's listener to this class, and in the activity, we implement `OnClickListener` along with overriding the `onClick()` method. In the `onClick()` method, we create an intent object and pass `RecognizerIntent.ACTION_RECOGNIZE_SPEECH` as an action string in the constructor. This constant will tell Android that the intent is for speech-recognition purpose. Then, we have to add some `extra` objects to provide more information to Android about the intent and speech recognition. The most important `extra` object to be added is `RecognizerIntent.EXTRA_LANGUAGE_MODEL`. This informs the recognizer about which speech model to use when recognizing speech. The recognizer uses this `extra` to fine-tune the results with more accuracy. This `extra` method is required and must be provided when calling the speech-recognition intent. We have passed the `RecognizerIntent.LANGUAGE_MODEL_FREE_FORM` model for speech. This is a language model based on a free-form speech recognition. Now, we have some optional `extra` objects that help the recognizer with more accurate results. We have added `extra` of `RecognizerIntent.EXTRA_PROMPT` and passed some string value in it. This will notify the user that speech recognition has been started.

Next, we add the `RecognizerIntent.EXTRA_MAX_RESULTS` extra and set its value as 1. Speech recognition's accuracy always varies. So, the recognizer will try to recognize with more accuracy. So, it creates different results with different accuracies and maybe different meanings. So, through this extra, we can tell the recognizer about how many results we are interested in. In our app, we have put it as 1. So, that means the recognizer will provide us with only one result. There is no guarantee that this result will be accurate enough; that's why it is recommended to pass a value greater than 1. For a simple case, you can pass a value upto 5. Remember, the greater the value you pass, the more time will it take to recognize it.

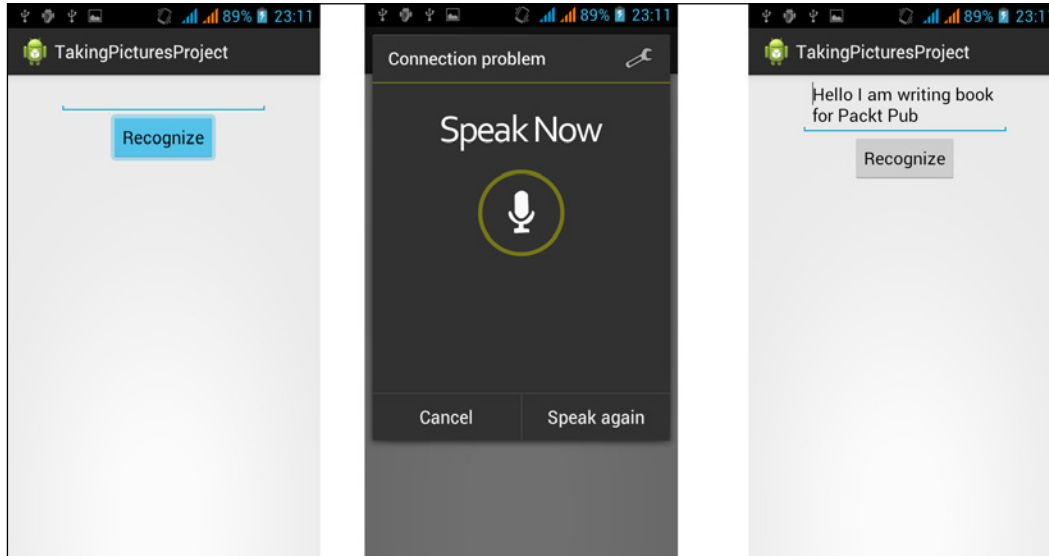
Finally, we put our last optional extra of language. We pass `Locale.ENGLISH` as the value of the `RecognizerIntent.EXTRA_LANGUAGE` extra. This will tell the recognizer about the language of the speech. So, the recognizer didn't have to detect the language, this results in more accuracy in speech recognition.

 The speech-recognition engine may not be able to understand all the languages available in the `Locale` class. Also, it is not necessary that all the devices will support speech recognition.

After adding all the extra objects, we have ensured that our intent object is ready. We pass it in the `startActivityForResult()` method with `requestCode` as 1. When this method is called, a standard voice-recognition dialog is shown with the prompt message that we had given. After we finish speaking, our parent activity's `onActivityResult()` method is called. We first check whether `requestCode` is 1 or not so that we can be sure that this is our result of speech recognition. After that, we will check `resultCode` to see whether the result was okay or not. After successful results, we will get an array list of strings containing all the words recognized by the recognizer. We can get these words' lists by calling the `getStringArrayListExtra()` method and passing `RecognizerIntent.EXTRA_RESULTS`. This list is only returned when `resultCode` is okay; otherwise, we will get a null value. After wrapping up the speech-recognition stuff, we can now set the text value to the result. For that, we first extract the `EditText` view from the layout, and set our result to the value of the text field by calling the `setText()` method.

 An active Internet connection is required to run speech recognition. The speech-recognition process is executed on the servers of Google. An Android phone takes the voice input, sends it to Google Servers, and it is processed there for recognition. After recognition, Google sends the results back to the Android phone, the phone informs the user about the results, and the cycle is complete.

If you run the project, you will see something similar to the following screenshots:



Speech recognition using intents

In the image, you can see that after clicking on the **Recognize** button, a standard voice-input dialog is shown. On speaking something, we will return back to our parent activity, and after recognizing the speech, it will print all the text in the text field.

Role of intents in text-to-speech conversion

In the previous section, we discussed how the Android system can recognize our speech and perform actions such as controlling the mobile phone via speech commands. We also developed a simple speech-to-text example using intents. This section is the opposite of the previous section. In this section, we will discuss how the Android system can convert our text into a beautiful voice narration. We can call it text-to-speech conversion. Android introduced the **Text-To-Speech (TTS) Conversion** engine in Version 1.6 API Level 4. We can use this API to produce speech from within our application, thus allowing our app to talk with our users. And if we add speech recognition, it will be like talking with our application. Text-to-speech conversion requires preinstalled language packs, and due to the lack of storage space on mobile phones, it is not necessary that the phone will have any language packs already installed in it. So, while creating any app using the text-to-speech engine, it is a good practice to check whether the language packs are installed or not.

We can't use text-to-speech conversion using intents. We can only use it through the text-to-speech engine called TTS. But, there is a minor role of intents in text-to-speech conversion. Intents are used only to check whether the language packs are preinstalled or not. So, creating any app that uses text-to-speech will first have to use intents to check the language packs' installation status. That's the role of intents in text-to-speech conversion. Let's look at the sample code of checking the language packs' installation state:

```
final int VAL_TTS_DATA = 1;
Intent intent = new Intent (Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(intent, VAL_TTS_DATA);
```

The first thing we will do for text-to-speech conversion is to check the language packs. In the code, we can see that we are creating an intent object. And we are passing the `Engine.ACTION_CHECK_TTS_DATA` constant that will tell the system that the intent will check the text-to-speech (TTS) data and language packs. We are then passing the intent in the `startActivityForResult()` method along with the `VAL_TTS_DATA` constant value used as `requestCode`. Now, if the language packs are installed and everything is okay, we will get `resultCode` as `RESULT_OK` in the `onActivityResult()` method. So, if the result is okay, we can use text-to-speech conversion. So, let's see the code sample for the `onActivityResult()` method, as shown in the following screenshot:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (requestCode == VAL_TTS_DATA)
    {
        if (resultCode == Engine.CHECK_VOICE_DATA_PASS)
        {
            // Let's Do Our Text To Speech Conversion Here
        }
        else
        {
            // It means there is no language pack installed.
            // Let's install language pack here
        }
    }
}
```

So, we first check `requestCode` of our passed code. Then, we check `resultCode` to `Engine.CHECK_VOICE_DATA_PASS`. This constant is used to tell whether voice data is available or not. If we have data available in our phone, we can do our text-to-speech conversion there. Otherwise, it is clear that we have to install voice data first before doing the text-to-speech conversion. You will be pleased to know that installing voice data is also very easy; it uses intents for this purpose. The following code snippet shows how to install voice data using intents:

```
Intent installLanguage = new Intent (Engine.ACTION_INSTALL_TTS_DATA);
startActivity(installLanguage);
```

We created an intent object and passed `Engine.ACTION_INSTALL_TTS_DATA` in the constructor. This constant will tell Android that the intent is for the installation of text-to-speech language packs' data. And then, we pass the intent into the `startActivity()` method to start installation. After the language pack's installation, we have to create an object of the `TextToSpeech` class and call its `speak()` method when we want to do some text-to-speech conversion. The following is the code implementation showing how to use the object of the `TextToSpeech` class in the `onActivityResult()` method:

```
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {

    if (requestCode == VAL_TTS_DATA) {
        if (resultCode == Engine.CHECK_VOICE_DATA_PASS) {
            TextToSpeech tts = new TextToSpeech(this,
                new OnInitListener() {
                    public void OnInit(int status) {
                        if (status == TextToSpeech.SUCCESS) {
                            tts.setLanguage(Locale.US);
                            tts.setSpeechRate(1.1f);
                            tts.speak("Hello, I am writing book for Packt",
                                TextToSpeech.QUEUE_ADD, null);
                        }
                    }
                });
        }
        else {
            Intent installLanguage = new Intent (
                Engine.ACTION_INSTALL_TTS_DATA);
            startActivity(installLanguage);
        }
    }
}
```

As seen in the code, after the successful installation of the language data packs, we have created an instance of `TextToSpeech` and passed an anonymous `OnInitListener` object. We have implemented the `onInit()` method. This method will set the initial settings of the `TextToSpeech` object. If the status is a success, we are setting the language, speech rate, and finally, we are calling the `speak()` method. In this method, we passed a string of characters, and Android will read these letters aloud.

Concluding the whole topic, the role of intents in text-to-speech conversion is of checking and installing voice-data packs. Intents don't contribute directly to text-to-speech conversion, but they just set the initial setup for text-to-speech conversion.

With text-to-speech conversion, we have finished the discussions on media components. In media components, we discussed taking pictures, recording videos, speech recognition, and text-to-speech conversion. In the next section, we will discuss motion components and see how intents play a role in these components.

Motion components

Motion components in an Android phone include many different types of sensors that perform many different tasks and actions. In this section, we will discuss motion and position sensors such as accelerometer, geomagnetic sensor, orientation sensor, and proximity sensor. All these sensors play a role in the motion and position of the Android phone. We will discuss only those sensors that use intents to get triggered. We have only one such sensor that uses intents and that is the proximity sensor. Let's discuss it in the following section.

Intents and proximity alerts

Before learning about the role of intents in proximity alerts, we will discuss what proximity alerts are and how these can be useful in various applications.

What are proximity alerts?

The proximity sensor lets the user determine how close the device is to an object. It's often useful when your application reacts when a phone's screen moves towards or away from any specific object. For example, when we get any incoming call on an Android phone, placing the phone on the ear shuts off the screen and holding it back in the hands switches the screen on automatically. This application is using proximity alerts to detect the distance between the ear and the proximity sensor of the device. The following figure shows it in the visual format:



Another example can be when our phone has been idle for a while and its screen is switched off, it will vibrate if we have some missed calls or give notifications hinting us to check our phone. This can also be done using proximity sensors.

Proximity sensors use proximity alerts that detect the distance between the sensor of the phone and any object. These alerts let your application set triggers that are fired when a user is moved within or beyond a set distance from a geographic location. We will not discuss all the details for the use of proximity alerts in this section, but we will only cover some basic information and the role of intents in using proximity alerts. For example, we set a proximity alert for a given coverage area. We select a point in the form of longitude and latitude, a radius around that point in meters, and some expiry time for the alert. Now, after using proximity alerts, an alert will fire if the device crosses that boundary. It can be either that the device moves from outside to within the radius or it moves from inside the radius to beyond it.

Role of intents in proximity alerts

When proximity alerts are triggered, they fire intents. We will use a `PendingIntent` object to specify the intent to be fired. Let's see some code sample of the application of the distance that we discussed in the earlier section in the following implementation:

```
static String DISTANCE_PROXIMITY_ALERT = "distance.proximity.alert";
String locationService = Context.LOCATION_SERVICE;
LocationManager locationManager;
locationManager = (LocationManager) getSystemService(locationService);
double latitude = 65.12345;
double longitude = 0.43215;
float radius = 10f;
long expire = -1;

Intent intent = new Intent(DISTANCE_PROXIMITY_ALERT);
PendingIntent pIntent = PendingIntent.getBroadcast(this, -1, intent, 0);
locationManager.addProximityAlert(latitude, longitude, radius, expire, pIntent);
```

In the preceding code, we are implementing the very first step to use proximity alerts in our app. First of all, we create a proximity alert that can be done through `PendingIntent`. We define the name of the alert as `DISTANCE_PROXIMITY_ALERT`, and then get the location manager service by calling the `getSystemService()` method of the current activity we have written the code in. We then set some random values for latitude, longitude, radius, and expiration to infinity. It should be remembered that these values can be set to any depending on the type of application you are creating.

Now comes our most important part of creating the proximity alert. We create the intent, and we pass our own alert name in the constructor to create our own intent. Then, we create an object of `PendingIntent` by getting a broadcast intent using the `getBroadcast()` method. Finally, we are adding the proximity alert in our location manager service by calling the `addProximityAlert()` method.

This code snippet has only created the alert and set initial values for it. Now, assume that we have completely finished our distance app. So, whenever our device passes the boundary that we specified in the app or gets inside it, `LocationManager` will detect that we have crossed the boundary, and it will fire an intent having an extra value of `LocationManager.KEY_PROXIMITY_ENTERING`. This value is a Boolean value. If its value is `true` it means we have entered into the boundary, and if it is `false`, we have left the boundary. To receive this intent, we will create a broadcast receiver and perform the action. The following code snippet shows the sample implementation of the receiver:

```
public class ProximityAlertReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Boolean isEntered = intent.getBooleanExtra(
            LocationManager.KEY_PROXIMITY_ENTERING, false);
        if (isEntered)
            Toast.makeText(context, "Device has Entered!",
                Toast.LENGTH_SHORT).show();
        else
            Toast.makeText(context, "Device has Left!",
                Toast.LENGTH_SHORT).show();
    }
}
```

In the code, you can see that we are getting the extra value of `LocationManager.KEY_PROXIMITY_ENTERING` using the `getBooleanExtra()` method. We compare the value and display the toast accordingly. It was quite easy as you can see. But, like all the receivers, this receiver will not work until it is registered in `AndroidManifest.xml` or via code in Java. The java code for registering the receiver is as follows:

```
IntentFilter filter = new IntentFilter(DISTANCE_PROXIMITY_ALERT);
registerReceiver(new ProximityAlertReceiver(), filter);
```

There is nothing to explain here except that we are calling the `registerReceiver()` method of the Activity class. We will discuss `IntentFilter` in more detail in the following chapters.

In a nutshell, intents play a minor role in getting proximity alerts. Intents are only used to tell the Android OS about the type of proximity alert that has been added, when it is fired, and what information should be included in it so that the developers can use it in their apps.

Summary

In this chapter, we discussed the common mobile components found in almost all Android phones. These components include the Wi-Fi component, Bluetooth, Cellular, Global Positioning System, geomagnetic field, motion sensors, position sensors, and environmental sensors. Then, we discussed the role of intents with these components. To explain that role in more detail, we used intents for Bluetooth communication, turning Bluetooth on/off, making a device discoverable, turning Wi-Fi turn on/off, and opening Wi-Fi settings. We also saw how we can take pictures, record videos, do speech recognition, and text-to-speech conversion via intents. In the end, we saw how we can use the proximity sensor through intents.

In the next chapter, we will see how intents can be used in transferring data between activities, services, and other mobile components.

5

Data Transfer Using Intents

Until now, we have learned the classification of intents, their uses in Android Components, and a step-by-step guide to implement them in your Android application. This is the right time to look at the most important part of an Android application. It is the necessity of an Android application to transfer data from one activity to another (whether implicit or explicit). The secure transfer and retrieval of data is the prime focus of this chapter.

This chapter includes the following topics:

- The need to transfer data
- Data transfer between activities - an *INTENTed* way
- Data transfer in explicit intents
- Methods of explicit data transferring using intents
- Data transfer in implicit intents

Finding the need to transfer data

Technically, an Android application is a combination of different activities. These activities consist of layouts, views, and some content. These contents are mostly not dynamic nor are they predecided. For example, if an Android layout consists of a button, the text in that button can be static or predefined. Similarly, if there is any text field or any List View present in an activity, it mostly consists of dynamic data that comes from any server or any other means.

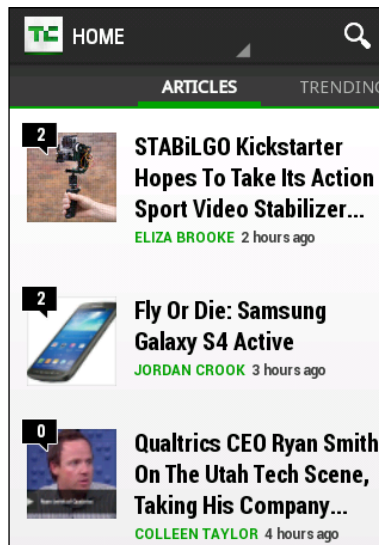
In these kinds of situations, we need some dynamic data that our application can fetch from the server (or somewhere else), and activities to transfer it between one another. This is the scenario in which the transfer of data takes place. Furthermore, the transfer of data is highly probable, where one activity performs some manipulation on the data and the other activity needs to show it in its Views.

Taking a simple example

In order to have a better understanding of the picture, let's take a theoretical example of why we need to perform data transfer between activities. The Reader application can be a good example to understand the reasons for data transfer.

The Reader application is an application in which there are different kinds of news present in a List View, and tapping them leads to the description page where the whole news is displayed with images and other texts. Let's have a step-by-step look at the flow of this application (taking the TechCrunch Android app as an example). The application will start with a splash screen, describing to the reader or the developer who made the app.

The following screenshot is the splash screen; the application will search for an Internet connection in order to display the feeds to the app's screen. Once the data is locally fetched, it parses it and places it inside the List View. Please note that the following screenshot of the List View is basically the custom List View that is not directly obtained by a built-in layout of Android. We need to make a custom layout for this and then populate it in the normal List View. For this, adapters are used (refer to the Internet in order to find how the basic List Views are created in Android).

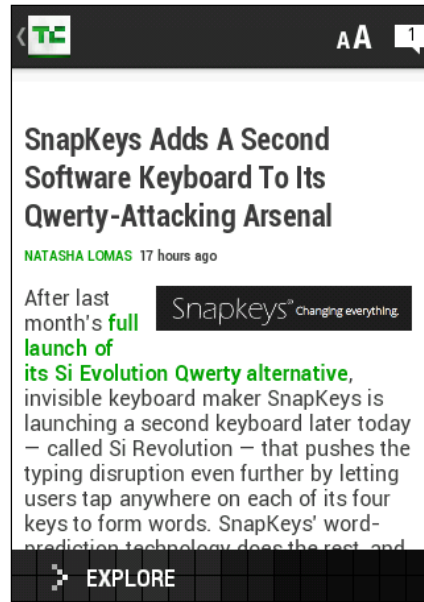


Activity of the Reader app in which the news are listed in the List View.

Now, there are two possibilities of data transfer that are described as follows:

- Whole data including the description is fetched

- Once the feed on the List View is clicked on, the description of that feed is fetched at that particular moment



Activity showing the description of the story as it was tapped in the preceding Activity

No matter what the case is, this step will require the data to be transferred from the first activity to the next activity. If the scenario is the first one, the description data that was parsed by the first activity will be delivered to the second activity in order to populate it in the View. Otherwise, in case of the second scenario, it will pass some URL to the second activity from where it can fetch the description of the news. Refer to the preceding screenshot.

Data transfer between activities – an INTENTed way

When we talk about data transfer between activities, we need to keep in mind that the only way to interact and manage the flow of the activities is through intents. In the previous chapter, we had a thorough discussion on how to move from one activity to the other using intents. Here, we will see how we can transfer data along with those intents and how to securely catch the transferred data in the destination activity.

Data transfer in explicit intents

You can begin to understand data transfer in intents by noticing its use in explicit intents. Recalling the definition of explicit intents, they are intents that direct towards another activity (within that application or another application). Explicit intents are usually directed to activities within the same application, but based on the application requirement, they can also be directed to activities belonging to other applications (for example, a device camera).

Methods of data transfer between activities

In this section, we will get started with the various data transfer techniques that are used in an Android application. The techniques have their own pros and cons. There are a total of three methods to transfer data explicitly from one activity to another. We will see them shortly along with their examples. The three methods are as follows:

- Data transfer using `putExtras()`
- Data transfer using `Parcelable` (only applicable to custom data objects)
- Data transfer using `Serializable` (only applicable to custom data objects)

Data transfer using `putExtras()`

In Android, the simplest way to transfer data from one activity to another is by sending it through extras. The intent extras support primitive data types to send the data. This means that you can send the data in the form of different data types such as `String`, `Boolean`, `Integer`, or `Float`.

Theoretically explaining, intent extras can be found inside the `Intent` class in the Android API. Developers need to make an object of the `Intent` class. This would be the same object that will be used in order to navigate through the activity. With this object, there will be multiple polymorphs of the `putExtras()` function. These polymorphs take different data types (as described earlier) as arguments and load the intent object with that data. With this, the object is finalized. Now, calling the `startActivity()` method from the `Activity` class starts the execution of the intent.

That was one side of the picture. This intent takes the flow of application towards the second activity; it's an activity of the same application in the case of an explicit calling, or it can be some other application in the case of an implicit intent. This new activity will receive the intent object and extract the data from it. From this, there is another method referred to as `getExtras()` that is present in the `Intent` class. As a result, it will give all the extras that were added by the source activity in the intent object, and using this, we can easily extract the desired data present in the extras of the intent.

This theoretical explanation may not make you understand each and everything of the data transfer using intent. We will learn more about data transfer using intents through examples in the next section, in which a step-by-step explanation of the data transfer will be given.

Implementation of `putExtras()`

In this section, we will study a step-by-step implementation of how to transfer data from one activity to another with the help of extras. As you may have previously read, this method is the simplest of all when considering data transferring between activities. In order to understand the working and implementation of this method, you must understand the activity life cycle, handling of different activities, and the implementation of intents in order to navigate between activities as prerequisites.

In order to begin with the first example, the first step is to make an Android project. The steps for making a project in the Android Studio are described in the previous chapters; you may refer to them if you want. You will end up making a project with various numbers of files and folders (as it comes by default with the Android project).

Implement a readymade tutorial

For the sake of simplicity, we are using the name `Activity1` for the source activity and `Activity2` for the destination activity. Now, follow the given steps in order to successfully implement the example.

1. First, create a new Android project or choose any existing project in which you want to implement the data transfer with intents. Implement the following code inside your newly created project in their respective classes:

```
//-----  
//Activity1 Class  
  
public class Activity1 extends Activity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {
```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.main_first);

final EditText editTextFieldOne = (EditText) findViewById(
    R.id.editttext1);
final EditText editTextFieldTwo = (EditText) findViewById(
    R.id.editttext2);
final EditText editTextFieldThree = (EditText) findViewById(
    R.id.editttext3);

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

String valueOne = editTextFieldOne.getText().toString();
String valueTwo = editTextFieldTwo.getText().toString();
String valueThree = editTextFieldThree.getText().toString();

transferButton.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        // TODO Auto-generated method stub

        Intent intent = new Intent(Activity1.this,
            Activity2.class);
        intent.putExtra("EDITTEXT_ONE_VALUE", valueOne);
        intent.putExtra("EDITTEXT_TWO_VALUE", valueTwo);
        intent.putExtra("EDITTEXT_THREE_VALUE", valueThree);
        Activity1.this.startActivity(intent);

    }
});

}
}

//-----
//Activity2.java

public class Activity2 extends Activity {

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

        Intent intent = getIntent();

        String valueOne = intent.getExtras().getStringKey(
            "EDITTEXT_ONE_VALUE");
```

```

String valueTwo = intent.getExtras().getStringKey(
    "EDITTEXT_TWO_VALUE");
String valueThree = intent.getExtras().getStringKey(
    "EDITTEXT_THREE_VALUE");

TextView textViewOne = (TextView) findViewById(
    R.id.textView1);
TextView textViewTwo = (TextView) findViewById(
    R.id.textView2);
TextView textViewThree = (TextView) findViewById(
    R.id.textView3);

textViewOne.setText(valueOne);
textViewTwo.setText(valueTwo);
textViewThree.setText(valueThree);
    }
}

//-----
//main_first.xml File

<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=".Activity1" >

    <EditText
        android:id="@+id/edittext1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/textView1"
        android:layout_alignBottom="@+id/textView1"
        android:layout_alignParentRight="true"
        android:layout_toRightOf="@+id/textView1"
        android:ems="10"
        android:inputType="textPersonName" >

        <requestFocus />
    </EditText>

    <EditText
        android:id="@+id/edittext2"

```



```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/textView2"
        android:layout_alignBottom="@+id/textView2"
        android:layout_alignLeft="@+id/edittext_enter_name"
        android:layout_alignRight="@+id/edittext_enter_name"
        android:ems="10" />

<EditText
    android:id="@+id/edittext3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/edittext_enter_surname"
    android:layout_alignRight="@+id/edittext_enter_surname"
    android:layout_alignTop="@+id/textView3"
    android:ems="10" />

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@+id/edittext_enter_address"
    android:layout_below="@+id/textView3"
    android:layout_marginRight="10dp"
    android:layout_marginTop="33dp"
    android:text="@string/enter_button_text" />

</RelativeLayout>

//-----
//main_second.xml File

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

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/null_string"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="@string/null_string"
        android:textAppearance="?android:attr/textAppearanceMedium" />

<TextView
    android:id="@+id/textView3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/null_string"
    android:textAppearance="?android:attr/textAppearanceMedium" />

</LinearLayout>

//-----
//AndroidManifest.xml File

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

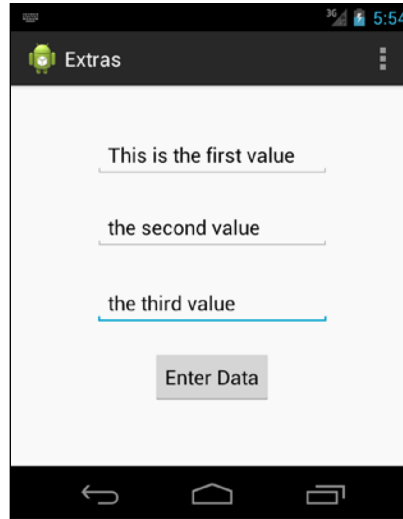
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.app.application.Activity1"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="
                    android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="com.app.application.Activity2"
            android:label="@string/app_name" >
        </activity>
    </application>

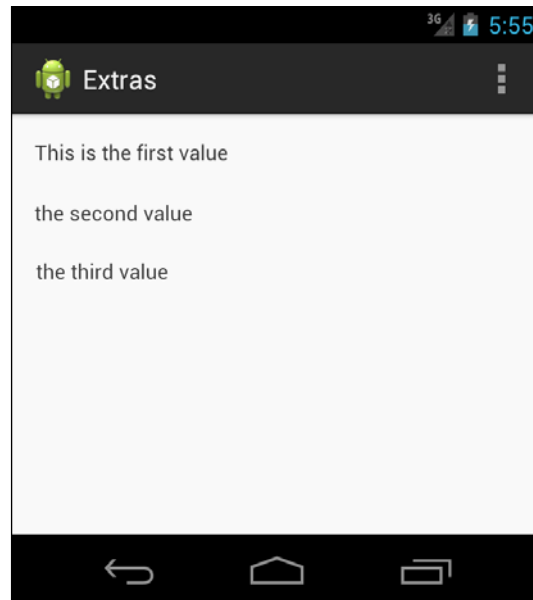
</manifest>
```

2. Run the project and the following screenshot will appear on the screen:



The Activity1.java layout for taking input from the user for PutExtra()

3. Fill the `EditText` fields and tap the button to transfer the data. The `Activity2` screen will appear with the form data that was entered in the `Activity1` screen:



The view of the Activity2.java file, which shows that the data is successfully caught and showed.

Understanding the code

The previous subsection, *Implement a readymade tutorial*, consists of five parts that we will see in detail in the following sections. Like every example presented in this book, we have described this example with respect to a new project in order to make it flexible and easy to understand. You can easily put this example inside your own application; it will not take any extra effort to do it once you have a proper understanding of the intent extras.

The Activity1.java class

Getting started, this is the source activity that will initiate the intent in order to navigate to the next activity.

This is a simple activity that is built when the new Android project is created. Recalling the basics, it will have an `onCreate()` method that will be executed in the first place when the activity is created by Android. Once the activity is created, the layout that is defined in the `main_first.xml` file in the `Layout` folder will be rendered on the screen.

Now, it is time to get the objects of all the `EditText` fields that are placed in the layout file. For this, we will add the following lines in the code which will find the View by ID and return the object:

```
EditText editTextFieldOne = (EditText) findViewById(  
    R.id.edittext1);
```




It is good to use meaningful names for your objects. Since this book is meant for beginners who don't normally work on huge applications, the object names are given to make the code as simple as possible.

The `findViewById()` method belongs to the `Activity` class, whose purpose is to find the particular View that can be the child of any layout and return the object. Similarly, we will get the other two `EditText` objects by writing the following lines:

```
EditText editTextFieldTwo = (EditText) findViewById(  
    R.id.edittext2);  
EditText editTextFieldThree = (EditText) findViewById(  
    R.id.edittext3);
```

At this moment, we have the objects of all the input fields that are present in the `Activity1.java` class. The next step is to implement the functionality of the button that will take the input from these fields, add them into the object of intent, and send it through.

 The return type of the `findViewById()` method is an object of the `View` class. So, while using `findViewById()`, we need to cast the returning object into a particular class type. For understanding this, you can see that the `View` is being casted to `EditText` in the preceding code.

Now, the next step is to implement the `onClickListener()` method on the button. For this, the first step is to get the object of the button using a method similar to the one used in the input fields.

```
Button transferButton = (Button) findViewById(R.id.button);
```

Once we get the button object, we will implement the `setOnClickListener()` method with an argument, `onClickListener()` and its implementation:

```
transferButton.setOnClickListener(new OnClickListener() {});
```

As you can see in the preceding line of code, we have attached an `OnClickListener()` object with `transferButton` along with its own `setOnClickListener` method. Keep in mind that it is still a raw method. It is now time to override the `onClick()` method.

In the preceding code, you can see that the definition of the `onClick()` method is given, and inside this method, we will get the data from the `EditText` fields and put it in the intent extras. As described in the code, the data is fetched from the `EditText` field by calling these lines on every `EditText` object:

```
String valueOne = editTextFieldOne.getText().toString();
```

This will get the current value present in the input field. We get the values of all the `EditText` fields and store them in `valueOne`, `valueTwo`, and `valueThree` consecutively.

Now, as we have the data that is to be put inside the intent object, we make an object of the intent using the previously described method. We set the source and destination activities (that is, `Activity1.java` as the source and `Activity2.java` as the destination). The next step is to pass the values inside the code. The `Intent.putExtra(String name, String data)` method is the most suitable one to put the string value in extras.

The arguments of `putExtra()` is somewhat like the key-value pairs. The first argument, `name`, is basically the key by which it will be identified once it reaches the destination activity. The other one is simply the value that is to be transferred in the extra associated with that key. So, by following line, we put the string inside an intent object with a key:

```
intent.putExtra("EDITTEXT_ONE_VALUE", valueOne);
```

Now that the value of the first `EditText` field is placed inside the intent object with the key of `EDITTEXT_ONE_VALUE`, we repeat this information for the other two values. Once the values are loaded in the intent object, we call the `startActivity()` method to execute the intent.

The Activity2.java class

This is the destination class in which the incoming intent will be handled. This class contains a simple layout file with three `TextView` Views in order to show the values coming from the previous activity. In the `onCreate()` method the intent is received by the `getIntent()` method. This method belongs to the `Activity` class and is used to get the intent which will be navigating to it.

There is a method inside the `Intent` class which is used to get all extras that are coming with that particular intent object. The following method is used to identify a particular set of data coming with the described key:

```
String valueOne = intent.getExtras().getStringKey(
    "EDITTEXT_ONE_VALUE");
```

The value associated with the key, `EDITTEXT_ONE_VALUE`, will be extracted from the intent object, and saved into the `valueOne` string. Similarly, all the data will be taken out of the intent object and saved in this destination class.

Once the data is saved in the variables, it is time to get the objects of the `TextView` Views and set these values to it. As previously explained, the `TextView` Views are obtained using the `findViewById()` method.

```
textViewOne.setText(valueOne);
```

The `setText()` method is used to set the text in `TextView`, and hence, it is saved by the value that is coming from the first activity. This is how the destination activity will get the data from the source activity using the `putExtras()` feature.

The main_first.xml file

The main_first.xml file is a simple XML file that contains three `EditText` fields used by the activity to take input from. Furthermore, it also has a button that is used to trigger the event in order to navigate to the next activity. The IDs for these Views are given as `edittext1`, `edittext2`, `edittext3`, and `button1` respectively.



You can make your desired layout file by dragging and dropping it in the GUI. The XML code of the layout file is simple, and explained in the previous chapters as well. But, keep in mind that *drag-and-drop is not recommended* especially for the new Android developers; so, the best way to implement it is via an XML file.

The main_second.xml file

This is the layout file for the second activity that is actually the destination and the data-receiving activity. The layout consists of the three `TextView` Views that are used to show `valueOne`, `valueTwo`, and `valueThree` as sent from `Activity1`, that is, the source activity.

The AndroidManifest.xml file

The `AndroidManifest.xml` file is the fundamental part of an Android application. It keeps track of the whole structure of the application. It contains all the activities, receivers, permissions, version-related issues, minimum and maximum SDK, and many other things. As we have two activities in our project, `Activity1` and `Activity2`, the `AndroidManifest.xml` file has these activities as also different things such as the application version name and the version code in the XML tags.



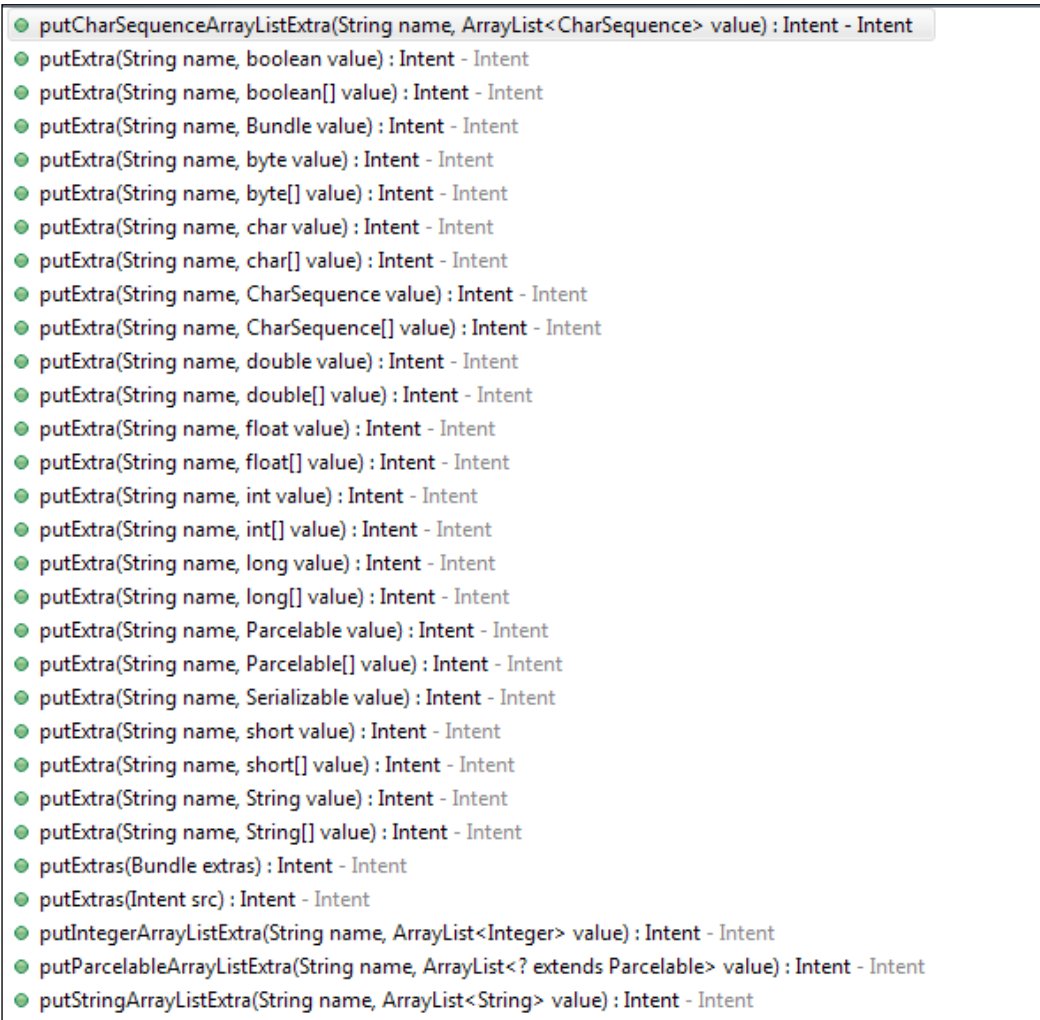
No special permission is required to send the data from one activity to another, but in case of data writing and reading on the SD Card or internal memory, we do need certain permissions to fulfill the task.

Future considerations

Sending data through parcels is one of the basic techniques that is used by Android intents. It has many more improvements and efficient enhancements that we will further study in the following methods of data transferring. We should also keep in mind that this method is restricted to certain limited data types (given in the next section). In order to transfer the custom object from one activity to another, we need to use the next method of data transferring.

Extras supported data types

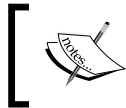
Intent's `putExtra()` method supports various data types that can be transferred to the destination activity. In the previous example, we used only one data type (`String`), but along with that, we can add various other data types. The methods are self-explanatory apart from `putParcelable()` and `putSerializable()`, which are the next major topics of this chapter. Take a look at the following screenshot showing the various data types:



Different data types that can be added inside the `putExtras()` intent

The concept of Android Bundles

The Android data Bundle is a bundle in which various values can be added and sent together. For example, if we want to send multiple values via `putExtra()`, we create a Bundle, add all of those values inside that Bundle, and then send this Bundle with the `intent.putExtras()` method.



You can provide data directly to the intent by adding all the values individually to the intent, or the second method is to do it by adding all the values in the Bundle and sending it through the intent.

We will now take a look at how it is possible to provide data to the Bundle and send this Bundle to the next activity, by taking the previous activity and modifying it a little. While sending data from `Activity1`, instead of adding different values directly to the intent, we do the following:

```
Intent myIntent = new Intent(MainActivity.this, Activity2.class);

Bundle newBundle = new Bundle();

newBundle.putString("EDITTEXT_ONE_VALUE", valueOne);
newBundle.putString("EDITTEXT_TWO_VALUE", valueTwo);
newBundle.putString("EDITTEXT_THREE_VALUE", valueThree);

myIntent.putExtras(newBundle);
startActivity(myIntent);
```

As you see in the code, `valueOne`, `valueTwo`, and `valueThree` are added inside the Bundle using the `newBundle.putString()` function with a unanimous key for each data value. Now, this Bundle is added inside the intent using the `intent.putExtras(newBundle)` function, and then we call the `startActivity()` function as it was called in the previous example.

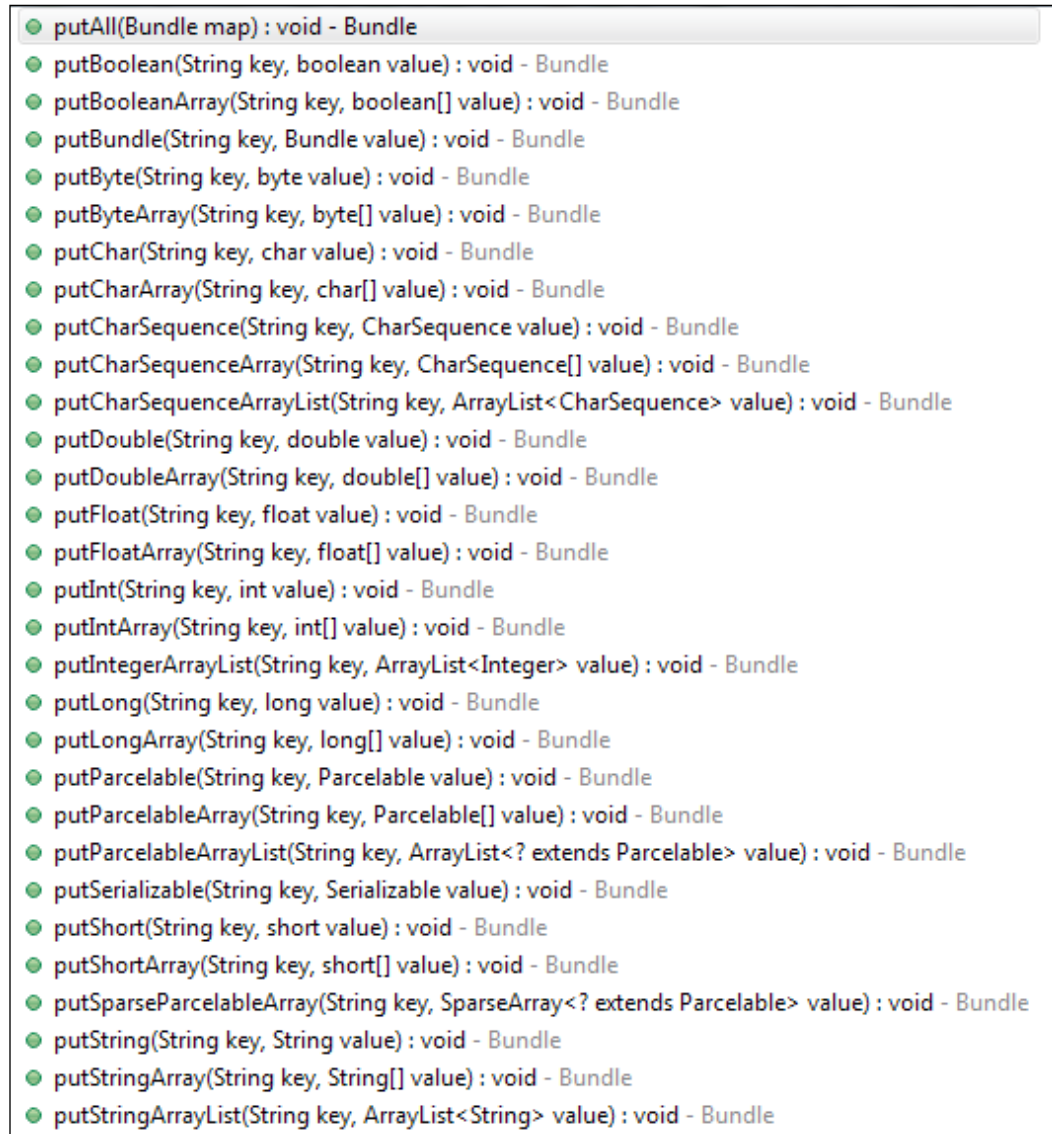
On the destination activity, we can directly catch the data by extracting the data bundle first using the `getIntent().getExtras()` function. This will return the Bundle object, and by referencing that specific key (that we added in the source activity), we can extract the data using following function for all the three values:

```
Bundle.getString("EDITTEXT_ONE_VALUE", "DEFAULT_VALUE");
```



The second parameter in `Bundle.getString(key, defaultValue)` is the default value that will be returned if the value of the specified key is not found.

Take a look at the following screenshot. You will see the different data types that can be simultaneously added into a Bundle:



Different functions that are used to add different data types inside a Bundle

Data transfer using Parcelable

The second and the most important method that is used to transfer data between activities is `Parcelable()`. The previous method has a restriction according to which we can only send the primitive data types such as `Strings`, `Integers`, `Doubles`, and `Floats`. In a practical scenario, when we work on projects, there are custom objects that we need to transfer between activities. These custom data objects hold information according to the need of the application. Hence, it should be transferred accordingly.

As it is now clear that the previous version is for the data transfer of basic data types only, `Parcelable` can be called as the subtype of the previous type.

In this method, the data class is inherited by implementing the `Parcelable` class interface in order to make its object compatible with the `putExtra()` intent method. We also need to override some of the methods from the `Parcelable` class interface in order to give it the functionality. Once it is done, the object of that class can be placed inside the intent or the `Bundle` to navigate it through the activities.

Implementation of Parcelable

In this section, we will learn how to implement `Parcelable` on a data class and then how to transfer that object between activities. There are two scenarios for this:

- Only one object is being sent from the source class to the destination class
- An array of the custom objects is being sent from the source class to the destination class

Keeping the tradition, we will start by creating a new Android project. In the given example, we call it the Parcel application. This project has `Activity1.java` as the default activity that will be created when the project is newly created. It will also behave as the source activity. The second activity will be `Activity2.java` that will act as the destination activity that will receive the parcel.

Implement a readymade tutorial

Once we are done with making a new project, insert this code inside your application. This will affect `Activity1.java`, `Activity2.java`, `layout_activity1.xml`, `layout_activity2.xml`, and `AndroidManifest.xml`, and introduce another class named `Person.java`.

```
//-----  
//The Activity1 Class  
  
public class Activity1 extends Activity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.layout_activity1);  
  
        final EditText nameText = (EditText) findViewById(  
            R.id.edittext_enter_name);  
        final EditText sirnameText = (EditText) findViewById(  
            R.id.edittext_enter_sirname);  
        final EditText addressText = (EditText) findViewById(  
            R.id.edittext_enter_address);  
  
        Button enterButton = (Button) findViewById(R.id.button1);  
        final Person firstPerson = new Person();  
        enterButton.setOnClickListener(new OnClickListener() {  
  
            @Override  
            public void onClick(View v) {  
                // TODO Auto-generated method stub  
                firstPerson.setFirstName(nameText.getText().toString());  
                firstPerson.setSirName(sirnameText.getText().toString());  
                firstPerson.setAddress(addressText.getText().toString());  
  
                Intent parcelIntent = new Intent(Activity1.this,  
                    Activity2.class);  
                parcelIntent.putExtra("FIRST_PERSON_DATA", firstPerson);  
                Activity1.this.startActivity(parcelIntent);  
  
            }  
        });  
    }  
  
    @Override  
    public boolean onCreateOptionsMenu(Menu menu) {  
        // Inflate the menu; this adds items to the action bar if it  
        //is present.  
  
        getMenuInflater().inflate(R.menu.activity1, menu);  
        return true;  
    }  
}  
  
//-----
```

//The MySecondActivity Class

```
public class Activity2 extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);
        setContentView(R.layout.layout_activity1);

        Person incomingPersonObj = getIntent().getParcelableExtra(
            "FIRST_PERSON_DATA");
        TextView nameTextView= (TextView) findViewById(
            R.id.person_name_text);
        TextView sirnameTextView= (TextView) findViewById(
            R.id.person_sirname_text);
        TextView addressTextView= (TextView) findViewById(
            R.id.person_address_text);

        nameTextView.setText(incomingPersonObj.getFirstName());
        sirnameTextView.setText(incomingPersonObj.getSirName());
        addressTextView.setText(incomingPersonObj.getAddress());

    }
}
```

//-----
//The layout_activity1.xml File

```
<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=".Activity1" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginTop="20dp"
        android:text="@string/name_text"
```

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

<EditText
    android:id="@+id/edittext_enter_name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/textView1"
    android:layout_alignBottom="@+id/textView1"
    android:layout_alignParentRight="true"
    android:layout_toRightOf="@+id/textView1"
    android:ems="10"
    android:inputType="textPersonName" >

<requestFocus />
</EditText>

<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/edittext_enter_name"
    android:layout_marginTop="20dp"
    android:layout_toLeftOf="@+id/edittext_enter_sirname"
    android:text="@string/sirname_text"
    android:textAppearance="?android:attr/textAppearanceMedium" />

<EditText
    android:id="@+id/edittext_enter_sirname"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/textView2"
    android:layout_alignBottom="@+id/textView2"
    android:layout_alignLeft="@+id/edittext_enter_name"
    android:layout_alignRight="@+id/edittext_enter_name"
    android:ems="10" />

<EditText
    android:id="@+id/edittext_enter_address"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/edittext_enter_sirname"
    android:layout_alignRight="@+id/edittext_enter_sirname"
    android:layout_alignTop="@+id/textView3"
    android:ems="10" />

<TextView
    android:id="@+id/textView3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
        android:layout_below="@+id/edittext_enter_sirname"
        android:layout_marginTop="20dp"
        android:layout_toLeftOf="@+id/edittext_enter_address"
        android:text="@string/address_text"
        android:textAppearance="?android:attr/textAppearanceMedium" />

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@+id/edittext_enter_address"
    android:layout_below="@+id/textView3"
    android:layout_marginRight="10dp"
    android:layout_marginTop="33dp"
    android:text="@string/enter_button_text" />

</RelativeLayout>

//-----
//The activity_two_layout.xml File

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

    <TextView
        android:id="@+id/person_name_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/null_string"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/person_sirname_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/null_string"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/person_address_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/null_string"
```

```
        android:textAppearance="?android:attr/textAppearanceMedium" />
</LinearLayout>

//-----
//The Person.java File

public class Person implements Parcelable {

    private String firstName;
    private String sirName;
    private String address;

    public Person(){
        firstName = null;
        sirName = null;
        address = null;
    }
    public Person(String fName, String sName, String add) {
        firstName = fName;
        sirName = sName;
        address = add;
    }

    public Person(Parcel in) {
        firstName = in.readString();
        sirName = in.readString();
        address = in.readString();
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getSirName() {
        return sirName;
    }

    public void setSirName(String sirName) {
        this.sirName = sirName;
    }

    public String getAddress() {
        return address;
    }
}
```



```
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public int describeContents() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        // TODO Auto-generated method stub
        dest.writeString(firstName);
        dest.writeString(sirName);
        dest.writeString(address);
    }

    public static final Parcelable.Creator<Person> CREATOR =
        new Parcelable.Creator<Person>() {

            public Person createFromParcel(Parcel in) {
                return new Person(in);
            }

            public Person[] newArray(int size) {
                return new Person[size];
            }
        };
}

//-----
//The AndroidManifest.xml File

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

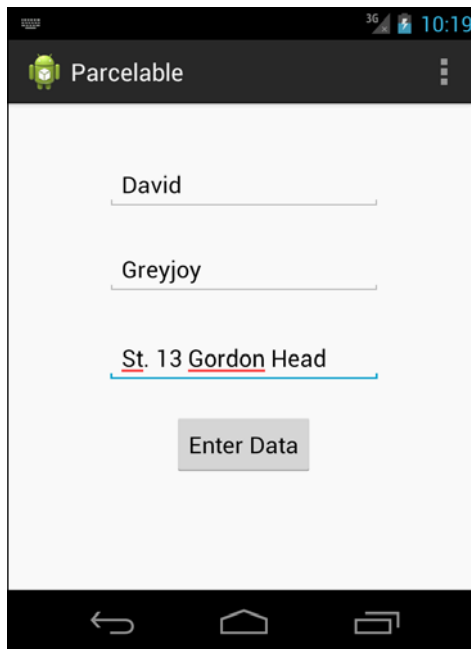
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
```

```
android:icon="@drawable/ic_launcher"
android:label="@string/app_name"
android:theme="@style/AppTheme" >
<activity
    android:name="com.app.parcelapplication.Activity1"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name="com.app.parcelapplication.Activity2"
    android:label="@string/app_name" >
</activity>
</application>

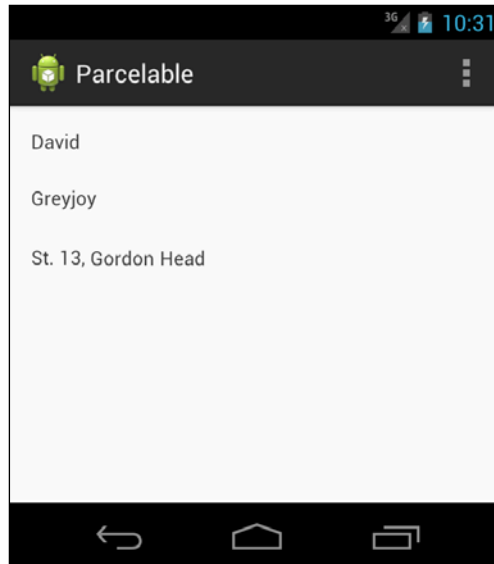
</manifest>
```

Run this application and it will bring the Activity1 output screen on your device. Take a look at the following screenshot to see how the application will appear on the Android screen:



Activity1 with three EditText fields in order to take the input from the user, and a button to transfer the data to the next activity

When you are on the first screen, enter the data and press the button to migrate to the next activity that will show the entered data. The screen will look as follows:



Activity2 showing the transferred data from Activity1 in the form of Parcelable

Understanding the Parcelable implementation

In order to understand the working of this example, we need to first understand how `Parcelable` works. In Android, there is a need to transfer custom data (that is, custom objects and arrays of custom objects) from one activity to another. Normally, the custom data classes are not compatible with the extras; so, we implement the `Parcelable` interface to that class.

What `Parcelable` does with the custom data class is that it develops compatibility with extras. The object or objects of the class implemented using `Parcelable` can be added easily inside the `putExtra()` method of the intent. Similarly, it can also be part of the `Bundle` object that can later be transferred via intents.

We can now go through the explanation of the preceding code.

The Activity1.java class

This is the source class from which the `Parcelable` object will start to migrate. It starts with the implementation of the `onCreate()` method. In this method, after setting the main `View`, we found `Views` by their IDs and brought their objects to the activity. The `Views` include three `EditText` fields and a button. They are used to take inputs from the user and trigger the event in order to start transferring the data to the next activity.

Inside the `button.setOnClickListener()` method, we pass a new `OnClickListener()` object inside which the `onClick()` method is overridden. We want the intent to start once the button is clicked; that is why we are implementing the intent and taking the data from the fields inside the `onClick()` method.

Now, we don't want the method to directly transfer the data to the intent. That is why we are making an object of the `Person.java` class that will hold the values obtained from the fields. We name the object as `firstPerson`. In order to set the values to this object, we implement the following line of code:

```
firstPerson.setFirstName(nameText.getText().toString());
```

The preceding line will set the first name of that object to the value that is obtained from what is written inside the first `EditText` field. The first `EditText` field, `nameText`, holds the value of the first name. So, using the `nameText.getText()` method, it will return the `Editable` object that can be easily converted by calling the `toString()` method on it.

The same method will be repeated in order to get the value from the second and the third `EditText` fields. They will be set inside the same `Person` object. You can see this being done using the following lines of code:

```
firstPerson.setSirName(sirNameText.getText().toString());  
firstPerson.setAddress(addressText.getText().toString());
```

At this stage, the `firstPerson` object is ready to be delivered from `Activity1` to `Activity2`. As the object is inherited by implementing `Parcelable`, we can directly add it inside the `extra`. We will learn how to implement `Parcelable` in the forthcoming section. Here, we will see how to add `Parcelable` inside the intent object.

Make an object of the `Intent` class and give it the source and destination, that is, the source context and `.class` reference of the destination class in order to let it know from where to initiate this intent and where to end. We can add the `Parcelable` by calling `parcelIntent.putExtra()`. See the following line of code:

```
parcelIntent.putExtra("FIRST_PERSON_DATA", firstPerson);
```

Using this, we can easily add the custom `Parcelable` data object inside the intent object, and in the next line, simply call the `startActivity()` function in order to start the intent.

The Activity2.java class

In this class, we will learn how to catch the transmitted `Parcelable` object in the destination class. For this, first of all, start with the normal procedure of implementing the `onCreate()` method of the activity. Set the Content View and bring in three text Views by finding the IDs from the layout. These three text views will show the received values of the first activity's object.

The `getIntent()` method will receive the intent object that was transmitted by the `Activity1.java` class that holds the data. Once the object is obtained, we can get its extras by calling the `getExtras()` method that will return the `Bundle` that holds the data. Call the `getParcelable()` function on that `Bundle` with the key in order to retrieve the object. This object is now taken by a new object of the `Person` class named `incomingPersonObj`.

Now, we have the same object that was initiated from the source class at the time of calling the intent from `startActivity()`. We will now set the text of the text views by calling the following lines of code:

```
nameTextView.setText(incomingPersonObj.getFirstName());
sirnameTextView.setText(incomingPersonObj.getSirName());
addressTextView.setText(incomingPersonObj.getAddress());
```

The `incomingPersonObj.getFirstname()` method will get the first name of the person from `incomingPersonObj` and set its value directly to `nameTextView` when the first method is called. The procedure is the same for the `sirnameTextView` and `addressTextView` objects.

The layout_activity1.xml file

This is the layout file that contains the Views of `Activity1.java`. As mentioned in the code, it contains three `EditText` fields with the IDs: `edittext_enter_name`, `edittext_enter_sirname`, and `edittext_enter_address`. Apart from that there are also three text Views which are used to simply indicate which `EditText` field contains which value.

Every activity requires an event trigger that is used to start any process. In this layout, the button will do the task; hence, it is also placed below the `EditText` fields.

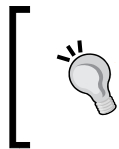
The layout_activity2.xml file

This layout file creates the layout of `Activity2.java` that is the destination activity. This activity is responsible for extracting the data and showing it in its layout. The layout consists of three `TextView` Views whose IDs are `person_name_text`, `person_sirname_text`, and `person_address_text`. These IDs are used to bring these Views to the code (as you can see in the second part of the code).

The Person.java class

The `Person` class is basically the data-holder class whose objects will be created anywhere in the application. This is also called the **bean class** that is used to hold the data coming from servers in JSON, XML, or any other format. In our `Person` class, it has three fields. All fields are private with their respective public getters and setters. The `firstName`, `lastName`, and `address` objects represent the kind of information they will hold. The `Activity1.java` class makes an object of this class, takes the data from the `EditText` fields, and adds it inside the object.

This class is inherited by implementing the `Parcelable` interface. This `Parcelable` interface needs some important things to be implemented. First of all, we will implement a constructor of this class that will take `Parcel` as an argument. This constructor will be used from inside this class while implementing the `Parcelable` interface. The `in.readString()` method is used to read the value from the parcel.



In order to make this technique work, read the parcel in the same order in which it was written in the `writeToParcel()` method. See the order of writing the parcel in the code. It is `firstName`, `lastName`, and `address`. The same thing can be observed in the constructor.

The `writeToParcel()` method is overridden in order to produce objects of the same class by `Parcelable` so that it can be used. `Parcelable.Creator<Person>` is used to create instances of that class as used by `Parcel`; it uses the `writeToParcel()` method to do the job.

Once the object is prepared, it is then forwarded to the next activity and caught by the `Activity2.java` class as it is explained in the first and second parts of the given code.

The AndroidManifest.xml file

The importance of this file cannot be neglected when you are talking about developing an Android application. We need to add both the activities in this file in order to get them recognized by the Android application. As you can see in the file, both activities have their own tags in the manifest file along with their parameters and intent filters.

Future Consideration

The preceding method of implementing `Parcelable` is for transferring only one `Parcelable` object inside the extras or a `Bundle`. Similarly, we can transfer an `Array` or `ArrayList` of the custom data beans by implementing `Parcelable`.

Data transfer using Serializable

The third type of data transfer method that is used in intents is `Serializable`. Many Java developers are already familiar with the term `Serializable` as it was used earlier, way before the introduction of Android. The biggest advantage of Android is that its development takes place on Java in SDK, and on C++ it's in NDK. This makes it extremely lenient and powerful.

The same is the case with the functionality; `Java.io.Serializable` is purely a function of Java that can be used as it is in Android development. The `putExtras()` intent has an option of transferring the Java-serialized object from one activity to another without any specific amount of effort. We start this section with the introduction of `Serializable` for non-Java users.

What is Serializable?

Java comes with a mechanism in which an object can be represented in a byte array. It's not just the data that is serialized, but the information about the object type and what kind of data is placed inside the object can also be found in it.

These serialized objects can be written in the file and stored in any external storage (such as an SD Card). The process of remaking the object is called deserialization in which the information that is hidden inside the byte array can be gathered to regenerate the object in the memory at the time of need.

The process of making and remaking any serialized object is completely JVM independent. It means that the object can be serialized in Java and can be remade in any language that also supports JVM with the same Java version as at the time of making it serialized.

In Java, the `ObjectOutputStream` class is used to serialize the object while the `ObjectInputStream` class is used to make an object when we are regenerating from an existing serialized object. These classes contain the `writeObject()` and `readObject()` methods respectively. The methods actually start the process of serialization or deserialization.

An example of Serializable

In this section, we will see how `Serializable` is done in Java. This is the inside mechanism of how Android handles these objects. This example contains two methods to perform the tasks of serialization and deserialization.

First of all, the method for serialization is as follows:

```
private void serializeObject(){
    Person person = new Person();
    person.name = "John Doe";
    person.address = "St. 16 China Town";

    try {
        FileOutputStream fileOut = new FileOutputStream("/tmp/person_data.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(person);
        out.close();
        fileOut.close();
        System.out.printf("Serialized data is saved in /tmp/person_data.ser");
    } catch (IOException i) {
        i.printStackTrace();
    }
}
```

The `Person` class implements the `Serializable` interface. This will enable the object to be recognized by the `ObjectOutputStream` class's object. First of all, we will create a serialized object as shown in the code by setting the name and address.

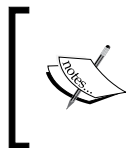
Once the object is created, it is ready to be serialized. We start by making an object of `FileOutputStream` that is used to write data into a file; the path that will refer to the location where that serialized file exists is also added. Make an `ObjectOutputStream` object that will have that file referenced by `ObjectOutputStream out = new ObjectOutputStream(fileOut)`. We are now ready to write the object by calling `out.writeObject(person)`. This will start to serialize the object (convert it to a byte array) and add it to the given location. We will then close the `out` and `fileOut` objects.

Reading data from a serialized source is the next step that we will see. See the following code:

```
private void deserializeObject() {
    Person person = null;
    try {
        FileInputStream fileIn = new FileInputStream("/tmp/person_data.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        person = (Person) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return;
    } catch (ClassNotFoundException c) {
        System.out.println("Person class not found");
        c.printStackTrace();
        return;
    }
    System.out.println("Deserialized Person...");
    System.out.println("Name: " + person.name);
    System.out.println("Address: " + person.address);
}
```

The code is simple to understand because it contains almost the same steps as those required for writing an object as a serialized file. We will create an instance of the `Person` class to hold the remade object. The `FileInputStream` object is created. It directs towards the location of the file that is to be deserialized. The `ObjectInputStream` object is used to get that file path and make it ready to be read. Using this, the `in.readObject()` method is called in order to deserialize the object, and it will return the `person` object. Once it is done, we will close the `in` and `fileIn` objects.

We now have the object of the deserialized `Person` class which can be in log or printed on the console.



The method of writing serializable objects in Java and Android are the same. While we are doing it in Android, we can write the file in an SD Card. The file can later be fetched and deserialized by any other activity or application.

Implementation of Serializable

Until now, we have understood the main reasons of using `Serializable`. A quick review is always good. The `Serializable` technique is used to convert an object into byte array; we can use it to write into a file and store it as a `.ser` file on an SD card or any other storage. This serialized object can then be read from any location irrespective of the activity.



`Serializable` is the simplest method to perform data transfer. It is also used for transferring one or more custom data objects from one activity to another.

It is not necessary for serialized files to always have the `.ser` extension.

Just like the other examples and implementations, we will start this by making a fresh project. This project will have two activities; one will be the source and the other will be the destination. The serialized object will start to navigate from one activity, and the destination activity will catch it in order to extract data from it. Android supports the native Java procedure of serializing and deserializing the object; that is why, we don't have to do anything because the phenomenon of serialization is handled by Android itself.

Passing Serializable – a tutorial

In this chapter, the implementation of `Serializable` will start by making a new project. By default, this project will have one activity (let's say `Activity1.java`). Implement the following steps that will lead you to make a project in which `Serializable` is implemented. We will then see its explanation.

Starting with the first step, implement the following code in your newly created Android project. This will introduce three new files:

- `Activity2.java`: This will act as the destination activity.
- `layout_activity2.xml`: This file will hold the layout for the destination activity.
- `Person.java`: This is the serialized class that is responsible for giving data beans' objects.

```
// =====
//The Activity1.java file

public class Activity1 extends Activity {

@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.layout_activity1);

    final EditText edittext1 = (EditText) findViewById(R.
id.editText1);
    final EditText edittext2 = (EditText) findViewById(R.
id.editText2);
    final EditText edittext3 = (EditText) findViewById(R.
id.editText3);

    Button button = (Button) findViewById(R.id.button1);
    button.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            // TODO Auto-generated method stub
            Person person = new Person();
            person.setName(edittext1.getText().toString());
            person.setSurname(edittext2.getText().toString());
            person.setAddress(edittext3.getText().toString());

            Intent intent = new Intent(Activity1.this, Activity2.class);
            intent.putExtra("PERSON_OBJECT", person);
            startActivity(intent);

        }
    });
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

}

//=====
//The Activity2.java file

public class Activity2 extends Activity {
```

```
String TAG = "MainActivity2";
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.layout_activity2);
    Person person = (Person) getIntent().getExtras().
        getSerializable("PERSON_OBJECT");
    TextView textView = (TextView) findViewById(R.id.data_text);

    if(person != null){
        textView.setText("Data Successfully caught");
        Log.d(TAG, person.getName());
        Log.d(TAG, person.getSirName());
        Log.d(TAG, person.getAddress());

    }else{
        textView.setText("Data Object is null");
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    // present.
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}

//=====
//The Person.java

public class Person implements Serializable {
    String name;
    String sirname;
    String address;

    private static final long serialVersionUID = 1L;

    public String getName() {
        return name;
    }
}
```

```
public void setName(String name) {
    this.name = name;
}

public String getSirname() {
    return sirname;
}

public void setSirname(String sirname) {
    this.sirname = sirname;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}
}

//=====
//The layout_activity1.xml file

<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" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="26dp"
        android:ems="10" >
```

```
        <requestFocus />
    </EditText>

    <EditText
        android:id="@+id/editText2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText1"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="41dp"
        android:ems="10" />

    <EditText
        android:id="@+id/editText3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/button1"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="22dp"
        android:ems="10" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        android:layout_marginBottom="153dp"
        android:text="Enter Data" />

</RelativeLayout>

//=====
// The layout_activity2.xml file

<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:id="@+id/data_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />

</RelativeLayout>

//=====
//The AndroidManifest.xml file

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

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.app.serializable.Activity1"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN"

/>

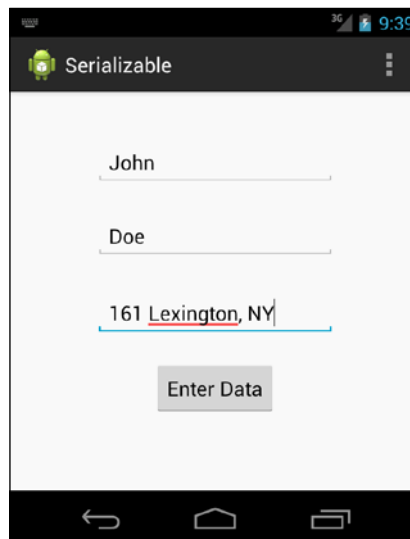
                <category android:name="android.intent.category.
LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="com.app.serializable.Activity2"
            android:label="@string/app_name" >
            <intent-filter>
```

```
        <action android:name="android.intent.action.MAIN"
/>

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

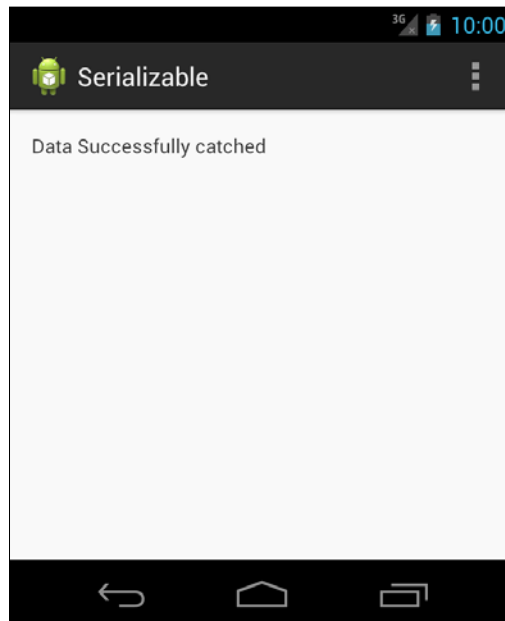
</manifest>
```

Once you are done with the implementation, run the project. The following screen will appear in which you need to add your data into the fields. Then click on the **Enter Data** button:



Source activity asking data to be entered to send it through serializable

As soon as you press the **Enter Data** button, `Activity2.java` will be opened and the following screen will be shown:



Activity2 showing that the data is successfully entered

Meanwhile, when you see the LogCat of the project, it will show the data that we logged in the `Activity2.java` file. See the following screenshot:

```
D 07-26 09:59:4... 493 493 com.app.serializable MainActivity2 John
D 07-26 09:59:4... 493 493 com.app.serializable MainActivity2 161 Lexington, NY
D 07-26 09:59:4... 493 493 com.app.serializable MainActivity2 Doe
```

LogCat showing the successful catching of data in Activity2

Walking through the Serializable code

In order to understand the working of `Serializable` in Android, you need to go through the previously described details of `Serializable` in Java. We recommend you to have a look at this if you haven't already done so. In this section, we will rather focus on the explanation of the preceding example and how to implement it in an Android project.

For the sake of simplicity, we have divided it into six parts. The explanation of each part is given in detail.

The Activity1.java class

The `Activity1.java` class will act as the source activity from which the intent will initiate. It will also act as the source activity because it is responsible for creating and sending the custom data object. Let's start with the very first part of the code, which is the implementation of the `Activity1.java` class.

As it was mentioned earlier, this class is responsible for taking data inputs from the user and making a data object out of it. Inside the `onCreate()` method, we will first set a particular layout that holds the Views using the `setContentView()` method. Once the layout is set, our next task is to bring those Views as objects in our Java code, as shown in the following code. This will help us perform various tasks on those objects that are bound on those Views in the layout.

```
final EditText edittext1 = (EditText) findViewById(  
    R.id.editText1);  
final EditText edittext2 = (EditText) findViewById(  
    R.id.editText2);  
final EditText edittext3 = (EditText) findViewById(  
    R.id.editText3);
```

Calling the `findViewById()` function brings the particular View with which the ID is associated. We cast it to the `EditText` class, and take it inside the `edittext1`, `edittext2`, and `edittext3` objects respectively.

These three fields will be used to take the input from the user, but we need an event trigger that is used to navigate the user from one activity to another and is also responsible for data transferring. For that, we implement a button in the layout, and we will fetch it in the Java code by calling the `findViewById()` method:

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

Now, we have all the necessary views in our Java code. Our next step is to implement the button functionality, that is, what it will do when it is clicked on. For that, we need to implement an `OnClickListener` interface on this button:

```
Button.setOnClickListener(new OnClickListener())
```

The preceding line of code is responsible for setting the click listener on the button. It takes an argument of `OnClickListener` inside which we implement the `onClick()` method. This `onClick()` method will be responsible for assigning a job to the button:

```
button.setOnClickListener(new OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // TODO Auto-generated method stub  
    }  
  
});
```

Once we are done with that, we will make an object of the `Person` class (the `Serializable` object) and set its values to the one that is obtained by taking the input from the `EditText` fields. Now, it has two parts; the first is to make a new object of the `Person` class. We will do this by calling its constructor:

```
Person person = new Person();
```

In the second part, we will get the values of the `EditText` object by calling the `getText()` method using that object. The `getText()` method returns an `Editable` object; so, in order to convert it into string, we call the `toString()` method on it. When you observe the code, we have performed all of these tasks together:

```
person.setName(edittext1.getText().toString())
```

First, the value from the `edittext1` object is brought and converted to `String`. Second, we are setting the person's name by its value. We will further set `sirName` and `address` of the `person` object using a similar procedure:

```
person.setSirname(edittext2.getText().toString());  
person.setAddress(edittext3.getText().toString());
```

Now, we have an object that is ready to be transferred. We will now make an object of `Intent` and assign its source and destination activity's contexts. It will represent which activity the intent is initiated and to which activity it will migrate. We will do this by calling the constructor:

```
Intent intent = new Intent(Activity1.this, Activity2.class);
```

Once the `intent` object is made, we will add the data object inside this `intent` and call the `startActivity()` method. In order to put the serialized object in the `intent` object, we will call the `intent.putExtra()` method. The final step for this part is to call the `startActivity()` method that will initiate the process of navigation.

The `Activity2.java` class

The destination activity's main purpose is to catch the `intent`, extract data object from it, and show it in the Views. We start with the implementation of the `onCreate()` method. First, the layout is set by calling the `setContentView()` method of the `Activity` class. Now, it is time to catch the `intent` object that was initiated from the `Activity1.java` class.

Just like the previous example in this chapter, we obtain the `intent` by calling the `getIntent()` function. This function returns an `intent` object that is used to launch this activity. Once the `intent` object is here, we call the `getExtras()` function. This will return a `Bundle` that contains all the extras that were added on this `intent` object by the sender activity.

On this Bundle, we will now call the `getSerializable()` method that will bring the `Serializable` object with the help of the `key` value that is given to it by the sender activity. That `key` value should be identical to that of the sender activity; otherwise, it will return a null value that may result in the crashing of your application due to `NullPointerException`.

The `Person` object is now in hand with all the values. Our next task is to log these values in the LogCat so that we can verify it. A null-pointer check is implemented in order to see whether or not the object is null. If it is not null, we log its values by getting it from `person.getName`, `person.getSirName` and `person.getAddress`. If the object is null, it will say `Data Object is null`, and hence, it will not crash.

```
Log.d(TAG, person.getName);
```

The Person.java class

When we talk about transferring data from `Serializable`, the `Person.java` class is the most important class that we need to implement in order to do the transferring. We made a Java class that consists of three private string variables. Each one of them has its own getter and setter functions in order to get and set the value from outside of the class. As in the previous method, we implemented our data bean class using the `Parcelable` interface; here, we will implement our class with `Serializable`.

Once it is implemented, Android is ready to treat the objects of this class as `Serializable`. Now, whenever it is added inside the intent object, Android will transfer it as a byte array. This is a slow process as compared to `Parcelable`, but its slowness is not noticeable when we do it on few objects. If we want to apply this method where there are thousands of data objects, it may take some time.

The layout_activity1.xml file

The layout file belongs to the `Activity1.java` class. When you run the code for the first time, the layout that will appear is described in this layout file. In the `Activity1.java` class inside the `onCreate()` method, we used the `setContentView()` method in order to paste the user interface.

In this XML file, there are four Views; three of them are `EditText` fields that are placed to take input from the user in `Activity1`. Apart from that, there is a button that is used to trigger an event after the data is completely filled in the fields. The IDs given to them are the default ones that were used by the `Activity1.java` class to fetch these Views inside the `.java` class, and `edittext1`, `edittext2`, and `edittext3` are the IDs for their respective fields.

The layout_activity2.xml file

This layout file contains the Views for the `Activity2.java` class. It consists of one `TextView` View that will tell us whether or not the values coming from the `Activity1.java` class have been correctly fetched. This text View will then show the **Data successfully caught** or **Data object is null** message in accordance with the data object.

The AndroidManifest.xml file

The `AndroidManifest.xml` file consists of all the activities, permissions, SDK information, version codes, and many other things. In short, it is used to keep all the information regarding the applications. In this, we have our activities, `Activity1.java` and `Activity2.java` classes along with their intent values. If you forgot to mention any of your activity in this file, it will produce an exception in the LogCat saying `ClassNotFoundException`.



`Parcelable` and `Serializable` are two methods used to transfer data objects from one activity to another. `Serializable` is the simplest one to implement while `Parcelable` is the fastest of all. In case you need to perform a task based on fewer objects and you want a simple solution, you should go for `Serializable`. But if you want a perfect method irrespective of the complexity of the implementation, you should go for `Parcelable`.

Data and the implicit intents

In the preceding examples, the need of data transfer was described within an application. It is now very clear that any application is incomplete without the transfer and manipulation of data. In this section of the chapter, we will see the scenarios in which there is a need to transfer data to implicit intents.

Recalling the definition of implicit intents, they are those intents that normally do not direct towards a specialized target rather they give the flow of the application to an outside application, or in other words, they start another activity in order to perform a certain task.

The outside applications require some data from your application in order to perform tasks. We will now see the scenarios in which the data transfer in the form of URI is passed to the implicit intent.

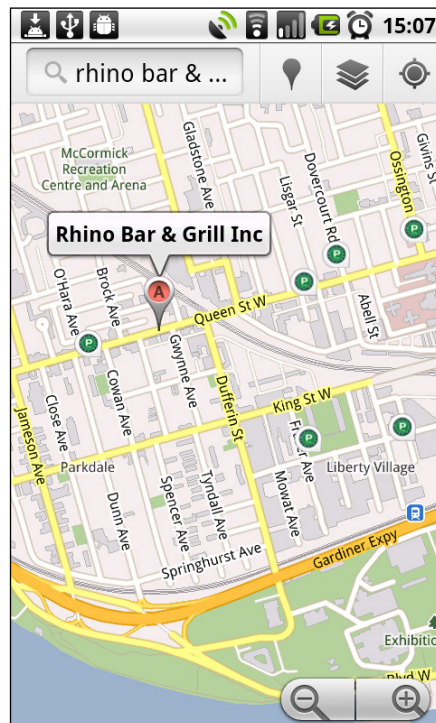
Viewing a map

Google Map can be initiated from your own application with a particular place. It means that you can send any location that you need to open in the Google Map via implicit intents. Based on the latitude and longitude, you can open a particular point in the Google Map. This latitude and longitude is given to the Google Map using a URI that is a way to send data to implicit intents.

In order to perform this task, we need to write the following line of code:

```
String uri = "geo:" + latitude + "," + longitude;
Intent mapViewIntent = new Intent(android.content.Intent.ACTION_VIEW, Uri.parse(uri));
startActivity(mapViewIntent);
```


There is a particular syntax according to which we need to write the URI string. In order to open the map with a particular location, the URI string consists of the `geo` keyword followed by the latitude and longitude (comma separated as shown in the code). This URI value is given to the implicit intent with the `android.content.Intent.ACTION_VIEW` action. This View action will take the URI and open the best available application to perform the task. We will then start the intent by calling the `startActivity(intent)` method.



Google Map View in an Android OS.

Opening a webpage

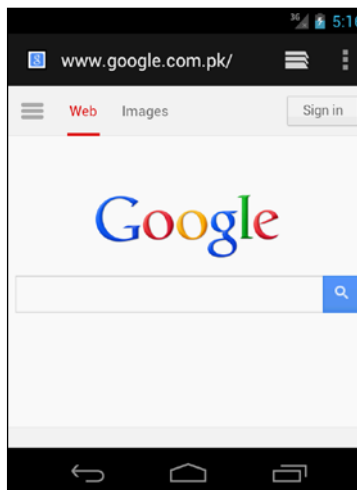
The implicit need of transferring data can also be categorized in case you want to open a webpage, where you need to open the default web browser with a particular loaded page. In this process, we need to transfer the URL that our application wants to open in the browser. This URL is also passed with the help of the `Uri.parse()` method.

[ Keep in mind that we are using the default web browser in this scenario. It is not the web View that comes as a part of an Android application.]

Implement the following lines of code in order to send and open the URL in the default web browser:

```
String url = "http://www.google.com";  
Intent browserIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));  
startActivity(browserIntent);
```

As you can see in the code, there is a string that contains the value (URL) that is to be opened in the web browser. This value is then entered in the constructor of the intent in the `Uri.parse()` function with an `Intent.ACTION_VIEW` action. This will choose the best available option to open the URL.



The view of a browser opening the Google.com webpage as called from our application

Sending an e-mail

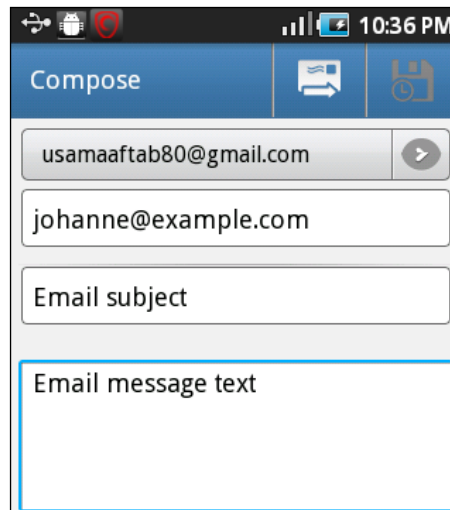
There is a probability that you need a function that needs to call the default Google Mail application with a particular typed e-mail and a particular recipient in your application. In this case, we again need to add the data, that is, the sender name, e-mail body, and e-mail subject into the intent object and start the activity with that.

In order to perform this task, we need to write the following lines of code:

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"Johanne@example.com"}); // recipients
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
startActivity(emailIntent);
```

Make an `Intent` object with the `Intent.ACTION_SEND` action. Its work is to open the intent with the sending option. Now, it's time to add the data inside this object. The Android API caters to all the scenarios that can occur; hence, there are certain constants defined in the `Intent` class that can be used to uniquely identify the data by Android. `Intent.EXTRA_EMAIL` is the keyword constant that is used in the `putExtra()` method while you are giving an e-mail address to the intent. Similarly, there is a keyword constant for mentioning the subject in the extras; `Intent.EXTRA_SUBJECT` and `Intent.EXTRA_TEXT` will be used to add the body of the e-mail.

Once we call the application with these parameters, it will open Gmail with these parameters filled in the fields. It will look something similar to the following screenshot:



A view of the Gmail application as called by our application

Making a call

If you want to initiate a call with a certain number inside your application, you need to call the dialer intent. Using `Intent.ACTION_DIAL`, you can provoke the dialer intent with a particular number given as the URI. Follow the given code to implement the dialer functionality in your application:

```
Uri number = Uri.parse("tel:33566264");
Intent callIntent = new Intent(Intent.ACTION_DIAL, number);
```

The URI string contains the telephone number with which the dialer should initiate. Once the dialer is opened, it will show the number as it is written, and the user can now dial that number.

Miscellaneous scenarios

There are various other scenarios that can be included in this chapter (such as calendar and time widgets), but due to limited space and constraints, we have taken only four scenarios into consideration. The implementation of data transferring between implicit intents is of extreme significance and can be done with great ease.

Summary

In this chapter, we had a detailed learning of how to play with data inside an Android application. We learned how to transfer data from one activity to the other using different methods and the simple transfer of default data structures using the `putExtra()` function of `Intent`. The custom data objects or the array of custom data objects can be sent to another activity using `Parcelable` and `Serializable`. We also learned how to implement all these kinds of data-transferring methods in our Android application. At the end of the chapter, we briefly covered four scenarios in which the data is sent to other applications (using implicit intents) when calling them through intents from our application.

This chapter is of great importance with respect to practical application development because transferring data between activities or even outside the application is a fundamental part of any Android application, and this can easily be done using Android intents.

In the following chapters, we will study the use of intents for accessing the Android features. We will also see how the intent filters work, what are the basics of the broadcasting intents, and at the end, we will see the implementation of intent service and pending intents.

6

Accessing Android Features Using Intents

In the last chapter, we discussed data transfer using components. We saw how to transfer data from one activity to the other, and why we should transfer data between different components. We also discussed the various methods of data transfer using intents. Android has a lot of components in the system, and intent provides an easy interface to make those components communicate with each other. In *Chapter 4, Intents for Mobile Components*, we discussed the different Android components that use system hardware such as Wi-Fi, Bluetooth, camera, microphone, and so on. We also discussed how these components can be utilized using intents and how we can make many different applications using Android hardware with no more than few lines of code.

Until now, we have only discussed hardware components and the role of intents with those components. This chapter is all about Android software features and how we can use those features in our applications using intents as the primary interface. Android contains a vast collection of libraries and APIs, by which a developer can use different Android features very easily. This chapter will walk us through the common Android features, and we will also develop some example applications that will show us the use of intents with those features.

This chapter includes the following topics:

- Features of Android OS
- Android features versus components
- Common Android OS features
- Android features and intents
- The `<uses-feature>` and `<uses-permission>` tags
- Sharing using the SEND action

- Sending SMS/MMS using intents
- Sending data messages using intents
- Confirming message delivery
- Receiving SMS
- Telephony and making calls using intents
- Sending notifications using intents
- Some other Android features



The concepts and structure of intents, as discussed in previous chapters, are the prerequisites for understanding this and the following chapters. If you don't have the basic concept of these things, we would recommend you to read *Chapter 3, Intents and Its Categorization* and *Chapter 4, Intents for Mobile Components* in order to move forward.

Features of Android OS

Android is an open source operating system and middleware framework for smart devices such as phones and tablets. The devices contain lots of features and functionalities that provide users with a way for an easy lifestyle. These features include hardware features such as audio, Bluetooth, camera, network, microphone, GSM, NFC, and sensors such as accelerometer, barometer, compass, gyroscope, and Wi-Fi.

Not only does it include hardware components, it also includes software features such as app widgets, home screen, input methods, live wallpapers, layouts, storage, messaging, multi-language support, browsers, Java support, media support, multi-touch, calls, messaging, multitasking, accessibility, external storage, and so on. We have already referred to the hardware features as mobile components and discussed them in the previous chapters. We will discuss the software features in this chapter with practical examples.



We are using two key terms here: components and features. Components such as camera, Bluetooth, and so on are the hardware parts of an android phone . The feature is the software part of an Android phone, such as an SMS feature, e-mail feature, and so on. This chapter is all about software features, their access, and their use through intents.

Android features versus components

Generally, the terms "Android features" and "components" are used interchangeably. But for the sake of clarification, we are referring to the keyword *components* as a feature that uses hardware and the keyword *features* as an Android feature that uses software in its backend. As we discussed in the previous section, Android contains lots of components and features that, when ported to any phone, makes it a smart phone. Not all the components and features can be used via intents. So, we will discuss only those features in detail that can be used by intents.

It should be noted that features that use hardware directly or indirectly require users to provide permissions to access it. These permissions are provided during the application's installation. If the user doesn't provide permissions to the application, the application can't access hardware; thus, it can't use that feature.

In this chapter, we will learn about those features that use software as backend but also require some permissions. We will provide more details about the permissions in the following sections.

Common Android features

Until now, we are only talking about Android features in a general way. In this section, we will discuss some of the most common Android features found in Android phones and tablets. Each Android device is unique in some way or the other and possesses many unique features and components different from other brands and phones. But there are some features that are found to be common in all the Android phones. Many of these features can be used in our apps irrespective of any specific model or phone, and intent is, without any doubt, the most asynchronous and easy way to use these features in our applications. Now let's see the Android features that are common among many devices and their functionality in a phone.

Layouts and display

Today, smartphones are getting bigger in size, and there are new sets of Android devices called tablets that are available. Bigger screens and higher-resolution displays have transformed mobiles into multimedia devices. These devices contain layout sizes from 240 x 320 to 1268 x 800 pixels and screen sizes from 3 to 11 inches. These are found in varying device screen densities such as low, medium, high, large, extra large, and so on. The following image shows three different devices with different resolutions:



Android devices with different screen sizes

To display high graphics, Android provides graphic libraries for 2D canvas drawings and 3D graphics with OpenGL using OpenGL-ES. A new rendering graphics library called RenderScript was introduced after Android Version 3.0. RenderScript is a scripting language for Android OS that allows developers to write high-performance graphic rendering and raw computational code. It has been primarily oriented for use with parallel data computations like dividing processing across multi-core processors such as CPUs, GPUs, or DSPs.



Before Android Version 3.0, which was developed for Android tablets, Android rendered its layouts, home screen, and mobile UI using 2D canvas. After RenderScript in Android 3.0, Android renders its layouts, home screens, and mobile UI with more beautiful and optimized graphics using RenderScript.

Data storage and retrieval

There is no Android device that doesn't use any kind of storage for running. For better performance, the device not only needs a volatile memory like RAM for the sake of processing and faster access, but it is also going to need a permanent storage such as an external SD Card in it. Android devices support data storage and retrieval in multiple ways that vary according to the developers and applications to be used.

If our applications use large data, the developers can use lightweight relational database features for each of their applications using SQLite. Developers can use the SQLite database to manage data with secret and efficient storage capability.

Not only databases, Android devices also provide features for file storage. As saving and loading data is essential for almost every application, Android provides many different methods to store and retrieve data to make our application persistent. File storage is not a good option, but sometimes, developers don't have any option other than reading and writing files to handle their application's persistent data. And fortunately, Android provides features that let developers create, save, and load files on a device's internal or external media, such as an SD card. These microSD cards are formatted with the FAT32, Ext3, or Ext4 file systems. Not only these file systems, but also some Android devices, mostly tablets, support high-capacity storage media such as USB flash drives.

Apart from storing large or heavy data in databases or disk files, Android provides a feature for storing simple application data such as UI state, game scores, and so on. This is achieved using the `SharedPreferences` method. The `SharedPreferences` method uses the name/value pair (NVP) mechanism to store the application's lightweight data.

Connectivity and communication

We can't call an Android device a smart phone until there is a connectivity feature in it. The technologies supported for connectivity, communication, and data transfer include GSM/Edge, IDEN, CDMA, EV-DO, Bluetooth, Wi-Fi, LTE, Near-Field Communication (NFC), WiMAX, and so on. Android provides a complete set of libraries for communication and connectivity. This allows developers to easily utilize and use these features in their applications. For example, through Bluetooth support, users can send files, access phone book and voice dialing, and exchange contacts between phones.



In Android 3.1 and the later versions, Android contains the native feature of connecting keyboard, mouse, and joystick devices with Android phones via Bluetooth communication. Before Android 3.1, some third-party applications provided a customized way for this purpose.

Android phones have communication support not only for data transfer but also for telephony and messaging. Android contains SMS and MMS for messaging along with threaded text messaging and **Android Cloud to Device Messaging (C2DM)**, and the new **Google Cloud Messaging (GCM)** is also part of Android Push Messaging services.

For telephony, Android supports calls, but it doesn't have native support for video calling (at the time of writing this book); however, some Android devices have customized versions of operating systems that allow developers and users to make video calls either via UMTS networks (as in Samsung Galaxy S) or over IP. Also, Google Hangout, the replacement of Google Talk, is available in Android 2.3.4 and higher versions. This allows users to make video calls using an Internet connection. To use Google Hangout video calling, users need a Google+ account. Skype, a third-party tool of Microsoft Corporation, is also used to make video calls in Android 2.3 and later versions.

Accessibility and multitouch

An Android device runs on a fully touch-based interface and contains few hard- or soft-touch buttons that vary according to the device. Android devices have native support for multi-touch. Multi-touch technology allows developers and users to use single touch, tap, double tap, long touch, pinch-zoom gesture, rotate gestures, swipe gestures in all directions, and much more. Android's latest version (which is Android 4.4 KitKat at the time of writing this book) contains some new touch gestures such as tap and long touch gesture, scroll gesture, and so on. Also, Samsung introduced touchless gestures that make use of their specific APIs called Look API. Through Look API, users can use their phones without touching the screen and moving their hands or head in air, and Android will perform the desired functionality. For example, moving the head up will scroll the page up, and moving the head down will scroll the page down on their phones. Also, many Android device manufacturers, such as Samsung, introduced pen features to allow the users to write on their phones and use them with pens more easily and accurately.



The multi-touch feature was first introduced in the HTC Hero Android phone. Before that, the feature was originally disabled at the Linux kernel level due to Apple's patents on touch-screen technology at that time.

Along with touch, users can access their phones with a voice or speech recognition engine natively introduced in Android phones. Also, Android contains a feature called Talkback that allows people with no or low vision to hear what their Android phone is doing at a particular time. These people can access their phones using voice actions such as calling, texting, navigation, and so on. These voice actions were introduced from Android 2.2 onwards. The ability to control hardware is not yet (at time of writing this book) available through voice actions in Android.



Android 4.1 and the later versions provide enhancement over voice actions to read answers from Google Knowledge Graph when queried with specific commands only.

Extensive content and media support


An Android device is not less than any computer with high-definition media support. Android offers comprehensive APIs for managing images, videos, and audio. The formats supported in Android devices include WebM, H.263, H.264, 3GP, MP4, MPEG-4, AMR, MP3, MIDI, OGG, WAV, JPEG, PNG, GIF, BMP, and WebP. Not only this, Android also provides features for streaming online media using RTP/RTSP protocols, HTML progressive downloads such as the HTML5 `<video>` tag, HTTP dynamic streaming protocol, and the Adobe Flash Streaming (RTMP) protocol provided by Flash plugins.

 New Android devices support 3D-image capturing, and 3D Video Support as their native features.

Along with extensive media support, Android also provides playback features, controls, players, hard buttons for sound control as with other mobile phones, fullscreen playback, and so on.

Android not only has media support but also has content support such as text files, word documents, HTML, and so on. The web browsers available in Android are based on the open source WebKit layout engine that was first developed by Apple Inc. This is coupled with Chrome's V8 JavaScript engine in Android.

While most Android applications are written in Java natively, Android doesn't support Java byte code due to the unavailability of the Java Virtual Machine in Android. This Java code is instead compiled in the Dalvik executable and run on the Dalvik Virtual Machine, a specialized virtual machine for Android systems. The most important thing in Dalvik, which separates it from the Java Virtual Machine, is that it is optimized for a low-battery life with limited memory and CPU.

 Android browser has got a 100/100 score on Acid3 test in Android 4.0 Version. The Acid3 test is a web test page from Web Standards Project that checks a web browser's compliance with elements of various web standards such as Document Object Model (DOM), JavaScript, and so on.

Hardware support

An Android device not only provides features of telephony, such as making phone calls, sending messages, and so on, but it also has lots of features with new hardware components that are used for many different purposes. Android has features of video cameras, touchscreens, Global Positioning System (GPS) for location-based applications, accelerometers, gyroscopes, barometers, magnetometers, proximity sensors, pressure sensors, thermometers, Wi-Fi, Bluetooth, and dedicated gaming controls.

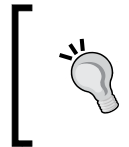


Some new Android phones, such as Samsung Galaxy S4, provide new sensors such as light and color sensors used to capture touchless gestures.

With GPS and location-based technology included in Android phones, Android systems have got native support for Google Maps, Google's GSM cell-based location technology used to determine a device's current position. To make maps more useful for developers and users, Android also provides native APIs for forward and reverse geocoding support that helps to translate coordinates into address and vice-versa.

Background services and multitasking

Due to limited screen dimensions in Android smart phones, only one application becomes visible on the user interface screen. But Android supports applications and services running in the background with its multitasking feature. Using background services, developers can perform automatic processing that doesn't require any user interaction. Some example applications for this feature include generating alerts; monitoring messages, statistics, and weather reports; downloading data from the Internet; or playing audio files in the background.



When an Android device gets low on memory, it stops applications with low priority in the background. Developers should store the necessary data and state of application before going in the background so that on getting stopped, an application can restore its state from the saved one.

Android also supports the notification feature, a standard traditional approach to alert users in their phones. Using Android libraries for notifications, developers can make notification alerts that can be audible, vibration supported, or maybe LED active. In addition to this, Android also allows developers to set notification UI icons, layouts, and so on.

These background applications can be standalone as well as dependent on other applications. Android provides features such as intents and content providers for inter-application communication methods and mechanisms.

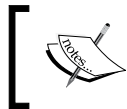
Enhanced home screen

The home screen is like a desktop screen of a computer or laptop. Android users get quick links, app shortcuts, and information on their home screen. Android provides customizable features for the home screen. Widgets, live folders, and live wallpapers make the home screen more interactive and beautiful for users. These apps let Android developers create dynamic application components that provide a window into your applications or offer useful and timely information directly on the home screen. Developers can also provide users with an option to add shortcuts on their home screens. These shortcuts will provide users with the necessary information, and they won't need to open their apps. For example, we have an app that tells us the current time and weather of the day. Now, whenever users want to check the time and weather, they have to open the app. So, instead of creating an app for this purpose, creating a home screen widget would be much better idea. This widget will show the weather and time on the home screen, and users wouldn't have to open the app then.

Other Android features


Android developers can develop applications in multiple languages, offering the local version of the application to the users. Android provides the feature of multilanguage applications.

Also, Android supports tethering that allows users to share the network connection of a device with other mobile phones and computers. This sharing can be achieved via Wi-Fi hot spot or USB tethering.



Tethering was introduced in Android 2.2 Version; so, the earlier versions had tethering support through third-party applications and manufacturers.

Pressing the power and volume-down hard buttons at the same time allows users to capture a screenshot of the device. This feature was first introduced in Android 4.0. The earlier versions are using third-party applications, but these applications need a rooted device as a prerequisite. Developers can also take screenshots using the DDMS tool via a PC connection.

 Rooting any Android device is not allowed, and it breaks all the warranty and guarantee deals and can sometimes be a risky procedure for mobiles.

Android features and intents

Until now, we have discussed the different features commonly found in Android phones and tablets, but we are still unaware of the connection between intents and these features. There are some features that can be used via intents and some cannot. Simply to remind you, intents are asynchronous messages between different applications and the Android system.

In this chapter, we will discuss a few features that can be used through intents, and see how intents perform various actions. We have divided the features into four sections: messaging, telephony, notifications, and alarms. We will develop some examples that will use intents and access these features, and we will discuss how these features are accessed and the role of intents in them.

Before we start discussing these example applications, we are going to discuss some basic terminology used in Android for the clarification of concepts between intents and features. In the next section, we will discuss two different tags, `uses-feature` and `uses-permission`, from the `AndroidManifest` file. These tags are used to declare some permissions and settings for any Android application. Let's see what they are for in the next section.

The `<uses-feature>` and `<uses-permission>` tags

Any Android application, by default, doesn't have the permission to perform any operations that impact any other application, system, or the user directly or indirectly. This includes reading or writing the user's private data such as contacts and messages, reading or writing other applications' files, or any other activity. The Android system allows applications to be standalone and sandboxed, but in case of sharing data, the applications must explicitly share it with each other. To achieve this objective of sharing more easily, Android allows developers to declare permissions in their applications for the activities that the app wants to perform. Users will be informed about the permissions that allow them to install the application on their devices.

Developers need to bear two things in mind regarding permissions: the permissions for device capabilities such as accessing camera or hardware and defining custom permissions. We will be discussing the first option of accessing device features and hardware and granting permissions to the application in this topic. This can be achieved using two tags in the manifest file: the `uses-feature` tag and the `uses-permission` tag.

Firstly, we will talk about the `<uses-feature>` tag. The `<uses-feature>` tag lets the developers declare any single hardware or software feature to be used by the application. This is declared in the `AndroidManifest` file in the `<manifest>` tag of the application, and as the name of tag suggests, this informs the application about the dependent entities to be accessed. The following code snippet shows the general declaration of the `<uses-feature>` tag:

```
<uses-feature
    android:name="string"
    android:required=["true" | "false"]
    android:glEsVersion="integer" />
```

You can see that there are three attributes in the `<uses-feature>` tag: `name`, `required`, and `glEsVersion`. The `android:name` attribute specifies any single hardware or software feature used by the application in the form of a string descriptor. The `android:required` attribute is quite an important attribute in the `<uses-feature>` tag. It is a Boolean value indicating if an application needs the feature that is specified in the `android:name` attribute. If the developer declares `android:required="true"` for any feature, it means that the application won't run without the specified feature available on the device. If the developer declares `android:required="false"` for the feature, it means that the application prefers the feature to be available on the device. If the feature is not available, the application won't work properly or may crash when using the feature due to its unavailability. The default for this attribute is `true`. The final attribute in the `<uses-feature>` tag is `android:glEsVersion`. This is a version number represented in 16 bits. This attribute specifies the OpenGL ES version that the application will use. For example, we are using a camera in our application. The following code snippet shows how to declare the permissions for a camera in the manifest file:

```
<uses-feature
    android:name="android.hardware.camera"
    android:required="true"
    android:glEsVersion="0x00010000" />
```

You can see in the code that we have used the `android.hardware.camera` string for the `android:name` attribute. This string declares the camera feature of Android, and other attributes declare that the application requires the camera feature and supports the OpenGL ES 1.0 Version for it to work properly. The developer must specify each feature used in the application in a separate `<uses-feature>` tag; so, if the application requires multiple features, multiple tags should be declared in the manifest file. It is a good practice to declare all the features used in an application. These declared tags of `<uses-feature>` only provide information, and the Android system doesn't check for matching features before the installation of the application.



Google Play uses the `<uses-feature>` tag declared in the manifest file to filter the application from devices that do not meet its software and hardware requirements.

The `<uses-feature>` tag was first introduced in API Level 4. The earlier versions simply ignore this tag if an application containing the `<uses-feature>` tag is running on lower-version devices.

The following tables show a list of a few feature types and name strings for hardware and software features respectively. They can be used in the `<uses-feature>` tag's `android:name` attribute:

Hardware features

Feature type	Feature descriptor (Android name)	Description
Bluetooth	<code>android.hardware.bluetooth</code>	This feature allows the application to use Bluetooth of the device.
Camera	<code>android.hardware.camera</code>	This feature allows the application to use the camera component of the device.
	<code>android.hardware.camera.flash</code>	This is subfeature that allows the application to use the device's camera's flash.
Location	<code>android.hardware.location.gps</code>	This subfeature allows the application to use the precise location coordinates obtained from the Global Position System (GPS) receiver of the device.

Feature type	Feature descriptor (Android name)	Description
Sensors	<code>android.hardware.sensor.accelerometer</code>	This feature allows the application to use motion reading from the accelerometer sensor of the device.
	<code>android.hardware.sensor.compass</code>	This feature allows the application to use directional readings from a compass of the device.
	<code>android.hardware.sensor.proximity</code>	This feature allows the application to use the proximity sensor of the device.
Screen	<code>android.hardware.screen.landscape</code>	This feature sets the application's screen orientation to landscape.
	<code>android.hardware.screen.portrait</code>	This feature sets the application's screen orientation to portrait.
Touchscreen	<code>android.hardware.touchscreen.multitouch</code>	This subfeature allows the application to use two-point multi-touch capabilities such as Pinch.
Wi-Fi	<code>android.hardware.wifi</code>	This feature allows the application to use the Wi-Fi component of the device.

Software features

Feature Type	Feature Descriptor (Android name)	Description
App Widgets	<code>android.software.app_widgets</code>	The feature allows the application to include app widgets and can be installed on devices having a home screen.
Home Screen	<code>android.software.home_screen</code>	The feature allows the application to behave as a home screen replacement of the device.
Input Method	<code>android.software.input_methods</code>	This feature allows the application to provide custom input methods.
Live Wallpaper	<code>android.software.live_wallpaper</code>	This feature allows the application to provide live wallpapers.

We haven't shown all the features and descriptors in the preceding tables. We have only presented some of the most commonly used features. The table shows the feature type of each feature, its feature name descriptor to be used in the `android:name` tag, and a short description of what the feature will do and how it will affect the application in the device.

Some features are categorized as hardware features and some as software features. Hardware features are the features that use hardware components on the backend. To access these hardware components, our application should have the permission to access the hardware. It should be noted that the `<uses-feature>` tag is just informative, and it only tells the user that the application is using some specific feature in the app. It doesn't allow access to the application for using any specific feature or component.

To allow the application to use any specific component, Android provides another tag, `<uses-permission>`. This tag provides access of a component to the application if the user allows it at the time of installation. The following code snippet shows the syntax for writing the `<uses-permission>` tag in the manifest file:

```
<uses-permission android:name="string" />
```

The `<uses-permission>` tag requests any specific permission that the application must be granted for it to operate properly. Permissions are only granted by the user at the time of the installation of the application. Unlike the `<uses-feature>` tag, the `<uses-permission>` tag only has a single `android:name` attribute. The only attribute of the tag specifies the name of the permission. The name of the permission can be defined using the `<permission>` tag (this is beyond the scope of the book, and we will not discuss it) or using standard permission names provided by the Android system. For example, to allow application to read phone contacts, we can write a code snippet like the following:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
```

You can see how we provided a standard permission name from the `android.permission` package for reading the contacts of the phone.



The features declared through the `<uses-feature>` tag are used by Google Play to filter the application, and the permissions declared through the `<uses-permission>` tag are presented to the user at the time of installation for granting access.

Some of the `<uses-feature>` tag name descriptors were added in the API after the `<uses-permission>` tag descriptors. Due to this, some applications using the `<uses-permission>` tag were able to use specific hardware without the need of declaring the `<uses-feature>` tag in the manifest file. To prevent the applications from any unexpected issues regarding this mismatch, some permissions are implied with some features. Google Play assumes that certain hardware-related permissions indicate that the underlying hardware features are required by default. The `<uses-feature>` tag allows Google Play to filter the applications in the market and show only those applications that the device is capable of running to the user. However, the `<uses-permission>` tag performs its duty when a user downloads the application and installs it. Before installation, the user is asked to grant access of all the permissions specified in the application. The application will only be installed when the user grants access. So, for those features that have both the `<uses-feature>` and `<uses-permission>` tag name descriptors, it is a good practice to declare both in the manifest of the application for it to work properly. The following table shows some of the features that are implied by the permissions:

Category	<code><uses-permission></code> descriptor	<code><uses-feature></code> descriptor
Bluetooth	<code>android.permission.BLUETOOTH</code>	<code>android.hardware.bluetooth</code>
Camera	<code>android.permission.CAMERA</code>	<code>android.hardware.camera</code>
Location	<code>android.permission.ACCESS_COARSE_LOCATION</code>	<code>android.hardware.location</code>
	<code>android.permission.ACCESS_FINE_LOCATION</code>	<code>android.hardware.location.network</code>
		<code>android.hardware.location.gps</code>
Microphone	<code>android.permission.RECORD_AUDIO</code>	<code>android.hardware.location</code> <code>android.hardware.microphone</code>

Category	<uses-permission> descriptor	<uses-feature> descriptor
Telephony	android.permission.CALL_PHONE	android.hardware.telephony
	android.permission.PROCESS_OUTGOING_CALLS	android.hardware.telephony
	android.permission.READ_SMS	android.hardware.telephony
	android.permission.RECIEVE_SMS	android.hardware.telephony
	android.permission.SEND_SMS	android.hardware.telephony
Wi-Fi	android.permission.WRITE_SMS	android.hardware.telephony
	android.permission.ACCESS_WIFI_STATE	android.hardware.wifi

You can see in the table that all the features that are implied by permissions are hardware features and require hardware components to run the application properly. So, it has already been made clear that developer should declare both the <uses-feature> and <uses-permission> tags to filter in Google Play and properly install it on the device without creating any hassle for the user and developer.

Sharing using the SEND action

Any cell phone's primary purpose is to provide an easy way of communication. And like all cell phones, Android smartphones provide an easier way of communication. In this era of the Internet and social networking, Android phones have proved to be quite productive in sharing and social networks. Android provides features such as sharing pictures, status, sending e-mails, social networking such as Facebook, Twitter, and so on. Fortunately for developers, all these sharing features can be used very easily using a few lines of intents. Intent has proved to be a very good way of performing asynchronous communication within Android's components and apps.

In *Chapter 3, Intents and Its Categorization*, we discussed an example of sharing status using intents. We will explain the same `SEND` intent in more detail in this chapter, and see how we can share images and text via any medium on the user's choice. When it comes to sharing anything on Android phones, the intents with the `SEND` action are used a lot. In this section, we will discuss intents with the `SEND` action to see what is possible with it.

To define the intent with the `SEND` action, the following code snippet shows the declaration:

```
Intent send = new Intent(Intent.ACTION_SEND);
```

You can see that we have passed a string constant of `Intent.ACTION_SEND` in the constructor of the intent. This string constant tells the Android system that the intent is meant to send anything on a device. We can execute the following intent by calling the `startActivity()` method as shown in the following code snippet:

```
Intent send = new Intent(Intent.ACTION_SEND);
startActivity(send);
```

Passing the `SEND` intent in the `startActivity()` method will allow the user to choose his favorite way of sending by providing a dialog of all the possible sharing applications. But if we pass the `SEND` intent in the `startActivity()` method without setting the intent type, it will throw a runtime exception. The following log shows some lines of the exception thrown at runtime:

```
AndroidRuntime FATAL EXCEPTION: main
AndroidRuntime java.lang.RuntimeException: Unable to start activity ComponentInfo{
com.gamyguru.shareusingsend/com.gamyguru.shareusingsend.MainActivity}:
android.content.ActivityNotFoundException: No Activity found to
handle Intent { act=android.intent.action.SEND }
AndroidRuntime at android.app.ActivityThread.performLaunchActivity(ActivityThread
.java:2077)
AndroidRuntime at android.app.ActivityThread.handleLaunchActivity(ActivityThread.
java:2104)
AndroidRuntime at android.app.ActivityThread.access$600(ActivityThread.java:134)
AndroidRuntime at android.app.ActivityThread$H.handleMessage(ActivityThread.java:
1247)
AndroidRuntime at android.os.Handler.dispatchMessage(Handler.java:99)
AndroidRuntime at android.os.Looper.loop(Looper.java:154)
AndroidRuntime at android.app.ActivityThread.main(ActivityThread.java:4624)
AndroidRuntime at java.lang.reflect.Method.invokeNative(Native Method)
AndroidRuntime at java.lang.reflect.Method.invoke(Method.java:511)
AndroidRuntime at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(Zygo
teInit.java:809)
AndroidRuntime at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:576)
AndroidRuntime at dalvik.system.NativeStart.main(Native Method)
```

In the log, you can see "Unable to start activity" and then `android.content.ActivityNotFoundException` is thrown. This exception is thrown when a call to the `startActivity(intent)` method or one of its variants fails because an activity cannot be found to execute the given intent. Not only the type of exception, but also the log shows the reason behind the failure of the activity. It says "No activity is found to handle the intent". You might be wondering why Android couldn't find the suitable activity to receive the intent. Recall implicit intents from earlier chapters, Android looks for all the possible activities matching the intent type and shows all those apps in a dialog. In our case, we haven't defined any type for the intent except its `Intent.ACTION_SEND` action; that's why we are getting a runtime exception of `ActivityNotFoundException`. Let's set the type of action and see the dialog that shows all the possible apps to receive the intent:

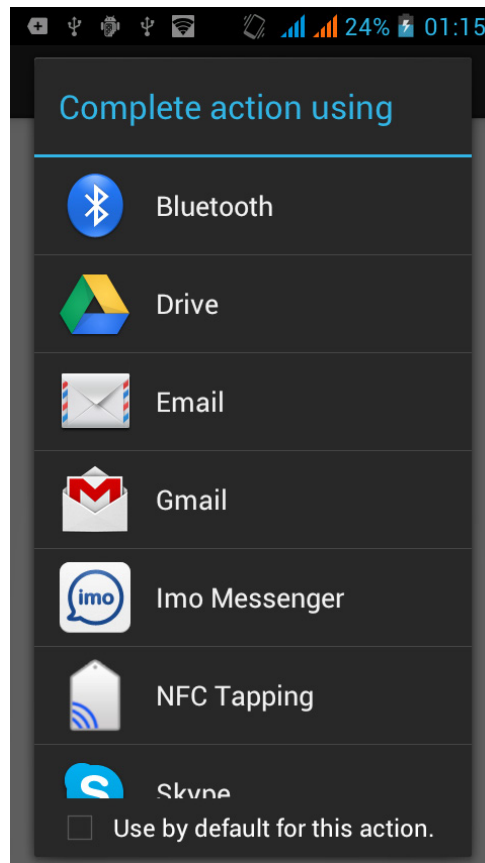
```
Intent send = new Intent(Intent.ACTION_SEND);
send.setType("text/html");
startActivity(send);
```

You can see that we have called the `setType()` method and passed a string of the `text/html` type. This method sets an explicit MIME data type of the intent. This is used to create intents that only specify type and not the data. These are the commonly used implicit intents in Android systems. This method clears any data of the intent that was set previously.



The MIME type matching in the Android framework is case sensitive. So, you should always write your MIME type with lowercase letters. You can also use `normalizeMimeType(String)` method to ensure that it is converted to lowercase.

We have passed `text/html` as the MIME type in method argument. This type tells the Android system that all those applications that support the HTML type of data and process it can receive this intent. So, in a result, Android pushes all those applications in a dialog to let the user choose his/her favorite application. The following image shows the dialog for the `text/html` type:



You can see that all the apps supporting HTML type content are shown in the image, such as **Email**, **Imo Messenger**, and **Skype**. You can see how easy it is to share content using the `SEND` intent in Android phones, and the job of choosing apps and launching them is left to Android.



You may have noticed that there is no SMS/MMS-sending application shown in the dialog because SMS/MMS are just plain text applications and they support only that type of content.

On choosing any option from the list, the app will start. As we haven't set any content to be shared, the application will be mostly empty. To set the content in the intent, we have to use extras. We will put extras for some information such as title, subject, or text. The following code snippet shows how to put some extras in the SEND intent:

```
Intent send = new Intent(Intent.ACTION_SEND);
send.setType("text/html");
send.putExtra(Intent.EXTRA_SUBJECT, "My Subject");
send.putExtra(Intent.EXTRA_TITLE, "My Title");
send.putExtra(Intent.EXTRA_TEXT, "My Text My Text My Text My Text My Text");
startActivity(send);
```

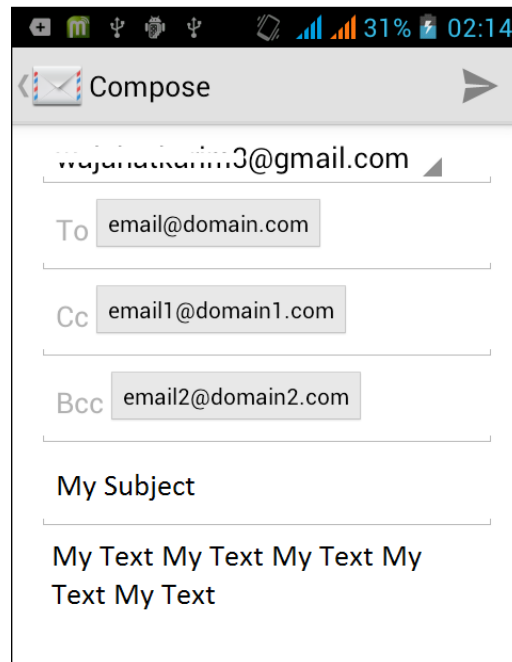
You can see in the code that after setting the MIME type of intent, we have called the `putExtra()` method few times. This method adds extended data to the intent. There are two parameters of the function: name and value. The name parameter must include a package prefix; for example, the app `com.android.contacts` would use names like `com.android.contacts.ShowAll`. We have passed three strings for subject, title, and text content of the intent. The names such as `Intent.EXTRA_SUBJECT`, `Intent.EXTRA_TITLE`, and `Intent.EXTRA_TEXT` for these types of data are already declared in the `Intent` class, and we can access those in a static manner. You might be thinking why we have passed the subject if we have passed the title string as well. Well, the SEND intent is an implicit intent, and Android shows all the apps supporting the intent. The user can choose any app as different apps are interested in different data. For example, any e-mail application will be interested in the Subject, To, and Body strings. And any SMS application will only be interested in the To and Body strings. So, for efficient usage of the SEND intent, you should add all the possible content to share it with every application effectively. Let's take an example of sending an e-mail using the SEND intent. The following code snippet shows how we can use the SEND intent to send an e-mail:

```
Intent send = new Intent(Intent.ACTION_SEND);
send.setType("text/html");
send.putExtra(Intent.EXTRA_SUBJECT, "My Subject");
send.putExtra(Intent.EXTRA_EMAIL, "email@domain.com");
send.putExtra(Intent.EXTRA_CC, "email1@domain.com");
send.putExtra(Intent.EXTRA_BCC, "email2@domain2.com");
send.putExtra(Intent.EXTRA_TEXT, "My Text My Text My Text My Text My Text");
startActivity(send);
```

Firstly, we have declared our SEND intent by passing the `Intent.ACTION_SEND` parameter of constructor. Then, we have set the type of intent by the calling `setType()` method to the "text/html" MIME type. We then add the extra content for the e-mail app as shown in the following list:

- `Intent.EXTRA_SUBJECT`: This name constant is used to add the Subject.
- `Intent.EXTRA_EMAIL`: This name constant is used to fill an e-mail address in the To field.
- `Intent.EXTRA_CC`: This name constant is used to fill the e-mail address in the Cc field.
- `Intent.EXTRA_BCC`: This name constant is used to fill the e-mail address in the Bcc field.

Finally, before calling the `startActivity()` method, we put body of the e-mail by the `Intent.EXTRA_TEXT` name constant and pass our text in the value parameter of the `putExtra()` method. The `startActivity()` method will show the same dialog as shown in the previous image, and on choosing an e-mail application, it will show the following screenshot:



An e-mail application already filled with content put in intent

You can see from the image that all the data we put into extras is already filled in the e-mail application such as subject, e-mail, text etc. Now, all that the users have to do is to tap the **Send** button and the e-mail will be sent. In this example application, we have sent an e-mail to one address directly using the To field and indirectly by Cc and Bcc to two other addresses. Android allows us to add multiple e-mail addresses as well. The name constant `Intent.EXTRA_EMAIL` is used for this purpose. We have passed an address in the code; we can also pass arrays of strings consisting of e-mail addresses to send the e-mails to.

From this section, we have mostly learned about how the `ACTION_SEND` intent is used and how much work we can do with just a few lines of code using this intent. If we choose Facebook, Twitter, or any other application from the dialog, we will see the same result of sharing data via that app. This is the power of using implicit intents to make it general in almost every possible way without doing any hard development work.

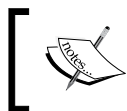


`ACTION_SEND` is an action of the intent. Like this action, there are other actions such as `ACTION_VIEW`, `ACTION_SEARCH` that can be used by passing in intents for other purposes in Android.

Telephony and making calls using intents

Not only Android phones, but any phone's primary purpose from the day of invention is to provide a way to communicate long-distance conversations. And like all other phones, Android phones also provide features for making and receiving calls, checking call logs such as missed calls and dialed numbers, storing contacts, editing/modifying/deleting contacts, and a lot more. As Android phones lie in the frame of smart phones, there is a lot to the call feature. Users can make video calls, record calls, conference calls, mobile to computer calls and vice versa, and much more. All these features provide a very effective product to users and let the developers use these features for more flexibility and productivity.

Android provides many APIs for telephony features for developers. These telephony APIs let your applications and developers access the underlying telephony hardware, thus making it possible to create custom dialers, integrate call handling or phone state monitoring, and so on.



Developers cannot customize the in-call screen of the phone due to security reasons. The in-call screen is shown when users make any calls or receive any incoming calls.

As this book is focused on intents, we will only discuss those telephony features that can be utilized using intents. From many features like making calls, receiving calls, checking the call log, accepting/rejecting calls, and so on, there are very few that can be utilized directly and only using intents. Fortunately, making calls is one of them. Let's discuss how we can make calls using intents in the next section.

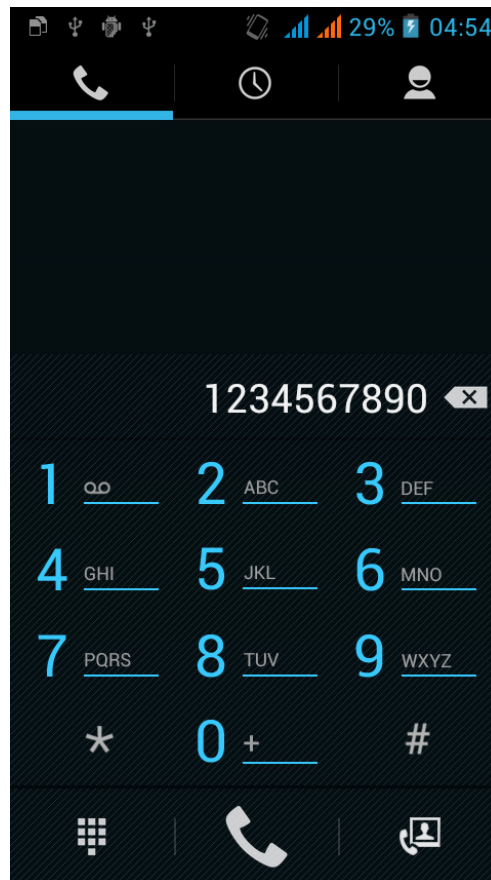
Making phone calls using intents

There are two methods of making phone calls in Android. Developers can either use the APIs provided by Android to make phone calls, or they can only initiate phone calls by sending the intent with the necessary information such as the phone number. We will explore the method of initiating phone calls later in this section.

In the preceding section, we saw how we can use actions in intents to tell the Android system about our intentions. We will be doing the same to make phone calls by telling Android about our intentions and the rest of the work is left to the system. The following code snippet allows the application to launch the dialer with the specified number already dialed, and the user can explicitly make a call by pressing the call button in it:

```
String phoneNumber = "tel:" + "1234567890";
Uri phoneNumUri = Uri.parse(phoneNumber);
Intent dialIntent = new Intent (Intent.ACTION_DIAL, phoneNumUri);
startActivity(dialIntent);
```

You can see that we have done very few changes in the code. We have declared a `phoneNumber` string that stores the number we want to dial. You might be wondering why we have concatenated a `tel:` prefix in the string. Well, that prefix is used in getting the **Universal Resource Identifier (URI)**, of the number. We get this URI by calling the static method `Uri.parse()` of the `Uri` class. This method returns the URI, which we pass in turn in the constructor's other parameter. We provide the `DIAL` action by passing `Intent.ACTION_DIAL` in the declaration of the intent, and finally, we call the `startActivity(intent)` method as always to execute the intent and tell the Android system to process our intentions. The following screenshot shows a dialer result of the previously mentioned code snippet:



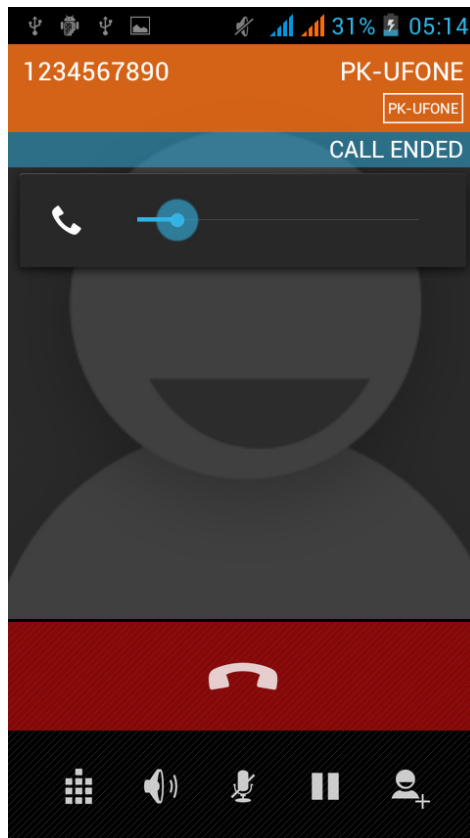
A dialler screen with a dialled number initiated by starting the DIAL intent

When we run the previous code, the application will start the default dialer of the Android phone and will dial the number provided in code in it. It will not call the number; it will just dial the number because we used `Intent.ACTION_DIAL`. The user can explicitly press the call button of the dialer and make a call.

If the user doesn't want to dial the number, it is also possible to directly call the number without going to the dialer first. Android provides the `Intent.ACTION_CALL` action for this purpose. The following code snippet shows how to make calls directly:

```
String phoneNumber = "tel:" + "1234567890";
Uri phoneNumUri = Uri.parse(phoneNumber);
Intent callIntent = new Intent (Intent.ACTION_CALL, phoneNumUri);
startActivity(callIntent);
```

You can see from the code that everything is the same except the action passed in the constructor of the intent. In the last example, we passed `Intent.ACTION_DIAL` and in this example we have passed `Intent.ACTION_CALL` to directly make the call. When we run this code snippet, the application will start making a call on an Android phone. The following screenshot shows the call:

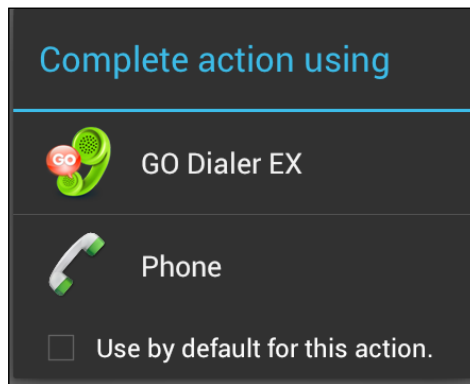


An in-call screen shown by starting the CALL intent

This action of `ACTION_CALL` to directly make a phone call requires the user to grant permission to the application. The following code snippet shows the permission to be placed in the `AndroidManifest` file to enable the app to work perfectly:


```
<uses-permission android:name="android.permission.CALL_PHONE"/>
```

It should be noted that `ACTION_CALL` cannot make calls to emergency numbers using intents; however, using `ACTION_DIAL` it is possible to dial emergency numbers. If the user has multiple dialers installed on the phone, the `ACTION_DIAL` action will present the list of dialers from which the users can choose a favorite dialer. The following screenshot shows the scenario of multiple dialers:



Multiple dialers to choose from the dialog

There is very little difference between the `ACTION_DIAL` and `ACTION_CALL` intents. The `ACTION_DIAL` intent only dials the number, and the user can explicitly call by pressing the call button, but `ACTION_CALL` directly makes the call without showing the dialer to the user.

[ There can be restrictions on applications on making phone calls directly. So, it is a good practice to use `ACTION_DIAL` in the apps unless `ACTION_CALL` is required.]

This is how we use intents to easily make phone calls and use the telephony features of Android. In the next section, we will see how we can send SMS, MMS, and data messages using intents. Along with sending, we can also confirm the message delivery as well as receive messages. Let's now discuss these in detail in the next section.

SMS/MMS using intents

In addition to the features of making calls, mobile phones support messaging services such as Short Messaging Services (SMS), Multimedia Messaging Services (MMS), and lately the data messages. The SMS/MMS features are most widely used in phones, and many people prefer it over making calls. Android provides APIs and framework that let developers send and receive messages from within their applications. Developers can even replace the native SMS application to send and receive text messages.



At the time of writing this book, there is no API or library for sending MMS messages from within your applications, but you can send them using the `ACTION_SEND` or `ACTION_SENDTO` intents.

This section will walk you through the various actions such as sending SMS, sending MMS, sending data messages, confirming message delivery, and receiving SMS using intents. We will then brief you about how all these actions are performed without using intents and how APIs of Android can be beneficial to us. Let's look at our first task of sending SMS messages using intents in the next subsection.

Sending SMS using intents

The best thing about using intents is that it passes the responsibility of our requirements to the Android system rather than we creating the full functionality from the core. If we use intents in our current case, which is to send an SMS to someone, we just have to provide the number to send the message to and the message to be sent. The rest is done by Android itself.

We have already had plenty of discussions over the same topic of sending something or sharing something using intents, and fortunately, there is nothing different that we have to absorb here. It's the same old method of creating an `ACTION_SEND` intent and executing it by calling the `startActivity(intent)` method. The following code snippet shows the `ACTION_SEND` intent example that we used previously:

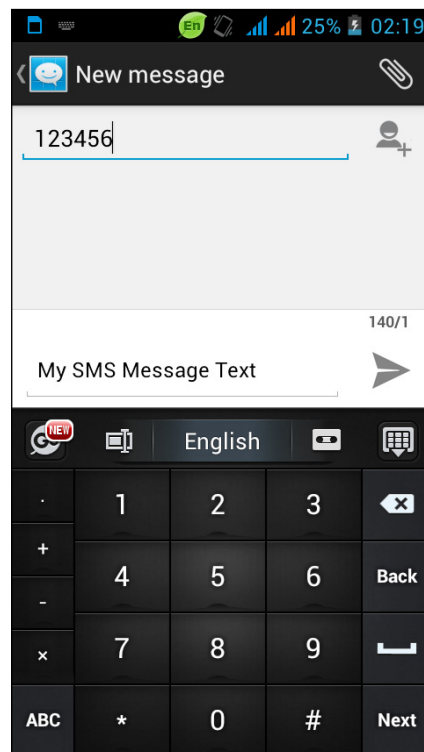
```
Intent send = new Intent(Intent.ACTION_SEND);
send.putExtra(Intent.EXTRA_TEXT, "My Text My Text My Text My Text");
startActivity(send);
```

Now, if we use this code, it is not useful to us because it doesn't perform our action of sending an SMS. Neither does it show the SMS-supporting applications in the chooser dialog, nor does it send any SMS with the data passed in the `EXTRA_TEXT` extra. To make use of `ACTION_SEND` for the purpose of sending an SMS, we have to take care of some extra things. There are two ways of sending an SMS using intents: by the `ACTION_SEND` intent and by the `ACTION_SENDTO` intent. Let's see how we can send an SMS using the `ACTION_SEND` intent.

We have to create intent with the `ACTION_SEND` action and then put an extra `"sms_body"` with the message embedded in it. Android will ask the user for the phone number of the recipient itself. But it still won't show any SMS support applications in the chooser-list dialog because we are still missing the type of intent. As SMSes are short text messages, we should set the intent type to `"text/html"`, but most SMS applications look for `"image/jpg"` or `"image/png"` as the intent type due to no native support for MMS messages. So, after setting the intent type to `"image/png"`, we will have the following code snippet:

```
Intent smsIntent = new Intent(Intent.ACTION_SEND);  
smsIntent.putExtra("sms_body", "My SMS Message Text");  
smsIntent.setType("image/png");  
startActivity(smsIntent);
```

When we execute this code, we will see the chooser dialog of various apps including SMS support applications, e-mail applications, and so on. When we select any SMS application, we will see something similar to following image:



Default SMS application shown after sending the SMS intent

You may have already noticed that the text part of the SMS application is already filled in with the content we added in the "sms_body" extra, and the user is typing the number of recipients of the message.



The previous image shows a default SMS application of the QMobile Noir A10 smartphone. Your device will show the SMS application that you have set as default on your phone, and it won't be the same as this application for sure.

This is how we can send an SMS using intents. Now, let's take our other case in which we want to set the number of recipients using coding. For that purpose, we have to use the ACTION_SENDTO intent instead of the ACTION_SEND intent. The following code snippet shows the use of the ACTION_SENDTO intent:

```
String phoneNumber = "sms:" + "1234567890";
Uri phoneNumUri = Uri.parse(phoneNumber);
Intent smsIntent = new Intent(Intent.ACTION_SENDTO, phoneNumUri);
smsIntent.putExtra("sms_body", "My SMS Message Text");
startActivity(smsIntent);
```

In the preceding code, you can see that we have made a few changes in the code that we discussed previously before sending the SMS messages. We have set the action to ACTION_SENDTO instead of ACTION_SEND. Also, we have passed another argument of the phone number URI in the constructor of the intent. We have created a string for the phone number and concatenated the "sms:" tag before the number. This tag lets the Uri class understand that the string is representing the phone number to send the message to and parse it accordingly. You may remember from the previous section, we used the "tel" tag for making calls to any number using intents. When you execute the code, it will ask the application to choose SMS. On selecting any SMS supported application, it will send the SMS directly to the phone number provided instead of asking the phone number as in the previous example. You may have noticed that we haven't set the type of intent in this code snippet. It is because when we are using the ACTION_SENDTO intent, we don't have to explicitly set the type of intent. Android will understand what the developer is trying to do from the tags such as "sms" or "tel" and from actions such as ACTION_SENDTO or ACTION_CALL.



If you want to use ACTION_SEND and set the recipient number explicitly using code, Android provides the "address" extra to put the string of the number in it without having to use any tags such as "tel" or "sms".

Until now, we have talked about using `ACTION_SEND` and `ACTION_SENDTO` to send SMS text messages. In the next section, we will see how we can send multimedia messages with pictures embedded in them using intents.

Sending MMS using intents

The only thing that differs in a text message and multimedia message is the rich media embedded in it. MMS messages contain rich media content such as photos, videos, and cards, and some text as message for the content. Currently, there is no library provided by Android that lets developers send MMS natively, unlike SMS. But fortunately, intents make a clear way out for us in order to send an MMS. As the real difference defines, we have to add some media in the text message intent with its type set to multimedia, such as `"image/png"`, and we have then finished sending MMS messages. The following code snippet shows how to send any MMS message using intents used for SMS messages:

```
// My Media Uri (any image stored in Gallery or other media)
Uri myMediaUri = Uri.parse("content://media/external/images/myMedia");

// Our MMS Intent
Intent mmsIntent = new Intent(Intent.ACTION_SEND);
mmsIntent.putExtra("sms_body", "My MMS Message Text");
mmsIntent.setType("image/png");
mmsIntent.putExtra(Intent.EXTRA_STREAM, myMediaUri);
startActivity(mmsIntent);
```

You can see that there are two parts of the code. In the first part, we are getting the URI of our required image stored in external storage in the `images` folder. In the second part, we are creating intent with `ACTION_SEND`. Then, we add our text by using an `"sms_body"` extra and set the type to `"image/png"` to make it meaningful for a multimedia message. After that, we attach our media using the `Intent.EXTRA_STREAM` extra and pass our image URI as value in it. Finally, we execute the intent by calling the `startActivity(intent)` method. The only difference was to attach the media URI using the `EXTRA_STREAM` extra, and the rest was the same as in the SMS messages. You should also note that we can use `ACTION_SENDTO` to specify the recipient number, or we can also add the `"address"` extra with the value of the phone number.



We have set the type of `"image/png"` in the previous example. This can only send PNG images. For other image formats, we can specify `"images/*"`.

Until now, we have only discussed sending SMS and MMS messages. But are we sure that those messages have been delivered successfully? Well, the next section is about confirming message delivery and understanding the role of intents in it. Let's see how we can confirm the message delivery using intents in the following section.

Confirming message delivery using intents

When we use intents to send messages, whether they are SMS or MMS, we just can't track those messages for actions, such as, confirming delivery. The reason behind this is the implicit use of intents and relying on the default action of the Android system. If we use intents to send messages, it means that we are passing our responsibility of sending messages to the Android system. Now, if we want to confirm the delivery status of the message, it means that we are asking Android about our message. Unfortunately, we lack two things to make it possible: one is to tell the Android system about our confirmation of some message and the other is that the Android system may not remember what message we are talking about.

In order to make it possible to confirm the delivery, we have to use the native API for sending messages manually. It is the job of this API to keep track of both the delivery status and the message we are talking about. Also, using this API, we can easily send our query, asking the Android system about the delivery confirmation.

Now, if we are using native APIs for sending messages, we have to think about MMS messages. As mentioned earlier, there is still no native support for MMS messages; so, we won't be able to track and confirm the delivery of MMS messages, but yes, we can confirm the delivery status of SMS messages. In this section, we will talk about how we can check the SMS delivery status using the native SMS API and how intents are used to achieve our goal.

Intents are an asynchronous way of communication in Android, and they are used everywhere. The only change is that they are used to achieve our goals and finish the requirements. In a short explanation about confirming the message delivery status, we will use the native SMS API called `SmsManager` to send the text message using the `SmsManager.sendMessage()` method. But to keep track of the message, we will use two intents: one for the sent action and one for the delivery action. Along with these two intents, we will also create two pending intents: one for the sent action and one for the delivery action. Finally, to put all four intents in action, we will create two broadcast receivers: one to check the sent action and the other to check the delivery action. It may seem quite complex here, but it is as easy as a charm. Let's have a look at the code snippet that declares our four intents: two intents and two pending intents:

```

// Sent Intent
Intent sentIntent = new Intent("sent_sms_action");
PendingIntent sentPI = PendingIntent.getBroadcast(
    getApplicationContext(), 0, sentIntent, 0);

//Delivery Intent
Intent deliveryIntent = new Intent("delivered_sms_action");
PendingIntent deliverPI = PendingIntent.getBroadcast(
    getApplicationContext(), 0, deliveryIntent, 0);

```

You can see that we have declared our intents in the usual way in the code. The only difference here is that we have used our own custom actions represented in strings such as "sent_sms_action" or "delivered_sms_action". Then, we have created two pending intents using the `getBroadcast()` factory method of the `PendingIntent` class. The `getBroadcast()` method will retrieve `PendingIntent` that will perform any broadcast, such as calling the `Context.sendBroadcast()` method.

So, after creating all four intents, we will now have to create and register the broadcast receivers that will put the pending intents in action. The following code snippet shows both receivers being implemented:

```

registerReceiver(new BroadcastReceiver()
{
    @Override
    public void onReceive(Context _context, Intent _intent)
    {
        switch (getResultCode())
        {
            case Activity.RESULT_OK:
                // Message is sent successfully, do your work here
                break;
        }
    }
}, new IntentFilter("sent_sms_action")
);

registerReceiver(new BroadcastReceiver()
{
    @Override
    public void onReceive(Context _context, Intent _intent)
    {
        // Message is delivered successfully. Do your work here
    }
}, new IntentFilter("delivered_sms_action")
);

```

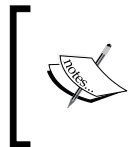
As seen in the previous code, we have registered two broadcast receivers using the `Activity.registerReceiver()` method and passed anonymous objects. The overridden method `onReceive()` serves our purpose. One `onReceive()` method is called when any message is sent, and the other `onReceive()` method is called when any message is delivered. We have put comments to show you where you can use your custom functionality in the code. You might be wondering how Android will know that these are the broadcast receivers for sent and delivery status. Android will know about it by checking the intent filters. You can see that we have passed our custom actions passed in intents in the constructors of intent filters, and these filters will filter the broadcasts, and the receiver will only receive those broadcasts for which it was registered. We have done our core work for confirming the message delivery until now. All that's left now is to put it in action, and here, the `SmsManager` API comes handy. We will create an instance of `SmsManager` and call its `sendTextMessage()` method to send the message and put all the intents in it, and then we are done. The following code snippet shows the `SmsManager` usage code:

```
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage(sendTo, null, myMessage, sentPI, deliverPI);
```

Remember, the `SmsManager` API uses the `android.permission.SEND_SMS` permission; so, don't forget to add it in your manifest file, as shown in the following code snippet:

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

So, this is how we can confirm the message delivery. We can only confirm the delivery status of text messages, and we have to ask the user to grant a `SEND_SMS` permission for the purpose. But, if we are using intents, we can only send messages and we won't be requiring any permission from the users.



Android emulator supports sending and receiving SMS messages. This can be accomplished by creating multiple instances of emulators and sending text messages to port the number of emulators.

Summarizing the role of intents in confirming the message delivery, intents are not performing the core action of confirming message delivery here. They are just providing a way of communication by carrying the necessary information such as which message's delivery is to be checked and so on. Then, these intents are used by broadcast receivers that constantly check for the delivery and sent status. Once it is done, they pass the status in our intents and then those intents provide us with an update of whether the message has been sent or delivered or not.

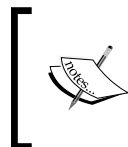
In the next section, we will be doing almost the same kind of stuff and coding, but this time, we will do it to receive messages. After using all these code snippets, we can develop our SMS application that can send and receive messages. Let's see how we can receive messages and the role of intents behind it.

Receiving SMS messages using intents

Until now, we have talked about sending SMS/MMS messages and the importance of intents in these applications. In this section, we will talk about how we can listen for incoming messages so that we can use them in our applications. Using this feature, we can develop messaging applications. Intents can send messages using the `ACTION_SEND` or `ACTION_SENDTO` intents directly, but these don't play a direct role in listening for incoming messages and receiving messages. Intents are used in the same way as `Broadcast Receiver`, and are used to get the data such as sender number, message, message time, and so on. Before we discuss how to listen for incoming messages, we have to learn about some classes that are used in the following application.

The SmsManager class

We have already used the `SmsManager` class in the previous subsections for confirming message delivery. This class is used to manage SMS operations such as sending data, SMS, and PDU messages. We can't instantiate this object using a constructor; we can get its instance by calling the static method of `SmsManager`. `getDefault()`. We can use this class to send messages.



There are two different classes with the name `SmsManager`: `android.telephony.SmsManager` and `android.telephony.gsm.SmsManager`. The later class in the GSM package is deprecated in API Level 4 and later versions.

The SmsMessage object

This class represents a simple SMS message object. On receiving the incoming messages, we will get an array of `SmsMessage` objects. This class is used to get information such as the message body, message time, and sender number.

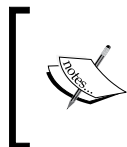
Protocol Data Unit (PDU)

A PDU is the industry format for an SMS message. Developers shouldn't worry about reading a PDU in detail or understanding the format, because the `SmsManager` class of Android reads and writes PDUs and provides methods for the developer to use PDUs.

These classes and concepts will be used in receiving the incoming messages' app. Now, let's discuss how messages are received in Android. When any new SMS message is received by any device, a new broadcast intent is fired. The action of this intent is `android.provider.telephony.SMS_RECEIVED`. We have to create a custom broadcast receiver that will look for this broadcast intent. Whenever we get any message, the `onReceive()` method of the broadcast receiver will be called. The following code snippet shows the implementation of our custom broadcast receiver for incoming messages:

```
public class IncomingMsgReceiver extends BroadcastReceiver {
    private static final String SMS_RECEIVED = "android.provider.
        Telephony.SMS_RECEIVED";
    public void onReceive(Context _context, Intent _intent) {
        if (_intent.getAction().equals(SMS_RECEIVED)) {
            // SMS Received. Write your code here.
            Bundle msgBundle = _intent.getExtras();
            getMessageData(msgBundle);
        }
    }
}
```

As always, we have extended our class from `BroadcastReceiver` and overridden the `onReceive()` method. This method is called when any incoming message is received by the device. We first check whether this intent contains any received messages or not. If the intent action is the same as our `SMS_RECEIVED` string literal, this means that we have received our message.



The SMS received action `android.provider.Telephony.SMS_RECEIVED` is unsupported in Android and is subject to change in any future platform releases. The developer should be cautious when using these unsupported hidden methods and attributes of Android.

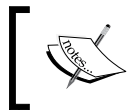
Once the action is verified after checking and comparing, we have to get the message data from the intent and perform our custom actions of the application. We first get the extras bundle from the intent by calling the `getExtras()` method and then we have passed that bundle in our method called `getMessageData()`. This is our custom method, and in this method, we will see how we can get the message data from the bundle. The following code implementation shows the method definition:

```

public void getMessageData(Bundle msgBundle)
{
    if (bundle != null) {
        Object[] pdus = (Object[]) bundle.get("pdus");
        SmsMessage[] messages = new SmsMessage[pdus.length];
        for (int i = 0; i < pdus.length; i++)
            messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
        for (SmsMessage message : messages) {
            String msg = message.getMessageBody();
            String to = message.getOriginatingAddress();
            String time = message.getTimestampMillis();
        }
    }
}

```

We first checked that our bundle is not a null object. Then we extracted the PDUs from the bundle by calling the `get ()` method and passing the "pdus" key.



If you don't know which key to pass in the `get ()` method, you can call the `Set<String> Bundle.keySet ()` method to get all the keys used in the bundle.

Recalling PDUs, PDU is the industry format for an SMS message. Once we have all the PDU objects in an array, we create an SMS message from those PDUs using the `SmsMessage.createFromPdu ()` method. After creating all the messages, we are traversing through the array and getting the message data such as the message body text, message sender number, and message time from it using the `SmsMessage.getMessageBody ()`, `SmsMessage.getOriginatingAddress ()`, and `SmsMessage.getTimestampMillis ()` methods. Now, we can use these data strings in our applications. It must be noted that any large message is broken into many small messages, which is why we are getting an array of objects.

This broadcast won't work until we register it in our application. To register it in our application, we have to write the following code in our main activity:

```

final String SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";
IntentFilter filter = new IntentFilter(SMS_RECEIVED);
BroadcastReceiver receiver = new IncomingMsgReceiver();
registerReceiver(receiver, filter);

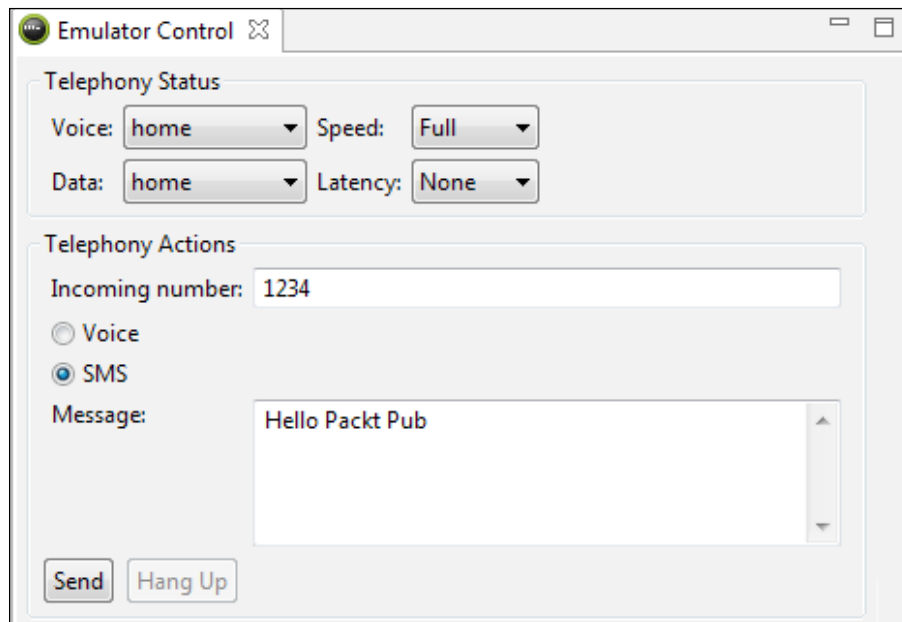
```

There is nothing new to discuss here. We are creating an intent filter with the `SMS_RECEIVED` action and an instance of our broadcast receiver. Then, we are passing both in the `registerReceiver()` method of our activity. The message receiver requires the `android.permission.RECEIVE_SMS` permission; so, don't forget to add this line in your manifest file:

```
<uses-permission
    android:name="android.permission.RECEIVE_SMS"
/>
```

This is how we can receive the incoming messages in our application and use them in many different ways. You might be wondering about the role of intents in this application. As mentioned earlier, intents are not used in this application directly. When any message is received by the device, a broadcast intent is fired. We are using that intent to extract data and messages from it, and those messages are used in our application. Intents play the role of providing the data about messages, after receiving them, in Android devices.

We can use Android debug tools of the **Dalvik Debug Monitor Server (DDMS)** panel to simulate incoming messages on our Android emulators. The following screenshot shows the **Emulator Control** panel in the DDMS view for simulating messages:

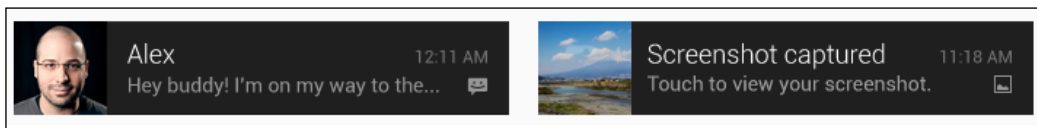


Emulator Control panel in the DDMS view for simulating messages

In this section, we learned about sending SMS, MMS, confirming message delivery, and receiving incoming messages. We also discussed the importance and use of intents in all these applications. In the next section, we will learn about notifications and how intents are used in making interactive notifications.

Notification using intents

From traditional phones to smart phones, every mobile phone uses some method to notify and alert the users about some event such as receiving messages or calls. Like these phones, an Android phone uses a notification system to alert the users. A notification is a message displayed out of the application's normal UI. When any new notification is triggered, it is shown in the notification area. The users can see notifications from the notification drawer and notification area at any time by pulling the drawer downward using the down gesture. The following screenshot shows two different examples of notifications in Android:



Notifications in Android phones

Notifications are like channels that alert the users about important events as they occur when the user is busy in some other mobile activity such as playing a game.

For any developer, a notification is a user interface (UI) element that the developer displays outside of the app's normal UI to indicate and notify the user that an event has occurred. Then, users can choose to view the notification while using other apps and respond to them when they wish. Using a notification is the preferred way for invisible application components, such as broadcast receivers and services, to alert the user about the occurrence of any event.

In this section, we will discuss notifications, their layouts, displaying additional information in notification layouts, and launching intents. We will learn the role of intents and create an example application with a custom notification layout and how intents are important in these types of applications. Before we start developing our example application, let's discuss some basic concepts used in notifications.

Notification forms

Notifications can take different forms like any persistent icon that goes into the status bar and can be accessed through the launcher. When this notification is selected by the user, any specified intent is triggered when some activity or service occurs. Notifications can also be used to turn on the flashing LEDs of the device. Also, devices can vibrate or play ringtones on receiving notifications.

The NotificationManager class

The `NotificationManager` class represents a system service that is used to handle the notifications' system in Android. We can't instantiate this class, but we can get its instance object by calling the `getSystemService()` method and passing `Context.NOTIFICATION_SERVICE`. The following code snippet shows how to get an instance of the `NotificationManager` class:

```
String service = Context.NOTIFICATION_SERVICE;
NotificationManager manager;
manager = (NotificationManager) getSystemService(service);
```

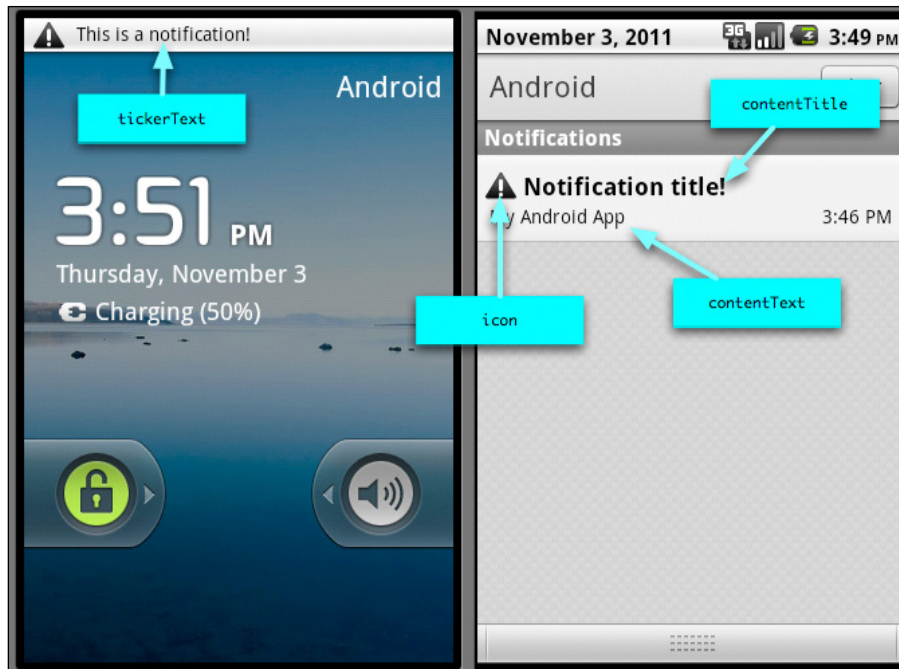
The Notification class

The `Notification` class represents any notification in Android. It provides APIs that allow developers to set the icon, title, time of notifications, and so on. The following code snippet shows us how to create a notification in Android:

```
Notification notf = new Notification(
    R.drawable.ic_launcher,
    "Hello Notification",
    System.currentTimeMillis()
);
```

The Notification layout

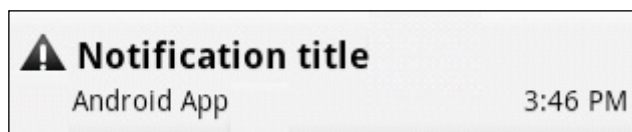
Each notification has an icon and ticker text, which is sometimes called status text. An icon is displayed when a notification has been launched and the notification drawer is closed. The ticker text scrolls along the status bar when a notification is fired and then it is set to the notification message text when the notification drawer is opened. The following screenshot gives an overview of the different aspects of a notification area:



Notification and notification area

You can see from the preceding screenshot that when any notification is fired, its ticker text is scrolled through the status bar. After scrolling through the entire text, its icon is displayed on the status bar. When a user opens the notification drawer by pulling it down, the notification's big icon along with the notification title, content text, and timestamp is shown. This is how any notification is fired in Android.

Now, we will discuss how notifications are triggered and how intents are used in notification applications. We will create a notification, which is shown in the following screenshot:



A simple notification

So, before moving forward to create a notification, we need the layout for our notification. The following code implementation shows our layout for the notification:

```
<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:background="#F6F3F6"
    tools:context=".MainActivity" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_margin="5dp"
        android:src="@drawable/ic_launcher" />

    <TextView
        android:id="@+id/title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_toRightOf="@+id/icon"
        android:text="Notification Title"
        android:layout_margin="3dp"
        android:textStyle="bold"
        android:textAppearance="?android:attr/textAppearanceLarge" />

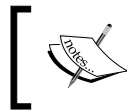
    <TextView
        android:id="@+id/details"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@+id/icon"
        android:layout_below="@+id/title"
        android:layout_margin="1dp"
        android:text="Android App Notification" />

    <TextView
        android:id="@+id/time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/title"
        android:text="3:46 PM"
        android:layout_margin="4dp"
        android:textAppearance="?android:attr/textAppearanceSmall" />

</RelativeLayout>
```

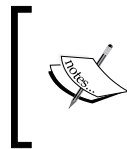
We have placed four views in `RelativeLayout`: an `ImageView` for the icon, a large `TextView` for the title, and two small `TextViews` for description and timestamp respectively. We have used the alignment of `RelativeLayouts` to place the views below, above, to the right of, and to the left of other views so that it can be displayed in the same way on every resolution of different smartphones. We have saved this file as `notification_layout.xml` in the layout folder of the resources directory. This was our layout for the notification. Now, let's learn how to create any notification that will use this layout.

To create a notification with custom layouts, we have two different methods in Android. The first method is to use the `setLatestEventInfo` method to update the details displayed in the standard extended status-notification display. This method is the easiest method and is used in more applications. The other method is to set the `contentView` and `contentIntent` properties of the notification to assign the custom UI layout for the extended display status using the `RemoteView` class.



`RemoteView` is a mechanism that allows developers to embed and control a layout embedded within any separate application. This is most commonly used in creating home screen widgets.

We will be using a difficult method in this section to create the notification as this method uses intents in its code. We will first create a `RemoteView` object and assign it to the `contentView` property of the notification object. The `contentView` `View` represents the notification in the expanded status bar. Notifications often represent a request for action, and this action is performed when a user clicks on the notification in the notification drawer area or expanded status bar. We can specify `PendingIntent` that will be fired when the user clicks on the notification item. Mostly, this intent opens our application and provides more information about our notifications. Along with setting `contentView`, we also need to set `contentIntent` to our created object of `PendingIntent` in which a custom content view is assigned to our notification. The `contentIntent` intent is the intent that must be executed when the expanded status entry is clicked on. If this is the intent of the activity, we must include `FLAG_ACTIVITY_NEW_TASK` that will start our activity in a new task.



When you manually set the `contentView` property, you must also set the `contentIntent` property; otherwise, an exception will be thrown when a notification is triggered causing any runtime crash of your application.

Once the `contentView` property is set to our custom remote view, we can't set our required views in a normal way. We have to use the set methods on the `RemoteView` object that modifies each of the views used in the layout defined. This is how any notification with a custom layout is developed. The following code shows the implementation of the notification with custom layout, and this can be added in any activity:

```
// Creating Notification Object
Notification notification = new Notification(
    R.drawable.ic_launcher,
    "My Ticker Text",
    System.currentTimeMillis()
);

// Creating Intent then PendingIntent Objects
Intent intent = new Intent(this, MyActivity.class);
PendingIntent.getActivity(this, 0, intent, 0);

// Setting contentView of Notification
notification.contentView = new RemoteViews(
    this.getPackageName(),
    R.layout.notification_layout
);

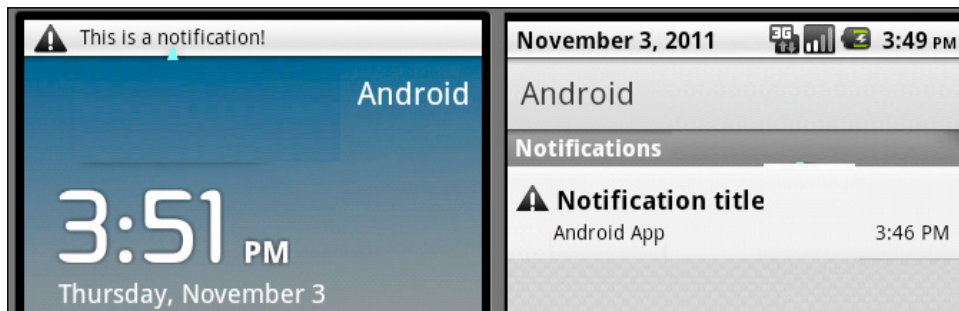
// Setting contentIntent of Notification
notification.contentIntent = pendingIntent;

// Setting values in view objects of layout
notification.contentView.setImageViewResource(
    R.id.icon, R.drawable.ic_launcher);
notification.contentView.setTextViewText(
    R.id.title, "New Notification Title");
notification.contentView.setTextViewText(
    R.id.details, "New Notification Details");
```

You can see from the code that we have first created an object of `Notification` with the initial icon, ticker text, and time of triggering the notification. Then, we create intent objects, `Intent` and `PendingIntent`, for specifying the action of our notification when it is clicked on. Then, we set `contentIntent` and `contentView` of the notification object. We create a new `RemoteView` object for `contentView` and pass our `notification_layout.xml` reference in it. This is how the notification layout is set to our custom layout passed in a `RemoteView` constructor. Then, we set our pending intent to `contentIntent`. And finally, we update the values of our layout using the set methods such as `setImageViewResource()` and `setTextViewText()`. Until now, we have developed our notification with a custom layout. Now, we will see how to trigger the notification. The following code snippet shows how to trigger the notification:

```
String service = Context.NOTIFICATION_SERVICE;
NotificationManager manager;
manager = (NotificationManager) getSystemService (service) ;
manager.notify(1, notification);
```

We are getting an instance of the `NotificationManager` class by calling the `getSystemService()` method. To trigger the notification, we are calling the `NotificationManager.notify()` method that receives two parameters: the first is the ID of the notification and the second is the notification object itself. The following screenshot shows an output of the application:



Notification fired from our application

So far, we have seen how to create notifications and set custom layouts for their view. You might be thinking about the importance and use of intents in this application. In this application, intent was used only for one purpose and that is to navigate the user to our required application or activity when the user clicks on the notification. We created an `Intent` object, and from that, we created a `PendingIntent` object that was used in the notification as `contentIntent`.

Summary

In this chapter, we discussed Android features. We learned about common Android features such as layouts, display, connectivity, communication, accessibility, touch, and hardware support and their comparison with Android mobile components. We then saw how the two most important tags, `<uses-feature>` and `<uses-permission>`, are used in the `AndroidManifest` file and for what purpose.


We also discussed the relation between hardware and software features and Android mobile components and their relationship with these manifest tags. Then, we saw the most common intent action `ACTION_SEND` that is used to send or share anything with other applications using the implicit intents' approach. Then, we expanded our knowledge of intents to more specific features of phones including making calls, sending SMS/MMS messages, confirming the delivery of messages, and receiving messages. We used intents as well as native Android APIs to perform these actions. We then discussed notifications and alerts, and learned how we can set custom layouts in notifications. We learned two different ways, and used one way in our example application. We learned how intents are used in these types of applications and also learned about their role with those classes.

In the next chapter, we will discuss intent filters and see how Android recognizes different intents and filters them according to the calls and applications.

7

Intent Filters

Intent filters are the advanced step to understand the minor significant details of Android Intents. In this chapter, we will take a look at the basics of intent filters and how they can be used effectively in an Android application. The chapter also deals with various kinds of tests that an intent should pass before it is delivered to the desired component.

 Intent filters can be found inside the `AndroidManifest.xml` file, under the `activity` tag.

In this chapter, we will cover the following topics:

- Intent object and its categorization
- Understanding what intent filters are
- Understanding what intent tests are
- Implementing an intent filter for a particular task

Intent object and its categorization

Intent objects come with a whole lot of information. This bundle of information will help the component to extract knowledge from it. For example, what kind of action should be taken on the data that is coming with the intent object; similarly, there is the information that is about the Android system. This information about the Android system is required when the system doesn't know the component that will handle the upcoming intent.

In order to have a better understanding about the example mentioned in the preceding paragraph, consider a scenario in which the intent is transferred in order to start a movie. In this case, the operating system must know which software is needed to perform this action.

The categories contained in the Android intent object are discussed in the following sections.

Component name

The intent object contains information about the component name which will be handling the data. Mostly, this component consists of the full class name. For example, `com.app.demoactivity.MyActivity` or `com.example.demoactivity.MainActivity`. This information about the component is optional for the intent object. If it is known to the intent object, Android will divert the data handling towards that particular component; if it is unknown, Android will identify what the best component is to handle this event.



The package part of the component name isn't necessarily the same as the project name in the `AndroidManifest.xml` file.

The component name is set by `setComponent()` or `setClassName()` methods that are provided by the Android APIs.

Intent resolution

Android intents are categorized into two parts (as described in *Chapter 3, Intent and Its Categorization*), **implicit intents** and **explicit intents**. For explicit intents, not assigning a component name to it does not cause any problems as the component has to be included in the intent object, and then Android will automatically direct the explicit intent towards the described component.

On the other hand, in implicit intents, if the component name is not given to the Android system, it will direct it automatically towards all the possible applications that can handle this incoming intent. This action will only take place if the intent has an intent filter, otherwise Android will not direct it. This term is called **Android intent resolution**; it is when you need not define the component for implicit intent, and it will automatically show a list of all the possible applications that can receive this intent.

Action

Action is a string that describes the action to be taken on an intent. For example, `ACTION_CALL`, `ACTION_BATTERY_LOW`, and `ACTION_SCREEN_ON`. You can find various other constants for actions at <http://developer.android.com/reference/android/content/Intent.html>. You can also make your own intent actions, but make sure to add the project name before it, for example, `com.example.myproject.SHOW_CONTACT`. The custom action is required when the developer wants to make an event that has not been previously added to Android SDK. This requirement can also occur when the developer wants to trigger/check an action which is closely related to that application only. Hence, it is not present in Android SDK.



`com.example.XXX` is the package name that is discouraged by Java and Android application development. It makes sure that the use of this package is mostly due to understanding the purpose of it in this example.

Action normally tells you how your intent is structured, especially the data and the extras. It is like a phenomenon of methods, where there are arguments and it returns values. It is a good practice to always use your action name as specifically as possible, and tightly couple them with the intent. Intent action can be set by using the Android API's method `setAction()`, and you can get it by using the `getAction()` method.

Some of the predefined constants for the intent action are given in the following table:

Constants	Component relation	Action
<code>ACTION_CALL</code>	Activity	Initiate a phone call
<code>ACTION_EDIT</code>	Activity	Display data from the user to edit
<code>ACTION_MAIN</code>	Activity	Start up as an initial activity with no data input and no returned output
<code>ACTION_SYNC</code>	Activity	Synchronized data on the server with the data on a mobile device
<code>ACTION_BATTERY_LOW</code>	Broadcast receiver	A warning that the battery is low
<code>ACTION_HEADSET_PLUG</code>	Broadcast receiver	A headset is plugged into the device
<code>ACTION_SCREEN_ON</code>	Broadcast receiver	The screen has turned on

Data

In Android intents, different types of actions are taken on the basis of the different types of data that are provided. Data is one of the fundamental parts on Android intents, especially in the implicit category. Let us look at some examples in order to have a better understanding of how to use data with its relevant action in Android intents.

Use of data in ACTION_EDIT

Consider an example of ACTION_EDIT. Whenever we call this action into intent, it is obvious that the edit functionality is to be implemented in a sort of the document. This document path is to be given in the form of a URI, which will then be handled by the Android intent. This URI is basically the part of the data that we put inside the intent object.

ACTION_EDIT can be used in a scenario where a developer wants to open the default Android's **Add new contact** screen in which the developer expects the user to edit. In this case, the intent which is called to open the **Add new contact** screen should have the ACTION_EDIT action defined.

Use of data in ACTION_CALL

Consider another example of ACTION_CALL. This action is used when we need to perform the call functionality through intent. So, in order to complete the task, we need to provide a telephone number by referencing it using a tel:// URI. This is the part of the data set that is to be provided with the intent so that Android may know on what data does it need to perform the dialling functionality.

Use of data in ACTION_VIEW

Moving towards our third example, that is ACTION_VIEW. In most cases, when this action is called, there is a website linked to it via a URI. This helps Android to understand the data on which the view action is to be performed. Normally, with ACTION_VIEW action, an http:// URI is attached, so that Android may process the functionality of viewing any web page.

Category

It is the additional information given to the intent in order to know the best kind of component required to perform that specific intent. For example, if there is a webpage that we want to view using ACTION_VIEW action, we can specify its category as CATEGORY_BROWSABLE, in order to let Android know that the data associated with the intent is safe and can easily be executed using the Android browser.

Some constants of categories that can be easily used in any Android program are listed in the following table:

Constants	Explanation
CATEGORY_BROWSABLE	The activity is safe to be executed on an Android browser using the data associated with the intent.
CATEGORY_GADGET	The activity is associated with another activity that is hosted by any Android gadget.
CATEGORY_HOME	The activity displays the home screen, or it is the first screen that the user sees when the Home button is pressed.
CATEGORY_LAUNCHER	The category of a particular activity is launcher, which means it is going to be the top of the stack activity.
CATEGORY_PREFERENCE	The destination activity is from the preference panel.

Extras

In the previous chapters, we had a good look in the extras feature and how can we can use it with intents. Just like the data, some extras are bound with the intent that is to be launched. For example, the ACTION_HEADSET_PLUG action has the extra "State" to indicate whether your headphones are connected to the cell phone or not.

These methods are parallel to those for bundle objects. So, the extras can be installed and read as a bundle using the `putExtras()` and `getExtras()` methods.

Intent filters

At this moment, we have a perfect understanding of Android intents and its implementation. Android intents are responsible for telling Android that a certain event has occurred, it is also used to give additional data on which a certain action should be taken. But how would Android know which component can facilitate the execution of any intent? For this, the concept of Intent filters comes in. Intent filters identify which component can react to a particular call to activities, services or broadcast intents.

Typically, intent filters are given to an activity or a service via `AndroidManifest.xml` file that consists of action, data, and category tests. In the case of broadcast receiver, intent filters can also be defined via code, dynamically.

For an implicit intent, it is necessary for it to pass all the three tests in order to deliver it to the particular component. Now, there can be two conditions based on these cases: one is when the intent does not pass any one of the tests, the intent will not be passed to the component. The other case is, when it has got its tests passed, it will directly be handed over to the respective component. In the first case, there is an exception that if it does not pass the test, it can be handed over to the next intent filter of the same activity. By this, it will be possible that it might be executed as per the expectation.



We can have multiple intent filters inside one activity in the `AndroidManifest.xml` file.

A normal XML tag of an activity having an intent filter inside looks like this:

```
<activity
  android:name=".MainActivity"
  android:label="@string/title_activity_main" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
</activity>
```

As you can see in the code, it consists of one activity tag which has everything inside it. This activity consists of only one intent filter that has two main components in it: **action** and **category**. The action to be taken at the execution of this intent is `android.intent.action.MAIN`, by calling this action any previous reference to the activity is removed, and the activity is executed with a fresh start. With this, the category is set as `android.intent.category.LAUNCHER`; this shows that the activity that is written inside the `AndroidManifest` file is the launcher's activity tag. That means, it is the first activity to be launched once the application is executed. If there are two or more activities described as launcher in the `AndroidManifest.xml` file, the Android operating system will ask the user which activity to start with.



`<intent-filter>` is part of the `AndroidManifest.xml` file and not of the Java code, because the information it contains is required before the project application is launched. For example, the category is to be determined if it is a launcher activity or not, before the start of the project application. As the `AndroidManifest.xml` file is executed before the start of the project application in order to extract the information about the project, intent filters are part of this file. The only exception is in the case of broadcast intent, in which the information can be modified dynamically from the Java code and not from the `AndroidManifest.xml` file.

Handling multiple intent filters

It is not a compulsion that any Android activity may have only one intent filter. One activity may incorporate various intent filters which occupy many sub components such as category, data, and actions. Take a look at the following screenshot which shows the two intent filters present with different types of parameters:

```
<activity android:name="MyList" android:label="@string/title_my_list">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <action android:name="android.intent.action.EDIT" />
        <action android:name="android.intent.action.PICK" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
    </intent-filter>
</activity>
```

The explanation of the code mentioned in the preceding screenshot will be covered in the upcoming topics of this chapter. For the time being, it is important to know the implementation of various intent filters inside an activity.

Test components of an intent filter

Filters are the representative of action, data, and category field of an intent object. Whenever an implicit intent is called, it is tested against these filters in order to get executed. If that intent does not fulfill any one of the test components, it will not be executed, or rather, it will be directed to a separate intent filter of the same activity (if it exists).

Now, in order to have a proper understanding of the intent filters, we need to go through a step-by-step evaluation of each test component associated with the intent filter. There are three test components present:

- Action test
- Data test
- Category test

Action test

Action describes what kind of action is to be executed by the coming intent. The `AndroidManifest.xml` file determines the requirements that are to be fulfilled by the incoming intent. If any intent is unable to match the specified action in the `AndroidManifest.xml` file, it will not be executed.

Action test is basically a test that is executed by the information given inside the manifest file of the project. All the action components are defined inside the `<intent-filter>` tags, and then matched in order to execute the intent. In the following screenshot, you can see how the `intent-filter` tag looks while having action tests:

```
<intent-filter>

    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    .....
</intent-filter>
```

In the code given in the preceding screenshot, there are three actions listed inside the `intent-filter` tags. These action tests will be determined by the Android operating system if the incoming intent is able to do these actions. There are three tests listed in the preceding code, descriptions of which you can see in the table given in the *Action* section. The following two conditions are to be followed:

If there is no action written inside the `intent-filter` tags, the Android operating system will refuse to process the intent as there is nothing available for matching.

If the `intent-filter` tag contains more than one action, but there is no action listed in the incoming intent, the intent will go through with it without any problem.

Writing conventions for `<action>`

There are certain conventions that Android follows while defining the actions. It should be kept in mind, for default actions we have to use the predefined constants that are given in the Android API. In the Android library, it is a convention that every action string starts with `ACTION_`, after which the real action name is written. For example, `ACTION_MAIN`, `ACTION_TIME_ZONE_CHANGED`, and `ACTION_WEB_SEARCH`.

Similarly, when it comes to the convention that is required to mention this string inside the `AndroidManifest.xml` file, Android follows the `android.intent.action.STRING` pattern. In this statement, the word *STRING* is replaced by the particular action that is to be matched but without the word *ACTION*. In order to understand the given statement, take the example of the `ACTION_MAIN` constant. If we want to mention it inside the `AndroidManifest.xml` file, we will not write `ACTION_`, instead we will write something like this:

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />
</intent-filter>
```

It is the same case with `ACTION_EDIT`, which enables the Android to edit any document whose reference is given in the URI. We will write the code, shown in the following screenshot, to make it understandable in the `AndroidManifest.xml` file:

```
<intent-filter>
  <action android:name="android.intent.action.EDIT" />
</intent-filter>
```

When it comes to the custom action, the action is defined by the user and not the Android API. There is a best practice that before writing it always starts with your package name in order to keep it unique. For example, if you want to make an action called `HIDE_OBJECTS`, you will have to write code, as shown in the following screenshot, in your XML file:

```
<intent-filter>
  <action android:name="com.yourapp.project.HIDE_OBJECTS" />
</intent-filter>
```


Category test

In order to pass the category test, it is necessary that the incoming intent category should be matched with at least one of the categories mentioned inside the `<category>` tag in `AndroidManifest.xml`. If an intent object is created without any knowledge of the category in it, it should always pass, no matter what categories are defined in the manifest file.

Keeping in mind that if we want to move between one activity to another using the `startActivity()` method, it is necessary that the activity that is willing to receive the implicit intent must have one default category mentioned in the `AndroidManifest.xml` file, which is `CATEGORY_DEFAULT` (as mentioned in Android API).



It is the same as writing the convention for action, the category should be written as `android.intent.category.DEFAULT`, without mentioning the `CATEGORY_` string in `AndroidManifest.xml`.

Although, this is not the case with launcher category; it is an exception. We mention `android.intent.category.LAUNCHER` in the launcher activity tags. The representation of the category test is shown in the following screenshot:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

In the code given in the preceding screenshot, there are two categories mentioned. The first category is `android.intent.category.DEFAULT`, which is because this particular activity is all set to receive the implicit intent. The other category that is mentioned in the manifest file is `android.intent.category.BROWSABLE`, which enables this activity to browse through the native Android browser present in the phone or any other applications that are for browsing websites.

Setting up the launcher activity

Setting up the launcher activity is primarily a part of the category. In this, we need to make sure that we completely understand the exception of the launcher activity with respect to intent. Since it is known that launcher activity is the one which is started just after the application is started for the first time, we can now move forward with its concept in category. The `DEFAULT` category is used if it is known that the activity will receive some implicit intent, but on the other hand, the `LAUNCHER` activity is the one that was started for the first time in any application.

In this sense, no launcher activity can be a default one at the same time. The result concludes that no activity can have `android.intent.category.DEFAULT` and `android.intent.category.LAUNCHER` at the same time in `AndroidManifest.xml`. The launcher activity presented in the manifest looks like the code shown in the following screenshot:

```
<activity android:name="com.example.android.application.MyList"
    android:label="@string/title_my_list">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

In code given in the preceding screenshot, the activity `com.example.android.application.MyList` is the launcher activity that will produce a list at the start of the application. Since this is the main entry point of the application, we provide `ACTION_MAIN` as the action in the manifest. While you can see the second tag, the category that is provided is given the name of `android.intent.category.LAUNCHER`.

Data test

Data tags are mentioned in order to facilitate the action taken on the executed activity. That is the reason why there can be multiple data tags inside one `<activity>` tag. The `<data>` tag consists of the information on a specific URI or MIME media type. For example, an activity may have the data tags, shown in the following screenshot, in it:

```
<intent-filter . . . >

  <data android:mimeType="video/avi" android:scheme="http" . . . />
  <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
  . . .

</intent-filter>
```

In the code given in the preceding screenshot, the intent filter contains two data tags. In each one of them, the MIME media type that is given under the `android:mimeType` attribute is the one that specifies the data format supported by the activity for a certain action. The `video/avi` value describes the video format of `.avi` files, which is supported by the activity. Similarly, if there is a need for mentioning the audio file type, we can use `audio/mpeg`.

It is also possible that we put an asterisk after the video or audio MIME Type. For example, see the following screenshot:

```
<intent-filter . . . >

  <data android:mimeType="video/*" android:scheme="http" . . . />
  <data android:mimeType="audio/*" android:scheme="http" . . . />
  . . .

</intent-filter>
```

This code is the same as the previous one, apart from the `video/*` and `audio/*` MIME types. The asterisk indicates that all possible subtypes of them are supported by this activity.

Now, there are some points that we need to make sure of:

- An intent object that does not contain any particular information about URI will only pass through the `intent-filter` tag if, and only if, there is no information of data is provided in the `AndroidManifest.xml` file
- An intent object that only contains the URI but not the data MIME type will only be passed if, and only if, it is matched with the URI specified in the filter and there is no filter specified for the data type
- An intent object that only contains the MIME type, but not the URI, will only be passed if, and only if, it is matched with the MIME type specified in the filter and there is no filter specified for the URI
- In the case where an intent object contains the URI, as well as the MIME type, it will only be passed if they are matched with the corresponding values of `intent-filters` specified in `AndroidManifest.xml`

Typical representation of the `<data>` tag

The `<data>` tag contains many attributes in order to make it complete information. The following syntax contains all the attributes that can be defined in the `<data>` tag, which will increase the knowledge of the activity while the processing of intent is happening:

```
<data android:host="string"
      android:mimeType="string"
      android:path="string"
      android:pathPattern="string"
      android:pathPrefix="string"
      android:port="string"
      android:scheme="string" />
```

In the code given in the preceding screenshot, there are various attributes which are all optional, yet they are more mutually dependent on one another. The list of the optional attributes is given as follows:

- `scheme`
- `host`
- `port`
- `path`
- `pathPrefix`
- `pathPattern`

Now, let's talk about their dependencies with one another. If the scheme is not mentioned in the data tag, no URI will remain valid after that. Similarly, if the host element is not defined, all the path tags and host tag values will be voided.

Summary

In this chapter, we had a detailed look at the intent filter and intent object. We saw the basic building blocks of the intent object, in which we define elements in the Java code, and on the other hand there are intent filters, which give knowledge to the Android OS about the activities present inside the application. We learned how `intent-filters` tags do their work by matching the incoming intent object and its attribute. Then, they decide whether or not the intent should be executed or not.

We also took a look on the action, data, and category, and how these work. How different data, categories, and actions are incorporated within a single activity in different intent filters, and what the main mechanism is if there are various filter choices available. We also looked at some writing conventions, the typical way of writing a launcher activity in the Android Manifest and how many MIME types are incorporated over when the data is valid for different subtypes of a format. In the next chapter, we will see how intents can be used with broadcast receivers, their practical examples, and the kind of issues that can arise because of them.

8

Broadcasting Intents

In the previous chapter, we learned about intent filters and how these filters provide information about different activities, services, and so on, to the Android OS. We also discussed how intent filters work and how they match the coming intent object with attributes. This chapter also provides information on action, data, and category tests.

Intents are the asynchronous way of sending messages between different components of the Android OS. So far, we have only learned to send and receive those messages, that is, the intents from one component to another component. But in each of the examples we discussed, we had the information about the receiver of the intent, such as which activity or service will receive the intent and will use the data embedded in the intent.

In this chapter, we will be extending our knowledge of sending intents to multiple broadcast receivers. We will learn how intents are broadcasted by the Android OS and how these broadcast intents are received.

This chapter includes the following topics:

- Broadcasting in the Android OS
- Broadcast intents in the Android OS
- System broadcasts in the Android OS
- Using the different system broadcasts of the Android OS
- Detecting the battery-low broadcast
- Detecting the screen `On/Off` broadcast
- Detecting the cell phone reboot completed broadcast
- Sending/receiving custom broadcast intents



The concept of intents and the structure of intents, as discussed in *Chapter 2, Introduction to Android Intents* and *Chapter 3, Intents and Its Categorization* are the prerequisites for understanding this chapter and the further chapters. If you don't have the basic concepts of these things, we would recommend that you read *Chapter 3, Intents and Its Categorization* and *Chapter 4, Intents for Mobile Components* in order to move forward.

Broadcasting in the Android OS

Any smartphone running Android OS has a lot of services and actions being executed at a particular time. These services and actions can be in the foreground or in the background. So the question that comes in our mind here is what these services and actions are actually doing. The answer is very simple. These services and actions are looking or listening for some events to occur, or performing some long operation in the background, or communicating with other components of Android OS, and so on. You might be wondering how these components listen for the occurrence of any event or how they communicate with other components, especially in background when user can't interact with the application directly. In the Android OS, these types of tasks are achieved by broadcasting. The Android OS continuously broadcasts the information about different actions, such as whether power has been connected and Wi-Fi has been turned on. We, the developers, use this broadcast information in our apps to make our apps more interactive and smart. In the next section, we will see how the Android OS broadcasts different information.

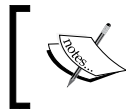
The broadcast intents

The broadcast intents are the `Intent` objects that are broadcasted via a method call to the `sendBroadcast()`, `sendStickyBroadcast()`, or `sendOrderedBroadcast()` methods of any `Activity` class. These broadcast intents provide a messaging and event system between different application components. Also, these intents are used by the Android OS to notify interested applications about system events such as low battery or whether headphones have been plugged in. To create an instance of the broadcast intent, we must include an action string in it. An action string is used to identify the broadcast intent, and it is unique. This action string typically uses the Java package name format.

In the following code snippet, we will create an instance of the broadcast intent and broadcast it:

```
Intent myBroadcast = new Intent();
myBroadcast.setAction("com.packt.myBroadcast");
sendBroadcast(myBroadcast);
```

You can see in the preceding code that there is no special class named `BroadcastIntent`. It is an ordinary `Intent` object. We have used these `Intent` objects in methods such as `startActivity()` or `startService()`. This time we have passed these `Intent` objects in the `sendBroadcast()` method of the `Activity` class. We have set its action string by calling the `setAction()` method. As discussed earlier, we have used the package-name format in the `setAction()` method. To broadcast any intent, we have used the `sendBroadcast()` method. This method broadcasts any given intent. Remember that this method call is asynchronous and will return immediately. You can't get any results from any receiver and receivers also can't abort any broadcast intent. The interested receivers match the action string of intent with their action string, and if matched, those receivers are executed.



From now on, we will use the keywords **broadcast** or **broadcasts** instead of **broadcast intent** in the whole chapter.

Built-in broadcasts in Android systems

The Android OS contains different types of broadcasts. The Android OS keeps broadcasting these intents to notify other applications about the various changes in the system. For example, when a device's battery gets low, the Android OS broadcasts an intent containing low-battery information; applications and services that are interested in this information receive it and perform actions accordingly. These broadcasts are predefined in the Android OS and we can listen for those intents in our application to make our apps more interactive and responsive.



You can find the list of all possible broadcasts in a text file named `broadcast_actions.txt`. This file is stored in the SDK folder under the `Android` folder.

```
<ANDROID_SDK_HOME>/platforms/android-<PLATFORM_
VERSION>/
  data/broadcast_actions.txt
```


The following table shows a list of some of the Android OS broadcasts with the description of their actions:

Broadcast intent action	Description
<code>android.intent.action.ACTION_POWER_CONNECTED</code>	This intent is broadcasted when a mobile phone is connected to a power source.
<code>android.intent.action.ACTION_POWER_DISCONNECTED</code>	This intent is broadcasted when a mobile phone is disconnected from any power source.
<code>android.intent.action.BATTERY_LOW</code>	This intent is broadcasted when a mobile phone's battery gets low.
<code>android.intent.action.BOOT_COMPLETED</code>	This intent is broadcasted when a mobile phone's booting completes.
<code>android.intent.action.DEVICE_STORAGE_LOW</code>	This intent is broadcasted when a mobile phone's device storage gets low.
<code>android.intent.action.NEW_OUTGOING_CALL</code>	This intent is broadcasted when a new outgoing call starts.
<code>android.intent.action.SCREEN_OFF</code>	This intent is broadcasted when a mobile's screen is turned on.
<code>android.intent.action.SCREEN_ON</code>	This intent is broadcasted when a mobile's screen is turned off.
<code>android.net.wifi.WIFI_STATE_CHANGED</code>	This intent is broadcasted when the WIFI state of a mobile phone is changed.
<code>android.media.VIBRATE_SETTING_CHANGED</code>	This intent is broadcasted when the vibrate settings of a mobile phone are changed.
<code>android.provider.Telephony.SMS_RECEIVED</code>	This intent is broadcasted when a mobile phone receives an SMS.

As we can see in the preceding table, the Android OS keeps informing different applications about various changes in the device's state by sending broadcasts. We can listen for these changes or broadcasts and can perform our custom actions to make our apps responsive.



You might have observed that some of the preceding intents, such as `android.provider.Telephony.SMS_RECEIVED`, are not included in the list in the SDK folder. Such intents are not supported in Android and are subject to change in any other future platform releases. Developers should be cautious when using these unsupported, hidden features in their apps.

Until now, we have only talked about broadcasts but we haven't still used them in practical examples. In the next section, we will develop some examples, in which we will listen for some Android OS's predefined broadcasts and perform actions accordingly.

Detecting the low-battery state of a device

In this section, we will implement a small application which will show an alert message when the phone's battery gets low. Now, let's get started with the development of our first example. But, in order to start this example, you need to build an Android project. You can use the Android Studio or Eclipse IDE (as per your convenience), but make sure that in the case of Eclipse, you have correctly installed JDK, ADT, and Android SDK along with their compatibility. If you don't know the difference between these IDEs, you can refer to *Chapter 1, Understanding Android*, of this book. Following those steps will help you to create a complete Android project with some predefined files and folders.

After creating an empty Android project, we have to modify two files: one main activity file and a manifest file. Also, we have added a receiver file as well. Let's look at these files in details now.

The BatteryLowReceiver.java file

As we are developing a Battery Low alert app, the first thing we have to do is to detect the low battery. For that purpose, we would have to create a `BroadcastLowReceiver` class, which will listen to the Battery Low broadcast. The following code shows the implementation of the receiver file:

```
public class BatteryLowReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        final Context ctx = context;
        if (intent.getAction() == Intent.ACTION_BATTERY_LOW)
        {
            Thread thread = new Thread(new Runnable() {

                @Override
                public void run() {
                    // TODO Auto-generated method stub
                    AlertDialog.Builder alert = new Builder(ctx);
                    alert.setTitle("Battery Low");
                    alert.setMessage("Please Recharge Your Phone");
                    alert.setNeutralButton("OK", null);
                    alert.show();
                }
            });
            thread.start();
        }
    }
}
```

As seen in the preceding code, we have extended our class from the `BroadcastReceiver` class and have overridden the `onReceive()` method. This method will be called when any broadcast is received. The first thing we have to do is to check whether this intent is the Battery Low intent or some other broadcast. To do so, we check the action string of the intent with the standard Android action which is `Intent.ACTION_BATTERY_LOW`. If the result is `true`, that means the device's battery is low, and we have to perform our custom action.

Next, we create a thread in which we pass an anonymous `Runnable` object. We override the `run()` method, and in this method, we create an instance of `AlertDialog` using the `AlertDialog.Builder` interface. We set the details, such as the title and the message of the alert, and then we display it.

You might be wondering why we have created a thread to show the alert. We could have shown alert without doing it in any thread. Well, it must be noted that broadcast receivers run for a very small amount of time. It is approximately about 4 milliseconds. Developer should be very careful when performing operations in receivers. It is a good practice to perform operations such as creating alerts and starting activities and services in threads from broadcast receivers.

Now, our `BatteryLowReceiver` class is ready. But, how is this receiver triggered and how can this class receive the Battery Low broadcasts from the Android OS? The answers to these questions are explained in the next section. Let's see our activity file now in detail.

The `BatteryLowActivity.java` class

This class represents our main activity of the application which means that whenever an application is launched, this activity will be started first. The following code shows the implementation of our activity file:

```
public class BatteryLowActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // TODO Auto-generated method stub
        super.onCreate(savedInstanceState);

        IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_LOW);
        BatteryLowReceiver batteryLow = new BatteryLowReceiver();
        registerReceiver(batteryLow, filter);
    }
}
```

As always, we extended our class from the `Activity` class. Then, we have overridden the `onCreate()` method of our activity. We created an instance of `IntentFilter` and passed the `Intent.ACTION_BATTERY_LOW` action string in its constructor. You can read more about intent filters in *Chapter 7, Intent Filters*. After that, we created an instance of our `BatteryLowReceiver` class. Finally, we call our `registerReceiver()` method and pass our receiver and filter objects in it. This method tells the Android OS that our application is interested in the Battery Low broadcast. This is how we can listen to the Battery Low broadcast. One thing to be noted here is that when you call the `registerReceiver()` method, it is the developer's responsibility to call the `unregisterReceiver()` method too, when an application is not interested in listening to the Battery Low broadcast. If the developer doesn't unregister it, this application, no matter whether it is opened or closed, will listen for the Battery Low broadcast and take an action accordingly.

This can be bad for the memory and the optimization of our application. We can call the `unregisterReceiver()` method in the `onDestroy()`, `onPause()`, or `onStop()` callbacks of our Activity class, as in the following code snippet:

```
public void onPause ()
{
    unregisterReceiver (batteryLow) ;
}
```

The AndroidManifest.xml file

Developers can also register a receiver in the `AndroidManifest.xml` file as well. The advantage of registering receivers in the manifest file is that developers don't have to unregister them manually by calling the `unregisterReceiver()` method. The Android OS takes care of these receivers on its own, and the developer doesn't have to worry about it anymore. The following is the code implementation of our `AndroidManifest.xml` file which registers our Battery Low receiver in it:

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

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

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".BatteryLow"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

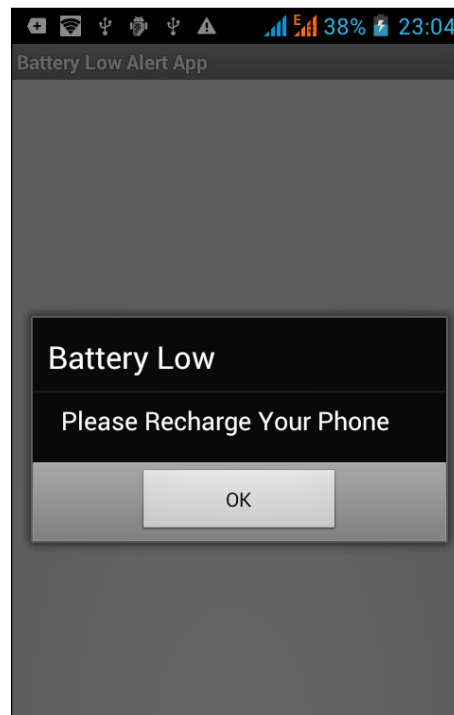
        <receiver android:name="com.batterylowexample.BatteryLowReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BATTERY_LOW"/>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

You can see in the preceding code that we have used the `<receiver>` tag in our `<application>` tag to register our broadcast receiver. We have inserted the whole package name of `BatteryLowReceiver`, as the name of receiver in the `android:name` attribute of the `<receiver>` tag. As we set the intent-filter action in our activity file by creating an instance of the `IntentFilter` class, we are embedding the `<intent-filter>` tag with the action name set to `android.intent.action.BATTERY_LOW`. This intent filter will tell the Android OS that the receiver is interested in the low-battery state information of the device.



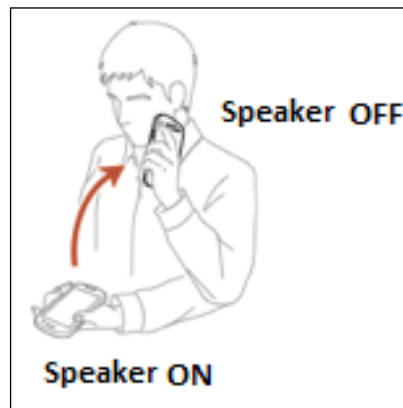
It must be noted that developers should register receivers by only one method; either from their activities by calling the `registerReceiver()` method or from their `AndroidManifest.xml` files. It is a good practice to use the `AndroidManifest.xml` file to register `BroadcastReceiver` of the application.

When we run our application, we will see a blank screen because we haven't set any layout for our activity. But when mobile phone gets low on battery, an alert box will be shown in our phone. The following screenshot shows an alert box from our `BatteryLowReceiver` class:



Detecting the screen on/off state of a phone

Almost in all Android phones, we have seen a very interesting feature while attending a phone call; we can see that the screen goes on or off. In addition to this, you might have observed that when you bring your phone near your ear, the screen turns off, and when you take it away from your ear and hold it in your hand, the screen automatically turns on. This is an interesting behavior of smartphones. Let's say that we want to develop an application in which whenever the screen turns on, we want to turn on the speaker mode so that other people with us can hear and participate in the phone conversation. And when we put it on our ear again and the screen turns off, we want to turn speaker mode off. The following figure shows the concept of this application:



Now, let's develop such an application in the following example. Let's start from creating an Android project in your favorite IDE. Then, we will have to first detect whether the screen has been turned on or off. To detect this, we will implement our custom `BroadcastReceiver` class. Let's implement our broadcast receiver class in next section.

The `ScreenOnOffReceiver.java` file

The `ScreenOnOffReceiver.java` file represents our custom broadcast receiver for detecting the screen on/off state of the phone. The following code implementation shows our screen on/off detecting receiver:

```
public class ScreenOnOffReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        final Context ctx = context;
        if (intent.getAction() == Intent.ACTION_SCREEN_ON)
        {
            // Screen Has been Turned ON
            AudioManager am = (AudioManager) ctx.getSystemService(Context.AUDIO_SERVICE);
            // Check whether we are currently in call or not.
            if (am.getMode() == AudioManager.MODE_IN_CALL)
            {
                // A Call is in progress, set speaker phone on.
                am.setSpeakerphoneOn(true);
            }
        }
        else if (intent.getAction() == Intent.ACTION_SCREEN_OFF)
        {
            // Screen Has been Turned Off, Set Speaker Phone Off
            AudioManager am = (AudioManager) ctx.getSystemService(Context.AUDIO_SERVICE);
            // Check whether we are currently in call or not.
            if (am.getMode() == AudioManager.MODE_IN_CALL)
            {
                // A Call is in progress, set speaker phone off.
                am.setSpeakerphoneOn(false);
            }
        }
    }
}
```

As in the previous example, we are extending our `ScreenOnOffReceiver` class from the `BroadcastReceiver` class and overriding the `onReceive()` method. This method will be called when any broadcast intent is received by our application. Our application first checks whether it is screen on/off intent or not by comparing the intent action with the `Intent.ACTION_SCREEN_ON` or `Intent.ACTION_SCREEN_OFF` constants. Remember, in the previous example we were listening for only a single broadcast intent. However in this example, we are listening for two broadcast intents: one for screen on and other for screen off.

In Android phones, the screen turns on/off not only during calls. It also becomes on/off when the phone is locked/unlocked. So before setting our speaker on/off, we have to check whether we are currently in a call or not. We can detect it by checking the mode of `AudioManager`. If the mode is `AudioManager.MODE_IN_CALL`, that means we are currently in any incoming or outgoing call conversation. Once we are confirmed about the call mode status, then we can set the speaker on/off. We are using the `AudioManager.setSpeakerphoneOn(boolean)` method for this purpose.

Until now, we have implemented our receivers. But we haven't registered these receivers. Remember from our previous example, we used two approaches to register our custom broadcast receivers: one from the activity class by using the `registerReceiver()` method and the other from the `AndroidManifest.xml` file. Let's choose the latter approach of the `AndroidManifest.xml` file to register our receivers.

The AndroidManifest.xml file

As in the previous example, we will register our `ScreenOnOffReceiver` broadcast receiver in this manifest file. It should be noted that in the previous example of the Battery Low application, we registered our receiver for only one filter, which was the low-battery state of the phone. However, in this example, we are listening for two state filters: screen on and screen off. But, we have implemented only one broadcast receiver. So, let's see how we can register one receiver with two intent filters in the following code implementation of the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packt.screen_on_off_example"
    android:versionCode="1" android:versionName="1.0" >
    |
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="15" />
    |
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        |
        <activity
            android:name=".ScreenOnOffExampleApp" android:label="@string/app_name" >
            |
            <intent-filter>
                |
                <action android:name="android.intent.action.MAIN" />
                |
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        |
        <receiver android:name="com.packt.screen_on_off_example.ScreenOnOffReceiver">
            |
            <intent-filter>
                |
                <action android:name="android.intent.action.SCREEN_ON" />
            </intent-filter>
            |
            <intent-filter>
                |
                <action android:name="android.intent.action.SCREEN_OFF" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

You can see in the preceding code that we have put the `<receiver>` tag in our `<application>` tag to register our receiver. Also, it should be noted that this time we have used the `<intent-filter>` tag twice with two different actions embedded in it: one for `android.intent.action.SCREEN_ON` and the other for `android.intent.action.SCREEN_OFF`. You can read more about multiple intent filters in *Chapter 7, Intent Filters*. These two intent filters along with the receiver embedded in our `AndroidManifest.xml` file registers our `ScreenOnOffReceiver` broadcast receiver with the Android OS to listen to the screen-on and screen-off state changes of the mobile phone.

Detecting the cell phone's reboot-completed state

Many android applications run services in the background when running multiple tasks and operations. For example, a weather application keeps checking the weather after a fixed time interval by using a background service. But have you ever wondered that when you reboot your cell phone or your battery dies and your phone is rebooted, then how these services start running again after reboot? Well, we will see how this can be done in this section.

When an Android phone is rebooted successfully, the Android OS broadcasts an intent notifying other applications that the reboot is completed. Then those applications start their background services again. In this section, we will create an application that will listen to the reboot-completed broadcast, and we will start our test service from it.

Let's create an empty Android project in any IDE such as Eclipse or Android Studio. As always, we will first implement our broadcast receiver class.

The PhoneRebootCompletedReceiver.java file

The PhoneRebootCompletedReceiver.java class represents our reboot-completed broadcast receiver file. The following code shows the implementation of the file:

```
public class PhoneRebootCompletedReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        final Context ctx = context;

        // If Android System Booting has completed
        if (intent.getAction() == Intent.ACTION_BOOT_COMPLETED)
        {
            // Start our Temp Service
            Intent service = new Intent(ctx, TempService.class);
            ctx.startService(service);
        }
    }
}
```

You can see in the preceding code that we haven't done anything new. We have extended our class from the `BroadcastReceiver` class. Then, we check for the `Intent.ACTION_BOOT_COMPLETED` action of the intent. If it is true, we start our temporary service by calling the `Context.startService()` method. Now, let's see what the `TempService` class does, in the next section.

The TempService.java file

The `TempService.java` class represents our service which will start when the Android system booting is completed.



In Android 3.0, the user needs to have started the application at least once before the application can receive the `android.intent.action.BOOT_COMPLETED` broadcast.

The following code shows the implementation of our `TempService` class:

```
public class TempService extends Service
{
    @Override
    public IBinder onBind(Intent intent) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // TODO Auto-generated method stub
        Toast.makeText(getApplicationContext(), "Service started", Toast.LENGTH_LONG).show();
        return START_STICKY;
    }
}
```

Like any usual service class, we have extended our class from `Service`. We have overridden two methods: `onBind()` and `onStartCommand()`. In the `onStartCommand()` method, we will display a toast by calling the `Toast.makeText()` method with the "**Service started**" text. When our phone's booting is complete, this toast will be displayed. We can implement our custom operations here in this method.

Now, all that we are left with is to inform the Android OS that our application is interested in listening out for the Boot Completed broadcast. As in the previous applications, we will register our receiver in the `AndroidManifest.xml` file. Let's see this in the next section.

The AndroidManifest.xml file

The AndroidManifest.xml file informs the Android OS that our application is interested in listening for the Boot Completed broadcast. The following code shows the implementation of the manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packt.boot_complete_example"
    android:versionCode="1" android:versionName="1.0" >
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="15" />

    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

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

        <service android:enabled="true" android:name=".TempService" ></service>

        <receiver android:name="com.packt.boot_complete_example.ScreenOnOffReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Almost everything is the same as in the previous example applications. We have registered our receiver using the `<receiver>` tag nested in the `<application>` tag with the intent filter of the `android.intent.action.BOOT_COMPLETED` action. We have also registered `TempService` by using the `<service>` tag nested within the `<application>` tag. It must be noted that the Boot Completed broadcast requires users to grant the `android.permission.RECEIVE_BOOT_COMPLETED` permission. We can ask the user to grant this permission by adding the `<uses-permission>` tag with the `android:name` attribute set to `android.permission.RECEIVE_BOOT_COMPLETED`. This is how we can start our custom services when a phone is rebooted.

Sending and receiving custom broadcasts

Until now, we have been only receiving broadcasts. And all those intents we have experimented with are the Android System broadcasts. In this section, we will talk about custom broadcasts. We will see how we can send our own custom broadcasts to other applications and how other applications can listen for our custom broadcast intents.

In the next section, we will create an example that will send custom broadcasts to other applications. Let's create the activity and layout file for the application now.

The activity_main.xml layout file

The activity_main.xml file represents the layout file of our activity. The following code shows the implementation of the manifest file:

```
<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" >

    <Button
        android:id="@+id/btnSendBroadcastIntent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Send Broadcast Intent" />

</RelativeLayout>
```

As you can see in the layout file, we have placed a button with the ID, btnSendBroadcastIntent. We will use this button in our activity file to send the broadcast to other applications. Let's see the activity file now.

The MainActivity.java file

The MainActivity.java file is the main launcher point of our application. This activity will use the activity_main.xml layout file as its visual part. The following code shows the implementation of the file:

```
public class MainActivity extends Activity {

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

        Button sendButton = (Button) findViewById(R.id.btnSendBroadcastIntent);
        sendButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // TODO Auto-generated method stub
                Intent broadcast = new Intent();
                broadcast.setAction("com.packt.CustomBroadcast");
                sendBroadcast(broadcast);
            }
        });
    }
}
```

You can see in the preceding code that we have obtained the Button object from our layout file by calling the findViewById() method. Then we set its OnClickListener() method, and in the overridden onClick() method, we perform our main operation of sending broadcasts to other applications. We create an Intent object and set its action string by calling the Intent.setAction() method. It should be noted that we have defined our own custom action value this time as the com.packt.CustomBroadcast string. We should follow the package-naming convention when we create our own custom broadcast receivers. Finally, we use that intent for broadcasting by calling the sendBroadcast() method of the Activity class. This is how our custom broadcast intent is sent to the Android OS and other applications. Now, all of the applications and receivers that are listening for this type of broadcast will receive it, and hence, can perform their custom operations. In the next section, we will implement our custom broadcast receiver class which will receive this type of intent and display a toast to notify the user.

The CustomReceiver.java file

The CustomReceiver.java file represents our custom broadcast-receiver class, which will receive our custom broadcast. This class can be in this application or any other application which is interested in listening for this custom type of broadcast. Like all of the previous examples, this class will be the same and extended from the BroadcastReceiver class. The only difference between the previous examples and this example is that we were using the Android OS's standard predefined constant action strings to detect the System broadcasts, but in this example, we are listening for our own custom broadcasts with custom action strings set. The following code shows the implementation of the file:

```
public class OurCustomReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        if (intent.getAction() == "com.packt.CustomBroadcast") {
            Toast.makeText(context, "Broadcast Intent Detected.",
                Toast.LENGTH_LONG).show();
        }
    }
}
```

You can see in the preceding code that we haven't done anything new which you aren't already familiar with. We have derived our class from BroadcastReceiver and overridden the onReceive() method. We then compared the action string of the intent with our own custom string of the com.packt.CustomBroadcast action. If it is true, we will display a toast saying Broadcast Intent Detected. We can perform our custom operations here in this method. Finally, we have to register this receiver so that the Android OS can notify our application about the broadcast.

The AndroidManifest.xml file

As always, the AndroidManifest.xml tells the Android OS that our application is listening for custom broadcasts. The following code shows the implementation of the file:

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

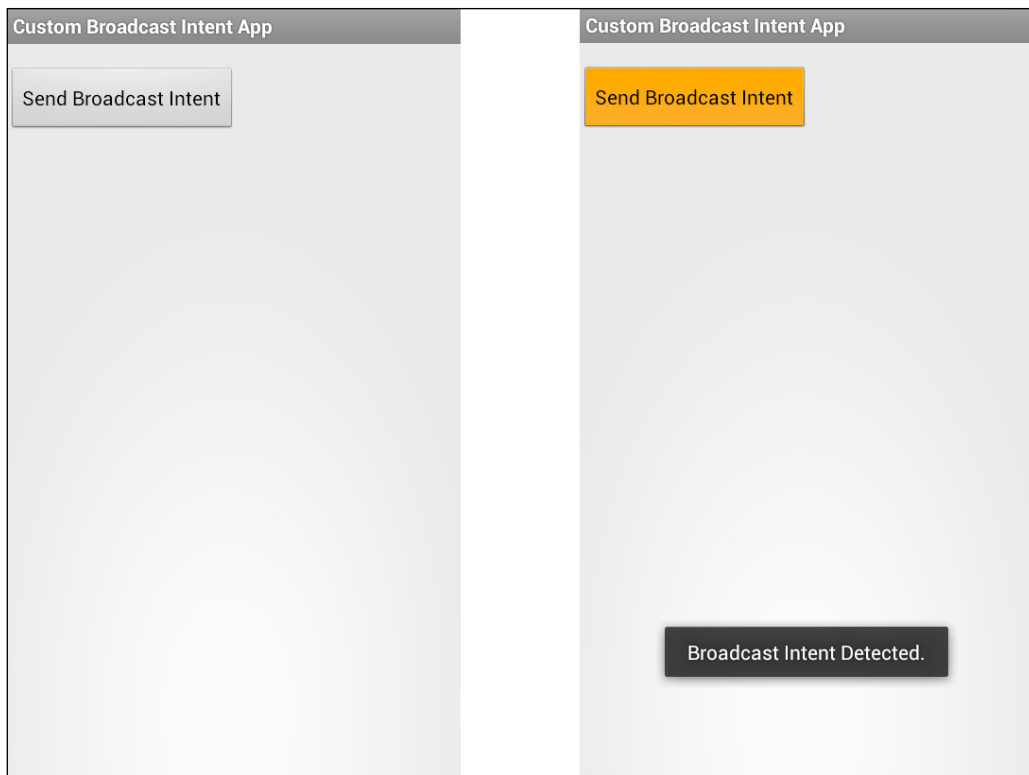
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >

        <activity
            android:name=".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>

        <receiver android:name=".OurCustomReceiver" >
            <intent-filter>
                <action android:name="com.packt.CustomBroadcast" > </action>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

You can see that we have registered our custom broadcast receiver in the same way as we have registered the receivers for Android System broadcasts. Now, when we run this application, we will see a button named **Send Broadcast Intent**. When we tap on the button, our custom broadcast will be broadcasted in the Android OS. As we have also created a receiver of this custom intent, so we will also receive this intent. On receiving the intent, our custom receiver will display a toast. The following screenshot shows the execution of this application:



Summary

In this chapter, we discussed about broadcasts. We also saw the different Android OS's System broadcast intents such as Battery Low, Power Connected and Boot Completed. Also, we saw how these broadcasts are received by registering our custom receivers and how we can perform our own custom operations in those receivers. Finally, we learned about sending our own custom broadcasts and receiving those custom intents as well.

In the next chapter, we will explore two special types of intents: `IntentService` and `PendingIntent`. Also, we will learn how these intents are used and what can be achieved by these intents.

9

Intent Service and Pending Intents

From the very beginning of this book, we have been studying different tasks that an intent can do to facilitate Android and their types. We have seen that intents can help to navigate in between the activities. They are also used to transfer data between them. We saw how we can put filters in order to verify whether the incoming intent can qualify the component test and in the end, we learned the role of intent in Broadcast Receivers. In this chapter, we will have an advanced look at how the intents can be used for doing handy things using Intent Services and Pending Intents.

In this chapter, we will have a look at the following topics:

- What is Intent Service?
- Usage and Implementation of Intent Service
- What is Pending Intent?
- Usage and Implementation of Pending Intent
- Summary

Intent Service

Intent Service is a simple kind of service that is used to handle asynchronous work which has nothing to do with the main thread. This can be done if the client sends the request by the `startService(Intent intent)` method. This new task will be handled by the worker thread and stops when it runs out of work.



Intent Service is inherited by the `Service` class present in Android API

Intent Service is used to offload the working thread, so that it does not become the bottleneck. It helps to make things go separately as of the main application thread. It is to be noted that though it works independently of the main thread, only one request can be processed at a given time.

Intent Service is the best way to offload the work from the UI thread of your application and into a work queue. There is no need to make asynchronous tasks and manage them for every processing. Rather, you define an Intent Service, enable it to handle the appropriate data that you want to send for the processing, and simply start the service. In the end, you can send the data back to the application by broadcasting it in an intent object and catching it from the Broadcast Receiver to use it in the application.

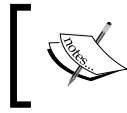
Comparison of four fundamentals

This section shows the basic difference between four of the most important elements (Service, Thread, Intent Service, and Async Task) of Android development including Intent Service.

Best case to use

Best case scenarios for Service, Thread, Intent Service, and Async Task are given in the following table:

Best case scenario	
Service	When task is not too long and has nothing to do with the main thread
Thread	When there is a long task to perform and more than one task has to be done in parallel
Intent Service	When there are long tasks without any intervention from the main thread and also where callbacks are needed
Async Task	When there are long tasks in which communication with the main thread is needed and also where there is a need for parallel work to be done



If there is a need for Intent Service to communicate with the main thread, we need to use Handler or Broadcast Intents.

Triggers

The difference in the triggers of Service, Thread, Intent Service, and Async Task has been discussed in the following table:

	Service	Thread	Intent Service	Async Task
Triggers	By using the <code>onStartService()</code> method	By using the <code>start()</code> method	By Intent	By using the <code>execute()</code> method
Cause of Triggers	Can be called from any thread	Can be called and run by any other thread	Can only be called from the main thread	Can only be called from the main thread
Runs on	Can be called from the main thread	Its own thread	Separate worker thread	Separate worker thread although the method of the main thread can be run in between
Limitations	Can block the main thread in certain scenarios	Has to be manually handled and the code may not be easily understandable	Cannot handle multiple tasks simultaneously and all tasks work on the same worker thread	Can have only one instance of one task and cannot be run in a loop

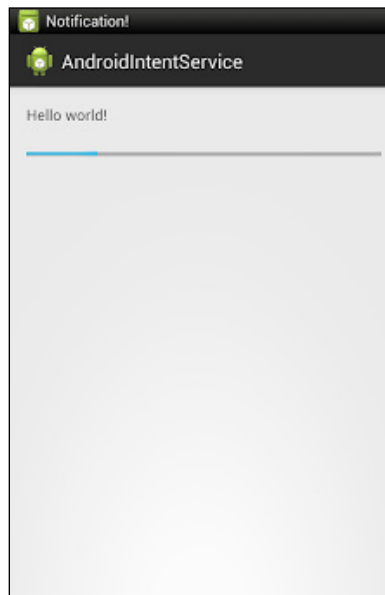
Usage and implementation of Intent Service

From the previous parts of this chapter, we have the clear view of the definition of Intent Service and what the fundamental differences are that it has with Threads, Async Tasks, and Service. It is now time to start with the implementation and usage of Intent Services. For this, we will start with the example which will help us to learn how to generate a fake notification from Intent Service.

Generating a fake notification from Intent Service

In this example, we will learn the use of Intent Service in producing the notification on your notification bar. The example will also explain the use of the `onHandleIntent()` method which is used to implement all the functionality of Intent Service that also includes sending the broadcast and the notification to the notification bar.

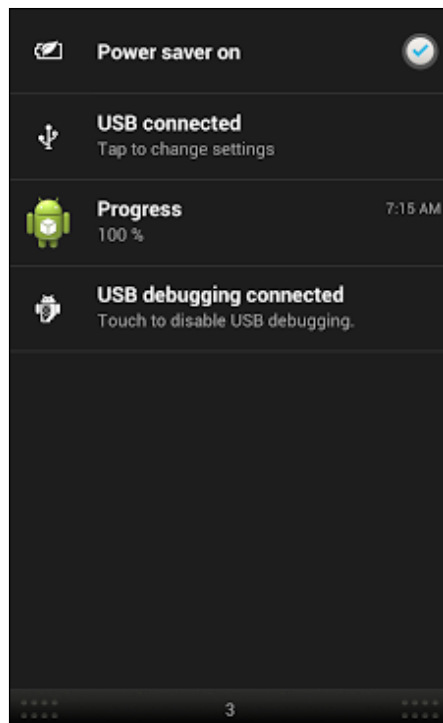
Moreover, at the end of this section you will learn the difference between it and Thread, or any other previously mentioned Android-defined method. After the completion of this code, start the activity and you will get a view of these screens:



Start of Activity will show the Hello World Screen



Note: Remember that in this example we will not go through the complete set of files that is used in the project. Since this is the last chapter of the book, we assume that you already got the basics of Android development in terms of XML file, resources, and layouts



The Notification panel showing the Progress notification

A glance at the code

The example refers to the use of Intent Service in a scenario when there is a need for sending a message to the notification bar about progressing or signaling of any particular event.

```
package com.app.intentservice;

import android.app.IntentService;
import android.app.Notification;
import android.app.NotificationManager;
import android.content.Context;
import android.content.Intent;
import android.support.v4.app.NotificationCompat;
```



```
public class CustomIntentService extends IntentService {

    private static final int NOTIFICATION_ID=1;
    NotificationManager notificationManager;
    Notification notification;

    public static final String ACTION_CUSTOM_INTENT_SERVICE
        = "com.app.intentservice.RESPONSE";
    public static final String ACTION_MY_UPDATE =
        "com.app.intentservice.UPDATE";
    public static final String EXTRA_KEY_IN = "EXTRA_IN";
    public static final String EXTRA_KEY_OUT = "EXTRA_OUT";
    public static final String EXTRA_KEY_UPDATE = "EXTRA_UPDATE";
    String activityMessage;

    String extraOut;

    public CustomIntentService() {
        super("com.app.intentservice.CustomIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {

        //get input
        activityMessage = intent.getStringExtra(EXTRA_KEY_IN);
        extraOut = "Hello: " + activityMessage;

        for(int i = 0; i <=10; i++){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            //send update
            Intent intentUpdate = new Intent();
            intentUpdate.setAction(ACTION_MY_UPDATE);
            intentUpdate.addCategory(Intent.CATEGORY_DEFAULT);
            intentUpdate.putExtra(EXTRA_KEY_UPDATE, i);
            sendBroadcast(intentUpdate);
        }
    }
}
```

```
//generate notification
String notificationText = String.valueOf((int)
    (100 * i / 10)) + " %";
notification = new NotificationCompat.Builder
    (getApplicationContext())
    .setContentTitle("Progress")
    .setContentText(notificationText)
    .setTicker("Notification!")
    .setWhen(System.currentTimeMillis())
    .setDefaults(Notification.DEFAULT_SOUND)
    .setAutoCancel(true)
    .setSmallIcon(R.drawable.ic_launcher)
    .build();

notificationManager.notify(NOTIFICATION_ID, notification);
}

//return result
Intent intentResponse = new Intent();
intentResponse.setAction(ACTION_CUSTOM_INTENT_SERVICE);
intentResponse.addCategory(Intent.CATEGORY_DEFAULT);
intentResponse.putExtra(EXTRA_KEY_OUT, extraOut);
sendBroadcast(intentResponse);
}

@Override
public void onCreate() {
    // TODO Auto-generated method stub
    super.onCreate();
    notificationManager = (NotificationManager)
        getSystemService(Context.NOTIFICATION_SERVICE);
}
}
```

Dive into the understanding

Build a new project and open the `src` folder. Create a new class file with the name of `CustomIntentService.java` which is the child class of `IntentService`. Extend the `IntentService` class and override the method `onHandleIntent(Intent intent)`.

At this point, you are all set to implement your own Intent Service that is responsible for sending a message to the notification bar and update it in the form of the Progress Percentage Bar format. Now, let's start understanding the code by going through the following steps:

1. The first step is to declare the variables `notificationManager` and `notification` in order to use them inside the `onHandleIntent()` method. There are some other static final variables that we will be using in this project. They are `NOTIFICATION_ID`, `ACTION_CustomIntentService`, `ACTION_MyUpdate`, `EXTRA_KEY_IN`, `EXTRA_KEY_OUT`, and `EXTRA_KEY_UPDATE`. Two new strings variables are also required in order to handle the notification string, stated as `activityMessage` and `extraOut`.
2. The main implementation of this `IntentService` will take place in the `onHandleIntent()` method where we will define the working which includes messages to the notification bar and broadcasting of messages.
3. At the start of this `onHandleIntent()`, the extras are obtained by the `intent.getStringExtra()` method and saved in the `msgFromActivity` variable which will later be sent to broadcast.
4. Our main objective is to send a notification which will show 0 to 100 % progress (a fake counter) and get updated in the notification bar. For that, we are initializing a for loop which will go from 0 to 10. At the start of this, we will call `Thread.sleep(1000)`, which will make the thread sleep and will not work for 1000 milliseconds.
5. Once the thread has slept for a certain time, the first counter of our fake progress update is done. Our next step is to send a broadcast whose main purpose is to give the update. In order to see this, we use the following lines of code:

```
//send update
Intent intentUpdate = new Intent();
intentUpdate.setAction(ACTION_MyUpdate);
intentUpdate.addCategory(Intent.CATEGORY_DEFAULT);
intentUpdate.putExtra(EXTRA_KEY_UPDATE, i);
sendBroadcast(intentUpdate);
```

A quick overview of how we send a broadcast: make a new intent object, and give it a name and action of `intentUpdate`; since it is a custom action, give it a name of `ACTION_MyUpdate` which you can see in the code; define its category which is also a custom category; put the counter information (the variable that shows the current counter of the loop) and send a broadcast for this intent.

6. The next step is to send the notification to the notification bar. The following lines of code can be seen inside the previous example:

```
//generate notification
String notificationText = String.valueOf((int)
    (100 * i / 10)) + " %";
myNotification = new NotificationCompat.Builder
    (getApplicationContext())
    .setContentTitle("Progress")
    .setContentText(notificationText)
    .setTicker("Notification!")
    .setWhen(System.currentTimeMillis())
    .setDefaults(Notification.DEFAULT_SOUND)
    .setAutoCancel(true)
    .setSmallIcon(R.drawable.ic_launcher)
    .build();

notificationManager.notify
    (MY_NOTIFICATION_ID, myNotification);
```

This code sets the value of `notificationText` to the current counter of the loop and converts it into the percentage; makes a new notification by calling the `NotificationCompat.Builder()` (which is basically a builder pattern described in Android SDK) and gives it the application context, sets its title content, text, ticker, when-to-appear, and some other properties. At the end, you have to call `notificationManager.notify()` in order to show it in the notification bar.

7. The last step is to send another broadcast as an acknowledgement, and it has the same procedure as that of the previous broadcast, as you can see in the following code:

```
//return result
Intent intentResponse = new Intent();
intentResponse.setAction(ACTION_MyIntentService);
intentResponse.addCategory(Intent.CATEGORY_DEFAULT);
intentResponse.putExtra(EXTRA_KEY_OUT, extraOut);
sendBroadcast(intentResponse);
```

8. The last step showed in the code is to override the `onCreate()` method. You must have noticed that we did not make a new object of the notification manager which will certainly give an error. So, in order to make a new object, we will get the system service of Android by using the notification manager `getSystemService (Context.NOTIFICATION_SERVICE)`.



This example will also need a Broadcast Receiver. If you still don't have an idea about it, you can refer to previous chapters.

Taking another example

The previous example mainly deals with the implementation of notification in the Android notification bar. It covered the implementation of Intent Service but not the making of Broadcast Receiver and its registration. In this example, we will learn how to use Intent Service and convert all the input data to uppercase and broadcast it back to Broadcast Receiver. The implementation of Broadcast Receiver is also a part of this example.

Starting with the example, use the following code to implement it on your development environment:

```
public class IntentServiceExampleTwo extends IntentService {
    private static final String TAG
        =IntentServiceExampleTwo.class.getSimpleName();

    public static final String INPUT_TEXT_STRING
        ="INPUT_TEXT_STRING";
    public static final String OUTPUT_TEXT="OUTPUT_TEXT";

    /**
     * initiate service in background thread with service name
     */
    public IntentServiceExampleTwo() {
        super(IntentServiceExampleTwo.class.getSimpleName());
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        Log.i(TAG, "onHandleIntent()");

        String data =intent.getStringExtra(INPUT_TEXT_STRING);
        Log.d(TAG, data);

        data=data.toUpperCase();

        SystemClock.sleep(4*1000);

        Intent stringBroadCastIntent =new Intent();

        stringBroadCastIntent.setAction
            (TextCapitalizeResultReceiver.ACTION_TEXT_CAPITALIZED);
```

```

        stringBroadcastIntent.addCategory(Intent.CATEGORY_DEFAULT);

        stringBroadcastIntent.putExtra(OUTPUT_TEXT, data);

        sendBroadcast(stringBroadcastIntent);
    }
}

```

This is almost the same implementation that was done before, in the first example. In this example, the working of the `onHandleIntent()` method is shown, in which the following steps are taking place:

1. In the `onHandleIntent()` method, the first step that you can see is getting data from the coming intent and saving it into a variable. The variable `data` contains the incoming data which we will convert into the uppercase.
2. The second step is logging the data into LogCat, which is obtained by using the method `Log.d(String, String)`. The first argument is `TAG`, which is normally the class name that is declared at the global level, so that any method may use it. This class name is important to distinguish your message from others (makes it easy to read). The second argument is the message string which is used to show any data in the process, so that at runtime the developer may see its value.
3. The third step is to convert this data into upper case. This will help to reflect the change in the broadcasted intent. Save this back into the `data` variable.
4. The rest of the steps are the same as the previous example in which the intent object is made, the categories and action are defined, data is put as an extra and is sent to be broadcast by the receiver.

The next step, is to set the receiver which will be received from the `sendBroadcast()` method. For this, take a look at the following code:

```

public class UpperCaseReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        TextView textViewResult =
            (TextView) findViewById(R.id.receiving_text_view);
        String result =intent.getStringExtra
            (ExampleIntentService.OUTPUT_TEXT);
        textViewResult.setText(result);
    }
};

```

The previous code is that part of the example where how to make a Broadcast Receiver is written and this will receive the broadcast back and set `textView`. You can see in the code that the `onReceive()` method is overridden where the class is extending the Broadcast Receiver. Inside the `onReceive()` method, the string is obtained by the `intent.getStringExtra()` method and is saved in the result string. This string will be used to set the text of the `textView`, so that you can see the changes as they are reflected in the `textView`.

Moving forward, the next step is to register this receiver with Intent Service. This will be done inside the activity where the receiver is linked to the Intent Filter, so that it can have its effect. This is shown in the following piece of code:

```
private void registerReceiver() {  
  
    IntentFilter intentFilter =new IntentFilter  
        (UpperCaseResultReceiver.ACTION_TEXT_CAPITALIZED);  
  
    intentFilter.addCategory(Intent.CATEGORY_DEFAULT);  
  
    capitalCaseReceiver=new UpperCaseResultReceiver();  
  
    registerReceiver(capitalCaseReceiver, intentFilter);  
}
```

The method `registerReceiver()` is declared inside your activity which will be called from the `onCreate()` or `onResume()` methods, so that it can register the Broadcast Receiver while starting or resuming the activity.

- The Intent Filter is declared and initialized with the object named `intentFilter`.
- The `intentFilter` object is set as default.
- The object of the Broadcast Receiver is initiated and registered with the Intent Filter by calling the `registerReceiver(Receiver, IntentFilter)` method.

After registering the receiver with the Intent Filter, the next step is to use this in your activity. For this, take a look at the following code. This code can be inside any event:

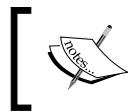
```
Intent textUpperCaseIntent = new Intent  
    (MainActivity.this, ExampleIntentService.class);  
  
textUpperCaseIntent.putExtra  
    (ExampleIntentService.INPUT_TEXT, inputText);  
  
startService(textUpperCaseIntent);
```

Initialize an intent in the traditional way by giving that the `IntentService` class you just made and put the input text that you want to convert in upper case. The extra data that is given to this intent is done by the `Intent.putExtra(String, String)` method. The last step is to start the service with this intent. We will use the `startService()` method and not the typical `startActivity()` method because we use `startActivity` for starting activities through intents in Android.

Pending Intents

Pending Intents are the intents which give token to other applications or you may call it a foreign application that may access your intent permission to run the predefined piece of code. This way, many other applications such as Alarm Manager and Calendar may use your application in order to execute their tasks.

Pending Intents are not run instantly; rather, they are run when some other activity wants it to run. Pending Intent is a reference that is maintained by the system so that it can be used at a later stage. That means that even if the application that contains the Pending Intent ends, another application can still use the context until `cancel()` is called over that intent.



To perform a broadcast via a Pending Intent, use `PendingIntent.getBroadcast()`.

The Pending Intents can be launched via three methods, `getActivity(Context context, int requestCode, Intent intent, int flags)`, `getBroadcast(Context context, int requestCode, Intent intent, int flags)`, and `getService(Context context, int requestCode, Intent intent, int flags)`. In order to have a view of Pending Intents and how it is made and used in an Android Application, you can proceed to the next section which deals with the implementation.

How to make Pending Intents work?

This section deals with the implementation and explanation of how Pending Intents work. In order to have a good understanding of this, we advise you to read the previously mentioned definition so that you understand it better.

In this example, we will show you how to make an application in which the user can input time (in seconds) into the `editText` field (in seconds) after which an alarm will go off and Android's Alarm Manager will make an alarm which will be played accordingly.

To understand more, take a look at the following code:

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

int i = Integer.parseInt(text.getText().toString());

Intent intent = new Intent(MainActivity.this, MyBroadcastReceiver.
class);

PendingIntent pendingIntent = PendingIntent.getBroadcast(
MainActivity.this, 234324243, intent, 0);

AlarmManager alarmManager = (AlarmManager) getSystemService(ALARM_
SERVICE);

alarmManager.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() +
(i * 1000), pendingIntent);

Toast.makeText(MainActivity.this, "Alarm set in " + i + " seconds",
Toast.LENGTH_SHORT).show();
```

The piece of code written previously can be inserted into any event which can be a button to get the input value in the `EditText` field and process it with `Pending Intents`. The list of steps that is required to understand the previous code are as follows:

1. Get the edit text from the layout file and make an object name `text` which holds the current state of that widget.
2. The integer variable `i` will hold the input value in the edit text which will be obtained by `text.getText().toString()`.
3. Create an explicit intent with a `BroadcastReceiver` class as the Intent's target class (which we will make after completing this).
4. In order to initiate the `Pending Intent`, we use `PendingIntent.getBroadcast(Context context, int requestCode, Intent intent, int flags, int, Intent, int)`. More descriptions about this method can be found at <http://developer.android.com/reference/android/app/PendingIntent.html>.
5. Get the system service of Alarm by putting `ALARM_SERVICE` in the `getSystemService()` method and direct it towards an `AlarmManager` object.
6. Set the values of the Alarm Manager by the value stored in `i` and give it the `Pending Intent` which will help it to start (since Alarm Manager is a service of Android).

7. The `alarmManager.set()` method consists of the arguments `int` type, `long triggerMilliSec` (in which you take the current system time and add your variable `i` by converting it into milliseconds) and the `Pending Intent`.
8. Make a toast in order to show the successful completion of the alarm management.

The next step is to make a `Broadcast Receiver` of your choice and implement that receiver. For this, make a `Broadcast Receiver` and override the method `onReceive()`. Take a look at the following code:

```
@Override
public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "Alarm is ringing...",
        Toast.LENGTH_LONG).show();

    Vibrator vibrator = (Vibrator)
        context.getSystemService(Context.VIBRATOR_SERVICE);
    vibrator.vibrate(2000);
}
```

This receiver has a toast which will indicate its status of alarming. The next thing is to declare an object of the vibrator kind which can be initiated by calling the `context.getSystemService(Context.VIBRATOR_SERVICE)` method. This method is responsible for returning back the object which will directly influence the physical vibrator of the cell phone. The last step is to start the vibration by calling the `vibrator.vibrate(int)` method.



To play the vibrator, you need to add the permission in the manifest file. You can do it by using the following piece of code:

```
<uses-permission android:name=
    "android.permission.VIBRATE" />
```

In the end, we have to declare this receiver in the `AndroidManifest.xml` file and we can do this simply by using the following piece of code:

```
<receiver android:name="MyBroadcastReceiver" >
</receiver>
```

The previous example is describing the use of pending `Intent` and `Broadcast Receivers` together.

Summary

The summary of the last chapter of this book can be considered as the conclusion of this book. In this chapter, we learned how to implement `IntentService` and `PendingIntents` and their best case scenarios. The `IntentService` feature and its comparison with three most commonly used Android features, such as the `Thread`, `Services`, and `Async Tasks`. Moreover, in this chapter, the example of `Pending Intent` is implemented with the explanation of each step. This chapter can be considered to be an advance version, or rather you may say, advance use of intents which can be done in Android. Keeping in mind that the use of these functionalities is not likely to be used, but under certain cases you have to let them work because there will be no other solution.

Index

Symbols

- <data> tag** 249
- <intent-filter>** 242
- <uses-feature> tag**
 - about 201
 - hardware features 203, 204
 - permission implied features 206, 207
 - software features 205, 206
- <uses-permission> tags**
 - about 201-203
 - hardware features 203, 204
 - permission implied features 206, 207
 - software features 204-206

A

- action** 239
- ACTION_CALL**
 - data, using in 59, 240
- ACTION_EDIT**
 - data, using in 240
- ACTION_HEADSET_PLUG action** 241
- ACTION_REQUEST_ENABLE method** 103
- ACTION_SEND** 213
- action test**
 - about 244
 - writing conventions 244, 245
- ACTION_VIEW**
 - data, using in 240
- Activity.finish() method** 66
- Activity.getContentResolver() method** 92
- activity life cycle flow** 30, 31

- activity_main2.xml file**
 - used, for activity start 69
- activity_main.xml file**
 - about 68
 - used, for activity start 67
 - used, for content sharing 79-83
 - used, for service start 75, 76
 - using, for image choosing 88, 89
- activity_main.xml Layout file** 267
- Activity.registerReceiver() method** 224
- activity, starting through explicit intent**
 - MainActivity.java class 64, 65
 - steps 63
- addProximityAlert() method** 139
- alarmManager.set() method** 287
- Android**
 - about 7
 - App Store, Google Play 10
 - history 11, 12
 - limitations 12
 - name attribute 202
 - overview 7
 - versioning 8, 9
 - versions, diagram 14
- Android 1.5** 9
- Android 1.6** 9
- Android 2.0/2.1** 9
- Android 2.2** 9
- Android 2.3** 9
- Android 3.x** 9
- Android 4.0** 8
- Android 4.1/4.2/4.3** 8

- Android Activities**
 - intent role 35
- Android Activity lifecycle**
 - about 24, 25
 - callback methods 26-29
 - diagram 30
 - flow 30, 31
 - fundamental states 26
- Android Application**
 - about 33
 - building blocks 16
 - intent role 34, 35
- Android camera**
 - intent role 36
- android.content.Intent class, constructors**
 - Intent() 49
 - Intent(Context c, Class<?> cls) 49
 - Intent(intent o) 49
 - Intent(String action) 49
 - Intent(String action, URI uri) 50
- Android Development Tools (ADT) 14**
- android.hardware.bluetooth feature 203**
- android.hardware.camera feature 203**
- android.hardware.camera.flash feature 203**
- android.hardware.location.gps feature 203**
- android.hardware.screen.landscape feature 204**
- android.hardware.screen.portrait feature 204**
- android.hardware.sensor.accelerometer feature 204**
- android.hardware.sensor.compass feature 204**
- android.hardware.sensor.proximity feature 204**
- android.hardware.touchscreen.multitouch feature 204**
- android.hardware.wifi feature 204**
- Android Inc 11**
- Android intent resolution 238**
- Android Intents**
 - about 33
 - activity_main.xml 57
 - activity_two_layout.xml 57
 - AndroidManifest.xml 57
 - code 56
 - implementing, for Activity Navigation 42
 - MainActivity.java 56
 - MySecondActivity.java 57
 - working, with example 50-55
- Android Intents implementation, for Activity Navigation**
 - about 42, 45
 - activity_main.xml 47
 - activity_two_layout.xml 47
 - android.content.Intent class, constructors 49
 - AndroidManifest.xml 48
 - MainActivity.java 46
 - MySecondActivity.java 47
- AndroidManifest.xml file**
 - about 22, 70
 - used, for activity start 69, 71
 - used, for content sharing 86, 87
 - used, for service start 76
- android:mimeType attribute 248**
- Android OS**
 - broadcasting 252
 - broadcast intents 252, 253
 - built-in broadcasts 253-255
 - common features 193
 - evolution 12, 13
 - features 192
- Android OS, common features**
 - accessibility 197
 - background services 199
 - communication 196
 - connection, with intents 201
 - connectivity 196
 - content support 198
 - data retrieval 195
 - data storage 195
 - enhanced home screen 200
 - hardware support 199
 - layouts and display 194, 195
 - media support 198
 - multilanguage applications 200
 - multitasking 199
 - multitouch 197
- Android OS features**
 - about 192
 - versus components 193

- Android phones**
 - components 98
 - features 98
- android:required attribute** 202
- Android Services**
 - intent role 38
- android.software.app_widgets feature** 204
- Android Software Development Kit (SDK)** 14
- android.software.home_screen feature** 204
- android.software.input_methods feature** 204
- android.software.live_wallpaper feature** 204
- Android Studio**
 - about 14, 15
 - features 15, 16
 - limitations 16
- assets folder** 18
- Async Task**
 - about 274
 - use case 274

B

- BatteryLowActivity.java class** 257
- BatteryLowReceiver class** 257
- BatteryLowReceiver.java file** 256, 257
- bean class** 171
- BitmapFactory.decodeStream() method** 92
- Bluetooth**
 - about 99
 - APIs 99
 - used, for communication 110
 - using, through intents 99, 100, 101, 105, 110
- BluetoothAdapter class** 100
- BluetoothAdapter.getDefaultAdapter() method** 101
- Bluetooth adapter state**
 - pairing visibility 109, 110
 - pairing visibility modes, monitoring 110
 - tracking 105
 - tracking, with BluetoothStateReceiver.java file 108
 - tracking, with MainActivity.java file 106, 107

- Bluetooth API**
 - about 100
 - classes 100
- Bluetooth API classes**
 - BluetoothAdapter 101
 - BluetoothDevice 100
- Bluetooth app**
 - turning on 101
 - turning on, with AndroidManifest.xml file 104, 105
 - turning on, with MainActivity.java file 102, 103
- Bluetooth components** 96
- BluetoothDevice.getAddress() method** 100
- BluetoothDevice.getBondState() method** 100
- Bluetooth transfer**
 - intent role 36
- broadcast**
 - built-in types 253
- broadcast intent.** *See* broadcast
- Broadcast Receiver**
 - intent role 38
- BroadcastReceiver class** 108, 269
- broadcasting**
 - in Android OS 252
- building blocks, Android application**
 - about 16
 - coding components 18
 - library components 23
 - media components 18
 - referencing components 23
 - XML components 19
- built-in broadcasts**
 - android.intent.action.ACTION_POWER_CONNECTED 254
 - android.intent.action.ACTION_POWER_DISCONNECTED 254
 - android.intent.action.BATTERY_LOW 254
 - android.intent.action.BOOT_COMPLETED 254
 - android.intent.action.DEVICE_STORAGE_LOW 254
 - android.intent.action.NEW_OUTGOING_CALL 254
 - android.intent.action.SCREEN_OFF 254
 - android.intent.action.SCREEN_ON 254

- android.media.VIBRATE_SETTING_CHANGED 254
 - android.net.wifi.WIFI_STATE_CHANGED 254
 - android.provider.Telephony.SMS_RECEIVED 254
 - Bundle.get() method 113**
 - button.setOnClickListener() method 89, 118, 169**
- C**
- callback methods, Activity life cycle 26-29**
 - category constants**
 - CATEGORY_BROWSABLE 241
 - CATEGORY_GADGET 241
 - CATEGORY_HOME 241
 - CATEGORY_LAUNCHER 241
 - CATEGORY_PREFERENCE 241
 - category test**
 - about 246
 - launcher activity, setting up 247
 - cellular component 97**
 - class parameter 62**
 - Cloud to Device Messaging (C2DM) 196**
 - coding components 18**
 - communication**
 - components 99
 - component 96, 193**
 - component name**
 - setting 238
 - content sharing, with implicit intents**
 - about 79
 - activity_main.xml file 80, 83
 - AndroidManifest.xml file 86, 87
 - app, registering 82, 83
 - MainActivity.java class 80-85
 - contentView property 234**
 - context parameter 62**
 - Context.sendBroadcast() method 223**
 - Context.startService() method 264**
 - Cupcake 9**
 - custom broadcasts**
 - receiving 270
 - sending 267
 - sending, with activity_main.xml Layout file 267
 - sending, with AndroidManifest.xml file 270
 - sending, with CustomReceiver.java file 269
 - sending, with MainActivity.java file 268
 - CustomReceiver.java file 269**
- D**
- Dalvik Debug Monitor Server (DDMS) 228**
 - data**
 - about 240
 - transferring, need for 143
 - using, in ACTION_CALL 240
 - using, in ACTION_EDIT 240
 - using, in ACTION_VIEW 240
 - data test**
 - <data> tag, representing 249, 250
 - about 248
 - important points 249
 - data transfer**
 - Android Bundles 158
 - between activities 145
 - example 144
 - in explicit intents 146
 - intent role 35
 - methods 146
 - need for 143
 - Parcelable() method, using 160
 - putExtras() method, using 146
 - Serializable, using 172
 - to implicit intents 186
 - data transfer, putExtras() method used**
 - about 146, 147
 - putParcelable() 157
 - putSerializable() 157
 - data transfer, transferring to implicit intents**
 - about 186
 - call, making 190
 - e-mail, sending 189
 - map, viewing 187
 - webpage, opening 188
 - Donut 9**

E

Eclair 9

e-mail posts

intent role 37, 38

Enter Data button 35, 181

environmental sensors 98

explicit intents

about 61, 62, 238

activity_main.xml file 67-69, 75, 76

AndroidManifest.xml file 69-71, 76

MainActivity.java class 64, 65

SecondActivity.java class 66

ServiceDemoActivity.java class 74, 75

ServiceExample.java class 72, 73

used, for activity starting 63-71

used, for service starting 72-76

using, in Android application 63

explicit intents, using in Android application 63, 72

F

feature 96, 193

findViewById() function 183

findViewById() method 66, 81, 118, 153, 154, 183, 268

fragments 25

Froyo 9

G

geo-location 97

geomagnetic field component 97

getAction() method 239

getApplicationContext() method 65

getAssets() function 18

getBooleanExtra() method 140

getContext() method 46

getExtras().get() method 123

getExtras() method 113, 226

getIntent() function 184

getIntent().getExtras() function 158

getIntent() method 85, 170

getParcelable() function 170

getSerializable() method 185

getState() method 114

getStringArrayListExtra() method 133

getSystemService() method 139, 235, 286

getText() method 184

Ginger Bread 9

Global Positioning System. *See* GPS

Google

official IDE 14

Google Cloud Messaging (GCM) 196

Google Play 11

GPS 97

GPS Sensor

intent role 36

H

hardware features

android.hardware.bluetooth 203

android.hardware.camera 203

android.hardware.camera.flash 203

android.hardware.location.gps 203

android.hardware.screen.landscape 204

android.hardware.screen.portrait 204

android.hardware.sensor.accelerometer 204

android.hardware.sensor.compass 204

android.hardware.sensor.proximity 204

android.hardware.touchscreen.multitouch 204

android.hardware.wifi 204

Honey Comb 9

I

Ice Cream Sandwich 8

image capturing

activity_main.xml file, using 121

AndroidManifest.xml file, using 125-127

MainActivity.java file, using 122-124

image selection, with implicit intents

about 88

activity_main.xml file 88, 89

MainActivity.java class 90-92

implicit intents

about 77, 78

data transfer, transferring to 186

used, for content sharing 78-87

used, for image selecting 88-92

using, in Android application 78, 88

- incomingPersonObj.getFirstname()**
 - method 170
- in.readObject() method 174**
- in.readString() method 171**
- intent action constants**
 - ACTION_BATTERY_LOW 239
 - ACTION_CALL 239
 - ACTION_EDIT 239
 - ACTION_HEADSET_PLUG 239
 - ACTION_MAIN 239
 - ACTION_SCREEN_ON 239
 - ACTION_SYNC 239
- intent actions**
 - ACTION_BATTERY_LOW 59
 - ACTION_CALL 59
 - ACTION_EDIT 59
 - ACTION_HEADSET_PLUG 59
 - ACTION_MAIN 59
 - ACTION_SCREEN_ON 59
 - ACTION_SYNC 59
 - ACTION_TIMEZONE_CHANGED 59
- Intent.chooser() method 81**
- intent components**
 - CATEGORY_BROWSABLE 58
 - CATEGORY_GADGET 58
 - CATEGORY_LAUNCHER 58
- Intent() constructor 49**
- Intent(Context c, Class<?> cls)**
 - constructor 49
- intent filter**
 - test components 243
- Intent filters 237, 241, 242**
- intent-filter tag 249**
- Intent.getAction() method 85**
- Intent.getData() method 92**
- Intent.getIntExtra() method 110**
- Intent.getStringExtra() method 85**
- Intent.getType() method 85**
- Intent(intent o) constructor 49**
- Intent object**
 - about 237
 - categorizing 238
- Intent object, categories**
 - action 239
 - category 240
 - component name 238
 - data 240
 - extras 241
- intent.putExtras() method 158**
- intent role**
 - in Android Activities 35
 - in Android Application 34, 35
 - in Android Services 38
 - in Bluetooth transfer 36
 - in Broadcast Receiver 38
 - in data transfer 35
 - in e-mail posts 37
 - in GPS Sensor 36
 - in Mobile Calls 36
 - in sending SMS/MMS 36
 - in social network posts 37, 38
 - in Status Bar 39
 - in time zones 39
 - in Wi-Fi transfer 35
- intents**
 - about 61
 - coding component 40
 - delivery, confirming 222
 - explicit 61
 - implicit 61
 - proximity alerts 139, 140
 - receiving 225
 - structure 58
 - technical overview 40
 - used, for image capturing 120-127
 - used, for making phone calls 213-217
 - used, for MMS sending 221, 222
 - used, for notifications sending 229
 - used, for SMS sending 218-220
 - used, for speech recognition 130-134
 - used, for video recording 127-129
 - XML component 41
- Intent Service**
 - about 273, 274
 - code 277-282
 - example 282-285
 - fake notification, generating 276
 - implementation 275
 - triggers 275
 - use case 274
- Intent.setAction() method 268**
- Intent(String action) constructor 49**
- Intent(String action, URI uri) constructor 50**

intent structure

- about 58
- actions 59
- component 58
- data 59
- extras 60

Internet connectivity status

- checking 111

Internet connectivity status check

- AndroidManifest.xml file, using 114, 116, 117
- NetworkStatusReceiver.java file, using 112-114

J

Jelly Bean 8

L

layout folder 20

library components 23

LogCat 77

long-distance conversation call

- making, intents used 213, 214

low-battery device state

- detecting 255
- detecting, with AndroidManifest.xml file 258, 259
- detecting, with BatteryLowActivity.java class 257
- detecting, with BatteryLowReceiver.java file 256, 257

M

MAC address 100

MainActivity.java class

- used, for activity start 64, 65
- used, for content sharing 80-85
- using, for image choosing 90-92

MainActivity.java file 268

media components

- about 18
- assets folder 18
- res folder 19

MediaPlayer class 129

menu folder

- about 20
- Context Menus 20
- Custom Menus 20
- Options Menus (with Action bar) 20
- Pop-up Menus 20

message delivery

- confirming 222-224

messaging 93

MMS

- sending, intents used 221, 222

Mobile Calls

- intent role 36

mobile components

- about 96
- Bluetooth components 96
- cellular component 97
- geo-location 97
- geomagnetic field component 97
- GPS 97
- sensor components 97
- Wi-Fi components 96

motion components 137

motion sensors 98

Multimedia Messaging Services. *See* MMS

multiple intent filters

- handling 243

N

nameText.getText() method 169

NetworkInfo.getTypeName() method 114

newBundle.putString() function 158

Notification class 230

NotificationManager class 230

NotificationManager.notify() method 235

notifications

- sending, intents used 229

notifications, sending with intents

- about 229
- Notification class 230
- notification forms 230
- notification layout 230-235
- NotificationManager class 230

O

ObjectInputStream class 172
onActivityResult() method 91, 92, 135, 136
onBind() method 73
onClickListener() method 66, 154
onClick() method 74
onCreate() method 31, 66, 81, 91, 132
onHandleIntent() method 283
onInit() method 137
onPause() method 31
onReceive() method 112, 226, 287
onRestart() method 31
onResume() method 31
onStartCommand() method 73, 265
onStart() method 29

P

Parcelable() method
implementing 160
readymade tutorial, implementing 160, 167
used, for data transfer 160
Parcelable() method implementation
about 168
Activity1.java class 168
Activity2.java class 170
AndroidManifest.xml file 171
layout_activity1.xml file 170
layout_activity2.xml file 170
Person.java class 171
Pending Intents
about 285
making, it work 285-287
permission implied features
android.permission.ACCESS_COARSE_ LOCATION 206
android.permission.ACCESS_FINE_ LOCATION 206
android.permission.ACCESS_WIFI_ STATE 207
android.permission.BLUETOOTH 206
android.permission.CALL_ PHONE 207
android.permission.CAMERA 206
android.permission.PROCESS_ OUTGO- ING_ CALLS 207
android.permission.READ_ SMS 207

android.permission.RECIEVE_ SMS 207
android.permission.RECORD_ AUDIO 206
android.permission.SEND_ SMS 207
android.permission.WRITE_ SMS 207

phone calls

making, intents used 213, 214

PhoneRebootCompletedReceiver.java file 264

phone reboot-completed state

detecting 263
detecting, with AndroidManifest.xml
file 266
detecting, with PhoneRebootCompletedRe-
ceiver.java file 264
detecting, with TempService.java file 264,
265

phone screen on/off state

detecting 260
detecting, with AndroidManifest.xml file
262, 263
detecting, with ScreenOnOffReceiver.java
file 260, 261

pickImage() method 91

position sensors 98

Protocol Data Unit (PDU) 225-229

proximity alerts

about 138
intents, using 139, 140

putExtra() method 128, 211

putExtras() function 146

putExtras() method

Activity1.java class 153, 154
Activity2.java class 155
AndroidManifest.xml file 156
implementing 147
main_first.xml file 156
main_second.xml file 156
readymade tutorial, implementing 147, 152
used, for data transfer 146

R

Recognize button 134

RecognizerIntent class 130

referencing components 23

registerReceiver() method 106, 107, 140, 262, 284

RelativeLayout 20
RemoteView 233
requestCode parameter 103
res folder 19
R.java file 23

S

ScreenOnOffReceiver class 261
ScreenOnOffReceiver.java file 260, 261
SecondActivity.java class
 about 66
 used, for activity start 66
SEND action
 used, for sharing 207-213
Send Broadcast Intent 270
sendBroadcast() method 268, 283
sendTextMessage() method 224
sensor components
 about 97
 environmental sensors 98
 motion sensors 98
 position sensors 98
Serializable
 about 172
 example 173, 174
 implementing 175, 181, 182
 used, for data transfer 172
Serializable code
 about 182
 Activity1.java class 183, 184
 Activity2.java class 184, 185
 AndroidManifest.xml file 186
 layout_activity1.xml file 185
 layout_activity2.xml file 186
 Person.java class 185
Service
 about 274
 triggers 275
 use case 274
ServiceDemoActivity.java class
 used, for service start 74, 75
ServiceExample.java class
 used, for service start 72, 73
setAction() method 253
setContentView() function 184

setContentView() method 132, 183, 185
setImageBitmap method 124
setLatestEventInfo method 233
setOnClickListener() method 46, 65, 74, 154
setResult() function 57
setText() method 133, 155
setType() method 209
setVideoURI() method 129
SharedPreferences method 195
Short Messaging Services. *See* SMS
show() method 47
SMS
 sending, intents used 218-221
SmsManager class 225
SmsManager.sendTextMessage() method 222
SmsMessage.createFromPdu() method 227
SmsMessage.getMessageBody() method 227
SmsMessage.getOriginatingAddress() method 227
SmsMessage.getTimestampMillis() method 227
SmsMessage object 225
SMS messages
 Protocol Data Unit (PDU) 225-229
 receiving 225
 SmsManager class, using 225
 SmsMessage object 225
SMS/MMS sending
 intent role 36
social network posts
 intent role 37, 38
software features
 android.software.app_widgets 204
 android.software.home_screen 204
 android.software.input_methods 204
 android.software.live_wallpaper 204
speak() method 137
speech recognition
 activity_main.xml file, using 131
 intents, using 130
 MainActivity.java file, using 132-134
startActivityForResult() method 56, 91-93, 123, 133-135
startActivity() method 65, 93, 246

startService() method 75, 285

Status Bar

intent role 39

status text 230

stopService() method 75

T

TempService.java file 264, 265

test components, intent filter

Action test 244

category test 246

data test 248, 249

Text-To-Speech. *See* TTS conversion

TextToSpeech class 136

TextView 20

Thread

about 274

About 274

triggers 275

use case 274

time zones

intent role 39

TTS conversion

intents, using 134-137

U

Universal Resource Identifier (URI) 214

unregisterReceiver() method 258

Uri.parse() method 188

URI (Universal Resource Identifier) 59

User ID (UID) 113

V

values folder 21

vibrator.vibrate(int) method 287

VideoView tag 127

W

Wi-Fi

using, through intents 111, 117

Wi-Fi components 96

Wi-Fi Settings App

opening 117

opening, with activity_main.xml file 117,
118

opening, with MainActivity.java file 118,
119

Wi-Fi transfer

intent role 36

World Magnetic Model 97

writeToParcel() method 171

X

XML

using 20

XML components

about 19

AndroidManifest.xml 22

layout folder 20

menu folder 20

values folder 21



Thank you for buying Learning Android Intents

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

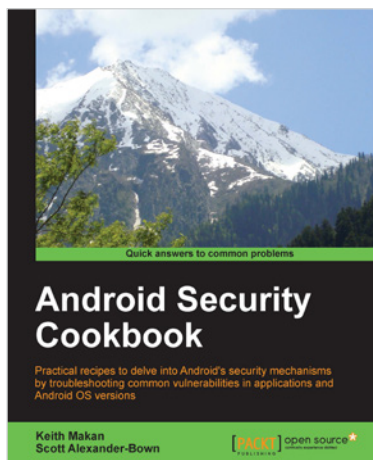


Android Development Tools for Eclipse

ISBN: 978-1-78216-110-3 Paperback: 144 pages

Set up, build, and publish Android projects quickly using Android Development Tools for Eclipse

1. Build Android applications using ADT for Eclipse
2. Generate Android application skeleton code using wizards
3. Advertise and monetize your applications



Android Security Cookbook

ISBN: 978-1-78216-716-7 Paperback: 350 pages

Practical recipes to delve into Android's security mechanisms by troubleshooting common vulnerabilities in applications and Android OS versions

1. Analyze the security of Android applications and devices, and exploit common vulnerabilities in applications and Android operating systems
2. Develop custom vulnerability assessment tools using the Drozer Android Security Assessment Framework
3. Reverse-engineer Android applications for security vulnerabilities
4. Protect your Android application with up to date hardening techniques

Please check www.PacktPub.com for information on our titles

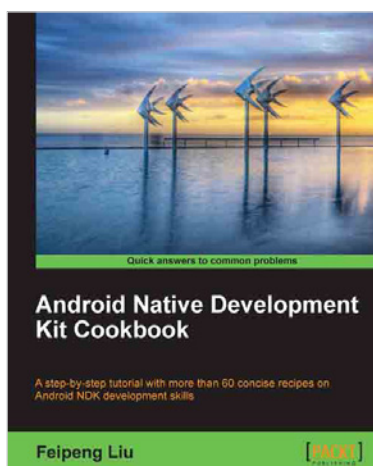


Android 4: New Features for Application Development

ISBN: 978-1-84951-952-6 Paperback: 166 pages

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

1. Learn new APIs in Android 4
2. Get familiar with the best practices in developing Android applications
3. Step-by-step approach with clearly explained sample codes



Android Native Development Kit Cookbook

ISBN: 978-1-84969-150-5 Paperback: 346 pages

A step-by-step tutorial with more than 60 concise recipes on Android NDK development skills

1. Build, debug, and profile Android NDK apps
2. Implement part of Android apps in native C/C++ code
4. Optimize code performance in assembly with Android NDK

Please check www.PacktPub.com for information on our titles