

Lin Guo

# The First Line of Code

Android Programming with Kotlin

*Translated by*  
Litao Shen



人民邮电出版社  
POSTS & TELECOM PRESS



Springer

# The First Line of Code

Lin Guo

# The First Line of Code

Android Programming with Kotlin



Springer

Lin Guo  
STCA WebXT Edge Mobile  
Microsoft  
Suzhou, Jiangsu, China

ISBN 978-981-19-1799-8              ISBN 978-981-19-1800-1 (eBook)  
<https://doi.org/10.1007/978-981-19-1800-1>

Jointly published with Posts & Telecom Press  
The print edition is not for sale in Mainland China. Customers from Mainland China please order the print book from: Posts & Telecom Press

© Posts & Telecom Press 2022

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publishers, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publishers nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publishers remain neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.  
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721,  
Singapore

# Preface

Phew, what a huge project!

Allow me to introduce myself. I am Lin Guo and am an Android developer from China. I started Android development in 2010 and became an Android GDE in 2020. Currently, I work for Microsoft.

*The First Line of Code* is my only book and incorporates my years of experience in Android development. This is the best-selling Android book in China and has helped tens of thousands of Chinese readers to start their Android development journey.

However, I have never thought that this book could become international and I appreciate the recognition from Springer.

I want to give special thanks to Litao Shen who is the translator of this book. He is a software engineer in Meta Inc. Although we have never met personally, we became friends because of this book. He mentioned that this book helped him to prepare the interview for Facebook back then in the feedback for this book. Thus, when I started the exploration for an English translator for this book, he immediately accepted this offer and challenge. Thanks for your hard work in such a short time.

Now, you are reading the newest version of *The First Line of Code*. It covers most of the important topics of Android and Kotlin. I hope you can read this book carefully as more learning means more happiness. Enjoy it!

## Target Audience

This book is not obscure and goes from easy to more complicated. It can help both beginners and professionals. You do not need to know anything about Android or Kotlin; however, some fundamental knowledge about Java helps smooth the learning curve as all the codes in this book are written in Kotlin which is based on Java.

You can start with any chapter in this book based on your condition as each chapter is self-contained. If you are a beginner, it is recommended to start from

Chap. 1 to ensure a smooth learning experience. If you already grasp some fundamental Android knowledge, you can pick whichever chapters that interest you. I recommend that do not miss the practice and Kotlin class section at the end of each chapter.

## Content Summary

As aforementioned, this book systematically covers essential Android development knowledge and ensures that the difficulty level is in ascending order. There are 15 chapters in this book which cover four main components, UI, fragment, data persistence, multimedia, networks, architecture, etc. for Android. For Kotlin, this book covers fundamental syntaxes, tips, high-order functions, generics, coroutine, DSL, etc. To make sure you can use them collectively, at the end of this book, we will create a weather app, build and publish an open-source library.

Besides these, Chaps. 6, 9, 12, and 15 cover Git knowledge and you cannot miss them if you want to learn Git.

Each chapter in this book is relatively isolated and independent, thus you can also use this book as reference material.

## Learning Resources

Download link: <https://file.ituring.com.cn/Original/2004fe62f809edc265f6>

Hope you all enjoy the reading!

Suzhou, Jiangsu, China  
8 February 2022

Lin Guo

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Your First Line of Android Code . . . . .</b>                      | <b>1</b>  |
| 1.1      | Android: An Overview . . . . .  | 2         |
| 1.1.1    | Android System Architecture . . . . .                                 | 2         |
| 1.1.2    | Released Versions . . . . .   | 3         |
| 1.1.3    | What's Special for Android Development? . . . . .                     | 4         |
| 1.2      | Set Up Development Environment Step by Step . . . . .                 | 5         |
| 1.2.1    | Prerequisite Tools . . . . .  | 5         |
| 1.2.2    | Set Up the Environment . . . . .                                      | 6         |
| 1.3      | Creating Your First Android Project . . . . .                         | 6         |
| 1.3.1    | Creating HelloWorld Project . . . . .                                 | 7         |
| 1.3.2    | Starting Emulator . . . . .   | 8         |
| 1.3.3    | Running HelloWorld . . . . .  | 10        |
| 1.3.4    | Analyzing Your First Android Project . . . . .                        | 11        |
| 1.3.5    | Resources in a Project . . . . .                                      | 23        |
| 1.3.6    | File of build.gradle . . . . .  | 24        |
| 1.4      | Mastering the Use of Logging Tools . . . . .                          | 28        |
| 1.4.1    | Using Android Log Tool . . . . .                                      | 28        |
| 1.4.2    | Log Vs println() . . . . .  | 29        |
| 1.5      | Summary . . . . .   | 31        |
| <b>2</b> | <b>Explore New Language: A Quick Introduction to Kotlin . . . . .</b> | <b>33</b> |
| 2.1      | Introduction to Kotlin . . . . .                                      | 34        |
| 2.2      | How to Run Kotlin Code . . . . .                                      | 35        |
| 2.3      | The Foundation of Programming: Variables and Functions . . . . .      | 38        |
| 2.3.1    | Variables . . . . .   | 38        |
| 2.3.2    | Functions . . . . .   | 41        |
| 2.4      | Flow Control . . . . .  | 44        |
| 2.4.1    | if Statement . . . . .  | 44        |
| 2.4.2    | when Statement . . . . .  | 46        |
| 2.4.3    | Loop Statement . . . . .  | 49        |

|          |   |           |
|----------|---|-----------|
| 2.5      | Object-Oriented Programming . . . . .                     | 52        |
| 2.5.1    | Class and Object . . . . .                                | 52        |
| 2.5.2    | Inheritance and Constructor Function . . . . .            | 54        |
| 2.5.3    | Interface . . . . .                                       | 59        |
| 2.5.4    | Data Class and Singleton . . . . .                        | 62        |
| 2.6      | Lambda Expression . . . . .                               | 65        |
| 2.6.1    | Creation and Iteration of Collection . . . . .            | 66        |
| 2.6.2    | Functional APIs of Collections . . . . .                  | 69        |
| 2.6.3    | Java Functional API . . . . .                             | 73        |
| 2.7      | Null Safety . . . . .                                     | 76        |
| 2.7.1    | Nullable Type . . . . .                                   | 76        |
| 2.7.2    | Nullability Check Tools . . . . .                         | 78        |
| 2.8      | Kotlin Tricks . . . . .                                   | 82        |
| 2.8.1    | String Interpolation . . . . .                            | 82        |
| 2.8.2    | Function Default Arguments . . . . .                      | 83        |
| 2.9      | Summary . . . . .   | 86        |
| <b>3</b> | <b>Start with the Visible: Explore Activity . . . . .</b> | <b>87</b> |
| 3.1      | What Is Activity? . . . . .                               | 87        |
| 3.2      | Activity Fundamentals . . . . .                           | 87        |
| 3.2.1    | Manually Creating Activity . . . . .                      | 88        |
| 3.2.2    | Creating and Mounting the Layout . . . . .                | 90        |
| 3.2.3    | Registering in AndroidManifest File . . . . .             | 92        |
| 3.2.4    | Using Toast in Activity . . . . .                         | 95        |
| 3.2.5    | Using Menu in Activity . . . . .                          | 98        |
| 3.2.6    | Destroying an Activity . . . . .                          | 101       |
| 3.3      | Using Intent to Communicate Between Activities . . . . .  | 102       |
| 3.3.1    | Explicit Intent . . . . .                                 | 103       |
| 3.3.2    | Implicit Intent . . . . .                                 | 106       |
| 3.3.3    | More on Implicit Intent . . . . .                         | 108       |
| 3.3.4    | Passing Data to the Next Activity . . . . .               | 112       |
| 3.3.5    | Return Data to the Last Activity . . . . .                | 113       |
| 3.4      | Activity Lifecycle . . . . .                              | 115       |
| 3.4.1    | Back Stack . . . . .                                      | 115       |
| 3.4.2    | Activity States . . . . .                                 | 116       |
| 3.4.3    | Activity Lifecycle . . . . .                              | 117       |
| 3.4.4    | Explore the Lifecycle of Activity . . . . .               | 118       |
| 3.4.5    | Recycling Activity . . . . .                              | 125       |
| 3.5      | Launch Mode of Activity . . . . .                         | 126       |
| 3.5.1    | Standard . . . . .  | 126       |
| 3.5.2    | singleTop . . . . .                                       | 127       |
| 3.5.3    | singleTask . . . . .                                      | 129       |
| 3.5.4    | singleInstance . . . . .                                  | 131       |
| 3.6      | Activity Best Practices . . . . .                         | 134       |
| 3.6.1    | Identifying the Current Activity . . . . .                | 134       |

|          |   |            |
|----------|---|------------|
| 3.6.2    | Exiting the App from Anywhere . . . . .                       | 135        |
| 3.6.3    | Best Practice to Start Activity . . . . .                     | 137        |
| 3.7      | Kotlin Class: Standard Functions and Static Methods . . . . . | 138        |
| 3.7.1    | Standard Functions: with, run, and apply . . . . .            | 139        |
| 3.7.2    | Define Static Methods . . . . .                               | 142        |
| 3.8      | Summary . . . . .   | 145        |
| <b>4</b> | <b>Everything About UI Development . . . . .</b>              | <b>147</b> |
| 4.1      | How to Create UI? . . . . .                                   | 147        |
| 4.2      | Common UI Widgets . . . . .                                   | 148        |
| 4.2.1    | TextView . . . . .  | 148        |
| 4.2.2    | Button . . . . .  | 151        |
| 4.2.3    | EditText . . . . .  | 153        |
| 4.2.4    | ImageView . . . . .   | 158        |
| 4.2.5    | ProgressBar . . . . .   | 160        |
| 4.2.6    | AlertDialog . . . . .   | 164        |
| 4.3      | Three Basic Layouts . . . . .                                 | 166        |
| 4.3.1    | LinearLayout . . . . .  | 166        |
| 4.3.2    | RelativeLayout . . . . .                                      | 172        |
| 4.3.3    | FrameLayout . . . . .   | 176        |
| 4.4      | Customize the Widgets . . . . .                               | 178        |
| 4.4.1    | Include Layout . . . . .                                      | 180        |
| 4.4.2    | Create Customized Widgets . . . . .                           | 182        |
| 4.5      | ListView . . . . .  | 185        |
| 4.5.1    | Simple Demonstration of ListView . . . . .                    | 185        |
| 4.5.2    | Customize ListView UI . . . . .                               | 186        |
| 4.5.3    | Optimize the Efficiency of ListView . . . . .                 | 189        |
| 4.5.4    | Click Event in ListView . . . . .                             | 192        |
| 4.6      | RecyclerView . . . . .  | 194        |
| 4.6.1    | Basics About RecyclerView . . . . .                           | 194        |
| 4.6.2    | Scroll Horizontally and Waterfall Flow Layout . . . . .       | 197        |
| 4.6.3    | RecyclerView Click Event . . . . .                            | 202        |
| 4.7      | Best Practice to Build UI . . . . .                           | 203        |
| 4.7.1    | Create 9-Patch Image . . . . .                                | 204        |
| 4.7.2    | Build Beautiful Chat User Interface . . . . .                 | 208        |
| 4.8      | Kotlin Class: Lateinit and Sealed Cass . . . . .              | 213        |
| 4.8.1    | Lateinit Variables . . . . .                                  | 214        |
| 4.8.2    | Optimization with Sealed Class . . . . .                      | 216        |
| 4.9      | Summary and Comment . . . . .                                 | 218        |
| <b>5</b> | <b>Support Phones and Tablets with Fragment . . . . .</b>     | <b>221</b> |
| 5.1      | What Is Fragment? . . . . .                                   | 221        |
| 5.2      | How to Use Fragment . . . . .                                 | 223        |
| 5.2.1    | Basic Use of Fragment . . . . .                               | 223        |
| 5.2.2    | Add Fragment Dynamically . . . . .                            | 227        |

|          |   |            |
|----------|---|------------|
| 5.2.3    | Back Stack for Fragment . . . . .                                   | 230        |
| 5.2.4    | Interaction Between Fragment and Activity . . . . .                 | 230        |
| 5.3      | Lifecycle of Fragment . . . . .                                     | 231        |
| 5.3.1    | Fragment State and Callbacks . . . . .                              | 231        |
| 5.3.2    | Experiment with Fragment Lifecycle . . . . .                        | 232        |
| 5.4      | Dynamically Load Layout . . . . .                                   | 236        |
| 5.4.1    | Use Qualifier . . . . .   | 236        |
| 5.4.2    | Use Smallest-Width Qualifier . . . . .                              | 237        |
| 5.5      | Fragment Best Practice: A Basic News App . . . . .                  | 240        |
| 5.6      | Kotlin Class: Extension Function and Operator Overloading . . . . . | 249        |
| 5.6.1    | Extension Function . . . . .  | 249        |
| 5.6.2    | Operator Overloading . . . . .                                      | 252        |
| 5.7      | Summary and Comment . . . . .                                       | 256        |
| <b>6</b> | <b>Broadcasts in Details . . . . .</b>                              | <b>259</b> |
| 6.1      | Introduction to Broadcast Mechanism . . . . .                       | 259        |
| 6.2      | Receive System Broadcast . . . . .                                  | 261        |
| 6.2.1    | Dynamically Register BroadcastReceiver for Time Change . . . . .    | 261        |
| 6.2.2    | Open App After Booting with Static Receiver . . . . .               | 262        |
| 6.3      | Send Customized Broadcast . . . . .                                 | 267        |
| 6.3.1    | Send Normal Broadcast . . . . .                                     | 267        |
| 6.3.2    | Send Ordered Broadcast . . . . .                                    | 270        |
| 6.4      | Best Practice of Broadcast: Force Logout . . . . .                  | 274        |
| 6.5      | Kotlin Class: Higher-Order Function . . . . .                       | 280        |
| 6.5.1    | Define Higher-Order Function . . . . .                              | 280        |
| 6.5.2    | Inline Functions . . . . .  | 286        |
| 6.5.3    | Noinline and Crossinline . . . . .                                  | 289        |
| 6.6      | Git Time: The First Look of Version Control Tools . . . . .         | 293        |
| 6.6.1    | Git Installation . . . . .  | 293        |
| 6.6.2    | Create Code Repository . . . . .                                    | 293        |
| 6.6.3    | Commit Local Code . . . . .   | 295        |
| 6.7      | Summary and Comment . . . . .                                       | 296        |
| <b>7</b> | <b>Data Persistence . . . . .</b>                                   | <b>297</b> |
| 7.1      | Introduction to Data Persistence . . . . .                          | 297        |
| 7.2      | Persisting Through File . . . . .                                   | 298        |
| 7.2.1    | Persisting Data in File . . . . .                                   | 298        |
| 7.2.2    | Read Data from File . . . . .                                       | 301        |
| 7.3      | SharedPreferences . . . . .   | 304        |
| 7.3.1    | Save Data in SharedPreferences . . . . .                            | 304        |
| 7.3.2    | Read Data from SharedPreferences . . . . .                          | 307        |
| 7.3.3    | Implement Remembering Password . . . . .                            | 308        |
| 7.4      | SQLite Database . . . . .   | 311        |
| 7.4.1    | Create Database . . . . .   | 312        |

|          |   |            |
|----------|---|------------|
| 7.4.2    | Upgrade Database . . . . .                                    | 317        |
| 7.4.3    | Add Data . . . . .  | 322        |
| 7.4.4    | Update Data . . . . .   | 324        |
| 7.4.5    | Delete Data . . . . .   | 326        |
| 7.4.6    | Query Data . . . . .  | 328        |
| 7.4.7    | Use SQL . . . . .   | 331        |
| 7.5      | SQLite Database Best Practice . . . . .                       | 333        |
| 7.5.1    | Transaction . . . . .   | 333        |
| 7.5.2    | Best Practice to Upgrade Database . . . . .                   | 335        |
| 7.6      | Kotlin Class: Application of Higher-Order Function . . . . .  | 338        |
| 7.6.1    | Simplify Use of SharedPreferences . . . . .                   | 338        |
| 7.6.2    | Simplify Use of ContentValues . . . . .                       | 340        |
| 7.7      | Summary and Comment . . . . .                                 | 343        |
| <b>8</b> | <b>Share Data Between Apps with ContentProvider . . . . .</b> | <b>345</b> |
| 8.1      | Introduction to ContentProvider . . . . .                     | 345        |
| 8.2      | Runtime Permissions . . . . .                                 | 346        |
| 8.2.1    | Android Runtime Permissions in Depth . . . . .                | 346        |
| 8.2.2    | Request Permission at Runtime . . . . .                       | 349        |
| 8.3      | Access Data in Other Apps . . . . .                           | 354        |
| 8.3.1    | Basic Use of ContentResolver . . . . .                        | 354        |
| 8.3.2    | Read System Contact . . . . .                                 | 358        |
| 8.4      | Create Your Own ContentProvider . . . . .                     | 362        |
| 8.4.1    | Create ContentProvider . . . . .                              | 362        |
| 8.4.2    | Share Data Between Apps . . . . .                             | 368        |
| 8.5      | Kotlin Class: Generics and Delegate . . . . .                 | 377        |
| 8.5.1    | Basic Use of Generics . . . . .                               | 377        |
| 8.5.2    | Class Delegation and Delegated Properties . . . . .           | 379        |
| 8.5.3    | Implement Lazy Function . . . . .                             | 382        |
| 8.6      | Summary and Comment . . . . .                                 | 384        |
| <b>9</b> | <b>Enrich Your App with Multimedia . . . . .</b>              | <b>387</b> |
| 9.1      | Run Application on Phone . . . . .                            | 387        |
| 9.2      | Notification . . . . .  | 388        |
| 9.2.1    | Create Notification Channel . . . . .                         | 389        |
| 9.2.2    | Basic Use of Notification . . . . .                           | 392        |
| 9.2.3    | Advanced Topics in Notification . . . . .                     | 399        |
| 9.3      | Camera and Album . . . . .                                    | 403        |
| 9.3.1    | Take Photos with Camera . . . . .                             | 404        |
| 9.3.2    | Select Images from Album . . . . .                            | 408        |
| 9.4      | Play Multi-Media Files . . . . .                              | 411        |
| 9.4.1    | Play Audio . . . . .  | 412        |
| 9.4.2    | Play Video . . . . .  | 416        |
| 9.5      | Kotlin Class: Use Infix to Improve Readability . . . . .      | 420        |
| 9.6      | Git Time: Advanced Topics in Version Control . . . . .        | 423        |

|           |  |            |
|-----------|--|------------|
| 9.6.1     | Ignore Files . . . . .   | 423        |
| 9.6.2     | Inspect Modified Content . . . . .                             | 424        |
| 9.6.3     | Revert the Uncommitted Changes . . . . .                       | 426        |
| 9.6.4     | Check Commit History . . . . .                                 | 427        |
| 9.7       | Summary and Comment . . . . .                                  | 428        |
| <b>10</b> | <b>Work on the Background Service . . . . .</b>                | <b>431</b> |
| 10.1      | What Is Service? . . . . .                                     | 431        |
| 10.2      | Android Multithreading . . . . .                               | 432        |
| 10.2.1    | Basic Use of Thread . . . . .                                  | 432        |
| 10.2.2    | Update UI in Worker Thread . . . . .                           | 433        |
| 10.2.3    | Async Message Handling Mechanism . . . . .                     | 435        |
| 10.2.4    | Use AsyncTask . . . . .  | 437        |
| 10.3      | Basic Use of Service . . . . .                                 | 440        |
| 10.3.1    | Define a Service . . . . .                                     | 440        |
| 10.3.2    | Start and Stop Service . . . . .                               | 442        |
| 10.3.3    | Communication Between Activity and Service . . . . .           | 445        |
| 10.4      | Service Life Cycle . . . . .                                   | 450        |
| 10.5      | More Techniques on Service . . . . .                           | 450        |
| 10.5.1    | Use Foreground Service . . . . .                               | 451        |
| 10.5.2    | Use IntentService . . . . .                                    | 453        |
| 10.6      | Kotlin Class: Advanced Topics in Generics . . . . .            | 456        |
| 10.6.1    | Reify Generic Types . . . . .                                  | 457        |
| 10.6.2    | Application of Reified Type in Android . . . . .               | 459        |
| 10.6.3    | Covariance and Contravariance . . . . .                        | 461        |
| 10.6.4    | Contravariance . . . . .                                       | 464        |
| 10.7      | Summary and Comment . . . . .                                  | 467        |
| <b>11</b> | <b>Exploring New World with Network Technologies . . . . .</b> | <b>469</b> |
| 11.1      | WebView . . . . .  | 469        |
| 11.2      | Use HTTP to Access Network . . . . .                           | 471        |
| 11.2.1    | Use HttpURLConnection . . . . .                                | 472        |
| 11.2.2    | Use OkHttp . . . . .   | 476        |
| 11.3      | Parse XML Data . . . . .                                       | 478        |
| 11.3.1    | Pull Parser . . . . .  | 481        |
| 11.3.2    | SAX Parser . . . . .   | 484        |
| 11.4      | Parse JSON Data . . . . .                                      | 487        |
| 11.4.1    | JSONObject . . . . .   | 488        |
| 11.4.2    | GSON . . . . .   | 489        |
| 11.5      | Implementing Network Callback . . . . .                        | 491        |
| 11.6      | The Best Network Lib: Retrofit . . . . .                       | 494        |
| 11.6.1    | Basic Use of Retrofit . . . . .                                | 494        |
| 11.6.2    | Process Complex Interface Address . . . . .                    | 499        |
| 11.6.3    | Best Practice for Retrofit Builder . . . . .                   | 502        |

|           |  |            |
|-----------|--|------------|
| 11.7      | Kotlin Class: Use Coroutine for Performant Concurrent App . . . . .    | 504        |
| 11.7.1    | Basic Use of Coroutine . . . . .                                       | 504        |
| 11.7.2    | More on Coroutine Scope Builder . . . . .                              | 510        |
| 11.7.3    | Simplifying Callback with Coroutine . . . . .                          | 514        |
| 11.8      | Summary and Comment . . . . .  | 517        |
| <b>12</b> | <b>Best UI Experience: Material Design in Action . . . . .</b>         | <b>519</b> |
| 12.1      | What Is Material Design? . . . . .                                     | 519        |
| 12.2      | Toolbar . . . . .  | 520        |
| 12.3      | Navigation Drawer . . . . .  | 527        |
| 12.3.1    | DrawerLayout . . . . .   | 527        |
| 12.3.2    | NavigationView . . . . .   | 530        |
| 12.4      | FloatingActionButton and Snackbar . . . . .                            | 534        |
| 12.4.1    | FloatingActionButton . . . . .   | 535        |
| 12.4.2    | Snackbar . . . . .   | 539        |
| 12.4.3    | CoordinatorLayout . . . . .  | 540        |
| 12.5      | CardView Layout . . . . .  | 542        |
| 12.5.1    | MaterialCardView . . . . .   | 542        |
| 12.5.2    | AppBarLayout . . . . .   | 548        |
| 12.6      | Pull to Refresh . . . . .  | 551        |
| 12.7      | Collapsible Toolbar . . . . .  | 553        |
| 12.7.1    | CollapsingToolbarLayout . . . . .                                      | 553        |
| 12.7.2    | Optimizing Using of System Status Bar . . . . .                        | 560        |
| 12.8      | Kotlin Class: Creating Utils . . . . .                                 | 564        |
| 12.8.1    | Find Max and Min in $N$ Numbers . . . . .                              | 565        |
| 12.8.2    | Simplifying Use of Toast . . . . .                                     | 566        |
| 12.8.3    | Simplifying Use of Snackbar . . . . .                                  | 568        |
| 12.9      | Git Time: Advanced Topics in Version Control . . . . .                 | 570        |
| 12.9.1    | Branch . . . . .   | 570        |
| 12.9.2    | Work with Remote Repo . . . . .  | 572        |
| 12.10     | Summary and Comment . . . . .  | 573        |
| <b>13</b> | <b>High-Quality Developing Components: Exploring Jetpack . . . . .</b> | <b>575</b> |
| 13.1      | Introduction to Jetpack . . . . .                                      | 575        |
| 13.2      | ViewModel . . . . .  | 576        |
| 13.2.1    | ViewModel Basics . . . . .   | 577        |
| 13.2.2    | Pass Param to ViewModel . . . . .                                      | 579        |
| 13.3      | Lifecycles . . . . .   | 583        |
| 13.4      | LiveData . . . . .   | 587        |
| 13.4.1    | LiveData Basics . . . . .  | 587        |
| 13.4.2    | map and switchMap . . . . .  | 590        |
| 13.5      | Room . . . . .   | 596        |
| 13.5.1    | Use Room to CRUD . . . . .   | 596        |
| 13.5.2    | Room Database Upgrade . . . . .  | 603        |
| 13.6      | WorkManager . . . . .  | 606        |

|           |   |            |
|-----------|---|------------|
| 13.6.1    | WorkManager Basics . . . . .                                    | 607        |
| 13.6.2    | Handling Complex Task with WorkManager . . . . .                | 608        |
| 13.7      | Kotlin Class: Use DSL to Construct Specific Syntax . . . . .    | 610        |
| 13.8      | Summary and Comment . . . . .                                   | 617        |
| <b>14</b> | <b>Keep Stepping Up: More Skills You Need to Know . . . . .</b> | <b>619</b> |
| 14.1      | Obtaining Context Globally . . . . .                            | 619        |
| 14.2      | Passing Objects with Intent . . . . .                           | 622        |
| 14.2.1    | Serializable . . . . .  | 622        |
| 14.2.2    | Parcelable . . . . .  | 623        |
| 14.3      | Creating Your Own Logging Tool . . . . .                        | 624        |
| 14.4      | Debug Android Apps . . . . .                                    | 626        |
| 14.5      | Dark Mode . . . . .   | 630        |
| 14.6      | Kotlin Class: Conversion Between Java and Kotlin Code . . . . . | 636        |
| 14.7      | Summary . . . . .   | 640        |
| <b>15</b> | <b>Real Project Practice: Creating a Weather App . . . . .</b>  | <b>641</b> |
| 15.1      | Analysis Before Start . . . . .                                 | 641        |
| 15.2      | Git Time: Host Code on GitHub . . . . .                         | 645        |
| 15.3      | Introduction to MVVM . . . . .                                  | 651        |
| 15.4      | Search City Data . . . . .                                      | 655        |
| 15.4.1    | Business Logic Implementation . . . . .                         | 656        |
| 15.4.2    | UI Implementation . . . . .                                     | 661        |
| 15.5      | Display Weather Data . . . . .                                  | 667        |
| 15.5.1    | Business Logic Implement . . . . .                              | 668        |
| 15.5.2    | Implement UI Layer . . . . .                                    | 674        |
| 15.5.3    | Record City Selection . . . . .                                 | 685        |
| 15.6      | Manual Refresh and Switch City . . . . .                        | 688        |
| 15.6.1    | Manual Refreshing Weather . . . . .                             | 688        |
| 15.6.2    | Switching Cities . . . . .                                      | 690        |
| 15.7      | Create App Icon . . . . .                                       | 695        |
| 15.8      | Generate Signed APK . . . . .                                   | 700        |
| 15.8.1    | Generating with Android Studio . . . . .                        | 701        |
| 15.8.2    | Generating with Gradle . . . . .                                | 702        |
| 15.9      | More To Do . . . . .  | 707        |

# Chapter 1

## Your First Line of Android Code



Welcome to the Android World! Android is by now the most popular mobile operating system (OS) in the world, and you can find Android phones wherever you go. But do you know how did Android has become the world's No. 1 mobile OS? Let us take a look at the history of Android.

In October 2003, Andy Rubin and a few others founded the Android Inc. In August 2005, Google acquired Android Inc. which was only 22 months old at that time. Andy Rubin stayed and continued to be responsible for the Android project. After years of R&D, Google released the first version of Android OS in 2008. However, ever since then Android has faced lots of backlashes. Steve Jobs insisted that Android was an iOS knockoff that stole the ideas of iOS, and he threatened to destroy Android at all costs. Android OS is based on Linux, but it was removed from the Linux kernel main branch by Linux team in 2010. Since all the apps in Android were developed with Java in the early days, Oracle also sued Google for intelligence infringement...

However, all of these couldn't stop Android from taking up the market share rapidly. Google made Android an open OS which means that any OEM or person can get the source code of Android OS for free, and can use and customize the OS freely. Samsung, HTC, Motorola, Sony-Ericsson, etc., all released their Android phones. After being released for 2 years, Android had already overtaken Nokia Symbian which had been the leader of the smartphone mobile OS market for more than 10 years. At that time, millions of new Android devices were activated every day. Today, Android has more than 70% of global smartphone OS market share.

Now you must feel so excited and are eager to be an Android developer. Just think that about 7 out of every 10 individuals' phone can run the app you write. Is there anything else that can be more exciting than this? Now, let's start the journey of learning Android development, and I will guide you how to be an excellent Android developer step by step.

## 1.1 Android: An Overview

More than 20 versions of Android have been released so far, and Google has built a holistic ecosystem during the past years. OEMs, developers, and users all play an important role in this ecosystem, and they work together to push the evolution of Android. Developers make a crucial part of this ecosystem, since no matter how good the OS is, if there are not enough apps available, users wouldn't feel like using the OS. Who will use an OS that does not support Facebook, Messenger, or Instagram? Furthermore, Google Play is the marketplace for Android apps, and if you can make high-quality apps that can attract users, then you can get good return from your apps. If your app is popular, you can even become an independent developer to support yourself or even create a startup!

Now let's take a look at the Android OS through the lens of an Android developer. It can be very boring to focus on the theoretical topics. Thus, I will cover only the important topics that you will use for developing Android apps.

### 1.1.1 *Android System Architecture*

Let's take a look at the system architecture to understand how Android OS works. Roughly, Android can be divided into four layers: Linux kernel layer, system lib layer, application framework layer, and the applications layer.

#### 1. Linux Kernel Layer

Android is based on Linux kernel, and this layer provides the drivers for the various hardware of Android devices, for example, display driver, audio driver, camera driver, blue-tooth driver, Wi-Fi driver, battery management, etc.

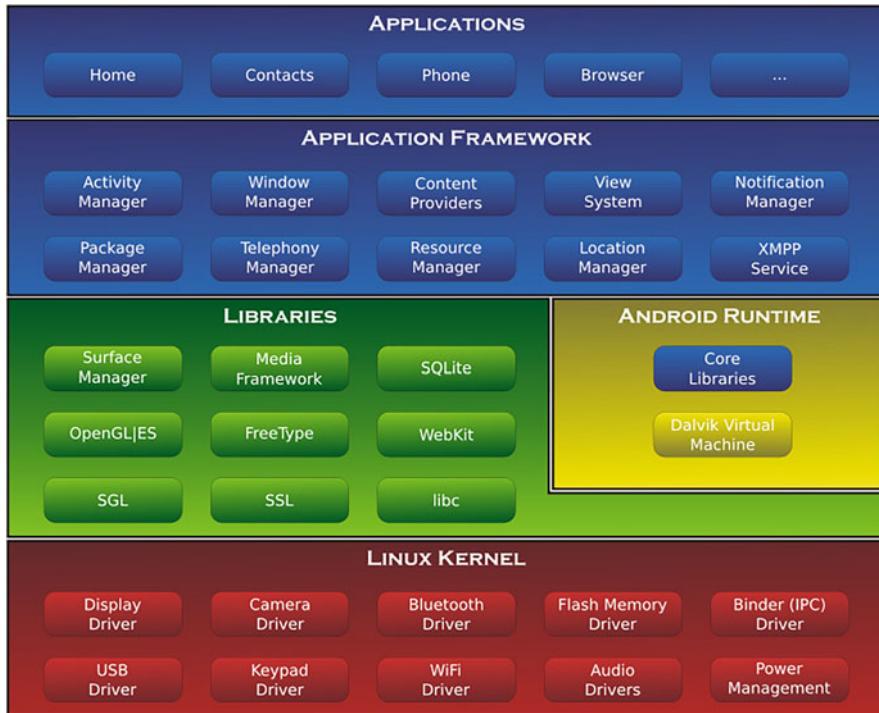
#### 2. System Libraries Layer

This layer provides primal support to the system with some C/C++ libraries. For example, SQLite provides database support for the database, OpenGL/ES provides support for 3D graphics, Webkit lib provides support for browser kernel, etc.

Android runtime is also in this layer, which provides some core libraries to allow developers to develop Android apps with Java. The Dalvik virtual machine is also in the runtime (replaced with ART after version 5.0), and it allows the apps to run in their own processes and have their own virtual machine instances. Compared with Java virtual machine (JVM), Dalvik and ART have been specifically optimized for mobile devices which usually have less memory and computing power.

#### 3. Application Framework Layer

This layer provides all kinds of APIs that developers use to build the apps. The Android system apps utilize these APIs.



**Fig. 1.1** Android system architecture. (Source: Clipped from official documentary page)

#### 4. Applications Layer

All the apps are belonging to this layer, for example, system apps like Contact, Message, and the apps you downloaded from Google Play, and of course the apps you're going to build.

See Fig. 1.1 to get have a better understanding of the system architecture of Android.

##### 1.1.2 *Released Versions*

In September 2008, Google released the first version of Android—Android 1.0. In the following years, Google kept updating Android at an amazing speed. v2.1, v2.2, and v2.3 helped Android to grab a huge market share. In February 2011, Google released Android 3.0, which was dedicated to tablet and was one of the few versions that didn't get traction. Soon, in October of the same year, Google released Android 4.0 which stopped treating phone and tablet differently and could be used in both phone and tablet. In 2014, Google released Android 5.0 which claimed to have the

**Table 1.1** List of different versions of Android

| Version number | Codename           | API | Market share |
|----------------|--------------------|-----|--------------|
| 2.3.3–2.3.7    | Gingerbread        | 10  | 0.3%         |
| 4.0.3–4.0.4    | Ice Cream Sandwich | 15  | 0.3%         |
| 4.1.x          | Jelly Bean         | 16  | 1.2%         |
| 4.2.x          |                    | 17  | 1.5%         |
| 4.3            |                    | 18  | 0.5%         |
| 4.4            | KitKat             | 19  | 6.9%         |
| 5.0            | Lollipop           | 21  | 3%           |
| 5.1            |                    | 22  | 11.5%        |
| 6.0            | Marshmallow        | 23  | 16.9%        |
| 7.0            | Nougat             | 24  | 11.4%        |
| 7.1            |                    | 25  | 7.8%         |
| 8.0            | Oreo               | 26  | 12.9%        |
| 8.1            |                    | 27  | 15.4%        |
| 9              | Pie                | 28  | 10.4%        |

most drastic changes in the history of Android. That version used ART to replace Dalvik virtual machine which greatly boosted the speed of apps. The concept of Material Design also came from that version to help optimize the UI design. Google also released OS that can be used in specific devices such as Android wear, Android Auto, and Android TV at the same time. After that, Google accelerated the speed of releasing Android OS updates and would release a new version every year. By 2019, Android was already at v10.0 which was also the latest version when I wrote this book. Check <https://developer.android.com/about/versions> for the newest updates. Combine this with Fig. 1.1 to get a better understanding (Table 1.1).

From Table 1.1, we can see that v5.0 and above already have more than 85% of Android market and this number is only going to increase, thus the apps we build will only target v5.0 and above.

### 1.1.3 What's Special for Android Development?

Let us first look at what does Android system provide for us to develop quality apps.

#### 1. Four Main Components

The four main components of Android application development are Activity, Service, BroadcastReceiver, and ContentProvider. Activity is used to display the UI of the app, and everything you can see are hosted in Activity. Unlike Activity, Service is invisible, instead, it can run in the background even if the user exits the app. BroadcastReceiver allows your app to receive the broadcast messages from apps like Phone and Message. Of course, your app can also send broadcast messages. ContentProvider can be used to share data between different apps.

For example, you need the help of ContentProvider to read the contacts in the system Contact app.

## 2. Rich System Widgets

Android provides plenty of system widgets to help developers build beautiful UIs. Of course, you can always create your own widgets to meet your special requirement.

## 3. SQLite Database

Android is embedded with SQLite database, which is a lightweight and superfast relational database. It supports standard SQL syntax and can be accessed using APIs provided by the system which make CRUD operations super-easy.

## 4. Powerful Multimedia Support

Android provides very powerful multimedia support like music, videos, voice recording, taking photos, etc. You can use the supported APIs to make miscellaneous apps.

With all the things Android has to offer, there is really no need to worry that you can't build the app you want in Android.

## 1.2 Set Up Development Environment Step by Step

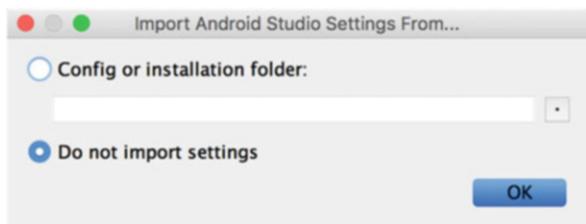
An old saying goes like this—to do a good job, an artisan needs the best tools. It's not a good idea to use notepad to build Android apps. Instead, a powerful IDE can boost your productivity dramatically. So, let us see how to set up the environment step by step.

### 1.2.1 *Prerequisite Tools*

You need the following to build Android apps.

- **JDK.** JDK is the Java language development kit which contains the Java runtime, basic libraries, etc.
- **Android SDK.** Android SDK is the Android development kit provided by Google; we need to use the APIs in the SDK to build Android app.
- **Android Studio.** Eclipse used to be the most popular IDE to develop Android apps which every Java developer should be very familiar with. To develop Android app in Eclipse, you just need to install the ADT plugin. However, in 2013, Google released Android Studio. It is not a plugin but a real IDE with a suite of tools to help you and it's way more powerful than Eclipse. Thus, we will use Android Studio exclusively in this book.

**Fig. 1.2** Select “Do not import settings”



### 1.2.2 Set Up the Environment

You don't need to download those tools one by one since Google already put everything together into a package. Go to the official website at <https://developer.android.google.cn/studio> to download the tools.

After downloading the installer, you can just click “Next” all the way to finish the process.

After installation, when you first open the IDE, Android Studio will ask to import the previous settings. It is not needed since it's our first time to install it, thus select “Do not import settings,” as shown in Fig. 1.2.

Click “OK” to go to the wizard screen as shown in Fig. 1.3.

Click “Next” to start configuration as shown in Fig. 1.4.

Here, you can choose from “Standard” and “Custom.” The “Standard” option will use the default configuration which is more convenient, and we will just use “Standard” for now. Click “Next” to go to the screen to select UI theme, as shown in Fig. 1.5.

There are two initial themes for you to choose from and I will just use the default “Light theme” here, click “Next” to finish the configuration as shown in Fig. 1.6.

Now click “Finish” to finish all the configuration steps. After this, Android Studio will try to connect to the network and then download some updates. After updates are done, click “Finish” should open the Android Studio welcome screen as shown in Fig. 1.7.

Now that the development environment has been set up, let us write our first line of Android code!

## 1.3 Creating Your First Android Project

By convention, we should write the Hello World program as our first program, and we will respect this convention in this book.

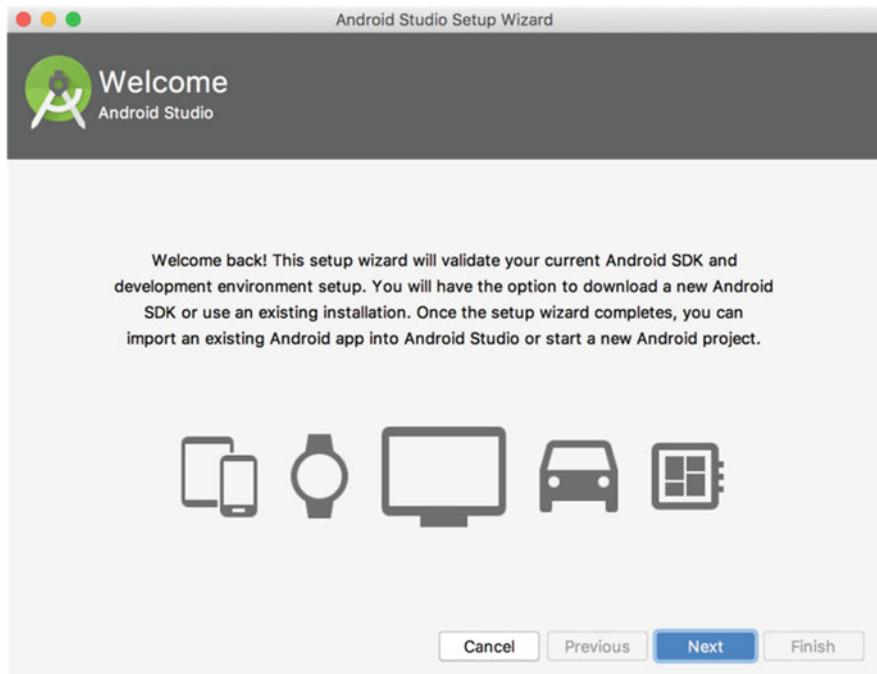


Fig. 1.3 Android Studio configuration screen

### 1.3.1 *Creating HelloWorld Project*

In the Android Studio welcome screen, click Start a new Android Studio project should open a screen to allow you to select project type as shown in Fig. 1.8.

From this screen, we can select project type for phones, tablets, wearable devices, TVs, and even for cars. We will focus on phone and tablet in this book. Android Studio also provides lots of templates which we won't use for now since most of the templates are too complicated for starters and we will start with Empty Activity.

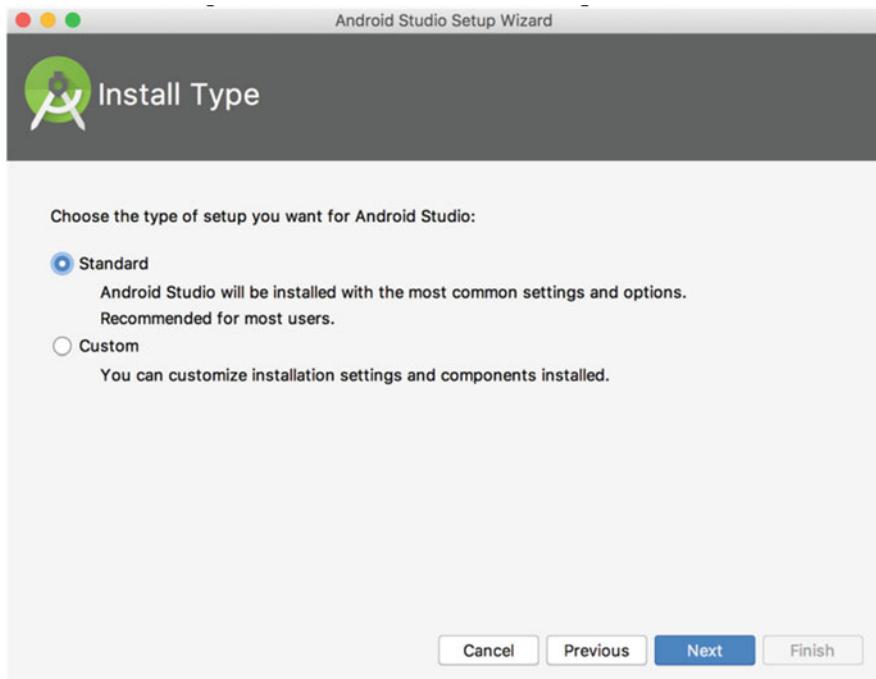
Click “Next” to go to the project configuration screen as shown in Fig. 1.9.

Use HelloWorld for the project Name.

Package name is the name that Android uses to distinguish between different apps, thus we need to make this name unique. Android Studio will help us generate a proper package name based on our app's name and you can change it as you wish.

Save location is where the project will be, and I will use the default selection here.

Language option is important and here it chooses Kotlin by default. Java used to be the language exclusively used to develop Android apps. In 2017, Google introduced Kotlin to the Android land and announced that Kotlin should be the first choice for developers in 2019. Thus, I decided to use Kotlin to write all the code in this book in V3. Don't worry if you don't know anything about Kotlin. I will cover Kotlin extensively in this book besides Android.



**Fig. 1.4** Select Install Type

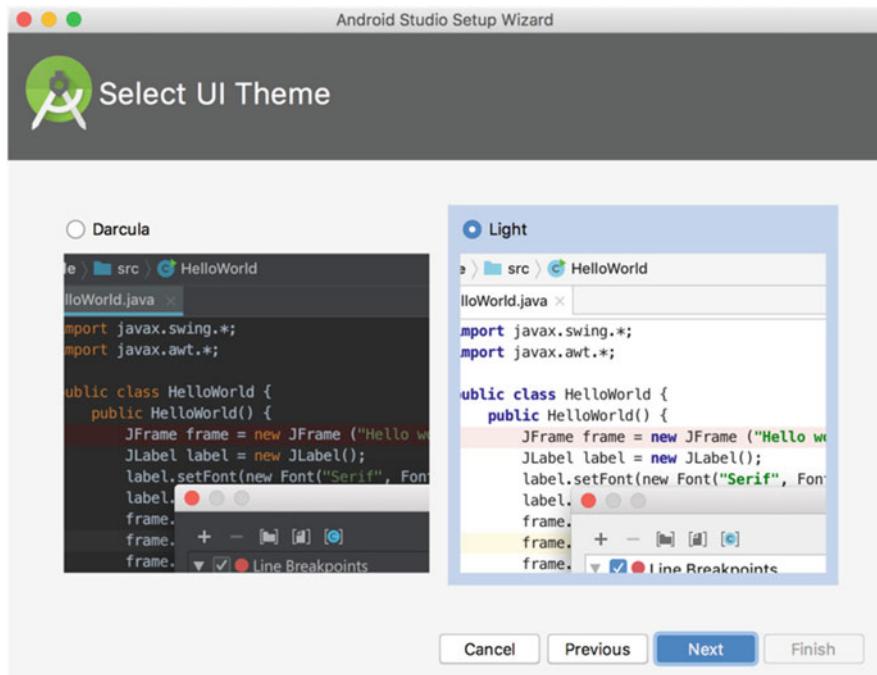
Minimum API level is the earliest version of Android that this project is compatible with. As mentioned earlier that Android 5.0 and above already take more than 85% of the Android market share, we will set the Minimum SDK to be API 21.

The instant run option allows the app code can be pushed dynamically and is outside the scope of this book. AndroidX is in the process of replacing the old Android Support Library and in Android Studio 3.5.2, this option has been selected by default. In later versions of Android Studio this option may be gone since all projects will have to use AndroidX.

Now click “Finish” and the project should successfully get created in a moment as shown in Fig. 1.10.

### 1.3.2 Starting Emulator

You don't need to write any code to be able to run this app because Android Studio already generated everything we need to run the app. But before that, we need something that can load our app which can be an Android phone or emulator. Let's use an emulator for now and if you want to install and run the app in your phone, please refer to Sect. 9.1.



**Fig. 1.5** Select UI Theme

Now let us create an Android emulator. You should find the icons at the top of the toolbar in Android Studio as shown in Fig. 1.11.

The middle button is used to create and start the emulator. Click this button should show a screen like shown in Fig. 1.12.

You should see an empty emulator list and click “Create Virtual Device” to start the creation process as shown in Fig. 1.13.

There are lots of devices we can choose from like phones, tablets, watches, etc. Here let us use Pixel which is my personal favorite. Click “Next” as shown in Fig. 1.14.

Apparently, we want to download the latest Android OS which will require downloading the image first. Click “Download” and after downloading finishes, click “Next” will show a window as shown in Fig. 1.15.

Here we can verify lots of configurations like name of emulator, screen resolution, etc. If there is no special requirement, we can use the default configuration. Click “Finish” and a window, as shown in Fig. 1.16, will show up.

Now we can see the emulator item in the emulator list and clicking the triangle icon in Actions will start the emulator. Emulator will have a booting process like real phones, and after that it should have a window as shown in Fig. 1.17.

Android emulators can largely emulate the real phones and you should be able to use emulator almost like using a real phone. Try it out yourself!

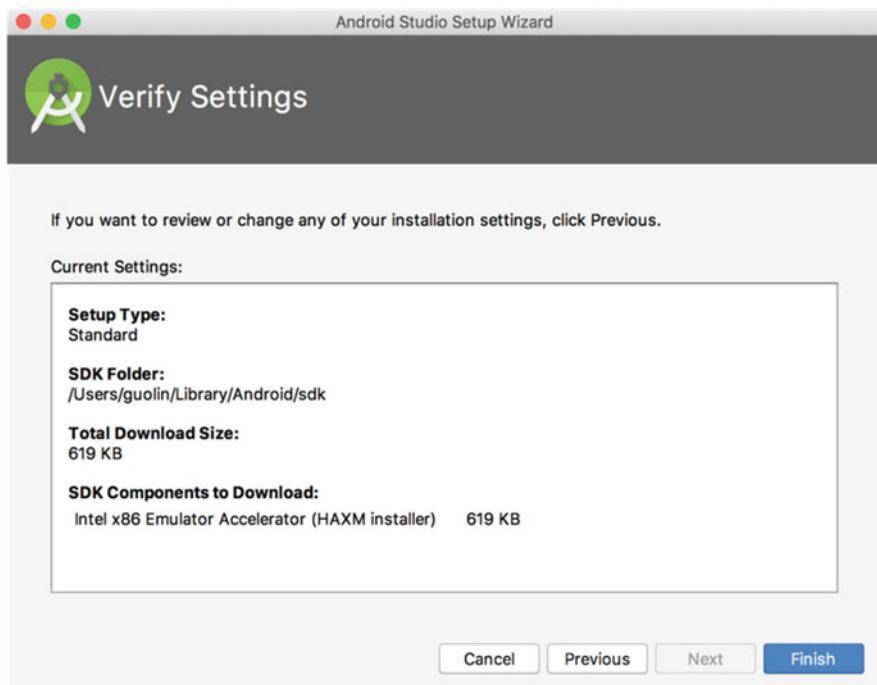


Fig. 1.6 Finish Configuration of Android Studio

### 1.3.3 *Running HelloWorld*

Now let's install and run the *HelloWorld* project in the emulator. At the top of the toolbar, you should see the buttons as shown in Fig. 1.18. Among them, the hammer button is used to compile the project. The first dropdown list is used to select the project to run and the second is for selecting the device which already shows the emulator we just created. The right-most triangle-shaped button is for running the app.

Now click the triangle button and wait for a few seconds until *HelloWorld* gets installed and run which should be like Fig. 1.19.

You will find that *HelloWorld* app has been installed and you can find it from the launcher as shown in Fig. 1.20.

Wait a second! We haven't even written a single line of code and Hello World already showed up! Where is our first line of Android code? Well, Android Studio already generated the code for us, and now let us use this project to understand Android project structure.



**Fig. 1.7** Welcome screen of Android Studio

#### **1.3.4 Analyzing Your First Android Project**

Now in Android Studio, you should be able to see the project structure as shown in Fig. 1.21.

A newly created project will by default use the Android mode for project structure which is not how the project is organized in the disk. This mode has a very clear structure which can help development, but does not help for starters to understand and navigate around. In the dropdown list (Fig. 1.21), choose Project mode (Fig. 1.22) as shown in Fig. 1.23.

An Android project has lots of folders and files inside and you might feel confused about what they are. Now let me explain them one by one to help you understand the project structure.

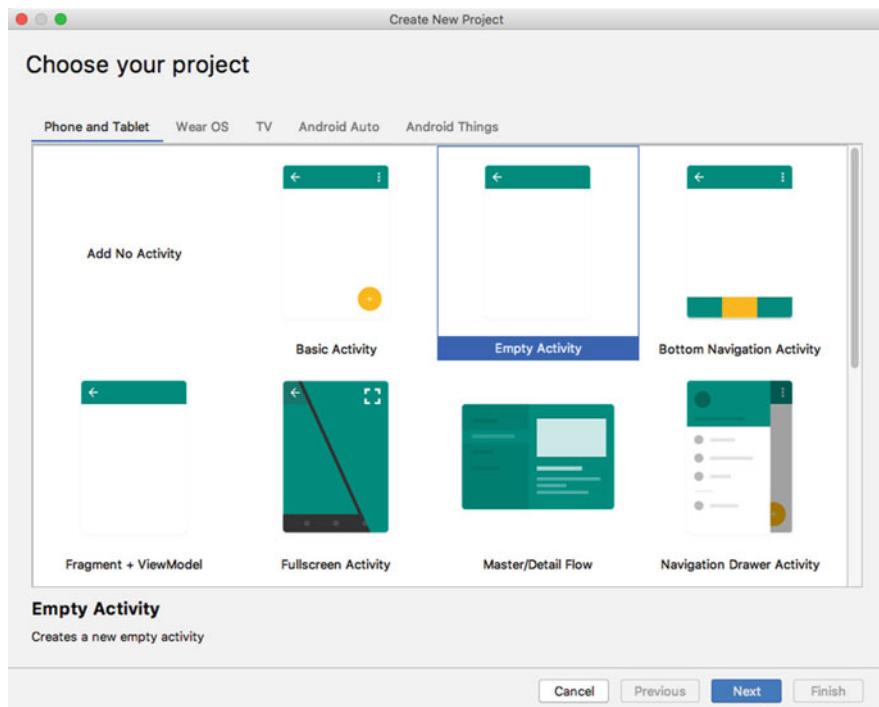
##### 1. .gradle and .idea

Under these two folders are some auto-generated files and we can ignore them for now and don't need to manually edit them.

##### 2. app

All the code files, resource files are under this folder and apparently our work will mostly be in this folder, and we will expand it to cover more details later.

##### 3. build



**Fig. 1.8** Select Project Type Screen

This folder mainly contains the files generated during the compilation process and again you can ignore for now.

#### 4. gradle

This folder has the gradle wrapper config file. Using gradle wrapper can avoid downloading gradle when not necessary. By default, Android Studio will use gradle wrapper and if you want to use offline mode, click File- > Settings- > Build, Execution, Deployment- > Gradle.

#### 5. .gitignore

This file is used to allow version control tool to ignore files which means the changes that applied to these files won't be tracked. We will learn more about this in Chap. 6.

#### 6. build.gradle

This is the global build script which usually doesn't need to be edited and we will cover the gradle script in detail later.

#### 7. gradle.properties

This is the config file for global gradle file which means the properties in this file will affect all the gradle scripts in this project.

#### 8. gradlew and gradlew.bat

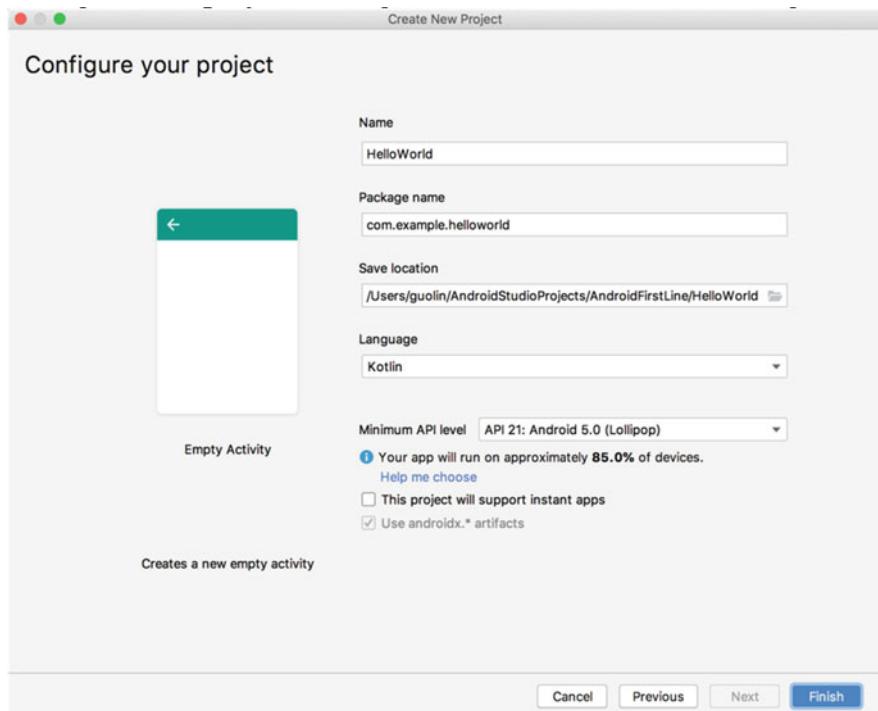


Fig. 1.9 Project Configuration Screen

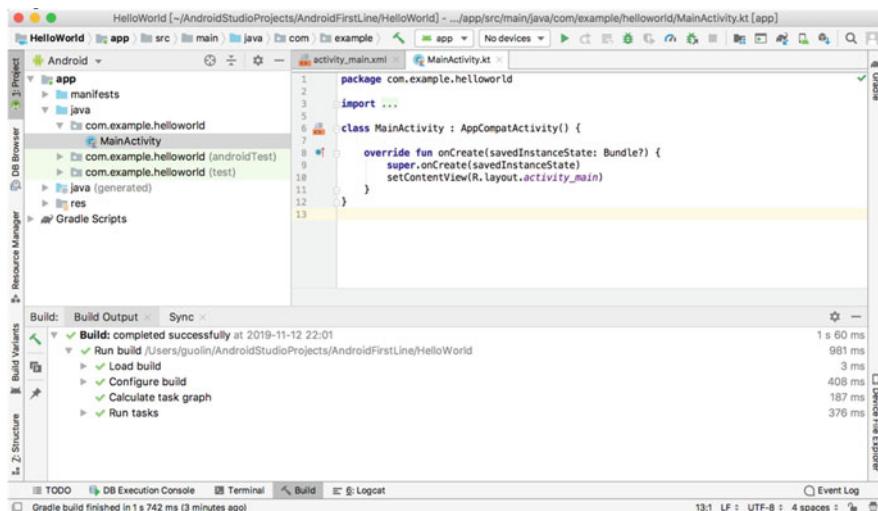
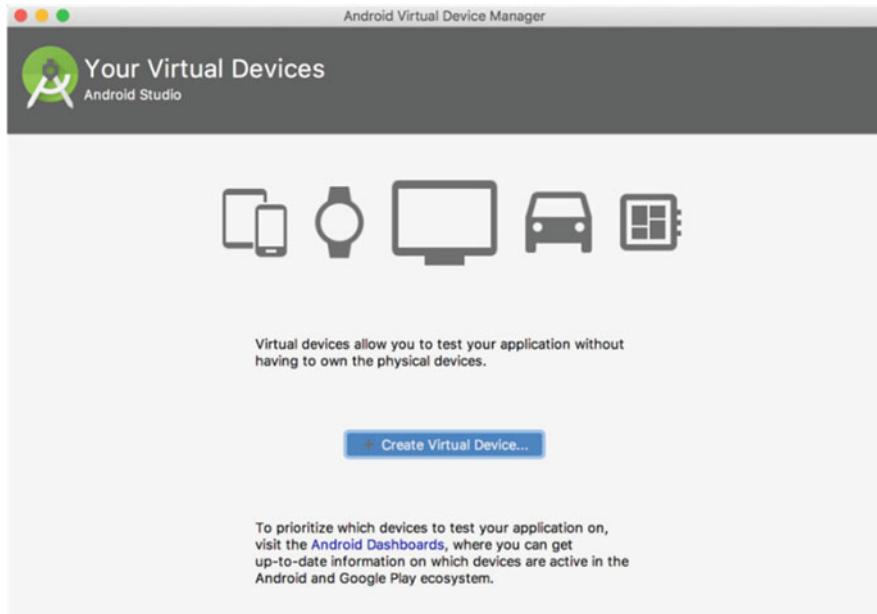


Fig. 1.10 Project created successfully

**Fig. 1.11** Icons at the top toolbar



**Fig. 1.12** Create emulator wizard

These two files are used to execute the gradle command in CLI. gradlew is for Linux and Mac OS. gradle.bat is for Windows.

#### 9. HelloWorld.iml

All the IntelliJ IDEA projects will generate an iml file and since Android Studio is based on IntelliJ IDEA thus there is an iml file in the project which we can ignore.

#### 10. local.properties

This file specifies the path of Android SDK in this machine and usually is auto generated, thus we can ignore it. If the path of Android SDK in your machine somehow gets changed, you can change to the newest path in this file.

#### 11. settings.gradle

This is used to specify all the imported modules. Since there is only one app module, we can only see the app here. Usually, importing modules is automatically done and we don't need to edit this file.

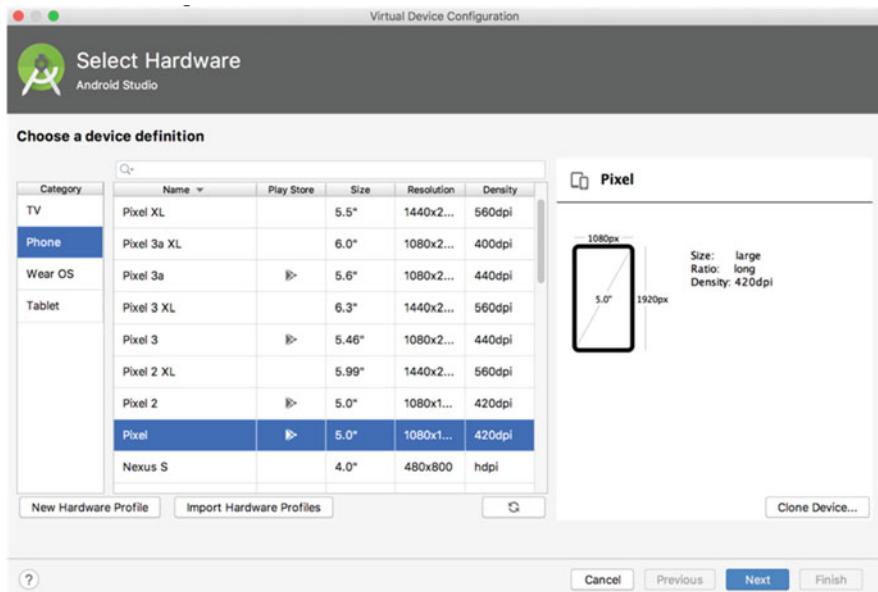


Fig. 1.13 Select the device

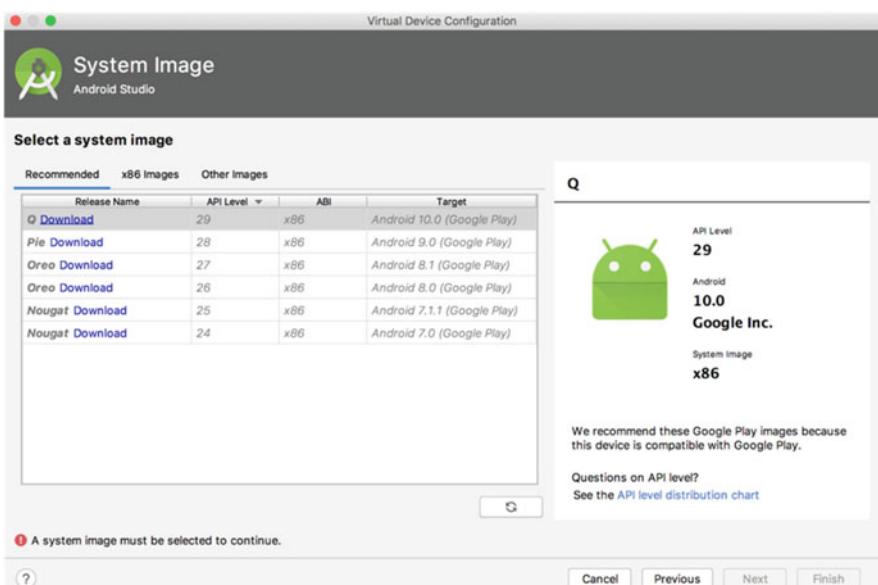


Fig. 1.14 Select the OS version of emulator

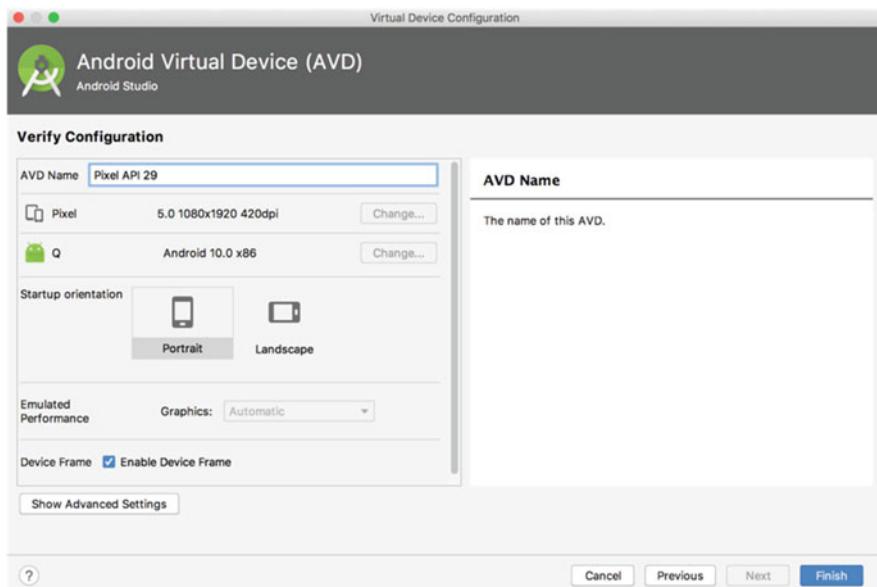


Fig. 1.15 Verify Emulator Configuration



Fig. 1.16 Emulator list

That's everything about the outer layer structure. Most of the directories and files except app directory are generated and we don't need to edit them. Let us expand the app directory which should look like Fig. 1.24.

**Fig. 1.17** Emulator Screen**Fig. 1.18** Buttons in the top toolbar

Next, let us take a deep dive into app directory.

#### 1. build

This folder is like the outer layer build folder and contains generated files during compilation although much more complex. Ignore for now.

#### 2. libs

You should put the third-party jar package in this folder which will be automatically added to the build path of this project.

#### 3. androidTest

The automated test code should reside in this directory.

#### 4. java

**Fig. 1.19** Run the HelloWorld project



Obviously, this is the directory that all of our Java (Kotlin) code will be. Expand the folder, you should see `MainActivity` file which is generated by the IDE.

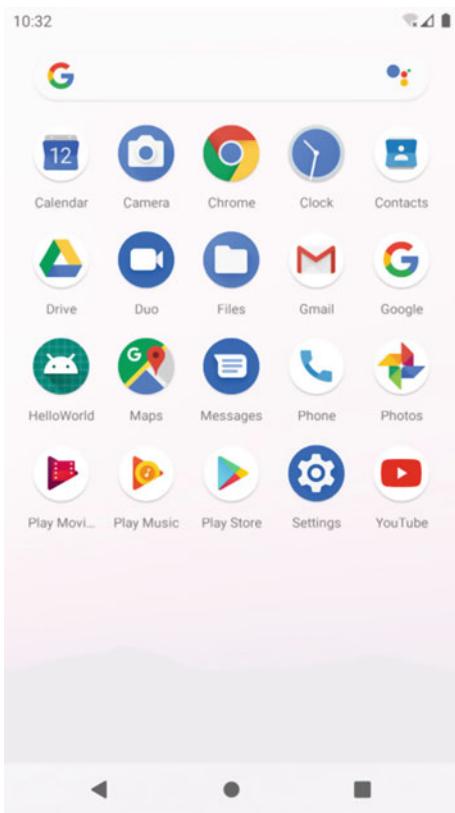
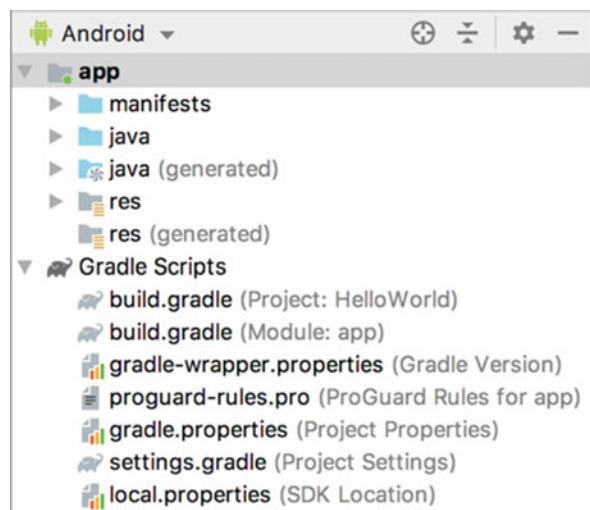
#### 5. res

There will be lots of things under this directory. Basically, all the resources you use for the project like images, layouts, strings, etc. should all be put under this directory. In order to keep them organized, they should be put into their corresponding folders like `drawable` folder for images, `layout` folder for layouts, etc.

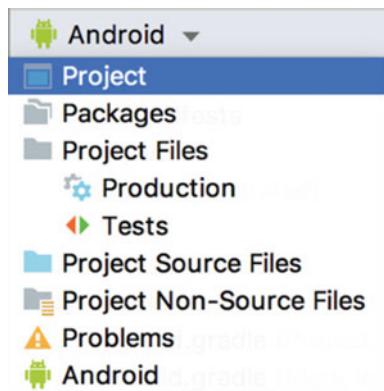
#### 6. `AndroidManifest.xml`

This is the config file for the whole Android project and all the four components you use in the project need to register in this file. You can add authority to the app in this file. This is a commonly used file and we will discuss this when we need to.

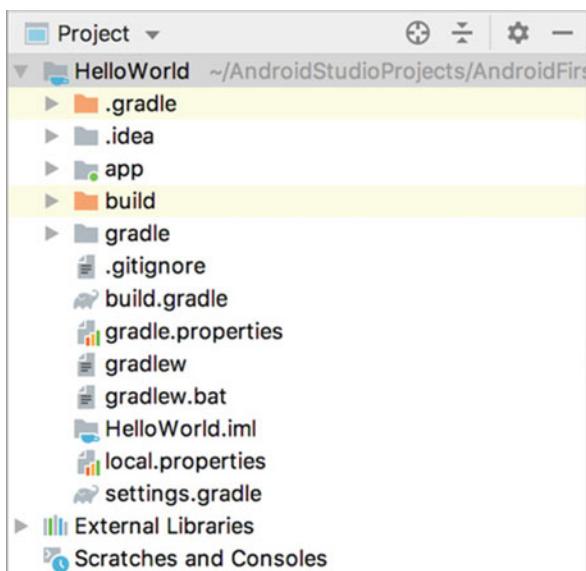
#### 7. test

**Fig. 1.20** Launcher**Fig. 1.21** Project structure in Android mode

**Fig. 1.22** Switching Project Structure Mode



**Fig. 1.23** Project Model Structure



This folder is used for unit test files.

#### 8. .gitignore

This file will allow the version control program to ignore files in app directory which is similar to the outer layer .gitignore file.

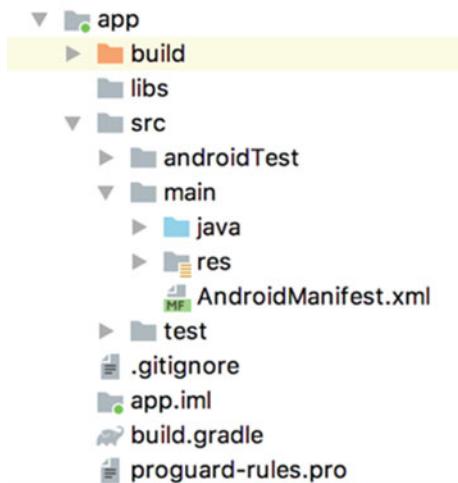
#### 9. app.iml

Another IntelliJ IDEA generated file and we don't need to care.

#### 10. build.gradle

This is the app module's gradle build script which will specify lots of configurations for building the project, we will cover the contents in the script shortly.

**Fig. 1.24** File structure under app



## 11. proguard-rules.pro

This is used for specifying the rules for code obfuscation. Obfuscation is used when you don't want your source code to be seen by other people.

That's all for the structure of the project. You might still have difficulty understanding everything mentioned here and that is totally normal. After you finish the whole book and look back, you will find the content here crystal clear and simple.

Now let us see what happens when HelloWorld starts running. First, open the `AndroidManifest.xml` file and you should find the code below:

```

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

Here, it registers `MainActivity` which is required so that an activity can be used. The two lines of code between `intent-filter` element are very important.

`<action android:name="android.intent.action.MAIN"/>` and `<category android:name="android.intent.category.LAUNCHER"/>` means that `MainActivity` is the main activity of this app and when user clicks the app icon in the phone, this activity will be started first.

What is the role of `MainActivity`? In the section that covers Android four main components, I mentioned that everything you can see in the app is hosted in the activity. Thus, what you see in Fig. 1.19 is `MainActivity`. Let's take a look at its implementation. Open the file and code should be as shown below:

```

class MainActivity : AppCompatActivity() {
  
```

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
}
}

```

First, we can see that `MainActivity` inherits `AppCompatActivity`. `AppCompatActivity` is a backward compatible activity provided in `AndroidX` and can ensure the same functionality in different versions of OS. Activity class is a base class provided by Android system and every activity defined in our app must inherit from it or its subclass to be able to inherit the functions and properties of activity. `AppCompatActivity`, for example, is a subclass of `activity`. The `onCreate()` method will be called when an activity is being created. But we cannot see “Hello World” at all. Where is this string?

We cannot find it in the activity because it is recommended to separate business logic and views in Android development. A widely adopted pattern is to write the UI in the layout file and then use the layout in the activity. We can see that `setContentView()` method gets called in `onCreate()` and it is this method that imported the `activity_main` layout for the current activity. So “Hello World” must be in this file! Let’s open it and take a look.

As mentioned before, layout files are all under `res/layout` folder, so you should be able to find `activity_main.xml` there. Open the file and switch to Text mode, code should be like follows:

```

<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

Feeling confused? That’s OK. I will cover more about layout in later chapters and now you just need to focus on `TextView` which is a widget provided by Android system that can show text in the layout. And you should see “Hello World!” there. Aha, so it is actually done by `android:text = “Hello World!”`.

Now that we’ve already learned the structure of `HelloWorld` project and the mechanism of how app runs, let’s take a look at the resources in a project.

### 1.3.5 Resources in a Project

Expand the res folder, you should find lots of folders and files in it as shown in Fig. 1.25.

It might be daunting at first glance, but they are actually well organized and easy to understand. All the folders start with “drawable” are used for images; “mipmap” directories are used for app icons; “values” directories are for strings, colors, etc.; and “layout” directories are used for layouts. Now the directories look simple and clear, right?

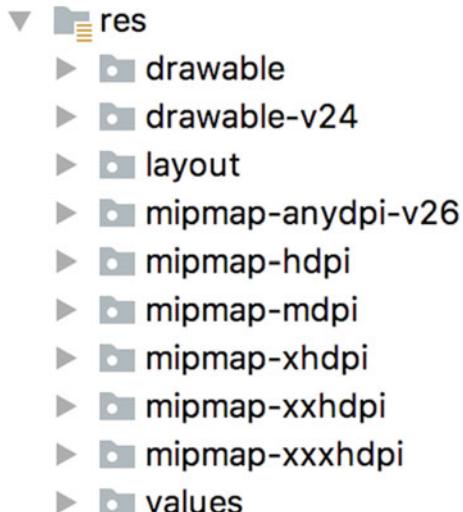
There are quite a few directories with prefix “mipmap” and “drawable.” This can help our app use different resources when run on different devices. Android Studio won’t generate the drawable-hdpi, drawable-xhdpi, drawable-xxhdpi folders and we need to create them by ourselves. When we develop an app, it’s recommended that we get different resolutions of the same image and put them into corresponding folders. Then the right resolution resource will be selected based on the resolution of current device. Of course, this is the ideal case most of the time—we might just get one picture, then we can put the images in drawable-xxhdpi since this is the most popular devices’ resolution.

Now let us look at how to use these resources. Open res/value/strings.xml file, and you should see code as follows:

```
<resources>
    <string name="app_name">HelloWorld</string>
</resources>
```

Here, a string of the app’s name gets defined and we have two ways to use it.

Fig. 1.25 Folders under res



- Use R.string.app\_name in Kotlin, Java code to get the reference of this string.
- Use @string/app\_name in XML file.

These are the two most common ways of getting reference of resources; we can replace string here with drawable to get image, with mipmap to get the icon, and so on.

Let us use an example to help understanding. Open AndroidManifest.xml file and find the code as shown below:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
</application>
```

Here, android:icon attribute specifies the HelloWorld app icon, android:label attribute specifies the name of the app. You can see the way to get these resources is exactly as we just mentioned.

Now you should know how to change the app icon or name of the app, right?

### ***1.3.6 File of build.gradle***

Unlike Eclipse, Android Studio uses Gradle to build the project. Gradle is an advanced tool to build the project which uses a DSL (Domain Specific Language) based on Groovy to configure the project and avoids the complicated configuration with XML-based tools like Ant and Maven.

From Sect. 1.3.4, we can see that there are two build.gradle files in HelloWorld project. One is at the outer layer and another one is under the app directory. These two files are instrumental to build the Android Studio projects and let us take a deep dive into these two files.

First look at the build.gradle file at the outer layer as shown below:

```
buildscript {
    ext.kotlin_version = '1.3.61'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.2'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$
kotlin_version"
```

```
        }
    }

allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

Code here is generated and its syntax may look confusing. However, if we ignore the syntax and focus on some key parts, it should be easy to understand.

First, the two repository closures all use `google()` and `jcenter()` methods. These two methods are used to specify the code repository that this project is going to use. The `google` repo is Google's Maven repo and `jcenter` is the repo mainly for third-party open sources libs. With these two methods, we can easily use any libs in the `google` and `jcenter` repo.

Next, in the dependencies closure, `classpath` specifies Gradle plugin and Kotlin plugin. Why need to specify Gradle plugin? This is because Gradle wasn't specifically created for building Android projects but for other types of projects written in Java, C++, etc. If we want to use it to build the Android project, we need to specify in Gradle to use `com.android.tools.build:gradle:3.5.2` plugin. The series number at the end is the plugin version number and should be the same as the current Android Studio version number. The Kotlin plugin just means that the current project is written with Kotlin. If you use Java to develop Android project, then there is no need to use this plugin. When I wrote this book the latest version of Kotlin plugin was 1.3.61.

That's everything for the outer layer `build.gradle`. Usually, you don't need to modify this file unless you want to make some global configuration changes.

Next, let us look at the `build.gradle` under the `app` directory, code should be the same as follows:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 29
    buildToolsVersion "29.0.2"
    defaultConfig {
        applicationId "com.example.helloworld"
        minSdkVersion 21
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

```

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$
kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.core:core-ktx:1.1.0'
    implementation 'androidx.constraintlayout:
constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test.ext:junit:1.1.1'
    androidTestImplementation 'androidx.test.espresso-
core:3.2.0'
}

```

This file is more complicated and let us go over them line by line. The first line applies a plugin which has two values to choose from: com.android.application means this is an application module; com.android.library means this is a lib module. The biggest difference between them is that the application module can run independently while lib module will need to be loaded in other apps so that it can run.

The next two lines apply the kotlin-android plugin and kotlin-android-extensions plugin. If you want to use Kotlin to develop Android app, then you must add the first plugin. The second plugin provides some useful extensions of Kotlin which you will find out in later chapters.

Next is a large android closure in which we can configure everything needed to build the project. Among them, compileSdkVersion is used to specify the SDK version used to compile the project and here we set it to SDK 29 which is Android V10.0. If there is a newer version, Android Studio will notify you.

Then we can see a defaultConfig closure within the Android closure which can specify more details about the project configuration. In this closure, applicationId is the unique identifier of the app and cannot be duplicated. By default, it will use the package name we specified when we created the project. minSdkVersion is used to specify the oldest version of Android that this project is compatible with, and we set it to 21 which is Android 5.0. targetSdkVersion specifies the target Sdk version which means that you've already extensively tested the app in this version and the system can open some new functionality or features to the app. For example, runtime permissions were introduced in Android 6.0. If you specify the targetSdkVersion to be 23 or higher, then the system will allow the app to use runtime permissions. However, if you specify targetSdkVersion to 22, then this means that you only have done extensive testing in Android 5.1 and lower versions, thus runtime permissions

cannot be used by this app. `versionCode` is used to specify the project version and `versionName` is used to specify the project version name. At last, `testInstrumentationRunner` is used to start the JUnit tests which are created to ensure the app can work as expected.

After the `defaultConfig` closure, let's look at the `buildTypes` closure. This closure is used to specify the configuration for the build file which usually has two sub-closures: `debug` and `release`. `Debug` closure is used to specify the configuration for the debug version of installer and `release` closure will specify the configuration for production version of installer. Notice that `debug` closure can be omitted thus we only see the `release` closure here. In the `release` closure, `minifyEnabled` is used to specify if we need to obfuscate the code or not. `proguardFiles` is used to specify the file with obfuscating rules. Here it specifies two files. The `proguard-android-optimize.txt` file is under `<Android SDK>/tools/proguard` directory and has general obfuscation rules for all projects. The `proguard-rules.pro` is under the root directory of current project which can be used to write the obfuscating rules for the current project. It is worth noting that all the installers generated by directly running Android Studio build and run are debug version. We will learn how to build the production version in Chap. 15.

That's all for the android closure. Next is the dependencies closure which can specify all the dependencies of the current project. There are three types of dependency: local binary dependency, local library module dependency, and remote binary dependency. Local binary dependency can add dependency to local jars or directories; local library dependency can be used to add dependency to local lib modules; remote dependency can be used to add dependency to the open-source projects in jcenter.

In dependencies closure, `implementation fileTree` is a local binary dependency, it will add all the files under `libs` directory with `.jar` suffix to the current project's build path. `Implementation` is for remote binary dependency and `android.appcompat:appcompat:1.1.0` is a standard remote dependency lib. `Androidx.appcompat` is the domain name which is used to distinguish from `libs` from other companies; `appcompat` is the project name which is used to differentiate from the `libs` in the same company; `1.1.0` is the version number that can differentiate the same lib with different versions. After adding this, Gradle will check if cache of this lib exists locally, if not, then it will download the lib and add to the build path. We don't have local lib dependency which has the format of implementation project + lib name. For example, if we have lib module with name `helper`, then in order to add dependency on this lib, we just need to add `implementation project(':helper')`. We will cover more detail on this in the last chapter. The `testImplementation` and `androidTestImplementation` are used for testing which we don't need for now.

## 1.4 Mastering the Use of Logging Tools

Now you've built your first Android app and been familiar with the structure of Android project and executing mechanism. Before we dive into more advanced topics, I'd like to cover log tools in Android since they will help you profoundly in your Android development journey.

### 1.4.1 Using Android Log Tool

The log class in Android is `Log(android.util.Log)` which provides the following 5 methods to print logs in the console. Here the verbosity is in descending order for these methods.

- `Log.v()`: used for the least meaningful information with the highest level of verbosity.
- `Log.d()`: used for the debugging information which should help you debug the app and investigate issues.
- `Log.i()`: used for important information like data that can help analyze the user behavior.
- `Log.w()`: used to print some warnings which means there is potential risks and need some attention to investigate into the issue.
- `Log.e()`: used to print error information such as the error info in the catch statement. When there are error info logs, it usually means your app has some serious issue that needs to get fixed immediately. This should have the least verbosity.

These 5 methods have their own overriding methods which shouldn't be difficult to understand and now let us try to use this tool in our `HelloWorld` project.

Open `MainActivity` and add log statement in `onCreate()` method as shown below:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        Log.d("MainActivity", "onCreate execute")  
    }  
}
```

`Log.d()` takes two params: the first param is `tag` which usually is the name of the current class and is used to filter the information; the second param is `msg` which is the content that needs to be printed.

Now reinstall and run `HelloWorld`. You can do so by clicking the Run button or use the shortcut Shift + F10(control + R in Mac). After the app finishes running, click



Fig. 1.26 Information in Logcat

the Android Monitor tab at the bottom toolbar. In Logcat you can see the printed information as shown in Fig. 1.26.

From the logs, you can see the content you specified, tag, package name, time stamp, and app process number.

You will find that Logcat has the logs coming from HelloWorld and other apps. I will cover how to filter out the logs that are coming from our own app in the next section.

Another thing worth noting is that you just wrote your first line of Android code, finally!

#### 1.4.2 Log Vs `Println()`

Lots of Java beginners use `System.out.println()` everywhere to print the information to help them debug and the corresponding method in Kotlin is `println()`. However, it is not recommended in more professional projects. If you use it in your company project, you will very likely face backlash in code review.

The reason why we shouldn't use `println()` is because there is nothing it can offer besides convenience. It is convenient because you just need to type "sout" and auto completion can fill the missing characters for you which should be one reason why beginners love to use it. However, you cannot control turning on and off of this method, also you cannot add tag for filtering purpose nor does this method provide information with different levels of verbosity control ...

Log class is not the silver bullet to solve every problem, but it is a tool with decent functionality. Let us take look at what the combination of Log and Logcat can do.

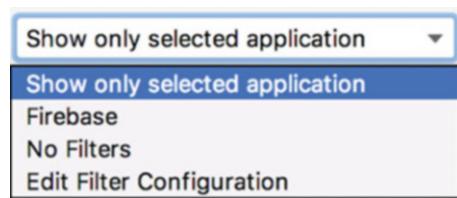
First, you can add filter easily in Logcat, and then you can see all the filter options in Fig. 1.27.

There are three filter options for now, Show only selected application and No filters are self-explanatory; Firebase is a platform provided by Google and we can ignore it for now. Now let us add our own filter.

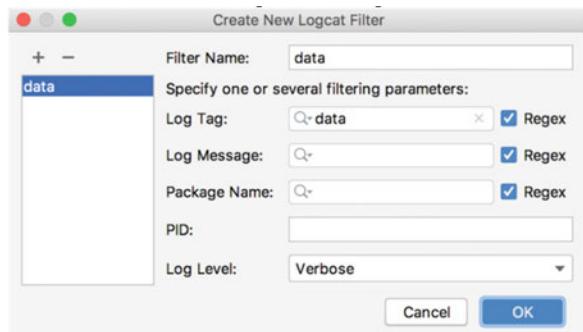
Click "Edit Filter Configuration" option, a window to show configuration for filter will pop up. We can name this filter data and let it filter the logs with tag named with data as shown in Fig. 1.28.

Click "OK" and you will find that now we have a filter named data. After selecting this filter, the logs that get printed in `onCreate()` are gone, this is because this filter will only show the logs that with tag value equals data. Now update the log

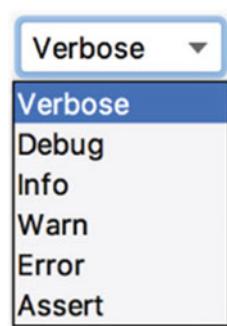
**Fig. 1.27** Filter options in Logcat



**Fig. 1.28** Filter configuration window



**Fig. 1.29** Verbosity levels in Logcat



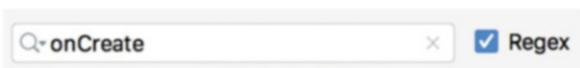
tag to data then reinstall and rerun the app, you should be able to see the log in Logcat.

For a few lines of logs, this feature doesn't seem very useful. However, when app prints hundreds or even thousands of lines of logs, you will need this feature, desperately.

Now let's take look at the verbosity level control in Logcat. Logcat has the same verbosity levels as mentioned previously as shown in Fig. 1.29.

When we choose Verbose which has the highest level of verbosity. This means that no matter which Log method we use to print the log, the log will show up. However, if we choose Debug, only logs that are at Debug or above level will show up, so on so forth. You can try it out by choosing Info, Warn or Error in Logcat and you will find that the logs we added won't show up since they used Log.d() method.

**Fig. 1.30** Keyword input box



The verbosity level control for logs can help you locate the logs you want much quicker. Imagine thousands of lines of logs in your screen and you just need to select Error in Logcat, then only error logs will show up.

Lastly, let us take look at keyword filtering. If you cannot identify the logs you want even with the combination of filter and verbosity level control, then you can use keyword filtering as shown in Fig. 1.30.

We can type in the keyword in the input box and then only those logs that have matches with the keyword will show up. It is worth noting that regex is supported here for more advanced filtering.

That is everything for using log tools in Android, I will stop here and you can continue the exploration of using Logcat by yourself. Let us summarize what we have learned in this section.

## 1.5 Summary

You must feel fulfilled and happy now. You should because you've already become a real Android developer. In this chapter, you first learned Android system, set up the Android development environment, built your first Android project, and then went over the structure of the Android project folders and the execution process of the app. At the end of the chapter, you also learned how to use the log tools in Android. Isn't it productive time?

However, there is a huge difference between an Android developer and an efficient Android developer. Before we start the next chapter, get some rest and digest the knowledge here and get ready for the next chapter.

# Chapter 2

## Explore New Language: A Quick Introduction to Kotlin



In the first 9 years after Android's birth, Java was the only language used to develop Android applications. Although Google introduced NDK in Android 1.5 to support the development of Android applications using C and C++, it never challenged the dominant role of Java.

This situation changed since 2017. Google announced that Kotlin became the first-class citizen for Android application development in Google I/O 2017 and became as important as Java. Android Studio also fully supported Kotlin since then. Two years later, in Google I/O 2019, Google announced that Kotlin became the first-choice language. Although Java could still be used, it was recommended to use Kotlin to write Android applications and the future official APIs would be mainly in Kotlin.

Based on statistics, among the top 1000 apps in Google Play store, more than 60% of the apps are written in Kotlin and this rate is increasing every year. Android official docs also preferably display the Kotlin version, and the official instruction videos and certain Google open-source projects switched to Kotlin.

Based on these facts, I made up my mind to use Kotlin in the third version of this book. There is no better time than today to learn and switch your Android projects to Kotlin.

To be honest, writing the third version of this book is quite challenging for me since I need to cover two broad topics in one book: Kotlin and Android. You need to learn Kotlin to write Android apps, however, if we spend half of the book discussing Kotlin syntax and then start to learn Android apps development that would be very tedious. Thus, I decided to mix them together, I will give a quick introduction to Kotlin with one chapter and then use the knowledge covered in that chapter to write Android apps, then in the chapters after, I will cover some Android knowledge and some advanced Kotlin topics. After finishing this book, hopefully you should be proficient in Kotlin and Android at the same time.

If you want to learn how to use java to develop Android apps, please refer to the second version of this book.

## 2.1 Introduction to Kotlin

I assume that Kotlin is a relatively new word for majority of the people, but it is actually not a very new language. Kotlin was designed and implemented by JetBrains. As early as 2011, JetBrains announced the first version of Kotlin and made it open source in 2012. But in the early days, it didn't get traction.

Kotlin version 1.0 was announced in 2016 which essentially meant that Kotlin was mature and stable enough, and JetBrains started to support Kotlin in its flagship IDE IntelliJ IDEA. Finally, developing Android apps provided a second choice. Kotlin gradually gained traction since then.

The rest is well known; in 2017, Google announced that Kotlin became the first-class citizen for Android development, and Android Studio added support for Kotlin, since then Kotlin took off.

Now you may feel confused. Android is provided by Google, then how can JetBrains as a third-party company, design its own language to develop Android apps?

To understand this, we need to examine the mechanism of Java language. Programming languages can mainly be divided into two categories: compiled and interpreted language. Compiled language is the language that the compiler compiles the source code into binary files that computers can execute directly. Both C and C++ are compiled languages. The interpreted language is different, it has an interpreter. When running the program, the interpreter will read the source code line by line and interpret the source code to binary code that computer can understand in real time, you can imagine that interpreted language usually is not as efficient as compiled language. Python and JavaScript are all interpreted language.

Here is the question: Is Java compiled or interpreted language? Even seasoned Java developers may give incorrect answers to this question. They argue that Java needs a compilation process since when everyone starts to learn Java, one learns to use javac command to compile. Thus, Java must be compiled language. But unfortunately, this is not the case. Although Java code does need compilation before running, the compiled code is not binary code that computers can understand but some special class files that can only be understood by Java virtual machine (it is called ART in Android land, an optimized virtual machine for mobile devices), and this virtual machine plays the role of interpreter. It will interpret the class files to binary code that computers can understand so that the class can run. Thus, strictly speaking, Java is an interpreted language.

Do you get any idea to answer the question after learning the Java running mechanism? So, Java virtual machine doesn't interact with your Java code but with the compiled class files. If I develop a new programming language and then make a compiler to compile the new language's code to the class file that respects the format, can Java virtual machine recognize such a class file? Your guess is right, this is how Kotlin works. Java virtual machines do not care if the class file comes from Java or Kotlin; as long as it is a legit class file, virtual machines will be able to run

it. Thus, JetBrains can design a language to develop Android apps as a third-party company.

Now you understand the mechanism of Kotlin, but why Kotlin can get traction in such a short time and become the official Android developing language only 1 year after the release of version 1.0?

There are many reasons for this, such as Kotlin having more elegant syntax. For the same functionality, Kotlin has 50% or even less footprint compared with Java. Kotlin syntax is more advanced. Compared with Java syntax, Kotlin sports many modern languages syntax sugars which can dramatically improve development efficiency. Additionally, Kotlin pays much attention to language safety and can almost eliminate NullPointerException, which is the top exception in the production environment. How can Kotlin achieve it? We will learn it shortly.

Among Kotlin's many excellent features, there is one crucial feature, that is, its 100% compatibility with Java. Kotlin can directly call Java code and use third-party APIs and frameworks seamlessly. This makes Kotlin inherit all the wealth of Java while enjoying rich new features.

Since Kotlin and Java have such an intertwined relationship, is it necessary to learn Java first before learning Kotlin? My answer is that if you learn Java first and then Kotlin, you will understand Kotlin better. If you have never learned Java but have experience in any other language, you can still learn Kotlin directly, though may find it relatively difficult to understand certain concepts compared with experienced Java developers. However, if you have zero experience in programming, then it is a good idea to start with some more fundamental books.

In addition, this book will not cover Java basics, so if you want to learn Java, please refer to other books. OK, we can stop here for the history of Kotlin. Before we officially start to learn Kotlin, let us see how to run Kotlin code.

## 2.2 How to Run Kotlin Code

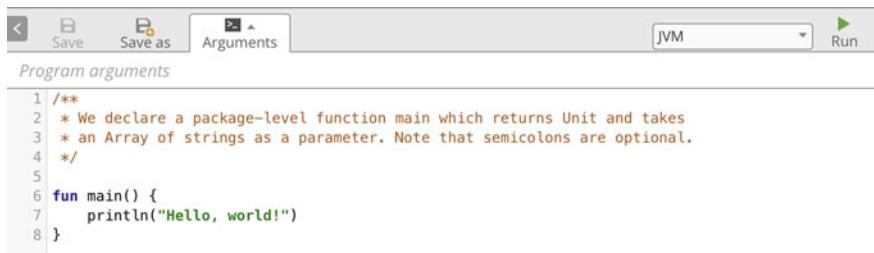
The goal of this chapter is to quickly grasp the fundamental ideas of Kotlin, thus there will be no Android topic in this chapter.

Since it has nothing to do with Android, the first question we need to solve is how to run a piece of Kotlin code independently.

There are mainly 3 methods to run Kotlin code, and I will cover them one by one.

The first method is to use IntelliJ IDEA. It is JetBrains' flagship IDE and perfectly supports Kotlin. You can simply create a Kotlin project inside IntelliJ IDEA and run the Kotlin code directly. But you need to download another IDE thus we don't recommend this method.

The second method is to run Kotlin code online. To help developers quickly experiment with Kotlin, JetBrains built a website that can run Kotlin code. The web address is <https://try.kotlinlang.org>, and it looks like Fig. 2.1.



The screenshot shows a code editor window with the following code:

```
1 /**
2 * We declare a package-level function main which returns Unit and takes
3 * an Array of strings as a parameter. Note that semicolons are optional.
4 */
5
6 fun main() {
7     println("Hello, world!")
8 }
```

Fig. 2.1 JetBrains website to run Kotlin

To run the Kotlin code in the editor, simply click the “Run” button. However, to run the code online only fits for small pieces of code in the same file. Considering this, we don’t adopt this method too.

The third method is to use Android Studio. Unfortunately, as an IDE specifically designed to develop Android apps, you can only create Android project instead of Kotlin project inside Android. But it is totally fine, we can just create an Android project and then write a Kotlin main() method, then we can independently run Kotlin code.

Let us just open the HelloWorld project from last chapter. First, find the location of MainActivity as shown in Fig. 2.2.

Next, create a LearnKotlin file under the same package as MainActivity. Right click com.example.helloworld → New → Kotlin File/Class. In the dialog, type LearnKotlin, as shown in Fig. 2.3. Click “OK” to finish creating the file.

Now let us create a main() method and print some logs inside this LearnKotlin file, as shown in Fig. 2.4.

You probably notice that, on the left side of main() method there is a small Run icon. Now we just need to click that icon and select the first Run option to run this piece of Kotlin code. The result will be shown under the Run tab, as shown in Fig. 2.5.

The “Hello Kotlin!” shows that our code runs successfully.

Maybe you will ask why we still use println() here? Isn’t it mentioned in last chapter that we should use Log instead of using println()? This is because Log is a utility provided by Android and we are running independent Kotlin code, it has nothing to do with Android, and we cannot use Log here.

This is the way to run Kotlin code inside Android Studio independently. We will use this method to run and test the Kotlin code we will learn in this chapter. Now let us officially start to learn Kotlin programming.

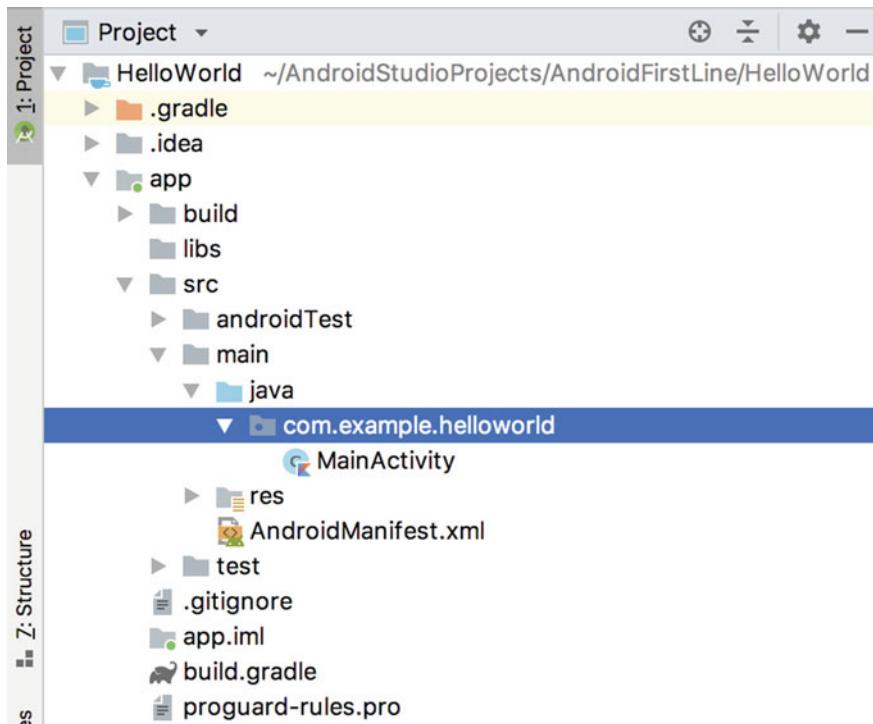
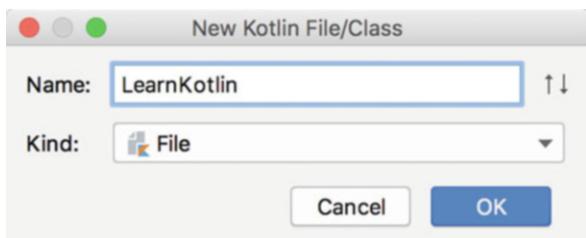


Fig. 2.2 The project structure of HelloWorld

Fig. 2.3 Dialog to create Kotlin file



The code editor displays the following Kotlin code:

```
1 package com.example.helloworld
2
3 fun main() {
4     println("Hello Kotlin!")
5 }
```

Fig. 2.4 Sample Kotlin code



Fig. 2.5 Running result

## 2.3 The Foundation of Programming: Variables and Functions

There are so many programming languages in the world and in fact there has more than 600 recorded programming languages, not mention of these that never get official record. Almost all of these programming languages support variables and functions. Variables and functions are the essential and fundamental parts of programming language. This chapter will cover how variables and functions are used in Kotlin.

### 2.3.1 *Variables*

Let's learn variables first. Kotlin adopts a very different way to declare variables. In Java, you need to specify the type of the variable, for example, int a means a is an integer, String b means b is a string. However, Kotlin only allows you to use two key words when declaring a variable: val and var.

Val (short for value) is used to declare a value variable that is immutable. This kind of variable cannot be assigned to another value after initialization, which corresponds to the final variable in Java.

Var (short for variable) is used to declare a mutable variable, which means that this variable can change to another value after initialization which corresponds to non-final variables in Java.

If you have experience in Java, you may wonder how can the compiler know which type is the variable if only these two key words are available? This has something to do with Kotlin type inference.

For example, let us open the LearnKotlin file created from the last section and type the following code in the main() function:



**Fig. 2.6** Print the value of variable a

**Table 2.1** Java and Kotlin data types comparison

| Java primitives | Kotlin class types | Data type |
|-----------------|--------------------|-----------|
| Int             | Int                | Integer   |
| Long            | Long               | Long      |
| Short           | Short              | Short     |
| Float           | Float              | Float     |
| Double          | Double             | Double    |
| Boolean         | Boolean            | Boolean   |
| Char            | Char               | Character |
| Byte            | Byte               | Byte      |

```
fun main() {
    val a = 10
    println("a = " + a)
}
```

Notice that there is no semi-colon at the end of statements, this is something you need to adapt if you are used to Java.

In the code above, we use val to declare a variable a and assign value of 10 to a, so a will be inferred as an integer variable. The reason is because if you assign an integer value to a variable, then it has to be integer type. If you assign a string to a then a has to be string type. This is how Kotlin type inference works.

Now let us run the main() function, and the result is shown in Fig. 2.6, as we expected.

However, type inference doesn't always work in Kotlin. For instance, if we late initiate a variable, Kotlin cannot automatically infer the type of the variable, and we need to declare the type. The corresponding syntax is as follows:

```
val a: Int = 10
```

Now, we declare a as integer type, and Kotlin will no longer try to infer type for this variable. If you assign a string to a, compiler will throw out type mismatch exception.

If you have experience in Java and are careful enough, you'd likely find that in Kotlin the first letter of Int is upper case while in Java it is lower case. This change is actually quite significant. Because this means that Kotlin totally eliminates the primitive types in Java and treats everything as class. Java int is a key word, but Kotlin Int is a class that has its own methods and inheritance relationships. Table 2.1 lists the Kotlin class types and the corresponding Java primitive types.



**Fig. 2.7** The result of mutating a's value

Now let us try to operate on these numbers. Let us try to make a 10 times larger, and you may have the following code:

```
fun main() {
    val a: Int = 10
    a = a * 10
    println("a = " + a)
}
```

Unfortunately, compiler will throw an error: Val cannot be reassigned. It tells us that the variable declared by val cannot be reassigned. The problem here is that we declare a as val and initialize it with value 10 and then try to assign another value to it; thus, the compiler will throw the error.

To solve this problem, as mentioned before, val is used to declare an immutable variable, and var. is used to declare a mutable variable; thus, we only need to replace val with var., as shown below:

```
fun main() {
    var a: Int = 10
    a = a * 10
    println("a = " + a)
}
```

Now compiler won't complain anymore and let's run the code again, the result should be like Fig. 2.7.

As you can see, the value of a is 100, which means the operation succeeded.

Now you wonder why we need keyword val if it has so many constraints. Why not just use var. all the time? Kotlin introduced val to solve the abuse of final keyword in Java.

In Java, unless you declare final for a variable, it will be mutable, and this is not a good idea. When the project is getting more complex, an increasing number of people join the team, and you will not be able to know when a mutable variable will be mutated by whom though this variable should never be mutated which leads to some really hard-to-debug bugs. Thus, a good programming habit is to add final to all the variables unless a variable absolutely needs to be mutated.

But not all the developers will have such good habit and I believe that at least 90% of Java developers don't keep it in mind all the time, simply because this is not

enforced in Java. Thus, Kotlin adopted a totally different approach in the design phase by providing val and var. keywords. Developers have to make a choice if a variable is immutable or not.

Now when should we use val and when should we use var.? My take is that, always try to declare a variable with val at first and if you find that you absolutely need to mutate it, then change it to var. These criteria should help build more robust and high-quality codebases.

### 2.3.2 *Functions*

Function is another essential and fundamental concept of programming, the main() function we used previously is a special function that serves as the entry point of the program. This means that when the whole program gets started, it will start running main() function first.

A program with a simple main() function is apparently very primitive and as other programming language, Kotlin also allows user to define functions freely. The syntax is as follows:

```
fun methodName(param1: Int, param2: Int): Int {  
    return 0  
}
```

Now let us take look at the code here, first we need to use fun(short for function) keyword to declare a function.

Immediately after fun is the name of the function which has no specific requirements, you can use any name as you want but a readable and meaningful name is always recommended.

Immediately after the name is a pair of parentheses and between which you can declare params that this function will take, and you can declare as many params as you want to. The example here shows that this function will take two params of Int. The way to declare param is “param name: param type“. You can name the param freely. If the function doesn’t need to take param, then just keep it empty between parentheses.

The return type of a function which is before the right brace is optional. The example above shows that this function will return data of type Int. If your function doesn’t need to return any data, then you can omit this part.

Between the braces are the contents of the function body which expresses the logic of the function. Inside this function, it simply returns 0 as it was declared to return data type of Int.

This is the standard way to declare a function. Kotlin actually has many other keywords to decorate the functions, but the above syntax should be enough to handle more than 80% of the use cases and we will learn other keywords in later chapters.

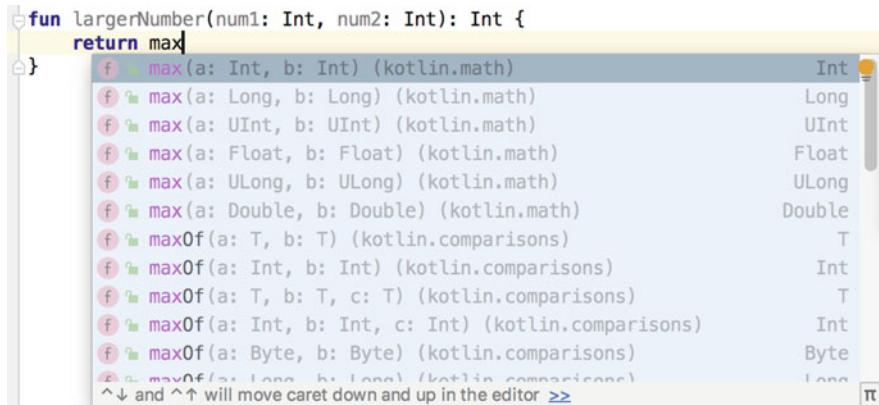


Fig. 2.8 Android Studio code auto completion

Now let's apply the syntax above to declare a meaningful function as shown below:

```
fun largerNumber(num1: Int, num2: Int): Int {
    return max(num1, num2)
}
```

Here we define a function called `largerNumbers` with super simple functionality. It takes two `Int` type params and then return the larger one of these 2 params.

Note that, this function used a function called `max()` which is an internal function provided by Kotlin that will return the larger number of two arguments, thus `largerNumber()` is actually a wrapper for the internal function `max()`.

When you start to type `max`, Android Studio will have the prompt as shown in Fig. 2.8.

Android Studio has excellent support for code autocompletion, usually you just need to type in a few letters, then, it will predict what you want to type and give a list of suggestions, we can use the UP and DOWN key to move in the list and press Enter to select the code.

I highly recommend you to use the code auto completion feature not only because it saves the typing time but also will help us import the library automatically. For example, if you manually type `max()`, there will be red error as shown in Fig. 2.9.

This is because you didn't import the package that has `max()` function. There are a few ways to import the package. By moving the cursor to the red error, you will see the shortcut of importing the package, but the best way to do so is always using the autocompletion as it is done automatically.

Now let's try to use the autocompletion to write the `max()` function and you will find that at the header of the `LearnKotlin` file, `max` package gets imported and no more errors, as shown in Fig. 2.10.

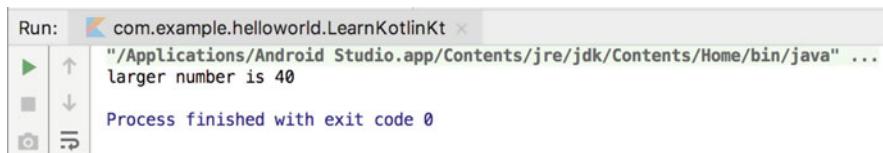
**Fig. 2.9** max() function error info

```
1 package com.example.helloworld
2 ? java.lang.Integer.max? (multiple choices...) ↗
3     run target number(num1: Int, num2: Int): Int {
4         return max(num1, num2)
5     }
6 }
```

**Fig. 2.10** auto import max() package

```
package com.example.helloworld
import kotlin.math.max

fun largerNumber(num1: Int, num2: Int): Int {
    return max(num1, num2)
}
```



**Fig. 2.11** The result of running largerNumber() function

Importing package is some fundamental Java knowledge, but after the first version of this book got published, a small number of users gave the feedback that after typing the code as shown in the book they still got errors and the reason was that package didn't get imported properly, thus I decided to cover Android Studio code autocompletion to solve the package importing problem.

Now, we can try to call the largerNumber() inside the main() as shown below:

```
fun main() {
    val a = 37
    val b = 40
    val value = largerNumber(a, b)
    println("larger number is " + value)
}

fun largerNumber(num1: Int, num2: Int): Int {
    return max(num1, num2)
}
```

Code here is simple. We define variable a and variable b with value 37 and 40 correspondingly, then we call the largerNumber() and pass a and b in as arguments. Then largerNumber() will return the larger one of these two numbers and print the return value. Now let's run the code and the result is shown as Fig. 2.11. This program works as we expected.

The code here is very simple and basic but after you master the basics of function syntax, you can create functions as complex as you want.

At the end of this section, let us learn some Kotlin syntax sugar which will play an important role later.

When there is only one line of code in a function, Kotlin allows to omit the braces by using an equal sign. Let's use largerNumber() function as an example, and it can be simplified like the code below:

```
fun largerNumber(num1: Int, num2: Int) : Int = max(num1, num2)
```

By using this syntax, even the return keyword can get omitted as the equal sign is enough to serve the same purpose. Also, still remember that Kotlin can infer data types? It can also be applied here. Since max() returns Int type, and largerNumbers() just returns the value of max(), then Kotlin will infer that largerNumber() will also return an Int type. Thus, there is no need to explicitly define the type of the return data, and code can be further simplified to the following:

```
fun largerNumber(num1: Int, num2: Int) = max(num1, num2)
```

You may think that there are not that many cases of one-line functions and this syntax sugar won't be very useful. But it is not true. It can be used together with some other Kotlin syntax features to simplify the code and we will learn them in later chapters.

## 2.4 Flow Control

Code can run in serial, conditionally and repeatedly. It is easy to see that code should run line by line, but it won't be enough to express our logic. Thus, we need to introduce the conditional statements and loop statements which will be covered in this section.

### 2.4.1 *if Statement*

There are two ways to make condition check in Kotlin: if and when.

First, let us learn how to use if which is almost identical in Kotlin and Java, and I will just use a simple example to explain.

Use the previous largerNumber() function as an example, we used the internal max() function to return the larger values of the two inputs. We can use if to accomplish the same function. Change the implementation of largerNumbers() to the following code:

```
fun largerNumber(num1: Int, num2: Int): Int {  
    var value = 0  
    if (num1 > num2) {  
        value = num1  
    } else {  
        value = num2  
    }  
    return value  
}
```

Anyone with basic programming knowledge should be able to understand it, but I would still want to highlight that I used var. keyword here because we need to initialize the value and reassign value to it.

Remember that I said syntax of if in Java and Kotlin was almost identical? Now let's see what the difference is. In Kotlin if has one extra function compared with Java. It can return value of the if statement. Thus, the above code can be simplified as below:

```
fun largerNumber(num1: Int, num2: Int): Int {  
    val value = if (num1 > num2) {  
        num1  
    } else {  
        num2  
    }  
    return value  
}
```

Notice how if uses the value to return and assign to val value. Since there is no reassigning, we can now use val to define value.

Look at the code again and we can find that value variable is redundant, and we can simplify the code even further as follows:

```
fun largerNumber(num1: Int, num2: Int): Int {  
    return if (num1 > num2) {  
        num1  
    } else {  
        num2  
    }  
}
```

Now you might think it is already very concise, but there is still room for improvement. Can you still recall that when there is only one line of code, we can omit the braces? Even though technically there are more than 1 line of code in largerNumber() function, however it functions as one line of code and just return the value of if statement, thus we can simplify the code further to the following format:

```
fun largerNumber(num1: Int, num2: Int) = if (num1 > num2) {
    num1
} else {
    num2
}
```

Now you can see why the syntax sugar I mentioned above is important as you can combine it with other syntax sugar Kotlin provided and can be applied to a lot of scenarios.

And you can make it even simpler by compressing it into truly one line:

```
fun largerNumber(num1: Int, num2: Int) = if (num1 > num2) num1 else num2
```

With a simple if statement, we discovered so many interesting Kotlin syntax usages, do you feel the power of Kotlin?

#### **2.4.2   *when Statement***

when statement in Kotlin is like the switch statement in Java but way more powerful.

If you're familiar with Java, you should know the limitation of switch statement in Java. First, switch can only take integer or type that is shorter than integer for condition check. Although JDK 1.7 and later versions started to support strings, if you want to use other data types for condition check, then switch won't support it. Secondly, after each case in switch statement, we must add a break there, otherwise code will continue to execute which caused numerous bugs.

The when statement solves all the problems above and provides more powerful features and sometimes is even simpler and easier to use than if.

We're going to build a function that can be used to return a student's score with the student's name. Let us use if statement to implement this function and write the following code inside LearnKotlin file:

```
fun getScore(name: String) = if (name == "Tom") {
    86
} else if (name == "Jim") {
    77
} else if (name == "Jack") {
    95
} else if (name == "Lily") {
    100
} else {
    0
}
```

Here we define a `getScore()` function which will take in student's name as param and use if to get the student's score and return it. The code above uses the single line function syntax sugar again.

Though it can work as expected, it looks super redundant to write so many if and else. We should use when statement when there are many condition checks and let's rewrite the code as follows:

```
fun getScore(name: String) = when (name) {
    "Tom" -> 86
    "Jim" -> 77
    "Jack" -> 95
    "Lily" -> 100
    else -> 0
}
```

You might find that when statement and if statement can both return value and thus, we can still use the single line function syntax sugar.

when statement allows using any data type as arguments and you can define the conditions inside the body of when statement, the format is:

Matching value -> {corresponding logic}

When you only have one line of code, {} can be omitted. With this, it should be very easy to understand the code above.

Besides data matching, when statement also allows type matching. Let me use another example to explain what is type matching. Let's define a `checkNumber()` function as follows:

```
fun checkNumber(num: Number) {
    when (num) {
        is Int -> println("number is Int")
        is Double -> println("number is Double")
        else -> println("number not support")
    }
}
```

In the code above, is keyword is the key for type matching which is equal to `instanceof` in Java. `checkNumber` function takes a param of `Number` type and it is an abstract class in Kotlin. Number-related types like `Int`, `Long`, `Float` and `Double` are all sub class of `Number` class. Thus, we can use type matching to decide what type of the argument is, and if it is `Int` or `Double`, then, print the type name otherwise print this type is not supported.

Now we can call `checkNumber()` inside the `main()` as shown below:

```
fun main() {
    val num = 10
    checkNumber(num)
}
```



**Fig. 2.12** Result when pass in Int type for checkNumber() function



**Fig. 2.13** Result when pass in Long type for checkNumber() function

Here we pass in an Int type to checkNumber() and running the code will yield the result as shown in Fig. 2.12.

Code here successfully predicts that the argument is Int type.

Now let's change to Long type:

```
fun main() {
    val num = 10L
    checkNumber(num)
}
```

Run the code again, the result is as shown in Fig. 2.13.

Apparently, our code here doesn't support this kind of argument.

when statement is usually used like this. There is another way of using when statement that is not used very often but can be extensible in certain scenarios.

Use getScore() function as an example, if we don't pass in the argument we can write like this:

```
fun getScore(name: String) = when {
    name == "Tom" -> 86
    name == "Jim" -> 77
    name == "Jack" -> 95
    name == "Lily" -> 100
    else -> 0
}
```

And you can see that we simply explicitly do condition check inside the statement body, also notice that in Kotlin we can use `==` to determine if two strings or objects

are equal instead of using equals() as Java does. You might feel the code above is redundant compared with the previous version, however there are scenarios that only by doing so can meet the needs. For example, assume the score of all the students' name that start with Tom is 86, then we cannot use when statement with the argument. Instead we can write something like the following:

```
fun getScore(name: String) = when {  
    name.startsWith("Tom") -> 86  
    name == "Jim" -> 77  
    name == "Jack" -> 95  
    name == "Lily" -> 100  
    else -> 0  
}
```

Now whether the argument is Tom or Tommy, anyone whose name starts with Tom will have score of 86.

From the section, you should see that when statement in Kotlin is much more flexible than the switch statement in Java.

### 2.4.3 Loop Statement

After the condition statements, let's start to learn loop statement in Kotlin.

There are mainly two kinds of loop statement in Java: while and for. Kotlin also provides while and for loop, and while loop is the same as in Java, thus we won't spend time on it. Even if you've never learnt Java, it is OK as long as you learned any mainstream programming language like C, C++ as their while statements are pretty much the same.

Now let's learn how to use for loop inside Kotlin.

Kotlin made many improvements to the syntax of for loop, for example it eliminates the for-i loop; instead of using for-each loop, Kotlin uses for-in loop.

Before that we need to introduce the concept of range which doesn't exist in Java. For instance, we can use the following Kotlin code to represent a range:

```
val range = 0..10
```

This looks strange but totally legit in Kotlin. It creates a closed range between 0 and 10 which means 0 and 10 are all included in this range and in mathematical representation it is [0,10]. The keyword to create range is ... and by specifying the two ends, we can create a range.

With that, we can use for-in to iterate the range and write code inside main() functions as follows:

```
fun main() {  
    for (i in 0..10) {
```



**Fig. 2.14** Use for-in to iterate through range

```
    println(i)
}
}
```

This is the simplest demonstration of for-in loop. It simply iterates through every element of the range and prints each value. Running the code will yield result as shown in Fig. 2.14.

In most cases, closed range is not as useful as semi-open range. Why? This is because the index of arrays starts with 0 and for an array of length 10, its index range is from 0 to 9 and a left-closed and right-open range is more common in programming. You can use until keyword in Kotlin to create a range that is left closed and right open as shown below:

```
val range = 0 until 10
```

The above code creates a left closed, right open range between 0 and 10 with mathematical representation [0,10) by replacing the .. operator with until. Run the code again, you will find that it won't print 10.

By default, for-in loop will increase by 1 in every iteration which is equivalent to i++ in Java's for-i loop, however if you want to step over certain elements, you can use step keyword:

```
fun main() {
    for (i in 0 until 10 step 2) {
        println(i)
    }
}
```

The above code means that when iterating through [0,10), the step difference is 2 and is equivalent to i = i + 2 in for-i loop. Rerun the code will yield the result as shown in Fig. 2.15.



**Fig. 2.15** Use step to skip the element in the range



**Fig. 2.16** Use downTo to iterate descending interval

Now you can see that all the odd values are skipped. With the step keyword we can implement some more complex loop logics.

Notice that by using .. operator and until keyword, we are creating a range that the right side is larger than the left side which is an ascending interval. If you want to create a descending interval, you should use downTo keyword as follows:

```
fun main() {
    for (i in 10 downTo 1) {
        println(i)
    }
}
```

Here, we created a [10, 1] descending interval and rerun the code will yield result as shown in Fig. 2.16.

The step keyword can also be applied to descending interval which I will skip here.

The for-in loop can be used to iterate not only the interval but also arrays, and sets. We will learn the concept of set later in this chapter and will learn how to use for-in there too.

To summarize, for-in loop is not as flexible as for-i loop but is much easier to use and can cover majority of the use cases. If for-in cannot meet the needs for certain cases, then we can use while loop instead.

## 2.5 Object-Oriented Programming

Just like many other modern advanced programming languages, Kotlin is object-oriented. Thus, it is very important to understand what object-oriented programming is. You can Google object-oriented programming to see more formal definition. However, in my opinion, those who don't understand object-oriented programming won't be able to understand even after reading the definition.

Here I would try to use my own understanding to help you understand what is object-oriented programming (OOP). OOP is different from structural programming languages (Like C) in a way that it can create classes. A class is an abstraction of things, for example people, cars, houses or books or any other things. We can abstract something into a class whose name usually is a noun. The class can have fields and functions. Fields represent the properties that this class has, like people have name and age, cars have brand name and price. The names of fields are usually nouns too. Functions are used to represent what behavior the class can have, for example, humans can eat and sleep, cars can be driven and maintained, the name of functions are usually verbs.

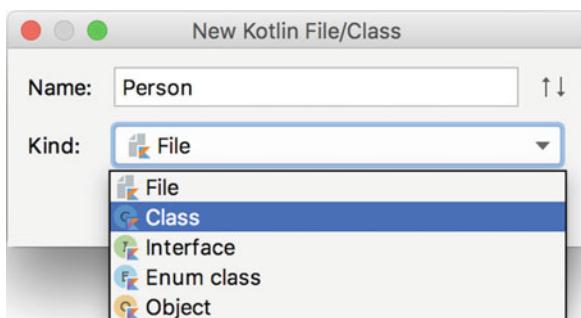
By encapsulating the fields and functions, we can create the instance or object of the class and then utilize the fields and functions to meet our programming requirements. This is the fundamental idea of OOP. Of course, there are other features of OOP like inheritance, polymorphism, etc., but all of these are based on this fundamental idea, and we can learn other features later.

### 2.5.1 Class and Object

Let's try to apply the idea we just learned to start OOP. First let us create a Person class, right click com.example.helloworld package ->New->Kotlin File/Class, then type in Person in the prompt dialog. By default, it will choose File for Kind, and let's choose Class here as shown in Fig. 2.17.

Clicking "OK" will finish the creation of the new class and will generate the following code.

**Fig. 2.17** Select creating new class



```
class Person {  
}
```

This class is empty for now, but you can tell that Kotlin also uses class keyword to declare a class which is the same as Java. Now let's add fields and functions to this class. I will add name and age field and eat() function since apparently everyone has a name and age and need to eat.

```
class Person {  
    var name = ""  
    var age = 0  
  
    fun eat() {  
        println(name + " is eating. He is " + age + " years old.")  
    }  
}
```

We use var. to declare name and age because we need to assign their values after we create the instance and if we use val, we cannot reassign the value after initiation. We also define a simple eat() function which will print a string.

After definition, let's use the following code to instantiate this class:

```
val p = Person()
```

Compared with Java, Kotlin doesn't need to use new keyword to instantiate the class and the reason is because when you call the constructor function of a class, the only possible reason is because you want to instantiate the class thus without keyword new, it still clearly shows your intention. Kotlin follows the minimalism principle and eliminates redundant structure like new keyword and semi-colons at the end of the statements.

Let's assign this value to variable p. Now p is an object and an instance of Person class.

Now let's start to operate on p inside the main() function.

```
fun main() {  
    val p = Person()  
    p.name = "Jack"  
    p.age = 19  
    p.eat()  
}
```

Now object p's name field has value of Jack and age field has value of 19. Then let us call eat() function, the result is shown as Fig. 2.18.

That's the basics of OOP. In summary, we need to abstract things and encapsulate into class, then we need to define the properties as fields and capabilities as functions, then we instantiate the class and use the fields and functions as needed.



**Fig. 2.18** The result of running eat()

### 2.5.2 Inheritance and Constructor Function

Now let us start to learn another important concept of OOP which is inheritance. Inheritance is another concept that is an abstraction of the real world which makes it quite easy to understand. For example, let's define a Student class and each student has their own student id number and grade, thus we can add sno and grade fields in the Student class. However, students are humans at the same time and they will have name, age and need to eat. If we define name, age fields and eat() function again inside the Student class, then, it will be too redundant. A better solution would be to allow Student to inherit the Person class and then Student will get the fields and functions in the Person for free and only needs to define its own fields and functions.

This is the basic idea behind the concept of inheritance in OOP, quite easy to understand right? Now let's try to implement it in Kotlin. Right click the com.example.helloworld > New > Kotlin File/Class and type in Student in the dialog then choose Class for Kind.

Click “OK” to finish the creation, and add student number and grade in Student class as shown below:

```
class Student {
    var sno = ""
    var grade = 0
}
```

Student doesn't inherit from Person for now and we need to do two things to make it happen.

First, we need to make Person inheritable. A lot of people especially those who are familiar with Java might get confused: isn't a class inheritable by default? And this is one area that Kotlin is different from Java. In Kotlin any non-abstract class is not inheritable by default which is equivalent as adding final keyword in Java. The idea is the same as val keyword, if a class is inheritable by default, then it would be difficult to avoid the risk of random inheritance implementation. In Effective Java, it is highlighted that if a class is not intentionally designed to be inherited, then we need to proactively add final keyword to prevent inheriting from this class.

Apparently Kotlin designers followed this principle and made it by default that all non-abstract classes are not inheritable. We highlight the non-abstract class here because the abstract classes cannot be instantiated and only the classes that inherit it

can be instantiated. I will skip the introduction to abstract class as it is the same in Java and Kotlin.

We just need to add the open keyword before the Person class to make it inheritable as shown below:

```
open class Person {  
    ...  
}
```

With the open keyword, we're telling the Kotlin compiler that Person is designed to be inheritable.

The second thing we need to do is to let Student class inherit Person class. In Java the keyword is extends and in Kotlin it is a colon, as shown below:

```
class Student : Person() {  
    var sno = ""  
    var grade = 0  
}
```

Notice that there are parentheses after the Person which do not exist in Java. It maybe a little bit confusing for the beginner since this has something to do with more advanced concepts like primary constructor, secondary constructor, etc. Here let me use plain language to help you understand the meaning and function of these parentheses and get some idea of the primary constructor and secondary constructor in Kotlin.

Any OOP language have the concept of constructor, so does Kotlin. However, Kotlin has two kinds of constructors: primary constructor and the secondary constructor.

The primary constructor is the constructor you use the most and each class will have a parameterless constructor, and of course you can also explicitly define the parameters there. The primary constructor has no function body and is defined right after the class name like the code below:

```
class Student(val sno: String, val grade: Int) Person() {  
}
```

Here we put student number and grades fields into the primary constructor which means that when instantiating the Student class, the caller has to pass in the parameters required by the constructor. The following code is an example.

```
val student = Student("a123", 5)
```

The code above creates an instance of the Student class and also assigns a value of a123 to student number and 5 to grade. Notice that the parameters are all defined as val because the values are passed in instantiation and no need to reassign the value.

Then, the question would be that, since there is no function body in the primary constructor, if we want to add some code in the primary constructor, what can I do? Kotlin actually provides an init block in which we can add the logic as shown below:

```
class Student(val sno: String, val grade: Int) : Person() {
    init {
        println("sno is " + sno)
        println("grade is " + grade)
    }
}
```

The code above will simply print out the student number and the value of grade, and if you try to instantiate the Student class then you should be able to see the printed result.

Everything still makes sense so far? However, how is this related to the parentheses? This has something to do with another rule of Java inheritance that is the constructor of sub class has to call the constructor of the parent class and Kotlin follows this rule.

Now let's go back to the Student class. We defined a primary constructor and based on the rule above, we need to call the Person's constructor. However, there is no function body in the primary constructor. So how can we call the parent's constructor? Your answer might be to call in the init block. This may work but is not ideal since under most circumstances, we don't need the init block.

Kotlin adopted a simple yet not straightforward approach to resolve the issue: parentheses. The sub class's primary constructor will call the parent's constructors by using parentheses. And now you should understand the code below.

```
class Student(val sno: String, val grade: Int) : Person() { }
```

Here the empty parentheses after the Person class means that the Student class's primary constructor will call the parameterless constructor of the Person class. The parentheses here cannot be omitted even if there is no param at all.

Modify the Person class as shown below:

```
open class Person(val name: String, val age: Int) {
    ...
}
```

Then you will see error in the Student class. Of course, if you keep the code for instantiating the Person class, then you will find error there too, since it is unrelated to what we're going to learn. You can fix it by yourself or just remove the instantiation code.

Now Student class will show error as in Fig. 2.19.

And the reason for the error here is very clear. The parentheses after the Person means that it is going to call the parameterless constructor of the Person class,

```

1 package com.example.helloworld
2
3 class Student(val sno: String, val grade: Int) : Person() {
4
5     init {
6         println("sno is " + sno)
7         println("grade is " + grade)
8     }
9
10 }

```

**Fig. 2.19** Error in Student Class

however, there is no parameterless constructor in the Person class anymore which causes the error.

To resolve the error, we need to pass in the name and age and we need to add name and age in the primary constructor of Student class. Then pass these two arguments to the constructor of person class as shown below:

```

class Student (val sno: String, val grade: Int, name: String, age: Int) :
  Person(name, age) {
  ...
}

```

Notice that we cannot declare name and age to be val as the var. and val keyword will automatically make them the fields of the class which will conflict with the name and age fields in the parent class. Thus, we don't need to add any keyword before the name and age parameter and the scope of these two parameters will be within the primary constructor.

We can use the code below to create an instance of the Student class:

```
val student = Student ("a123", 5, "Jack", 19)
```

Now you should have a good understanding of the primary constructor in Kotlin and the usage of parentheses in inheriting. However, parentheses in inheritance are more than that as we haven't covered the secondary constructor.

In real world, you might never need to use the secondary constructor since Kotlin provides a way to set the default value to the parameters which should be able to replace the secondary constructor completely. However, in order to give you a complete picture of how parentheses are used, I will still cover secondary constructor and how parentheses are used in secondary constructor.

As you already know, any class can only have one primary constructor but can have multiple secondary constructors. We can use secondary constructor to instantiate a class which is no different from the primary constructor, except that it will have a function body.

In Kotlin, when the class has primary constructor and secondary constructor, all the secondary constructor(s) must call the primary constructor directly or indirectly. Let me use the following code to explain:

```
class Student (val sno: String, val grade: Int, name: String, age: Int) : Person(name, age) {
    constructor(name: String, age: Int) : this("", 0, name, age) {
    }

    constructor() : this("", 0)
}
```

The secondary constructor is declared with the keyword `constructor` and here we define two secondary constructors: the first secondary constructor takes name and age parameter, and then it calls primary constructor by using `this` keyword and initializes sno and grade; the second secondary constructor does not have any parameter and it uses `this` keyword to call the first secondary constructor we just defined and initializes name and age. The second secondary constructor indirectly calls the primary constructor, and thus, it is still legit.

Now we have 3 ways to instantiate `Student` class. They are parameterless constructor, constructor with 2 parameters, and constructor with 4 parameters as shown below:

```
val student1 = Student()
val student2 = Student("Jack", 19)
val student3 = Student("a123", 5, "Jack", 19)
```

We just discussed the usage of secondary constructor, but we still haven't covered the advanced use of parentheses yet.

Now let's see a very special case: there is only secondary constructor but no primary constructor. This is very rare but legit case in Kotlin. When a class does not explicitly define primary constructor but instead defines secondary constructor, then it is still legit, and we can use the following code to demonstrate:

```
class Student : Person {
    constructor(name: String, age: Int) : super(name, age) {
    }
}
```

Notice the differences here. First, `Student` class does not explicitly define the primary constructor, and since it defines secondary constructor, there is no primary constructor inside the `Student` class. Since there is no primary constructor, there is no need to add the parentheses when inheriting the `Person` class. And this should clarify when to add parentheses and when not to do so which is confusing for a lot of beginners.

Besides this, since there is no primary constructor, the secondary constructor can only call the parent constructor directly and in the code above, this gets replaced by super which is similar to Java and easy to understand. I won't go further on this.

In this section, we take a deep dive into inheritance and constructor in Kotlin which is difficult to understand for many starters.

### 2.5.3 *Interface*

The last section was a bit long and complicated and may be a little bit challenging. This section will be much simpler since interface in Kotlin is almost identical as in Java.

Interface is an important concept in polymorphism. We all know that Java only supports single inheritance which means that any class can only inherit one super class while being able to implement any number of interfaces. Kotlin is exactly like Java in this.

We can define a series of abstract functions and then the concrete classes will implement these abstract functions. Let's use code to demonstrate this. First, create a Study interface and then define a few study-related functions. Right click com.example.helloworld package- > New- > Kotlin File/Class and then type in "Study", select "Interface" for Kind option.

Then add a few functions related to study in the Study interface, notice that the function body or the implementation of the function is not required as shown below:

```
interface Study {  
    fun readBooks()  
    fun doHomework()  
}
```

Then we can get the Student class to implement the interfaces of Study. Here I modified the original Student class a little bit to highlight the difference between inheritance and implementing interface:

```
class Student(name: String, age: Int) : Person(name, age), Study {  
    override fun readBooks() {  
        println(name + " is reading.")  
    }  
  
    override fun doHomework() {  
        println(name + " is doing homework.")  
    }  
}
```

In Java, extends keyword is used for inheritance and implements keyword is used for implementing interface. In Kotlin both inheritance and implements are done by



**Fig. 2.20** Call the function in the interface

using colon and between them are commas. Here Student class inherits Person class and implements Study interface. There are no parentheses after interface since there is no constructor to call.

The Study interface declares two abstract functions `readBooks()` and `doHomework()`, thus, Student class has to implement these two functions. Kotlin uses `override` keyword to override the function in parent class or to implement the abstract function in the interface. Here we simply print a string in the console.

Now we can call the functions in the interface as shown below:

```
fun main() {
    val student = Student("Jack", 19)
    doStudy(student)
}

fun doStudy(study: Study) {
    study.readBooks()
    study.doHomework()
}
```

I made it a little bit more complicated than necessary to demonstrate polymorphism. First, create an instance of Student class. We can directly call the `readBooks()` and `doHomework()` functions of the instance, but I didn't do so. Instead, an instance of Student is passed to the `doStudy()` function. This function will take an argument of type Study and since the Student class implemented Study interface, the Student instance can be passed to `doStudy()` function without any issue. Then the code calls the `readBooks()` and `doHomework()` implementations of the Study interface. This is called interface-oriented programming or polymorphism. Let's run the code and the result is shown as in Fig. 2.20.

And that is pretty much everything for interface in Kotlin. Isn't it simple? There is one more thing, though. In order to make the interface more flexible, Kotlin added another extra functionality which is to allow default implementation for the function declared in the interface. Java started to support this since JDK1.8. So overall, Kotlin and Java are pretty much the same in interface design.

Now let's see how to have default implementation for the function in the interface. Let's change the Study interface as follows:



**Fig. 2.21** Call the default implementation of the interface

```
interface Study {
    fun readBooks()

    fun doHomework() {
        println("do homework default implementation.")
    }
}
```

We added the function body to `doHomework()` which will print a string in the console. If a function in the interface has function body, then the function body is its default implementation. Now when a class tries to implement the `Study` interface, `readBooks()` won't need to be implemented by the class and if not implemented, the default implementation will be used.

Get back to the `Student` class. If we delete `doHomework()`, there will be no error but deleting `readBooks()` will cause error. And let's delete the `doHomework()` function and rerun the `main()` function. The result is as expected and shown in Fig. 2.21.

Now you've already learned some of the most important topics in OOD in Kotlin. Let's start to learn something that is different from Java—the scope of functions.

In Java, we have four values of scope for functions which are public, private, protected, and default (no keyword). There are 4 types of scope in Kotlin too which are public, private, protected, and internal. We can add the keyword before the `fun` keyword to apply the scope. I'm going to talk about the difference between Java and Kotlin.

The scope of private in these two languages is the same which means that the function is only accessible inside the class. Public is mostly the same which means accessible to all classes. However, in Kotlin public is the default scope and in Java package private is the default scope. We didn't add any scope keywords for our functions before and by default they are public. For Java, protected means accessible to the current class, sub class, and the classes in the same package. However, for Kotlin it is only accessible to the current class and sub class. Kotlin eliminates the package private scope (default in Java), and instead introduced another accessibility concept which is accessible by classes in the same module and the keyword for it is internal. For example, if we implement a module for other developers to use, and if certain functions can only be called within the module, then we can declare functions to be internal. We will learn modular development in the last chapter of this book.

**Table 2.2** compares the difference between scope in Java and Kotlin

| Decorator | Java  | Kotlin                      |
|-----------|---|-----------------------------|
| Public    | All   | All(default)                |
| Private   | Current class                                       | Current class               |
| Protected | Current class, sub class, class in the same package | Current class and sub class |
| Default   | Class in the same package (by default)              | N/A                         |
| Internal  | N/A   | Class in the same module    |

Table 2.2 compares more visually the differences between function visibility modifiers in Java and Kotlin.

#### 2.5.4 Data Class and Singleton

In more formal frameworks, data classes play a very important role. They provide the data models for programming logics and map the data in the database or on the server to memory. You've probably heard frameworks like MVC, MVP, MVVM, and M here are data classes.

For data classes, usually you need to override the equals(), hashCode(), toString() functions. The equals() function is used to determine if two data class instances are equal. The hashCode() function needs overriding otherwise, HashMap, HashSet, and other hash-related classes won't work. The toString() function can be used to provide more useful information about the class, for example, if we try to print a class, by default it will print the memory address of the class which is not useful in most cases.

Now let's create a simple cellphone class which has brand field and price field. If we use Java to implement this data class, it should be something like the following code:

```
public class Cellphone {
    String brand;
    double price;

    public Cellphone(String brand, double price) {
        this.brand = brand;
        this.price = price;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Cellphone) {
            Cellphone other = (Cellphone) obj;
            return other.brand.equals(brand) && other.price == price;
        }
        return false;
    }
}
```

```

@Override
public int hashCode() {
    return brand.hashCode() + (int) price;
}

@Override
public String toString() {
    return "Cellphone(brand=" + brand + ", price=" + price + ")";
}

```

It already looks cluttered even without any real business logic and just serves as a container for data. Kotlin will provide a much simpler way to implement the same functionality. Right click com.example.helloworld package ->New ->Kotlin File/ Class, type in “Cellphone” in the dialog and select “Class” for the Kind option. Then type in the following code in the class:

```
data class Cellphone(val brand: String, val price: Double)
```

And that's it! With one line of code! The magic happens because of the data keyword. When the class is declared with the data keyword, you explicitly show your intent to make it a data class and Kotlin will automatically generate the equals(), hashCode(), toString(), and other frequently used functions and can reduce the workload of development drastically.

Also notice that when there is no code inside the class, the braces at the end can be omitted.

Now let's do some test around this data class and type code as follows:

```

fun main() {
    val cellphone1 = Cellphone("Samsung", 1299.99)
    val cellphone2 = Cellphone("Samsung", 1299.99)
    println(cellphone1)
    println("cellphone1 equals cellphone2 " + (cellphone1 ==
cellphone2))
}

```

Here we create two Cellphone instances and print the first instance, then, compare if these two instances are equal. Let's run the code and the result is shown as in Fig. 2.22

Apparently, Cellphone data class works as expected. However, if there is no data keyword in front of the Cellphone class, the result will be quite different and you can try it out to see what will happen.

Now let's look at another special functionality Kotlin provides—singleton class.

You've probably heard about the Singleton Pattern, which is one of the most widely used design patterns that can prevent creating duplicated instances. For instance, if we only want to create a single instance globally, then we should use



**Fig. 2.22** Result of testing the Cellphone data class

Singleton Pattern. There are many ways to implement a singleton and here is a common way to do it in Java:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public void singletonTest() {
        System.out.println("singletonTest is called.");
    }
}
```

It's very straightforward. In order to prevent other classes to create instances of the Singleton class, we make the scope of the constructor private and then provide a static function getInstance() to get the instance of the Singleton. Then in the getInstance() method, if there is no instance of the class, then we create a new instance and return it; otherwise, we just return the existing one. This is how Singleton Pattern works.

If we want to call the method inside the singleton, for example the singletonTest() method, then we can write code like code below:

```
Singleton singleton = Singleton.getInstance();
singleton.singletonTest();
```

Although it is simple to implement singleton in Java, Kotlin provides a better way to do so. As the design of data class, Kotlin will also hide the logic for some frequently used functions and provides an elegant way to implement the class.

It is extremely easy to create a singleton class in Kotlin, you just need to replace the class keyword to object keyword. Now let's create a Singleton class in Kotlin, right click com.example.helloworld package ->New ->Kotlin File/Class, type in

“Singleton” in the dialog and select “Object” for Kind option, click “OK” to finish creation and the following code will be generated.

```
object Singleton {  
}
```

Now, Singleton is a singleton and we can write functions inside this class, for instance, a singletonTest() function:

```
object Singleton {  
    fun singletonTest() {  
        println("singletonTest is called.")  
    }  
}
```

As you can see, we don't need to create private constructor, nor static method like getInstance(). All we need to do is to declare the class with object keyword and then we get a singleton class. To use the method in the singleton is just like using static method in Java:

```
Singleton.singletonTest()
```

It looks like we're calling a static method, but under the hood, Kotlin created an instance of the Singleton class and can guarantee there will be only one Singleton instance globally.

And that's all for OOD in Kotlin. This chapter is particularly important because in real world, almost all the projects are OOP.

## 2.6 Lambda Expression

Lambda expression sounds like a new concept, but it got support from many modern advanced programming languages a while ago. However, Java only started to support Lambda expression since JDK1.8. Thus, a lot of the programs written by Java (which also include some Android apps) didn't use Lambda expression.

Lambda expression has been supported by Kotlin since the very first version and is a powerful feature in Kotlin. I personally feel that Lambda expression is the soul of Kotlin. However, this is just an introductory chapter of Kotlin, and I cannot cover everything about Lambda expression in this section. We will cover some basics about Lambda expression and will cover more advanced topics like higher-order functions, DSL, etc. in successive chapters.

### 2.6.1 Creation and Iteration of Collection

The APIs of collections are perfect examples we can use to learn about Lambda expression, but before that, we must learn how to create collection objects.

Usually when we talk about collections, we are talking about List and Set, and key-value data structures like Map. List, Set, and Map are interfaces in Java. The ArrayList class and LinkedList class are the most used classes that implement List. HashSet is the most used class that implements Set. HashMap is the most used class that implements Map.

Now, think about this product requirement: create a collection of fruit names. Using Java, you might instantiate an instance of ArrayList and then add the names in the ArrayList one by one. Of course, we can do the same thing in Kotlin as the code below:

```
val list = ArrayList<String>()
list.add("Apple")
list.add("Banana")
list.add("Orange")
list.add("Pear")
list.add("Grape")
```

Kotlin is a concise language and it provides a listOf() method to simplify the initialization of lists as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
```

Yes! It can be done by one line of code!

Remember we mentioned that for-in can be used to not only iterate the interval but also iterate the collections? Now let's try it out to iterate the fruit names list, and add the code below to main() function:

```
fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
    for (fruit in list) {
        println(fruit)
    }
}
```

Run the app, and the result should be as shown in Fig. 2.23.

Notice that listOf() function will create an immutable collection which means that this list is read-only and we cannot add, update, or remove items in the list.

The reason behind this is the same as the design of val keyword and class is not inheritable by default. What if we need to create a mutable collection? Then we can just use mutableListOf() function, as shown below:

**Fig. 2.23** Iterate the Collection**Fig. 2.24** Iterate elements in Collection

```
fun main() {
    val list = mutableListOf("Apple", "Banana", "Orange", "Pear",
"Grape")
    list.add("Watermelon")
    for (fruit in list) {
        println(fruit)
    }
}
```

Here we use `mutableListOf()` to create a mutable list and then add a new fruit name in this list. Finally, we use `for-in` to iterate the list. Rerun the app. The result should be shown as in Fig. 2.24

We can see that the newly added fruit has been printed out successfully.

Set collection is used almost the same way as List. We can use `setOf()` and `mutableSetOf()` for corresponding functionality. For instance, we can have the code as below:

```
val set = setOf("Apple", "Banana", "Orange", "Pear", "Grape")
for (fruit in set) {
    println(fruit)
}
```

It is worth noting that under the hood, Set uses hash to store the data. Thus, the elements in the set don't have order which is the biggest difference from List. Of course, this is about data structure and I won't cover further.

Lastly, let's take a look at how to use Map. Map is a key-value pair data structure and is quite different from List and Set. The convention to use Map is to instantiate an instance of HashMap, then add the key-value pairs to the map one by one. For example, we can give each fruit a corresponding index as shown below:

```
val map = HashMap<String, Int>()
map.put("Apple", 1)
map.put("Banana", 2)
map.put("Orange", 3)
map.put("Pear", 4)
map.put("Grape", 5)
```

This is very close to Java code and should be easy to understand. However, in Kotlin, it is not recommended to use get() and put() to read and write Map. The recommended way to read and write is as shown below:

```
map["Apple"] = 1
val number = map["Apple"]
```

The above code can be optimized as shown below:

```
val map = HashMap<String, Int>()
map["Apple"] = 1
map["Banana"] = 2
map["Orange"] = 3
map["Pear"] = 4
map["Grape"] = 5
```

Apparently, this is not the most concise way. Since Kotlin provides mapOf() and mutableMapOf() to help use map, we can have code below to do the same thing:

```
val map = mapOf("Apple" to 1, "Banana" to 2, "Orange" to 3, "Pear" to
4, "Grape" to 5)
```

The to word seems like a keyword to connect the key value pair. However, to is actually not a simple keyword but an infix function which we will cover in detail in Chap. 9.

Lastly, let's look at how to iterate elements in Map. We can still use for-in as shown below:

```
fun main() {
    val map = mapOf("Apple" to 1, "Banana" to 2, "Orange" to 3, "Pear" to
4, "Grape" to 5)
    for ((fruit, number) in map) {
        println("fruit is " + fruit + ", number is " + number)
    }
}
```



**Fig. 2.25** Iterate elements in Map

Compared with iterating Set, here we need to use key-value pair in parentheses of for-in, then during the iteration, each element's key and value will be assigned to these two variables. Rerun the app, and the the result is as shown in Fig. 2.25.

That's it for creating and iterating collections and now let us get into Lambda expression with functional APIs of collections.

### 2.6.2 Functional APIs of Collections

There are many functional APIs and it is not my intention nor possible to learn all of them here. Instead, I will use a few examples to understand the syntax of functional APIs which has the same syntax as Lambda expression.

How do we find the fruit that has the longest name? This is a simple problem that can be solved with a few approaches and the code below is a very straightforward way to do so:

```

val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
var maxLengthFruit = ""
for (fruit in list) {
    if (fruit.length > maxLengthFruit.length) {
        maxLengthFruit = fruit
    }
}
println("max length fruit is " + maxLengthFruit)

```

Code above is very straightforward. However, we can make it more concise by using the functional APIs as below:

```

val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
val maxLengthFruit = list.maxBy { it.length }
println("max length fruit is " + maxLengthFruit)

```

We just need one line of code to accomplish the same goal. The syntax here looks confusing at first glance but should be easy to understand after we finish this section.

So, what is Lambda? In plain words, Lambda is a few lines of code that can be used as an argument. This is very useful because we used to can only pass data as argument, but Lambda allows to pass a few lines of code as argument. I use a few lines twice now, but how many lines of code can be considered as a few lines? Kotlin actually doesn't have limitation on this, but it is recommended to not write very long code in Lambda expression for the sake of readability. The syntax of Lambda expression looks like this:

```
{param name 1: param type, param name 2: param type -> function body}.
```

Notice that we have `->` between the params list and function. We can write any lines of code in the function though it is recommended to keep the code short and the result of the last line of code will be the return value.

This is the complete form of Lambda expressions; however, it has simpler forms which are harder to understand, and I will show you step by step how the expression evolves to be more and more simple.

Back to the code that used functional API to find the longest fruit name, the functional API syntax looks rather odd. However, `maxBy` is just a normal function that takes a Lambda expression as param and when we iterate the collection, every value will be passed as argument to the Lambda expression.

Now let's use functional API with the complete form as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
val lambda = { fruit: String -> fruit.length }
val maxLengthFruit = list.maxBy(lambda)
```

We can see that `maxBy` function takes a Lambda expression as argument, and this Lambda expression is using the syntax we just mentioned, so it should be easy to understand.

Now let's start to simplify it step by step.

First, we do not need the `lambda` variable, instead, we can pass the Lambda expression directly to `maxBy` function, after this, code should look like:

```
val maxLengthFruit = list.maxBy({ fruit: String -> fruit.length })
```

Then in Kotlin, Lambda expression can be moved to be outside of function parentheses when it is the last param of the function, thus code can be simplified to the following form:

```
val maxLengthFruit = list.maxBy() { fruit: String -> fruit.length }
```

Next, if the Lambda expression is the only param then we can remove the parentheses of the function. Now it looks like:

```
val maxLengthFruit = list.maxBy { fruit: String -> fruit.length }
```

There is still room to simplify! Because of the type inference mechanism, there is no need to explicitly define the type of the param for most cases, thus we can remove the type and simplify as below:

```
val maxLengthFruit = list.maxBy { fruit -> fruit.length }
```

Lastly, when there is only one param in the param list of Lambda expression, then there is no need to declare a name for the param but use it keyword to replace. After this, code should be simplified to the following:

```
val maxLengthFruit = list.maxBy { it.length }
```

And we just get the code that is the same as we saw earlier. Now the syntax here should be much easier to understand, right?

As we mentioned previously, our focus is to learn the syntax of the functional APIs through example, so that readers can quickly learn other functional APIs in a short time.

Next let's go over a few other widely used functional APIs in collections which should be super easy for you to understand now.

The map function is another widely used functional API which will maps every value in the collection to another value and the mapping rules are defined in the Lambda expression. This function will create a new collection. For example, if we want to let all the letters in fruit names to be upper case then we can use the code below:

```
fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
    "Watermelon")
    val newList = list.map { it.toUpperCase() }
    for (fruit in newList) {
        println(fruit)
    }
}
```

We can see that the Lambda expression in the map function changed the specified word to upper case. Rerunning the app will show result as shown in Fig. 2.26.

The map function is very powerful and supports arbitrary mapping for the elements in the collection and the code above is just a simple example to demonstrate this capability. You can update the names to lower cases, get the first character of the word or calculate the length of the word, and so on. All you need to do is to write the conversion logic in Lambda expression.

Next, let's look at filter function which can be used separately or together with the map function.



**Fig. 2.26** Map Fruit name to capital letters



**Fig. 2.27** Filter Based on Fruit Name Length

For instance, if we only want to keep fruit whose name has no more than 5 letters, then we can use filter function to do so like shown below:

```
fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
    val newList = list.filter { it.length <= 5 }
        .map { it.toUpperCase() }
    for (fruit in newList) {
        println(fruit)
    }
}
```

We chain filter and map function together and use Lambda expression to apply the constraint of no more than 5 letters. Rerun the app, the result should be as in Fig. 2.25.

It is worth noting that here we used filter first and then use map. You can also use map function first and then use filter, however, the efficiency will be negatively affected. This is because it is not necessary to map all the elements and then do filtering. If we can filter first, there will be less items to map thus will be much more efficient.

Next, we will cover any and all functions. The any function is used to determine if any element in the collection that meets the condition. All function is used to determine if all elements in the collection meet the condition. Let's look at the simple example below:



**Fig. 2.28** Result of running any and all functions

```
fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape",
"Watermelon")
    val anyResult = list.any { it.length <= 5 }
    val allResult = list.all { it.length <= 5 }
    println("anyResult is " + anyResult + ", allResult is " + allResult)
}
```

Here we still have the same condition check in Lambda. Then any will check if there is any word in the collection that has no more than 5 letters and all will check if all the words in the collection have no more than 5 letters. Rerun the app and the result should be as shown in Fig. 2.28.

That's all for Lambda expression syntax and functional APIs. There are many other functional APIs, but if you understand the syntax, it will be very easy to learn them by reading the documentation.

### 2.6.3 Java Functional API

We can call Java methods in Kotlin with functional APIs though with some limitations. That is, if we call a Java method in Kotlin and this method takes only one functional interface as argument, then we can use functional API. Java functional interface is an interface that contains only one abstract method. If there are multiple abstract methods in an interface, then it is not a functional interface.

If you still feel confused, then let's use an example to explain this. The Runnable interface is a widely used functional interface in Java which has only one abstract method—run():

```
p public interface Runnable {
    void run();
}
```

As mentioned above, any Java method that takes Runnable param should be able to use functional API, then what are the Java methods that can take Runnable as param? There are many such use cases but Runnable interface usually is getting used in thread and let's use the Java Thread class to demonstrate it.

The constructor in Thread class takes a Runnable param and we can use the following Java code to create and run this thread:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}).start();
```

Notice that anonymous class syntax is applied here. We create an instance of anonymous class that implements Runnable interface and pass it to the constructor of the Thread class. Then use start() method in Thread class to run this thread instance.

The Kotlin version could look like the following:

```
Thread(object : Runnable {
    override fun run() {
        println("Thread is running")
    }
}).start()
```

There are some syntax differences for anonymous class in Kotlin compared with Java as Kotlin eliminates new keyword. Thus, it uses object to create an instance. Now let's apply the rules we mentioned above to try to simplify it as shown below:

```
Thread(Runnable {
    println("Thread is running")
}).start()
```

Now it is more concise and not confusing at all. This is because Runnable class only has one abstract method and even we do not explicitly override run() method, Kotlin knows that the Lambda expression after Runnable is the implementation of run() method.

Also, if we only have one functional interface as param then we can omit the interface name too as shown below:

```
Thread({
    println("Thread is running")
}).start()
```

And that's not the end of it yet. Like the functional APIs in Kotlin, when the Lambda expression is the last param of a method, then the Lambda expression can be moved to the outside of the parentheses. And if the Lambda expression is the only param of the method, then parentheses can be omitted altogether. The final version of simplified code is as below:



**Fig. 2.29** Result of Java functional API

```
Thread {
    println("Thread is running")
}.start()
```

Copy and paste the code above into the main() method and rerun the code. You should see result as shown in Fig. 2.29.

You might think that since all the codes are written in Kotlin in this book, Java functional API should be rarely used. But that's not the case as Android SDK is written in Java and when we need to call interfaces in SDK, we will have a chance to apply the syntax learned here.

For example, OnClickListener is a widely used click event interface which is defined as below:

```
public interface OnClickListener {
    void onClick(View v);
}
```

We can see that this is another functional interface. Imagine we have a button instance and then if we register the click event in Java, we can use the code below:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

In Kotlin, we can use functional API to simplify the code to the following:

```
button.setOnClickListener { }
```

As you can see it is more concise to apply functional API and we will use this very frequently in later chapters.

Lastly, notice that all the Java functional API usages here are all about calling Java method in Kotlin and the functional interface must be defined in Java. This is because Kotlin has higher-order functions to implement even more powerful

functional APIs thus there is no need to use functional interface as Java does. We will cover higher-order functions in Chap. 6.

## 2.7 Null Safety

I saw some reports mentioned that the most frequent exception happening in Android is NullPointerException(NPE). NPE is not a problem only for Android but also other systems. The reason is that null pointer is runtime exception that may not always be avoidable by compiler checks. It usually depends on developers to find out the possible cases of NPE which is too challenging even for the most experienced developer.

Let's look at the simple Java code below:

```
public void doStudy(Study study) {  
    study.readBooks();  
    study.doHomework();  
}
```

This is the doStudy() method we created in Sect. 2.5.3 and I converted it into Java. The code here is very straightforward. It takes a Study param and calls readBooks() and doHomework() on this instance.

Is the code here safe? Not really. If we pass in null to doStudy(), then apparently we will have NPE here. Thus, a safer way to do so is to check nullability before calling the method of the object like shown below:

```
public void doStudy(Study study) {  
    if (study != null) {  
        study.readBooks();  
        study.doHomework();  
    }  
}
```

Now no matter what value the argument is, code here won't throw NPE.

From this simple example, we can conclude that in large-scale project, it would be impossible to avoid all the potential NPEs, and that's the reason why NPE is the most likely reason for crashes.

### 2.7.1 Nullable Type

Kotlin solved this problem in a systematic way. Its compiler will check nullability during compilation, and thus eliminates the possibility of NPE. Although it makes coding more difficult, Kotlin also provides a suite of tools to help us.

**Fig. 2.30** Pass null to doStudy()

```

fun main() {
    doStudy(study: null)
}

fun doStudy(study: Study) {
    study.readBooks()
    study.doHomework()
}

```

Back to the previous doStudy() method, let's convert it back to Kotlin as shown below:

```

fun doStudy(study: Study) {
    study.readBooks()
    study.doHomework()
}

```

This looks very similar to Java version at first glance; however, it doesn't have NPE risk at all. This is because by default, all the variables and params cannot be null thus study cannot be null and we can call any method in it without worrying about NPE. Now if you pass null to doStudy() method then you will see the error information like shown in Fig. 2.30.

Basically, what happens here is that Kotlin will check nullability during compilation, and if there is any possibility of NPE, then error message will show up and compilation will fail. Thus, there is no chance for NPE to happen.

Now you might ask how can we make all params and variables not null since in our business logics, we usually need to make certain params or variables to be null. Well, Kotlin actually provides another type of system that allows nullable, and under this system, we need to check nullability for any possible nulls otherwise it won't compile.

So, what is nullable type? It can be defined with? after the type. For instance, Int is nonnull integer type, while Int? is nullable integer type. String is nonnull string type, while String? is nullable string type.

Back to the doStudy() function, if we need to make the param to be nullable then we need to declare the param to Study? instead of Study as shown in Fig. 2.31.

You won't see error information with doStudy() in main() when you pass null to doStudy() with this change. However, you will find that there are some red lines under the readbooks() and doHomework(), why this happens?

The reason is actually fairly simple, this is because the param now is Study? type which is nullable and it is possible that readBooks() and doHomework() will trigger NPE, thus Kotlin won't allow it to compile.

To resolve the issue we need to eliminate the possibility of NPE which can be done by simply adding nullability check as shown below:

**Fig. 2.31** Allow Study param to be nullable

```
fun main() {
    doStudy(study: null)
}

fun doStudy(study: Study?) {
    study.readBooks()
    study.doHomework()
}
```

```
fun doStudy(study: Study?) {
    if (study != null) {
        study.readBooks()
        study.doHomework()
    }
}
```

Compilation will succeed now. Meanwhile, code here will be NPE free.

Now we know what is nullable types and NPE check, however the approach mentioned above is very verbose as we need to write lots of if statements. To solve the problem, Kotlin provides a suite of tools for us to avoid NPE while keeping the code concise. Let's look at them one by one.

### 2.7.2 Nullability Check Tools

The first one is the ?. operator. This operator will only allow the method following the operator gets called when the object before it is nonnull. Use the following code as example:

```
if (a != null) {
    a.doSomething()
}
```

It can be simplified to

```
a?.doSomething()
```

Let us apply this to our doStudy() method as shown below:

```
fun doStudy(study: Study?) {
    study?.readBooks()
    study?.doHomework()
}
```

Now we don't have if statement in the method anymore. And you will truly appreciate this when you have lots of places to check nullability.

Let us look at another commonly used operator ?: . This operator can take two expressions and if the expression on the left side is not null, then return the result on the left side. Otherwise, return the result of expression on the right side. Use the code below as an example:

```
val c = if (a != null) {  
    a  
} else {  
    b  
}
```

By using ?: operator, it can be simplified to:

```
val c = a ?: b
```

Next let's use them together. We can write the following code to get a function that can return the length of a string:

```
fun getTextLength(text: String?) : Int {  
    if (text != null) {  
        return text.length  
    }  
    return 0  
}
```

Basically, we check if the string is null, if not evaluate its length and return the length, otherwise return 0.

Then we can apply the operator we just learned and simplify the code as given below:

```
fun getTextLength(text: String?) = text?.length ?: 0
```

Here we used ?. and ?: together. Since text is nullable, so we need to use ?. when trying to access length field and when text is null, text?.length will return null and ?: will return 0.

However, Kotlin nullability check is not always smart enough. Sometimes even after we rule out the possibility of NPE, Kotlin compiler won't be able to figure out the logic and still fails. Look at the following example code:

```
var content: String? = "hello"  
  
fun main() {  
    if (content != null) {  
        printUpperCase()  
    }  
}
```

```

        }
    }

fun printUpperCase() {
    val upperCase = content.toUpperCase()
    println(upperCase)
}

```

In the code above, we define a nullable global variable content, then in main() we check nullability of content, and only when content is not null will we call printUpperCase(). After this is in printUpperCase(), content will be converted to all capital letters and gets printed.

Logically, there is no risk for NPE here, however it will fail to compile. This is because the printUpperCase() method doesn't know content variable has been checked for nullability thus when try to call toUpperCase() method of content, compiler will think there is risk for NPE.

Under such circumstance, if we want to compile without checking nullability again we can add!! after the object like shown below:

```

fun printUpperCase() {
    val upperCase = content!!.toUpperCase()
    println(upperCase)
}

```

This risky syntax tells Kotlin that you're sure that the object here won't be null and you can take the responsibility of NPE.

So!! operator helps us to compile, but you should use it only when you cannot find a better way. Because NPE is very likely to happen when you're very confident that NPE will never happen.

Last, let's look at a special tool—let. It is not an operator nor a keyword but a function. This function provides the interface for functional API and uses the original object that is calling this function as param to pass into Lambda expression. Here is an example:

```

obj.let { obj2 ->
    // business logic
}

```

Here I use obj2 to prevent duplicating naming of variable, but it is the same object as obj. Obj itself gets passed into Lambda expression and Lambda will get executed immediately.

let function is a standard function (scope function) in Kotlin and we will cover more on Kotlin standard function.

You might ask what does let have to do with nullability? It can be used to check nullability when used together with ? ..

Back to doStudy() function, code is as shown below:

```
fun doStudy(study: Study?) {
    study?.readBooks()
    study?.doHomework()
}
```

Though with `?.` it can compile, it is a little bit verbose here since if we just need to check null once why do we need to use `?.` twice? The above code is equivalent to the following code by using `if` statement:

```
fun doStudy(study: Study?) {
    if (study != null) {
        study.readBooks()
    }
    if (study != null) {
        study.doHomework()
    }
}
```

Now let's use `let` to optimize the code as shown below:

```
fun doStudy(study: Study?) {
    study?.let { stu ->
        stu.readBooks()
        stu.doHomework()
    }
}
```

Here, when `study` object is not null then `let` function will be evaluated. Then `let` will pass `study` as argument to the Lambda expression, and since `study` is not null, then we don't need to add redundant nullability check anymore.

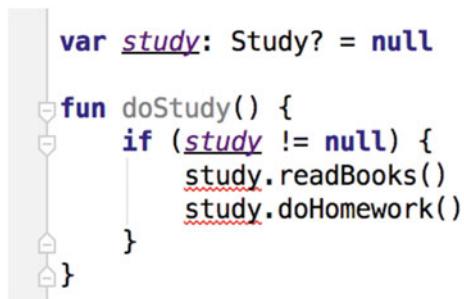
Also, remember that when there is only one param in the Lambda expression param list, we don't need the param name and can use it to represent the param. So the code above can be further optimized to the following:

```
fun doStudy(study: Study?) {
    study?.let {
        it.readBooks()
        it.doHomework()
    }
}
```

Before we conclude this section, let's see how `let` function solves the problem of nullability check for global variables which cannot be done by using `if` statement. For example, if we declare the `study` variable as a global variable then, even if we use `if` statement to check nullability, code will still not compile as shown in Fig. 2.32.

This is because global variables can be mutated by other thread at any time, thus even with `if` statement to check nullability, NPE risk still exists.

**Fig. 2.32** Use if statement to check nullability for global variable



OK, that is everything we have for Kotlin nullability check. By applying the knowledge here, you should be able to write robust and NPE free apps.

## 2.8 Kotlin Tricks

You've learned lots of knowledge about Kotlin up to now and should be able to handle daily Kotlin development. Before we finish this chapter let's learn a few tricks in Kotlin which are more helpful than they sound.

### 2.8.1 *String Interpolation*

String interpolation is the feature that I personally feel should be supported in Java since lots of advanced programming languages have this very useful feature. I don't know why Java doesn't support it even today.

But luckily, Kotlin supports this feature since the very beginning. In Kotlin we don't need to concatenate the strings using very clumsy ways as in Java, instead, we can put expression in string and even building very complex string can be very simple task.

Up to now, we've only used + operator to concatenate strings. But after this section, we will say goodbye to + operator.

The syntax of string interpolation in Kotlin is shown as code below:

```
"hello, ${obj.name}. nice to meet you!"
```

First, we can see that Kotlin allows to use \${} expression and when this string is getting used, the expression will be evaluated and will be replaced by the evaluated result.

Also, if there is only one variable, then we can omit the braces like shown below:

```
"hello, $name. nice to meet you!"
```

Let's use an example to see how it can optimize our code. In Sect. 2.5.4, we used Java to write a Cellphone data class, and its `toString()` method was written with verbose string concatenation method as shown below:

```
val brand = "Samsung"
val price = 1299.99
println("Cellphone(brand=" + brand + ", price=" + price + ")")
```

We can see that there are four `+` operators which is not easy to write and readability is also very poor.

However, with string interpolation, it will be much easier, as shown below:

```
val brand = "Samsung"
val price = 1299.99
println("Cellphone(brand=$brand, price=$price)")
```

Apparently, the code above is much easier to read and write and is the recommended way in Kotlin. This trick should help you a lot in real project.

## 2.8.2 *Function Default Arguments*

Next let's learn another useful trick – function default arguments.

As I mentioned before, secondary constructor is rarely used in Kotlin because Kotlin provides default value to function arguments which can functionally replace the secondary constructor.

To be more specific, we can assign a default value to any argument and when this function gets called, the caller doesn't need to pass value to this argument and if there is no value passed in then the default value of the argument will be used.

To use function default arguments is simple, as shown below:

```
fun printParams(num: Int, str: String = "hello") {
    println("num is $num, str is $str")
}
```

Here we assign a default value to the second param of `printParams()` method. When we call `printParams()`, passing value to the second param is optional and if no argument gets passed in, default value will be used.

Now let's test it by modifying the code as shown below:

```
fun printParams(num: Int, str: String = "hello") {
    println("num is $num, str is $str")
}
```



**Fig. 2.33** Result of using default value for argument

**Fig. 2.34** Type Mismatch  
Error

```

fun printParams(num: Int = 100, str: String) {
    println("num is $num , str is $str")
}

fun main() {
    printParams(num: "world")
}

```

Type mismatch.  
Required: Int  
Found: String

```

fun main() {
    printParams(123)
}

```

Notice that the second param does not get any value. Running the code will yield result as shown in Fig. 2.33.

printParams() method uses the default argument. The above code set the default value for the last param, what will happen if we set default value for the first param?

```

fun printParams(num: Int = 100, str: String) {
    println("num is $num , str is $str")
}

```

How to allow num use the default value? The previous format will fail to compile because of type mismatch error as shown in Fig. 2.34.

Don't worry. Kotlin provides named argument to solve this problem which doesn't even need to follow the order of params. For instance, to use printParams () we can write code like below:

```
printParams(str = "world", num = 123)
```

Then order really doesn't matter since apparently Kotlin will use name to match the argument. Also, we can even omit num param with named argument as shown below.



**Fig. 2.35** Result of using default value for param num

```
fun printParams(num: Int = 100, str: String) {
    println("num is $num , str is $str")
}

fun main() {
    printParams(str = "world")
}
```

Rerun the application, the result is shown as in Fig. 2.35.

So how can function default arguments replace secondary constructor?

Here is the code we wrote in secondary constructor section:

```
class Student(val sno: String, val grade: Int, name: String, age: Int) :
    Person(name, age) {
    constructor(name: String, age: Int) : this("", 0, name, age) {
    }

    constructor() : this("", 0) {
    }
}
```

The code above has one primary constructor and two secondary constructors. The secondary constructors are used to instantiate the Student class with less params. The parameterless constructor will call the constructor with two params and assign values to these two params. The constructor with two params will call the primary constructor with four params and assign values to the missing two params.

However, Kotlin has better solution. We can just write one primary constructor and use function default arguments to do the same thing. The code is shown as below:

```
class Student(val sno: String = "", val grade: Int = 0, name: String =
"", age: Int = 0) :
    Person(name, age) { }
```

After assigning default values to each parameter, we can use any combination of params to instantiate Student class which includes the cases of the two secondary constructors.

Now you should see how useful the function default arguments are.

## 2.9 Summary

We learned the most important topics in Kotlin in this chapter which include variables, functions, flow control, OOP, Lambda expressions, nullability check, and so on. Although this is not everything Kotlin has to offer, I can confidently say that you're ready to use Kotlin for Android development now.

Thus, starting from the next chapter, we will focus on Android development. I will still cover some advanced Kotlin topics in each chapter that are related to that chapter so that you can make progress in both Kotlin and Android. Now get some rest and let's continue our journey!

# Chapter 3

## Start with the Visible: Explore Activity



In Chap. 1, you've already learned how to create the first Android project. Apparently, that is not enough and it's time to learn something more advanced. Where should we begin? Imagine that you've already created a high-quality app and want to introduce it to your very first user, where should you begin? Of course, you will start with the user interface! This is because users don't care about the algorithms, system design. They are only interested in what they see, so let's begin with what's visible.

### 3.1 What Is Activity?

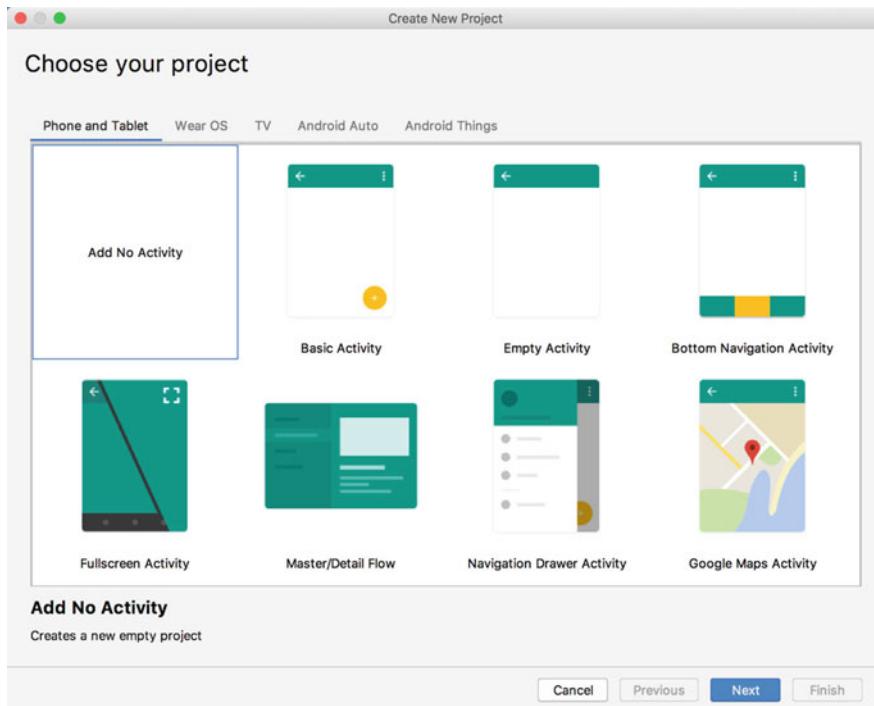
Activity is what attracts the user. It is a component that hosts the visual components and interacts with the user. An app can have 0 or multiple activities; however, it is very rare for an app to have 0 activity since nobody wants their app to be invisible to the user.

In fact, you've already used Activity in Chap. 1 and learned some basics of activity. But at that time, our focus was to build your first Android project and we didn't cover activity in detail. We will learn more details about Activity in this chapter.

### 3.2 Activity Fundamentals

In Chap. 1, Android Studio auto-generated MainActivity for us and in order to get a deeper understanding of the activity, let's manually create an activity.

In Android Studio, you can keep only one project open; thus, you need to close the current project by clicking navigation bar File->Close Project. Then let's follow the same steps in Chap. 1 to create a new Android project, except for the step in



**Fig. 3.1** Select “Add No Activity”

Fig. 1.8. We don’t select the “Empty Activity” but instead, select “Add No Activity” this time as shown in Fig. 3.1.

Click “Next” to go to the project configuration screen.

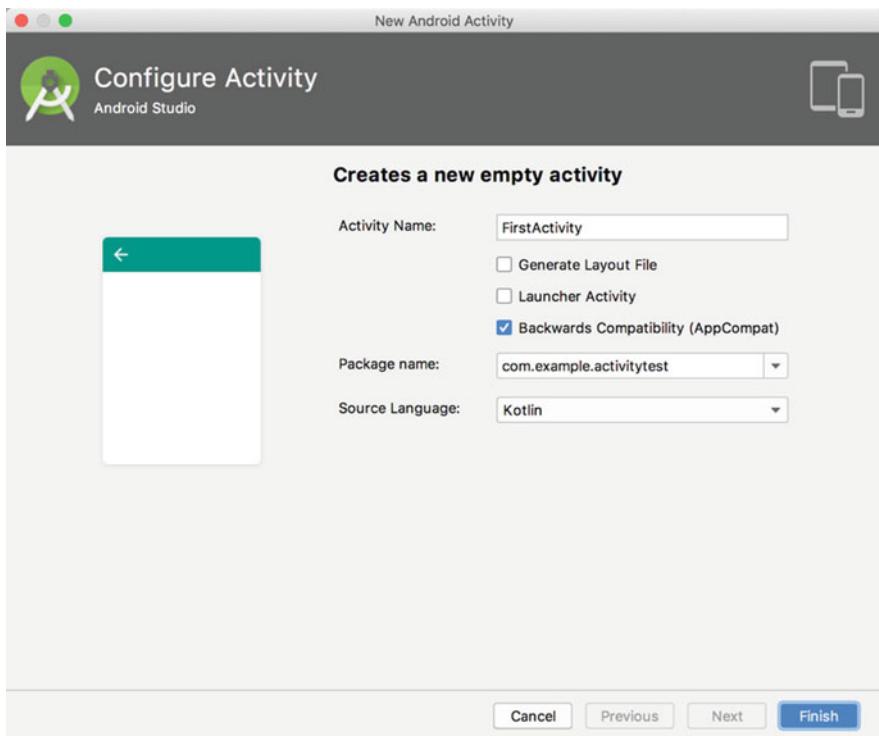
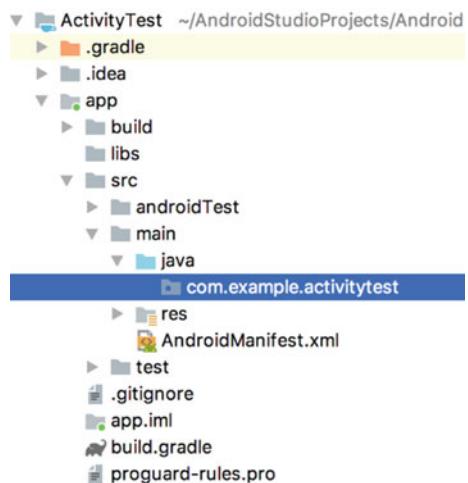
Let’s name the project as ActivityTest and use the default package name com.example.activitytest. We can keep other configurations the same as the project created in Chap. 1. Click “Finish,” and after Gradle finishes running, the project gets created successfully.

### 3.2.1 Manually Creating Activity

After successfully creating the project, Android Studio will use the default Android project structure and let’s change it to Project mode and we will do the same thing for later projects in this book and won’t highlight this anymore. Now there are a lot of generated files in ActivityTest project, but app/src/main/java/com.example.activitytest directory should be empty as shown in Fig. 3.2.

Now, right click com.example.activitytest package->New->Activity->Empty Activity, there will be a dialog to create the Activity, let’s name it as FirstActivity

**Fig. 3.2** Initial project structure



**Fig. 3.3** Dialog to create Activity

and don't select Generate Layout File nor the Launcher Activity options, as shown in Fig. 3.3.

By selecting Generate Layout File option, a layout file will be created for FirstActivity and by selecting Launcher Activity option, FirstActivity will be set

as the main activity of the current project. Since we want to learn what happens under the hood, don't select these options and let's add them manually. Select Backwards Compatibility to make sure the project is compatible for older versions of OS, then click "Finish."

In any Activity, we need to override the `onCreate()` function, and Android Studio's already done it for us for `FirstActivity` as shown below:

```
class FirstActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

}
```

As you can see, `onCreate()` method here is simply calling the parent class's `onCreate()` method. Of course, this is default implementation, and we can add our code later.

### 3.2.2 Creating and Mounting the Layout

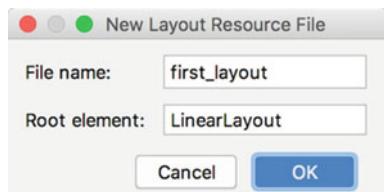
We mentioned in previous chapter that we need to separate the business logic from the views in Android programs, and each Activity should have a corresponding layout that will be used to show the views, now let's create a layout file.

Right click `app/src/main/res` dir ->New->Directory, there will be dialog to create a new directory. Here let's create a directory with name `layout` and then right click it ->New->Layout resource file which will bring up a dialog to create new layout resource file, and let's name the new file as `first_layout`, Select `LinearLayout` as the Root element as shown in Fig. 3.4.

Click OK to finishing layout creation and you will see a layout editor as shown in Fig. 3.5.

In this visual layout editor, you can preview the current layout. From the bottom left of the window, you can switch between Design and Text. Design is the visual layout editor where you can preview the layout and edit the layout by drag and drop. And Text mode will allow you to edit the layout by editing the XML file. By selecting Text, you should see the following code:

**Fig. 3.4** Create new layout resource file



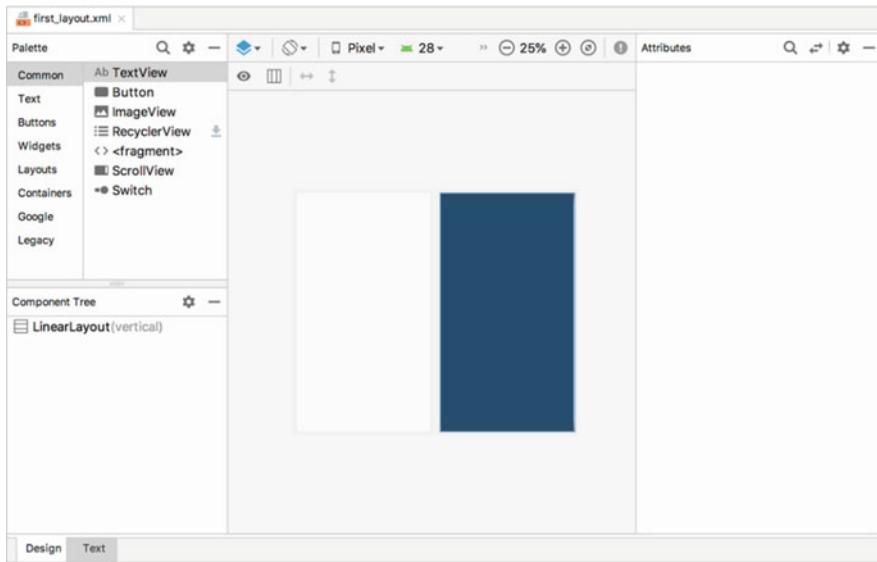


Fig. 3.5 Layout Editor

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

</LinearLayout>
```

This is because we selected `LinearLayout` as the root element. Now let's add a button to this layout as shown below:

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

<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button 1"
/>

</LinearLayout>
```

Here we added a Button element, and then set a few attributes inside the button. Android:id is used to set the unique ID for the current element. If you feel strange for the plus sign in @ + id/button1, then it is because we're defining an ID. If we need to reference an ID in XML then we can use @id/id\_name. Next, android:layout\_width specifies the width and here we use match\_parent to make the width of the current element the same as the parent element(container). We should use android:layout\_height to specify the height of the current element and here we use wrap\_content to make the height of current element enough to show the content. Android:text is used to specify the text content of the element. We will cover more details about creating layout in next chapter. Now you can use the Preview tool to preview the visual effect of the current layout as shown in Fig. 3.6.

Now as the button can be successfully shown in the preview, we have finished a simple layout. In the next step let's load this layout in the Activity.

Let's go back to FirstActivity and add the following code:

```
class FirstActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.first_layout)
    }
}
```

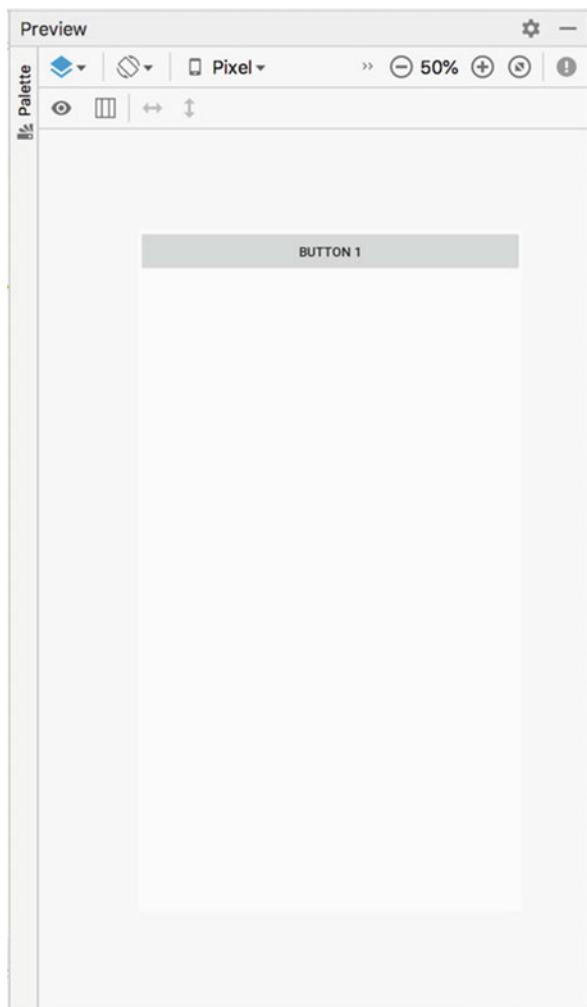
We are using setContentView() method to load the layout for the Activity by passing the ID of the layout. In Chap. 1, I mentioned that by adding any resource in the project will generate a resource ID in the R file; thus the ID of our newly created layout file first\_layout.xml has been added to the R file. And you've already learned how to use the layout file which is simply using R.layout.first\_layout to get the ID of the first\_layout.xml file and then pass the value to setContentView() method.

### 3.2.3 Registering in AndroidManifest File

In Chap. 1, we learned that activity need to register in AndroidManifest.xml file so that it can be used. FirstActivity has already been registered in AndroidManifest.xml file and we can open app/src/main/AndroidManifest.xml to verify, code is as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.activitytest">
```

**Fig. 3.6** Preview of current layout



```
<application  
    android:allowBackup="true"  
    android:icon="@mipmap/ic_launcher"  
    android:label="@string/app_name"  
    android:roundIcon="@mipmap/ic_launcher_round"  
    android:supportsRtl="true"  
    android:theme="@style/AppTheme">  
    <activity android:name=".FirstActivity">  
        </activity>  
    </application>  
  
</manifest>
```

We can see that registration of Activity happens within application element and we use the element <activity> to register for the Activity. Apparently here Android Studio auto registered the activity for us. This is because a lot of people would forget to register Activity and other system components which caused lots of crashes in the past. Android Studio will do the job automatically for us now.

Within <activity> element, we used android:name to set which activity to register and. FirstActivity is short for com.exmaple.activitytest.FirstActivity. We can do so because in the outermost <manifest> element, we already set the package name for the project to be com.exmaple.activitytest. We can omit this part and only use. FirstActivity.

However, after registering the Activity, we still cannot run the program as we haven't configured the main Activity yet. The main Activity is the first Activity to start when the app starts to run. We've already covered how to configure the main Activity in Chap. 1 which is by adding <intent-filter> element within the <activity> then add <action android:name="android.intent.action.MAIN" /> and <category android:name="android.intent.category.LAUNCHER" /> .

We can use android:label to specify the content of the navigation bar in the Activity which is shown at the top of the app. Notice that the text used for the label of the main Activity will not only be the content of the navigation bar but also the name of the app in the Launcher.

The modified AndroidManifest.xml file is as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.activitytest">

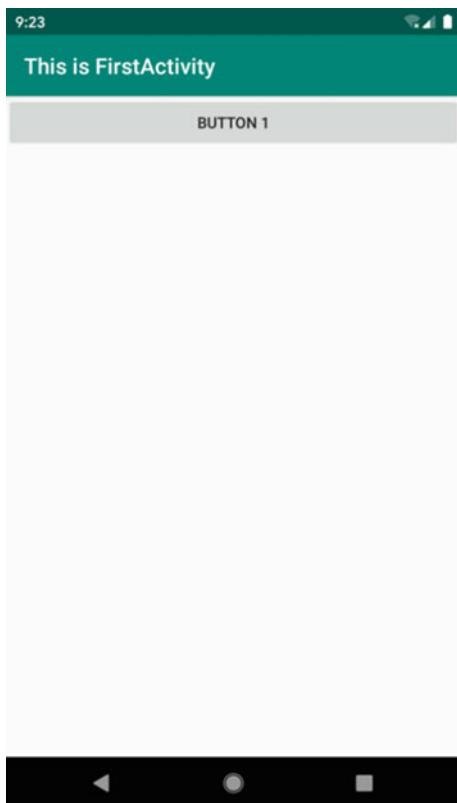
    <application
        ...
        <activity android:name=".FirstActivity"
            android:label="This is FirstActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

With this code change, FirstActivity becomes the main Activity of our app, and by clicking the icon of the app, this Activity will be started first. Notice that, if you don't specify the main Activity in the app, then the app can still be installed, though you cannot see it in the launcher or open this app. This kind of app usually is used as third-party service to be utilized by other apps.

Now let's run the app and the result is as Fig. 3.7.

**Fig. 3.7** Running the app for the first time



At the top is the nav bar which has the content when we register the Activity. Under the nav bar is the UI backed by `first_layout.xml` which has a button we just defined. Now as you learned how to manually create the Activity, let's see what you can do inside the Activity.

### **3.2.4 Using Toast in Activity**

Toast is a widget to notify the user with some short messages and will disappear in a short time and doesn't take a lot of space. Now let's learn how use Toast inside the Activity.

First, we need to decide how to trigger the Toast and since we have a button in the screen, let's use the button click event to trigger showing Toast. Add the following code inside `onCreate()` method:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.first_layout)
    val button1: Button = findViewById(R.id.button1)
    button1.setOnClickListener {
        Toast.makeText(this, "You clicked Button 1", Toast.LENGTH_SHORT).
show()
    }
}

```

In activity, we can use `findViewById()` method to get the element defined in the layout file and here let's pass `R.id.button1` which is specified in `first_layout.xml` file with `android:id` attribute. `findViewById()` method will return template object that inherited from `View`. Thus, Kotlin cannot infer whether it is a `Button` or some other widget. Therefore, we need to explicitly declare `button1` variable as `Button`. After acquiring the instance of the button, we will use `setOnClickListener()` method to register a listener to the button. Clicking the button will trigger the `onClick()` method of the listener. Then we need to write the code inside `onClick()` method to trigger the `Toast`.

We can use the static method `makeText()` to create a `Toast` instance, and then use `show()` method to display the `Toast`. Notice that `makeText()` method takes 3 parameters and the first parameter is `Context` and since `Activity` itself is `Context`, we can pass in this. The second parameter is the text content shown in the toast. The last parameter is the time that determines how long the `Toast` should display which has two internal constant values we can select from: `Toast.LENGTH_SHORT` and `Toast.LENGTH_LONG`.

Rerun the app and clicking the button will show something as shown in Fig. 3.8.

I'd like to cover more about the `findViewById()` method. As we know, `findViewById()` is used to get the instance of the element in the layout file, and the example here simply has one button. What if there are 10 widgets in the layout that we need to get instance? Do we have to call `findViewById()` 10 times? The answer is, yes, we can! But it is pretty clumsy and we have third-party libs like `ButterKnife` to simplify using `findViewById()`.

However, this is actually not an issue for Kotlin as the Android project written in Kotlin will auto import the `kotlin-android-extension` plugin in the `app/build.gradle` file. This plugin will generate a variable that has the same name of the element in the layout file. Then we can use this variable directly in the `Activity` instead of using `findViewById()` method, as shown in Fig. 3.9.

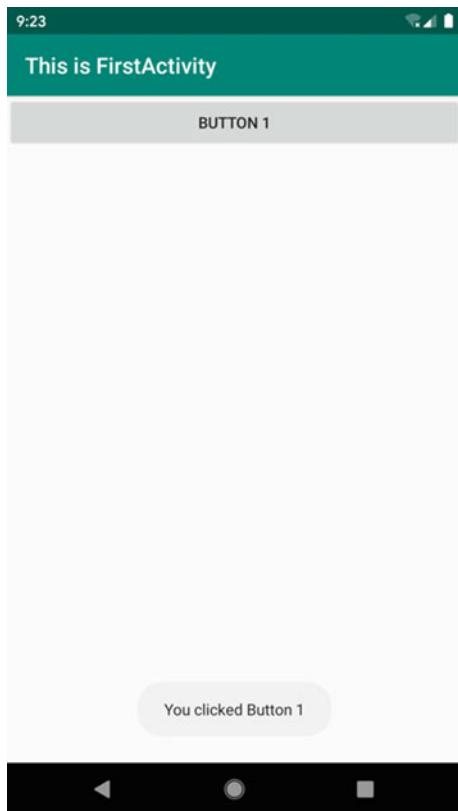
I still recommend you use the autocompletion feature provided by `Android Studio` as the auto-generated variables also need to import related package.

Now we can simplify the code to the following:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.first_layout)
    button1.setOnClickListener {
        Toast.makeText(this, "You clicked Button 1", Toast.LENGTH_SHORT).

```

**Fig. 3.8** Show Toast

```
class FirstActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.first_layout)  
        button1  
        button1 from first_layout.xml for Activity (Android Ex... Button💡  
        Press ^ to choose the selected (or first) suggestion and insert a dot afterwards >>  
    }  
}
```

The screenshot shows the Android Studio code editor with the following code:  

```
class FirstActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.first_layout)  
        button1  
        button1 from first_layout.xml for Activity (Android Ex... Button💡  
        Press ^ to choose the selected (or first) suggestion and insert a dot afterwards >>  
    }  
}
```

A tooltip appears over the variable "button1", indicating it is a "Button" from the XML layout file. It also suggests pressing '^' to choose the selected suggestion and insert a dot afterwards.**Fig. 3.9** Use the auto-generated button1 variable

```
show()  
}  
}
```

As you can see, we don't need to use the `findViewById()` method anymore. This is the recommended pattern for programming in Kotlin with some exceptions. In this book we won't use `findViewById()` method any more. I covered the usage of

findViewById() method here although you rarely need to use it simply because under the hood, kotlin-android-extension plugin used findViewById() method to find the instance.

### 3.2.5 Using Menu in Activity

Cellphones have very limited screen space compared with laptops or desktops. Thus, it is very important to make best use of the screen space for our apps. If you need to show many menu items, it will look very crowded since just the menu items will likely occupy more than a third of the screen space. What should we do to solve this problem? No worries, Android provides a way to show the menu items while taking minimum amount of space.

First create a menu folder under the res directory. Right click res directory->New ->Directory, “menu” for folder name, click “OK.” Then create a file in this folder with name “main,” right click the menu folder->New->Menu resource file as shown in Fig. 3.10.

Type “main” for the file name and click “OK” to finish creation, then add code below to main.xml.

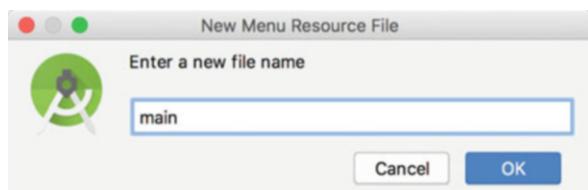
```
<item  
    android:id="@+id/add_item"  
    android:title="Add"/>  
<item  
    android:id="@+id/remove_item"  
    android:title="Remove"/>  
</menu>
```

Here we create two menu items, <item> element is used to create a menu item and then we can use android:id to set a unique identifier to the menu item, and we can use android:title to give the menu item a label.

Let's go back to the FirstActivity file to override the onCreateOptionsMenu() method, to do so we can use shortcuts to help us. Use Ctrl+O in Windows and use control + O in Mac OS, as shown in Fig. 3.11.

Then write the following code in onCreateOptionsMenu() method:

**Fig. 3.10** Create new Menu res folder



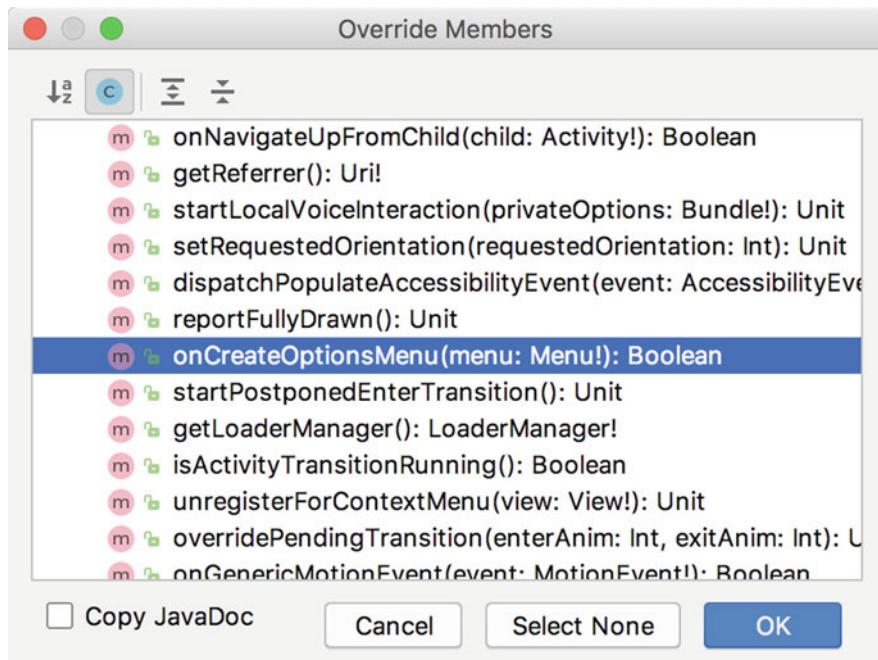


Fig. 3.11 Override onCreateOptionsMenu() method

```
override fun onCreateOptionsMenu(menu: Menu?) : Boolean {
    menuInflater.inflate(R.menu.main, menu)
    return true
}
```

Before explaining the code, I'd like to introduce another Kotlin syntax sugar. If you're familiar with Java, then you should know the concept of Java Bean which is a very simple class that will generate the Getter and Setter method for the fields in the class as shown below:

```
public class Book {

    private int pages;

    public int getPages() {
        return pages;
    }

    public void setPages(int pages) {
        this.pages = pages;
    }
}
```

In Kotlin, it can be more succinct, and we can use the following code to read and write the pages field in Book class:

```
val book = Book()
book.pages = 500
val bookPages = book.pages
```

Here, it looks like that we did not call the setPages() nor getPages() method in the Book class, but instead, directly read and write the pages field. This is actually the syntax sugar Kotlin provided which will auto convert the code above to call setPages() and getPages() method behind the scene.

The menuInflater in onCreateOptionsMenu() uses this syntax sugar which actually calls the getMenuInflater() method in the parent class. getMenuInflater() method can get an instance of MenuInflater and then by calling its inflate() method we can create a menu for the activity. Inflate() method takes two parameters: the first parameter is used to specify which resource file to use to create the menu, and here we pass in R.menu.main; The second parameter is used to specify which menu the menu item will be added to and we will use the menu argument passed from onCreateOptionsMenu. At last, return true for this method to allow the display of the created menu, if return false then menu won't be shown.

Of course, just being able to show the menu is not enough, we need to allow the menu to listen and handle the click event. Let's override the onOptionsItemSelected() method in FirstActivity as shown below:

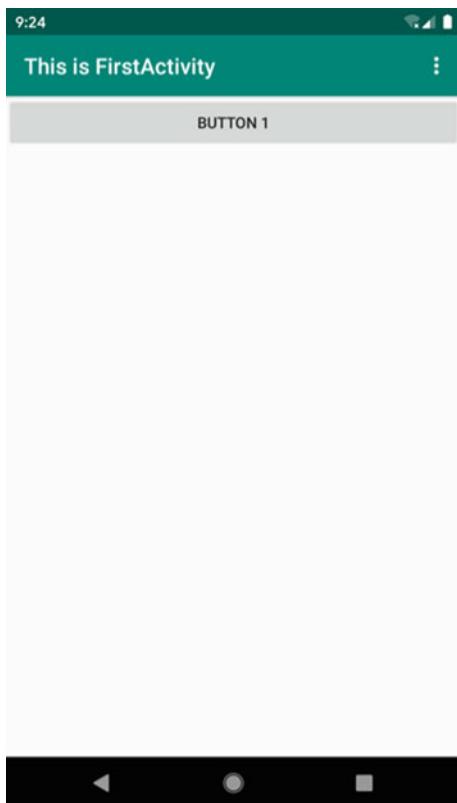
```
override fun onOptionsItemSelected(item: MenuItem) : Boolean {
    when (item.itemId) {
        R.id.add_item -> Toast.makeText(this, "You clicked Add",
            Toast.LENGTH_SHORT).show()
        R.id.remove_item -> Toast.makeText(this, "You clicked Remove",
            Toast.LENGTH_SHORT).show()
    }
    return true
}
```

In onOptionsItemSelected() method, we use item.itemId to determine which menu item is getting clicked which also applies the syntax sugar we just learned. Then we can pass it to when statement and add the logic to handle each menu item. Here we can practice the knowledge we just learned about Toast and let's show the corresponding Toast message.

Rerun the app and you can find a vertical ellipsis at the right side of the nav bar, which is the menu button as shown in Fig. 3.12.

As you can see, the menu items will be hidden by default and only by clicking the menu button will the menu items be shown. Thus, it won't take the space of the Activity when not needed as shown in Fig. 3.13.

**Fig. 3.12** Activity with Menu



If you click Add menu item, you will see the Toast with message “You clicked Add” as shown in Fig. 3.14. If you clicked Remove item, then Toast with message “You clicked Remove” will show up.

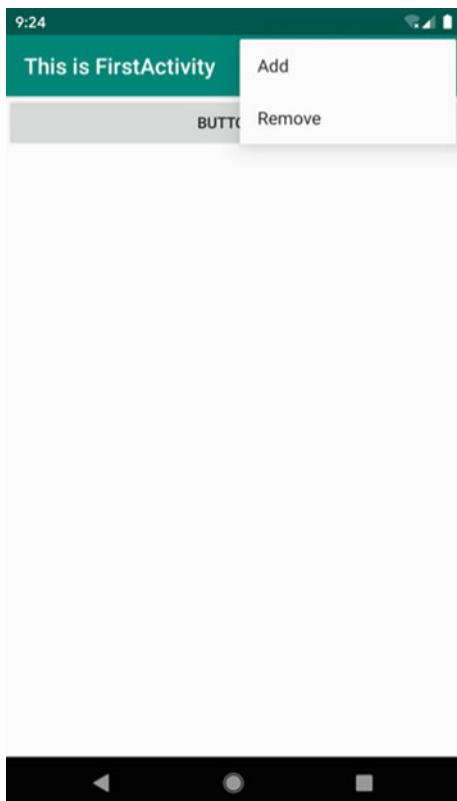
### 3.2.6 *Destroying an Activity*

You should know how to create an activity and create menu and toast from last section. Maybe you will ask the question: how to destroy an activity?

The answer is quite simple, clicking the Back button will destroy the current activity. However, if you don't want to use the back button but want to write code to destroy the activity then activity class provides finish() method to destroy the activity.

Let's update the listener's code of the menu item as follows:

**Fig. 3.13** Activity with menu items being shown



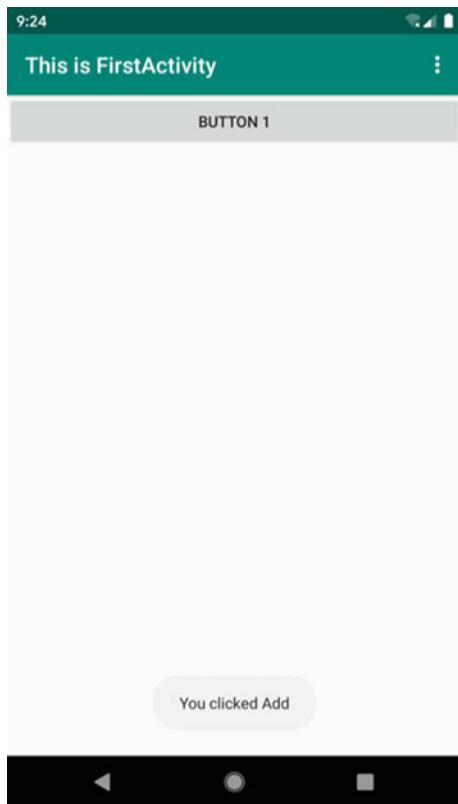
```
button1.setOnClickListener {  
    finish()  
}
```

Rerun the app and click the button, the current Activity will get destroyed which is the same as pressing the Back button.

### 3.3 Using Intent to Communicate Between Activities

It is easy to create a single activity app, but you can create more than one activity for your app to enrich the experience. The question will be how to open another activity since, when you launch the app, only the main activity will be opened.

**Fig. 3.14** Clicked Add menu item



### 3.3.1 Explicit Intent

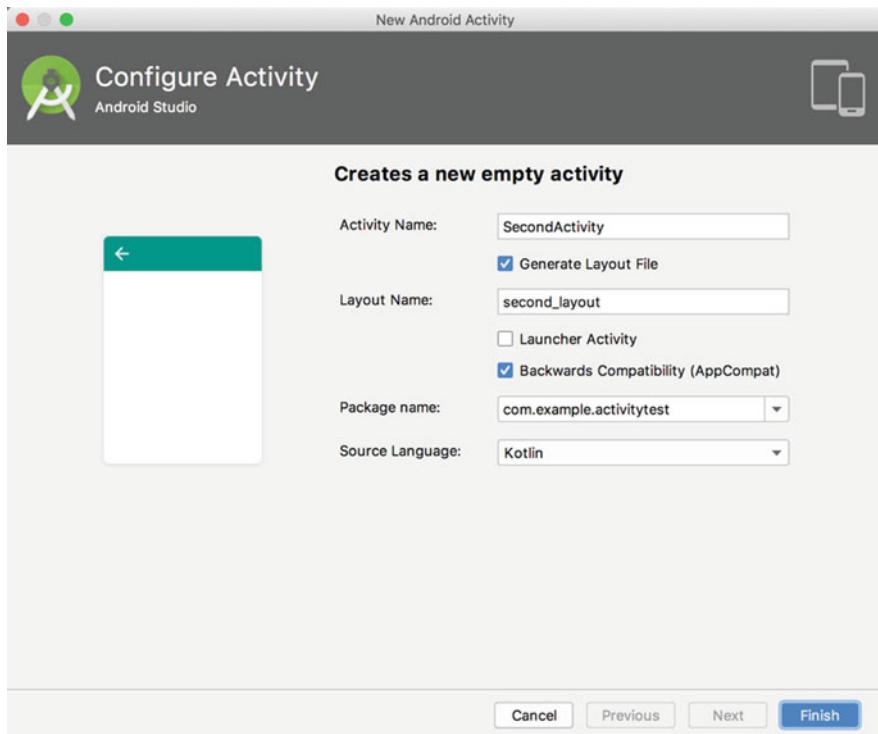
Now let's create a second Activity with the same process we learned and name it as SecondActivity and select "Generate Layout File" and then give the layout name second\_layout without selecting Launcher Activity as shown in Fig. 3.15.

Click "Finish," then Android Studio will auto-generate SecondActivity.kt and second\_layout.xml. The auto-generated layout file may be confusing for you. So let's replace the generated code with the code below:

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

    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add" />

```



**Fig. 3.15** Create SecondActivity

```
    android:text="Button 2"
/>
</LinearLayout>
```

We define a button with “Button2” text.

SecondActivity already has some auto-generated code, and we can keep it as it is like shown below:

```
class SecondActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.second_layout)
    }
}
```

Android Studio already helped us to register the Activity and AndroidManifest.xml file will look like code below:

```
<application
    ...
    <activity android:name=".SecondActivity">
    </activity>
    <activity
        android:name=".FirstActivity"
        android:label="This is FirstActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Since SecondActivity is not the main Activity, there is no need to configure the contents for `<intent-filter>` which makes the registration process much easier. Now since the second Activity has been created, the question of how to start it remains. We need to introduce a new concept: intent to solve the problem.

Intent is an important component in Android apps to communicate with each other. It can specify the actions of the current component and pass data between the different components. Intent is widely used for starting Activity, starting Service and Broadcast. We will cover the latter two use cases in the upcoming chapters and will focus on starting Activity for now.

There are mainly two kinds of Intent: explicit Intent and implicit Intent. Let's first look at the explicit Intent.

There are multiple constructor overrides for Intent and one of them is `Intent(Context packageContext, Class<?> cls)`. The first parameter of this function provides the context to start the Activity and the second parameter Class is used to specify the target Activity that will be started. How do we use this Intent? Activity class provides `startActivity()` method to start Activity which takes an Intent parameter. Here we can pass the constructed Intent to the `startActivity()` method to start the target Activity.

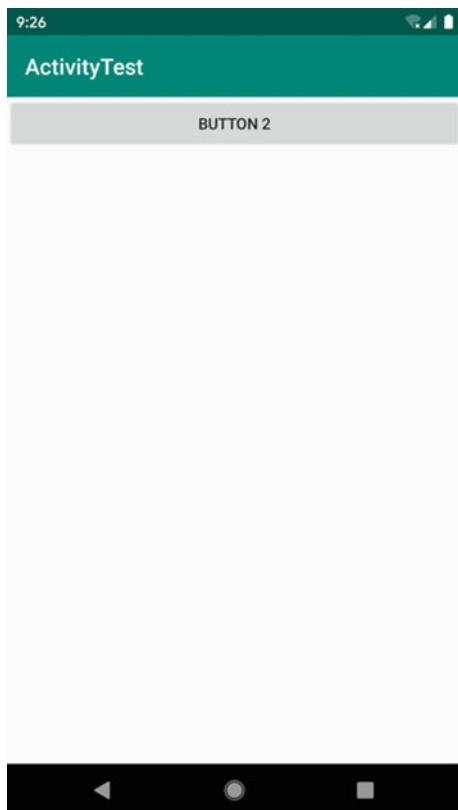
Modify the click event handling code in FirstActivity as shown below:

```
button1.setOnClickListener {
    val intent = Intent(this, SecondActivity::class.java)
    startActivity(intent)
}
```

In the code above, we first create an instance of Intent, pass this FirstActivity instance as context, pass `SecondActivity::class.java` as the target Activity and this makes our “intent” very clear: startingSecondActivity from FirstActivity. Notice that `SecondActivity::class.java` is the same as `SecondActivity.class` in Java. Next, we can use `startActivity()` to run the Intent.

Rerun the app and click the button in FirstActivity and the result is as shown in Fig. 3.16.

**Fig. 3.16** UI of SecondActivity



So now we have successfully started SecondActivity, what if we want to go back to the last Activity? By simply pressing the Back button we can destroy the current Activity and go back to the last Activity.

The approach above to start an Activity is explicit, and thus we call the intent here as explicit Intent.

### 3.3.2 *Implicit Intent*

Compared with the explicit Intent, the implicit Intent doesn't specify which Activity it is going to start, but instead, specifies a series of abstract information such as action, category, etc. Then the system will analyze the Intent and find the correct Activity to start.

So which activity is the correct Activity? Simply put, it is the Activity that can respond to this implicit Intent. Then what implicit Intent can SecondActivity respond? None, for now, but it will be able to shortly.

By configuring the attributes of `<intent-filter>` within `<activity>`, we can specify the action and category that the current Activity can respond. Open `AndroidManifest.xml` and add the following code:

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

In `<action>` element, we set that `SecondActivity` can respond to the action of `com.example.activitytest.ACTION_START`, and `<category>` element contains some extra information and only when an Intent's action and category matches with the content specified here can `SecondActivity` respond to this Intent.

Modify the click event handling code in `FirstActivity` to the following:

```
button1.setOnClickListener {
    val intent = Intent("com.example.activitytest.ACTION_START")
    startActivity(intent)
}
```

Here we use another constructor of Intent and pass in the action string which means that we want to start the Activity that can respond to `com.example.activitytest.ACTION_START` action. We just mentioned that action and category should match at the same time so that Activity can respond and handle the intent. Why we don't need to do so here? This is because `android.intent.category.DEFAULT` is a default category and when `startActivity()` is called, this category will be added to Intent instance directly.

Rerun the app, click the button in `FirstActivity`, and you should be able to start `SecondActivity` with implicit Intent which means that our configuration in `AndroidManifest.xml` works.

Each Intent instance can only specify one action but can contain multiple categories. We only have one category for now and let's add more to it.

Modify the button click event code as follows:

```
button1.setOnClickListener {
    val intent = Intent("com.example.activitytest.ACTION_START")
    intent.addCategory("com.example.activitytest.MY_CATEGORY")
    startActivity(intent)
}
```

We can use the `addCategory()` method in Intent to add category and here let's define a customized category—`com.example.activitytest.MY_CATEGORY`.

Now rerun the app and click the button in `FirstActivity`, you will find that app will crash! If this is the first time you're debugging crash, don't worry. Let's take a look

```
Process: com.example.activitytest, PID: 24027
android.content.ActivityNotFoundException: No Activity found to handle Intent {
act=com.example.activitytest.ACTION_START cat=[com.example.activitytest.MY_CATEGORY] }
```

**Fig. 3.17** Exception information

at the Logcat to see what happens, and you should find the exception information shown in Fig. 3.17.

The exception tells us that no Activity can respond to our Intent. This is because we just added a category in the Intent while the `<intent-filter>` element of SecondActivity didn't get configured to respond to this category. Thus there is no Activity that can respond to this Intent. Let's add it now as follows:

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="com.example.activitytest.MY_CATEGORY" />
    </intent-filter>
</activity>
```

Rerun the app and follow the same process above, and you will find that everything will work as expected now.

### 3.3.3 More on Implicit Intent

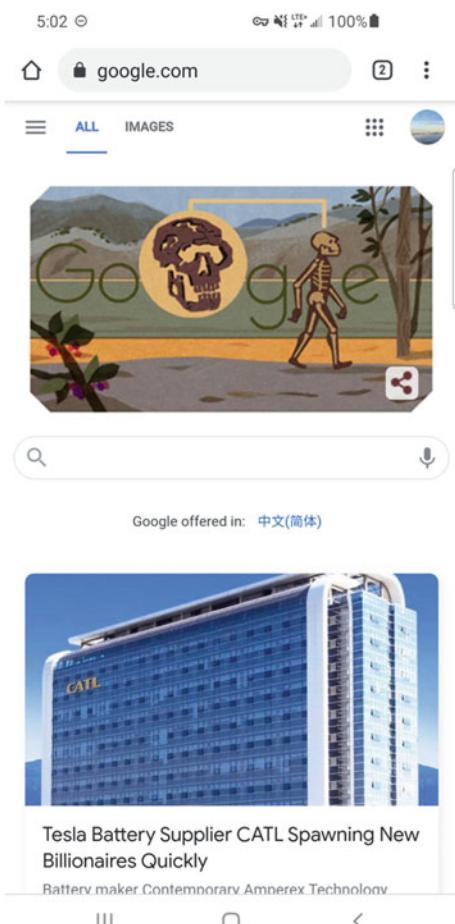
From the last section, you should know how to start Activity by using implicit Intent and there are actually more usages around implicit Intent and we will use this section to cover them.

By using the implicit Intent, we can start the Activity not only inside our app but also the activity in other apps which makes inter-app communication and sharing possible. For instance, if you want to display a web page in your app, then you don't need to implement a browser but instead using the system browser to open it.

Modify the click event handling code in FirstActivity as follows:

```
button1.setOnClickListener {
    val intent = Intent(Intent.ACTION_VIEW)
    intent.data = Uri.parse("https://www.google.com")
    startActivity(intent)
}
```

Here we first specify the action of the Intent to be Intent.ACTION\_VIEW which is an Android system action and the value of it is android.intent.action.VIEW. We can use Uri.parse() method to parse the string and build a Uri object. Then use

**Fig. 3.18** System browser

setData to pass in the Uri object and we use again the syntax sugar mentioned earlier which looks like we're assigning the value directly to the field of Intent.

Rerun the app and clicking the button in FirstActivity will open the system browser as shown in Fig. 3.18.

We can add <data> element within the <intent-filter> which can specify the data that the Activity can respond to more precisely. In the <data> element, you can configure the following items:

- Android:scheme – used to specify the scheme, for example https.
- Android:host – used to specify host, for example [www.google.com](http://www.google.com)
- Android:port – used to specify port usually immediately after the host.
- Android:path – used to specify the content after port.
- Android:mimeType - used to specify the data type the Activity can handle.

Only when the data in the Intent is the same as specified in <data> can an Activity respond to the Intent. However, usually the <data> element won't specify all the details and using the example above, if we specify android:scheme to be https, then the Activity will respond to all the Intent that use https protocol.

To understand this better, let's create an activity that can respond to the intent that will open the web page. Right click com.example.activitytest package->New->Activity->Empty Activity, create ThirdActivity and select the "Generate Layout File" option and name the layout file to be third\_layout, click "Finish" to finish creation. Next, edit the third\_layout.xaml by replacing the code inside to the following:

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

    <Button
        android:id="@+id/button3"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 3"
    />

</LinearLayout>
```

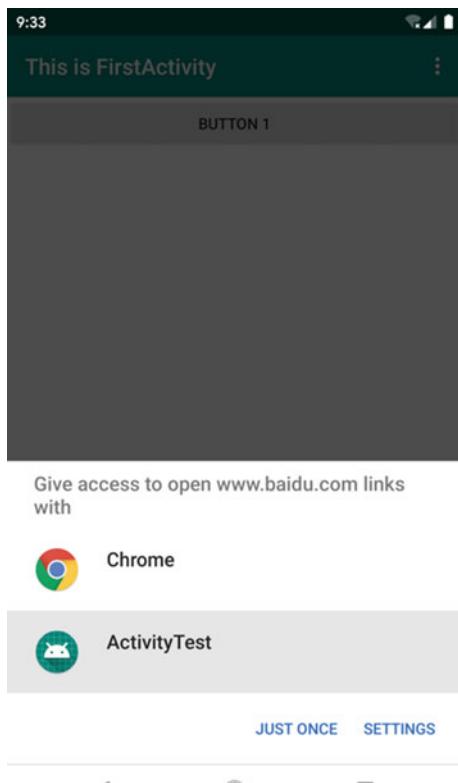
We can keep the code of ThirdActivity as it is and modify the registration information for ThirdActivity in AndroidManifest.xml as follows:

```
<activity android:name=".ThirdActivity">
    <intent-filter tools:ignore="AppLinkUrlError">
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="https" />
    </intent-filter>
</activity>
```

We set the action that ThirdActivity can handle to be Intent.ACTION\_VIEW and use the default category and set the scheme to be https. Now ThirdActivity will be able to respond to Intent that is going to open web page just like a browser. We add tools:ignore attribute to ignore the warning since Android Studio concludes that all Activity that can respond to ACTION\_VIEW should be a category of BROWSABLE which is used for deep link. However, this is not related to what we're doing; thus let's just ignore the warning.

Now, let's run the app again and click the button in FirstActivity, the result is as shown in Fig. 3.19.

**Fig. 3.19** Select the app that can respond to the Intent



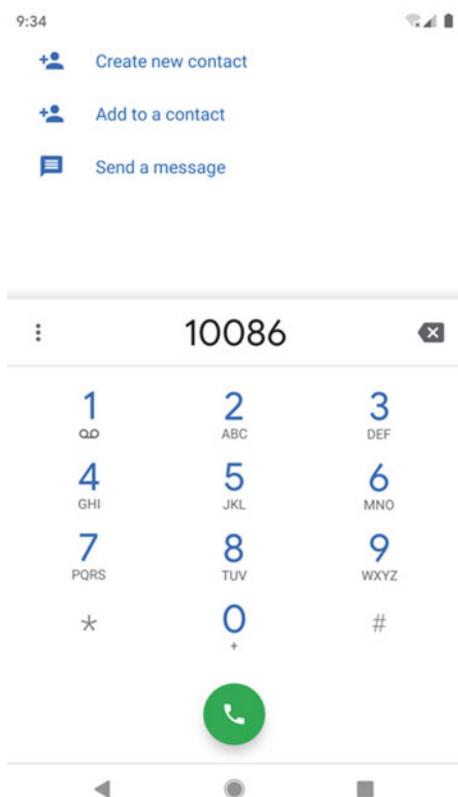
A system bottom sheet will pop up to display all the apps that can respond to the intent. If we select ActivityTest then ThirdActivity will be started. Though we made ThirdActivity to respond the intent of opening the web page, we actually didn't add functionality to do so. In real world, never do something like this as this will confuse the user.

Besides the https protocol, we have other options too; for example, using geo to display geo location and tel to make phone calls. Use the code below to bring up the system phone call app:

```
button1.setOnClickListener {  
    val intent = Intent(Intent.ACTION_DIAL)  
    intent.data = Uri.parse("tel:10086")  
    startActivity(intent)  
}
```

The code above first specifies that the action of the intent is Intent.ACTION\_DIAL which is another Android system action. Then in the data, we

**Fig. 3.20** System dialing UI



specify the scheme to be tel and phone number to be 10,086. Let's rerun the app and click the button in FirstActivity, and the result will be shown as Fig. 3.20.

### 3.3.4 Passing Data to the Next Activity

Now we have learned how to use Intent to start an Activity and let's see how to use Intent to pass data to the next Activity.

To pass data is actually fairly easy. Intent provides a series of functions that override `putExtra()` method and we can put our data in Intent and then when another Activity gets started, we can get the data from the intent. For example, if there is a string in FirstActivity and we want to pass the string to SecondActivity, then you can write the code as follows:

```
button1.setOnClickListener {
    val data = "Hello SecondActivity"
    val intent = Intent(this, SecondActivity::class.java)
```



Fig. 3.21 Print data info in SecondActivity

```
    intent.putExtra("extra_data", data)
    startActivity(intent)
}
```

Here we still use the explicit Intent to start SecondActivity and used putExtra() method to pass in a string. Notice that there are two parameters for putExtra() method: the first param is a key which will be used to get data from intent later, and the second param is the data we want to pass.

Then we can use the following code to get the data from Intent and print in the console:

```
class SecondActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.second_layout)
        val extraData = intent.getStringExtra("extra_data")
        Log.d("SecondActivity", "extra data is $extraData")
    }
}
```

We're using the syntax sugar again as the intent in the code above actually will call the getIntent() method in AppCompatActivity. Since we're passing string here, we should use getStringExtra() to get the string data. We can use getIntExtra() for getting integer, getBooleanExtra for Boolean so on so forth.

Rerun the app and click the button in FirstActivity will open SecondActivity and Logcat should print the information as shown in Fig. 3.21.

We can see that SecondActivity successfully get the data passed from FirstActivity.

### 3.3.5 *Return Data to the Last Activity*

As we can pass the data to the next Activity, a question will raise naturally: can we return data to the last Activity? The answer is yes. However, to return to the last Activity, we need to press the Back button and we don't explicitly start the last Activity here. Then how can we use Intent here? So, Activity actually has startActivityForResult() method to start an Activity when we expect to get data from the new Activity.

`startActivityForResult()` will take two parameters: the first param is Intent; the second param is the requestCode which will be used to determine the source of the data. Let's modify the button click event handling code in `FirstActivity` as follows:

```
button1.setOnClickListener {
    val intent = Intent(this, SecondActivity::class.java)
    startActivityForResult(intent, 1)
}
```

Here, we use `startActivityForResult()` to start `SecondActivity`, and requestCode should be a unique value. We just assign 1 to it. Next let's add the event handling code in `SecondActivity` and return the data after clicking the button as shown below:

```
class SecondActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.second_layout)
        button2.setOnClickListener {
            val intent = Intent()
            intent.putExtra("data_return", "Hello FirstActivity")
            setResult(RESULT_OK, intent)
            finish()
        }
    }
}
```

In the code above, we create an intent only for passing the data without action or target activity. Then we use `setResult()` method to return the data. `setResult()` method takes two parameters: the first param is used to return the status of the result like `RESULT_OK` or `RESULT_CANCELED`; the second param is the intent that has the data need to be returned. Then we call `finish()` method to destroy the current Activity.

Since we use `startActivityForResult()` to start `SecondActivity`, after `SecondActivity` gets destroyed, `onActivityResult()` method in `FirstActivity` will be called. Thus we need to override this method to get the data from `SecondActivity` as shown below:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    when (requestCode) {
        1 -> if (resultCode == RESULT_OK) {
            val returnedData = data?.getStringExtra("data_return")
            Log.d("FirstActivity", "returned data is $returnedData")
        }
    }
}
```



Fig. 3.22 Print data coming from SecondActivity

onActivityResult() has 3 parameters: requestCode should be the number used in FirstActivity, resultCode, and data which is the intent. The same activity can start multiple activities, and thus we have to use the requestCode to determine which activity returns the current result. We can use resultCode to determine if the result is success or not.

Rerun the app, click the button in FirstActivity to open SecondActivity and then click the button in SecondActivity should bring you back to FirstActivity screen, Logcat should show information as shown in Fig. 3.22.

We can see that the data from SecondActivity has been successfully returned to FirstActivity.

You might ask that what if user just press the Back button? Then we can just override the onBackPressed() as follows:

```
override fun onBackPressed() {  
    val intent = Intent()  
    intent.putExtra("data_return", "Hello FirstActivity")  
    setResult(RESULT_OK, intent)  
    finish()  
}
```

When the user presses the Back button, onBackPressed() method will be called and we will be able to return the data to FirstActivity again.

## 3.4 Activity Lifecycle

It's very important to understand the lifecycle of Activity for any Android developer. Only after you master the lifecycle of Activity can you create apps that are smooth to use.

### 3.4.1 Back Stack

Now you probably noticed that Activity can be stacked and every time we start a new activity, it will be on top of the old activity and pressing Back button will destroy the activity at the top and then the activity below the top one will show up.

Under the hood, Android uses task to manage the activity. A task is a combination of activities in the stack and this stack is called back stack. Stack is a LIFO data structure. By default, when we start a new activity, it will be pushed in stack and at the top of the stack. When we press Back button or call finish() method to destroy an activity, the activity at the top of the stack will pop out of the stack and the activity below will be at the top of the stack. The system will show the activity that is at the top of the stack.

Figure 3.23 shows how the back stack manages the stack.

### 3.4.2 Activity States

There are 4 different states an activity can have in the lifecycle.

#### 1. Running

When an activity is at the top of the back stack, Activity is in the running state and system is the least likely to recycle the activity that is running since this will cause very negative user experience.

#### 2. Paused

When an activity is no longer at the top of the stack but still visible, then activity is in the paused state. You might feel confused why this is possible. This is because not every activity will take up the whole screen. Some activity will show up as dialog, etc. which will just take a portion of the screen. The paused activity is still alive, and system will be less likely to recycle activities in this state

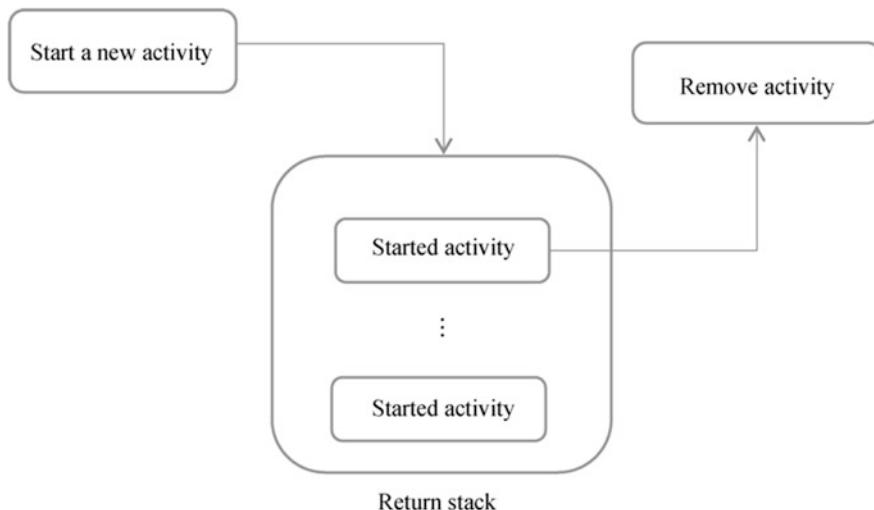


Fig. 3.23 Diagram of how back stack works

since it is still visible to the user. Only when the memory is extremely low will system try to recycle activities in this state.

### 3. Stopped

When an activity is not at the top of the stack and invisible to the user, then it is in the stopped state. System will still keep the state and variables of the activity; however, this is not reliable at all as when other apps need more memory, activities in this state will very likely be recycled by the system.

### 4. Destroyed

After an activity gets popped out of the stack, it will be in destroyed state and system will preferably recycle activities in this state to make sure other apps can have enough memory.

## ***3.4.3 Activity Lifecycle***

There are 7 callback methods in Activity which cover the whole lifecycle of activity, and let's look at these 7 callback methods.

- onCreate, you see this a few times now and you should have your initialization code here like loading the layout, binding, etc.
- onStart happens when activity is becoming visible.
- onResume() when the activity is ready to interact with the user and activity is at the top of the stack and running.
- onPause() when system is going to start or resume another activity and we usually release resources and save data in the callback, but we should make the logic inside fast enough to not affect the new Activity.
- onStop() called when the activity becomes invisible. If the new activity is a dialog style activity then onPause() will be called instead of onStop(),
- onDestroy() called before activity is destroyed, after this the activity will be in destroyed state.
- onRestart() called when activity is changing from stopped state to running state which means activity is getting restarted.

The 7 methods are symmetrical except the onRestart() method, and thus we can summarize the lifecycle into 3 different lifecycles.

- Full Lifecycle. Activity lifecycle between onCreate() and onDestroy(). Normally activity will initialize in onCreate() method and then release the memory in onDestroy() method.
- Visible Lifecycle. Between onStart() and onStop(), activity is visible to the user though may not always interactable with the user. We can manage the resources in these two methods, for example, load the resources in onStart() and release the resources in onStop() to ensure that the stopped activity won't take too much memory.
- Foreground Lifecycle. Activity between onResume() and onPause() is in running state which means that it can interact with the user.

In order to help you better understand, Android official website provides a diagram to show the lifecycle of Activity as shown in Fig. 3.24.

### 3.4.4 Explore the Lifecycle of Activity

Now let's apply what we have learned and build a project! We will create a new project instead of working on ActivityTest. Click File->Close Project, then create a new project called ActivityLifeCycleTest. We can use Android Studio's default configuration to help us create activity and layout to save some work. After that we still need to create two activities—NormalActivity and DialogActivity.

Right click com.example.activitylifecycletest package ->New->Activity->Empty Activity and create NormalActivity, name the layout file as normal\_layout. Then, follow the same process to create DialogActivity and name the layout as dialog\_layout.

Now, edit normal\_layout.xml file and replace the code as follows:

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This is a normal activity"
    />

</LinearLayout>
```

In this layout, we use a TextView to display a string and you will see more use cases for TextView.

Then edit dialog\_layout.xml and replace the code inside with following code:

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This is a dialog activity"
    />

</LinearLayout>
```

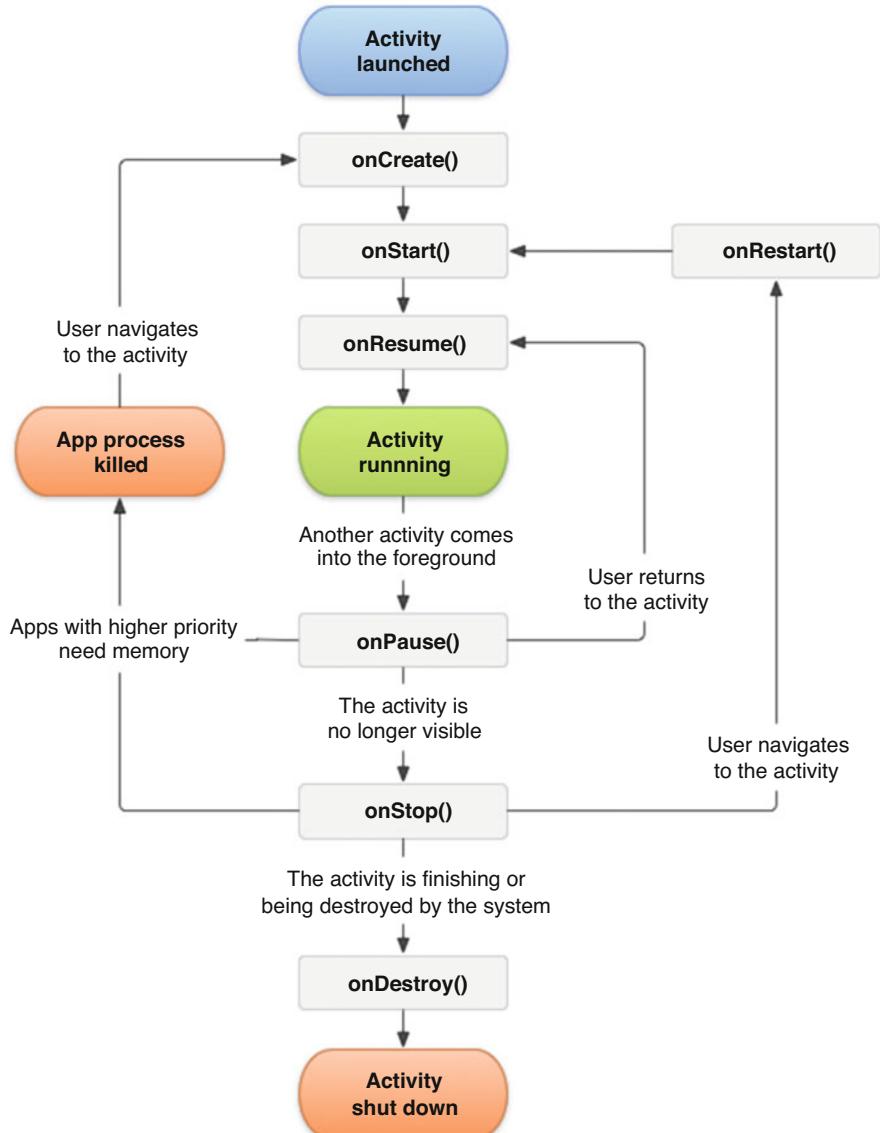


Fig. 3.24 Android Activity Lifecycle (source:clipped from official documentary page)

These two layouts are almost identical except for the text displayed in the TextView. We can keep the code in NormalActivity and DialogActivity as it is.

From the name, you can tell that the first activity is a normal activity while the second activity is a dialog style activity and they are the same for now. Let's make

the second activity a real dialog style activity. First, configure the AndroidManifest.xml file as follows:

```
<activity android:name=".DialogActivity"
    android:theme="@style/Theme.AppCompat.Dialog">
</activity>
<activity android:name=".NormalActivity">
</activity>
```

Notice that we add android:theme property to DialogActivity which will set the theme for the activity. There are many themes provided by the system and we can also build our own themes. Apparently @style/Theme.AppCompat.Dialog is meant to allow the DialogActivity use the dialog theme.

Now let's edit activity\_main.xml, and customize the layout of main activity by replacing the code to the following:

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

    <Button
        android:id="@+id/startNormalActivity"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start NormalActivity" />

    <Button
        android:id="@+id/startDialogActivity"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start DialogActivity" />

</LinearLayout>
```

We added two buttons in the LinearLayout to start NormalActivity and DialogActivity, respectively. Then update the MainActivity as follows:

```
class MainActivity : AppCompatActivity() {

    private val tag = "MainActivity"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(tag, "onCreate")
        setContentView(R.layout.activity_main)
        startNormalActivity.setOnClickListener {
            val intent = Intent(this, NormalActivity::class.java)
```

```
        startActivity(intent)
    }
startDialogActivity.setOnClickListener {
    val intent = Intent(this, DialogActivity::class.java)
    startActivity(intent)
}
}

override fun onStart() {
    super.onStart()
    Log.d(tag, "onStart")
}

override fun onResume() {
    super.onResume()
    Log.d(tag, "onResume")
}

override fun onPause() {
    super.onPause()
    Log.d(tag, "onPause")
}

override fun onStop() {
    super.onStop()
    Log.d(tag, "onStop")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(tag, "onDestroy")
}

override fun onRestart() {
    super.onRestart()
    Log.d(tag, "onRestart")
}
}
```

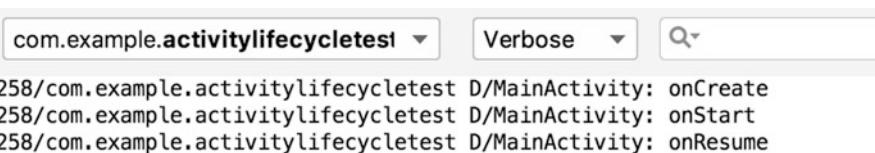
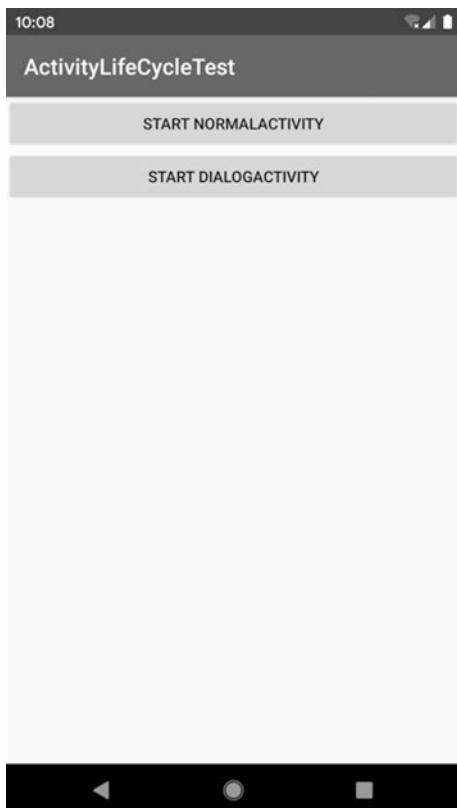
In onCreate() method, we register the button click event. Clicking the first button will start NormalActivity and clicking the second button will start DialogActivity. We will print a string that can reflect which method gets called to all the 7 callbacks mentioned previously. Then we can use Logcat to see how lifecycle looks like in Activity.

Run the app and UI should look like Fig. 3.25.

Logcat should show something like Fig. 3.26.

We can see from the logs that when MainActivity got created for the first time, the execution order of these callbacks is onCreate(), onStart() and onResume(). Now click the first button which will start NormalActivity as shown in Fig. 3.27.

**Fig. 3.25** UI of MainActivity



**Fig. 3.26** Logs when start the activity

And Logcat will show something like Fig. 3.28.

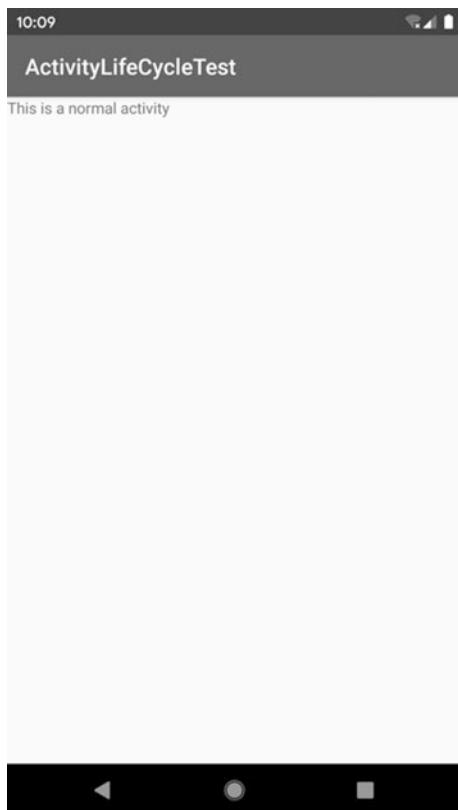
Since NormalActivity has already covered MainActivity, onPause() and onStop() had been called. Press Back button and go back to MainActivity, the logs are shown in Fig. 3.29.

MainActivity had been in stopped state, and thus onRestart() had been called and then onStart(), onResume() would be called. Notice that onCreate() method didn't get called because MainActivity didn't get recreated.

Now click the second button to start DialogActivity as shown in Fig. 3.30.

Look at Logcat information, it should be like Fig. 3.31.

**Fig. 3.27** UI of  
NormalActivity



```
com.example.activitylifecycletest ▾ Verbose ▾ Q▼  
258/com.example.activitylifecycletest D/MainActivity: onPause  
258/com.example.activitylifecycletest D/MainActivity: onStop
```

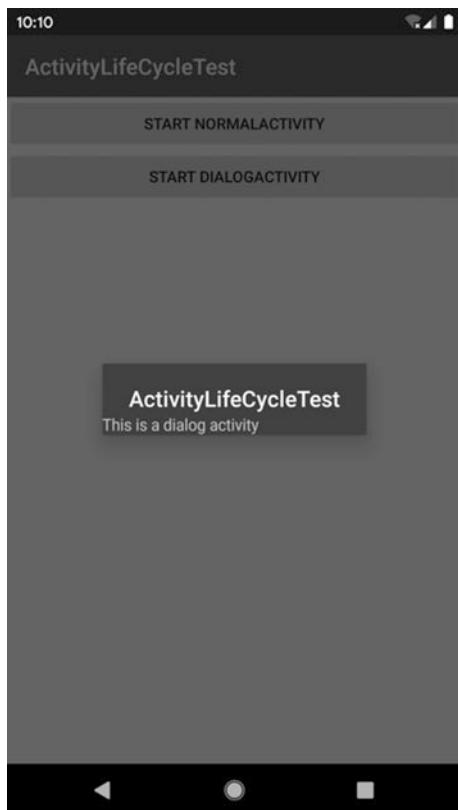
**Fig. 3.28** Logs when opening NormalActivity

```
com.example.activitylifecycletest ▾ Verbose ▾ Q▼  
258/com.example.activitylifecycletest D/MainActivity: onRestart  
258/com.example.activitylifecycletest D/MainActivity: onStart  
258/com.example.activitylifecycletest D/MainActivity: onResume
```

**Fig. 3.29** Logs when go back to MainActivity

From the logs, we can see that only onPause() method got called and onStop() didn't get called. This is because DialogActivity didn't cover the whole screen and

**Fig. 3.30** UI of DialogActivity



com.example.activitylifecyclest ▾ Verbose ▾ Q  
258/com.example.activitylifecycletest D/MainActivity: onPause

**Fig. 3.31** Logs when open DialogActivity

we can still see MainActivity. Thus MainActivity is just in paused state instead of stopped state.

Then, by pressing Back button, only onResume() method will be called for MainActivity as shown in Fig. 3.32.

Finally, click Back button again to exit MainActivity and the app. The logs will look like Fig. 3.33.

We can see that onPause(), onStop() and onDestroy() methods were called and then MainActivity got destroyed.

Hopefully the above experiment can help you understand the lifecycle of Activity easier.



Fig. 3.32 Logs when go back to MainActivity again

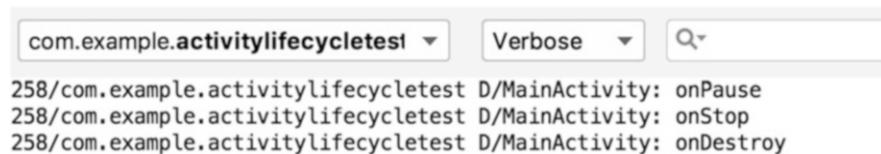


Fig. 3.33 Logs when exiting the app

### 3.4.5 Recycling Activity

We mentioned previously that, when an activity gets into stopped state, it can be recycled by the system. Now imagine that there is Activity A in the app, and user starts Activity B from A, then A will be in stopped state. Unfortunately, the usable memory is low, and system recycles activity A. Then, user press Back button to go back to A, what will happen here? So activity A will still be shown. However, `onRestart()` won't get called; instead, `onCreate()` will be called since activity A will be recreated in this scenario.

This seems fine. However, activity A might have some temporary data and states. For example, there is a text input field in MainActivity, and you input a string, then, start NormalActivity, and then MainActivity gets recycled, then you click Back to go back to MainActivity, you will find that the string you typed in is gone since MainActivity has been recreated.

Such user experience is less than ideal, and we need to solve this problem. Luckily Activity provided `onSaveInstanceState()` callback to help us solve this problem. This method will be called before the activity gets recycled.

`onSaveInstanceState()` method takes a param of type `Bundle` which provides a series of methods to save the data. We can use `putString()` method to save string and `putInt()` to save integer, etc. Every put method takes two params, the first is the key which will be used to get value from `Bundle` and the second param is the data that needs to be saved.

We can add the following code to save the temporary data:

```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
    val tempData = "Something you just typed"  
    outState.putString("data_key", tempData)  
}
```

After saving the data, how can we recover the data? You might have noticed that in `onCreate()` method, we also have param of type `Bundle` which usually is null. However, if you use `onSaveInstanceState()` to save the data before activity gets recycled by the system then this param will have all the data that have been saved.

Modify the `onCreate()` method in `MainActivity` to the following code:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(tag, "onCreate")
    setContentView(R.layout.activity_main)
    if (savedInstanceState != null) {
        val tempData = savedInstanceState.getString("data_key")
        Log.d(tag, tempData)
    }
    ...
}
```

After getting the value, we can do whatever we need to do with the data, like set the string to the text input field, etc. Here we simply print the string.

You probably noticed that to save and get data in `Bundle` is very similar with `Intent`. We can save the data in `Bundle` object first and put the `bundle` object into intent, then in the target activity we can get the `bundle` object from intent and get data from `bundle` object. I won't list the code here; you can try it out yourself!

When the screen has been rotated, activity will be recreated and data will be lost without saving. Though this can be solved by using `onSaveInstanceState()` method, it is not recommended to do so. We will learn how to properly handle such cases in Sect. 13.2.

## 3.5 Launch Mode of Activity

There are 4 types of launch mode for activity: `standard`, `singleTop`, `singleTask`, and `singleInstance`. We can use the `<activity>` element in `AndroidManifest.xml` file to specify the launch mode with `android:launchmode` attribute. Let's see what's the difference between these modes.

### 3.5.1 Standard

`Standard` mode is the default mode of activity. As you can easily tell every activity we have implemented so far used the `standard` mode. Now you should know that Android uses back stack to manage the activity and in `standard` mode, when a new activity gets started, it will be pushed into the stack and at the top of the stack. For

activities that use standard mode, system will also create a new instance of the activity when starting the activity.

Now let's use an example to demonstrate this. We can use ActivityTest project to test. So close ActivityLifeCycleTest project and open ActivityTest project. Modify the onCreate() method in FirstActivity as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("FirstActivity", this.toString())
    setContentView(R.layout.first_layout)
    button1.setOnClickListener {
        val intent = Intent(this, FirstActivity::class.java)
        startActivity(intent)
    }
}
```

So, we're starting FirstActivity from FirstActivity which is a little bit odd as who will see the same screen on top of the old screen. We don't normally do so in real world, but it is a good example to learn the standard mode. We added the code to print the current instance of the activity.

Reinstall and run the app, click the button in FirstActivity twice, and you should see the logs as shown in Fig. 3.34.

We can easily tell from the logs that every time the button has been clicked, a new instance of FirstActivity will be created and thus there will be 3 instances of FirstActivity which means that you need to click Back button 3 times to exit the app.

Figure 3.35 demonstrates how standard mode works.

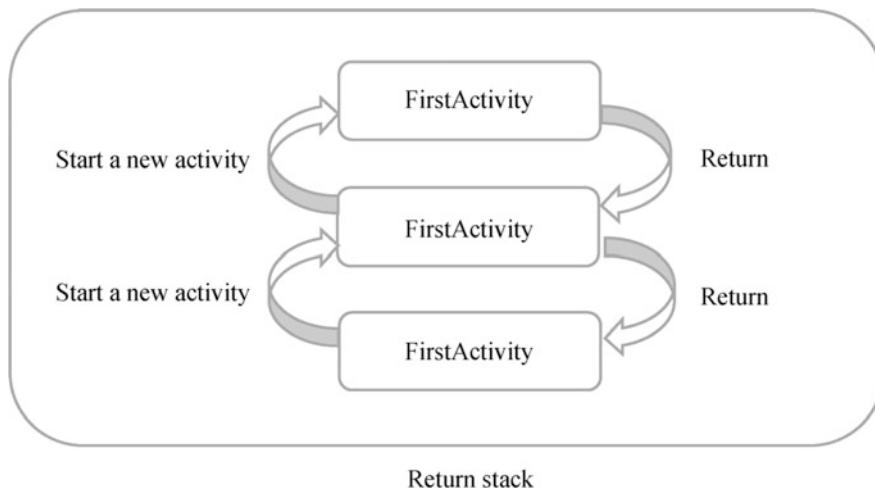
### 3.5.2 singleTop

Sometimes standard mode doesn't make a lot of sense since if an activity is already at the top of stack, why create a new activity instance? Android system does provide other ways to launch an activity, for example singleTop mode. If activity is configured with singleTop mode, and if the top of the stack is already an instance of the activity, then the system will use the existing instance instead of creating a new instance.

Let's use real example to demonstrate this. Modify the launch mode of FirstActivity in AndroidManifest.xml as follows:



Fig. 3.34 Logs in standard mode



**Fig. 3.35** Diagram to illustrate standard mode



**Fig. 3.36** Logs in singleTop mode

```
<activity
    android:name=".FirstActivity"
    android:launchMode="singleTop"
    android:label="This is FirstActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Reinstall and run the app, check the Logcat, and you will see an instance of `FirstActivity` has been created as shown in Fig. 3.36.

Then, no matter how many times you click the button, there will be no more logs captured. This is because the current activity instance is already at the top of the stack and every time when starting `FirstActivity`, system will just reuse the activity at the top of the stack. Since there is only one instance of `FirstActivity`, you just need to press Back button once to exit the app.

However, if `FirstActivity` is not at the top of the stack, starting `FirstActivity` will still create a new instance. Let's modify `onCreate()` method in `FirstActivity` as follows:

```
283/com.example.activitytest D/FirstActivity: com.example.activitytest.FirstActivity@2913896
283/com.example.activitytest D/SecondActivity: com.example.activitytest.SecondActivity@4b8b9f6
283/com.example.activitytest D/FirstActivity: com.example.activitytest.FirstActivity@361d8c4
```

**Fig. 3.37** Logs in singleTop mode

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("FirstActivity", this.toString())
    setContentView(R.layout.first_layout)
    button1.setOnClickListener {
        val intent = Intent(this, SecondActivity::class.java)
        startActivity(intent)
    }
}
```

This time we will start SecondActivity. Then modify the onCreate() method in SecondActivity as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("SecondActivity", this.toString())
    setContentView(R.layout.second_layout)
    button2.setOnClickListener {
        val intent = Intent(this, FirstActivity::class.java)
        startActivity(intent)
    }
}
```

The SecondActivity will print a string and will start FirstActivity when button gets clicked. Rerun the app, in FirstActivity, click the button to open SecondActivity, and then click the button there to open FirstActivity again.

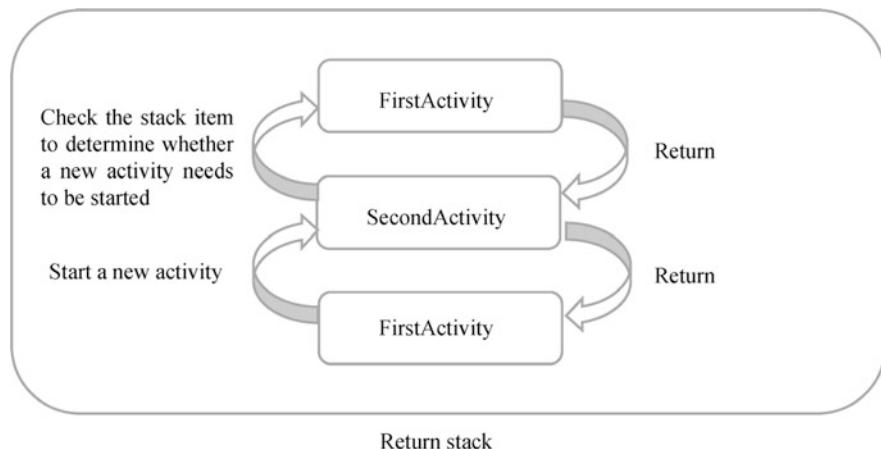
The Logcat will show information as in Fig. 3.37.

We can see that system created two different instances of FirstActivity. This is because when SecondActivity tries to start FirstActivity, the top of back stack is SecondActivity. Thus a new instance of FirstActivity will be created. Press the Back button will go back to SecondActivity, press the Back button again will go to FirstActivity, and then pressing Back button again will exit the app.

The mechanism of how singleTop works can be illustrated by Fig. 3.38.

### 3.5.3 *singleTask*

We can use singleTop to solve the problem of same activity instances stack up. However, as you can see from last section, if an instance of activity is not at the top of back stack, then a new instance can still be created. Is there any way to



**Fig. 3.38** How singleTop mode works

only allow one instance of activity get created through the app? We can use singleTask to do it. When an activity's launch mode is singleTask, every time when starting this activity, system will check the existence of this activity in the back stack and if there is an instance of it, system will pop all activities above out of stack and reuse this instance.

Let's modify the launch mode of FirstActivity in AndroidManifest.xml file as follows:

```
<activity
    android:name=".FirstActivity"
    android:launchMode="singleTask"
    android:label="This is FirstActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Then, override onRestart() method in FirstActivity and print log as below:

```
override fun onRestart() {
    super.onRestart()
    Log.d("FirstActivity", "onRestart")
}
```

Then override onDestroy() method in SecondActivity and print log as below:

```
override fun onDestroy() {
    super.onDestroy()
```

```

com.example.activitytest (23890) ▾ Verbose ▾ Q
890/com.example.activitytest D/FirstActivity: com.example.activitytest.FirstActivity@7438bfa
890/com.example.activitytest D/SecondActivity: com.example.activitytest.SecondActivity@ab6434f
890/com.example.activitytest D/FirstActivity: onRestart
890/com.example.activitytest D/SecondActivity: onDestroy

```

**Fig. 3.39** Logs in singleTask mode

```

    Log.d("SecondActivity", "onDestroy")
}

```

Rerun the app from FirstActivity, click the button to open SecondActivity, and then click the button there to open FirstActivity.

Logcat should show something like Fig. 3.39.

From the logs we can see that when SecondActivity starts FirstActivity, there is an instance of FirstActivity below SecondActivity, SecondActivity will pop and FirstActivity will be at the top of stack again, and thus FirstActivity's onRestart() method and SecondActivity's onDestroy() method will be called. Now there is only one activity instance in the stack which is FirstActivity and press Back button once will exit the app.

The mechanism of singleTask can be illustrated with Fig. 3.40.

### 3.5.4 *singleInstance*

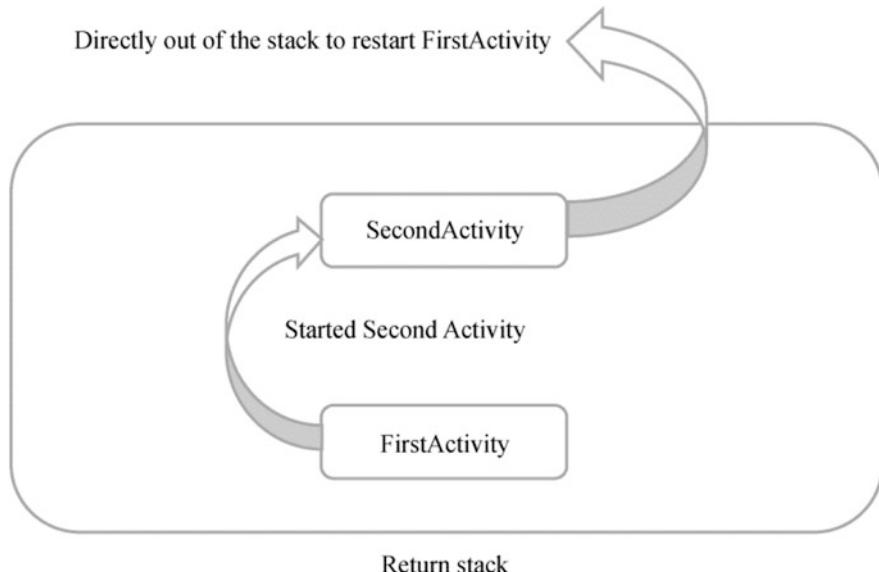
singleInstance mode is the most complicated mode among all the 4 modes. Unlike the other 3 modes, when starting an activity with singleInstance, system will create a new back stack to manage this activity (if singleTask mode specifies different taskAffinity, then another back stack will be used too). What's the meaning of this? Imagine the following scenario, if there is an activity in our app that can be used by other apps, and we need to share the instance of this activity, how to achieve this? Apparently the previous 3 modes cannot serve the purpose since each app has their own back stack and the same activity will have different instances in different back stack. But we can use singleInstance mode to solve the problem. In this mode, there will be a separate back stack to manage this activity and the back stack is shared among all the apps that are using this activity.

Let's use some examples to help understand. Modify SecondActivity's launch mode in AndroidManifest.xml as follows:

```

<activity android:name=".SecondActivity"
    android:launchMode="singleInstance">
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="com.example.activitytest.MY_CATEGORY" />
    </intent-filter>

```



**Fig. 3.40** singleTask mode mechanism

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

We first make the launch mode of SecondActivity to be singleInstance, and then modify the onCreate() method in FirstActivity as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("FirstActivity", "Task id is $taskId")
    setContentView(R.layout.first_layout)
    button1.setOnClickListener {
        val intent = Intent(this, SecondActivity::class.java)
        startActivity(intent)
    }
}
```

Here we print the back stack's ID in onCreate() method. Notice another syntax sugar example with taskId which actually will call getTaskId() method in the parent class. Then modify onCreate() method in SecondActivity as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d("SecondActivity", "Task id is $taskId")
    setContentView(R.layout.second_layout)
```



The screenshot shows the Android Logcat interface. At the top, there are dropdown menus for 'com.example.activitytest (27345)' and 'Verbose'. A search bar with a magnifying glass icon is also at the top. Below the header, the log output is displayed in three lines:

```
345/com.example.activitytest D/FirstActivity: Task id is 37  
345/com.example.activitytest D/SecondActivity: Task id is 38  
345/com.example.activitytest D/ThirdActivity: Task id is 37
```

Fig. 3.41 Logs under singleInstance

```
button2.setOnClickListener {  
    val intent = Intent(this, ThirdActivity::class.java)  
    startActivity(intent)  
}  
}
```

We print the current back stack's ID in onCreate() method and then modify the button click event handling code to start ThirdActivity. Next modify onCreate() in ThirdActivity as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    Log.d("ThirdActivity", "Task id is $taskId")  
    setContentView(R.layout.third_layout)  
}
```

We also print the ID of current back stack here.

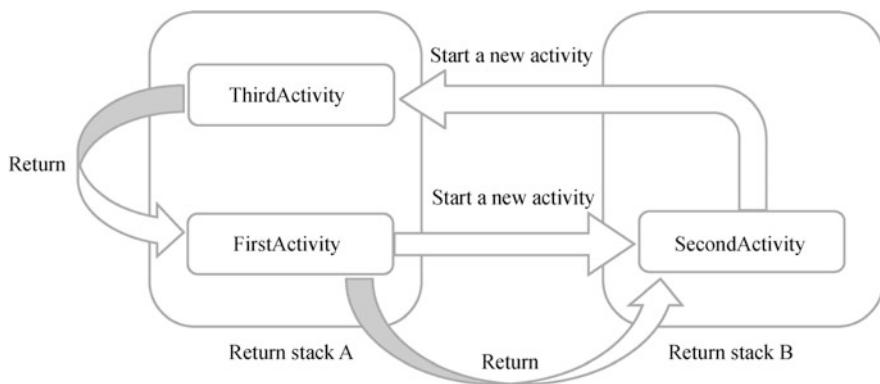
Rerun the app and in FirstActivity, click the button to open SecondActivity, then click the button there to open ThirdActivity.

The Logcat should have information as shown in Fig. 3.41.

From the logs we can see that the task ID of SecondActivity is different from FirstActivity's and ThirdActivity's which means that SecondActivity instance is indeed in a separate back stack and there is only one activity in the stack which is the instance of SecondActivity .

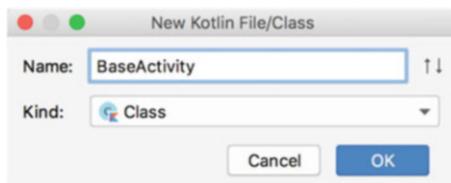
Now if we press Back button, you will find that FirstActivity will show up. Pressing Back button again will show SecondActivity, then the third press on Back button will exit the app. Why this happens? This is because FirstActivity and ThirdActivity are in the same back stack, so Back button will first pop ThirdActivity's out of stack and then FirstActivity would be at the top now, and FirstActivity will show up. After FirstActivity has been popped out, that back stack is empty and another back stack where the instance of SecondActivity is will be used to show. Popping the SecondActivity will cause all back stacks to empty, and thus app will exit.

The mechanism of singleInstance is shown in Fig. 3.42.



**Fig. 3.42** Mechanism of singleInstance mode

**Fig. 3.43** Creating BaseActivity class



## 3.6 Activity Best Practices

We've already covered a lot about activity. Let's learn some best practices to correctly use activity that will help you a lot in real world app development.

### 3.6.1 Identifying the Current Activity

We will see how to determine the current activity in this section. Imagine that you need to make some UI changes on some codebase that you are not familiar, and you need to know which activity the UI component is in. It may be not easy all the time. Using the technique here, you can easily find the corresponding activity.

Let's work on top of ActivityTest. First, create BaseActivity class by right clicking com.example.activitytest □ New □ Kotlin File/Class. Set the name to BaseActivity and set Kind to Class, as shown in Fig. 3.43.

Notice that unlike regular Activity, we do not need to register BaseActivity in AndroidManifest.xml, and thus we can make it a regular Kotlin class. Let BaseActivity inherit AppCompatActivity and override onCreate() as shown below:



Fig. 3.44 Logs from BaseActivity

```
open class BaseActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        Log.d("BaseActivity", javaClass.simpleName)  
    }  
  
}
```

We add log in `onCreate()` to print the class name of the current activity instance. Notice that, `javaClass` in Kotlin gets the `Class` object of the current instance which is equal to `getClass()` in Java. In Kotlin, `BaseActivity::class.java` gets the `Class` object of `BaseActivity` class which is equal to `BaseActivity.class` in Java. In the above code, we first get the `Class` object of the current instance and then use `simpleName` to get the class name of the current instance.

Next, we need to make `BaseActivity` to be the super class of all `Activity` in `ActivityTest` by decorating it with `open` keyword and then modify `FirstActivity`, `SecondActivity` and `ThirdActivity` to let them inherit `BaseActivity` instead of `AppCompatActivity`. Since `BaseActivity` inherits `AppCompatActivity`, the existing functionality is not affected as the classes still inherit everything in `Activity`.

Run the app again and click buttons to open `FirstActivity`, `SecondActivity` and `ThirdActivity`. Observe Logcat window and you should see logs as shown in Fig. 3.44.

With the above code, whenever we open an `Activity`, the name of this `Activity` will be logged to help us know which `Activity` we are looking at.

### 3.6.2 Exiting the App from Anywhere

Recall the test we just did for launch mode. When we're at `ThirdActivity`, we need to press the Back button 3 times to exit the app. Pressing Home button will only put the app into background but won't exit the app. How can we exit the app whenever and wherever?

The solution is quite easy. We can use a controller that has reference to all the activities to manage the activities in the app.

Let's create a new singleton `ActivityCollector` to host the collection of activities as shown below:

```
object ActivityCollector {

    private val activities = ArrayList<Activity>()

    fun addActivity(activity: Activity) {
        activities.add(activity)
    }

    fun removeActivity(activity: Activity) {
        activities.remove(activity)
    }

    fun finishAll() {
        for (activity in activities) {
            if (!activity.isFinishing) {
                activity.finish()
            }
        }
        activities.clear()
    }

}
```

Here we use singleton because we only need one collection in one app. In the controller. We use an `ArrayList` to store the activities, then we can use `addActivity()` method to add activity in the `ArrayList`; use `removeActivity()` to remove activity from `ArrayList`; use `finishAll()` to destroy all the activities in `ArrayList`. Notice that, before destroy an activity, we need to use `activity.isFinishing` to check if the activity is already in the process of getting destroyed, if not we will call `finish()` method to destroy it.

Let's modify  `BaseActivity` as follows:

```
open class BaseActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(" BaseActivity ", javaClass.simpleName)
        ActivityCollector.addActivity(this)
    }

    override fun onDestroy() {
        super.onDestroy()
        ActivityCollector.removeActivity(this)
    }

}
```

The `onCreate()` method in  `BaseActivity` will call the `addActivity()` method in `ActivityController` to add the current activity to the collection. Then `onDestroy()` in  `BaseActivity` gets overridden with calling the `removeActivity()` in `ActivityController` to remove a soon to be destroyed activity from the collection.

After this, you can exit app anywhere you want. This can be done by simply calling `ActivityController.finishAll()` method. For example, if we want to exit the app in `ThirdActivity` by clicking the button there, we can modify the code as follows:

```
class ThirdActivity : BaseActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d("ThirdActivity", "Task id is $taskId")
        setContentView(R.layout.third_layout)
        button3.setOnClickListener {
            ActivityCollector.finishAll()
        }
    }
}
```

Of course, you can kill the current process after destroying all the actives to ensure the app is totally killed by using the code below:

```
android.os.Process.killProcess(android.os.Process.myPid())
```

`killProcess()` method is used to kill a process which will take a process ID param which can be acquired by using `myPid()` method. It is worth noting that `killProcess()` can only kill the current app's process, not other apps'.

### 3.6.3 Best Practice to Start Activity

As we've already covered previously, we can start the activity by building the intent and then use `startActivity()` or `startActivityForResult()` to start the activity. We also use the intent to pass data between activities.

If `SecondActivity` needs two string params then it would be very straightforward for us to write the following code to start the activity:

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("param1", "data1")
intent.putExtra("param2", "data2")
startActivity(intent)
```

This will work for sure, but in real projects, this may not be the best approach. For example, if you're not the author of `SecondActivity` and now you need to start

SecondActivity, but you're not sure about what params are needed to start SecondActivity, then you need to either read SecondActivity or ask the author of this class. Both will take time and we can solve this problem by using another approach.

Let's modify the code in SecondActivity as follows:

```
class SecondActivity : BaseActivity() {
    ...
    companion object {
        fun actionStart(context: Context, data1: String, data2: String) {
            val intent = Intent(context, SecondActivity::class.java)
            intent.putExtra("param1", data1)
            intent.putExtra("param2", data2)
            context.startActivity(intent)
        }
    }
}
```

Here we used companion object and defined actionStart() method inside it. We use companion object because in Kotlin, all the methods defined in companion object can be used as Java static methods and I will cover more about companion object in later part of this chapter.

In the actionStart() method, we created an intent instance and saved the required data into the intent, then used startActivity() method to start SecondActivity.

What's the benefit of this approach? The most important one is the clarity it provides. All the params that are needed are in the method params and without reading the code of SecondActivity nor asking the author. You should know what params are needed to start SecondActivity. Also, this approach simplifies the code to start activity as we only need one line of code to start SecondActivity as following code has shown:

```
button1.setOnClickListener {
    SecondActivity.actionStart(this, "data1", "data2")
}
```

This approach will help reduce the time to read and write code and maybe questions from your colleague.

## 3.7 Kotlin Class: Standard Functions and Static Methods

This is our first Kotlin Section which almost all the chapters after will have too. Basically, we only covered a fraction of Kotlin in Chap. 2 without any advanced topics covered. Thus I'd like to dedicate one section in each chapter to introducing more advanced Kotlin knowledge especially those that are used in that chapter.

### 3.7.1 Standard Functions: *with, run, and apply*

The Kotlin standard functions are functions defined in Standard.kt file. You can call the standard functions anywhere in Kotlin code.

Though the amount of standard functions is not that huge, it would still take some time to learn all of them. Thus we will cover a few most commonly used ones.

In last chapter, we already learned how to use let which can be used together with? . operators to check nullability and we won't spend time on it here.

Let's begin with with function. The with function takes two params: the first param can be an object of any type and the second param is a Lambda expression. The with function will provide the context for the first param in Lambda expression and then return the last line of code of the Lambda expression. Here is an example:

```
val result = with(obj) {
    // here is the context of obj
    "value" // return value of with function
}
```

What does this function do? It will make the code more succinct when we need to call the different methods of the same object multiple times. Let me use an example to explain.

Imagine that we have a list of fruits and we want to eat all of them and then print the fruits names. Then we can have the code below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
val builder = StringBuilder()
builder.append("Start eating fruits.\n")
for (fruit in list) {
    builder.append(fruit).append("\n")
}
builder.append("Ate all fruits.")
val result = builder.toString()
println(result)
```

The logic here is very simple. We use StringBuilder to construct the string of fruits and then print it. Running the above code will yield result as shown in Fig. 3.45.

It is easy to find that we used the methods of builder object multiple times and let's use with function to make the code more concise as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
val result = with(StringBuilder()) {
    append("Start eating fruits.\n")
    for (fruit in list) {
        append(fruit).append("\n")
    }
    append("Ate all fruits.")
```



**Fig. 3.45** Print the result of fruit strings

```
    toString()
}
println(result)
```

It may look confusing at first glance, but the logic is actually very simple. We pass `StringBuilder` object as the first argument of `with` function, then the context of the Lambda expression is this object. Thus we don't need to use builder again and again, but instead we can directly call `append()` and `toString()` methods. The last line of code is the result which we will just print.

The results are the same for these two approaches, but you can easily tell that `with` function can make the code much more compact.

Next, let's learn another widely used standard function: `run` function. `Run` function is similar to `with` function but with some syntax changes. First, it cannot be called directly, and we have to call an object's `run` function. Secondly, `run` function only takes a Lambda param and provides the context for the object in Lambda expression. Other than these two differences, `run` function is the same as `with` function. An example is:

```
val result = obj.run {
    // context of obj
    "value" // return value
}
```

Now we can use `run` function to implement the functionality mentioned above as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
val result = StringBuilder().run {
    append("Start eating fruits.\n")
    for (fruit in list) {
        append(fruit).append("\n")
    }
    append("Ate all fruits.")
    toString()
```

```

    }
    println(result)
}

```

Overall it is not very different from run function and the result will be the same.

In the end let's learn how to use apply function. Apply function is extremely like run function in a way that it will be applied to an object and only takes Lambda expression as the only param, also will provide the context of the calling object in the Lambda expression, however, apply will only return the object itself. An example is as follows:

```

val result = obj.apply {
    // obj context
}
// result == obj

```

Now let's apply the apply function to rewrite the logic above as shown below:

```

val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
val result = StringBuilder().apply {
    append("Start eating fruits.\n")
    for (fruit in list) {
        append(fruit).append("\n")
    }
    append("Ate all fruits.")
}
println(result.toString())

```

Notice that apply function cannot return the value but the obj itself, and thus the result is actually an object of StringBuilder and we need to call its `toString()` method before we can print, and the result will be the same as run and with functions and I will not attach here.

Now we've finished the introduction of 3 of the most widely used Kotlin standard functions and you can see that with, run, and apply functions are quite similar and under most circumstances, we can use them interchangeably. Recall the code we wrote to start the activity in best practice section:

```

val intent = Intent(context, SecondActivity::class.java)
intent.putExtra("param1", "data1")
intent.putExtra("param2", "data2")
context.startActivity(intent)

```

Here for every param we need to call `intent.putExtra()` once, and if we have 10 params we will call it 10 times. This makes it a perfect scenario we can apply standard functions we just learned, for instance we can use the following code:

```

val intent = Intent(context, SecondActivity::class.java).apply {
    putExtra("param1", "data1")
}

```

```
    putExtra("param2", "data2")
}
context.startActivity(intent)
```

As you can see that the context of Lambda expression here is the intent object and we don't need to type intent.putExtra() but instead will use putExtra() directly. The more params there are, more obvious the benefits of using the standard functions will be.

OK, that concludes the introduction to Kotlin standard functions in this section, in later chapters of this book, we will have more code examples that will use the standard functions.

### 3.7.2 Define Static Methods

Static methods are called class methods in some programming languages as it doesn't require creation of an instance of the class so that it can be used. All major languages support static methods.

By simply adding the static keyword to the method, we can define a static method in Java as shown below:

```
public class Util {

    public static void doAction() {
        System.out.println("do action");
    }

}
```

This is a very simple utility class and doAction() is a static method. As mentioned, we don't need to create an instance of the Util class to use this method but can use it directly by Util.doAction(). As you can tell those static methods are good for util methods since there is no need to create the instance of the util class and will be used globally.

However, unlike majority of the programming languages, Kotlin weakened the concept of static methods, and it is actually not trivial to define a static method.

Why Kotlin is so different? This is because Kotlin provides a better solution which is singleton class which we already learned in last section.

A util class should be a singleton class in Kotlin, for example, we can implement the above util class in Kotlin using the following code:

```
object Util {  
  
    fun doAction() {  
        println("do action")  
    }  
  
}
```

Though `doAction()` method here is not static function, but we can still use it by the syntax `Util.doAction()`.

However, if we use singleton, then all the methods inside the singleton will function like static methods. What if we want a certain method in a class to be static method? Then we can use the companion object we learned from the best practice section, here is an example:

```
class Util {  
  
    fun doAction1() {  
        println("do action1")  
    }  
  
    companion object {  
  
        fun doAction2() {  
            println("do action2")  
        }  
  
    }  
  
}
```

Here we made `Util` a normal class, defined `doAction1()` method and then defined `doAction2()` method in companion object. Now these two methods will be quite different as we need to instantiate `Util` class so that we can use `doAction1()` method, but we can use `doAction2()` method directly with syntax `Util.doAction2()`.

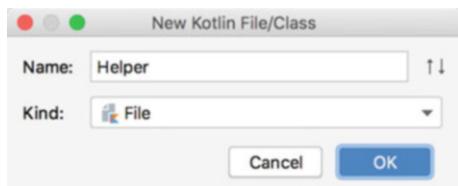
However, the `doAction2()` method is not actually a static method. The `companion object` keyword actually creates a companion class inside the `Util` class, in which the `doAction2()` method is defined as an instance method. Kotlin ensures that there is only one companion object in the `Util` class, so a call to `Util.doAction2()` is actually a call to the `doAction2()` method of the companion object in the `Util` class.

In conclusion, Kotlin doesn't provide keywords to define static methods. However, it provides some syntax support to achieve the same goal as static methods.

However, if you really want to define real static methods there are two ways you can do in Kotlin: annotation and package-level functions.

First let's look at annotation. Singleton and companion object look like static in syntax. However, they are not real static methods; thus, if you call the methods in Java as calling static method, then you will find that these methods actually don't

**Fig. 3.46** Create a new Kotlin File



exist. However, if we add `@JvmStatic` annotation to the methods in singleton or companion object, then Kotlin compiler will compile these methods into real static methods. We can use the code below as an example:

```
class Util {

    fun doAction1() {
        println("do action1")
    }

    companion object {

        @JvmStatic
        fun doAction2() {
            println("do action2")
        }
    }
}
```

Notice that `@JvmStatic` annotation can only be applied to the methods in singleton or companion object, and if you try to apply to a normal method, there will be syntax error.

Now since `doAction2()` became a real static method, we can call it by use `Util.doAction2()` both in Java and Kotlin.

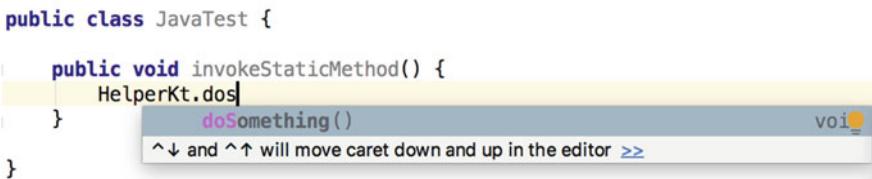
Next, let's take a look at package-level methods which are the methods that are not defined in classes such as the `main()` function we created in last section. Kotlin will compile all the package-level functions into static methods, and thus if you define a package-level method, it has to be a static method.

In order to define a package-level method, we need to create a Kotlin file. Right click any package - > New - > Kotlin File/Class, type in the file name in the dialog and select File for Kind as shown in Fig. 3.46.

Click OK to finish creation and you should see a `Helper.kt` file in the directory of the package you selected. Now all the methods we define in this file are package-level functions. Let's define a method with name `doSomething()` as shown below:



**Fig. 3.47** Call doSomething() method in Kotlin



**Fig. 3.48** Use doSomething() in Java

```

fun doSomething() {
    println("do something")
}

```

We already learned that Kotlin will compile all the package-level functions into static methods, then how should we call doSomething() method?

We can use doSomething() anywhere we want in Kotlin without creating an instance of it and we don't need to care about the package path too as shown in Fig. 3.47.

However, you won't be able to find this method yet in Java, this is because Java has no concept of package-level method and every method has to be defined in some classes. So where is the doSomething() method? Since the file we just created has the name Helper.kt, Kotlin compiler will create a HelperKt Java class automatically, and doSomething() will be a static method inside the HelperKt class. Thus we can call it HelperKt.doSomething() as shown in Fig. 3.48.

That's it for static method. The topics we discussed in this section are widely used except @JvmStatic annotation. Singleton class, companion object, package-level methods are very useful, hope you can get familiar with them.

## 3.8 Summary

This chapter is very long and filled with various topics and you might feel exhausted which is understandable. I covered programming theories and practices, from the fundamentals of activity to starting activity and sharing data, to activity lifecycle and

activity launch mode, you almost learned everything important about activity. At the end of the chapter, you also learned some best practices that can be applied to using activity.

We also learned some more advanced topics in Kotlin like how to use Kotlin standard functions and the ways to define static methods in this chapter's Kotlin section.

Your journey to learn Android has just started, there are way more things to learn in the future. So, relax and get rest to prepare for the next chapter!

# Chapter 4

## Everything About UI Development



It is natural for mobile engineers to focus on how to implement the functionality of the app when they use an app. However, users will just focus on the UI and functionality of the app instead of the implementation. While it is not our duty to come up with beautiful designs, we can use the rich UI tools provided by Android to build beautiful UIs.

You must feel bored of using the buttons for so long and now let us start to learn more about Android UI development.

### 4.1 How to Create UI?

In the past, Android UI was mainly built by writing XML manually. We can have a deeper understanding of the UI components and create UIs that are more compatible with manual code. Thus, after you master the way of writing UI with XML, you should feel comfortable to create complex UI and update the legacy UI.

However, in the recent years, a new layout released by Google got traction and that is ConstraintLayout. This layout is not fit to manually writing code. Instead, it is very easy to develop in visual editor with drag and drop. But due to the limitation of books being static, it will be hard to demonstrate the whole process of manipulating the UI with visual editor. Also, with the concern that writing XML is an essential technical skill for Android UI development, I will write XML manually to help learn how to build UI.

Now let us start first with the most used UI widgets in Android.

## 4.2 Common UI Widgets

We can build beautiful UI even with some of the most widely used widgets provided by Android system, and I will cover a few of them.

First, create a UIWidgetTest project. And to simplify the process, we can use the activity created by Android Studio with the default activity and layout.

### 4.2.1 *TextView*

TextView is one of the simplest widgets of Android which is used to display a string in the UI such as the one you see in Chap. 1 which displays Hello World!

Let us take a look at the more advanced features of TextView and update activity\_main.xml with the following code:

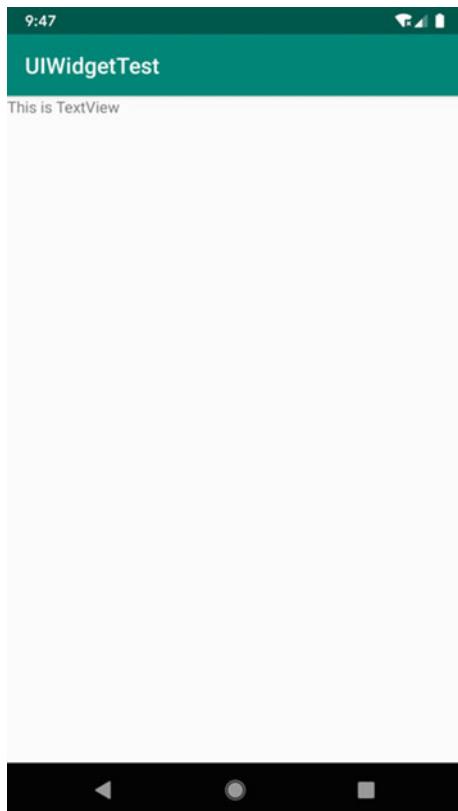
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

</LinearLayout>
```

We can ignore LinearLayout for now, in the TextView we use to define a unique identifier for the current widget which was covered in last chapter. Then we can use android:layout\_width and android:layout\_height to specify the width and height of the widget, respectively. There are three kinds of values we can use: match\_parent, wrap\_content, and fixed value. match\_parent means that the current widget has the same dimension as the parent widget, and wrap\_content means that the current widget's dimension should be determined by the content. For fixed value means it will have a fixed size which we usually use dp for unit. The dp unit is a pixel density independent unit and can accommodate the different resolutions of phones very well. The code above will make TextView has the same width of the parent layout which is the width of the phone and the height will be the height of the content. Run the app and it should be as shown in Fig. 4.1.

We can change the alignment of the content with android:gravity with optional values: top, bottom, start, end, center etc. We can also use “|” to apply multiple of them. And “center” alone will be equivalent to “center\_vertical|center\_horizontal.” Update the code as below:

**Fig. 4.1** TextView Test

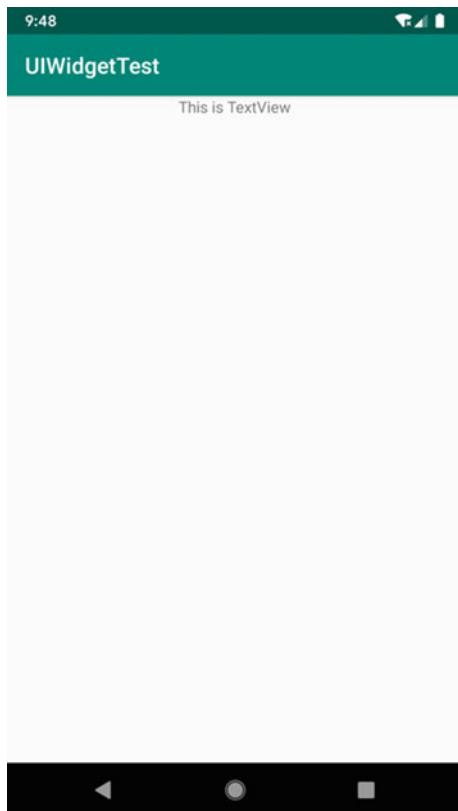
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="This is TextView"/>

</LinearLayout>
```

Now rerun the code and the result will be as shown in Fig. 4.2.

We can also modify the color and font of the TextView. Update the code as follows:

**Fig. 4.2** TextView centered

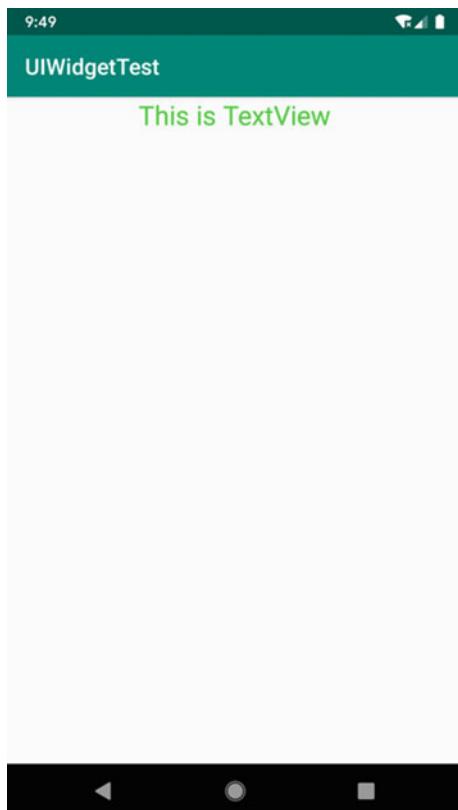
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textColor="#00ff00"
        android:textSize="24sp"
        android:text="This is TextView"/>

</LinearLayout>
```

Use android:textColor to specify the color and use android:textSize to specify the size of font which will use sp. as unit. If user changes the font size in the system

**Fig. 4.3** Update the size and color of TextView



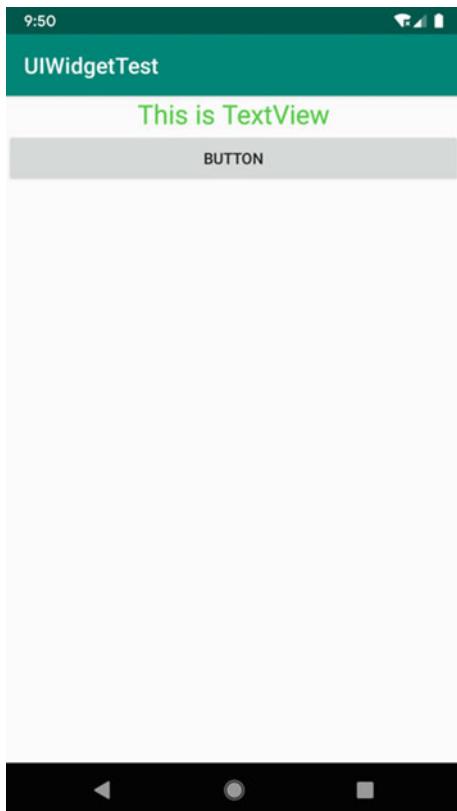
settings, then the size of the app will change accordingly. Now rerun the app and result is shown as in Fig. 4.3.

Of course, there are other properties in the TextView that you can configure. I will not cover all of them here and you can just search the official doc whenever you need to.

#### 4.2.2 *Button*

We have used button several times in last chapter. Button has similar configurable properties as TextView and we can add button in activity\_main.xml as the code shown below:

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

**Fig. 4.4** Add Button

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

    ...

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

</LinearLayout>
```

After adding button, our test app will show up like Fig. 4.4.

You will notice that we used text Button in XML; however, in the UI, it is BUTTON. This is because Android will show text in button in upper cases. You can change it by adding `android:textAllCaps = “false”` in XML to preserve the text format.

Next let us add a click event listener for button in MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        button.setOnClickListener {  
            // Add event handling logic  
        }  
    }  
}
```

Code here used functional API to set the click event because this interface is a functional interface. Every time user clicks the button, the Lambda expression will be executed. Refer Sect. 2.6.3 for more information about functional API.

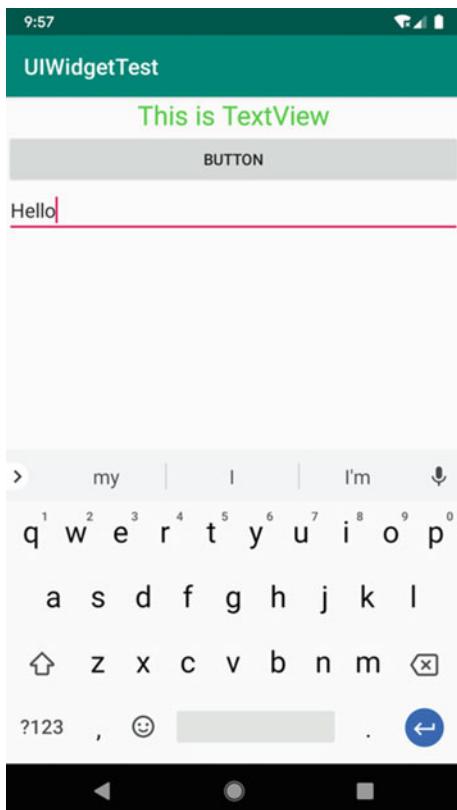
Besides the functional API, registering the listener can be done by implementing the interface as shown below:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        button.setOnClickListener(this)  
    }  
  
    override fun onClick(v: View?) {  
        when (v?.id) {  
            R.id.button -> {  
                // Event handling logic  
            }  
        }  
    }  
}
```

MainActivity implemented the View.OnClickListener interface and overrided the onClick() method. MainAcvitiy itself gets passed in the setOnClickListener() method. Refer to Sect. 2.5.3 for more information about Kotlin interface.

#### 4.2.3 *EditText*

EditText is an important widget that allows users to input the content. It is widely used in sending text message, tweets, and Messenger chats. Update activity\_main.xml with the following code:

**Fig. 4.5** EditText

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

    ...
    ...
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

Code here is very similar as previous examples. We need to define an ID, set the height and width of the widget, then set other properties unique to this widget, and that's it. Now rerun the app and it should be as shown in Fig. 4.5.

You may notice that some apps will have some hints in the input box, and when user starts to type, the hints are gone. It's very easy to add this function in Android. Update the activity\_main.xml as shown below:

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

    ...

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type something here"
        />

</LinearLayout>
```

It sets the hint text by using android:hint property. Rerun the app as shown in Fig. 4.6.

If the height of the EditText view is set to wrap\_content, as more and more characters are typed in, it will keep growing and won't look good if there are too many contents inside. We can set android:maxLines property to restrict the lines of the EditText view. Update the activity\_main.xml as shown below:

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

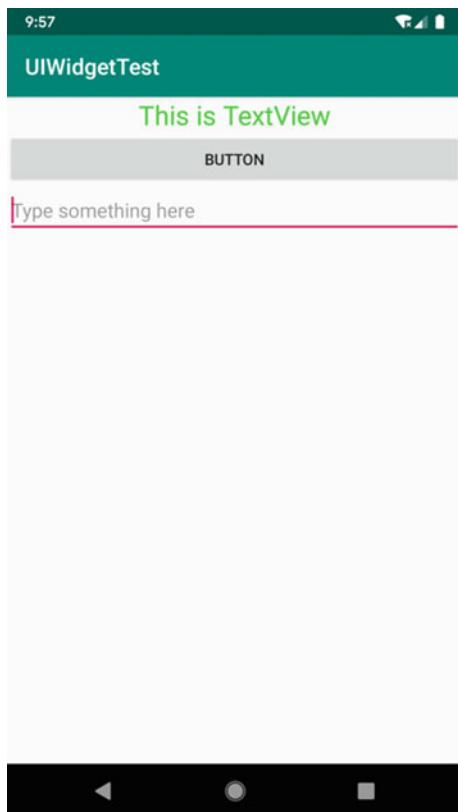
    ...

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type something here"
        android:maxLines="2"
        />

</LinearLayout>
```

By setting android:maxLines to 2, when the input text occupies more than 2 lines, text will roll up and EditText view will not keep growing as shown in Fig. 4.7.

**Fig. 4.6** Add hint to EditText



The widgets can interact with each other. For example, clicking the button to get the content of EditText can be done by the code shown below:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
    ...  
    override fun onClick(v: View?) {  
        when (v?.id) {  
            R.id.button -> {  
                val inputText = editText.text.toString()  
                Toast.makeText(this, inputText, Toast.LENGTH_SHORT).show()  
            }  
        }  
    }  
}
```

**Fig. 4.7** Set maxLines in EditText



The code here uses `getText()` to get the input content, then call `toString()` method to convert the content to string, and finally uses `Toast` to display the content.

It also used the syntax sugar for using Getter and Setter method which makes text a property of `EditText`, but under the hood, it is calling `getText()` method of `EditText`.

There is no need to remember this syntax sugar at all, as Android Studio's auto completion will show the optimized code after applying syntax sugar as shown in Fig. 4.8.

Once we type in `getText`, the first option in auto completion will be `text` and we just need to click Enter to use this syntax sugar.

With that, I will use the real methods under the hood for Getter and Setter methods throughout this book which may be different from the actual code, but you should know this is because of the syntax sugar.

Run the app again and type in some content in the `EditText` view, click the button, and the result is as shown in Fig. 4.9.

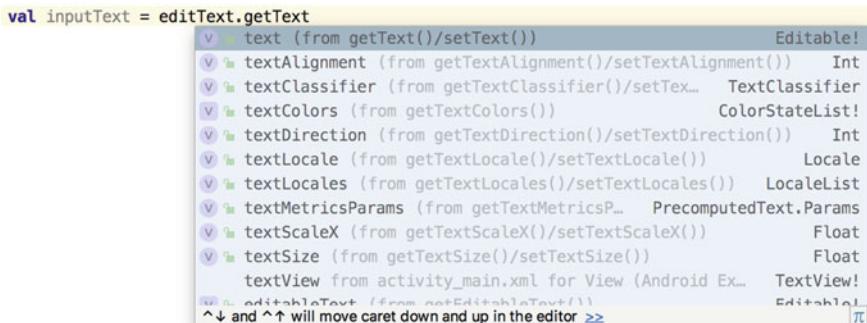
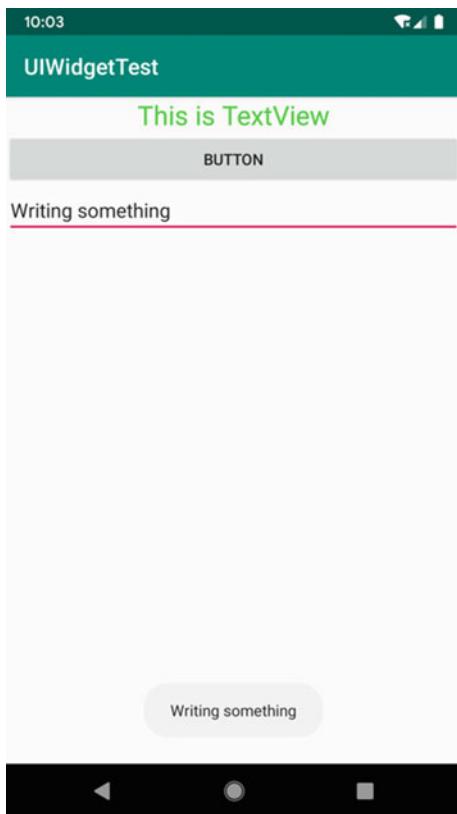


Fig. 4.8 Syntax Sugar in Android Studio

Fig. 4.9 Get the content of the EditText view



#### 4.2.4 ImageView

ImageView widget is used to show images in the UI. The images used in the examples can be downloaded from resources link in the preface but you can use

any image to replace those in the example. Images are usually in the folders that have drawable prefix followed by resolution information. Currently, most of the mobile phones have resolution xxhdpi or higher; thus, create a drawable-xxhdpi folder under the res directory, then put img\_1.png and img\_2.png in this folder.

Modify activity\_main.xml as shown below:

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

    ...
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/img_1"
    />

</LinearLayout>
```

The android:src attribute is used to set the image source for ImageView. The width and height of ImageView are set to wrap\_content because the dimension is unknown. Run the app again and the result should be as shown in Fig. 4.10.

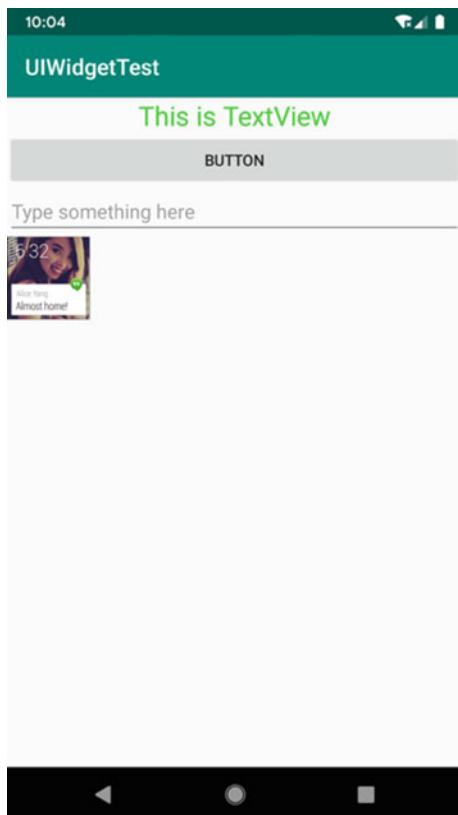
To update the image source dynamically, modify MainActivity as shown below:

```
class MainActivity : AppCompatActivity() , View.OnClickListener {

    ...
    override fun onClick(v: View?) {
        when (v?.id) {
            R.id.button -> {
                imageView.setImageResource(R.drawable.img_2)
            }
        }
    }
}
```

In the button click event handler, we use setImageResource() method of Imageview to update the image to img\_2. Run the app again, click the button, and you should see that the image of Imageview has been updated as shown in Fig. 4.11.

**Fig. 4.10** Show image in ImageView



#### 4.2.5 *ProgressBar*

ProgressBar widget is useful if we want to show the progress of some time-consuming operations like downloading. It is quite simple to use. Modify activity\_main.xml as code below:

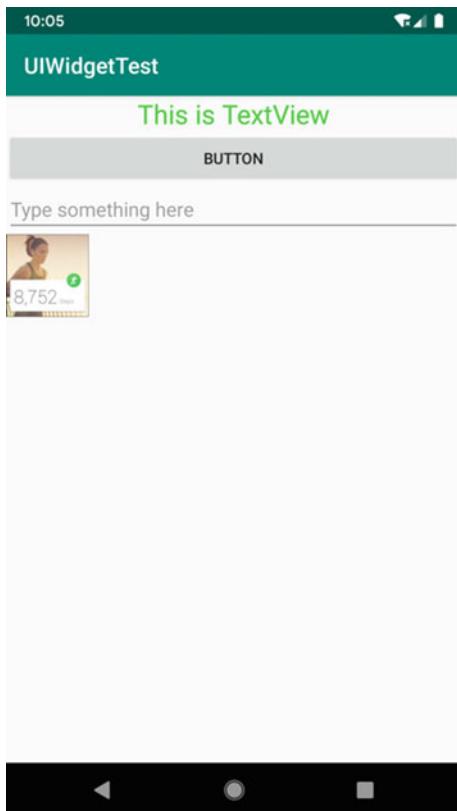
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    ...

    <ProgressBar
        android:id="@+id/progressBar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />

</LinearLayout>
```

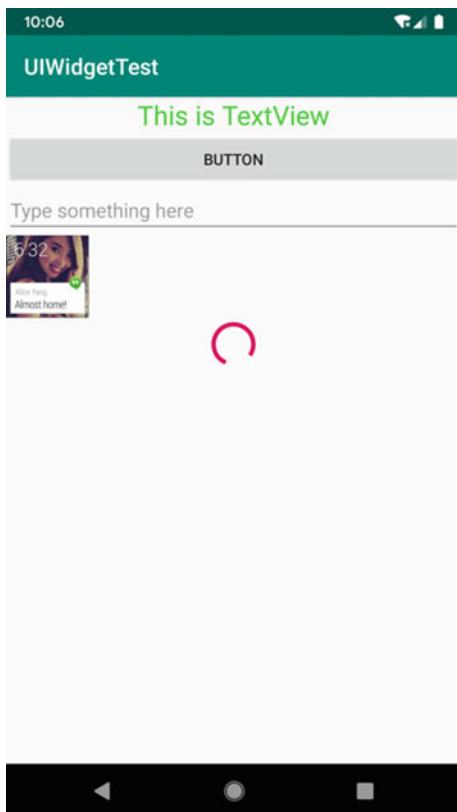
**Fig. 4.11** Dynamically update the image of ImageView



Run the app again and you should see a throbber rotating at the center of the screen as shown in Fig. 4.12.

How to make the throbber disappear when the downloading finishes? For the visibility attribute of the Android widget, there are three values: visible, invisible, and gone. Visibility can be set by using android:visibility and by default it will have a value of visible. Invisible means that the widget cannot be seen but is still at the original position and the size also doesn't change for the invisible widget. You can think of invisible as transparent. However, gone means that the widget cannot be seen and at the same time won't take any screen space. We can also dynamically set the visibility of widget by using setVisibility() method which can take View.VISIBLE, View.INVISIBLE, and View.GONE as argument.

We can use the following code to make the progress bar disappear by clicking the button and then reappear by clicking the button again.

**Fig. 4.12** ProgressBar

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
    ...  
    override fun onClick(v: View?) {  
        when (v?.id) {  
            R.id.button -> {  
                if (progressBar.visibility == View.VISIBLE) {  
                    progressBar.visibility = View.GONE  
                } else {  
                    progressBar.visibility = View.VISIBLE  
                }  
            }  
        }  
    }  
}
```

In the button click event, we use `getVisibility()` to determine if `ProgressBar` is visible or not, and update the state of the progress bar every time it gets triggered.

Run the app again and by clicking the button, you should be able to show and hide the progress bar by clicking the button on the screen.

We can also use style attribute to change how the progress bar looks. To change the progress bar from rotating loading throbber to the horizontal progress bar, modify activity\_main.xml as shown below:

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

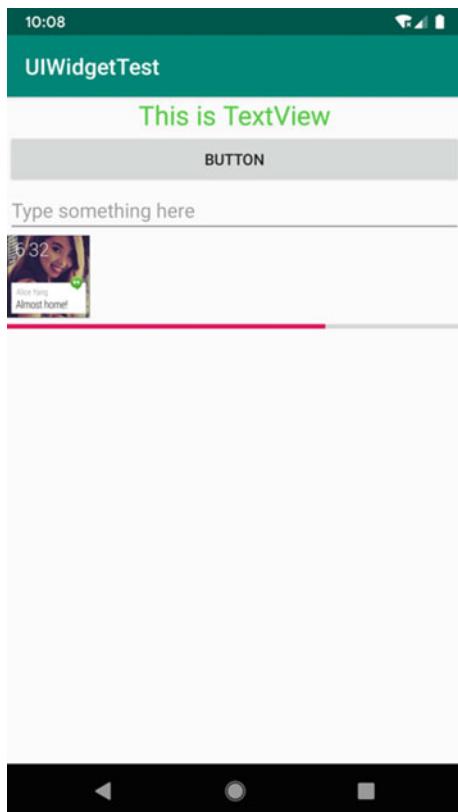
    ...
    <ProgressBar
        android:id="@+id/progressBar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="?android:attr/progressBarStyleHorizontal"
        android:max="100"
        />
</LinearLayout>
```

For horizontal progress bar, we can use android:max attribute to set the maximal value of the progress bar, and then dynamically change the progress state. Modify MainActivity with the code below:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {
    ...
    override fun onClick(v: View?) {
        when (v?.id) {
            R.id.button -> {
                progressBar.progress = progressBar.progress + 10
            }
        }
    }
}
```

After each click event, we will get the current state of the progress, and every time the state will be increased by 10. Run the app again and click the button a few times. The result is as shown in Fig. 4.13.

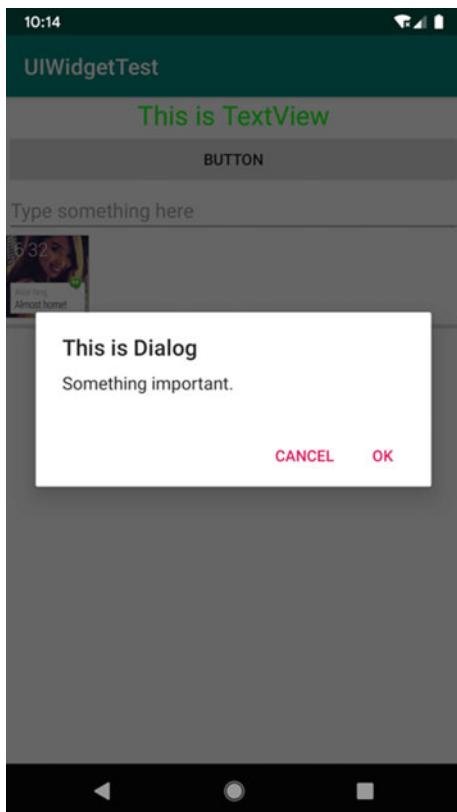
**Fig. 4.13** Horizontal progress bar



#### 4.2.6 AlertDialog

AlertDialog widget can show a dialog pop on the screen, which is above all the UI components that blocks the interaction with other components. Thus, AlertDialog should only be used to show very important information or warning message. For instance, in order to prevent user from accidentally deleting the important files, we can show a confirmation dialog before deletion. To do so, modify MainActivity with the code below:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
    ...  
    override fun onClick(v: View?) {  
        when (v?.id) {  
            R.id.button -> {  
                AlertDialog.Builder(this).apply {  
                    setTitle("This is Dialog")  
                    setMessage("Something important.")  
                    setCancelable(false)  
                    setPositiveButton("OK") { dialog, which ->  
                        ...  
                    }  
                }.show()  
            }  
        }  
    }  
}
```

**Fig. 4.14** ActionDialog

```
        }
        setNegativeButton("Cancel") { dialog, which ->
        }
        show()
    }
}
}
```

```
}
```

Notice that we used the Kotlin standard function—apply function. With apply function, we can set the title, message, and enable or disable dismissing the dialog with back button, and so on. We can use setPositiveButton() and setNegativeButton() to set the content and click event for the action buttons. The show() method will display the created dialog on the screen. Run the app again and clicking the button should show dialog as Fig. 4.14.

That's everything I want to cover for commonly used Android widgets. It is not realistic nor possible to cover everything about the Android widgets in one chapter.

This section is meant to serve as an introduction to Android widgets, and you should be able to read documents and search the web to learn more about widgets. If we need to use widgets in the following chapters that are not covered in this section, I will cover them in detail by then.

## 4.3 Three Basic Layouts

Layout is the container for various widgets to ensure that they are in the right position, and layout can be embedded in layout too. With layout, we can build complex UIs. The relationship between layout and widgets is illustrated in Fig. 4.15.

I will cover three basic layouts in this chapter. First, create a project called UILayoutTest and keep everything as it is for the activity and layout Android Studio creates for us.

### 4.3.1 *LinearLayout*

*LinearLayout* is a commonly used layout that positions the widgets inside it vertically or horizontally. As you can see from examples in last chapter, the widgets were in *Linearlayout* and they were positioned vertically, because we set the android:orientation attribute with value of vertical. Modify activity\_main.xml as follows:

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 1" />

```

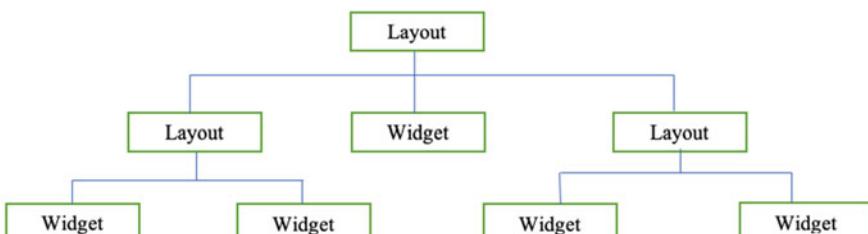


Fig. 4.15 Relationship between Layout and Widget

**Fig. 4.16** Vertical orientation for LinearLayout

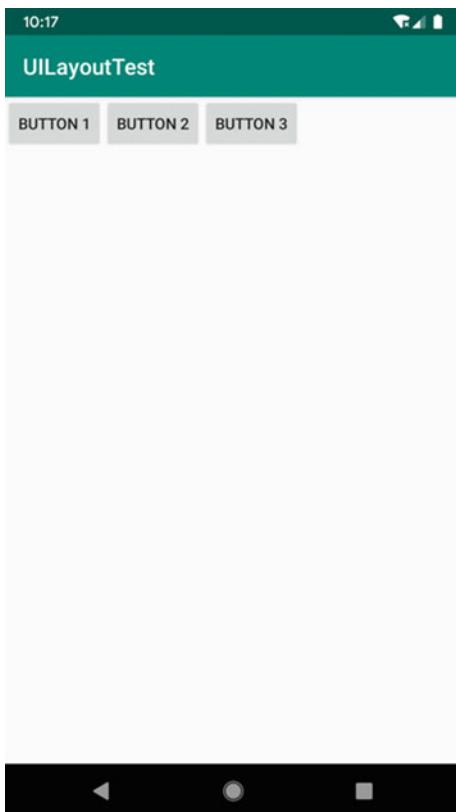


```
<Button  
    android:id="@+id/button2"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Button 2" />  
  
<Button  
    android:id="@+id/button3"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Button 3" />  
  
</LinearLayout>
```

We add three Button widgets in the LinearLayout. Then we set height and width to be wrap\_content and set the orientation to be vertical. Running the app should show the result as in Fig. 4.16.

Update the orientation to be horizontal as shown below:

**Fig. 4.17** Horizontal orientation for LinearLayout



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

    ...
</LinearLayout>
```

We set `android:orientation` attribute specifically and by default the orientation will be horizontal. Running the app again should show the result as in Fig. 4.17.

You should notice that, if the orientation is horizontal, then the width of the widgets inside cannot be `match_parent` since this will make a single widget take up the whole screen horizontally. For the same reason, if the orientation is vertical, then the height of the widget inside cannot be `match_parent`.

Now let us look at the `android:layout_gravity` attribute which looks like `android:gravity` we learned from last section. The difference is that `android:gravity` is used to specify the alignment of the text in the widget, and `android:layout_gravity` is used to

specify the alignment of the widgets in the layout. These two attributes have similar options for the values. Notice that when the orientation of the LinearLayout is horizontal, only the alignment values used for vertical direction will be effective. This is because the width of this LinearLayout is not fixed, and each time a new widget is added, the width will be changed accordingly, which makes it meaningless to specify the alignment horizontally. For the same reason, when the orientation of the LinearLayout is vertical, only the alignment values for horizontal direction will be effective. Modify activity\_main.xml with the code below:

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"
        android:text="Button 3" />

</LinearLayout>
```

Since the orientation of the LinearLayout is horizontal, we can only specify the alignment on the vertical direction like top, center\_vertical, and bottom as shown in the code above. Run the app again and the result should look like Fig. 4.18.

The android:layout\_weight attribute is used to set the size of widget by percentage and is useful to make the UI compatible in phones with different screen sizes. For instance, to build a screen with a text input and send button we can use the code below in activity\_main.xml:

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

**Fig. 4.18** Effects of layout\_gravity



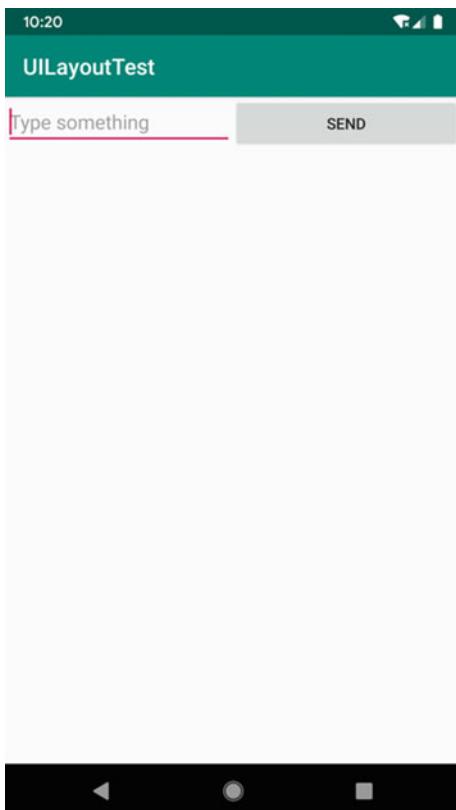
```
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<EditText
    android:id="@+id/input_message"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:hint="Type something"
    />

<Button
    android:id="@+id/send"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Send"
    />

</LinearLayout>
```

**Fig. 4.19** Set the layout\_weight



Although we set the width of EditText and Button to be 0dp, they will still show up. This is because we use android:layout\_weight attribute which makes the final decision of the width of the widget. To set the width to 0dp is the recommended practice.

By setting the android:layout\_weight to 1, EditText view and Button view will equally take the space horizontally.

Run the app again and you should see the result as shown in Fig. 4.19.

The mechanism for calculating the width here is that, the system will get the sum of the layout\_weight of all the widgets in the LinearLayout, and the size of widget is proportional to the percentage of its weight. For instance, if EditText view needs to take 3/5 of the screen width, and Button view needs to take 2/5 of the screen width, then we just need to set the layout\_weight of EditText view and Button view to be 3 and 2, respectively.

We can use the layout\_weight attribute to build better user experience. Modify activity\_main.xml as the code below:

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

    <EditText
        android:id="@+id/input_message"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:hint="Type something"
        />

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

</LinearLayout>
```

We only set the android:layout\_weight attribute of EditText and changed the width of Button view to be wrap\_content. This will make the Button take enough screen space to show and the EditText view takes the rest of the screen. This will make the UI looks good in different screens. Run the app again and the result should be as shown in Fig. 4.20.

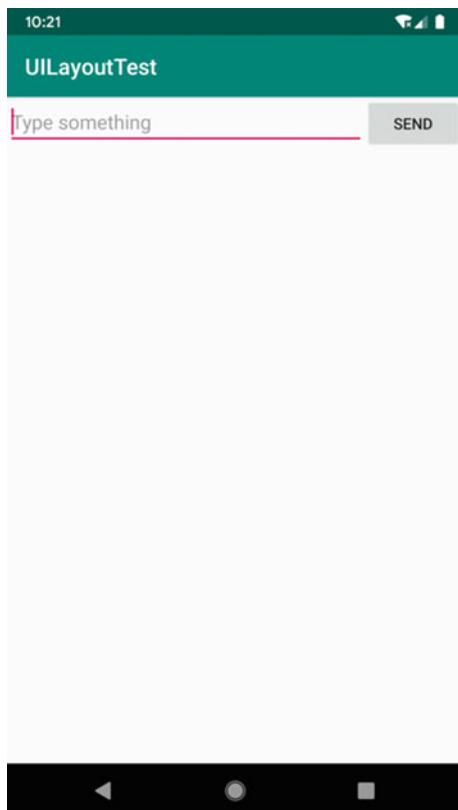
### 4.3.2 *RelativeLayout*

RelativeLayout is a layout that can position the view in any place by setting the relative position alignment. It has a lot of attributes, but these attributes have patterns to follow. So they are easy to understand and memorize. Let us learn through practice. Modify activity\_main.xml with the code shown below:

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

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
```

**Fig. 4.20** Use  
layout\_weight to make  
width adaptive



```
        android:layout_alignParentTop="true"
        android:text="Button 1" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_alignParentTop="true"
    android:text="Button 2" />

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="Button 3" />

<Button
    android:id="@+id/button4"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentLeft="true"
    android:text="Button 4" />

<Button
    android:id="@+id/button5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:text="Button 5" />

</RelativeLayout>
```

The codes above are self-explanatory. We create five buttons, put four of them at corners, and put one in the center of the screen. The attributes `android:layout_alignParentLeft`, `android:layout_alignParentTop`, `android:layout_alignParentRight`, `android:layout_alignParentBottom`, and `android:layout_centerInParent` are not covered previously, but their names already explain everything. Run the app again and the result should be as shown in Fig. 4.21.

The widgets above are relative to the parent view, and they can be relative to each other, too. Modify `activity_main.xml` with the code below:

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

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Button 3" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/button3"
        android:layout_toLeftOf="@+id/button3"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/button3"
```

**Fig. 4.21** Effect of  
RelativeLayout



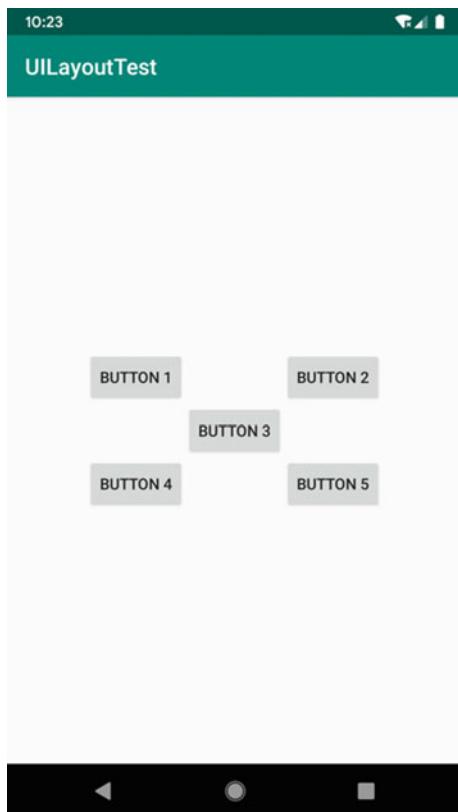
```
        android:layout_toRightOf="@+id/button3"
        android:text="Button 2" />

<Button
    android:id="@+id/button4"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/button3"
    android:layout_toLeftOf="@+id/button3"
    android:text="Button 4" />

<Button
    android:id="@+id/button5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/button3"
    android:layout_toRightOf="@+id/button3"
    android:text="Button 5" />

</RelativeLayout>
```

**Fig. 4.22** Effect of Relative to Widgets



It is more complex but still has patterns to follow. `Android:layout_above`, `android: layout_below`, `android:layout_toLeftOf`, and `android:layout_toRightOf` are still self-explanatory and can be used to specify the relative position of the views. Notice that when view A needs to refer to view B, A has to be defined after view B, otherwise it will fail to find the ID of view B. Run the app again and the result should be as shown in Fig. 4.22.

There are other `RelativeLayout` attributes to position the widgets like `android: layout_alignLeft`, `android:layout_alignRight`, `android:layout_alignTop`, and `android: layout_alignBottom`. They are all self-explanatory and they follow the same pattern. Please try out yourself.

### 4.3.3 *FrameLayout*

`FrameLayout` is a simple layout that will put every widget at the top-left corner of the layout by default. It doesn't have rich positioning attributes as `LinearLayout` nor as

RelativeLayout. Again, let us learn through some examples. Modify activity\_main.xml as the code below:

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

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

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

</FrameLayout>
```

There is a TextView and Button in the FrameLayout. Run the app again and the result should be as shown in Fig. 4.23.

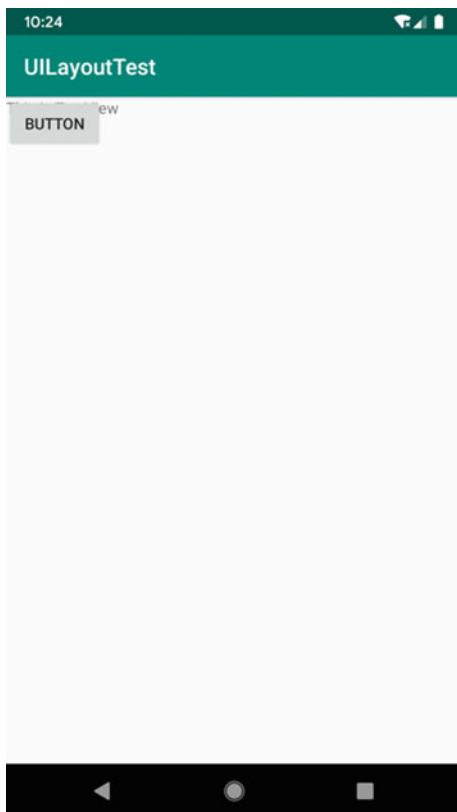
Both views are at the top-left corner, and since Button is added after TextView, it is on top of the TextView.

As in LinearLayout, we can use layout\_gravity attribute to specify the position of the widgets in the layout. Modify activity\_main.xml as the code below:

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

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

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
    />
```

**Fig. 4.23** FrameLayout

```
    android:text="Button"  
    />  
  
</FrameLayout>
```

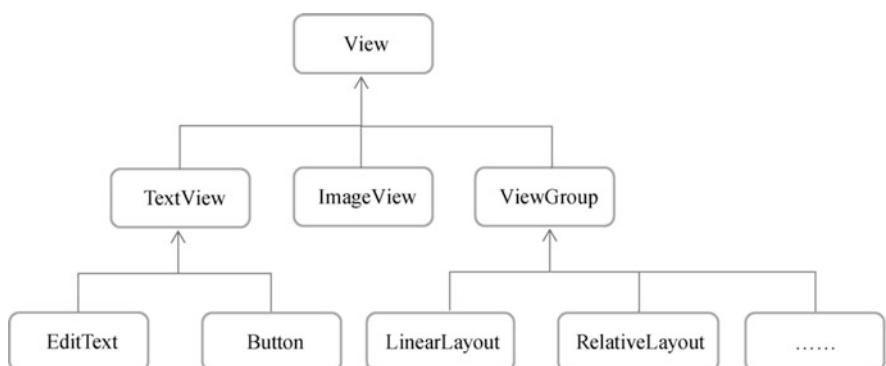
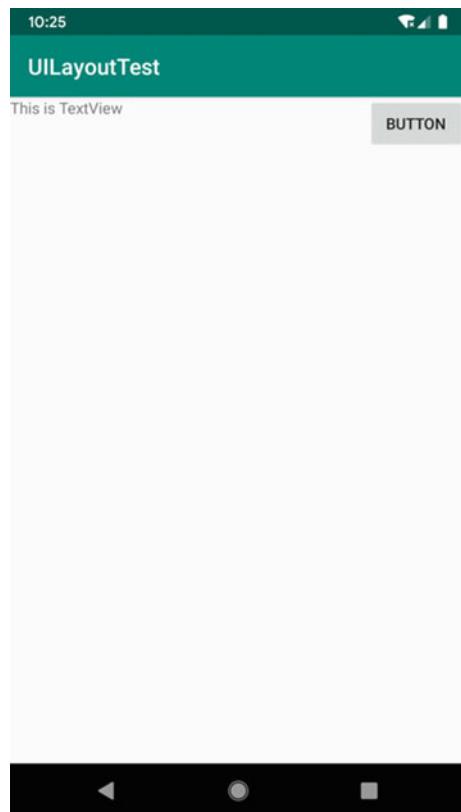
TextView is set to align to the left, and ImageView is set to align to the right. Run the app again and the result should be as shown in Fig. 4.24.

Overall, because of lacking enough ways of positioning the views, FrameLayout is less frequently used. But in the next chapter about Fragment, we will use it again.

## 4.4 Customize the Widgets

We've already covered the common widgets and layouts in Android. They're actually closely related to each other. Fig. 4.25 shows the inheritance structure of them.

**Fig. 4.24** Set layout\_gravity in FrameLayout



**Fig. 4.25** Inheritance hierarchy of common views and layouts

All the widgets inherit from View directly or indirectly. All the layouts inherit from ViewGroup directly or indirectly. View is the most basic and fundamental UI component in Android. It can draw a rectangle on the screen and respond to events in

this rectangle. All the widgets are just adding their own functionality on top of View. ViewGroup is a special View that functions like a container for widgets and layouts.

Apply the logic above, we can conclude that we should be able to inherit from View and build customized views. The conclusion is correct, and to learn how to create customized widgets, let us create a UICustomViews project by following the steps mentioned before.

#### 4.4.1 *Include Layout*

In iPhone, almost every screen of the app will have a title bar (action bar or tool bar) at the top that will provide one or two buttons for backing or other actions since there is no physical back button on iPhone. Many Android apps also adopted this design pattern and put a title bar at the top. Though Android already provides ToolBar widget for this, let's create our own customized toolbar.

It should be easy for you to create a toolbar layout now. We just need to create two Buttons and one TextView and put them into a layout. However, if we have many activities, we really don't want to write the same code repeatedly. To solve this problem, we can include the layout to reuse the code. Create title.xml layout in the layout folder as the code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@drawable/title_bg">

    <Button
        android:id="@+id/titleBack"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="5dp"
        android:background="@drawable/back_bg"
        android:text="Back"
        android:textColor="#ffff" />

    <TextView
        android:id="@+id/titleText"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="1"
        android:gravity="center"
        android:text="Title Text"
        android:textColor="#ffff"
        android:textSize="24sp" />
```

```
<Button  
    android:id="@+id/titleEdit"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center"  
    android:layout_margin="5dp"  
    android:background="@drawable/edit_bg"  
    android:text="Edit"  
    android:textColor="#fff" />  
  
</LinearLayout>
```

Most of the attributes above should be familiar to you now. I'd like to cover a few new attributes here. Android:background is used to set the background of the widget which can be a color or an image. Here I use images as background and the downloading link for images can be found in the prologue. Android:layout\_margin attribute can specify the margins at the four directions. You can also use android:layout\_marginLeft or android:layout\_marginTop to set the margin at specific direction.

To use the toolbar, modify activity\_main.xml with the code below:

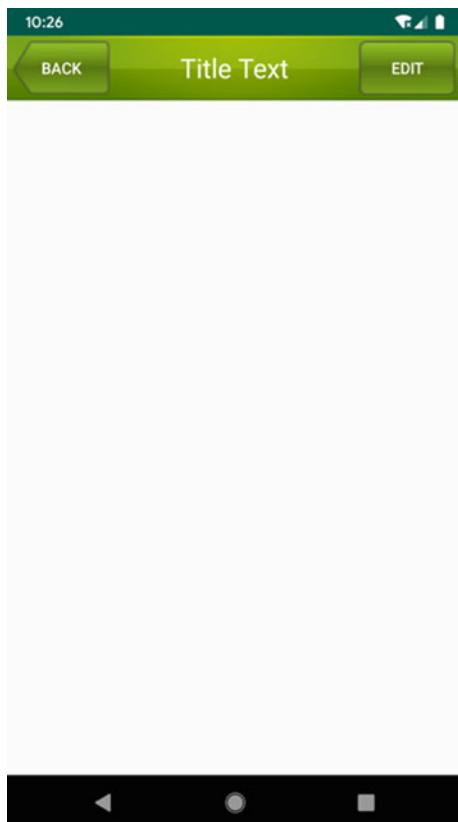
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" >  
  
    <include layout="@layout/title" />  
  
</LinearLayout>
```

Yes! We just need to use include to do so and it is just one line of code! Make sure to hide the native system toolbar from MainActivity as below:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        supportActionBar?.hide()  
    }  
  
}
```

We use getSupportActionBar() to get the instance of the ActionBar and then call hide() on it to hide itself. Since ActionBar is nullable, we need to use? . operator. I will cover more about ActionBar in Chap. 12. Run the app and the result should be as shown in Fig. 4.26.

**Fig. 4.26** Include the customized toolbar



In all the layouts that we need to add this toolbar, we just need one line of code as shown above.

#### 4.4.2 *Create Customized Widgets*

Using include can solve the problem of duplicating code. However, if the widget in the layout needs to respond to certain events, we still need to register the event handler in each of the activities. For instance, the back button in the toolbar only does one thing, which is to destroy the current activity. It is not a good idea to duplicate the event handling code in every activity. We can use the customized widget to solve this problem.

Create TitleLayout that inherits LinearLayout and we will make it our customized toolbar. The code is as follows:

```
class TitleLayout(context: Context, attrs: AttributeSet) :  
    LinearLayout(context, attrs) {  
  
    init {  
        LayoutInflater.from(context).inflate(R.layout.title, this)  
    }  
  
}
```

We declare context and attrs as the params of the primary constructor and when TitleLayout gets created, this primary constructor will be called. The from() method of LayoutInflater in the init will create an instance of LayoutInflater and its inflate() method will dynamically mount a layout. Inflate() takes two params: the first param is ID of the layout file and we pass R.layout.title to it; the second param is the parent layout of the mounted layout and we pass this to set it to the TitleLayout.

To add this customized widget, modify activity\_main.xml with the code below:

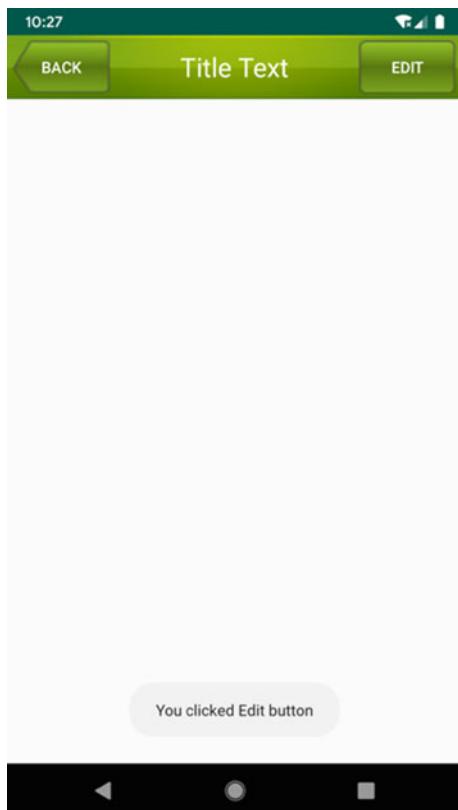
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
    android"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent" >  
  
    <com.example.uicustomviews.TitleLayout  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content" />  
  
</LinearLayout>
```

It is the same as adding a system widget except that we need to use the complete name, and the package name cannot be omitted.

Run the app again and you will see the same result as previously.

To register click event listener for the buttons in the toolbar, modify TitleLayout with the code below:

```
class TitleLayout(context: Context, attrs: AttributeSet) :  
    LinearLayout(context, attrs) {  
  
    init {  
        LayoutInflater.from(context).inflate(R.layout.title, this)  
        titleBack.setOnClickListener {  
            val activity = context as Activity  
            activity.finish()  
        }  
        titleEdit.setOnClickListener {  
            Toast.makeText(context, "You clicked Edit button",  
                Toast.LENGTH_SHORT).show()  
        }  
    }  
}
```

**Fig. 4.27** Click Edit button

The code above registers the click event listener for back and edit button. Clicking the back button will destroy the current activity. Clicking the edit button will show a toast with some messages.

Notice that the context param that `TitleLayout` takes is actually an instance of `Activity`. In the click event handler, we need to cast it to `Activity` and then call `finish()` on it to destroy the current activity. Type casting in Kotlin is done with the `keyword as`.

Run the app again and click the Edit button, the result will be as shown in Fig. 4.27. Click the back button and the current screen will disappear. This means that our customized widget is working as expected.

Now, when we use `TitleLayout` in any layout, the event listeners for back button and edit button have already been registered and there is no need to duplicate the code.

## 4.5 ListView

ListView is one of the most used UI components in Android. We can use ListView to show large amount of data in the limited screen space by allowing user to scroll. You interact with this UI component everyday like browsing chat history, checking the latest notifications, and so on.

Compared with other widgets, ListView is more complicated and this section is dedicated to cover the detailed usage of ListView.

### 4.5.1 Simple Demonstration of ListView

Create a new Android project with name ListViewTest. Use the Activity that Android Studio creates, then modify activity\_main.xml as code below:

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

    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Set an ID for the ListView and make the height and width of the ListView to be match\_parent so that it can take up the whole layout.

Modify MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    private val data = listOf("Apple", "Banana", "Orange", "Watermelon",
        "Pear", "Grape", "Pineapple", "Strawberry", "Cherry", "Mango",
        "Apple", "Banana", "Orange", "Watermelon", "Pear", "Grape",
        "Pineapple", "Strawberry", "Cherry", "Mango")

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val adapter = ArrayAdapter<String>(this, android.R.layout.
            simple_list_item_1, data)
        listView.adapter = adapter
    }
}
```

We need to prepare some data for the ListView to display and it can be downloaded from the internet or read from the database. Here we will use a data set for testing purpose which contains some fruit names. We can initialize the set with `listOf()` which we learned in Chap. 2.

However, the data in the set cannot be used directly by ListView. We need to use adapter to bridge the ListView and data. There are a few classes that implemented adapter functionality and among them `ArrayAdapter` is one of the best implementations. It uses generics to set the data type and then takes the data in the constructor. There are a few overriding methods for constructor to choose from. Here we have data of `String` class thus set the type parameter to `String`. Then we pass the instance of activity, the ID of the ListView item, and the data source. Notice that we use `android.R.layout.simple_list_item_1` as the ID of the ListView item. This is an item that Android provides which only has a `TextView` inside. Now we have an instance of the adapter.

By calling the `setAdapter()` method of the ListBiew, we can pass the adapter instance to bridge the ListView and data.

Run the app and the result is as shown in Fig. 4.28. You can scroll to see the data outside the screen.

#### 4.5.2 Customize ListView UI

ListView can show more complex UI than just some text. Let us customize the ListView UI to show more content.

First, download the images of the fruits in the set and the download link can be found in the prologue.

Then define Fruit class for the adapter of ListView as code below:

```
class Fruit(val name:String, val imageId: Int)
```

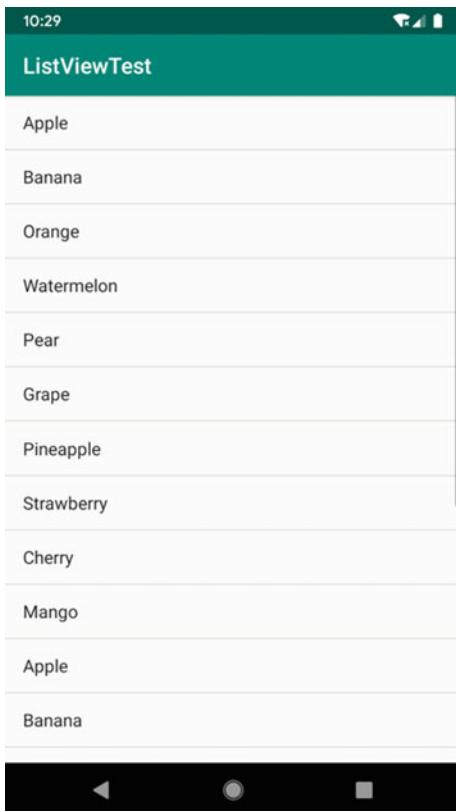
There are two fields in the Fruit class: `name` and `imageId`.

Define a customized layout for the item view in the ListView by creating `fruit_item.xml` in layout folder as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
        android:layout_width="match_parent"
        android:layout_height="60dp">

    <ImageView
        android:id="@+id/fruitImage"
        android:layout_width="40dp"
        android:layout_height="40dp"
        android:layout_gravity="center_vertical"
        android:layout_marginLeft="10dp"/>
```

**Fig. 4.28** ListView demonstration



```
<TextView  
    android:id="@+id/fruitName"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_vertical"  
    android:layout_marginLeft="10dp" />  
  
</LinearLayout>
```

In this layout, we define an ImageView to display the image of the fruit, define a TextView to display the fruit name, and position them vertically.

Next, we need to create a customized adapter which will inherit from ArrayAdapter and set the type parameter to Fruit class. Create FruitAdapter class as code below:

```
class FruitAdapter(activity: Activity, val resourceId: Int, data:  
List<Fruit>) :  
    ArrayAdapter<Fruit>(activity, resourceId, data) {
```

```

override fun getView(position: Int, convertView: View?, parent:
ViewGroup): View {
    val view = LayoutInflater.from(context).inflate(resourceId,
parent, false)
    val fruitImage: ImageView = view.findViewById(R.id.fruitImage)
    val fruitName: TextView = view.findViewById(R.id.fruitName)
    val fruit = getItem(position) // Gets the fruit instance of the
current item
    if (fruit != null) {
        fruitImage.setImageResource(fruit.imageId)
        fruitName.text = fruit.name
    }
    return view
}
}

```

The primary constructor takes the instance of activity, ID of ListView item view, and data source as arguments. It also overrides `getView()` method. This method will be called when every item view gets scrolled into the screen.

In `getView()`, `LayoutInflater` will inflate the item view with the layout that gets passed in. `inflate()` method takes 3 params and we already know the first params from previous learnings. Set the third param to false to make the layout we declare for the parent layout effective but won't add the parent layout for the view. This is because if a view has parent layout then it cannot be added to ListView. You might find it is difficult to understand the previous two sentences, and you can ignore it for now. After you understand View deeper, you will understand these two sentences.

Next, we use `findViewById()` to get the reference of `ImageView` and `TextView`; use `getItem()` to get instance of the Fruit item and then use `setImageResource()` and `setText()` to set image and text, respectively. At the end of the method, return the view.

Notice that, `kotlin-android-extensions` plugin cannot be used in the adapter for ListView, thus we still need to use `findViewById()` to get the instance of UI controls. The main use case for `kotlin-android-extension` is in activity and fragment that will be introduced in the next chapter.

Modify `MainActivity` as code below:

```

class MainActivity : AppCompatActivity() {

    private val fruitList = ArrayList<Fruit>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initFruits() // initialize fruit data
        val adapter = FruitAdapter(this, R.layout.fruit_item, fruitList)
        listView.adapter = adapter
    }
}

```

```

private fun initFruits() {
    repeat(2) {
        fruitList.add(Fruit("Apple", R.drawable.apple_pic))
        fruitList.add(Fruit("Banana", R.drawable.banana_pic))
        fruitList.add(Fruit("Orange", R.drawable.orange_pic))
        fruitList.add(Fruit("Watermelon", R.drawable.watermelon_pic))
        fruitList.add(Fruit("Pear", R.drawable.pear_pic))
        fruitList.add(Fruit("Grape", R.drawable.grape_pic))
        fruitList.add(Fruit("Pineapple", R.drawable.pineapple_pic))
        fruitList.add(Fruit("Strawberry", R.drawable.strawberry_pic))
        fruitList.add(Fruit("Cherry", R.drawable.cherry_pic))
        fruitList.add(Fruit("Mango", R.drawable.mango_pic))
    }
}
}

```

Here, the logic that initializes the fruit list has been abstracted into a private function. The repeat function is used to add the fruit data twice so that we can have enough data to fill a whole screen. This function is another commonly used standard function in Kotlin which will take a param n and then execute the Lambda expression n times. In onCreate(), the FruitAdapter get instantiated and passed to ListView. That is all we need to do for customizing the ListView UI.

Run the app again and the result should be as shown in Fig. 4.29.

This is just a simple demonstration of how to customize ListView and you can create much more complex UI by following this pattern here.

#### 4.5.3 *Optimize the Efficiency of ListView*

ListView is complicated because devil is in the detail. There are many things we can improve for ListView and running efficiency is an important aspect. The current implementation of ListView is slow because in getView() of FruitAdapter, the item view's layout gets inflated separately for every item view and this will drag the performance of the app when the ListView is fast scrolling.

You may find that there is a convertView param in getView(). This param can be used to cache the inflated view and reuse it later which can be used to optimize the performance of ListView. Modify FruitAdapter as code below:

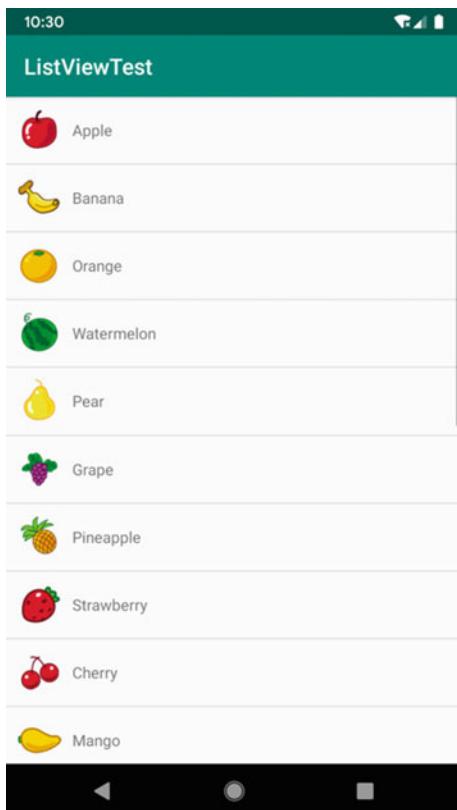
```

class FruitAdapter(activity: Activity, val resourceId: Int, data:
List<Fruit>) :
    ArrayAdapter<Fruit>(activity, resourceId, data) {

    override fun getView(position: Int, convertView: View?, parent:
ViewGroup): View {
        val view: View
        if (convertView == null) {

```

**Fig. 4.29** Customized ListView



```
        view = LayoutInflater.from(context).inflate(resourceId, parent,
false)
    } else {
        view = convertView
    }
    val fruitImage: ImageView = view.findViewById(R.id.fruitImage)
    val fruitName: TextView = view.findViewById(R.id.fruitName)
    val fruit = getItem(position) // get the current instance of Fruit
    if (fruit != null) {
        fruitImage.setImageResource(fruit.imageId)
        fruitName.text = fruit.name
    }
    return view
}
```

In `getView()`, if `convertView` is null, then use `LayoutInflater` to inflate the layout; if not, then reuse the `convertView`. By reducing the inflation action to once, fast scrolling will be faster and smoother.

However, there is still room for optimization. Every time `getView()` gets called, `findViewById()` method will get called to get reference of the view. We can use `ViewHolder` to optimize this. Modify `FruitAdapter` as code below:

```
class FruitAdapter(activity: Activity, val resourceId: Int, data: List<Fruit>) :  
    ArrayAdapter<Fruit>(activity, resourceId, data) {  
  
    inner class ViewHolder(val fruitImage: ImageView, val fruitName: TextView)  
  
    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {  
        val view: View  
        val viewHolder: ViewHolder  
        if (convertView == null) {  
            view = LayoutInflater.from(context).inflate(resourceId, parent, false)  
            val fruitImage: ImageView = view.findViewById(R.id.fruitImage)  
            val fruitName: TextView = view.findViewById(R.id.fruitName)  
            viewHolder = ViewHolder(fruitImage, fruitName)  
            view.tag = viewHolder  
        } else {  
            view = convertView  
            viewHolder = view.tag as ViewHolder  
        }  
  
        val fruit = getItem(position) // get the current instance of Fruit  
        if (fruit != null) {  
            viewHolder.fruitImage.setImageResource(fruit.imageId)  
            viewHolder.fruitName.text = fruit.name  
        }  
        return view  
    }  
}
```

The inner class `ViewHolder` is used to cache the `ImageView` and `TextView` instance. When `convertView` is null, a `ViewHolder` instance will be created to store the `ImageView` instance and `TextView` instance. The `setTag()` method of `View` will set the tag to be the instance of `ViewHolder`. When `convertView` is not null, `getTag()` will get the instance of `ViewHolder`. This instance has the instances of `TextView` and `ImageView`. Thus there is no need to use `findViewById()` again.

After these two steps of optimization, this customized `ListView` will perform much more efficiently.

#### 4.5.4 Click Event in ListView

If we can only see the ListView but cannot interact with the items in it, ListView would be less useful. Now let us start to learn how to handle click event in ListView.

Modify MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    private val fruitList = ArrayList<Fruit>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initFruits() // Initialize fruit data
        val adapter = FruitAdapter(this, R.layout.fruit_item, fruitList)
        listView.adapter = adapter
        listView.setOnItemClickListener { parent, view, position, id ->
            val fruit = fruitList[position]
            Toast.makeText(this, fruit.name, Toast.LENGTH_SHORT).show()
        }
    }

    ...
}
```

The `setOnItemClickListener()` method will register a listener for ListView which will execute the Lambda expression when user click any item in the ListView. The `position` param indicates which item user is clicked and can be used to get the corresponding instance of the item. A toast message will show the fruit name after clicking event.

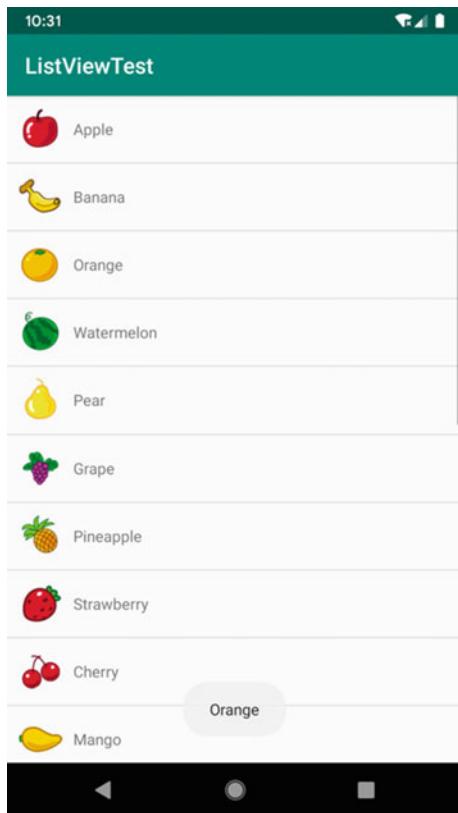
Run the app again and click the orange item. The result should show as in Fig. 4.30.

The Lambda expression above declared four params. How do we know what params we need to declare? There is shortcut for it: click `setOnItemClickListener()` while pressing Ctrl(command for Mac) to see the source code and you will find that `setOnItemClickListener()` takes a param of `OnItemClickListener` which is a Java functional interface which is defined as shown in Fig. 4.31.

The four params that `onItemClick()` takes are the params that should be defined in the Lambda expression.

Although the four params must be declared in Lambda expression, only `position` has been used. In Kotlin, underline can be used to replace the unused params as shown below:

```
listView.setOnItemClickListener { _, _, position, _ ->
    val fruit = fruitList[position]
```

**Fig. 4.30** Click ListView

```
/**  
 * Interface definition for a callback to be invoked when an item in this  
 * AdapterView has been clicked.  
 */  
public interface OnItemClickListener {  
  
    /**  
     * Callback method to be invoked when an item in this AdapterView has  
     * been clicked.  
     * <p>  
     * Implementers can call getItemAtPosition(position) if they need  
     * to access the data associated with the selected item.  
     *  
     * @param parent The AdapterView where the click happened.  
     * @param view The view within the AdapterView that was clicked (this  
     *            will be a view provided by the adapter)  
     * @param position The position of the view in the adapter.  
     * @param id The row id of the item that was clicked.  
     */  
    void onItemClick(AdapterView<?> parent, View view, int position, long id);  
}
```

**Fig. 4.31** Definition of OnItemClickListener interface

```
    Toast.makeText(this, fruit.name, Toast.LENGTH_SHORT).show()
}
```

The above pattern is legit and the recommended pattern for unused params. Notice that, the order of the params cannot be changed and thus position param is still the third param in the params list.

## 4.6 RecyclerView

ListView was widely used in the past and even today there are still numerous apps that are using it. However, developers need to know the techniques to optimize the performance of ListView as you have seen and its extensibility is also poor. For instance, ListView can only handle scrolling vertically and there is no way to scroll horizontally for ListView.

RecyclerView is a powerful widget that provides more functionality than the ListView and at the same time, easy to use and very performant. It is the officially recommended widget to replace ListView. More and more apps are migrating from ListView to RecyclerView.

Before we start to learn RecyclerView, create a project named RecyclerViewTest and use Android Studio to generate the activity.

### 4.6.1 Basics About RecyclerView

Unlike the widgets we learned so far, RecyclerView is a relatively new widget. To make it available in all Android versions, Google defined RecyclerView widget in AndroidX. By adding the dependency to RecyclerView in build.gradle, all versions of Android OS can use RecyclerView.

Open app/build.gradle file and add code below in dependencies closure:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$
kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:
constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso-e
core:3.1.1'
}
```



Fig. 4.32 Android Studio notice for new version of Lib

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. [Sync Now](#)

Fig. 4.33 Notice after changing the gradle File

You may need to update the version number when you try the code above. Android Studio will notify you what is the latest version as shown in Fig. 4.32.

After updating any of the gradle file, Android Studio will have a pop as shown in Fig. 4.33.

Click “Sync Now” to allow gradle importing the RecyclerView lib to the current project.

Update activity\_main.xml as code below:

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

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

To add RecyclerView in the layout is the same as adding other widgets except the name has to be the complete name with package path. This is because RecyclerView is not in the system SDK.

We can reuse the images, Fruit class, and fruit\_item.xml from ListViewTest to reduce duplicated work.

Create FruitAdapter class that inherits RecyclerView.Adapter and set the param type to be FruitAdapter.ViewHolder. ViewHolder is an inner class defined in FruitAdapter as shown in code below:

```
class FruitAdapter(val fruitList: List<Fruit>) :
    RecyclerView.Adapter<FruitAdapter.ViewHolder>() {

    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view)
    {
        val fruitImage: ImageView = view.findViewById(R.id.fruitImage)
```

```

    val fruitName: TextView = view.findViewById(R.id.fruitName)
}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val view = LayoutInflater.from(parent.context)
        .inflate(R.layout.fruit_item, parent, false)
    return ViewHolder(view)
}

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val fruit = fruitList[position]
    holder.fruitImage.setImageResource(fruit.imageId)
    holder.fruitName.text = fruit.name
}

override fun getItemCount() = fruitList.size
}

```

This is the standard way to create adapter for RecyclerView which has more methods to implement compared with ListView's adapter. However, it should be easier to understand. ViewHolder is an inner class that inherits from RecyclerView.ViewHolder. It takes a View param in the primary constructor which usually is outer layout of the item view in RecyclerView. The instances of ImageView and TextView can be acquired by findViewById().

The data source will be passed in the primary constructor of FruitAdapter.

FruitAdapter inherits RecyclerView.Adapter, thus onCreateViewHolder(), onBindViewHolder() and getItemCount() methods need to be overrided. onCreateViewHolder() is used to create instance of ViewHolder. fruit\_item layout will be inflated then ViewHolder will be instantiated with the inflated view. onBindViewHolder() will set value for the item which will be executed when the item has been scrolled into the screen. We can use position to get the instance of the current item then use the item to set image and text for ImageView and TextView in ViewHolder. getItemCount() is self-explanatory and should return the size of the data set.

Now the RecyclerView is ready to use. Change MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    private val fruitList = ArrayList<Fruit>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initFruits() // initialize fruits data
        val layoutManager = LinearLayoutManager(this)
        recyclerView.layoutManager = layoutManager
    }
}

```

```
    val adapter = FruitAdapter(fruitList)
    recyclerView.adapter = adapter
}

private fun initFruits() {
    repeat(2) {
        fruitList.add(Fruit("Apple", R.drawable.apple_pic))
        fruitList.add(Fruit("Banana", R.drawable.banana_pic))
        fruitList.add(Fruit("Orange", R.drawable.orange_pic))
        fruitList.add(Fruit("Watermelon", R.drawable.watermelon_pic))
        fruitList.add(Fruit("Pear", R.drawable.pear_pic))
        fruitList.add(Fruit("Grape", R.drawable.grape_pic))
        fruitList.add(Fruit("Pineapple", R.drawable.pineapple_pic))
        fruitList.add(Fruit("Strawberry", R.drawable.strawberry_pic))
        fruitList.add(Fruit("Cherry", R.drawable.cherry_pic))
        fruitList.add(Fruit("Mango", R.drawable.mango_pic))
    }
}
```

}

initFruit() is reused here to initialize the data for fruits. An instance of LinearLayoutManager gets created in onCreate() and gets used to specify the RecyclerView's layoutManager. LinearLayoutManager will position the items like ListView. Next we instantiate the FruitAdapter with the fruits data and assign this adapter to the RecyclerView instance to bridge RecyclerView and data.

Run the app and the result should be as shown in Fig. 4.34.

The RecyclerView implemented the same UI as ListView with similar amount of code. However, the logic of RecyclerView should be clearer compared with ListView. Now let us see what RecyclerView offers that ListView doesn't.

#### 4.6.2 Scroll Horizontally and Waterfall Flow Layout

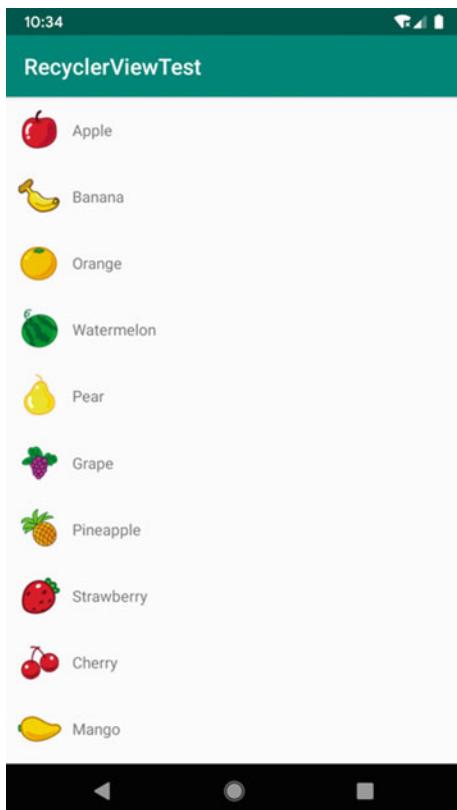
RecyclerView supports scrolling horizontally which is beyond the capability of ListView. It is also very easy to implement with RecyclerView.

First, update the fruit\_item layout to make it fit for horizontal scrolling. Change fruit\_item.xml as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="80dp"
    android:layout_height="wrap_content">

    <ImageView
        android:id="@+id/fruitImage"
        android:layout_width="40dp"
        android:layout_height="40dp" />

```

**Fig. 4.34** RecyclerView

```
    android:layout_height="40dp"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="10dp" />

<ImageView
    android:id="@+id/fruitIcon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="10dp" />

<TextView
    android:id="@+id/fruitName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="10dp" />

</LinearLayout>
```

Make orientation of the LinearLayout to be vertical and set its width to be 80dp. We use fixed value for width because wrap\_content will make the item view with different width as the length of fruits' names vary; match\_parent will make one item view's width same as the whole screen's width.

Set the layout\_gravity of the ImageView and TextView to be centered horizontally and use layout\_marginTop to give some room between the text and image.

Change MainActivity as code below:

```
class MainActivity : AppCompatActivity() {  
  
    private val fruitList = ArrayList<Fruit>()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        initFruits() // Initialize fruit data  
        val layoutManager = LinearLayoutManager(this)  
        layoutManager.orientation = LinearLayoutManager.HORIZONTAL  
        recyclerView.layoutManager = layoutManager  
        val adapter = FruitAdapter(fruitList)  
        recyclerView.adapter = adapter  
    }  
    ...  
}
```

The only change here is to make the orientation of the layoutManager to be horizontal instead of vertical so that RecyclerView can be scrolled horizontally.

Run the app again and the result should be as shown in Fig. 4.35.

Now you can scroll horizontally to see the data outside the screen.

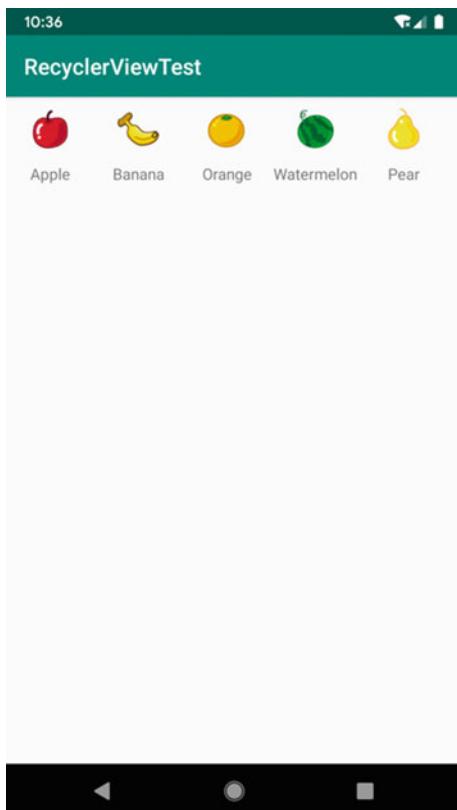
ListView manages the layout internally however RecyclerView delegates this to LayoutManager which has a suite of interfaces for layout. The sub classes of LayoutManager just need to implement the well-defined interfaces to support different styles of layouts.

Besides LinearLayoutManager, LayoutManager has two other implementations supported natively: GridLayoutManager and StaggeredGridLayoutManager. Let us experiment with the fancier StaggeredGridLayoutManager to show waterfall flow layout. You can try GridLayoutManager by yourself.

Change fruit\_item.xml as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
    android"  
        android:orientation="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_margin="5dp">  
  
    <ImageView  
        android:id="@+id/fruitImage"  
        android:layout_width="40dp"  
        android:layout_height="40dp"  
        android:layout_gravity="center_horizontal"  
        android:layout_marginTop="10dp" />  
  
    <TextView  
        android:id="@+id/fruitName"  
        ...>
```

**Fig. 4.35** Horizontal RecyclerView



```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left"
        android:layout_marginTop="10dp" />

    </LinearLayout>
```

Change the width of the LinearLayout from 80 dp to match\_parent since the width of the item view should be determined by the column number of the layout instead of a fixed value. Use layout\_margin to give some space between the item views. Change the layout\_gravity of TextView to left because we will make the longer text later which makes left alignment more sensible.

Change MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    private val fruitList = ArrayList<Fruit>()
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    initFruits() // Initialize fruit data
    val layoutManager = StaggeredGridLayoutManager(3,
        StaggeredGridLayoutManager.VERTICAL)
    recyclerView.layoutManager = layoutManager
    val adapter = FruitAdapter(fruitList)
    recyclerView.adapter = adapter
}

private fun initFruits() {
    repeat(2) {
        fruitList.add(Fruit(getRandomLengthString("Apple"),
            R.drawable.apple_pic))
        fruitList.add(Fruit(getRandomLengthString("Banana"),
            R.drawable.banana_pic))
        fruitList.add(Fruit(getRandomLengthString("Orange"),
            R.drawable.orange_pic))
        fruitList.add(Fruit(getRandomLengthString("Watermelon"),
            R.drawable.watermelon_pic))
        fruitList.add(Fruit(getRandomLengthString("Pear"),
            R.drawable.pear_pic))
        fruitList.add(Fruit(getRandomLengthString("Grape"),
            R.drawable.grape_pic))
        fruitList.add(Fruit(getRandomLengthString("Pineapple"),
            R.drawable.pineapple_pic))
        fruitList.add(Fruit(getRandomLengthString("Strawberry"),
            R.drawable.strawberry_pic))
        fruitList.add(Fruit(getRandomLengthString("Cherry"),
            R.drawable.cherry_pic))
        fruitList.add(Fruit(getRandomLengthString("Mango"),
            R.drawable.mango_pic))
    }
}

private fun getRandomLengthString(str: String): String {
    val n = (1..20).random()
    val builder = StringBuilder()
    repeat(n) {
        builder.append(str)
    }
    return builder.toString()
}
```

The way to use StaggeredGridLayoutManager doesn't change much from using LinearLayoutManager. StaggeredGridLayoutManager constructor takes two params: the first param is the columns of the layout and there are three columns in this example; the second param will specify the orientation of the layout which will be vertical for this example.

**Fig. 4.36** Waterfall flow layout



We just need to change one line of code to make the style looks like waterfall flow. The visual effect will be more prominent if the heights of item views are different. To make the height different for the items views, we can use random() to generate a random value for repeating the text different times.

Run the app again and the result should look like Fig. 4.36.

As the repeating times are random, every time you run the app, the visual effect will change.

### 4.6.3 RecyclerView Click Event

Unlike ListView, RecyclerView doesn't provide a way as setOnItemClickListener() to register listener for click event. We need to register the click event for the specific view component in the item view. ListView can also register click event for the specific view component in the item view but is quite difficult to implement. RecyclerView was designed to register click event listeners for specific views.

To register click event in RecyclerView, we can use the following code as an example. Change FruitAdapter as code below:

```
class FruitAdapter(val fruitList: List<Fruit>) :  
    RecyclerView.Adapter<FruitAdapter.ViewHolder>() {  
    ...  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
        ViewHolder {  
        val view = LayoutInflater.from(parent.context)  
            .inflate(R.layout.fruit_item, parent, false)  
        val viewHolder = ViewHolder(view)  
        viewHolder.itemView.setOnClickListener {  
            val position = viewHolder.adapterPosition  
            val fruit = fruitList[position]  
            Toast.makeText(parent.context, "you clicked view ${fruit.  
name}",  
                Toast.LENGTH_SHORT).show()  
        }  
        viewHolder.fruitImage.setOnClickListener {  
            val position = viewHolder.adapterPosition  
            val fruit = fruitList[position]  
            Toast.makeText(parent.context, "you clicked image ${fruit.  
name}",  
                Toast.LENGTH_SHORT).show()  
        }  
        return viewHolder  
    }  
    ...  
}
```

The registration happens in onCreateViewHolder(). The above code registers the click event listener for the outer layout and also ImageView. We can use the position to get the instance of the Fruit and then show different contents for the different click event.

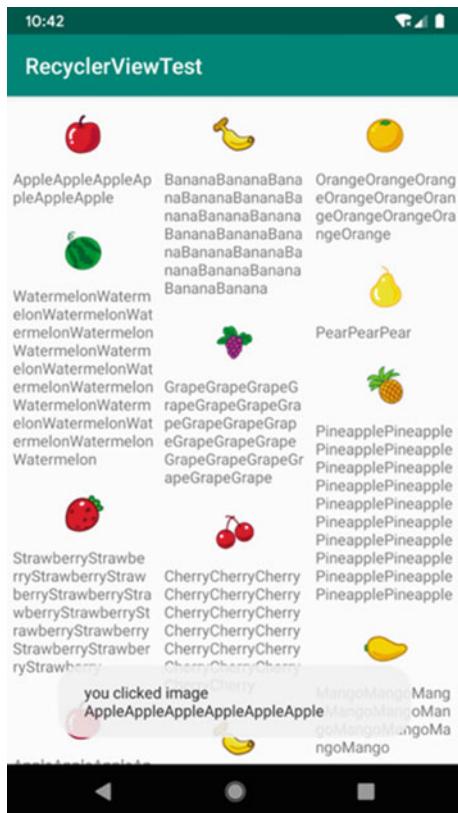
Run the app again and clicking the apple image should show toast as shown in Fig. 4.37. As you can see, the ImageView click event gets triggered.

Then click the orange item's text. Since TextView doesn't register click event listener, the click event will be captured by the outer layout. You should see toast as shown in Fig. 4.38.

## 4.7 Best Practice to Build UI

After learning so many topics about UI development, it is the time to apply what we have learned and build something more complex and interesting. Let us create a UIBestPractice project to start with.

**Fig. 4.37** Click Apple image



#### 4.7.1 Create 9-Patch Image

Before starting to code, we need to create 9-patch image which may sounds new to you. But essentially it is processed png file that can specify which areas can be stretched.

Let us use an example to demonstrate why 9-patch images are needed. First, import message\_left.png (download link is in the prologue) in to UIBestPractice. The image is as shown in Fig. 4.39.

Set this image to be the background image of LinearLayout by changing activity\_main.xml as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:background="@drawable/message_left">
</LinearLayout>
```

**Fig. 4.38** Click Orange text**Fig. 4.39** Bubble-like image

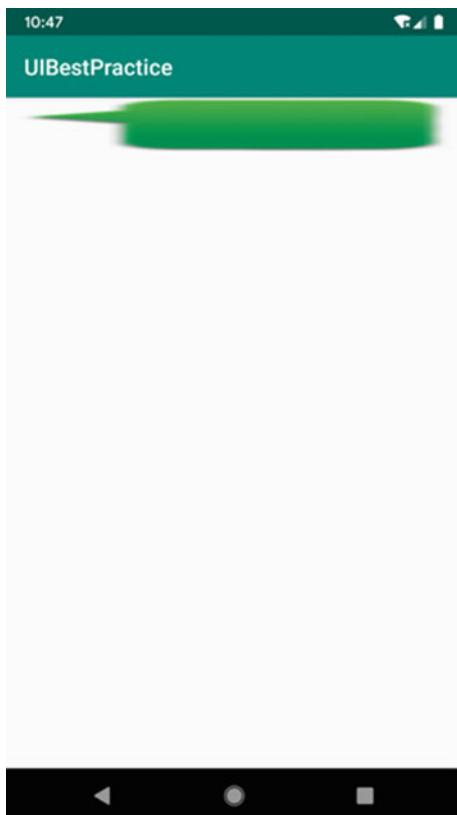
This LinearLayout's width will be the width of the screen since its width is set to match\_parent. The result is as shown in Fig. 4.40.

Since message\_left image's width is not enough to fill the whole screen, it gets evenly stretched horizontally. Apparently, this looks horrible and users cannot tolerate such UX. 9-patch images can mitigate this problem.

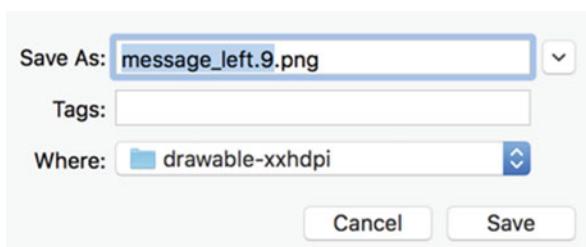
We can use Android Studio to turn png images to 9-patch images. Right click message\_left.png ->Create 9-patch file and a dialog like Fig. 4.41 will show up.

Keep the file name as it is and click "Save." Then the editor will show up as shown in Fig. 4.42.

**Fig. 4.40** Bubble gets evenly stretched horizontally



**Fig. 4.41** Dialog to create 9-patch image



Use the mouse cursor and drag along the top and left border of the image to highlight the area that can stretch; drag along the bottom and right border to highlight the area that content will be positioned. To erase, press Shift while dragging. You should follow the method here to achieve something as shown in Fig. 4.43.

Make sure to remove the original `message_left.png` and only keep the newly created `message_left.9.png` file. This is because Android project doesn't allow two images to have the same name in the same folder even though they have different file suffix. Run the app again and the result should be as shown in Fig. 4.44.

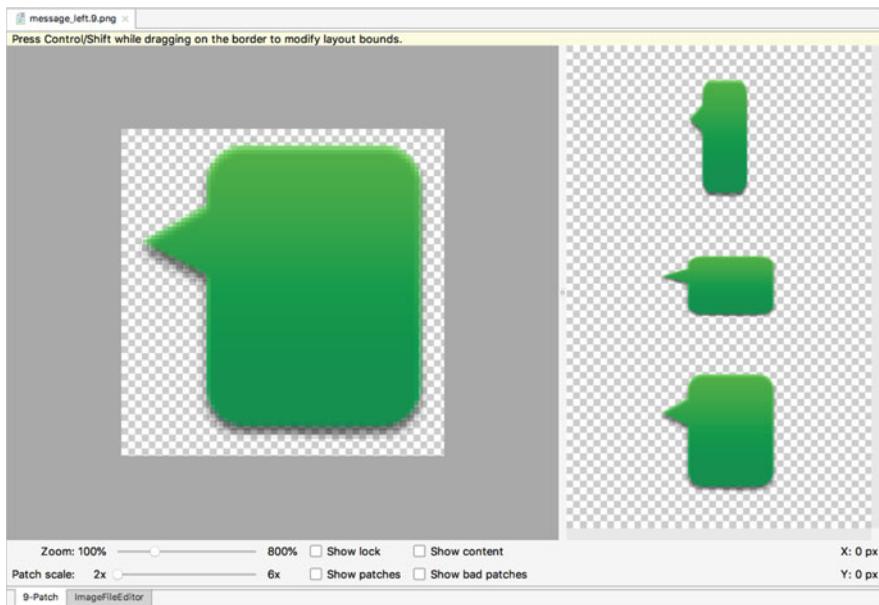


Fig. 4.42 Editor for 9-patch image

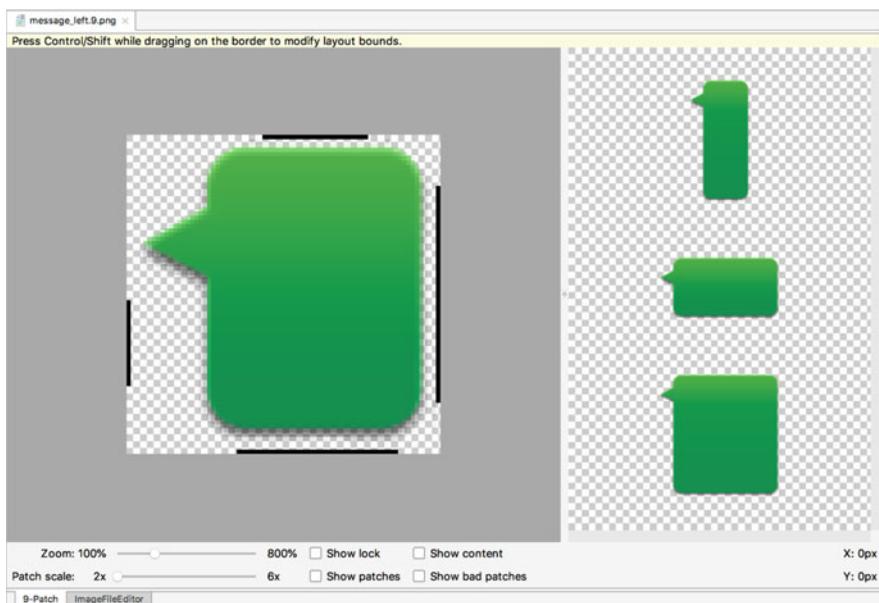
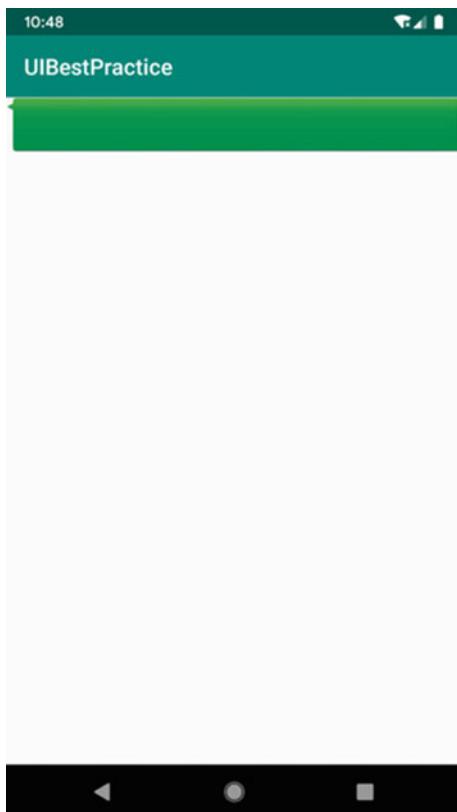


Fig. 4.43 message\_left.png after editing

**Fig. 4.44** Bubble Stretch Effect



When the 9-patch image needs to stretch, it will only stretch the designated area which provides much better user experience.

#### 4.7.2 Build Beautiful Chat User Interface

There are received messages and sent messages in chat UI, and we can use message\_left.9.png to be the background of the received message. You can follow the same process to make message\_right.9.png to be the background of the sent message.

Add dependency as shown below in app/build.gradle so that we can use RecyclerView in our project.

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$  
kotlin_version"
```

```
implementation 'androidx.appcompat:appcompat:1.0.2'  
implementation 'androidx.core:core-ktx:1.0.2'  
implementation 'androidx.constraintlayout:  
constraintlayout:1.1.3'  
implementation 'androidx.recyclerview:recyclerview:1.0.0'  
testImplementation 'junit:junit:4.12'  
androidTestImplementation 'androidx.test:runner:1.1.1'  
androidTestImplementation 'androidx.test.espresso:espresso-  
core:3.1.1'  
}
```

To build the main UI, change activity\_main.xml as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
android"  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:background="#d8e0e8" >  
  
<androidx.recyclerview.widget.RecyclerView  
    android:id="@+id/recyclerView"  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    android:layout_weight="1" />  
  
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" >  
  
    <EditText  
        android:id="@+id/inputText"  
        android:layout_width="0dp"  
        android:layout_height="wrap_content"  
        android:layout_weight="1"  
        android:hint="Type something here"  
        android:maxLines="2" />  
  
    <Button  
        android:id="@+id/send"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Send" />  
  
    </LinearLayout>  
  
</LinearLayout>
```

The RecyclerView is used to display the chat messages. EditText is used to input the message and the Button is used to send the message. All of these have been covered before, so I won't explain further.

Create Msg class to represent message as code below:

```
class Msg(val content: String, val type: Int) {
    companion object {
        const val TYPE_RECEIVED = 0
        const val TYPE_SENT = 1
    }
}
```

There are only two fields in Msg class to store message content and message type. Message type has two values: TYPE\_RECEIVED and TYPE\_SENT. Both are self-explanatory and are defined as constant values. Notice that const can only be used in singleton, companion object, or top-level method.

Create msg\_left\_item.xml for item view as code below:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="10dp" >

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="left"
        android:background="@drawable/message_left" >

        <TextView
            android:id="@+id/leftMsg"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:layout_margin="10dp"
            android:textColor="#fff" />

    </LinearLayout>

</FrameLayout>
```

This is the layout for received message with the message text left-aligned and uses message\_left.9.png as background image.

Similarly, create msg\_right\_item.xml for sent message as code below:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="10dp" >
```

```
<LinearLayout  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="right"  
    android:background="@drawable/message_right" >  
  
    <TextView  
        android:id="@+id/rightMsg"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_gravity="center"  
        android:layout_margin="10dp"  
        android:textColor="#000" />  
  
  </LinearLayout>  
  
</FrameLayout>
```

This layout has the message text right-aligned and uses message\_right.9.png as background image.

Create MsgAdapter for the RecyclerView as code below:

```
class MsgAdapter(val msgList: List<Msg>) : RecyclerView.  
Adapter<RecyclerView.ViewHolder>() {  
  
    inner class LeftViewHolder(view: View) : RecyclerView.ViewHolder  
(view) {  
        val leftMsg: TextView = view.findViewById(R.id.leftMsg)  
    }  
  
    inner class RightViewHolder(view: View) : RecyclerView.ViewHolder  
(view) {  
        val rightMsg: TextView = view.findViewById(R.id.rightMsg)  
    }  
  
    override fun getItemViewType(position: Int): Int {  
        val msg = msgList[position]  
        return msg.type  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =  
    if (viewType ==  
        Msg.TYPE_RECEIVED) {  
            val view = LayoutInflater.from(parent.context).inflate(R.layout.  
msg_left_item,  
                parent, false)  
            LeftViewHolder(view)  
        } else {  
            val view = LayoutInflater.from(parent.context).inflate(R.layout.  
msg_right_item,  
                parent, false)
```

```

        RightViewHolder(view)
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
position: Int) {
    val msg = msgList[position]
    when (holder) {
        is LeftViewHolder -> holder.leftMsg.text = msg.content
        is RightViewHolder -> holder.rightMsg.text = msg.content
        else -> throw IllegalArgumentException()
    }
}

override fun getItemCount() = msgList.size
}

```

Compared with the previous RecyclerView example, code above overrides getItemViewType to return the item's type at current position, and has two ViewHolders to cache msg\_left\_item.xml and msg\_right\_item.xml components, respectively. The viewType is used to determine which layout (viewHolder) to use in onBindViewHolder().

Change MainActivity to initialize data for RecyclerView and register click event for send button as code below:

```

class MainActivity : AppCompatActivity(), View.OnClickListener {

    private val msgList = ArrayList<Msg>()

    private var adapter: MsgAdapter? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initMsg()
        val layoutManager = LinearLayoutManager(this)
        recyclerView.layoutManager = layoutManager
        adapter = MsgAdapter(msgList)
        recyclerView.adapter = adapter
        send.setOnClickListener(this)
    }

    override fun onClick(v: View?) {
        when (v) {
            send -> {
                val content = inputText.text.toString()
                if (content.isNotEmpty()) {
                    val msg = Msg(content, Msg.TYPE_SENT)
                    msgList.add(msg)
                    adapter?.notifyItemInserted(msgList.size - 1) // when there is
                }
            }
        }
    }
}

```

```
new message, refresh RecyclerView
    recyclerView.scrollToPosition(msgList.size - 1) // scroll
    RecyclerView to the last item
        inputText.setText("") // clear the EditText view
    }
}
}

private fun initMsg() {
    val msg1 = Msg("Hello guy.", Msg.TYPE_RECEIVED)
    msgList.add(msg1)
    val msg2 = Msg("Hello. Who is that?", Msg.TYPE_SENT)
    msgList.add(msg2)
    val msg3 = Msg("This is Tom. Nice talking to you. ", Msg.
    TYPE_RECEIVED)
    msgList.add(msg3)
}

}
```

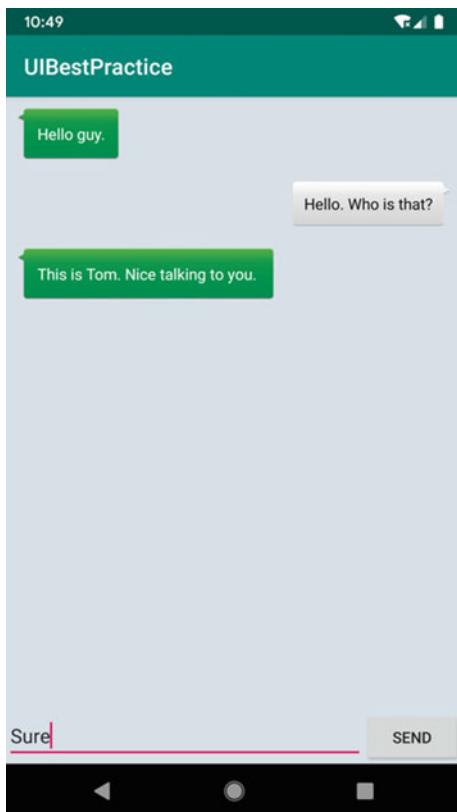
We first initialize data in `initMsg()` and the use the previous pattern to configure the `RecyclerView` with `LayoutManager` and `adapter`.

The send button event listener gets the content of `EditText` view and if the content is not empty string, a new instance of `Msg` will be created and added to `msgList`. The `notifyItemInserted()` method will notify the change and refresh the `RecyclerView` to show the latest message. You can also use the `notifyDataSetChanged()` method of the adapter which will refresh all the visible elements in the `ReyclerView`. Thus whether it is insertion, deletion, or updating, UI will show the latest data. However, this is relatively slow as you can imagine. The `scrollToPosition()` method will scroll the `ReyclerView` to the last item. This is to make sure the latest message will show up. The `setText()` method will clear the `EditText` view.

That is everything we need to do. Run the app and you should be able to type in the message and “send” the message as shown in Fig. 4.45.

## 4.8 Kotlin Class: Lateinit and Sealed Cass

As I mentioned before, I will cover some Kotlin knowledge at the end of each chapter. In this section, I will cover `lateinit` and `sealed class`.

**Fig. 4.45** Chat UI

#### 4.8.1 Lateinit Variables

Previous chapters cover some Kotlin safety features such as immutable val, nonnull variable, and so on. These language features can help build safe program; however, they can also be inconvenient from time to time.

For example, if there are lots of global variables, to successfully compile, you need to write lots of condition checks even you can make sure that these variables are not nullable.

Use UIBestPractice as an example. The adapter related code has different pattern from previous examples:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {  
  
    private var adapter: MsgAdapter? = null  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        adapter = MsgAdapter(msgList)
```

```

    ...
}

override fun onClick(v: View?) {
    ...
    adapter?.notifyItemInserted(msgList.size - 1)
    ...
}
}

```

The adapter is declared as global variable but gets initialized in onCreate(). Thus adapter was initialized with value of null and its type is `MsgAdapter?`.

We initialize the adapter in onCreate() and can make sure that onClick() will be called after onCreate(). However compiler will fail to compile if we don't add null check in onClick().

As you add more and more global variables, this problem will become more and more serious. You probably will need to write lots of null check code just to make the code can compile.

One of the solutions is using `lateinit`. This keyword tells Kotlin compiler that the variable declared with it will be initialized later and there is no need to assign null to the variable when gets declared.

We can optimize the above code as code below:

```

class MainActivity : AppCompatActivity(), View.OnClickListener {

    private lateinit var adapter: MsgAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        adapter = MsgAdapter(msgList)
        ...
    }

    override fun onClick(v: View?) {
        ...
        adapter.notifyItemInserted(msgList.size - 1)
        ...
    }
}

```

With the `lateinit` keyword, there is no need to assign null to the variable and the variable type becomes `MsgAdapter` instead of `MsgAdapter?`. This eliminates the need to add nullability check in onClick().

However, if adapter gets referenced before initialized, then program will crash and throw `UninitializedPropertyAccessException` as show in Fig. 4.46.

```
Caused by: kotlin.UninitializedPropertyAccessException: lateinit property adapter has not been initialized
    at com.example.ubestpractice.MainActivity.onCreate(MainActivity.kt:22)
    at android.app.Activity.performCreate(Activity.java:7136)
    at android.app.Activity.performCreate(Activity.java:7122)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1271)
```

**Fig. 4.46** UninitializedPropertyAccessException

So, make sure that the lateinit variables are initialized before using them to ensure null safety of the program.

We can check if a global variable has been initialized or not to avoid the exception above and unnecessary initialization. To do so, follow the code below:

```
class MainActivity : AppCompatActivity(), View.OnClickListener {

    private lateinit var adapter: MsgAdapter

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        if (!::adapter.isInitialized) {
            adapter = MsgAdapter(msgList)
        }
        ...
    }

}
```

The syntax here may look strange. ::adapter. isInitialized determines if adapter has been initialized. If the negation of it is true then adapter hasn't been initialized yet and we need to initialize adapter with proper value.

#### 4.8.2 Optimization with Sealed Class

Sealed class can be used together with ViewHolder in RecyclerView's adapter. Thus I will cover it in this section. Of course, it has many other use cases and can help you write consolidated and safe code.

Let us use an example to learn sealed class. Create Result.kt file and write the following code:

```
interface Result
class Success(val msg: String) : Result
class Failure(val error: Exception) : Result
```

We define a Result interface that represent the result of certain operation and there is no content in the interface. Then we define two classes to implement Result interface: Success class and Failure.

Next define getResultMsg() to get the message of the result as code below:

```
fun getResultMsg(result: Result) = when (result) {
    is Success -> result.msg
    is Failure -> result.error.message
    else -> throw IllegalArgumentException()
}
```

`getResultMsg()` method takes a param of `Result` type. If `Result` is `Success` then, return the message of `result`; if `Result` is `Failure`, then return the error message. Notice that we also need to add an `else` statement to inform Kotlin compiler that we've exhausted all the conditions to prevent compilation failure. Although this statement should never get used since `Result` can only be `Success` or `Failure`.

There is a hidden risk for the `else` statement here. If we add a new class `Unknown` that implements `Result` interface but forget to add the corresponding condition check in `getResultMsg()`, then compiler won't throw any error. When the `Unknown` is passed in, `else` statement will get called and throw exception to crash the app.

Of course, this kind of redundant condition handling is a problem not limited to Kotlin but also exists in other programming languages like Java.

The sealed class in Kotlin can solve this problem. ~~and let us see how to use it.~~

It's very easy to use sealed class and we can change `Result` interface to a sealed class as code below:

```
sealed class Result
class Success(val msg: String) : Result()
class Failure(val error: Exception) : Result()
```

The changes here are using sealed class to replace interface to declare `Result`; adding brackets after `Result` because it is inheriting instead of implementing interface now.

With this change, there is no need to add `else` statement in `getResultMsg()` as shown in code below:

```
fun getResultMsg(result: Result) = when (result) {
    is Success -> result.msg
    is Failure -> "Error is ${result.error.message}"
}
```

When the `when` statement gets a sealed class as argument, Kotlin compiler will check what are the sub classes that inherits this class then check if all the cases are covered which can eliminate the `else` statement. If we add `Unknown` class that inherits `Result` without adding the corresponding condition check in `getResultMsg()`, compiler will fail to compile.

One last thing about sealed class that is worth noting is that sealed class and its sub classes can only be defined at the top level of the same file and cannot be embedded in other classes which is a limitation of the sealed class's implementation.

Now let us take a look at how to use sealed class with `ViewHolder` and optimize `MsgAdapter`.

The current onBindViewHolder() has a redundant else statement that just throw an exception. We can use sealed class to optimize it. Create MsgViewHolder.kt file with the code below:

```
sealed class MsgViewHolder(view: View) : RecyclerView.ViewHolder(view)

class LeftViewHolder(view: View) : MsgViewHolder(view) {
    val leftMsg: TextView = view.findViewById(R.id.leftMsg)
}

class RightViewHolder(view: View) : MsgViewHolder(view) {
    val rightMsg: TextView = view.findViewById(R.id.rightMsg)
}
```

Now the sealed class MsgViewHolder only has two sub classes, thus in the when statement, only two cases need to be handled. Change MsgAdapter as code below:

```
class MsgAdapter(val msgList: List<Msg>) : RecyclerView.
Adapter<MsgViewHolder>() {

    ...

    override fun onBindViewHolder(holder: MsgViewHolder, position:
Int) {
        val msg = msgList[position]
        when (holder) {
            is LeftViewHolder -> holder.leftMsg.text = msg.content
            is RightViewHolder -> holder.rightMsg.text = msg.content
        }
    }
    ...
}
```

The param type of the generics becomes sealed class MsgViewHolder and we only need to handle LeftViewHolder and RightViewHolder, no more else statement. This is the recommended practice of creating RecyclerView adapter.

Now, the code in UIBestPractice is more consolidated and let's summarize what we've covered.

## 4.9 Summary and Comment

Although there are lots of topics in this chapter, hopefully you can enjoy the process of learning. Unlike the previous chapter in which we have to use the button again and again, this chapter covers lots of UI widgets and teaches you how to make beautiful and interactive UIs. You should feel much more fulfilled, right?

This chapter starts with introduction to some of the commonly used UI widgets in Android, followed by introduction to basic layouts, customization of UI widgets, and detailed coverage of ListView and RecyclerView. This chapter covers almost all the important UI related topics in Android. In the Kotlin topics section, you learned how to use lateinit and sealed class and applied them to optimize the code. With so many concrete examples, you should have solid understanding of these topics.

However, up to now, we only covered the techniques to build apps for Android phones. The next chapter will cover topics on Android tablet app development. Android started to support app running in tablet and phone since Android 4.0. Get some rest and let us continue the learning journey!



The number of mobile devices is increasing at a great speed in today's world. Mobile phones are essential to people's daily lives and tablets are becoming more and more popular. The biggest difference between phones and tablets is the size: phone screen sizes are between 3 and 6 in.; tablet screen sizes are between 7 and 10 in. The difference in size will make the same UI looks different. For instance, some designs may look beautiful in phones but may look distorted in tablets as certain UI components will grow disproportionately or too much space between the UI components.

Professional Android developers need to know how to make their apps support both phones and tablets. Fragment was introduced in Android 3.0 which can help to make UI display in tablets. Let us learn how to use fragment.

## 5.1 What Is Fragment?

Fragment is a piece of UI components that can be placed in Activity. It can help to make full use of the big screen thus is widely used in tablets. You should find that it is easy to learn this new concept as it is very similar to Activity- both have layouts, lifecycle. You can even think of Fragment as mini-Activity, although this mini-Activity can be the same size as normal Activity.

How to use Fragment to make full use of tablet screen? Imagine that we are building a news app. In one screen, the RecyclerView will display the title of the news and clicking the title will open another screen to display the full article of the corresponding news. If the app only runs in phones, we can put the news title list in an Activity and put the full article in another Activity as shown in Fig. 5.1.

However, if we use the same design in tablets, as the titles of the news are usually not very long, there will be lots of empty space as shown in Fig. 5.2.

A better solution is to put the title list and content in two Fragments and place them in the same Activity so that the space can be fully utilized as shown in Fig. 5.3.

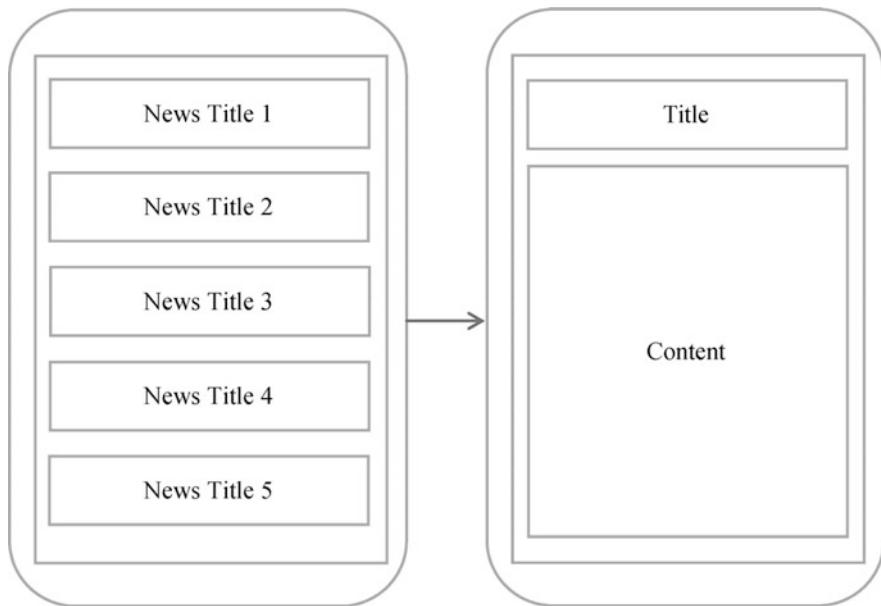
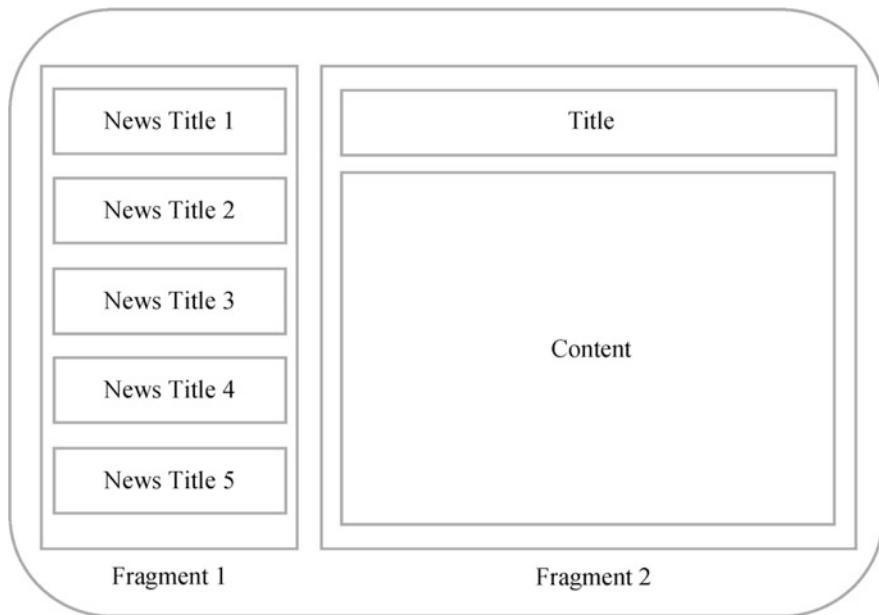


Fig. 5.1 Phone UI design



Fig. 5.2 News Title list in tablet



**Fig. 5.3** Two-pane layout for tablet

## 5.2 How to Use Fragment

Now let us learn how to use Fragment. First, create a tablet emulator by following the process in Chap. 1. Create a Pixel C tablet emulator and run the emulator. It should show up as in Fig. 5.4.

Next create FragmentTest project so that we can experiment with Fragment.

### 5.2.1 Basic Use of Fragment

Let us start with the basic use of Fragment. First, create two Fragments in Activity and let them share the screen equally.

Create left\_fragment.xml for left Fragment as code below:

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

    <Button
        android:id="@+id/button"
```

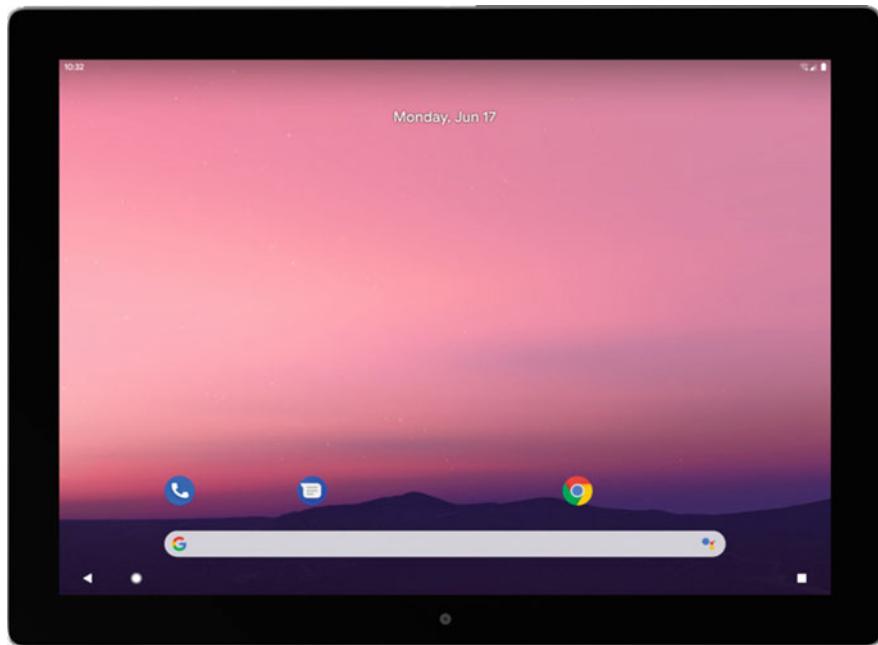


Fig. 5.4 Pixel C tablet emulator

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Button"
    />

</LinearLayout>
```

This simple layout only has one button at the center of the layout.  
Then, create right\_fragment.xml for right Fragment as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:background="#00ff00"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="24sp"
```

```
    android:text="This is right fragment"
  />

</LinearLayout>
```

This layout has green background and a TextView to display some text.

Next, create LeftFragment that inherits Fragment. There are two packages that can provide Fragment API: Android internal android.app.Fragment and AndroidX android.fragment.app.Fragment. Make sure to use the Fragment in AndroidX to ensure the app is compatible in all Android versions. The internal Fragment became obsolete since Android 9.0. There is no need to add extra dependency in build.gradle to use Fragment in AndroidX. By selecting the Use android.\*artifacts option during the project creation process, Android Studio will import the necessary AndroidX library.

Change code in LeftFragment as follows:

```
class LeftFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?,
    savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.left_fragment, container, false)
    }
}
```

Code here only overrides onCreateView() of Fragment and inflates the left\_fragment layout dynamically. Apply the same pattern to RightFragment as code below:

```
class RightFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?,
    savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.right_fragment, container, false)
    }
}
```

Next, change activity\_main.xml as code below:

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



**Fig. 5.5** Basic use of Fragment

```
<fragment
    android:id="@+id/leftFrag"
    android:name="com.example.fragmenttest.LeftFragment"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1" />

<fragment
    android:id="@+id/rightFrag"
    android:name="com.example.fragmenttest.RightFragment"
    android:layout_width="0dp"
    android:layout_height="match_parent"
    android:layout_weight="1" />

</LinearLayout>
```

We basically add the Fragment by using the `<fragment>` element and most of the attributes here should look familiar to you. Notice that we need to use `android:name` attribute to explicitly set the name of the Fragment and the name must be complete name with package name.

That is everything for our simple Fragment demo, run the app and the result should be as shown in Fig. 5.5.

The result is as we expected. The two fragments share the screen equally. However, this is very basic use of fragment and not so useful in real-world projects. Next I will introduce more advanced uses of Fragment.

### 5.2.2 *Add Fragment Dynamically*

Last section covers how to add Fragment in layout file. However, Fragment can be added dynamically in Activity and this makes fragment very useful. You can customize UI to be more diversified as fragment can be dynamically added based on the needs.

We can work on top of previous project. Create another\_right\_fragment.xml as code below:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:orientation="vertical"
    android:background="#ffff00"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="24sp"
        android:text="This is another right fragment"
        />

</LinearLayout>
```

This layout is very similar to right\_fragment.xml. It only changes the background to yellow and changes the text. Create AnotherRightFragment to be the right-side Fragment as code below:

```
class AnotherRightFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
        savedInstanceState: Bundle?) : View? {
        return inflater.inflate(R.layout.another_right_fragment,
        container, false)
    }
}
```

The above code inflates the another\_right\_fragment layout in onCreateView(). To add this Fragment dynamically in Activity, change activity\_main.xml as code below:

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

    <fragment
        android:id="@+id/leftFrag"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/rightLayout"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" >
    </FrameLayout>

</LinearLayout>
```

The right-side Fragment gets replaced by FrameLayout which will position the UI components inside it to the left upper corner. We just need a layout that we can place the Fragment without any position requirement. Thus FrameLayout is the perfect choice.

Change MainActivity to dynamically add fragment as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        button.setOnClickListener {
            replaceFragment(AnotherRightFragment())
        }
        replaceFragment(RightFragment())
    }

    private fun replaceFragment(fragment: Fragment) {
        val fragmentManager = supportFragmentManager
        val transaction = fragmentManager.beginTransaction()
        transaction.replace(R.id.rightLayout, fragment)
        transaction.commit()
    }
}
```



**Fig. 5.6** Dynamically add Fragment

```
}
```

The click event listener gets registered to the button and uses replaceFragment() to dynamically add AnotherRightFragment when the listener gets triggered. The replaceFragment() method also gets used to add RightFragment. From the code above, we can see there are five steps to add Fragment dynamically.

1. Create the instance of Fragment.
2. Acquire FragmentManager which can be done by getSupportFragmentManager() in Activity.
3. Begin transaction by using beginTransaction().
4. Replace (add) Fragment in container which is done through replace() and needs the container id and the Fragment instance.
5. Commit the transaction with commit() on transaction.

These are all the steps to dynamically add Fragment in Activity. Run the app again and you should see the same screen as previous example and clicking the button should show something as in Fig. 5.6.

### 5.2.3 Back Stack for Fragment

We implemented the functionality to dynamically add Fragment in Activity. If you click the button to add another Fragment and then click Back button, app will exit directly. What if we want the previous Fragment to show up instead of exiting the app just like the Activity back stack?

It is very straightforward to implement. FragmentTransaction has addBackStack() to add a transaction to back stack. Change MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    ...

    private fun replaceFragment(fragment: Fragment) {
        val fragmentManager = supportFragmentManager
        val transaction = fragmentManager.beginTransaction()
        transaction.replace(R.id.rightLayout, fragment)
        transaction.addBackStack(null)
        transaction.commit()
    }

}
```

Before the transaction gets committed, it gets added to the back stack by calling addBackStack() which takes a string as param. This string is the name for the back stack state and is optional. Usually, we just pass null for the name. Run the app again and click the button on the screen to add AnotherRightFragment instance to Activity, then click Back button. You will find that app will not exit but go back to the RightFragment screen. Click Back button again, RightFragment will disappear too. After another click, app will exit.

### 5.2.4 Interaction Between Fragment and Activity

Though Fragment is placed in Activity to display. However, they are loosely coupled and exist in different independent classes. They don't have a way to directly interact with each other. What if we need to call the Fragment methods in Activity or vice versa?

FragmentManager provides findFragmentById() to get instance of Fragment with layout id. For instance, to get LeftFragment, use the code below:

```
val fragment = supportFragmentManager.findFragmentById(R.id.
leftFrag) as LeftFragment
```

After getting the instance of Fragment with `findFragmentById()` in Activity, we can call the eligible methods in Fragment easily.

What about calling Activity methods from Fragment? That is even easier. Inside the Fragment, the instance of Activity that the Fragment instance is attached to can be acquired by `getActivity()` as shown below:

```
if (activity != null) {  
    val mainActivity = activity as MainActivity  
}
```

We need nullability check because `getActivity()` can return null. Once Fragment gets the instance of Activity, it is straightforward to call the methods in Activity. If Fragment instance needs Context instance, then `getActivity()` can also serve the purpose. Because Activity instance is an instance of Context.

You may ask another question after the problem of communication between Fragment and Activity has been figured out that is how to communicate between different Fragment instances.

## 5.3 Lifecycle of Fragment

Just like Activity, Fragment also has its own lifecycle that is quite similar to Activity lifecycle. Let us take a look at it in this section.

### 5.3.1 Fragment State and Callbacks

Can you still recall how many states are in Activity's lifecycle? There are four states: running, paused, stopped, and destroyed. Similarly, Fragment also has these states with some minor differences.

#### 1. Running

When an Activity that a Fragment is attached to is in running state, then this Fragment is also in running state.

#### 2. Paused

When an Activity gets into paused state (another Activity that is not full screen gets added to the top of the back stack), then the Fragments that are attached to it will get into paused state.

#### 3. Stopped

When an Activity gets into stopped state, then the Fragments that are attached to it will get into stopped state. Another scenario is that: if `addToBackStack()` gets called before `FragmentTransaction` commits `remove()` or `replace()`, then Fragment

will gets into stopped state. Fragment is invisible when in stopped state and can be recycled by the OS.

#### 4. Destroyed

Fragment's existence has dependency on Activity. Thus, when Activity gets destroyed, the Fragments that are attached to it will get destroyed too. If `addToBackStack()` doesn't get called before committing `remove()`, `replace()`, Fragment will also get destroyed.

You should find it easy to understand the Fragment states with the previous knowledge about Activity states. Same as Activity, Fragment class also provides a series of callbacks to cover every state of its lifecycle. Almost all the lifecycle callbacks existing in Activity also exist in Fragment. Fragment provides some extra callbacks and below are a few that are worth more attention.

- `onAttach()`: called when Fragment gets attached to Activity.
- `onCreateView()`: called when Fragment creates the view (mount the layout).
- `onActivityCreated()`: called when the Activity that the Fragment is attached to has finished instantiation.
- `onDestroyed()`: called when the view of the Fragment has been removed.
- `onDetach()`: called when Fragment gets detached from Activity.

The full lifecycle of Fragment is shown in Fig. 5.7.

### 5.3.2 Experiment with Fragment Lifecycle

For simplicity, let us work on top of FragmentTest project. Change `RightFragment` as code below:

```
class RightFragment : Fragment() {

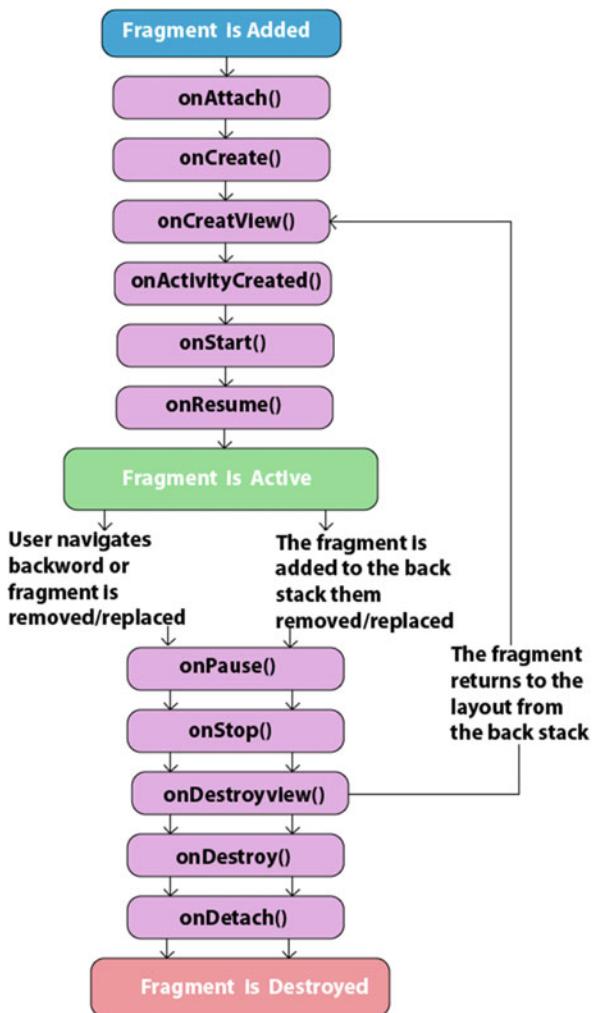
    companion object {
        const val TAG = "RightFragment"
    }

    override fun onAttach(context: Context) {
        super.onAttach(context)
        Log.d(TAG, "onAttach")
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.d(TAG, "onCreate")
    }

    override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
```

**Fig. 5.7** Fragment lifecycle  
 (Source: <https://www.javatpoint.com/android-fragments>)



```

    savedInstanceState: Bundle?) : View? {
        Log.d(TAG, "onCreateView")
        return inflater.inflate(R.layout.right_fragment, container, false)
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        Log.d(TAG, "onActivityCreated")
    }

    override fun onStart() {
        super.onStart()
    }
  
```

```

        Log.d(TAG, "onStart")
    }

override fun onResume() {
    super.onResume()
    Log.d(TAG, "onResume")
}

override fun onPause() {
    super.onPause()
    Log.d(TAG, "onPause")
}

override fun onStop() {
    super.onStop()
    Log.d(TAG, "onStop")
}

override fun onDestroyView() {
    super.onDestroyView()
    Log.d(TAG, "onDestroyView")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(TAG, "onDestroy")
}

override fun onDetach() {
    super.onDetach()
    Log.d(TAG, "onDetach")
}

}

```

The TAG const is created for filtering logs. You can declare const using the same pattern in companion object, singleton, or top level of the class.

When any of the callback methods get called, the corresponding log will get called. Run the app again and observe the Logcat. The result should be as shown in Fig. 5.8.

You can see the order of callbacks when RightFragment gets shown for the first time: onAttach(), onCreate(), onCreateView(), onActivityResult(), onStart(), and onResume(). Click the button in LeftFragment, Logcat should show logs as shown in Fig. 5.9.

Since AnotherRightFragment replaced RightFragment, RightFragment is in stopped state and onPause(), onStop(), and onDestoryView() would execute one by one. If addToBackStack() wasn't called, then RightFragment would be in destroyed state and onDestory(), onDetach() would be executed in order.

Logcat screenshot showing the following log entries:

```
com.example.fragmenttest (12059) D/RightFragment: onAttach  
com.example.fragmenttest (12059) D/RightFragment: onCreate  
com.example.fragmenttest (12059) D/RightFragment: onCreateView  
com.example.fragmenttest (12059) D/RightFragment: onActivityCreated  
com.example.fragmenttest (12059) D/RightFragment: onStart  
com.example.fragmenttest (12059) D/RightFragment: onResume
```

Fig. 5.8 Logs when starting app

Logcat screenshot showing the following log entries:

```
com.example.fragmenttest (12059) D/RightFragment: onPause  
com.example.fragmenttest (12059) D/RightFragment: onStop  
com.example.fragmenttest (12059) D/RightFragment: onDestroyView
```

Fig. 5.9 Logs when Fragment gets replaced

Logcat screenshot showing the following log entries:

```
com.example.fragmenttest (12059) D/RightFragment: onCreateView  
com.example.fragmenttest (12059) D/RightFragment: onActivityCreated  
com.example.fragmenttest (12059) D/RightFragment: onStart  
com.example.fragmenttest (12059) D/RightFragment: onResume
```

Fig. 5.10 Logs when back to RightFragment

Logcat screenshot showing the following log entries:

```
com.example.fragmenttest (12059) D/RightFragment: onPause  
com.example.fragmenttest (12059) D/RightFragment: onStop  
com.example.fragmenttest (12059) D/RightFragment: onDestroyView  
com.example.fragmenttest (12059) D/RightFragment: onDestroy  
com.example.fragmenttest (12059) D/RightFragment: onDetach
```

Fig. 5.11 Logs when exit app

Click Back button, RightFragment should appear again and the Logcat logs should be as shown in Fig. 5.10.

RightFragment goes back to running state. Thus onCreateView(), onActivityCreated(), onStart(), and onResume() methods will get executed. Notice that onCreate() doesn't get called because the addToBackStack() method prevents RightFragment gets destroyed.

Click the Back button again and Logcat should show logs as shown in Fig. 5.11.

onPaus(), onStop(), onDestroyView(), onDestroy(), and onDetach() get executed consecutively and Fragment gets destroyed. After experimenting the whole lifecycle of Fragment, you should have deeper understanding of the Fragment lifecycle.

It is worth noting that you can also save data with onSaveInstanceState() in Fragment as the stopped Fragment can be recycled when due to memory constraint. The saved data can be accessed in onCreate(), onCreateView(), and onActivityCreated() with savedInstanceState argument that is Bundle type. Refer to Sect. 3.4.5 for code example.

## 5.4 Dynamically Load Layout

Dynamically adding Fragment can solve lots of practical problems. However, it can only add or replace UI components in layout. It would be great if we can load the layout at runtime based on the screen resolution and/or size. In this section, I will cover how to dynamically load the layout.

### 5.4.1 Use Qualifier

If you are a frequent user of tablet, you should notice that a lot of the tablet apps adopt the two-pane mode as we discussed in the beginning of the chapter. This is because tablet screens are large enough to display contents from two panes and phones only have space to display one pane.

We can use qualifier to decide the display mode with qualifier at runtime. Let us use an example to learn to how use qualifier. Change activity\_main.xml in FragmentTest as code below:

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

    <fragment
        android:id="@+id/leftFrag"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Only the left Fragment has been kept and fills the parent layout. Next, create layout-large folder in res folder. In the newly created folder, create a new layout activity\_main.xml as code below:

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

    <fragment
        android:id="@+id/leftFrag"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/rightFrag"
        android:name="com.example.fragmenttest.RightFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />

</LinearLayout>
```

We can see that layout/activity\_main layout has only one Fragment which means that it is for one pane mode while layout-large/activity\_main has two Fragments inside which means that it is for two-pane mode. Here, large is a qualifier. The devices that are considered to be large will auto load the layout under layout-large folder while small screen devices will load layouts under layout folder.

Next, comment out the code in replaceFragment() and run the app again in the tablet emulator. The result should be as shown in Fig. 5.12.

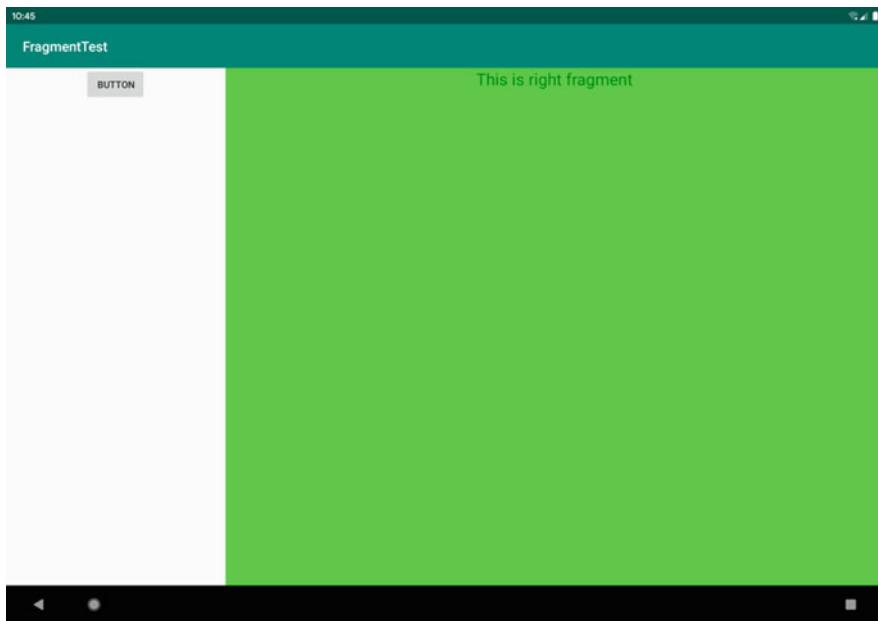
Start a phone emulator and run the app in it. The result should be as shown in Fig. 5.13.

With the code above, we just implemented the functionality of dynamically load layout at runtime.

Table 5.1 shows some of the commonly used qualifiers.

#### 5.4.2 Use Smallest-Width Qualifier

The large qualifier solves the problem of deciding when to use one-pane mode and when to use two-pane mode. But how large is large? Sometimes we just want to use two-pane mode layouts even if system doesn't think the device is large. We can use smallest-width qualifier to solve the problem.



**Fig. 5.12** Two-pane mode display

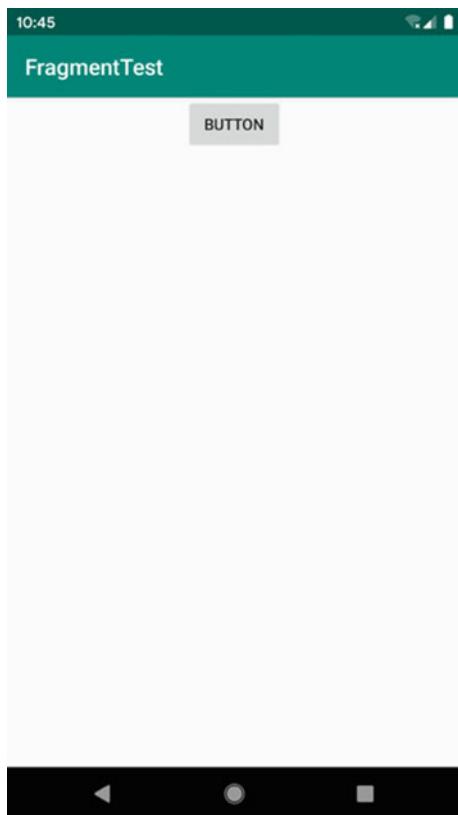
This qualifier allows use to set a minimum value using dp as unit. At runtime, different layouts get loaded by comparing the screen width to this value.

Create layout-sw600dp folder under res and create activity\_main.xml in this new folder as code below:

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

    <fragment
        android:id="@+id/leftFrag"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/rightFrag"
        android:name="com.example.fragmenttest.RightFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />
```

**Fig. 5.13** Single-pane mode display**Table 5.1** Android frequently used qualifiers

| Screen property | Qualifier | Note  |
|-----------------|-----------|---|
| Size            | Small     | Resource for small screen                                       |
|                 | Normal    | Resource for normal screen                                      |
|                 | Large     | Resource for large screen                                       |
|                 | xlarge    | Resource for xlarge screen                                      |
| Resolution      | ldpi      | Resource for low-density device(below 120 dpi)                  |
|                 | mdpi      | Resource for medium-density device(120 dpi ~ 160 dpi)           |
|                 | hdpi      | Resource for high-density device(160 dpi ~ 240 dpi)             |
|                 | xhdpi     | Resource for extra-high-density device(240 dpi ~ 320 dpi)       |
|                 | xxhdpi    | Resource for extra-extra-high-density device(320 dpi ~ 480 dpi) |
| Orientation     | Land      | Resource for device in landscape mode                           |
|                 | Port      | Resource for device in portrait mode                            |

```
</LinearLayout>
```

With the above code, when the app is running in a device that has width equal or more than 600dp, layout-sw600dp/activity\_main will be used, and if the device has width less than 600dp, the default layout/activity\_main will be used.

## 5.5 Fragment Best Practice: A Basic News App

It is time to apply what we have learned so far to build an app.

As mentioned previously, Fragment is used widely to develop apps for tablets since it can maximize the use of screen. But does it mean that we need to develop different apps specifically for tablet and phone? Some companies do have different apps for phone and tablet which is an expensive investment. This is because when releasing a new feature or fix a bug, developers need to write code twice. In this section, I will cover how to write apps that support both phones and tablets.

Do you still remember the news app mentioned in the beginning of this chapter? We can apply what we have learned so far to develop a very basic news app and make it support phone and tablet. Create FragmentBestPractice project and let's begin!

First add dependency to RecyclerView by changing code in app/build.gradle as code below:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$
    kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    implementation 'androidx.constraintlayout:
constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:
core:3.1.1'
}
```

Next, create News class for news as code below:

```
class News(val title: String, val content: String)
```

This class is self-explanatory. Create layout file news\_content\_frag.xml to host the content of the news as code below:

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

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

    <LinearLayout
        android:id="@+id/contentLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:visibility="invisible" >

        <TextView
            android:id="@+id/newsTitle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:padding="10dp"
            android:textSize="20sp" />

        <View
            android:layout_width="match_parent"
            android:layout_height="1dp"
            android:background="#000" />

        <TextView
            android:id="@+id/newsContent"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:padding="15dp"
            android:textSize="18sp" />

    </LinearLayout>

    <View
        android:layout_width="1dp"
        android:layout_height="match_parent"
        android:layout_alignParentLeft="true"
        android:background="#000" />

</RelativeLayout>
```

There are two main parts of the content layout: title at the top and article at the bottom. Between them, we use a horizontal thin line as divider. We also use a vertical think line to divide the news list and the news content. To make a thin line, we just need to create a View and set its width or height to have value of 1 dp and use background attribute to set the color of the line. Here we set the color of the thin line to be black.

We also need to set the content layout to be invisible since, in two-pane mode, this layout should not appear without selecting news.

Next, create NewsContentFragment class that inherits Fragment as code below:

```

class NewsContentFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
    savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.news_content_frag, container,
    false)
    }

    fun refresh(title: String, content: String) {
        contentLayout.visibility = View.VISIBLE
        newsTitle.text = title
        newsContent.text = content
    }

}

```

The code above inflates the `news_content_frag` layout in `onCreateView()`. `refresh()` will show the title and content of news in the layout just mentioned. Notice that `refresh()` also set the content layout to be visible.

Now the news content fragment and layout are ready for use in two-pane mode. If we want to use them in one pane mode, we need to create another Activity. Right click `com.example.fragmentbestpractice->New->Activity->Empty Activity` and create `NewsContentActivity` and use the default name for layout (`activity_news_content`). Change `activity_news_content.xml` as code below:

```

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

    <fragment
        android:id="@+id/newsContentFrag"
        android:name="com.example.fragmentbestpractice.
NewsContentFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    />

</LinearLayout>

```

We can reuse the previous code by using `NewsContentFragment` directly which will use the `news_content_frag` layout automatically.

Change `NewsContentActivity` as code below:

```
class NewsContentActivity : AppCompatActivity() {
```

```

companion object {
    fun actionStart(context: Context, title: String, content: String)
    {
        val intent = Intent(context, NewsContentActivity::class.java)
        apply {
            putExtra("news_title", title)
            putExtra("news_content", content)
        }
        context.startActivity(intent)
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_news_content)
    val title = intent.getStringExtra("news_title") // get the news
title
    val content = intent.getStringExtra("news_content") // get the
news content
    if (title != null && content != null) {
        val fragment = newsContentFrag as NewsContentFragment
        fragment.refresh(title, content) //refreshNewsContentFragment
    }
}
}

```

In onCreate() we can acquire the news title and news content from Intent and then kotlin-android-extensions. Next we can use refresh() to display the title and content.

If you forget actionStart(), please refer to Sect. 3.6.3 for more detail.

Next create the layout that can display the list of news titles. Create news\_title\_fragment.xml as code below:

```

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

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/newsTitleRecyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

</LinearLayout>

```

It is self-explanatory and only has a RecyclerView inside. Next, create news\_item.xml as the item view layout for RecyclerView as code below:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/
    android"
        android:id="@+id/newsTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:maxLines="1"
        android:ellipsize="end"
        android:textSize="18sp"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:paddingTop="15dp"
        android:paddingBottom="15dp" />
```

The item view is also very simple. It is just a TextView. There are a few attributes I haven't covered: android:padding adds some space within the control to make some room between the text view and control border; android:maxLines sets the limit to lines of text to display; android:ellipsize set the way to display ellipsis and here the ellipsis will be at the end of the text.

After creating the list and item view layouts, we need a place to display them. Create NewsTitleFragment to be the Fragment that will host the news title list as code below:

```
class NewsTitleFragment : Fragment() {

    private var isTwoPane = false

    override fun onCreateView(inflater: LayoutInflater, container:
    ViewGroup?,
        savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.news_title_frag, container,
    false)
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        isTwoPane = activity?.findViewById<View>(R.id.newsContentLayout)
    != null
    }
}
```

NewsTitleFragment is short and mostly self-explanatory. In onActivityCreated(), the existence of the View with id newsContentLayout is used as the indicator if the current display mode is one-pane or two-pane. Thus, we need to make View with id newsContentLayout to only appear in two-pane mode. Notice that getActivity() can return null, and thus we need to use? operator to ensure null safety.

We can use the qualifier to make newsContentLayout View to only appear in two-pane mode. Change activity\_main.xml as code below:

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

    <fragment
        android:id="@+id/newsTitleFrag"
        android:name="com.example.fragmentbestpractice.
    NewsTitleFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        />

</FrameLayout>
```

Code above will only show a Fragment with title list.

Next, create layout-sw600dp folder and under this folder create activity\_main.xml as code below:

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

    <fragment
        android:id="@+id/newsTitleFrag"
        android:name="com.example.fragmentbestpractice.
    NewsTitleFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/newsContentLayout"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" >

        <fragment
            android:id="@+id/newsContentFrag"
            android:name="com.example.fragmentbestpractice.
    NewsContentFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </FrameLayout>

</LinearLayout>
```

In two-pane mode, we have two Fragments and place the Fragment for news content in a FrameLayout whose id is newsContentLayout. Thus, if this view can be referenced then display mode is two-pane otherwise one-pane.

That is almost everything we need to do. One last thing to display the news list in the RecyclerView is to create the adapter. We can create an inner class NewsAdapter for the RecyclerView as shown below:

```
class NewsTitleFragment : Fragment () {

    private var isTwoPane = false

    ...

    inner class NewsAdapter(val newsList: List<News>) :
        RecyclerView.Adapter<NewsAdapter.ViewHolder>() {

        inner class ViewHolder(view: View) : RecyclerView.ViewHolder
            (view) {
            val newsTitle: TextView = view.findViewById(R.id.newsTitle)

        }

        override fun onCreateViewHolder(parent: ViewGroup, viewType:
        Int): ViewHolder {
            val view = LayoutInflater.from(parent.context)
                .inflate(R.layout.news_item, parent, false)
            val holder = ViewHolder(view)
            holder.itemView.setOnClickListener {
                val news = newsList[holder.adapterPosition]
                if (isTwoPane) {
                    // if two-pane, refreshNewsContentFragment
                    val fragment = newsContentFrag as NewsContentFragment
                    fragment.refresh(news.title, news.content)
                } else {
                    // if one-pane start NewsContentActivity
                    NewsContentActivity.actionStart(parent.context, news.title,
                        news.content)
                }
            }
            return holder
        }

        override fun onBindViewHolder(holder: ViewHolder, position: Int) {
            val news = newsList[position]
            holder.newsTitle.text = news.title
        }

        override fun getItemCount () = newsList.size
    }
}
```

The code above should be easy to understand as we have practiced RecyclerView several times. Notice that previous adapter examples are all individual classes and adapter can be an inner class too. The benefit of this is that the adapter can access the variables of NewsTitleFragment directly, for instance, isTwoPane.

The click event listener in onCreateViewHolder() first gets the News instance at the clicked position, then use isTwoPane to decide the display mode. If it is one-pane mode, then start a new Activity to display the content; if it is two-pane mode, then refresh NewsContentFragment with the new data.

The last step is to fill the RecyclerView with data. Change NewsTitleFragment as code below:

```
class NewsTitleFragment : Fragment() {  
    ...  
    override fun onActivityCreated(savedInstanceState: Bundle?) {  
        super.onActivityCreated(savedInstanceState)  
        isTwoPane = activity?.findViewById<View>(R.id.newsContentLayout)  
    } = null  
    val layoutManager = LinearLayoutManager(activity)  
    newsTitleRecyclerView.layoutManager = layoutManager  
    val adapter = NewsAdapter(getNews())  
    newsTitleRecyclerView.adapter = adapter  
}  
  
private fun getNews(): List<News> {  
    val newsList = ArrayList<News>()  
    for (i in 1..50) {  
        val news = News("This is news title $i", getRandomLengthString  
("This is news  
        content $i. "))  
        newsList.add(news)  
    }  
    return newsList  
}  
  
private fun getRandomLengthString(str: String): String {  
    val n = (1..20).random()  
    val builder = StringBuilder()  
    repeat(n) {  
        builder.append(str)  
    }  
    return builder.toString()  
}  
...  
}
```

The above code uses the pattern mentioned before to use RecyclerView. To use RecyclerView in Fragment is the same as using in Activity and no further explanation is needed. The getNews() method is used to initialize 50 news content by using getRandomLengthString() to generate random length news content to make sure that the news content look different from each other.

**Fig. 5.14** One-Pane News Title List



That is everything we need to do. Let us run it in the emulator! First run the app in phone emulator and the result should be as shown in Fig. 5.14.

You should see lots of news titles and clicking the title of the news will start a new Activity to show the news details as shown in Fig. 5.15.

Next, run the app in tablet emulator and click the news title. The result should be as shown in Fig. 5.16.

As you can see, with the same code, the UI will appear differently in phone and tablet now. This example should help you get a deeper understanding of Fragment. I will stop here for Fragment-related topics and it is time for Kotlin Class again.

**Fig. 5.15** One-Pane News Content Details



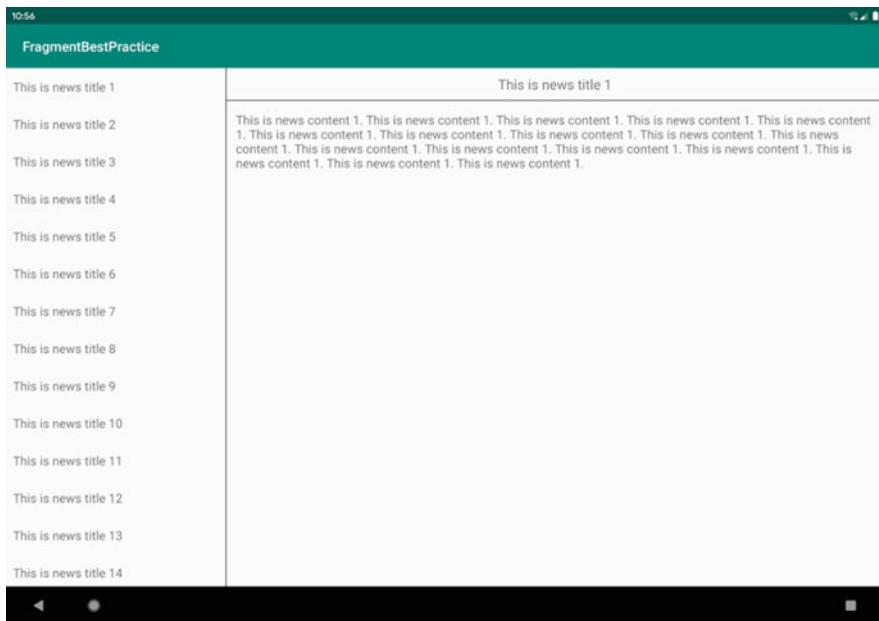
## 5.6 Kotlin Class: Extension Function and Operator Overloading

With practices from the previous chapters, you should be fluent in Kotlin now. It is time to explore some more advanced topics in Kotlin which is the purpose of this section.

### 5.6.1 Extension Function

Quite a few modern advanced programming languages have the concept of extension functions. However, Java never supports this feature which is somewhat disappointing. Good thing is that Kotlin supports extension functions and we cannot miss this topic in this book!

So, what is extension function? Extension function allows you to add new functionalities to a class without changing the code of that class.



**Fig. 5.16** News title and content details in two-pane mode

To help you better understand this, let's use an example to demonstrate. Assume a text string will have alphabetic letters, numbers, and special characters and we want to calculate the number of alphabetic letters in the string. What can you do to implement this functionality? I think the following code looks like the natural solution for most people:

```
object StringUtil {

    fun lettersCount(str: String): Int {
        var count = 0
        for (char in str) {
            if (char.isLetter()) {
                count++
            }
        }
        return count
    }
}
```

The StringUtil singleton class defines lettersCount() function which takes a string as param. In this method, for-in loop statement will iterate through all the characters in the string and if the character is an alphabetic letter, increase the counter by 1. At the end of the code, return the value of the counter.

Now we can apply this singleton class to any string-like code below:

```
val str = "ABC123xyz!@#"  
val count = StringUtil.lettersCount(str)
```

Apparently, this will work and is the standard pattern of Java programming. However, with extension functions, we can implement this functionality that is more OOD. For instance, we can add lettersCount() to String class.

The extension function syntax is quite simple which is as shown below:

```
fun ClassName.methodName(param1: Int, param2: Int): Int {  
    return 0  
}
```

Compared with the normal function, extension function just needs to add ClassName. Before the actual function name. And now this function can be used by the specified class.

Let us apply the syntax above to solve the problem. Since we want to add an extension function to String class, first we need to create String.kt. The name of the file has no requirement but I'd recommend to use the class name that the extension function will add to. This should help you search the method later. Extension function can be declared in any Class and creating new file is optional. However, it is a good idea to declare the extension function to be top-level method so that this function is globally accessible.

Put the following code in String.kt:

```
fun String.lettersCount(): Int {  
    var count = 0  
    for (char in this) {  
        if (char.isLetter()) {  
            count++  
        }  
    }  
    return count  
}
```

Notice that, since lettersCount() is an extension function of String class, it will have the context of String class automatically. There is no need to take a string as param and instead this will be used to reference the String instance.

After the definition is finished, counting the alphabetic letters in a string can be done by code below:

```
val count = "ABC123xyz!@#".lettersCount()
```

As you can see, it looks like String class defines lettersCount() internally.

Extension can make API more succinct, rich and object-oriented in lots of scenarios. Let us use String class as an example again. String is a final class and no class can inherit it. In Java, this means that String's APIs will be limited by the official String class definition. However, in Kotlin, with the help of extension functions, we can add whatever extension functions we want to String class which can greatly enrich the API of String. For instance, reverse() and capitalize() are all extension functions Kotlin provides internally. Extension function makes coding much easier.

Extension function has no limitation on which class can be extended or not. You can add extension function to any class. If you can use this feature correctly, it will help you write code with higher quality and at a faster speed.

### 5.6.2 *Operator Overloading*

Operator overloading is an interesting syntax sugar in Kotlin. We all know that Java has lots of internal operators, for instance, `+-*/%++--`. Kotlin allows overloading for all the operators and even keywords to extend the use of operators and keywords.

This section's topics are more advanced than previous Kotlin topics. But I guarantee you that this is a very interesting section and you will benefit a lot from this section.

The arithmetic operators are designed to be applied to numbers. However, Kotlin operator overloading feature allows to apply these arithmetic operators to any objects.

Of course, we need to make sure that operator overloading actually makes sense when we use it. For instance, adding two Student objects does not make a lot of sense but adding two Money object apparently makes more sense.

Let us take a look at the basic syntax of operator overloading then we will apply the syntax to implement functionality to add two Money objects.

Operator overloading is done by simply adding operator keyword before the specified function. However, what this function will do is a more important question. The `+` operator's overloading function is `plus()` and the `-` operator's overloading function is `minus()`.

Use the plus operator as an example. In order to allow adding two objects, we can use the code as shown below:

```
class Obj {  
  
    operator fun plus(obj: Obj) : Obj {  
        // Logic to handle adding  
    }  
}
```

In the code example above, operator keyword and function name plus are fixed; the params and return value are based on your logic. Code above will add one object to another object and return a new object of Obj. The way to use this operator is as follows:

```
val obj1 = Obj()  
val obj2 = Obj()  
val obj3 = obj1 + obj2
```

This is just another syntax sugar that Kotlin provides. It will be converted to obj1. plus(obj2) during compilation.

With the example above, let's try to make plus operation work for Money objects.

First, let us define the structure of the Money class. Here I will have the primary constructor of Money to take a value param of Int type for the value of the currency. Create Money.kt file with code as shown below:

```
class Money(val value: Int)
```

After the definition, we can use operator overloading to implement adding two Money objects:

```
class Money(val value: Int) {  
  
    operator fun plus(money: Money): Money {  
        val sum = value + money.value  
        return Money(sum)  
    }  
  
}
```

As you can tell, we use operator keyword before plus() function which is required for operator overloading. Inside plus(), we add the value of the current Money object and the value of the passed-in Money object, then use the sum to instantiate another Money object and return this object. Now two Money objects can be added together.

We can use the code below to verify if plus operator overloading works or not:

```
val money1 = Money(5)  
val money2 = Money(10)  
val money3 = money1 + money2  
println(money3.value)
```

The printed value will be 15, you can verify by yourself.

It would be inconvenient if we only allow Money object to be directly added with another Money object. We can also allow Money object add with number directly because Kotlin supports different overloading functions for the same operator as shown in the code below:

```

class Money(val value: Int) {

    operator fun plus(money: Money) : Money {
        val sum = value + money.value
        return Money(sum)
    }

    operator fun plus(newValue: Int) : Money {
        val sum = value + newValue
        return Money(sum)
    }
}

```

In the code above, we overload the `plus()` function once again with a different param which is an integer type. The rest of the function is the same as the previous overloading function.

Now `Money` object can add with integers directly as show below:

```

val money1 = Money(5)
val money2 = Money(10)
val money3 = money1 + money2
val money4 = money3 + 20
println(money4.value)

```

Code above will add 20 to the `money3` object and the printed result will be 35.

Of course, you can extend the example here and add currency rate exchange functionality. For example, add the `Money` object with RMB currency with `Money` object with USD currency type and return a new `Money` object. There are lots of ways to play with operator overloading.

We spend significant time on `plus` operator overloading; however, there are many more operators and keyword for overloading that Kotlin supports. Apparently, we cannot cover them one by one. Thus I list the commonly used operators that can be overloaded and the syntax sugar examples in Table 5.2. If you want to overload any of the operator or keyword, please refer to the pattern mentioned above.

Notice the last row, the corresponding syntax sugar for `a in b` is `b.contains(a)` instead of `a.contains(b)`. For example, the `String` class in Kotlin overloads `contains()`. Thus, if we want to check if “hello” string contains substring “he,” we can use the code below:

```

if ("hello".contains("he")) {
}

```

With overloading syntax sugar, we can replace the code above as follows:

```

if ("he" in "hello") {
}

```

**Table 5.2** Syntax Sugar expressions and corresponding functions

| Syntax sugar expressions | Corresponding function |
|--------------------------|------------------------|
| a + b                    | a.plus(b)              |
| a - b                    | a.minus(b)             |
| a * b                    | a.times(b)             |
| a / b                    | a.div(b)               |
| a % b                    | a.rem(b)               |
| a++                      | a.inc()                |
| a--                      | a.dec()                |
| +a                       | a.unaryPlus()          |
| -a                       | a.unaryMinus()         |
| !a                       | a.not()                |
| a == b                   | a.equals(b)            |
| a > b                    | a.compareTo(b)         |
| a < b                    |                        |
| a >= b                   |                        |
| a <= b                   |                        |
| a..b                     | a.rangeTo(b)           |
| a[b]                     | a.get(b)               |
| a[b] = c                 | a.set(b, c)            |
| a in b                   | b.contains(a)          |

They are essentially the same but the latter is more succinct.

That is everything for operator overloading and next, let's optimize the code we implemented previously with what we just learned.

In Chap. 4 and this chapter, we used a function that can generate string of random length as shown below:

```
fun getRandomLengthString(str: String): String {
    val n = (1..20).random()
    val builder = StringBuilder()
    repeat(n) {
        builder.append(str)
    }
    return builder.toString()
}
```

This function essentially repeats n times for the string that gets passed in. If we can write code like str\*n to repeat the str n times, it would be great! And we can do so in Kotlin.

If we need to let a string times a number, then we need to overload \* operator in String class which is a class provided by the system and there is no way to edit it by ourselves. However, extension function can unblock us and add a new function to String class.

Open the String.kt file we created previously and add the code below:

```
operator fun String.times(n: Int): String {
    val builder = StringBuilder()
    repeat(n) {
        builder.append(this)
    }
    return builder.toString()
}
```

It should be easy to understand the code above now and I want to highlight that: operator keyword is required; by referring Table 5.2, the function name has to be times; since this is an extension method, we need to add String. before the function name. In times(), we can use StringBuilder and repeat function to repeat the strings n times and return the result.

Now, String can be multiplied. For instance, running the following code will yield result of: abcabcabc.

```
val str = "abc" * 3
println(str)
```

It is worth noting that String class in Kotlin already has repeat() function to repeat the string n times. Thus times() can be simplified to code below:

```
operator fun String.times(n: Int) = repeat(n)
```

With the code above, we can have the code below to implement getRandomLengthString():

```
fun getRandomLengthString(str: String) = str * (1..20).random()
```

Isn't it an elegant solution? There are lots of ways to play with extension functions and operator overloading, we will cover more about this in chapters later.

## 5.7 Summary and Comment

As you can see that it is complicated work to build News app in the Best Practice section. Comparing with the app that can only be used in one form factor, there are much more to consider for apps that can support phones and tablets. However, being thoughtful and considerate during implementation will help us maintain the codebase later and be well worth the effort.

Let us review what we have learned in this chapter. First you learned the fundamentals of Fragment and its use scenarios, then with a few examples, you should master the common uses of Fragment. Then you learned about Fragment lifecycle and dynamically loading the layout and applied all these in the Best Practice section of this chapter. With so many practices, you should be quite familiar

with Fragment-related knowledge and know how to use Fragments proficiently. This chapter's Kotlin Class section is particularly rich in contents as you learned extension function and operator overloading which are very useful in real world. You also learned how to combine them together to create some interesting functionalities.

This chapter is a milestone for Android UI topics. We will not cover new UI knowledge in the next few chapters, instead, we will utilize what we've learned to better understand the new topics in each chapter. What shall we learn next? Remember the four main components of Android mentioned in Chap. 1? We only covered Activity so far and next chapter, we are going to learn BroadcastReceiver. Let us continue the journey!

# Chapter 6

## Broadcasts in Details



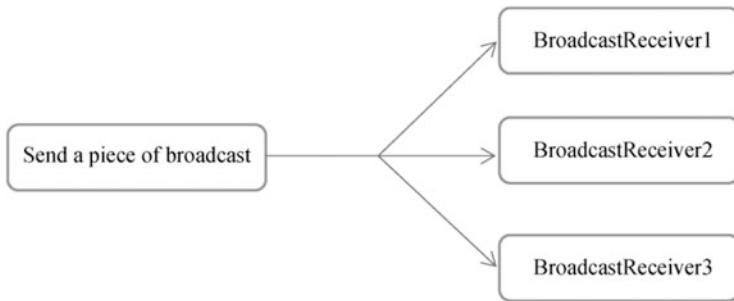
Back to my school days, each classroom would have a speaker that was connected to the broadcast control room, and whenever there was something important to announce, there would be a broadcast to notify everyone in the school. The same mechanism is widely used in computer science. If you're familiar with computer networking and communication, you should know that in IP address, the highest number is used for broadcast address. For instance, if a network's IP range is among 192.168.0.XXX, the subnet mask is 255.255.255.0, then the broadcast address for this network is 192.168.0.255. The broadcast packet will be sent to all the end points of the same network, then all the host machines in this network will receive this broadcast.

In order to make it easy to implement system level information notification, Android also introduced a similar broadcast mechanism. Compared with the two examples I mentioned above, the Android broadcast framework is much more flexible, and this chapter will cover this topic in great detail.

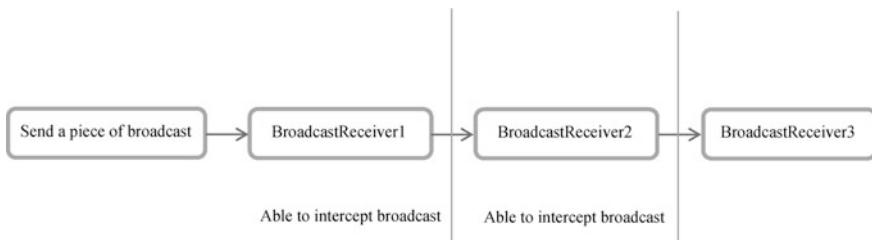
### 6.1 Introduction to Broadcast Mechanism

Why Android broadcast framework is more flexible? This is because each Android app can register the broadcast that this app is interested. This means that apps will only receive the broadcast that is interesting to the specific app which can come from the system or other apps. Android provides a suite of holistic APIs to allow apps send and receive broadcast freely. The way to send the broadcast was briefly mentioned before, and you might be able to recall. It is done with the help of Intent which we covered in Chap. 3. A new concept—BroadcastReceiver—is introduced to receive the broadcast.

I will cover the instruction of how to use BroadcastReceiver in the next section. In this section, let's first take a look at different types of broadcast. Android broadcasts can be divided mainly into two categories: normal broadcast and ordered broadcast.



**Fig. 6.1** Mechanism of normal broadcast



**Fig. 6.2** Mechanism of ordered broadcast

- Normal broadcast is a type of asynchronous broadcast. This means that after broadcast is sent, all the BroadcastReceivers will receive the broadcast simultaneously, and there is no order for which receiver will receive first. This kind of broadcast is very efficient but cannot be intercepted. Figure 6.1 demonstrates the mechanism of normal broadcast.
- Ordered broadcast is a type of synchronous broadcast. After the broadcast is sent, only one BroadcastReceiver will receive the broadcast at one time. Only when the current BroadcastReceiver finishes execution can the broadcast continue. The BroadcastReceiver with higher priority will receive the broadcast first, and the BroadcastReceiver at the front can intercept the broadcast such that the BroadcastReceiver after it will not be able to receive the broadcast. The mechanism of ordered broadcast is shown in Fig. 6.2.

With these contexts, we can start to learn how to use broadcast, and let's begin with receiving the system broadcast.

## 6.2 Receive System Broadcast

Android provides many system level broadcasts, and we can listen to these broadcasts to get the state of the system in our apps. For instance, starting the system will send a broadcast, battery capacity change will send a broadcast, system time change will also send a broadcast, etc. To listen to these broadcasts, we need BroadcastReceiver. Let us take a look how to use this class.

### 6.2.1 *Dynamically Register BroadcastReceiver for Time Change*

We can register whatever we're interested for BroadcastReceiver, and when the corresponding broadcast gets sent out, the registered BroadcastReceiver will be able to receive it and handle it inside the BroadcastReceiver. There are two ways to register the BroadcastReceiver: register in the code or inside AndroidManifest.xml. The former is called dynamic receiver, and the later is called static receiver.

How to create a BroadcastReceiver? It is actually quite simple. We just need to create a new class that inherits BroadcastReceiver and overrides onReceive(). Then when there is broadcast, onReceive() will get executed. The business logic is inside this method.

Next, let us learn the basic use of BroadcastReceiver by creating an app that can listen to the time change broadcast with dynamic receiver. Create BroadcastTest project, update MainActivity as code below;

```
class MainActivity : AppCompatActivity() {  
  
    lateinit var timeChangeReceiver: TimeChangeReceiver  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        val intentFilter = IntentFilter()  
        intentFilter.addAction("android.intent.action.TIME_TICK")  
        timeChangeReceiver = TimeChangeReceiver()  
        registerReceiver(timeChangeReceiver, intentFilter)  
    }  
  
    override fun onDestroy() {  
        super.onDestroy()  
        unregisterReceiver(timeChangeReceiver)  
    }  
  
    inner class TimeChangeReceiver : BroadcastReceiver() {
```

```

        override fun onReceive(context: Context, intent: Intent) {
            Toast.makeText(context, "Time has changed", Toast.
LENGTH_SHORT).show()
        }
    }
}

```

The code above defines an inner class TimeChangeReceiver which inherits BroadcastReceiver and overrides onReceive() of the super class. Now whenever the system time changes, onReceive() will be called. Here it will show some text message with Toast.

Then observe the onCreate() method. First we create an instance of IntentFilter and add an action to it with the value android.intent.action.TIME\_TICK. Why add this value? Because when the system time changes, the system sends out a broadcast with the value of android.intent.action.TIME\_TICK, that is to say, our BroadcastReceiver wants to listen to what broadcast, add the corresponding action here. Next, an instance of TimeChangeReceiver gets created and we use registerReceiver() with instance of TimeChangeReceiver and IntentFilter to register the receiver. Then TimeChangeReceiver instance will receive all the broadcast with action value of android.intent.action.TIME\_TICK which basically implements the functionality of listening to system time.

Lastly, never forget to unregister BroadcastReceiver for dynamic receivers. Here we unregister the receiver by using unregisterReceiver() in onDestroy().

Overall, the required code is very simple. Run the app and wait until time changes. System will send broadcast of android.intent.action.TIME\_TICK every minute, thus we only need to wait for 1 min at most to receive this broadcast as shown in Fig. 6.3.

This is the basic use of BroadcastReceiver. I used one system broadcast as an example, and receiving other system broadcast follows the same pattern. Android system will send broadcast when screen gets turned on and off, battery capacity changes, network changes, and so on. The complete system broadcasts can be found in this path.

```
<Android SDK>/platforms/<android api version>/data/
broadcast_actions.txt
```

### **6.2.2 Open App After Booting with Static Receiver**

Dynamic receivers can register and unregister to broadcast freely. Its advantage is its flexibility, and it has its disadvantage which is that only after application is running can it receive the broadcast. This is because the registration logic is inside onCreate

**Fig. 6.3** Listen to system time change

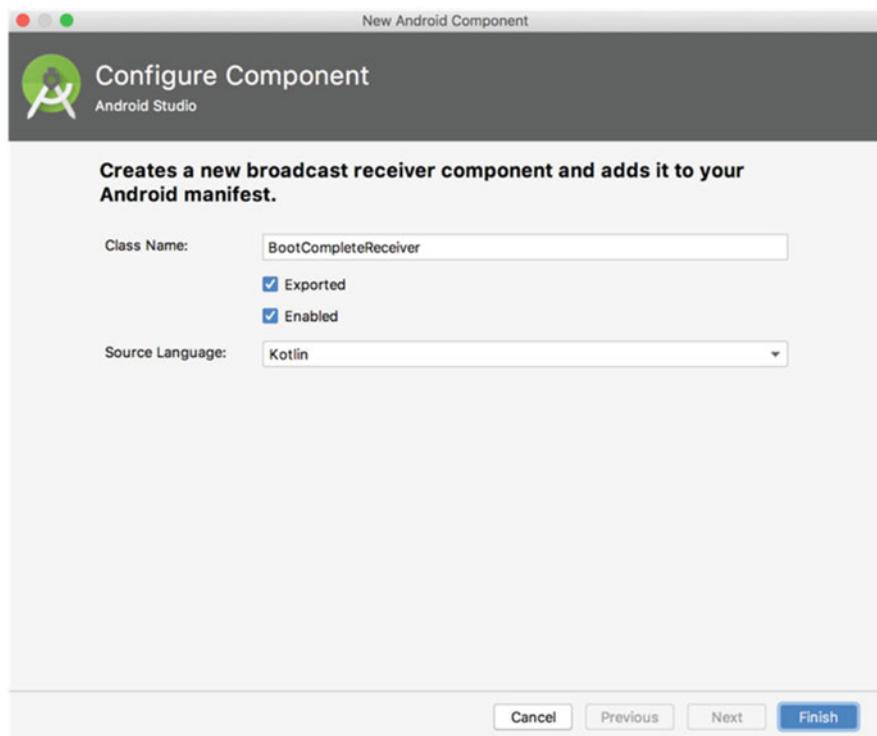


Q. Is there any way to allow apps receive broadcast before the app is running? Then static receivers come into the play.

Technically, all the system broadcasts that dynamic receiver can listen, so should the static receivers. This is the case for the older versions of Android OS. However, due to a large number of malicious applications using this mechanism to listen to system broadcasts without the program starting, any application can be woken up frequently from the background. This negatively impacted user's phone battery and speed. Thus almost every version of Android OS reduced the functionality of static BroadcastReceiver.

After Android 8.0, all the implicit broadcast cannot be listened by static receivers. Implicit broadcasts are those broadcasts that don't specify which app the broadcasts are sent to, and almost all the system broadcasts are implicit broadcasts with a few exceptions which can still be listened by static receivers. These exceptions can be found in <https://developer.android.google.cn/guide/components/broadcast-exceptions.html>.

Among these exceptions, one broadcast is android.intent.action.BOOT\_COMPLETED which is the broadcast sent after booting. Let's use it as an example to learn static receivers.



**Fig. 6.4** Window to create BroadcastReceiver

We will make the app start after booting. At boot time, our application is definitely not started, so this function obviously can't be implemented using dynamic registration, but should use static registration to receive the boot broadcast, and then execute the corresponding logic in the `onReceive()` method, so that the boot start function can be implemented.

In the previous subsection we created `BroadcastReceiver` by using the internal class method, but in fact you can also create it by using the shortcut provided by Android Studio. Right click `com.example.broadcasttest` package → New → Other → `BroadcastReceiver`. A window as shown in Fig. 6.4 will pop up.

As you can see, we created a class with the name `BootCompleteReceiver`, the `Exported` property specifies if this `BroadcastReceiver` can receive the broadcasts which do not belong to this app, `Enabled` property specifies if this `BroadcastReceiver` is enabled or not. Check these two properties and click `Finish` to finish creation.

Then update `BootCompleteReceiver` as code below:

```
class BootCompleteReceiver : BroadcastReceiver {  
  
    override fun onReceive(context: Context, intent: Intent) {  
        Toast.makeText(context, "Boot Complete", Toast.LENGTH_LONG).show  
    }  
}
```

```
()  
}  
  
}
```

We simply show a toast message in onReceive().

Static BroadcastReceiver has to register in AndroidManifest.xml to be enacted. However, since we use Android Studio's shortcut to create the BroadcastReceiver, thus registration is done automatically. Open AndroidManifest.xml, you should see code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/  
android"  
    package="com.example.broadcasttest">  
  
    <application  
        android:allowBackup="true"  
        android:icon="@mipmap/ic_launcher"  
        android:label="@string/app_name"  
        android:roundIcon="@mipmap/ic_launcher_round"  
        android:supportsRtl="true"  
        android:theme="@style/AppTheme">  
        ...  
        <receiver  
            android:name=".BootCompleteReceiver"  
            android:enabled="true"  
            android:exported="true">  
        </receiver>  
    </application>  
  
</manifest>
```

As you can see there is a new attribute <receiver> within <application>. All the static BroadcastReceivers have to register here. It is actually quite similar to <activity> attribute in a way that it also uses android:name to specify which BroadcastReceiver is getting registered here, then enabled and exported property is generated by checking the box.

However, the current BootCompleteReceiver still cannot receive the boot completed broadcast, we need to modify the AndroidManifest.xml to make it work as shown in the code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/  
android"  
    package="com.example.broadcasttest">  
  
    <uses-permission android:name="android.permission.  
RECEIVE_BOOT_COMPLETED" />
```

```

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
    <receiver
        android:name=".BootCompleteReceiver"
        android:enabled="true"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED"
/>
        </intent-filter>
    </receiver>
</application>

</manifest>

```

Since the broadcast will have value of android.intent.action.BOOT\_COMPLETED, we need to add <intent-filter> attribute within <receiver> attribute and specify the action there.

Notice that, in order to protect user's privacy and safety, Android system made it mandatory to declare the permission for some sensitive actions, otherwise, app will crash instantly. Receiving booting completed broadcast is one of such actions, thus we use <uses-permission> attribute to declare android.permission.RECEIVE\_BOOT\_COMPLETED permission.

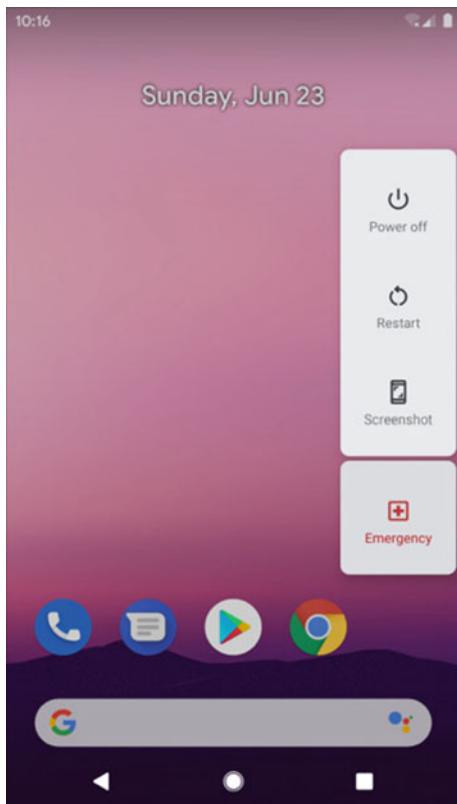
This is the first time we need to declare permission while there are many actions in Android that requires permission declaration which we will learn gradually. Receiving the booting completed broadcast action just requires declaring permission in the AndroidManifest.xml. Android 6.0 introduced more strict run time permissions which can better protect user privacy and safety. We will learn these in Chap. 8.

Run the app again and our app can receive the booting completed broadcast now. Long pressing the Power button on the right side of the emulator tool bar will pop some options, and you can restart from there as shown in Fig. 6.5.

Click Restart after booting finishes, our app will receive the booting completed broadcast as indicated in Fig. 6.6.

Up to now, we only use Toast to display some message in onReceive() of BroadcastReceiver which can be replaced by your business logic in real world project. However, it is worth noting that we don't add too many operations or time consuming operations in onReceive() because BroadcastReceiver cannot start a new thread, and if onReceive() has been running for too long, exceptions will be thrown.

**Fig. 6.5** Restart option of emulator



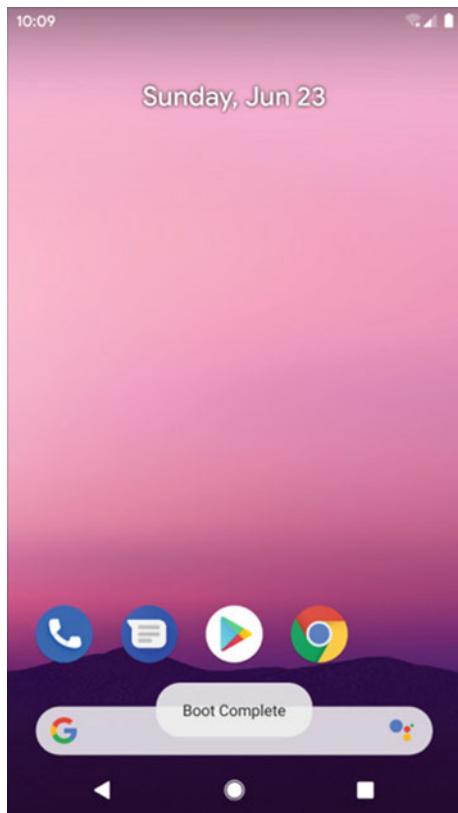
## 6.3 Send Customized Broadcast

After learning how to receive broadcast with BroadcastReceiver, let's see how to send customized broadcast in our app. As mentioned before, there are mainly two types of broadcasts: normal broadcast and ordered broadcast. In this section, we will learn the difference through some examples.

### 6.3.1 Send Normal Broadcast

Before sending the broadcast, we first need to define a BroadcastReceiver to receive this broadcast, otherwise it will be useless to send the broadcast. Create MyBroadcastReceiver and add code below in onReceive():

**Fig. 6.6** Receive system booting broadcast



```
class MyBroadcastReceiver : BroadcastReceiver() {  
  
    override fun onReceive(context: Context, intent: Intent) {  
        Toast.makeText(context, "received in MyBroadcastReceiver",  
            Toast.LENGTH_SHORT).show()  
    }  
}
```

When MyBroadcastReceiver receives the customized broadcast, it will show a toast with message “received in MyBroadcastReceiver.”

Then modify the BroadcastReceiver in AndroidManifest.xml as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/  
    android"  
        package="com.example.broadcasttest">  
    ...  
    <application  
        android:allowBackup="true"
```

```
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
<receiver
    android:name=".MyBroadcastReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="com.example.broadcasttest.
MY_BROADCAST"/>
    </intent-filter>
</receiver>
</application>
</manifest>
```

As shown in highlighted code, MyBroadcastReceiver will receive the broadcast with value of com.example.broadcasttest.MY\_BROADCAST, thus when we send the broadcast later, we need to have our broadcast have this value.

Update activity\_main.xml as code below:

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

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

</LinearLayout>
```

The code above defines a button that can trigger sending the broadcast. Update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        button.setOnClickListener {
            val intent = Intent("com.example.broadcasttest.MY_BROADCAST")
            intent.setPackage(packageName)
            sendBroadcast(intent)
        }
    }
}
```

```

    ...
}
...
}
}

```

The highlighted code adds click event to the button that has the business logic of sending the customized broadcast.

First, we need to create an Intent instance and pass the value of the broadcast we want to send. Then we need to use the setPackage() method of Intent to pass in the package name of the current app. Finally we use sendBroadcast() to send the broadcast, and all the BroadcastReceivers that listen to com.example.broadcasttest.MY\_BROADCAST will receive the broadcast. This broadcast is a normal broadcast.

I want to discuss more about setPackage(). As mentioned before, after Android 8.0, static BroadcastReceivers cannot receive implicit broadcasts and by default all of our customized broadcasts are implicit broadcasts. Thus we need to call setPackage() to specify which app is going to receive this broadcast to make it an explicit broadcast, otherwise the static BroadcastReceiver won't be able to receive this broadcast.

Run the app again and click Send Broadcast button, a toast should show up as shown in Fig. 6.7.

And we just successfully implemented the functionality of sending customized broadcast!

Also, since broadcast is sent with Intent, you can also pass in some data for receiving BroadcastReceiver which is similar to Activity.

### 6.3.2 Send Ordered Broadcast

Unlike normal broadcast, ordered broadcast is synchronous broadcast and can be intercepted. To prove this, we need to create a new BroadcastReceiver. Create AnotherBroadcastReceiver as code below:

```

class AnotherBroadcastReceiver : BroadcastReceiver {

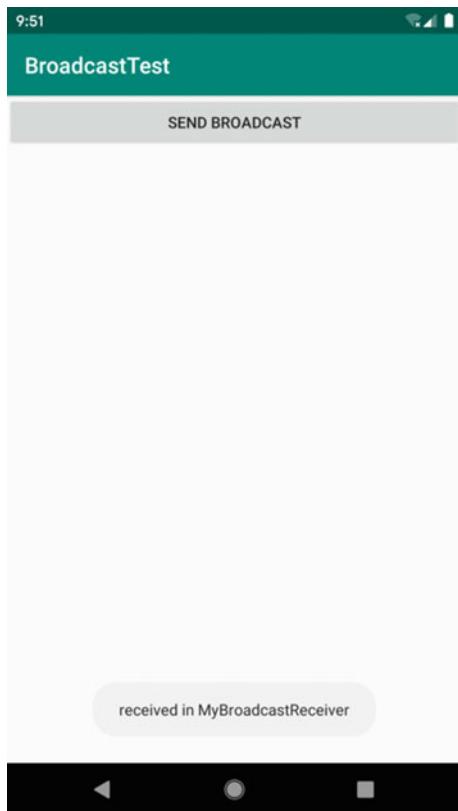
    override fun onReceive(context: Context, intent: Intent) {
        Toast.makeText(context, "received in AnotherBroadcastReceiver",
                      Toast.LENGTH_SHORT).show()
    }
}

```

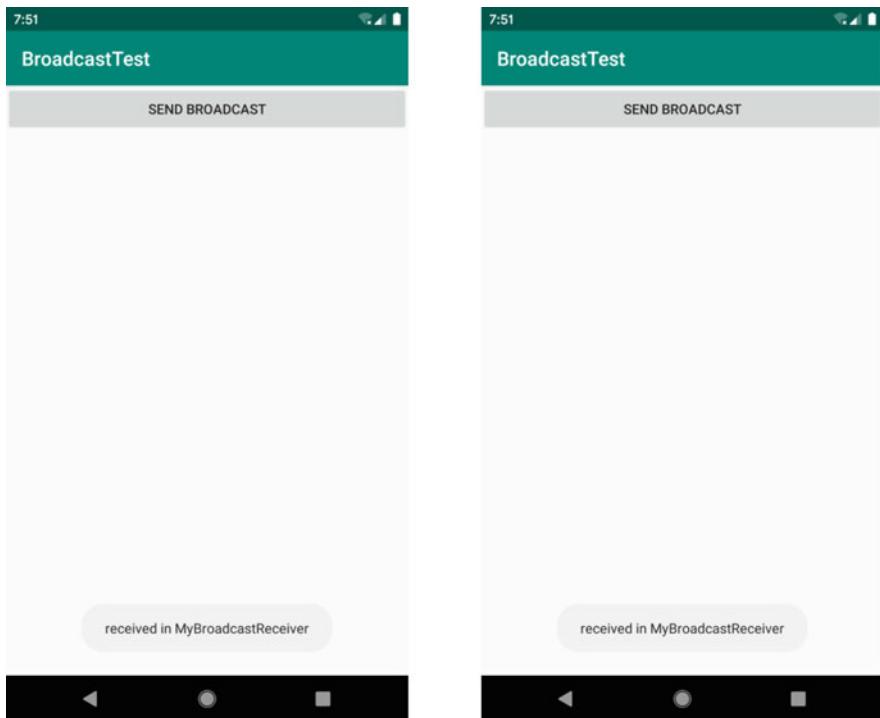
Again, we will show toast message in onReceive().

Then update the configuration of this BroadcastReceiver in AndroidManifest.xml as code below:

**Fig. 6.7** Receive customized broadcast



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest">
    ...
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <receiver
            android:name=".AnotherBroadcastReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.example.broadcasttest.
MY_BROADCAST" />
            </intent-filter>
        </receiver>
```



**Fig. 6.8** Two BroadcastReceivers received customized broadcast

```
</application>  
</manifest>
```

The highlighted code shows that AnotherBroadcastReceiver also listens to broadcast with the value of com.example.broadcasttest.MY\_BROADCAST. Run the app again and click Send Broadcast and it will show two toast messages as shown in Fig. 6.8.

Up to now, our broadcasts are all normal broadcasts, let's try to send ordered broadcasts. Go back to the BroadcastTest project and update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {  
    ...  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        button.setOnClickListener {  
            val intent = Intent("com.example.broadcasttest.MY_BROADCAST")  
            intent.setPackage(packageName)  
            sendOrderedBroadcast(intent, null)  
        }  
    }  
}
```

```
    ...
}
```

As highlighted code has shown, to make the broadcast an ordered broadcast, only one line of code change is needed. We just need to use sendOrderedBroadcast() instead of sendBroadcast(). sendOrderedBroadcast() takes two params, the first param is Intent type, and the second param is a string related with permission which we will pass null for now. Run the app again, click Send Broadcast button, and you will find that both BroadcastReceivers are still able to receive the broadcast.

Looks no difference. However, don't forget that now the BroadcastReceiver has order, and the BroadcastReceiver that first receives the broadcast will be able to intercept the broadcast and stop the propagation of the broadcast.

Then how to set the order of the BroadcastReceiver? Of course we need to set it during registration, update AndroidManifest.xml as code below;

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.broadcasttest">
    ...
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
    <receiver
        android:name=".MyBroadcastReceiver"
        android:enabled="true"
        android:exported="true">
        <intent-filter android:priority="100">
            <action android:name="com.example.broadcasttest.
MY_BROADCAST"/>
        </intent-filter>
    </receiver>
    ...
</application>
</manifest>
```

As shown in the highlighted code, the android:priority property sets the priority of the BroadcastReceiver, and the receivers with higher priority number will receive the broadcast first. Here MyBroadcastReceiver has priority of 100 which will ensure it receives the broadcast before AnotherBroadcastReceiver.

With this, MyBroadcastReceiver can determine if the broadcast can continue propagation or not. Update MyBroadcastReceiver as code below:

```

class MyBroadcastReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        Toast.makeText(context, "received in MyBroadcastReceiver",
                      Toast.LENGTH_SHORT).show()
        abortBroadcast()
    }
}

```

The `abortBroadcast()` method will stop the propagation of the broadcast, and the `BroadcastReceivers` with lower priority won't be able to receive this broadcast.

Run the app again and click Send Broadcast button, you will find that only the toast message in `MyBroadcastReceiver` will pop up which proves that the broadcast stops after `MyBroadcastReceiver`.

## 6.4 Best Practice of Broadcast: Force Logout

This chapter doesn't have as many contents as previous chapters, and you must feel that this chapter is easy right? It is time to comprehensively apply what we have learned so far with an example.

Forcing logout is a commonly seen feature. Certain apps only allow you to log in the app with one device at a time. The basic user experience is very simple, we just need to pop up a dialog and block the user from other interactions, except clicking the Confirm button in the dialog to go to the login screen. The problem is that: user can be at any screen when they are notified to logout, then do we need to write dialog logic in all the screens? This is actually not necessary at all. We can use broadcast to implement this functionality easily. Create `BroadcastBestPractice` project and let us begin.

Forcing logout needs to exit all the activities then go to the login screen. You probably can recall that in Chap. 3's best practice section, we already learned how to finish all the activities, and we will use the same solution here. First, create an `ActivityCollector` class to manage all the activities as code below:

```

object ActivityCollector {

    private val activities = ArrayList<Activity>()

    fun addActivity(activity: Activity) {
        activities.add(activity)
    }

    fun removeActivity(activity: Activity) {
        activities.remove(activity)
    }
}

```

```
fun finishAll() {
    for (activity in activities) {
        if (!activity.isFinishing) {
            activity.finish()
        }
    }
    activities.clear()
}
```

Then create BaseActivity class as the super class of all activities as code below:

```
open class BaseActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ActivityCollector.addActivity(this)
    }

    override fun onDestroy() {
        super.onDestroy()
        ActivityCollector.removeActivity(this)
    }
}
```

All the code above are reusing the existing code, easy right? From now on, we need to write some new code. First create LoginActivity as the login screen, and use the Android Studio generated for layout file. Update activity\_login.xml as code below:

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

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="60dp">
        <TextView
            android:layout_width="90dp"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:textSize="18sp"
            android:text="Account:" />
```

```

<EditText
    android:id="@+id/accountEdit"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:layout_gravity="center_vertical" />
</LinearLayout>

<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="60dp">
    <TextView
        android:layout_width="90dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:textSize="18sp"
        android:text="Password:" />

    <EditText
        android:id="@+id/passwordEdit"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_gravity="center_vertical"
        android:inputType="textPassword" />
</LinearLayout>

<Button
    android:id="@+id/login"
    android:layout_width="200dp"
    android:layout_height="60dp"
    android:layout_gravity="center_horizontal"
    android:text="Login" />

</LinearLayout>

```

The code above uses LinearLayout as container to host the login layout. At the outer most layer is a vertical LinearLayout which contains three rows of elements. The first row is a horizontal LinearLayout which is used to input the account information; the second row is another horizontal LinearLayout which is used to input the password information; and the third row is a login button. All the components used here have been covered in previous chapters, and you should not feel any difficulty to understand.

Next update LoginActivity as code below:

```

class LoginActivity : BaseActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

```

```

        setContentView(R.layout.activity_login)
        login.setOnClickListener {
            val account = accountEdit.text.toString()
            val password = passwordEdit.text.toString()
            // Correct account is admin and password is 123456
            if (account == "admin" && password == "123456") {
                val intent = Intent(this, MainActivity::class.java)
                startActivity(intent)
                finish()
            } else {
                Toast.makeText(this, "account or password is invalid",
                    Toast.LENGTH_SHORT).show()
            }
        }
    }
}

```

Here we implemented a very naïve login screen. First, we let LoginActivity inherit from BaseActivity and make the condition check for the input account and password: if the account is admin and password is 123456, then we consider the login information is successful and start the MainActivity; otherwise, we show login error message to the user.

Thus, you can look on MainActivity as the main screen after successful login as we don't need any extra functionality except forcing logout. Update activity\_main.xml as code below:

```

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

    <Button
        android:id="@+id/forceOffline"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send force offline broadcast" />

</LinearLayout>

```

We simply put a button to trigger forcing logout. Update MainActivity as code below;

```

class MainActivity : BaseActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}

```

```

        forceOffline.setOnClickListener {
            val intent = Intent("com.example.broadcastbestpractice.
FORCE_OFFLINE")
            sendBroadcast(intent)
        }
    }
}

```

It is also very simple. In the highlighted code, we send a broadcast with the value of com.example.broadcastbestpractice.FORCE\_OFFLINE in the button click event handler which is used to notify the app to force logout. This means that the forcing logout logic is not inside the MainActivity but in the BroadcastReceiver that receives this broadcast. Then the forcing logout logic will not be bound to any screen. No matter which screen user is interacting, we just need to send this broadcast to force logout.

Without any doubt, next we need to create a BroadcastReceiver to receive this broadcast. The question is where should we create this receiver? We need to show a dialog to block user interaction with other components inside the BroadcastReceiver. Thus, we cannot use static receiver as there is no way to show dialog in onReceive() for static receivers. We also cannot register a dynamic receiver in every activity.

The solution is quite straightforward, we only need to register a BroadcastReceiver dynamically in BaseActivity as all activities in this app inherits BaseActivity.

Update BaseActivity as code below:

```

open class BaseActivity : AppCompatActivity() {

    lateinit var receiver: ForceOfflineReceiver

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ActivityCollector.addActivity(this)
    }

    override fun onResume() {
        super.onResume()
        val intentFilter = IntentFilter()
        intentFilter.addAction("com.example.broadcastbestpractice.
FORCE_OFFLINE")
        receiver = ForceOfflineReceiver()
        registerReceiver(receiver, intentFilter)
    }

    override fun onPause() {
        super.onPause()
        unregisterReceiver(receiver)
    }
}

```

```
override fun onDestroy() {
    super.onDestroy()
    ActivityCollector.removeActivity(this)
}

inner class ForceOfflineReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        AlertDialog.Builder(context).apply {
            setTitle("Warning")
            setMessage("You are forced to be offline. Please try to login again.")
            setCancelable(false)
            setPositiveButton("OK") { _, _ ->
                ActivityCollector.finishAll() // destroy all activities
                val i = Intent(context, LoginActivity::class.java)
                context.startActivity(i) // restart LoginActivity
            }
            show()
        }
    }
}
```

First let us look at the code in ForceOfflineReceiver, and this time onReceive() doesn't simply show a Toast but adds more business logic. First AlertDialog.Builder is used to create a dialog. Notice that we have to use setCancelable() to disable cancelling the dialog, otherwise user can click Back button to close the dialog and continue using the app. Then use setPositiveButton() to register confirm button. When user clicks OK button, call finishAll() of ActivityCollector to destroy all activities and start LoginActivity.

Next, let us take a look at how to register ForceOfflineReceiver. Code above overrides onResume() and onPause(), and registers and unregisters ForceOfflineReceiver in these two lifecycle methods respectively.

Why we need to do so? Didn't we register in onCreate() and unregister in onDestroy() in BroadcastReceiver before? This is because we need to make sure that only the Activity at the top of the back stack will receive this broadcast, and the activities that are not at the top of the stack shouldn't receive this broadcast. When an activity is no longer at the top of the stack, it will unregister BroadcastReceiver in onPause().

That's all for the business logic of forcing logout, next we need to update AndroidManifest.xml as code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcastbestpractice">
```

```

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".LoginActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
    </activity>
    <activity android:name=".MainActivity">
    </activity>
</application>
</manifest>

```

We just need to make the main Activity to be LoginActivity instead of MainActivity as we don't want user get into the main screen without login first.

Now run the app and login screen will show up first, and then input account and password as shown in Fig. 6.9.

If the account is admin and password is 123456, then click login button and it will open the main screen as shown in Fig. 6.10. Click the send broadcast button and it will send out a broadcast to force logout, and ForceOfflineReceiver will show a dialog to notify user that they are offline now after receiving the broadcast as shown in Fig. 6.11.

Now user cannot interact with any UI components except click OK button to go to the login screen. That is everything we need for forcing logout.

After the best practice section, next is the Kotlin Class time of this chapter.

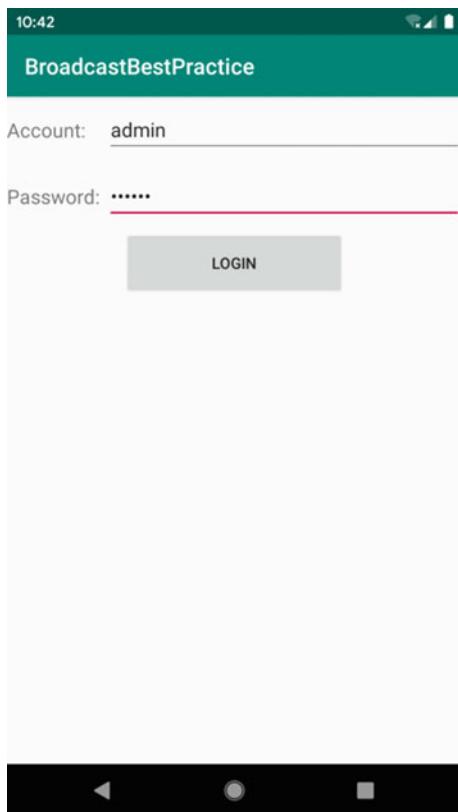
## 6.5 Kotlin Class: Higher-Order Function

Up to now, you have mastered the basics of Kotlin, and starting from this chapter's Kotlin class, I will cover some more advanced topics of Kotlin.

Let us begin with higher-order function.

### 6.5.1 Define Higher-Order Function

Higher-order function is closely related to Lambda expression. In Chap. 2, a quickstart to Kotlin programming, we have learned the basics of Lambda programming and the use of some of the collection-related functional apis, such as map, filter,

**Fig. 6.9** Login screen

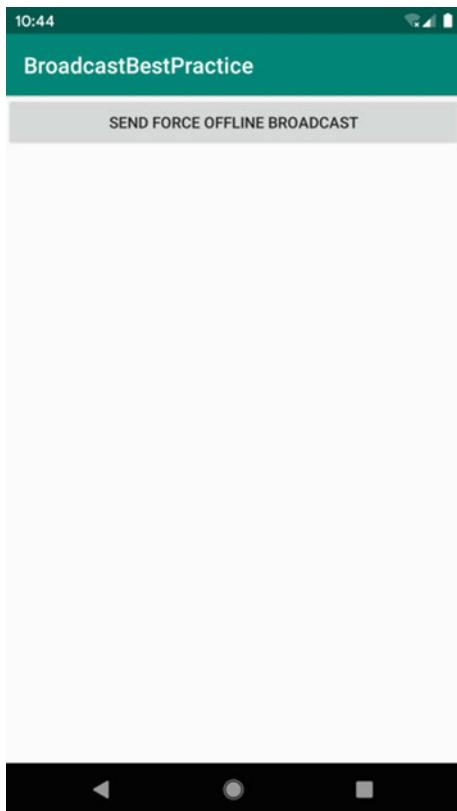
and so on. In the Kotlin Class section in Chap. 3, we learned Kotlin standard functions such as `run`, `apply`, etc.

You should notice that all these functions have something in common: they all require a Lambda expression as param. The functions that take Lambda expression as param are called functional APIs. If you want to define your own functional API, you need higher-order function to do so which is what we will focus on in this section.

First, let us take a look at the definition of higher-order function. If a function takes another function as param or if the return type is another function, then this function is a higher-order function.

It might be difficult for some to understand this definition since how can a function take another function as param? This is related to another concept: function type. We all know that most programming languages have integer, Boolean types for variables, and Kotlin introduced function type. If we use this type as the param or return value in a function, then this function is a higher-order function.

Now let us take a look at how to define a function type. Unlike a normal field type, function type has special syntax, and the basic syntax format is as shown below:

**Fig. 6.10** Main screen

```
(String, Int) -> Unit
```

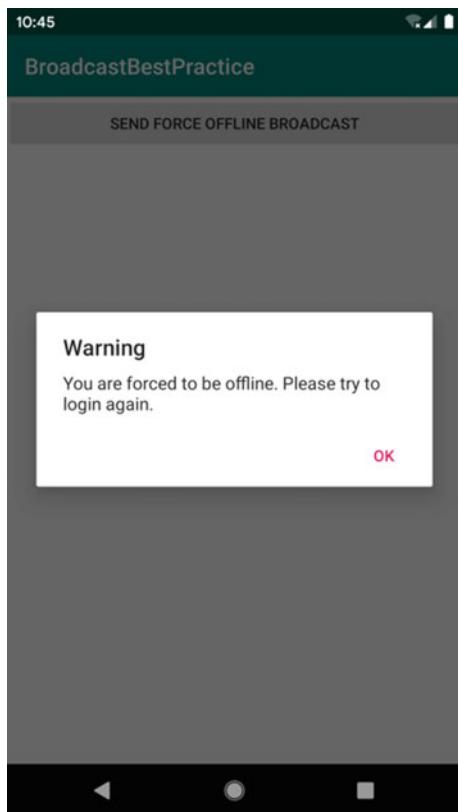
It might look strange at first glance but let me explain it with details, and then you should find it easy to understand.

Since we need to define function type, thus we need to declare what are the data types for params and what is the return value. Thus, the left side of `->` is used to specify the params this function takes, and between the params are commas. If there is no param needed, then a pair of empty parentheses is enough. On the right side of `->` is the return value type, if there is no return value then use `Unit` which is similar to `void` in Java.

Now use the above function type as param or return value to a function, and that function becomes a higher-order function as shown below:

```
fun example(func: (String, Int) -> Unit) {  
    func("hello", 123)  
}
```

**Fig. 6.11** Force logout screen



The example() function takes a function type param, and thus example() is a higher-order function. The syntax to use function type param is similar as using a normal function, we just need to add a pair of parentheses after the param name and pass in the required params for the function type.

We just learned how to define higher-order function, but what is higher-order function used for? Higher-order functions are widely used and, in my opinion, higher-order function allows the function type param to determine the business logic inside it. This means that the same higher-order function can have different business logic and return value with different function type argument. Let us use an example to illustrate this.

I will define a higher-order function with the name num1AndNum2(), and let it take two params of integer type and a function type param. We will operate on the two integers and return the result, and the operation is determined by the function type param.

Create HigherOrderFunction.kt with code below:

```
fun num1AndNum2(num1: Int, num2: Int, operation: (Int, Int) -> Int): Int {
```

```

    val result = operation(num1, num2)
    return result
}

```

This is a very simple higher-order function which may not be super useful in real world, but it is a good example for demonstration purpose. There is no need to explain the first two params, the second param is a function type param that takes two integer param and return value of integer type. In num1AndNum2(), we do not specify any operations. Instead, we pass num1 and num2 to the function type param and return the return value of the function type param.

How do we use this higher-order function? Since num1AndNum2() takes a function type param, thus we need to define the corresponding function. Add the following code in HigherOrderFunction.kt:

```

fun plus(num1: Int, num2: Int): Int {
    return num1 + num2
}

fun minus(num1: Int, num2: Int): Int {
    return num1 - num2
}

```

We define two functions that match the param list and also the return value of the function type param in num1AndNum2(). Among them, plus() will add the two integers together and return the sum, and minus() will calculate the difference and return.

With the functions above, we can call num1AndNum2(). Write the following code in main():

```

fun main() {
    val num1 = 100
    val num2 = 80
    val result1 = num1AndNum2(num1, num2, ::plus)
    val result2 = num1AndNum2(num1, num2, ::minus)
    println("result1 is $result1")
    println("result2 is $result2")
}

```

Notice the way to call num1AndNum2(), the third argument uses ::plus and ::minus syntax. This is a way to reference the functions which means passing plus() and minus() as an argument to num1AndNum2(). Since num1AndNum2() uses the passed in function type argument to determine the specific arithmetic logic, thus we are actually using plus() and minus() to do the operation.

Run the app and the result should be shown as Fig. 6.12.

The result is as what we expected.

```
Run: com.example.uibestpractice.HigherOrde... ×
▶ 🔍 "/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
result1 is 180
result2 is 20
|| ⏪ Process finished with exit code 0
```

**Fig. 6.12** Result of higher-order function

The approach here to reference the function can work as expected, however it would be too complicated if we have to define a function that matches the params list and return value every time we call higher-order function.

Yes, Kotlin does provide other ways to use higher-order function such as Lambda expression, anonymous function, member reference, and so on. Among them, Lambda expression is the most commonly seen and widely used approach to use higher-order functions. This is the topic we want to focus on next.

With Lambda expression, we can update the above as code below:

```
fun main() {
    val num1 = 100
    val num2 = 80
    val result1 = num1AndNum2(num1, num2) { n1, n2 ->
        n1 + n2
    }
    val result2 = num1AndNum2(num1, num2) { n1, n2 ->
        n1 - n2
    }
    println("result1 is $result1")
    println("result2 is $result2")
}
```

We have learned the syntax of Lambda expression in Sect. 2.6.2, thus the above code should not look strange to you. You should find that Lambda expression can also have exactly the same param list and return value (the last line of Lambda expression will be evaluated, and the value will be returned as return value) but more concise.

Now you can delete plus() and minus(). Run the code again, you should see exactly the same result.

Let us continue exploring higher-order functions. Recall the apply function in Chap. 3, it can provide context for Lambda expression, and when there is need to call multiple methods of the same object, apply will make the code much more concise. StringBuilder is a good example. Next, let us use higher-order function to implement something similar to it.

Update HigherOrderFunction.kt by adding the following code:

```
fun StringBuilder.build(block: StringBuilder.() -> Unit) :
StringBuilder {
    block()
}
```

```

        return this
    }
}

```

The above code defines an extension function with the name build which takes a function type param and the return value is also StringBuilder.

Notice the difference of declaring the param: code above adds StringBuilder before the function type. This actually is the complete form of higher-order function syntax. By adding ClassName before function type means this function type is defined in this specific class.

What is the benefit of defining this function type in StringBuilder class? This is used to provide the context of StringBuilder for the Lambda expression when we call build function. And this is how apply function gets the context.

Now we can use the build function which we just created to simply the way StringBuilder to construct the string. Use the previous fruit examples again, and we can have the following code:

```

fun main() {
    val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
    val result = StringBuilder().build {
        append("Start eating fruits.\n")
        for (fruit in list) {
            append(fruit).append("\n")
        }
        append("Ate all fruits.")
    }
    println(result.toString())
}

```

We can see that to use build is the same as using apply except that build function can only be applied to StringBuilder class and apply function can be applied to all classes. In order to implement the same behavior as apply, we need to use generic related knowledge which will be introduced in Chap. 8.

Now you should know the basics about higher-order function, and let's take a look at some more advanced topics.

### 6.5.2 *Inline Functions*

Higher-order functions are widely used. However, do you know the mechanism behind it? Of course, this is not something every developer need to know. But in order to understand inline function—our next topic—better, let us briefly take a look at the mechanism of higher-order functions.

Use the num1AndNum2() as an example, as shown in code below:

```

fun num1AndNum2(num1: Int, num2: Int, operation: (Int, Int) -> Int) :
Int {
}

```

```
    val result = operation(num1, num2)
    return result
}

fun main() {
    val num1 = 100
    val num2 = 80
    val result = num1AndNum2(num1, num2) { n1, n2 ->
        n1 + n2
    }
}
```

The above code calls num1AndNum2() and uses Lambda expression to sum the two integer arguments. This is the basic use of higher-order function and is easy to understand. However, we all know that Kotlin will be complied to Java bytecode, and Java has no such concept as higher-order function at all.

What Kotlin has done to make Java support higher-order function? Thanks to the powerful Kotlin compiler, it will convert the higher-order function syntax to syntax that Java supports. The above Kotlin code will be converted to something like below in Java:

```
public static int num1AndNum2(int num1, int num2, Function operation)
{
    int result = (int) operation.invoke(num1, num2);
    return result;
}

public static void main() {
    int num1 = 100;
    int num2 = 80;
    int result = num1AndNum2(num1, num2, new Function() {
        @Override
        public Integer invoke(Integer n1, Integer n2) {
            return n1 + n2;
        }
    });
}
```

I made some changes to the code for readability, so it is not exactly the same as the converted code. We can see that the third param of num1AndNum2() becomes a Function interface, which is a Kotin internal interface with a function invoke() that needs implementation. num1AndNum2() actually calls the invoke() of Function interface and passes num1 and num2 to the interface implementation.

When calling num1AndNum2(), the previous Lambda expression becomes an anonymous implementation of the Function interface and implements the logic of  $n1 + n2$  inside invoke() then return the result.

This is what happens behind the Kotlin higher-order function. Basically, the Lambda expression has been turned to anonymous class under the hood. This

```

inline fun num1AndNum2(num1: Int, num2: Int, operation: (Int, Int) -> Int): Int {
    val result = [operation(num1, num2)]←
        return result
}

fun main() {
    val num1 = 100
    val num2 = 80
    val result = num1AndNum2(num1, num2) { n1, n2 ->
        [n1 + n2]
    }
}

```

**Fig. 6.13** First step for replacement

means that every time we call Lambda expression, an instance of anonymous class will be created which leads to extra cost on memory and performance.

In order to solve this problem, Kotlin provides inline function which can eliminate the runtime cost of Lambda expressions.

To use inline function, we simply add the **inline** modifier before the higher-order function as shown in code below:

```

inline fun num1AndNum2(num1: Int, num2: Int, operation: (Int, Int) ->
Int): Int {
    val result = operation(num1, num2)
    return result
}

```

Then what is the mechanism of inline functions? It is actually not complicated at all. Kotlin compiler will put the code inside inline functions to the place that is calling it, which means there is no extra cost at run time.

It might be difficult to understand the above statement, let me use images to illustrate this process.

First, Kotlin will put the code inside the Lambda expression to the place that the function type is getting used as shown in Fig. 6.13.

Next, all the code inside the inline function will be placed to where function is called, as shown in Fig. 6.14.

The final result is as shown in Fig. 6.15.

Because of the mechanism above, inline function could eliminate the runtime extra cost of Lambda expression.

```

inline fun num1AndNum2(num1: Int, num2: Int): Int {
    val result = num1 + num2
    return result
}

fun main() {
    val num1 = 100
    val num2 = 80
    val result = num1AndNum2(num1, num2)
}

```

**Fig. 6.14** Second step for replacement**Fig. 6.15** Final result of replacement

```

fun main() {
    val num1 = 100
    val num2 = 80
    val result = num1 + num2
}

```

### 6.5.3 *Noinline and Crossinline*

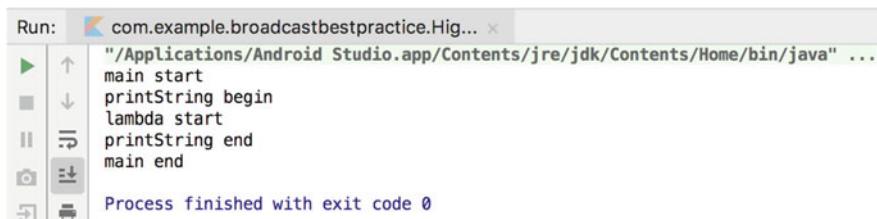
Next, we will cover some more special cases. For example, if a higher-order function takes two or more function type params, and if we add inline modifier to the function, then Kotlin compiler will inline all the referenced Lambda expressions.

However, what if we only want to inline one of the Lambda expression? Then noinline modifier comes into play, as shown below:

```
inline fun inlineTest(block1: () -> Unit, noinline block2: () -> Unit) {
}
```

Code above uses inline modifier for inlineTest(). Without noinline modifier, the Lambda expressions that block1 and block2 function type params reference will be inlined. After using noinline before block2, only the Lambda expression referenced by block1 will be inlined. This is what noinline will do.

We mentioned the benefits of inline functions, why we need noinline in Kotlin? This is because the inline function type params will be replaced at compile type, which makes them not to have the property of param. Noinline function type param can be assigned to other functions because it is a real param while inline function type param can only be assigned to another inline function which is its biggest limitation.



**Fig. 6.16** Result of local return

There is also another big difference between inline and noinline function. That is the Lambda expression for inline function can use return keyword to return the value while noinline function only allows local return. Let us use an example to illustrate this.

```

fun printString(str: String, block: (String) -> Unit) {
    println("printString begin")
    block(str)
    println("printString end")
}

fun main() {
    println("main start")
    val str = ""
    printString(str) { s ->
        println("lambda start")
        if (s.isEmpty()) return@printString
        println(s)
        println("lambda end")
    }
    println("main end")
}

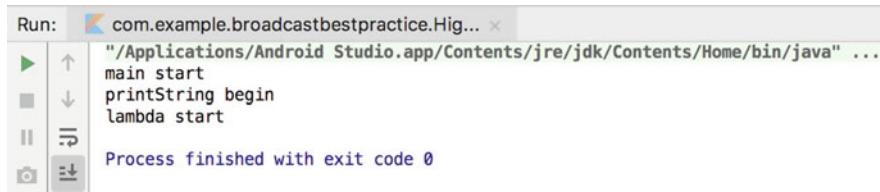
```

Code above defines a higher-order function named `printString()` which will print the string argument in the Lambda expression. If the string argument is null then it won't print anything. Notice that `return` is not allowed in Lambda expression. Code here uses `return@printString` to do local return and does not execute the rest of the code in Lambda expression.

Now let us make `str` to be an empty string and run the code, the result should be as shown in Fig. 6.16.

As you can see, the statements after `return@printString` in the Lambda expression did not get executed, and other logs were printed out as expected, which means that `return@printString` indeed can only do local return.

However, if we declare the `printString()` to an inline function, it will have different behavior as shown below:



**Fig. 6.17** Result of non-local return

```
inline fun printString(str: String, block: (String) -> Unit) {
    println("printString begin")
    block(str)
    println("printString end")
}

fun main() {
    println("main start")
    val str = ""
    printString(str) { s ->
        println("lambda start")
        if (s.isEmpty()) return
        println(s)
        println("lambda end")
    }
    println("main end")
}
```

As `printString()` becomes inline function, we can use `return` in Lambda expression now. The `return` statement is for the out layer function that is calling `printString()` which is `main()`. Refer to the inline function code replacement procedure in last section, if you find it hard to understand.

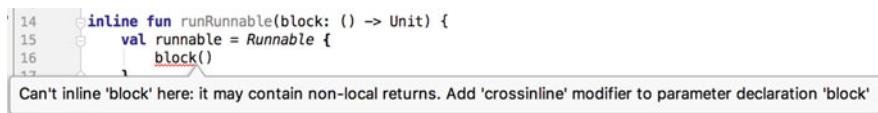
Run the app again and the result is as shown in Fig. 6.17.

As expected, all the code for `main()` and `printString()` stop execution after the `return` statement.

Declare higher-order function to be inline function is a recommended practice and as a matter of fact, most of the higher-order functions can be declared as inline functions with a few exceptions. Observe the following code example:

```
inline fun runRunnable(block: () -> Unit) {
    val runnable = Runnable {
        block()
    }
    runnable.run()
}
```

The code above works without `inline` modifier, however after adding `inline` keyword, Android Studio will show error as shown in Fig. 6.18.



**Fig. 6.18** Error when using inline function

The reason behind this error is a little bit complicated. First, in `runRunnable()`, we create an object of `Runnable` and in the Lambda expression of `Runnable`, we call the function type argument. However, during compilation, Lambda expression will be converted to anonymous class which means that the above code actually calls the function type argument in the anonymous class.

The Lambda expression referenced by the inline function allows the return keyword to be used to return the function. However, since we are calling the function type parameter in an anonymous class, it is impossible to return the outer call function. At most, we can only return the function call in an anonymous class.

In other words, if we create another Lambda expression or use anonymous class implementation in higher-order function and use function type param, then inline function will throw error.

So, we cannot inline function in this scenario? That is not the case neither. We can use `crossinline` modifier to solve the problem.

```
inline fun runRunnable(crossinline block: () -> Unit) {
    val runnable = Runnable {
        block()
    }
    runnable.run()
}
```

As you can see, after adding the `crossinline` modifier, code can compile successfully.

So, what does `crossinline` modifier do? As we mentioned before, the error in Fig. 6.18 is thrown because return is allowed in Lambda expression in inline function, which conflicts with the rule that higher-order anonymous class implementation does not allow return keyword. The `Crossinline` keyword is used as a contract to ensure that the return keyword is never used in the inline function's Lambda expression, so that the conflict does not exist and the problem is neatly solved.

Apparently, with `crossinline`, we cannot use return in Lambda expression when calling `runRunnable` function while we can still use `return@runRunnable` for local return. Overall, except the difference of using return keyword, `crossinline` reserves all the other features of inline function.

These are almost all the important topics in higher-order functions. I hope you can be proficient in these topics since a lot of the knowledge related to Lambda and higher-order functions are based on the topics introduced in this chapter.

After the Kotlin Class, we will start a special section. As you know, most of the high-quality projects are not done by a single developer but by a team. Then it

becomes important for developers to sync their code changes. Thus, version control tools were introduced. The most commonly used version control tools are SVN and Git. I will cover Git in this book in detail in different chapters. Now, let us take a look at the basic use of Git.

## 6.6 Git Time: The First Look of Version Control Tools

Git is an open-source distributed version control tool and its inventor is the famous Linux initiator Linus Torvalds. Git was initially invented to better manage the Linux kernel code but is widely used in projects of all sizes globally now. This is our first class of Git and will cover some basic use of it. Let us start with installing Git.

### 6.6.1 *Git Installation*

Since Git and Linux share the same author, you probably can tell that it is easiest to install Git in Linux. For instance, if you are using Ubuntu, you just need to open terminal and type in `sud apt-get install git`, press Enter and type in the password then Git will be installed.

Installing Git in Mac is similar. If you've installed Homebrew, you just need to type in `brew install git` in terminal to install.

In Windows, it will be a little bit more complicated. We need to install the installer of Git. Visit Git for Windows official website which should look like Fig. 6.19.

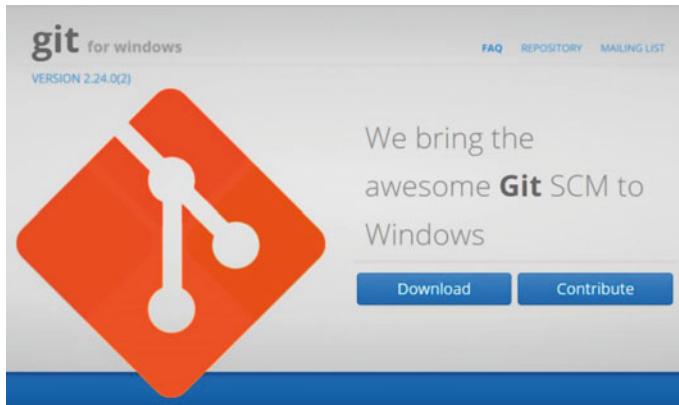
Click Download to download the installer. After downloading, double click installer to install, click Next all the way to finish installation.

### 6.6.2 *Create Code Repository*

Though you can install Git with GUI in Windows and Android Studio also supports interacting with Git with GUI, I'd recommend you not to do so. This is because Git commands are what you should master. No matter which system you're on, Git command is universal. GUI is just a way to potentially improve work efficiency on the precondition that you are proficient in using Git commands.

Now let us try to use Git commands. If you're on Linux or Mac, open the terminal; if you're on Windows then find Git Bash from Start and open it.

First you need to configure your identity so that Git will know who is committing the code. The commands are as follows:



**Fig. 6.19** Git for windows official website

```
[guolin@MacBook-Pro:~ guolin$ git config --global user.name
Tony
[guolin@MacBook-Pro:~ guolin$ git config --global user.email
tony@gmail.com
guolin@MacBook-Pro:~ guolin$ ]
```

**Fig. 6.20** Verify Git username and email

```
[guolin@MacBook-Pro:~ guolin$ cd AndroidStudioProjects/AndroidFirstLine/BroadcastBestPractice/
guolin@MacBook-Pro:BroadcastBestPractice guolin$ ]
```

**Fig. 6.21** Get into directory of BroadcastBestPractice project

```
git config --global user.name "Tony"
git config --global user.email "tony@gmail.com"
```

After configuration, you can use the same commands without the name nor the email to verify if the configuration is successful or not as shown in Fig. 6.20.

Then we can start to create code repository. The repository has the information for version control. All the locally committed code will be inside the repo, if needed will be pushed to remote repo.

Let us try to create a code repo for BroadcastBestPractice project. Get into the directory of BroadcastBestPractice as shown in Fig. 6.21.

Then type in the following command:

```
git init
```

Yes, with this simple command we can finish creating the code repo as shown in Fig. 6.22.

```
[guolin@MacBook-Pro:BroadcastBestPractice guolin$ git init
Initialized empty Git repository in /Users/guolin/AndroidStudioProjects/AndroidFirstLine/BroadcastBestPractice/.git/
guolin@MacBook-Pro:BroadcastBestPractice guolin$ ]
```

Fig. 6.22 Create code repository

```
[guolin@MacBook-Pro:BroadcastBestPractice guolin$ ls -al
total 72
drwxr-xr-x 16 guolin staff 512 7 4 23:02 .
drwxr-xr-x 16 guolin staff 512 6 25 21:07 ..
drwxr-xr-x 9 guolin staff 288 7 4 23:02 .git
-rw-r--r-- 1 guolin staff 203 6 25 21:07 .gitignore
drwxr-xr-x 5 guolin staff 160 6 25 21:08 .gradle
drwxr-xr-x 11 guolin staff 352 7 4 21:57 .idea
-rw-r--r-- 1 guolin staff 898 6 25 21:18 BroadcastBestPractice.iml
drwxr-xr-x 9 guolin staff 288 7 4 21:48 app
drwxr-xr-x 3 guolin staff 96 6 25 22:41 build
-rw-r--r-- 1 guolin staff 661 6 25 21:07 build.gradle
drwxr-xr-x 3 guolin staff 96 6 25 21:07 gradle
-rw-r--r-- 1 guolin staff 1163 6 25 21:16 gradle.properties
-rwxr--r-- 1 guolin staff 5296 6 25 21:07 gradlew
-rw-r--r-- 1 guolin staff 2260 6 25 21:07 gradlew.bat
-rw-r--r-- 1 guolin staff 436 6 25 21:07 local.properties
-rw-r--r-- 1 guolin staff 15 6 25 21:07 settings.gradle
guolin@MacBook-Pro:BroadcastBestPractice guolin$ ]
```

Fig. 6.23 Check .git directory

After creating the repo, a hidden .git repository will be generated under the root directory of BroadcastBestPractice project which is used to record all the local Git actions. You can use ls -al to take a look as shown in Fig. 6.23.

If you want to delete the local repository, just delete this directory.

### 6.6.3 Commit Local Code

After creating the code repository, you can commit your code which is very simple—only add and commit commands are needed. add is used to add the code that is going to be committed, and commit is to actually commit the code. For example, if we want to add build.gradle file, we can type in the following command:

```
git add build.gradle
```

This is to add a single file. What if we want to add a folder? You can just add the folder name after add. For instance, to add all the files under app folder, type in the following command:

```
git add app
```

Even this will be too much work if there are lots of folders. Is there any way to add all the files? Then we just need to add a dot after add as shown below:

```
git add .
```

Now, all the files in the BroadcastBestPractice folder are added which means they will be tracked by Git for changes. We can commit the code with the following command:

```
git commit -m "First commit."
```

Notice that we need to use -m param to add the commit message. Omit this will cause command to fail. With the above command, all the code will be committed.

That is everything I want to cover for Git in this section. Though it is not a long section but you should've already mastered the basic use of Git. I will cover more Git related topics in chapters later. It is time to summarize this chapter.

## 6.7 Summary and Comment

In this chapter, we mainly studied Android's broadcast mechanism in depth, not only understanding the theoretical knowledge of broadcast, but also mastering the use of receiving broadcast, sending custom broadcast and local broadcast. BroadcastReceiver is one of the four main components of Android. So now you have already learned two of the four main components.

In the Best Practice section, you applied the broadcast knowledge in this chapter together with techniques learned from previous chapters through example which should help you understand the related topics better. The Kotlin Class in this chapter also has rich content. Higher-order function is very important, and you should make sure you're proficient in this topic.

We added an extra section in this chapter which is Git Time. In this section, I gave an introduction to Git for version control purpose, and I will cover more in later chapters.

Next chapter, we should learn another one of the four main components of Android, however since we need to know data persistence before learning ContentProvider, thus we will discuss this topic in next chapter.

# Chapter 7

## Data Persistence



When we interact with an app, we are actually interacting with the data behind the scene. For instance, when we use Messenger, NewsBreak, and Twitter, we are interacting with corresponding content which is the data in the app. Without data, apps are useless. Then where does the data come from? Most of the data are generated by users these days; for example, sending tweets and commenting news. When they are doing all of these behaviors, they are generating data.

Among the examples we used in previous chapters, some of them also used data. For instance, the chat content of the chat UI app in Chap. 4, and login information for Login UI in Chap. 6.

All of these data are temporary data, which means that these data are stored in the memory, and will get lost when app exits, or memory gets recycled. This is unacceptable for some important data, for instance, nobody can accept that the tweet just sent will disappear after refreshing the app. How to make sure that the important data will persist? Then we need to use data persistence related knowledge.

### 7.1 Introduction to Data Persistence

Data persistence is to store the temporary data in the memory into data storage devices that can store the data for longer term, such that even when the phone or computer is off, data will not get lost. The data in memory is temporary and ephemeral, but data in the data storage devices is persistent. Data persistence mechanism can let data switch between temporary and persistent state.

Data persistence is widely used in system design, and this section is going to discuss data persistence in Android. Android provides three ways for data persistence: file, SharedPreferences, and database.

Let me cover these approaches in detail.

## 7.2 Persisting Through File

Persisting data in file is the most basic data persistence method in Android. It will not format the content; instead, all the data will be stored into the file without any formatting, thus it is a good choice for storing some simple text data or binary data. If you want to store data with more complex structures, then you would need to define the data format in order to facilitate parsing the data from files later.

Now let us take a look at how to persist data through file in Android.

### 7.2.1 Persisting Data in File

Context class provides `openFileOutput()` to store the data in the specified file. This method will take two params. The first param is the file name which is used for creating the file. Notice that the file name cannot include the file path because all the files are stored under `/data/data/<package name>/files/`. The second param is the operation mode to the file which has two values: `MODE_PRIVATE` and `MODE_APPEND`. The default is `MODE_PRIVATE` which means that the new data will overwrite the original file's content if it exists. `MODE_APPEND` means that the new data will be appended at the end of the file if it exists, if not a new file will be created. There used to be another two modes, we could choose from: `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE`, which allow other apps to read and write the current app's file, and you can easily tell it is very dangerous. Thus these two modes got deprecated in Android 4.2.

`openFileOutput()` will return an instance of `FileOutputStream`. After acquiring this object, we can use Java stream to write data into the file. The following code demonstrates how to save some text into a file:

```
fun save(inputText: String) {
    try {
        val output = openFileOutput("data", Context.MODE_PRIVATE)
        val writer = BufferedWriter(OutputStreamWriter(output))
        writer.use {
            it.write(inputText)
        }
    } catch (e: IOException) {
        e.printStackTrace()
    }
}
```

If you're familiar with Java stream, the code above should be easy to understand. It gets an object of `FileOutputStream` through `openFileOutput()`, and it uses this object to construct an object of `OutputStreamWriter`, and then uses the new object to construct an object of `BufferedWriter`. Then we can use the `BufferedWriter` object to write texts into files.

Notice here, it uses an internal extension function use which will automatically close the stream of outer layer after the outer layer of the Lambda expression. Thus there is no need for us to write the finally statement to close the stream manually.

Now, let us use a more complete example to learn how to use file to store data in Android project. First, create FilePersistenceTest project and modify activity\_main.xml as code below:

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

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type something here"
        />

</LinearLayout>
```

The code above just adds an EditText view for text input.

Now you can run the app, and there should be a text input view. Type something in the text input view and press Back button, then the content you just typed in will be lost, since this is just some temporary data, and its corresponding memory will be recycled after Activity gets destroyed. What we need to do is to put the content into a file before it gets recycled. Modify MainActivity as code below:

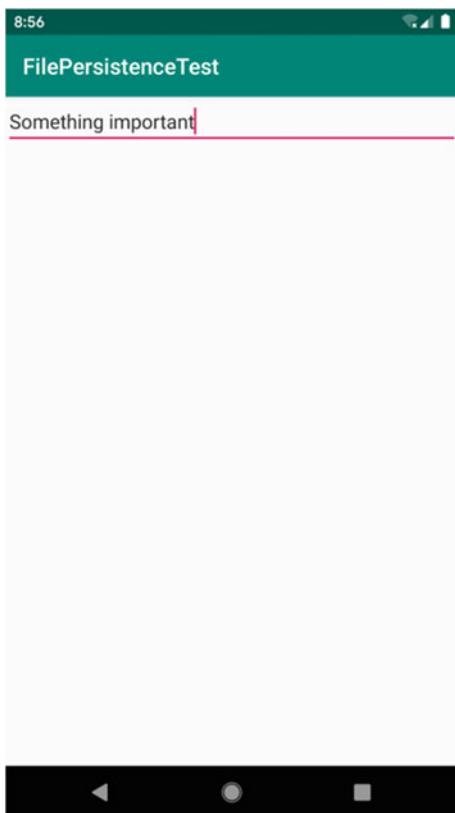
```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    override fun onDestroy() {
        super.onDestroy()
        val inputText = editText.text.toString()
        save(inputText)
    }

    private fun save(inputText: String) {
        try {
            val output = openFileOutput("data", Context.MODE_PRIVATE)
            val writer = BufferedWriter(OutputStreamWriter(output))
            writer.use {
                it.write(inputText)
            }
        }
    }
}
```

**Fig. 7.1** Input in EditText view

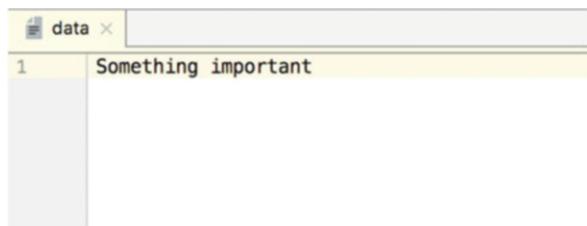


```
        } catch (e: IOException) {
            e.printStackTrace()
        }
    }
```

As you can see, we override `onDestroy()` to make sure that `save()` can be called before Activity gets destroyed. In `onDestroy()`, we get the content in the EditText view and call `save()` to save the content into file with the name “data.” The code in `save()` is similar to the previous example, so I will not explain again. Run the app again and type something in EditText view as shown in Fig. 7.1.

Press Back button to close the app, and the input content should be stored in a data file. How to verify this? We can use Device File Explorer tool to take a look. It is at bottom right of the Android Studio. If you cannot find it, you can also use `Ctrl + Shift + A` (`command + shift + A` in Mac) shortcut to open the search bar, and type in Device File Explorer to find it.

| Name                                | Permissions       | Date                    | Size        |
|-------------------------------------|-------------------|-------------------------|-------------|
| ▶ com.example.activitylifecycletest | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ com.example.activitytest          | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ com.example.broadcastbestpractice | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ com.example.broadcasttest         | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▼ com.example.filepersistencetest   | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ cache                             | drwxrws--x        | 2019-07-06 20:41        | 4 KB        |
| ▶ code_cache                        | drwxrws--x        | 2019-07-06 20:41        | 4 KB        |
| ▼ files                             | drwxrwx--x        | 2019-07-06 20:57        | 4 KB        |
| <b>data</b>                         | <b>-rw-rw----</b> | <b>2019-07-06 20:57</b> | <b>19 B</b> |
| ▶ com.example.fragmentbestpractice  | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ com.example.fragmenttest          | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |
| ▶ com.example.uiwidgettest          | drwxrwx--x        | 2019-06-12 07:47        | 4 KB        |

**Fig. 7.2** Generated data file**Fig. 7.3** Content inside data file

This tool is essentially a device file browser, find `/data/data/com.example.filepersistencetest/files` folder, and you should find a file with the name “data” in it as shown in Fig. 7.2.

Double click this file to see the content inside as shown in Fig. 7.3.

This verifies that the content in the EditText view has been successfully stored in the file.

However, it is not enough to only store the data, we also need to be able to recover the data and put it back to the EditText view when the app starts next time. Thus, in the next section, let us learn how to read data from file.

### 7.2.2 *Read Data from File*

Context class provides `openFileInput()` to read data from file just like writing data in the file. This method is simpler compared with `openFileOutput()` as it only takes one param, which is the file name that will be read, and then system will go to `/data/data/<package name>/files/` folder to load this file and return an instance of `FileInputStream`. Once we acquire this object, we can use stream to read the data.

The code below demonstrates how to read text data from file:

```

fun load(): String {
    val content = StringBuilder()
    try {
        val input = openFileInput("data")
        val reader = BufferedReader(InputStreamReader(input))
        reader.use {
            reader.forEachLine {
                content.append(it)
            }
        }
    } catch (e: IOException) {
        e.printStackTrace()
    }
    return content.toString()
}

```

First, the code above gets an instance of FileInputStream through openFileInput(), then use this instance to create an instance of InputStreamReader; after that, use this instance to create an instance of BufferedReader, with this object we can read the data in the file line by line and append them into the StringBuilder instance. Then it returns what has been read.

Notice that Kotlin internal extension function forEachLine is used here to pass each line in the file to the Lambda expression, which has the appending logic.

Now that you have the knowledge of reading data from files, let us continue to improve the previous example so that after restarting the app, EditText view still has the content we typed in from last time. Modify MainActivity as code below:

```

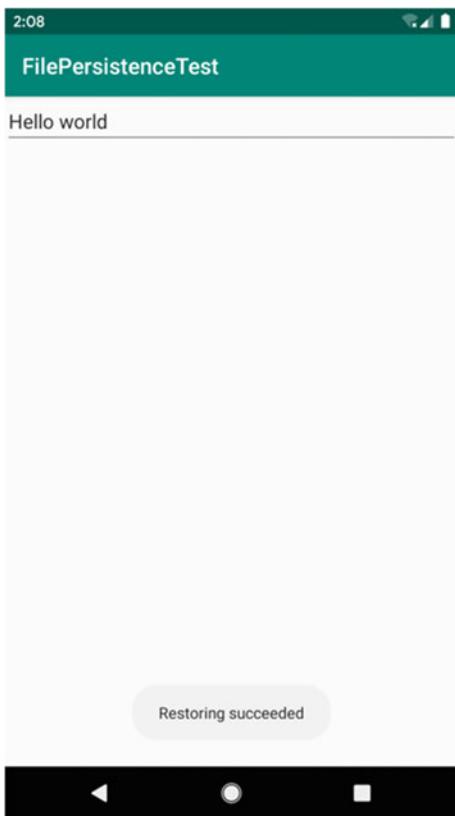
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val inputText = load()
        if (inputText.isNotEmpty()) {
            editText.setText(inputText)
            editText.setSelection(inputText.length)
            Toast.makeText(this, "Restoring succeeded", Toast.
LENGTH_SHORT).show()
        }
    }

    private fun load(): String {
        val content = StringBuilder()
        try {
            val input = openFileInput("data")
            val reader = BufferedReader(InputStreamReader(input))
            reader.use {
                reader.forEachLine {
                    content.append(it)
                }
            }
        } catch (e: IOException) {
            e.printStackTrace()
        }
        return content.toString()
    }
}

```

**Fig. 7.4** Recover the saved content



```
        }
    } catch (e: IOException) {
        e.printStackTrace()
    }
    return content.toString()
}
...
}
```

Here, the idea is simple. In `onCreate()`, we call `load()` to read the text content in the file, and if the content is not empty, we call `setText()` of `EditText` to show the text content in the `EditText` view, and then use `setSelection()` to move the cursor to the end of the text, in order to allow user to continue typing, and then we pop out a toast message showing the data recovered successfully. We've already covered the details of `load()` previously so we won't repeat here.

Now, run the app again and the previous Something important string will fill the `EditText` view, continue typing something like "Hello World," then press Back button to exit the app. Restart the app, you would see the new content getting saved, as shown in Fig. 7.4.

That is everything for storing data through file. The `openFileInput()` and `openFileOutput()` provided by the `Context` class are key concepts for this. Then we just need to use streams to read and write.

However, as I mentioned before, file is not a good choice to store data with more complicated structures. Thus, let us take a look at another data persistence mechanism in Android, which is simpler than storing data through file, and it can provide convenient methods to read and write certain types of data.

## 7.3 SharedPreferences

Unlike storing data through file, `SharedPreferences` uses key-value pairs to store data. This means that when we try to save the data, we need to provide a key for the data. When we read the data, we can get the corresponding value by the key. `SharedPreferences` supports miscellaneous types of data. If the data is saved as integer, then read operation will return integer; if the data is saved as string, then read operation will return string.

Now, you may feel that using `SharedPreferences` to persist data is more convenient than using files. Let us take a look at how to use it.

### 7.3.1 *Save Data in SharedPreferences*

To use `SharedPreferences` to store data, we need to get an instance of `SharedPreferences` first. Android mainly provides two ways to get the instance of `SharedPreferences`.

#### 1. `getSharedPreferences()` in Context Class

This method takes two params. The first param is used to specify the file name of the `SharedPreferences`. If the specified file does not exist, then the corresponding file will be created. `SharedPreferences` files are under the path `/data/data/<package name>/shared_prefs/`. The second param is used to specify the operation mode which has only one choice—`MODE_PRIVATE`. This mode is equivalent to passing `0` directly, which means that only the current application can read and write this `SharedPreferences`. All of the other modes have been deprecated; `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` got deprecated since Android 4.2, `MODE_MULTI_PROCESS` got deprecated since Android 6.0.

#### 2. `getPreferences()` in Activity

This method is similar to the `getSharedPreferences()` in Context except that it only takes one operation mode, because this method will use the current Activity's class name as the name of the `SharedPreferences` file.

After acquiring an instance of SharedPreferences, we can save data in SharePrefrences file, and this can be broken down into the following three steps.

1. Get an instance of SharedPreferences.Editor through edit() of SharedPreferences object.
2. Put data in the SharedPreferences.Editor object. For instance, use putBoolean() to put Boolean type data, use putString() to put the string type data, etc.
3. Use apply() to commit the data, in order to complete data saving action.

That's enough for the introduction of theories, and let us use an example to experiment with SharedPreferences. Create SharedPreferencesTest project and modify activity\_main.xml as code below:

```
<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/saveButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Save Data"
        />

</LinearLayout>
```

Here we don't do anything complicated, but simply place a button for storing some data into SharedPreferences file. Then modify MainActivity as code below:

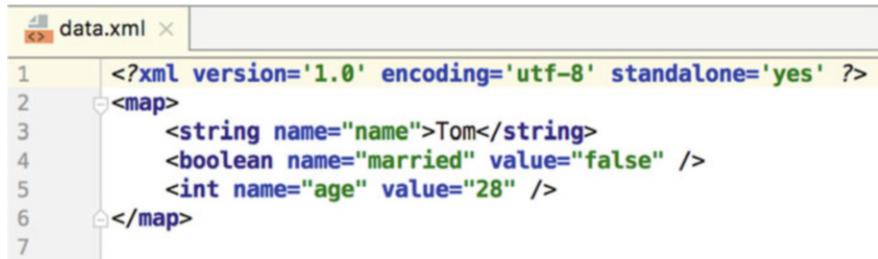
```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        saveButton.setOnClickListener {
            val editor = getSharedPreferences("data", Context.
                MODE_PRIVATE).edit()
            editor.putString("name", "Tom")
            editor.putInt("age", 28)
            editor.putBoolean("married", false)
            editor.apply()
        }
    }
}
```

As you can see, the code above registers a click event for the button, set the SharedPreferences file name "data" through getSharedPreferences() inside the click

| Name                                | Permissions | Date             | Size  |
|-------------------------------------|-------------|------------------|-------|
| ▶ com.example.broadcasttest         | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ com.example.filepersistencetest   | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ com.example.fragmentbestpracti    | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ com.example.fragmenttest          | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▼ com.example.sharedpreferencestest | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| cache                               | drwxrws--x  | 2019-07-07 18:56 | 4 KB  |
| code_cache                          | drwxrws--x  | 2019-07-07 18:56 | 4 KB  |
| ▼ shared_prefs                      | drwxrwx--x  | 2019-07-07 18:57 | 4 KB  |
| data.xml                            | -rw-rw----  | 2019-07-07 18:57 | 186 B |
| com.example.uwidgettest             | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |

**Fig. 7.5** Generated data.xml file



```

data.xml x
1  <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2  <map>
3      <string name="name">Tom</string>
4      <boolean name="married" value="false" />
5      <int name="age" value="28" />
6  </map>
7

```

**Fig. 7.6** Content in data.xml

event, and get an instance of SharedPreferences.Editor. After that, three different types of data have been put into this object. These operations get committed through apply().

Really simple, isn't it? Now, run the app. In the main screen of the app, click "Save Data" button and it should save the data successfully. To verify that, we can use Device File Explorer. Open Device File Explorer, and under /data/data/com.example.sharedpreferencestest/shared\_prefs/ we can see data.xml file as shown in Fig. 7.5.

Double click to open this file and the content should be as shown in Fig. 7.6.

We can see that the data we tried to add in the button click event has been saved successfully and SharedPreferences file uses XML format.

Next, we will take a look at how to read the data from SharedPreferences file.

### 7.3.2 *Read Data from SharedPreferences*

You may feel that saving data with SharedPreferences is very easy; fortunately, reading data from SharedPreferences is even easier. The SharedPreferences object provides a series of get methods to read the data, and each of the get method has its corresponding put method in SharedPreferences.Editor. For instance, we can use getBoolean() to read Boolean data and use getString() to read the string data. All of the get methods will take two params: the first param is the key, which is corresponding to the key that is used when saving the data; the second param is the default value, which will be returned if no data can be found with the key passed in.

Let us use an example to experiment with. We can continue working on SharedPreferencesTest project. Modify activity\_main.xml as code below:

```
<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/saveButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Save Data"
        />

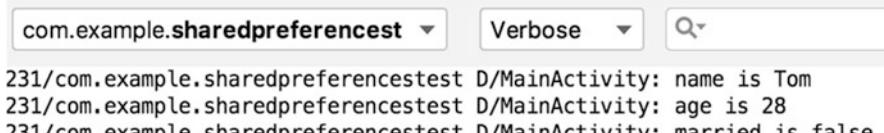
    <Button
        android:id="@+id/restoreButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Restore Data"
        />

</LinearLayout>
```

The highlighted code adds a button to recover the data, and we expect to click this button to read the data from SharedPreferences. Modify MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        restoreButton.setOnClickListener {
            val prefs = getSharedPreferences("data", Context.MODE_PRIVATE)
            val name = prefs.getString("name", "")
        }
    }
}
```



```
com.example.sharedpreferencestest Verbose
231/com.example.sharedpreferencestest D/MainActivity: name is Tom
231/com.example.sharedpreferencestest D/MainActivity: age is 28
231/com.example.sharedpreferencestest D/MainActivity: married is false
```

**Fig. 7.7** Print the Data Saved in data.xml

```
    val age = prefs.getInt("age", 0)
    val married = prefs.getBoolean("married", false)
    Log.d("MainActivity", "name is $name")
    Log.d("MainActivity", "age is $age")
    Log.d("MainActivity", "married is $married")
}
}
```

As you can see, the highlighted code gets an object of SharedPreferences by getting SharedPreferences() in the click event, and then calling getString(), getInt(), and getBoolean() inside it to get the saved name, age, and marriage status. If there is no corresponding values, default values will be used. These values will be logged.

Run the app again and click “Restore data” button, then Logcat should show something as shown in Fig. 7.7.

All the saved data has been successfully read! Through this example, we have learned everything about SharedPreferences. It is much simpler compared with storing data through file and can be applied to more scenarios—for example, lots of app preference implementations use SharedPreferences. In the next section, we will implement the feature of remembering password, in order to help you understand SharedPreferences better.

### 7.3.3 *Implement Remembering Password*

We do not need to start from scratch, as we have already implemented the login UI from last chapter, so we can just reuse it here. Open the BroadcastBestPractice project and edit the login UI layout. Modify activity\_login.xml as code below:

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

```

```
<LinearLayout  
    android:orientation="horizontal"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content">  
  
    <CheckBox  
        android:id="@+id/rememberPass"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content" />  
  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:textSize="18sp"  
        android:text="Remember password" />  
  
</LinearLayout>  
  
<Button  
    android:id="@+id/login"  
    android:layout_width="match_parent"  
    android:layout_height="60dp"  
    android:text="Login" />  
  
</LinearLayout>
```

The code above uses a new widget: CheckBox, which will be used to specify if the app will remember the password or not.

Modify LoginActivity as code below:

```
class LoginActivity : BaseActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_login)  
        val prefs = getPreferences(Context.MODE_PRIVATE)  
        val isRemember = prefs.getBoolean("remember_password", false)  
        if (isRemember) {  
            // Set the account and password info in the EditText views  
            val account = prefs.getString("account", "")  
            val password = prefs.getString("password", "")  
            accountEdit.setText(account)  
            passwordEdit.setText(password)  
            rememberPass.isChecked = true  
        }  
        login.setOnClickListener {  
            val account = accountEdit.text.toString()  
            val password = passwordEdit.text.toString()  
            // success for account is admin and password is123456  
            if (account == "admin" && password == "123456") {  
                val editor = prefs.edit()
```

```
        if (rememberPass.isChecked) { // check if CheckBox is selected or
not
            editor.putBoolean("remember_password", true)
            editor.putString("account", account)
            editor.putString("password", password)
        } else {
            editor.clear()
        }
        editor.apply()
        val intent = Intent(this, MainActivity::class.java)
        startActivity(intent)
        finish()
    } else {
        Toast.makeText(this, "account or password is invalid",
        Toast.LENGTH_SHORT).show()
    }
}
}
```

The code above first gets an instance of SharedPreferences in onCreate(), then calls its getBoolean() method to get the value corresponding to the key remember\_password. Apparently, when the app starts for the first time, there is no corresponding value at all, so we should set default value to be false. If we login successfully, we can use isChecked() to check the status of CheckBox. If the CheckBox is checked, it means that user expects the app to remember the password, then we set remember\_password to be true, save the current values of account and password in SharedPreferences, and commit the action; if the CheckBox is not checked, then we simply call clear() to erase the data in SharedPreferences.

When user has checked the CheckBox and login successfully once, the corresponding value of the key remember\_password is true. Now if user restarts the app and comes to the login screen, account and password will be read from SharedPreferences, fill the EditText views, and then select the CheckBox. That's it for remembering password feature!

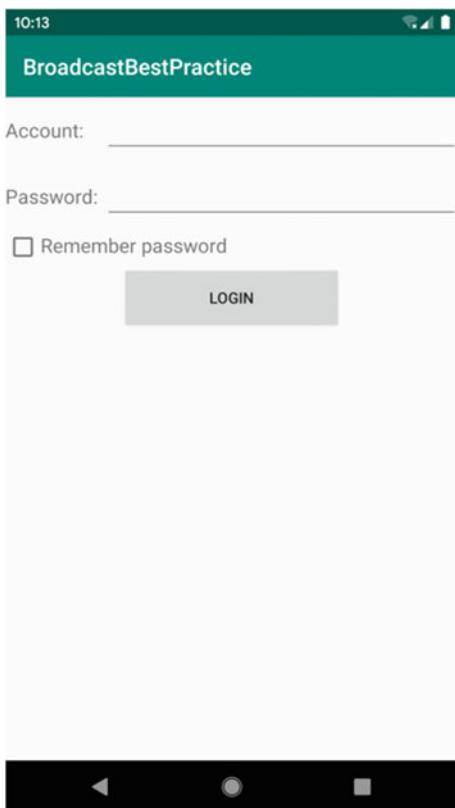
Run the app again and you should see a CheckBox for remembering password in the login screen as shown in Fig. 7.8.

Then, use “admin” for account and “123456” for password, select the CheckBox, click “login,” MainActivity will show up. Then send a broadcast in MainActivity to force log out and app will go back to the login screen. Now you should see that account and password fields have already been filled as shown in Fig. 7.9.

Now, with the help of SharedPreferences, we implemented the functionality of remembering the password, and you should have a better understanding of SharedPreferences.

Notice that this functionality is for demonstration purpose and cannot be used in real world project. Because it is not secure to save password in plain text in SharedPreferences and it can be easily stolen by hackers. Thus, you need some encryption mechanism to encrypt the password in the real world project.

**Fig. 7.8** Login screen with Remember password CheckBox



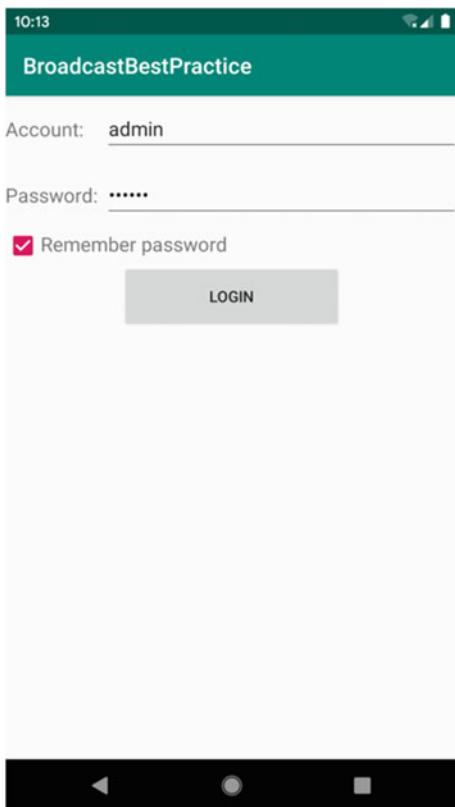
That is everything for SharedPreferences. In the next section, we will discuss the most important topic in this chapter: Android database.

## 7.4 SQLite Database

I could not believe that Android system embedded database, when I began learning Android! SQLite is a light-weight relational database which is fast and uses little resources. Usually it takes a few KB memory, thus is a perfect choice for mobile devices. SQLite does not only support standard SQL syntax but also is ACID. Thus, if you have used other relational database, you should be able to ramp up rather quickly. SQLite is simpler than most databases, and even does not require user name nor password to use. Android data persistence performance got a leap forward by embedding this powerful database in the operating system.

File and SharedPreferences can only be used to store some simple data and key-value pairs, but cannot cope with large amount of data nor complicated relational data. For instance, our Message app can have multiple threads, each of the

**Fig. 7.9** Account and password have been remembered



thread can have multiple messages, and most of the messages are with certain contacts in the Contact app. It is difficult to even think about using file or SharedPreferences to store this kind of data. However, with SQLite, we can do it. Now, let us take a look at how to use SQLite in Android.

#### 7.4.1 *Create Database*

Android provides SQLiteOpenHelper class to help us manage the database conveniently. With this class, we can easily create and upgrade the database. Now, let us take a look at the basic use of SQLiteOpenHelper.

First, SQLiteOpenHelper is an abstract class which means that we need to create a class that inherits it, so that we can use this class. There are two abstract methods in SQLiteOpenHelper: onCreate() and onUpgrade(). We need to override these two methods in our class and implement the business logic, when we create and upgrade database.

There are two very important methods in SQLiteOpenHelper: `getReadableDatabase()` and `getWritableDatabase()`. Both of these two methods can create or open an existing database (if database exists then open, if not then create a new one), then return an object that can read and write. The difference is that, when database is not writable (for instance, disk full), object from `getReadableDatabase()` will open the database with read-only mode and `getWritableDatabase()` will throw an exception.

There are two constructors in SQLiteOpenHelper we can override and usually we just use the one with less params. This constructor takes four params: the first param is Context which is essential to operate on the database; the second param specifies the name of the database; the third params allows us to return a customized Cursor when we query the data and usually we just pass null; and the fourth param is to specify the version number of the current database that can be used to upgrade the database. After constructing the instance of SQLiteOpenHelper, we can use its `getReadableDatabase()` or `getWritableDatabase()` to create the database. The database files are under `/data/data/<package name>/databases/`. Then the `onCreate()` method will get called, thus usually we should write the logic to create table here.

Let us use an example to see how to use SQLiteOpenHelper. First, create DatabaseTest project.

Here, we want to create a database with the name BookStore.db and create a Book table with id (primary key), author, price, number of pages, book name, etc. Of course we need SQL statements to create the table, and the Book table can be created with the following code:

```
create table Book (
    id integer primary key autoincrement,
    author text,
    price real,
    pages integer,
    name text)
```

If you have experience in SQL, you shouldn't feel it is hard to understand. Unlike other databases, SQLite data types are simple: integer, real (for float data), text, blob (for binary data). In the above statement, we set id as the primary key and use autoincrement keyword to make id column increment automatically.

Then, we need to execute this SQL statement to finish creating the table. Create MyDatabaseHelper class that inherits SQLiteOpenHelper as code below:

```
class MyDatabaseHelper(val context: Context, name: String, version: Int) :
    SQLiteOpenHelper(context, name, null, version) {

    private val createBook = "create table Book (" +
        "id integer primary key autoincrement, " +
        "author text, " +
        "price real, " +
```

```

    "pages integer, " +
    "name text) "

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(createBook)
        Toast.makeText(context, "Create succeeded", Toast.LENGTH_SHORT).show()
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
    newVersion: Int) {
    }

}

```

As you can see, we convert the SQL statement into a string and in onCreate() we call execSQL() to execute this statement, and use a toast message to show that database and table have been created successfully.

Now update activity\_main.xml as code below:

```

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

    <Button
        android:id="@+id/createDatabase"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Create Database"
        />

</LinearLayout>

```

The code above simply add a button to create the database. Lastly, update MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val dbHelper = MyDatabaseHelper(this, "BookStore.db", 1)
        dbHelper.setClickListener {
            dbHelper.writableDatabase
        }
    }
}

```

**Fig. 7.10** Successfully Created Database

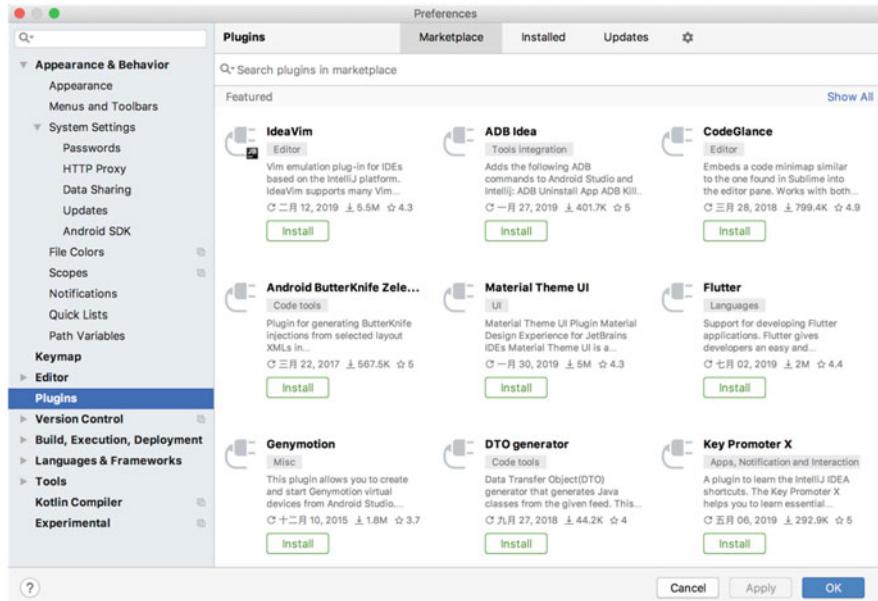


}

The code above constructs an instance of MyDatabaseHelper in onCreate() and sets the database name to BookStore.db and version to 1, then the Create Database button click event calls getWritableDatabase(). Thus, when user clicks the Create Database button for the first time, it will find that current app does not have BookStore.db and will create it, then onCreate() in MyDatabaseHelper will be called, thus Book table will also be created. After all of this, a toast message will show up to notify the user database has been created successfully. Click Create Database button again, since BookStore.db has already been created, it will not get recreated.

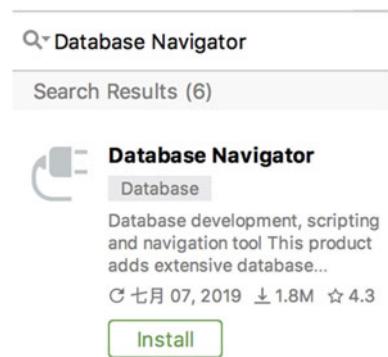
Run the app and click Create Database button in the main screen, the result should be as shown in Fig. 7.10.

Now BookStore.db and Book table have been created successfully, since when you click Create Database button again, no more toast message will show up. But back to the old question: how to approve they indeed get created?



**Fig. 7.11** Plugin Manager

**Fig. 7.12** Database Navigator Plugin



We can still use Device File Explorer but it can only show that BookStore.db file is in the database folder, and we cannot see Book table from this tool. Database Navigator can help us to see the tables.

Android Studio is based on IntelliJ IDEA, thus we can also use the rich plugins from IntelliJ IDEA in Android Studio. From the navigation bar in Android Studio, click Preferences → Plugins to open the Plugin Manager as shown in Fig. 7.11.

This is an official marketplace for plugins, and you can find the Database Navigator through search bar as shown in Fig. 7.12.

Click Install, then Android Studio will download and install the plugin, after installation, restart Android Studio as instructed to use the newly installed plugin.

| Name                                 | Permissions | Date             | Size  |
|--------------------------------------|-------------|------------------|-------|
| ▶ com.example.broadcastbestpractices | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ com.example.broadcasttest          | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▼ com.example.databasetest           | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ cache                              | drwxrwxs--x | 2019-07-08 21:49 | 4 KB  |
| ▶ code_cache                         | drwxrwxs--x | 2019-07-08 21:49 | 4 KB  |
| ▼ databases                          | drwxrwx--x  | 2019-07-08 21:49 | 4 KB  |
| BookStore.db                         | -rw-rw----  | 2019-07-08 21:49 | 20 KB |
| BookStore.db-journal                 | -rw-rw----  | 2019-07-08 21:49 | 0 B   |
| ▶ com.example.filepersistencetest    | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |
| ▶ com.example.fragmentbestpractices  | drwxrwx--x  | 2019-06-12 07:47 | 4 KB  |

**Fig. 7.13** Generated BookStore.db File

Open Device File Explorer and you should find BookStore.db in /data/data/com.example.databasetest/databases/ as shown in Fig. 7.13.

In the same folder, there is BookStore.db-journal file which is a temporary log file to support transactions and usually has size of 0 byte, and we can ignore this file for now.

Right click Save As on the file of BookStore.db to save this file in your computer. You should find DB Browser tool which we just installed on the left side bar. If you cannot find this tool, use shorts Ctrl + Shift + A (for Mac it is command + shift + A) to open search and type in DB Browser to find it.

To open the database file, click the plus sign at the top left corner and select SQLite as shown in Fig. 7.14.

Then select the BookStore.db file in the pop up window as shown in Fig. 7.15.

Click OK to finish configuration, now DB Browser will show all the content in BookStore.db as shown in Fig. 7.16.

As you can see, there is a Book table in the database and the columns match the SQL statement we used before which proves that BookStore.db and Book table have been created successfully.

#### 7.4.2 Upgrade Database

You can find an empty method called onUpgrade() in MyDatabaseHelper, which, as the name implies, is used to upgrade the database. This is an important method to manage the database, and let us take a look at it.

We have a Book table to store the data about books in DatabaseTest project, what if we want to add a Category table?

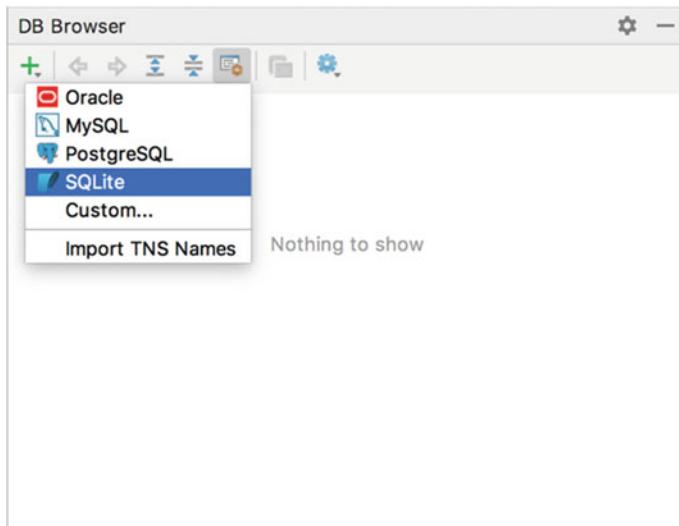


Fig. 7.14 Select SQLite in DB Browser

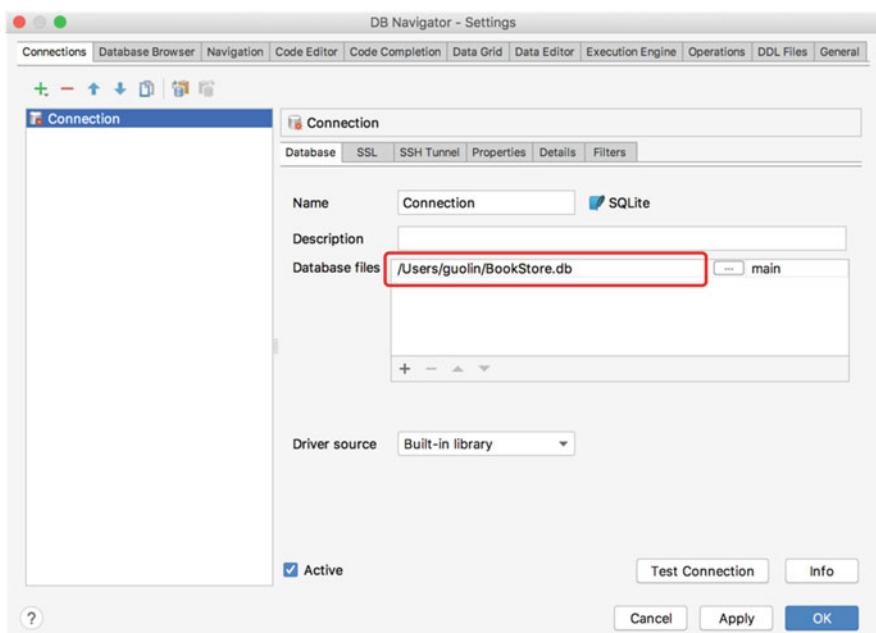


Fig. 7.15 Select BookStore.db File

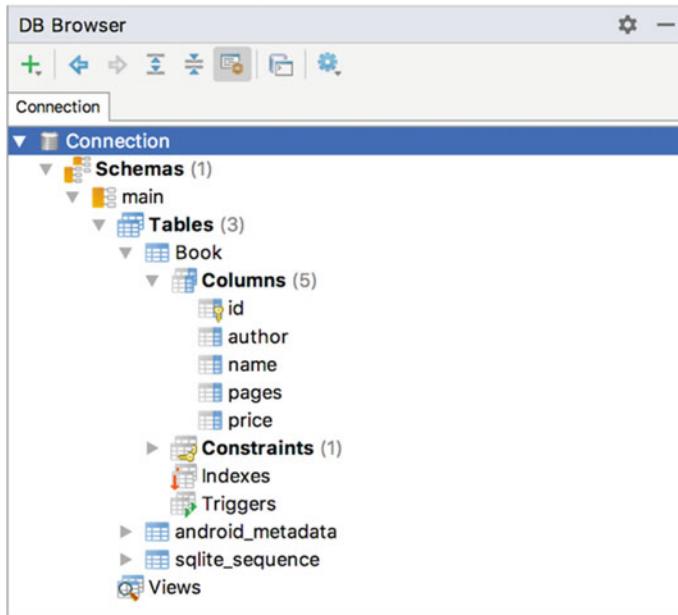


Fig. 7.16 Content in BookStore.db

Assume Category table has id, category name and category code columns, then we can use the following SQL statement to create the table:

```
create table Category (
    id integer primary key autoincrement,
    category_name text,
    category_code integer)
```

Next, let us convert this statement to Kotlin code in MyDatabaseHelper, as shown below:

```
class MyDatabaseHelper(val context: Context, name: String, version: Int) :
    SQLiteOpenHelper(context, name, null, version) {
    ...
    private val createCategory = "create table Category (" +
        "id integer primary key autoincrement, " +
        "category_name text, " +
        "category_code integer) "

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(createBook)
        db.execSQL(createCategory)
        Toast.makeText(context, "Create succeeded", Toast.LENGTH_SHORT) .
```

```

show()
}

override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
newVersion: Int) {
}

}

```

It looks right at the first glance. However, run the app again and click Create Database button, no toast message will show up, and you can also check the database from DB Browser, and you should find that there is no Category table.

The reason is because BookStore.db has already been created, onCreate() in MyDatabaseHelper will never get called again, which means the new table will not get created.

To solve this problem, we can uninstall the app and run the app again. Since, there is noBookStore.db in the fresh install, after clicking Create Database, onCreate() in MyDatabaseHelper will be called and Category table can be created.

Apparently, this is very poor user experience, and we can use SQLiteOpenHelper to upgrade the database without reinstalling the app. Modify MyDatabaseHelper as code below:

```

class MyDatabaseHelper(val context: Context, name: String, version: Int):
    SQLiteOpenHelper(context, name, null, version) {
    ...
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
    newVersion: Int) {
        db.execSQL("drop table if exists Book")
        db.execSQL("drop table if exists Category")
        onCreate(db)
    }
}

```

We call DROP statement twice in onUpgrade() to delete the two tables, if they exist and then onCreate() will recreate the tables. This is because if table exists when creating the table, an exception will be thrown.

The next question will be how to get onUpgrade() called. The fourth param in SQLiteOpenHelper constructor is version number of current database which we passed in 1 previously, now we just need to pass in a number that is larger than 1 to get onUpgrade() called. Update MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
}

```

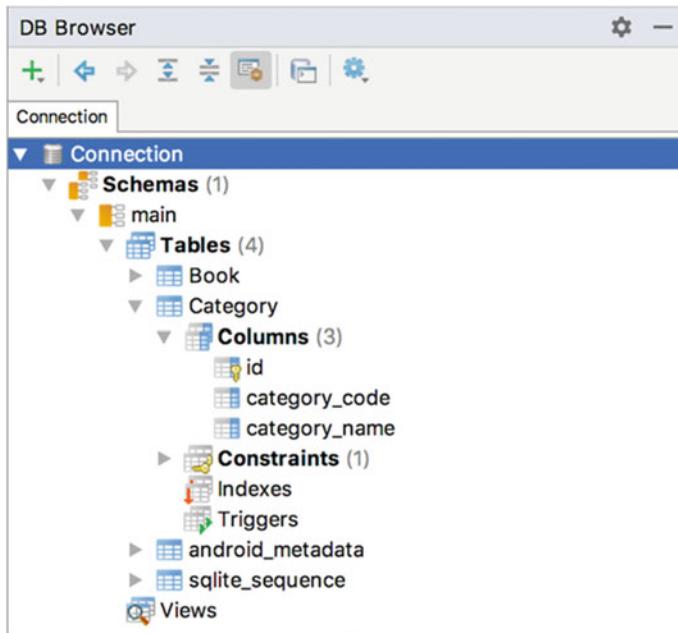


Fig. 7.17 BookStore.db Content after upgrade

```
val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
createDatabase.setOnClickListener {
    dbHelper.writableDatabase
}
}
```

Here, we set the version number to be 2, to show that we are upgrading the database. Run the app again and click Create Database, the toast message will show up again.

To validate the existence of Category table, we can use the same approach as mentioned previously to export the BookStore.db file to computer and replace the old BookStore.db file, then import this file in DB Browser to load the new BookStore.db as shown in Fig. 7.17.

As you can see, Category table has been created which means that our upgrading logic works.

### 7.4.3 Add Data

After creating and upgrading database, let us take a look at how to operate on the data in the database. There are four operations, we can do on the data which is called CRUD. C stands for create, R stands for retrieve, U stands for update, and D stands for delete. Each of the operation has its corresponding SQL command. If you're familiar with SQL, you should know that, we need to use insert to add data, use select to query, use update to upgrade, and use delete to delete data. However, since not every developer is familiar with SQL, Android provides a suite of assistant methods to help you do CRUD without SQL statements.

As mentioned before, `getReadableDatabase()` and `getWritableDatabase()` in `SQLiteOpenHelper` can be used to create and upgrade database. These two methods will return an instance of `SQLiteDatabase`, and we can use the instance to do CRUD operations.

Next let us take a look at how to add data into database dynamically. `SQLiteDatabase` provides `insert()` to add data. It takes three params: the first param is the table name that we want to add data to; the second param is used to set default value to NULL to nullable columns when the value is not specified which is not used very often, we can just pass null to it; the second param is an object of `ContentValues` which provides a suite of overrides for `put()` that can add data to `ContentValues` and only requires the column name and corresponding value.

Next, let us use an example to learn how to add data. Modify `activity_main.xml` as code below:

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

    ...

    <Button
        android:id="@+id/addData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add Data"
        />
</LinearLayout>
```

As you can see, we add a new button and add the code to add data in the click event handler of this button. Continue updating the `MainActivity` as shown below:

```
class MainActivity : AppCompatActivity() {
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
    ...
    addData.setOnClickListener {
        val db = dbHelper.writableDatabase
        val values1 = ContentValues().apply {
            // construct the first row of data
            put("name", "The Da Vinci Code")
            put("author", "Dan Brown")
            put("pages", 454)
            put("price", 16.96)
        }
        db.insert("Book", null, values1) // insert the first row of data
        val values2 = ContentValues().apply {
            // construct the second row of data
            put("name", "The Lost Symbol")
            put("author", "Dan Brown")
            put("pages", 510)
            put("price", 19.95)
        }
        db.insert("Book", null, values2) // insert the second row of data
    }
}
```

In the click event code, we first get an instance of SQLiteDatabase and then use ContentValues to construct the data entry. You probably can notice that we only specified four columns data of the Book table without assigning data to id column. This is because when we create the table, we set id column to auto increase and its value will be auto generated when data gets inserted into database and no manual operation needed. Next, we can use insert() to add data into the table, notice that we add two rows of data.

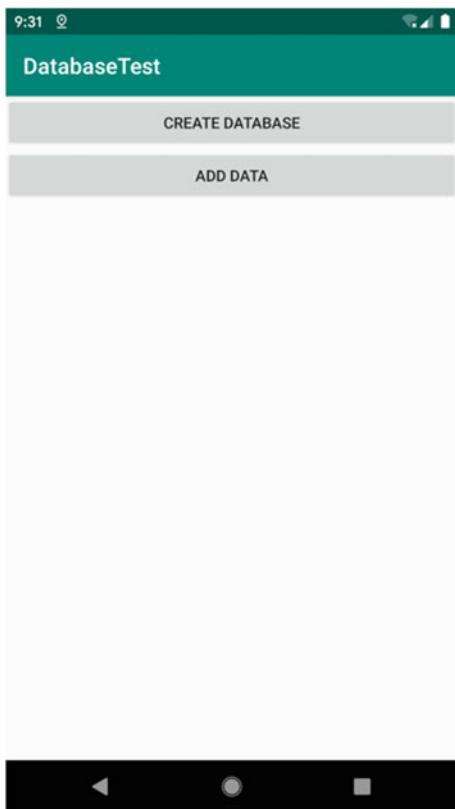
Run the app again, the main screen should be as shown in Fig. 7.18.

Click Add Data button and the two rows of data should be added which can be validated by DB Browser as mentioned before, double click the table to see the content of the table. Double click Book table, and it will show a window as shown in Fig. 7.19.

This window is used to set the filter for query, and we do not need any filter here. Click No Filter and you should see the data as shown in Fig. 7.20.

As you can tell, we successfully added the two rows of data to Book table.

**Fig. 7.18** Add the Add Data button



#### 7.4.4 Update Data

Next, let us take a look at how to update the existing data in the table. SQLiteDatabase provides update() to update the existing data. This method takes four params: the first param is the table name to specify which table to update which is the same as insert(); the second param is the ContentValues object which contains the new data; the third and fourth params are used to specify which row(s) are to be updated and if not set, all rows will be updated.

Let us continue working on DatabaseTest to experiment with updating data. Imagine that the first book in the database does not sell very well, and we want to lower the price to attract more customers. What should we do? First, update activity\_main.xml as code below;

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```

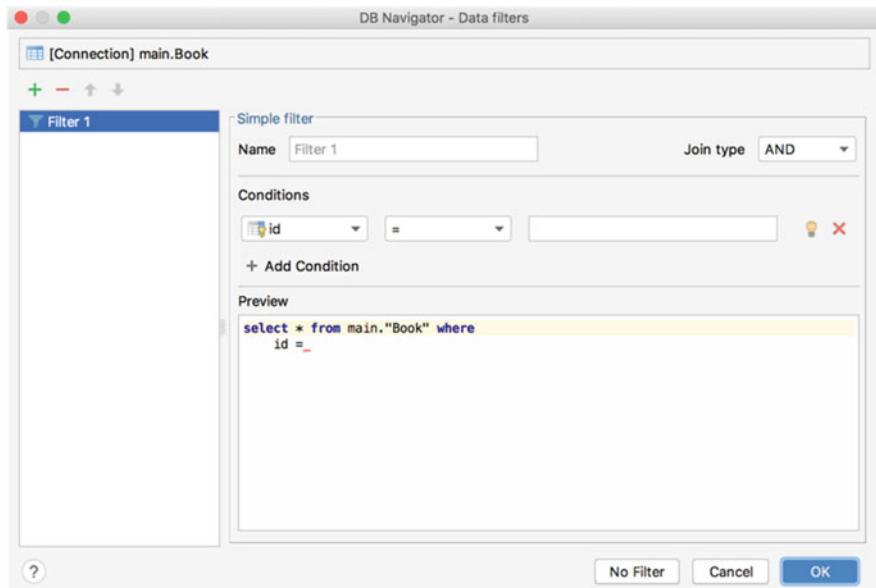


Fig. 7.19 Window to Set Query Filter

| Book |    |           |       |       |                   |
|------|----|-----------|-------|-------|-------------------|
|      | id | author    | price | pages | name              |
| 1    | 1  | Dan Brown | 16.96 | 454   | The Da Vinci Code |
| 2    | 2  | Dan Brown | 19.95 | 510   | The Lost Symbol   |

Fig. 7.20 Data in Book Table

```
    android:layout_height="match_parent"
    >

    ...

<Button
    android:id="@+id/updateData"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Update Data"
    />
</LinearLayout>
```

The highlighted code adds a new button to update the data. Next modify MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
        ...
        updateData.setOnClickListener {
            val db = dbHelper.writableDatabase
            val values = ContentValues()
            values.put("price", 10.99)
            db.update("Book", values, "name = ?", arrayOf("The Da Vinci
Code"))
        }
    }
}

```

The highlighted code adds click event handling logic. It creates an instance of ContentValues and only set the price, which means that we only want to update the price column to be 10.99. Then it uses update() in SQLiteDatabase to execute the updating operation. As you can see, the third and fourth params are used to specify which row(s) to update. The third parameter corresponds to the WHERE part of the SQL statement, indicating that all rows with name equal to ? rows, and ? is a placeholder, you can specify the contents of each placeholder in the third parameter by using an array of strings provided in the fourth parameter. The arrayOf() method is a built-in method provided by Kotlin for easy array creation. Thus the highlighted code basically tries to update the price of The Da Vinci Code to 10.99.

Run the app again and main screen should be as shown in Fig. 7.21.

Click Update Data button and use the method mentioned before to see the content in Book table, the result should be as shown in Fig. 7.22.

As you can see, the price of The Da Vinci Code has been updated to 10.99.

#### 7.4.5 Delete Data

As you can see, adding and updating data are both easy, and now let us take a look at how to delete data from table.

Deleting data is even easier as all the concepts needed are already covered. SQLiteDatabase provides delete() to do deletion. This method takes three params: the first param is the table name; the second and third params are used to specify which row(s) of data to be deleted and by default it will apply to all rows.

Now, let us experiment with it. Update activity\_main as code below:

```

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

```

**Fig. 7.21** Add Update Data button



|   | id | author    | price | pages | name              |
|---|----|-----------|-------|-------|-------------------|
| 1 | 1  | Dan Brown | 10.99 | 454   | The Da Vinci Code |
| 2 | 2  | Dan Brown | 19.95 | 510   | The Lost Symbol   |

**Fig. 7.22** Updated Data

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

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

```

        android:layout_height="wrap_content"
        android:text="Delete Data"
    />
</LinearLayout>
```

The highlighted code adds a button in the layout to delete data. Next, update MainActivity as shown below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
        ...
        deleteData.setOnClickListener {
            val db = dbHelper.writableDatabase
            db.delete("Book", "pages > ?", arrayOf("500"))
        }
    }
}
```

As you can see, the highlighted code adds deletion logic in the click event handler. The second and third param are used to specify that we only want to delete the books that have more than 500 pages. Of course, this strange requirement is just for demonstration purpose. You can take a look at the current Book table, The Lost Symbol has more than 500 pages which means that if we click the delete button, this record will be deleted.

Run the app again and the main screen should be as shown in Fig. 7.23.

Click Delete Data and check the data in the Book table, the result is as shown in Fig. 7.24.

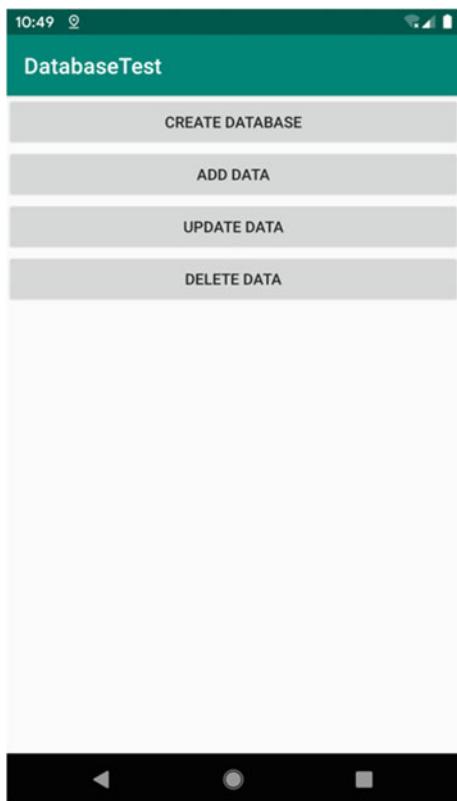
#### 7.4.6 *Query Data*

The last operation type we will discuss is query and after this, you will know all the CRUD operations. Query is the most complicated one among all the four operations.

SQL is short form for Structured Query Language, and most of the functionalities are about query, only a small amount of functionalities are about adding, deleting, and updating. SQL has too many topics to discuss, thus I will not cover everything about it. Instead I will only cover query in Android. If you're interested in SQL, you can find a dedicated SQL book to deep dive into it.

As you can probably tell, SQLiteDatabase also provides query() to query the data. This method has complicated param combinations, even the override method that requires the least params still needs 7 params. Let us take a look at the 7 params.

**Fig. 7.23** Add Delete Data button



| Book |    |           |       |                   |
|------|----|-----------|-------|-------------------|
|      | id | author    | price | pages             |
| 1    | 1  | Dan Brown | 10.99 | 454               |
|      |    |           |       | The Da Vinci Code |

**Fig. 7.24** Data after Deletion

Obviously, the first param should be the name of the table to specify which table we are going to query against. The second param is to specify which column(s) to query, and by default it is for all columns. The third and fourth params are to specify which row(s) to query, and by default it is for all rows. The fifth param is to specify the column to group by, and by default it will not group by for the result. The sixth param is to further filter the result after group by, and by default there is no filtering. The seventh param is to specify the rule of ordering, and it will apply the default ordering rule if without specification. Refer to Table 7.1 for more details. Other override methods are very similar, and you can figure them out by yourself and will not be covered here.

**Table 7.1** query() Params

| query() Params | Corresponding SQL Statement | Description                             |
|----------------|-----------------------------|---|
| table          | from table_name             | Specify table name                      |
| columns        | select column1, column2     | Specify column name                     |
| selection      | where column = value        | Specify where condition                 |
| selectionArgs  | -                           | Provide value for place holder in where |
| groupBy        | group by column             | Specify column to group by              |
| having         | having column = value       | Filter result after group by            |
| orderBy        | order by column1, column2   | Specify order of result                 |

Though there are quite a few params in query(), we do not need to assign values to all the params most of the time. query() will return an object of Cursor, and all the data can be retrieved from this object.

Let us use examples to experiment with query. Modify activity\_main.xml as code below:

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

    ...
    <Button
        android:id="@+id/queryData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Query Data"
        />
</LinearLayout>
```

The highlighted code simply adds a button to query the data. Update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
        ...
        queryData.setOnClickListener {
            val db = dbHelper.writableDatabase
            // query all data in Book table
            val cursor = db.query("Book", null, null, null, null, null, null)
            if (cursor.moveToFirst()) {
```

```
        do {
            // Get all the data from Cursor object and print
            val name = cursor.getString(cursor.getColumnIndex("name"))
            val author = cursor.getString(cursor.getColumnIndex
("author"))
            val pages = cursor.getInt(cursor.getColumnIndex("pages"))
            val price = cursor.getDouble(cursor.getColumnIndex("price"))
            Log.d("MainActivity", "book name is $name")
            Log.d("MainActivity", "book author is $author")
            Log.d("MainActivity", "book pages is $pages")
            Log.d("MainActivity", "book price is $price")
        } while (cursor.moveToNext())
    }
    cursor.close()
}
}
```

In the code above, we first call query() of SQLiteDatabase to query data. This call only provides the Book table name while keeping all the other params to be null, which means that we want to query all the data in the table though there is only row of data in the table. After query, we get an instance of Cursor and its moveToFirst() will move the cursor to the first row of the data. Then, we will try to iterate through all the data with a loop statement. In the loop, getColumnIndex() will return the index of a column in the table, then with this index we can get the corresponding value from the cursor. Next, we use Log to print the data in Logcat to validate the query result and use close() to close the Cursor.

Run the app again and the main screen should be as shown in Fig. 7.25.

Click Query Data button and Logcat should show result as shown in Fig. 7.26.

As you can tell, we got the only row of data from Book table.

Of course, this is just a simple demonstration of the query method. In the real world project, you will need queries that are much more complicated. You can explore the more advanced uses of the query method with all the rest of params.

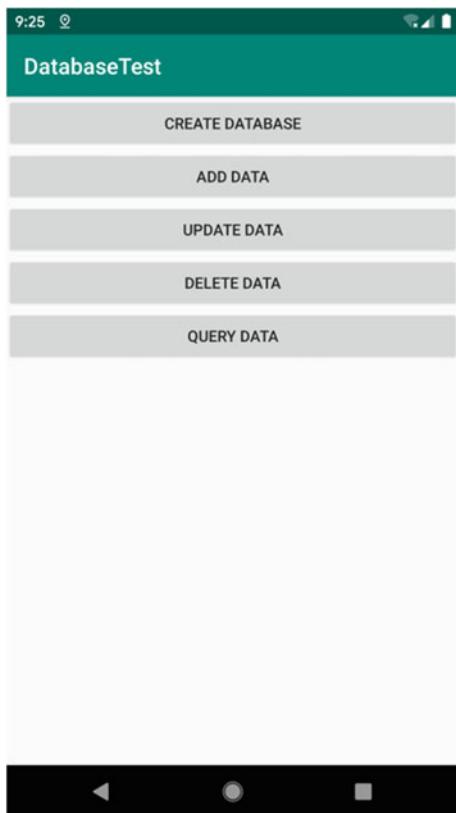
#### 7.4.7 Use SQL

Although, Android provides many convenient APIs to access database, some developers just prefer to use SQL directly. To meet this requirement, Android also provides a suite of tools to access database directly with SQL.

Now, let me use a few simple examples to demonstrate how to use SQL to do CRUD.

To add data:

**Fig. 7.25** Add Query button



com.example.databasetest (14117) ▾ Verbose ▾ Q<sup>v</sup>

```
I117/com.example.databasetest D/MainActivity: book name is The Da Vinci Code
I117/com.example.databasetest D/MainActivity: book author is Dan Brown
I117/com.example.databasetest D/MainActivity: book pages is 454
I117/com.example.databasetest D/MainActivity: book price is 10.99
```

**Fig. 7.26** Print result from query

```
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",  
        arrayOf("The Da Vinci Code", "Dan Brown", "454", "16.96")  
    )  
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",  
        arrayOf("The Lost Symbol", "Dan Brown", "510", "19.95")  
    )
```

To update data:

```
db.execSQL("update Book set price = ? where name = ?", arrayOf("10.99",  
"The Da Vinci Code"))
```

To delete data:

```
db.execSQL("delete from Book where pages > ?", arrayOf("500"))
```

To query data:

```
val cursor = db.rawQuery("select * from Book", null)
```

As you can see, only the query method uses rawQuery() of SQLiteDatabase, other three operations all use execSQL(). The results are exactly the same as the CRUD methods we learned previous and based on your preference, you can choose whichever you like.

## 7.5 SQLite Database Best Practice

The last section covers basic use of SQLite database, and there are more advanced techniques in SQLite database. Let us experiment with them in this section.

### 7.5.1 *Transaction*

We all know that SQLite database supports transaction which can ensure that a series of operations to be unit operation, which means that these operations either all finish or none will finish. When should we use transaction? Imagine that you need to make a money transfer and bank will first withdraw money from your account and then deposit the same amount of money in the receiver's account. Looks simple and no risk at all, right? However, what if after withdraw succeeds but somehow receiver failed to receive the money? Will money just disappear? Apparently banks have considered this kind of scenario and will guarantee that withdraw and deposit will both succeed or fail. And the technology used here is related to transaction.

Now, let us take a look at how to use transaction in Android. We will work on top of DatabaseTest project. Assume that data in Book table becomes obsolete, and we want to replace them with new data, then we can use delete() to delete the old data from Book table and then use insert() to add the new data into the table. We need to make sure that deletion and adding data can succeed or fail altogether. Update activity\_main.xml as code below:

```
<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/replaceData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Replace Data"
    />
</LinearLayout>
```

As you can see, we add another button to replace the data. Update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val dbHelper = MyDatabaseHelper(this, "BookStore.db", 2)
        ...
        replaceData.setOnClickListener {
            val db = dbHelper.writableDatabase
            db.beginTransaction() // start transaction
            try {
                db.delete("Book", null, null)
                if (true) {
                    //manually throw exception to fail the transaction
                    throw NullPointerException()
                }
                val values = ContentValues().apply {
                    put("name", "Game of Thrones")
                    put("author", "George Martin")
                    put("pages", 720)
                    put("price", 20.85)
                }
                db.insert("Book", null, values)
                db.setTransactionSuccessful() // transaction succeed
            } catch (e: Exception) {
                e.printStackTrace()
            } finally {
                db.endTransaction() // transaction finishes
            }
        }
    }
}
```

The above code demonstrates the standard way to do transaction in Android. First, we use beginTransaction() in SQLiteDatabase to start a transaction, then we execute the database business logic in try and catch block. After everything finishes, we call setTransactionSuccessful() to indicate that transaction succeeds and finally call endTransaction() in finally block to end the transaction. Notice that after deletion operation, we manually throw a NullPointerException so that adding data will not be called. However, since these operations are in a transaction, the old data should not be deleted.

Run the app and click Replace Data button, then click Query Data button. You will find that Book table still only has the old data which means that our transaction code works as expected. Now, remove the code that manually throw the exception and run the app again. Click Replace Data should update the Book table with the new data, and you can use Query Data to verify.

### 7.5.2 Best Practice to Upgrade Database

The way we upgrade the database in Sect. 7.4.2 is brutally simple: we simply delete all the tables in onUpgrade() and execute onCreate() to create the data to make sure data in the database is fresh. You cannot do so in production environment. Imagine that your new feature is live and your app gets downloaded by significant amount of people. Now, the new feature requires database upgrading and user finds that after upgrading the app, data has lost! Then you might lose majority of the users.

This is scary experience, but with proper handling we can ensure that data will not get lost after database upgrade.

Now, let us take a look at how to implement proper upgrade. As you know, every database at any time will have a corresponding version number, and when the specified version number is larger than the current database version number, onUpgrade() will be triggered to upgrade the database. For each version, we need to specify the logic to upgrade the database, and in onUpgrade() we need to check the current database version to decide what upgrade logic to apply.

Next, let us simulate a database upgrade case by using MyDatabaseHelper. The first version is simple, we just need to create Book table and code in MyDatabaseHelper as shown below:

```
class MyDatabaseHelper(val context: Context, name: String, version: Int) :  
    SQLiteOpenHelper(context, name, null, version) {  
  
    private val createBook = "create table Book (" +  
        " id integer primary key autoincrement, " +  
        "author text, " +  
        "price real, " +  
        "pages integer, " +  
        "name text)"
```

```

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(createBook)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
newVersion: Int) {
    }

}

```

A few weeks later, a new requirement comes in to add Category table. Thus, we can update the code in MyDatabaseHelper as code below:

```

class MyDatabaseHelper(val context: Context, name: String, version:
Int) :
    SQLiteOpenHelper(context, name, null, version) {

    private val createBook = "create table Book (" +
        "id integer primary key autoincrement," +
        "author text," +
        "price real," +
        "pages integer," +
        "name text)"

    private val createCategory = "create table Category (" +
        "id integer primary key autoincrement," +
        "category_name text," +
        "category_code integer)"

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(createBook)
        db.execSQL(createCategory)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,
newVersion: Int) {
        if (oldVersion <= 1) {
            db.execSQL(createCategory)
        }
    }
}

```

As you can see, in onCreate() we add a new statement to create table and add a condition check in onUpgrade() to only create Category table when the old database version number is equal or smaller than 1.

With the code above, when user install the second version of the app directly, onCreate() will be called to create the two tables. When user upgrade the app to

version 2, `onUpgrade()` will be called, and since Book table exists, we just need to create Category table.

Then comes a new requirement that requires us to build connection between Book table and Category table by adding `category_id` column in Book table. Update MyDatabaseHelper as code below:

```
class MyDatabaseHelper(val context: Context, name: String, version: Int) :  
    SQLiteOpenHelper(context, name, null, version) {  
  
    private val createBook = "create table Book (" +  
        "id integer primary key autoincrement," +  
        "author text," +  
        "price real," +  
        "pages integer," +  
        "name text," +  
        "category_id integer)"  
  
    private val createCategory = "create table Category (" +  
        "id integer primary key autoincrement," +  
        "category_name text," +  
        "category_code integer)"  
  
    override fun onCreate(db: SQLiteDatabase) {  
        db.execSQL(createBook)  
        db.execSQL(createCategory)  
    }  
  
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int,  
        newVersion: Int) {  
        if (oldVersion <= 1) {  
            db.execSQL(createCategory)  
        }  
        if (oldVersion <= 2) {  
            db.execSQL("alter table Book add column category_id integer")  
        }  
    }  
}
```

As you can see, we add `category_id` column in Book creation statement. Then, when user directly install V3, this new column will be automatically added. However, if user has previous versions, then upgrade will happen. In `onUpgrade()`, we add a new condition check, when the current version is 2, then alter will be called to add column `category_id` for Book table.

Notice that, whenever we upgrade the database, we need to add a corresponding `if` statement in `onUpgrade`. The reason for this is to make sure that all versions of app can upgrade successfully. For example, if user upgrades from V2 to V3, then only the second condition will be executed. If user upgrades from V1 to V3 directly, then

both conditions will be called. Using this kind of method to manage the database upgrade, no matter how user upgrade the app, we can ensure that database is the newest, and data will not get lost because of upgrade.

That is everything for SQLite Database Best Practice. In this section, we learned the most conventional way to operate the database in Android. Google released a database framework—Room—specifically for Android. Compared with the conventional database API, Room is more complicated but more formal and reasonable and more compliant to modern high-quality App development paradigm. We will cover Room in Chap. 13.

Without further due, let us start the Kotlin class of this chapter.

## 7.6 Kotlin Class: Application of Higher-Order Function

The Kotlin class in the last chapter discussed how to use higher-order functions. In this chapter's Kotlin class, let us see where we can apply higher-order functions. If you are familiar with higher-order functions, then this section will be easier.

Higher-order functions can be used to simplify API calls. After simplification with higher-order functions, a lot of the APIs can greatly improve usability and readability.

To demonstrate this, we will use higher-order functions to simplify SharedPreferences and ContentValues.

### 7.6.1 Simplify Use of SharedPreferences

Before try to simplify the use of SharedPreferences, let us recall how to use SharedPreferences. To add data in SharedPreferences, we can be break down to the following three steps:

1. Use edit() of SharedPreferences to get instance of SharedPreferences.Editor;
2. Add data to SharedPreferences.Editor instance;
3. Use apply() to commit the operation;

The corresponding code is shown below:

```
val editor = getSharedPreferences("data", Context.MODE_PRIVATE) .  
edit()  
editor.putString("name", "Tom")  
editor.putInt("age", 28)  
editor.putBoolean("married", false)  
editor.apply()
```

```

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.2.0'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.2.0'
}

```

**Fig. 7.27** Auto Imported KTX Lib

The above code is already simple enough but the code style is very Java and we can do better in Kotlin.

Next, let us use higher-order function to simplify the use of SharedPreferences. Create SharedPreferences.kt file and add the following code:

```

fun SharedPreferences.open(block: SharedPreferences.Editor.() ->
Unit) {
    val editor = edit()
    editor.block()
    editor.apply()
}

```

Code above is short but enough to cover the essence of higher-order function. Let me explain further.

We add an open function for SharedPreferences class using extension function. As it can take a param that is function type, thus open function is a higher-order function.

Since, open function has the context of SharedPreferences, we can directly call edit() to get instance of SharedPreferences.Editor. Also open function takes a function type param that is of SharedPreferences.Editor, thus we need to call editor:block() to call the function type param. Then, we can add data in the function param. We also need to call editor.apply() to commit the operation.

After definition of open function, we can use SharedPreferences easier as shown in the code below:

```

getSharedPreferences("data", Context.MODE_PRIVATE).open {
    putString("name", "Tom")
   .putInt("age", 28)
    putBoolean("married", false)
}

```

As you can see, we directly call open on the object of SharedPreferences and add data in the Lambda expression. Notice that Lambda expression now has the context of SharedPreferences.Editor, thus we can directly call put to add data. We no longer need to call apply to commit the action because open will do it.

I have to mention that KTX extension lib which is provided by Google has already included the simplified use above and will be auto imported to the dependencies in build.gradle by Android Studio when creating a new project, as shown in Fig. 7.27.

Thus, we can actually directly write the following code to store data in SharedPreferences:

```
getSharedPreferences("data", Context.MODE_PRIVATE).edit {
    putString("name", "Tom")
   .putInt("age", 28)
    putBoolean("married", false)
}
```

As you can easily tell, the difference is just the function name which replaced open with edit, which makes more sense. I named it as open function to prevent duplicate naming with KTX edit function to reduce confusion.

The purpose of creating open function is to help you understand how KTX works, and as KTX can only provide limited APIs, by understanding the mechanism, you can extend limitless APIs.

## 7.6.2 Simplify Use of ContentValues

Let us take a look at how to simplify ContentValues now.

The basic use of ContentValues has been covered in Sect. 7.4. It is used to add and update the data in the database with APIs in SQLiteDatabase as shown in the following code:

```
val values = ContentValues()
values.put("name", "Game of Thrones")
values.put("author", "George Martin")
values.put("pages", 720)
values.put("price", 20.85)
db.insert("Book", null, values)
```

You might think that we can simplify the code here with apply function. This will work but we can do even better.

Before we start the work, I need to introduce something else. Remember the mapOf() function in Sect. 2.6.2? It allows us to use syntax like “Apple” to 1 to quickly create key value pair. Actually in Kotlin the A to B syntax will create Pair object and I will unveil why this is the case in Chap. 9’s Kotlin Class.

With this context, we can start. Create ContentValues.kt and define cvOf() as code below:

```
fun cvOf(vararg pairs: Pair<String, Any?>) : ContentValues {
}
```

This method will create an object of ContentValues. The cvOf() method takes a param of Pair type which can be created by A to B syntax. The vararg keyword is corresponding to variable arguments in Java which allows us to pass 0,1,2 or any number of arguments of Pair type. All these arguments will be stored in pairs, and we can use for-in loop to get all the arguments.

Next, let us take a look at Pair. It is a data structure of key value pair, thus we need to use generics to specify the type of key and value. Luckily, all the keys are string type in ContentValues, and we can specify the key of Pair to String. But since the value of ContentValues has multiple types to choose from (integer, float, string, or even null), we need to set the value to be type of Any?. This is because Any is the base class of all classes in Kotlin which is corresponding to Object in Java. Any? Means that null value is allowed.

Next, let us implement function logic for cvOf(). The basic idea is that we create an object of ContentValues and iterate the pair argument list to get the value and add into ContentValues object and return it. The idea is simple but a question remains: the value of Pair argument is type of Any?. How to match with the data types that ContentValues supports? We can use when statement to check all the supported data types of ContentValues. The code below should help you understand better:

```
fun cvOf(vararg pairs: Pair<String, Any?>) : ContentValues {
    val cv = ContentValues()
    for (pair in pairs) {
        val key = pair.first
        val value = pair.second
        when (value) {
            is Int -> cv.put(key, value)
            is Long -> cv.put(key, value)
            is Short -> cv.put(key, value)
            is Float -> cv.put(key, value)
            is Double -> cv.put(key, value)
            is Boolean -> cv.put(key, value)
            is String -> cv.put(key, value)
            is Byte -> cv.put(key, value)
            is ByteArray -> cv.put(key, value)
            null -> cv.putNull(key)
        }
    }
    return cv
}
```

The code above follows the idea just mentioned. It uses for-in loop to iterate the pairs argument lists and get the key and value in the loop and uses when statement to check the type of value. Notice that we covered all the supported data types of ContentValues. Then the values are added into ContentValues one by one and in the end return the object of ContentValues.

The code above also applies Smart Cast in Kotlin. For instance, here for the code that is in the Int condition block, value will be automatically cast to Int type instead

of Any?. Thus, we do not need to do type casting like Java. This applies to if statement.

With cvOf(), it is easier to use ContentValues. For example, to add data to database we can have the code below:

```
val values = cvOf("name" to "Game of Thrones", "author" to "George
Martin",
    "pages" to 720, "price" to 20.85)
db.insert("Book", null, values)
```

Now we can use syntax like mapOf() to construct the ContentValues object, isn't it great?

Of course, though cvOf() is very helpful, it has nothing to do with higher-order function at all. Because cvOf() takes params of variable argument list that are of Pair type and return ContentValues object. Both the params and return values are not of function type, thus it is not higher-order function.

cvOf() is not a higher-order function, but it can be further simplified with the help of higher-order function. For instance, with apply function, cvOf() can have a more elegant implementation:

```
fun cvOf(vararg pairs: Pair<String, Any?>) = ContentValues().apply {
    for (pair in pairs) {
        val key = pair.first
        val value = pair.second
        when (value) {
            is Int -> put(key, value)
            is Long -> put(key, value)
            is Short -> put(key, value)
            is Float -> put(key, value)
            is Double -> put(key, value)
            is Boolean -> put(key, value)
            is String -> put(key, value)
            is Byte -> put(key, value)
            is ByteArray -> put(key, value)
            null -> putNull(key)
        }
    }
}
```

Since, the return value of apply is the object it is calling, we can use single line function syntax to replace return value declaration with equal sign. Also, Lambda expression in apply will automatically have the context of ContentValues, thus we can call the various put methods of ContentValues here. With higher-order function, do you feel it is more elegant?

You'd probably think that KTX lib may provide something similar and indeed. KTX provides contentValuesOf() which has the same functionality. It can be used as code below:

```
val values = contentValuesOf("name" to "Game of Thrones", "author" to
    "George Martin",
    "pages" to 720, "price" to 20.85)
db.insert("Book", null, values)
```

Again, we can always use the functions in KTX, and the goal here is to help you understand how these functions get implemented so that you know how to create such functions when you need them but KTX does not offer.

## 7.7 Summary and Comment

This is a long chapter, and it is time to summarize what we have learned in this chapter. This chapter focuses on common approaches for data persistence in Android which includes file, SharedPreferences, and database. Among them, file is a good choice to store some text and binary data; SharedPreferences is a good choice to store key-value pairs while database should be used to store the more complicated relational data. You should select based on your project's need.

In this chapter's Kotlin Class, we did not introduce a lot of new concepts but focused on improving the understanding the use of higher-order functions and extend the existing APIs with higher-order functions and other techniques through two examples.

As mentioned in the Summary and Comment section of previous chapter, after we covered Android data persistence, we will continue to learn about the four main components of Android. In the next chapter, we will discuss ContentProvider.

# Chapter 8

## Share Data Between Apps with ContentProvider



In the last chapter, we discussed Android data persistence which includes file, SharedPreferences, and database. You probably notice that all the data mentioned before is only accessible within the app. Although file and SharedPreferences used to provide MODE\_WORLD\_READABLE and MODE\_WORLD\_WRITABLE to allow other apps to access data, they got deprecated in Android 4.2. Android introduced safer ContentProvider to replace them.

You might ask why do we need to share data with other apps? Of course, this is required and depends on the use case. For instance, login information apparently cannot be shared with other apps, but we should be able to share some other information like the contacts in system Contact app. If we cannot access the contacts in the system Contact, then a lot of the third-party apps will not be able to provide crucial services. Besides Contact, Message, Media, and some other apps also can share data between apps, and they all use ContentProvider to do so, and we will discuss this in this chapter.

### 8.1 Introduction to ContentProvider

ContentProvider is mainly for securely sharing data between apps through systematic mechanisms. It is the standard approach to share data between apps in Android.

Unlike file and SharedPreferences whose read and write mode is globally effective, ContentProvider can specify which part of the data is sharable to ensure safety of private data.

Before we start learning of ContentProvider, we need to talk about another very important topic: Android runtime permissions. This will be applied in the ContentProvider examples and a lot of other scenarios. So, you should master Android runtime permissions.

## 8.2 Runtime Permissions

Permissions in Android is not something new as it existed since the very first version of Android OS. However, it used to be very limited to protect user's safety and privacy. This is especially true for apps coming from some big names. To solve this problem, Android introduced runtime permissions in Android 6.0, and let us discuss this topic in this section.

### 8.2.1 *Android Runtime Permissions in Depth*

Let us first take a look at the previous permission mechanism. In Chap. 6's BroadcastTest project, we touched upon the Android permissions for the first time. In the example, in order to get the permission to listen to the broadcast, we declared permission request in AndroidManifest.xml as code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.broadcasttest">  
  
    <uses-permission android:name="android.permission.  
RECEIVE_BOOT_COMPLETED" />  
    ...  
</manifest>
```

This is because the device start broadcast is a sensitive information, and we have to add the permission declaration in the AndroidManifest to prevent crash of the app.

So the question is, what is the impact to the user and why this can help protect the safety of user device?

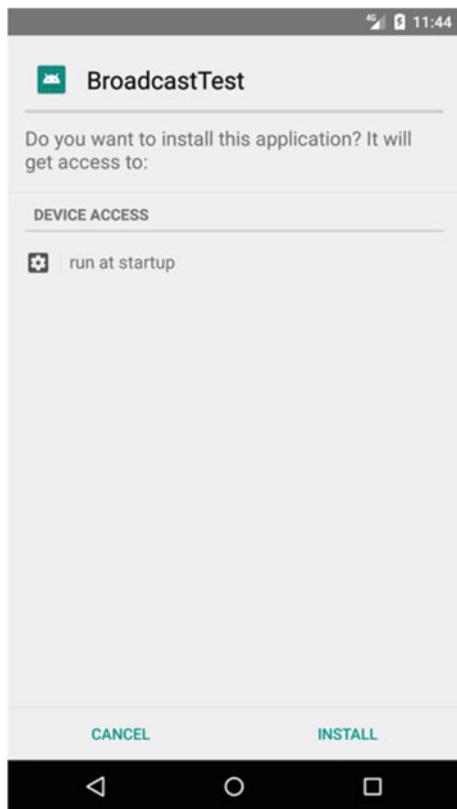
Users get protected from two ways. First, if the user installs the app in devices with OS version earlier than Android 6.0, then the install windows will show alert as shown in Fig. 8.1. This can help user understand what permissions this app is requesting and then decide if they really need to install this app.

On the other hand, user can check the permissions that an app requests in the app management UI as shown in Fig. 8.2. This keeps the permissions status of the app transparent to the user.

The idea here is simple, if user allows the permission request, then install the app if not just refuse to install.

But this only works in an ideal world. A lot of the apps we use abuse the permissions, and they will request the permissions that they really should not use. For instance, Fig. 8.3 shows the permission request of xx app.

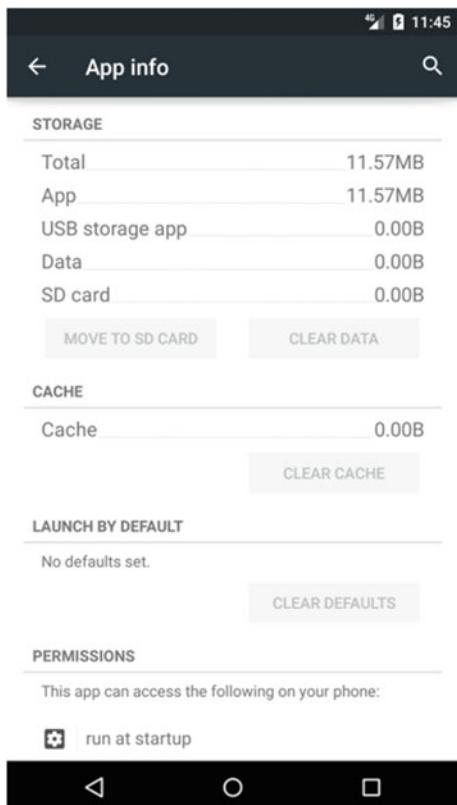
This is just half of the permissions that wechat requests as one screen can only show around half of the requests. I do not think some of the permission request

**Fig. 8.1** Installer UI

makes sense for example, why wechat needs to read my message? However, I cannot refuse to install because I need to use it to communicate with my friends.

Android team also realized this problem and introduced runtime permissions in Android 6.0. This means that user does not need to grant all the permission requests when installing the app. Instead, user can grant permission request for the permission request when the app is running. For instance, if a camera app requests geo location permission, even the user declines this request, user is still able to use other functionality of this app instead of not being able to install at all.

Of course, not all the permissions need to request during runtime, and it is inconvenient for the user to grant permission request constantly. Android now divide the commonly used permissions into three categories: normal permissions, runtime permissions, and special permissions. Special permissions are rarely used, and we will not discuss about it in this book. The normal permissions are deemed not dangerous for user safety and privacy. For these permission request, system can grant for us, and user does not need to do anything. For instance, the permission request in BroadcastTest project is normal permission. The runtime permissions or dangerous permissions are those that can expose risk to user safety and privacy, for

**Fig. 8.2** Management UI

example, device contacts, geo location, and so on, they all need to be granted by the user otherwise, app cannot access the data.

There are more than a hundred permissions in Android, how do we know what are normal permissions and what are dangerous permissions? So there are only a few dangerous permissions and most of the other permissions are normal permissions. Table 8.1 lists all the dangerous permissions in Android 10 which has 11 groups and 30 permissions.

Most of the permissions here may look new to you but you actually do not need to know all of them. Instead just use it as a reference and whenever you need to request certain permission, just check in this table, and if the permission is in this table, you need to request runtime permission. Otherwise, you can just declare the permission in `AndroidManifest.xml`.

Notice that, we should use permission name instead of group name, and for the dangerous permissions in the same permission group, if one permission gets granted then other permissions in that group will also get granted by the system. However, keep it in mind that do not implement your logic based on this rule because Android system can change the grouping at any time.

**Fig. 8.3** Permission listA screenshot of the 'System Permissions' screen. At the top right is a back arrow and the title 'System Permissions'. Below the title is a horizontal line with a central gap. The list of permissions is as follows:

- Location**: Used for sending or sharing location. Provides location-related recommendations and search services, such as Nearby, Shake, and Channels.
- Contacts**: Used for uploading and matching mobile contacts.
- Storage**: Used for sending photos, media, and files, etc. from your device.
- Microphone**: Used for sending voice messages and making voice/video calls.
- Camera**: Used for taking and sending photo or video, making video calls, and scanning.
- Physical Activity**: Used for accessing step count data to join WeRun rankings.
- Floating Window**: Used for minimizing voice/video calls, live streams, and incoming call alerts.

That is everything for Android permission mechanism and let us see how to request permission during runtime.

### 8.2.2 Request Permission at Runtime

First, create `RuntimePermissionTest` project to allow user experiment with runtime permissions. All the permissions in Table 8.1 are available for us to experiment with and for simplicity, we will use `CALL_PHONE` as an example.

`CALL_PHONE` is dangerous permission and used for apps that need to make phone calls because of fees and safety concern. Before Android 6.0, to implement phone call function was effortless. Update activity\_main.xml as code below:

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

**Table 8.1** Dangerous permissions in Android 10

| Permission group     | Permission                 |
|----------------------|----------------------------|
| CALENDAR             | READ_CALENDAR              |
|                      | WRITE_CALENDAR             |
| CALL_LOG             | READ_CALL_LOG              |
|                      | WRITE_CALL_LOG             |
|                      | PROCESS_OUTGOING_CALLS     |
| CAMERA               | CAMERA                     |
| CONTACTS             | READ_CONTACTS              |
|                      | WRITE_CONTACTS             |
|                      | GET_ACCOUNTS               |
| LOCATION             | ACCESS_FINE_LOCATION       |
|                      | ACCESS_COARSE_LOCATION     |
|                      | ACCESS_BACKGROUND_LOCATION |
| MICROPHONE           | RECORD_AUDIO               |
| PHONE                | READ_PHONE_STATE           |
|                      | READ_PHONE_NUMBERS         |
|                      | CALL_PHONE                 |
|                      | ANSWER_PHONE_CALLS         |
|                      | ADD_VOICEMAIL              |
|                      | USE_SIP                    |
|                      | ACCEPT_HANDOVER            |
| SENSORS              | BODY_SENSORS               |
| ACTIVITY_RECOGNITION | ACTIVITY_RECOGNITION       |
| SMS                  | SEND_SMS                   |
|                      | RECEIVE_SMS                |
|                      | READ_SMS                   |
|                      | RECEIVE_WAP_PUSH           |
|                      | RECEIVE_MMS                |
| STORAGE              | READ_EXTERNAL_STORAGE      |
|                      | WRITE_EXTERNAL_STORAGE     |
|                      | ACCESS_MEDIA_LOCATION      |

```

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

    <Button
        android:id="@+id/makeCall"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Make Call" />

</LinearLayout>

```

We defined a button in layout, and clicking the button will trigger the phone call logic. Update MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        makeCall.setOnClickListener {
            try {
                val intent = Intent(Intent.ACTION_CALL)
                intent.data = Uri.parse("tel:8009378997")
                startActivity(intent)
            } catch (e: SecurityException) {
                e.printStackTrace()
            }
        }
    }
}

```

In the button click event handler, we constructed an implicit Intent and its action is Intent.ACTION\_CALL, then we set the data to be an Uri with protocol to be tel and phone number to be 8009378997. In fact, we have already seen this part of the code in Sect. 3.3.3, except that the action specified at that time was Intent.ACTION\_DIAL, which means that the dialing interface is opened, and this does not need to declare permissions, while Intent. In addition, in order to prevent the program from crashing, we put all operations in the exception catch block.

Next update AndroidManifest.xml and declare permission as code below:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.runtimepermissiontest">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
    </application>

</manifest>

```

That is all we need to implement for phone call function, and the above code can work in devices with Android versions earlier than Android 6.0. However, if we run the app in phones with Android 6.0 or later versions, clicking Make Call button will do nothing, and you can see error message as shown in Fig. 8.4 from Logcat.

```

java.lang.SecurityException: Permission Denial: starting Intent { act=android.intent.action.CALL
    at android.os.Parcel.createException(Parcel.java:2069)
    at android.os.Parcel.readException(Parcel.java:2037)
    at android.os.Parcel.readException(Parcel.java:1986)
    at android.app.IActivityTaskManager$Stub$Proxy.startActivity(IActivityTaskManager.java:3827)
    at android.app.Instrumentation.execStartActivity(Instrumentation.java:1705)
    at android.app.Activity.startActivityForResult(Activity.java:5173)
    at androidx.fragment.app.FragmentActivity.startActivityForResult(FragmentActivity.java:767)
    at android.app.Activity.startActivityForResult(Activity.java:5131)
    at androidx.fragment.app.FragmentActivity.startActivityForResult(FragmentActivity.java:754)
    at android.app.Activity.startActivity(Activity.java:5502)
    at android.app.Activity.startActivityForResult(Activity.java:5470)
    at com.example.runtimepermissiontest.MainActivity$onCreate$1.onClick(MainActivity.kt:19)

```

**Fig. 8.4** Error message in Logcat

The error message said Permission Denial, and this is because we have to treat dangerous permission as runtime permissions for Android 6.0 and above.

Let us try to fix this problem by updating MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        makeCall.setOnClickListener {
            if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.CALL_PHONE) != PackageManager.
PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(this,
                    arrayOf(Manifest.permission.CALL_PHONE), 1)
            } else {
                call()
            }
        }
    }

    override fun onRequestPermissionsResult(requestCode: Int,
        permissions: Array<String>, grantResults: IntArray) {

        super.onRequestPermissionsResult(requestCode, permissions,
        grantResults)
        when (requestCode) {
            1 -> {
                if (grantResults.isNotEmpty() &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                    call()
                } else {
                    Toast.makeText(this, "You denied the permission",
                        Toast.LENGTH_SHORT).show()
                }
            }
        }
    }
}

```

```
private fun call() {
    try {
        val intent = Intent(Intent.ACTION_CALL)
        intent.data = Uri.parse("tel:10086")
        startActivity(intent)
    } catch (e: SecurityException) {
        e.printStackTrace()
    }
}
```

The code above covers the end to end flow of runtime permission. Essentially runtime permission is to ask user to grant permission for the app to do risky action while the app is running. So the first step is check if the permission is granted or not by ContextCompat.checkSelfPermission(). This method takes two params, the first param is Context and the second param is the name of the permission, for example, Manifest.permission.CALL\_PHONE. Then we can compare the returned value with PERMISSION\_GRANTED to see if the user has granted this permission or not.

If permission granted then we can continue executing the calling logic, and here we encapsulate the calling logic in call(). If permission not granted, then we can use ActivityCompat.requestPermissions() to request permission. This method takes three params: the first param should be an instance of Activity; the second param is a String array which will host the permission name we're requesting; and the third param is the request code which only requires a unique value and here we pass in 1.

requestPermission() will bring up a permission request dialog to let the user grant or decline our request. The result will be encapsulated in grantResults argument and passed to onRequestPermissionsResult() callback. Then we can check the final result of the request, if user grants the request then we can call call() to make phone calls, if not then we have to give up and show a message to the user.

Run the app again and click Make Call, the result should be as shown in Fig. 8.5.

Since this is the first time, thus permission is not granted yet and the dialog for permission will show up. If we click Deny, the result will be as shown in Fig. 8.6.

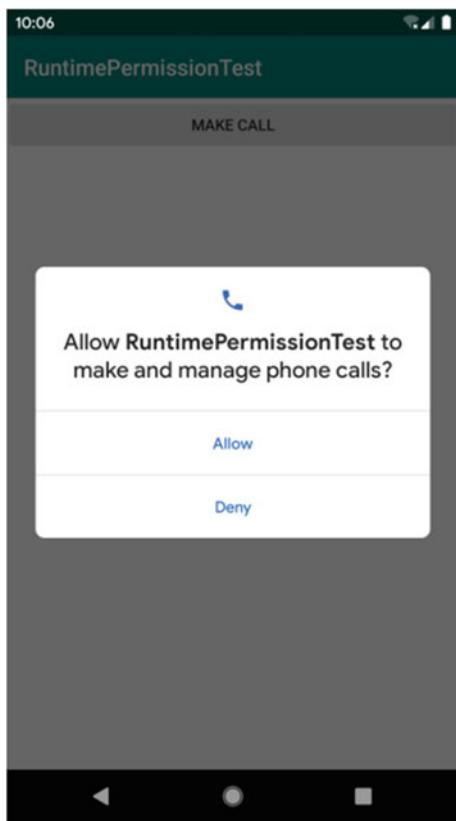
Since permission is not granted, we can only show a message to notify operation failed. Next time we click Make Call, the dialog for permission request will still show up, click Allow this time and the result will be as shown in Fig. 8.7.

As you can see this time, we can reach to the dial screen, and since permission already gets granted, click Make Call button will not trigger the permission request dialog and will call directly. What if user wants to revert the decision? No problem! They can revoke this permission at any time by Settings → Apps & notifications → RuntimePermissionTest → Permissions as shown in Fig. 8.8.

User can manage the permission as shown in Fig. 8.8.

That is everything for runtime permissions, and you should be able to handle the permission-related issues in Android, and let's start to look at the core topic of this chapter—ContentProvider.

**Fig. 8.5** Dialog for permission request



### 8.3 Access Data in Other Apps

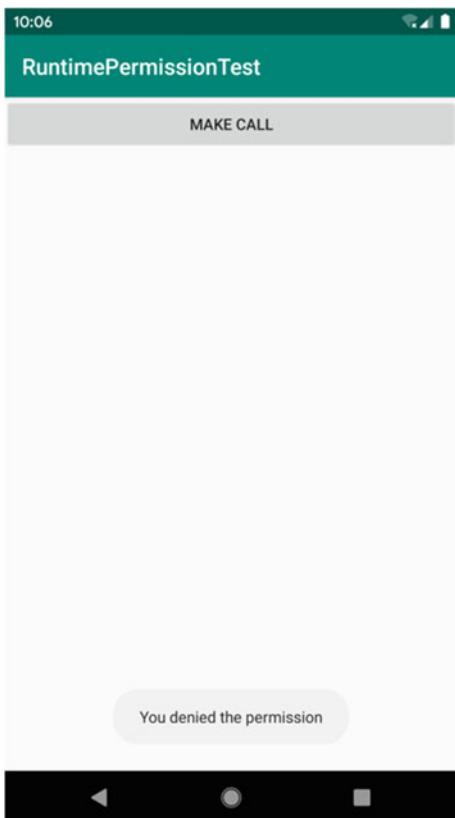
We can use the existing ContentProvider to read and write the data in the corresponding app and also create our own ContentProvider to provide access to other apps. Let us start with using the existing ContentProvider.

If an app provides access to its data with ContentProvider to other apps, then any other app can have access to this part of data. The Android system apps like Contact, Message, and Media, etc. provide similar access interface to help third-party apps implement rich features. Now let us take a look at how to use ContentProvider.

#### 8.3.1 Basic Use of ContentResolver

To access the data shared through ContentProvider, we need to get an instance of ContentResolver which can be acquired by `getContentResolver()` in Context. ContentResolver provides methods to do CRUD: `insert()`, `update()`, `delete()`, and

**Fig. 8.6** User declined permission request



query(). They look so similar right? Yes! SQLiteDatabase also uses these methods to do CRUD, however these methods have different params in these two classes.

Unlike SQLiteDatabase, these CRUD methods in ContentResolver do not take the table name but instead take an Uri param. This param is called content URI. Content URI is the unique identifier of the data in ContentProvider which consists of authority and path. Authority can differentiate different apps and to avoid duplication, we should use package name. For instance, if an app's package name is com.example.app, then the corresponding authority should be com.example.app.provider. Path is to differentiate the tables in the same app which usually gets appended to authority. For instance, if an app has two tables, table1 and table2, then we can name the path as /table1 and /table2 correspondingly, then we can concatenate with authority and the content URI will be com.example.app.provider/table1 and com.example.app.provider/table2 respectively. It is still difficult to tell these two strings are two content URIs, and we need to add protocol at the beginning of the string. The standard format of content URI is as shown below:

```
content://com.example.app.provider/table1  
content://com.example.app.provider/table2
```

**Fig. 8.7** Dial UI

10:06

Signal



Calling...

10086



Mute



Keypad



Speaker



Add call



As you can see, the content URI can clearly express the data in which table and in which app. Thus ContentResolver's CRUD methods take Uri instance as argument. If we use the table name alone, system cannot tell which app to look for the table.

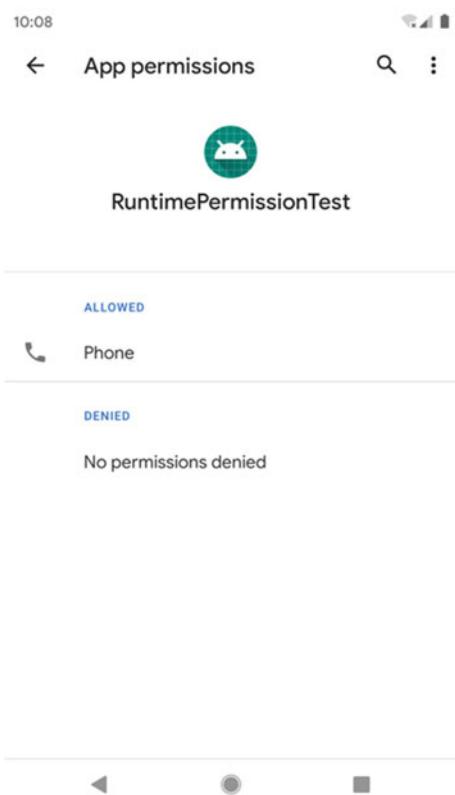
After getting the content URI string, we need to parse the string to Uri instance to use as argument. Parsing is easy and is as shown below:

```
val uri = Uri.parse("content://com.example.app.provider/table1")
```

We just need to call Uri.parse() to parse the content URI string to an instance of Uri.

Now we can use this Uri instance to query the data in table1 as shown below:

```
val cursor = contentResolver.query(  
    uri,  
    projection,  
    selection,  
    selectionArgs,  
    sortOrder)
```

**Fig. 8.8** App permission management screen**Table 8.2** query() Params

| query() params | SQL counterpart           | Description                           |
|----------------|---------------------------|---------------------------------------|
| uri            | from table_name           | Specify the table name                |
| projection     | select column1, column2   | Specify the columns                   |
| selection      | where column = value      | Specify where constraints             |
| selectionArgs  | —                         | Set the value of placeholder in where |
| sortOrder      | order by column1, column2 | Specify the order of results          |

The params are similar to the params of query() in SQLiteDatabase but overall simpler as this is querying data in other apps and complicated query statement is not essential. Table 8.2 explains these params in detail.

The query result is still a Cursor object, and we can use the Cursor object to read the data one by one. The basic idea is still to move the cursor position to iterate all the rows in the Cursor and get the data from each row as shown in the code below:

```

while (cursor.moveToNext()) {
    val column1 = cursor.getString(cursor.getColumnIndex("column1"))
    val column2 = cursor.getInt(cursor.getColumnIndex("column2"))
}
cursor.close()

```

Other CRUD operations will be easier. To add a row in table1 can be done with code below:

```

val values = contentValuesOf("column1" to "text", "column2" to 1)
contentResolver.insert(uri, values)

```

We still use ContentValues to construct the data then use insert() in ContentResolver to add data with Uri and ContentValues.

If we want to update this newly added row to make its column1 to be empty string, we can use update() in ContentResolver as code below:

```

val values = contentValuesOf("column1" to "")
contentResolver.update(uri, values, "column1 = ? and column2 = ?",
arrayOf("text", "1"))

```

Notice that, the above code uses selection and selectionArgs to constraint the data that will be updated.

Lastly, we can use delete() in ContentResolver to delete this row with code below:

```
contentResolver.delete(uri, "column2 = ?", arrayOf("1"))
```

That is everything for CRUD methods in ContentResolver, isn't it easy to understand? Because this knowledge was already mastered when you learned about SQLiteDatabase in the previous chapter, the only parameter you need to pay special attention to is the uri param. Next, let us apply what we just learned and explore how to read the contact information in system Contact app.

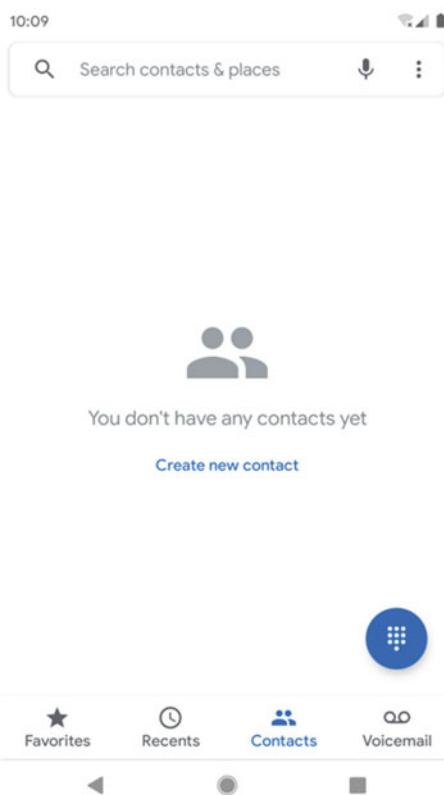
### 8.3.2 *Read System Contact*

The system Contact app in emulator does not have contact yet, and we need to add by ourselves. Open Contact app as shown in Fig. 8.9.

There is no contact there yet but we can create contact by clicking Create new contact. Let us create two contacts first and fill their names and phone number as shown in Fig. 8.10.

That is all for the preparation work. Now create ContactsTest project and begin the real work.

First, create the layout. We want to show the contacts information in ListView, thus modify activity\_main.xml as code below:

**Fig. 8.9** System contact UI

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

    <ListView
        android:id="@+id/contactsView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </ListView>

</LinearLayout>
```

For simplicity, we just have a ListView in LinearLayout. I choose to use ListView instead of RecyclerView because our focus is on reading the system contacts, and RecyclerView is heavier which can easily distract the attention.

Next modify MainActivity as code below:

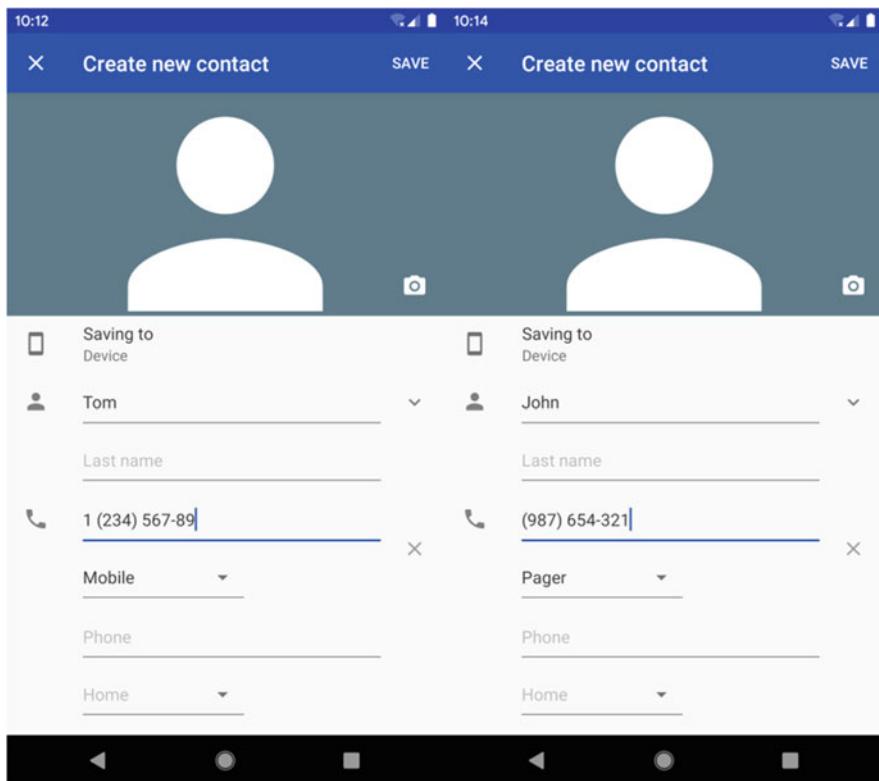


Fig. 8.10 Create two contacts

```
class MainActivity : AppCompatActivity() {  
  
    private val contactsList = ArrayList<String>()  
    private lateinit var adapter: ArrayAdapter<String>  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        adapter = ArrayAdapter(this, android.R.layout.  
simple_list_item_1, contactsList)  
        contactsView.adapter = adapter  
        if (ContextCompat.checkSelfPermission(this, Manifest.  
permission.READ_CONTACTS)  
            != PackageManager.PERMISSION_GRANTED) {  
            ActivityCompat.requestPermissions(this,  
                arrayOf(Manifest.permission.READ_CONTACTS), 1)  
        } else {  
            readContacts()  
        }  
    }  
}
```

```
override fun onRequestPermissionsResult(requestCode: Int,
permissions: Array<String>,
grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions,
grantResults)
    when (requestCode) {
        1 -> {
            if (grantResults.isNotEmpty() && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                readContacts()
            } else {
                Toast.makeText(this, "You denied the permission",
                Toast.LENGTH_SHORT).show()
            }
        }
    }
}

private fun readContacts() {
    // query the contacts
    contentResolver.query(ContactsContract.CommonDataKinds.Phone.
CONTENT_URI,
        null, null, null, null)?.apply {
    while (moveToNext()) {
        // get contact name
        val displayName = getString(getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME))
        // get contact phone number
        val number = getString(getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.NUMBER))
        contactsList.add("$displayName\n$number")
    }
    adapter.notifyDataSetChanged()
    close()
}
}
```

In onCreate(), we initialize ListView using the standard approach then we start to handle the runtime permission request logic. This is because READ\_CONTACTS is a runtime permission, and the logic here should be familiar to you. If we can get the permission then we will call readContacts() to read the contacts' information in system Contact app.

Next let us focus on readContacts(), as you can see we used query() in ContentResolver to query the contact data. But why does the Uri argument look strange? Why does Uri.parse() not get called to parse the content URI string? This is because ContactsContract.CommonDataKinds.Phone has already encapsulated this and provides a CONTENT\_URI constant which is the result of Uri.parse(). We then iterate over the Cursor object returned by the query() method, using the ?. operator and the apply function to simplify the traversal code. In apply function, we get the

contact name and phone number one by one. The column for contact name is `ContactsContract.CommonDataKinds.Phone.DISPLAY`, and the column for contact phone number is `ContactsContract.CommonDataKinds.Phone.NUMBER`. We can concatenate the results and between them we add line break, then add the concatenated data to ListView's data source and refresh the ListView. At the end, do remember to close the Cursor object.

But that is not the end. We also need to declare the permission for reading the system Contact. Modify `AndroidManifest.xml` as code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.contactstest">

    <uses-permission android:name="android.permission.READ_CONTACTS"
/>
    ...
</manifest>
```

Essentially, we need to declare permission for `android.permission.READ_CONTACTS` so that we can have access to the system contacts. That is everything we need to run the app the result should be as shown in Fig. 8.11.

It will first pop up a dialog to request permission for access to contacts, click Allow and the result is as shown in Fig. 8.12.

We can see the two newly created contacts' information! This proves that we indeed implemented the function to access data across apps.

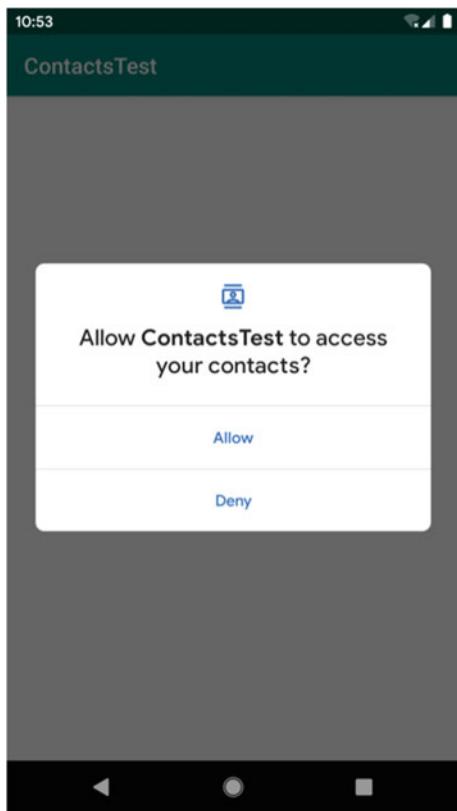
## 8.4 Create Your Own ContentProvider

In the previous section, we learned how to access data from other applications in our own program. In general, the idea is still very simple, you just need to get the content URI of the application and then add, delete, and check with the help of ContentResolver. But have you ever wondered how those applications that provide external access interfaces achieve this functionality? How do they secure the data so that private data is not leaked out? After learning this section, your doubts will be solved one by one.

### 8.4.1 Create ContentProvider

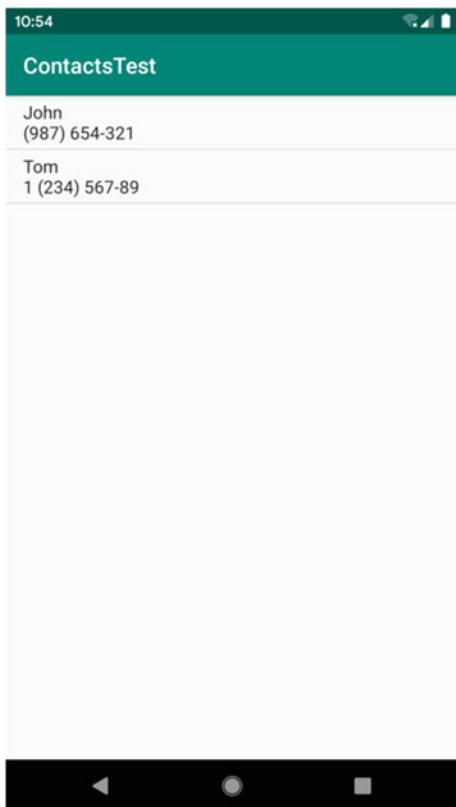
As mentioned before, to share data between apps, we can create a new class that inherits `ContentProvider`. There are six abstract methods in `ContentProvider` class which all need to be overridden. Here is an example:

**Fig. 8.11** Dialog for access to contacts permission



```
class MyProvider : ContentProvider() {  
  
    override fun onCreate(): Boolean {  
        return false  
    }  
  
    override fun query(uri: Uri, projection: Array<String>?, selection:  
String?,  
        selectionArgs: Array<String>?, sortOrder: String?): Cursor? {  
        return null  
    }  
  
    override fun insert(uri: Uri, values: ContentValues?): Uri? {  
        return null  
    }  
  
    override fun update(uri: Uri, values: ContentValues?, selection:  
String?,  
        selectionArgs: Array<String>?): Int {
```

**Fig. 8.12** System contacts information



```
        return 0
    }

    override fun delete(uri: Uri, selection: String?, selectionArgs:
Array<String>?): Int {
    return 0
}

    override fun getType(uri: Uri): String? {
        return null
    }
}
```

You should be familiar with most of these methods, and let me explain them one by one.

`onCreate()`. This method will be called during initialization of ContentProvider. Usually creating and upgrading database logic should reside in this method. It will return a boolean to indicate if the initialization is successful or not.

`query()`. This method is used to query data from ContentProvider. The `uri` argument is to specify the table to query against. The `projection` argument is used to specify the columns to query against. `Selection` and `selectionArgs` arguments are to constraint which rows to query against. `Sortorder` argument is used to order the result of the query. The result will be returned in the `Cursor` object.

`insert()`. This method will add data to ContentProvider. `Uri` argument is used to specify which table the data will be added to. Data that will be added is in the `values` argument. After adding the data, an `Uri` instance that can reference the new record will be returned.

`update()`. This method can update the existing data in the ContentProvider. The `uri` argument is used to specify which table to update. The new data is in the `values` argument, `selection` and `selectionArgs` arguments are used to specify which rows to update. The return value is the number of rows that get affected.

`delete()`. This method is used to delete the data from ContentProvider. The `uri` argument is used to specify the data in which table will be deleted, and the `selection` and `selectionArgs` arguments are used to constrain the rows that will be deleted. The return value is the number of deleted rows.

`getType()`. This method will return the corresponding MIME type based on the passed in content URI.

All of these methods have `uri` argument which is passed from the CRUD method of ContentResolver. Now we need to parse the `uri` argument to get the table and data.

Recall that a standard content URI follows the pattern as shown below:

```
content://com.example.app.provider/table1
```

This means that the caller wants to access the data in `table1` of `com.example.app`. We can also append an `id` to the content URI, for example:

```
content://com.example.app.provider/table1/1
```

This means that caller wants to access the data of `table1` with `id 1` in `com.example.app`.

Content URI mainly has the above two formats. If the `uri` ends with `path`, it means that caller wants to access all the data in the table. If the `uri` ends with `id`, then it means caller wants to access the data with the corresponding `id`. We can use wild cards in `uri` too, with the following rules:

`token` means string of any length.

`#token` means number of any length.

Thus, a content uri that can match any table can be as follows:

```
content://com.example.app.provider/*
```

A content uri that can match any row of data in `table1` can be as follows:

```
content://com.example.app.provider/table1/#
```

With the help of UriMatcher, we can implement different ways to match content Uri. UriMatcher provides addUri() which takes three params that allows authority, path, and code to be passed in. Then when we call match() of UriMatcher, we can pass in an instance of Uri and the return value is the code that matches this uri instance. With the code, we can derive the data that the caller expects to access. Update MyProvider as code below:

```
class MyProvider : ContentProvider() {

    private val table1Dir = 0
    private val table1Item = 1
    private val table2Dir = 2
    private val table2Item = 3

    private val uriMatcher = UriMatcher(UriMatcher.NO_MATCH)

    init {
        uriMatcher.addURI("com.example.app.provider", "table1",
        table1Dir)
        uriMatcher.addURI("com.example.app.provider ", "table1/#",
        table1Item)
        uriMatcher.addURI("com.example.app.provider ", "table2",
        table2Dir)
        uriMatcher.addURI("com.example.app.provider ", "table2/#",
        table2Item)
    }
    ...
    override fun query(uri: Uri, projection: Array<String>?, selection:
    String?,
        selectionArgs: Array<String>?, sortOrder: String?): Cursor? {
        when (uriMatcher.match(uri)) {
            table1Dir -> {
                // query all data in table1
            }
            table1Item -> {
                // query single row of data in table1
            }
            table2Dir -> {
                query all data in table2
            }
            table2Item -> {
                // query single row of data in table2
            }
        }
        ...
    }
    ...
}
```

The highlighted code adds four integer variables, table1Dir means all the data in tabel1, table1Item means a single row of data in table1, table2Dir means all the data in table2, table2Item means a single row in table2. Next, UriMatcher instance gets created when MyProvider gets instantiated and addUri() adds the content URI format that will match. Notice that the path param can use wild card. Then when query() gets called, match() of UriMatcher will try to match with the passed in Uri instance, and if certain content URI format in UriMatcher matches this Uri instance, then the corresponding code will be returned, and we can tell what data the caller expects to access.

The above code only uses query() as example, insert(), update(), and delete() are all similar which will take uri param and then use match() of UriMatcher to determine which table the caller is trying to access and then execute the corresponding operation.

There is a new method: getType(). It is a method that all ContentProvider subclasses have to override which is used to get the MIME type of the Uri instance that it is referring to. A content URI's corresponding MIME string consists of three parts and Android has the following format requirement to these three parts.

- Must start with vnd;
- If content URI ends with path, then android.cursor.dir/ needs to be appended; if ends with id, then android.cursor.item/ needs to be appended.
- Append vnd.<authority>.<path> at the end.

Thus for content URI content://com.example.app.provider/table1/1, a legit MIME type can be:

```
vnd.vnd.android.cursor.dir/vnd.com.example.app.provider.table1
```

For content URI content://com.example.app.provider/table1/1, a legit MIME type can be

```
vnd.android.cursor.item/vnd.com.example.app.provider.table1
```

Now, we can continue to enrich MyProvider by implementing getType() as code below:

```
class MyProvider : ContentProvider() {  
    ...  
    override fun getType(uri: Uri) = when (uriMatcher.match(uri)) {  
        table1Dir -> "vnd.android.cursor.dir/vnd.com.example.app.  
provider.table1"  
        table1Item -> "vnd.android.cursor.item/vnd.com.example.app.  
provider.table1"  
        table2Dir -> "vnd.android.cursor.dir/vnd.com.example.app.  
provider.table2"  
        table2Item -> "vnd.android.cursor.item/vnd.com.example.app.  
provider.table2"  
    }
```

```

        else -> null
    }
}

```

With the above code, complete ContentProvider is done and any app can use ContentResolver to access the data in our app. But how to prevent user privacy data from leaking? We actually solved this by ContentProvider's design without realizing it. This is because all the CRUD operation need to match to certain content URI format, and we cannot add URI of privacy data in UriMatcher; thus privacy data is not accessible to other apps and there is no security concern. After finishing all the steps of creating ContentProvider, let us see how to apply it.

#### 8.4.2 Share Data Between Apps

For simplicity, we continue to work on top of DatabaseTest created in the previous chapter. Let us use ContentProvider to provide access to its data to other apps. Open DatabaseTest project and remove the Toast message logic in MyDatabaseHelper as we cannot use toast when we try to access data in another app. Then create a ContentProvider, right click com.example.databasetest package → New → Other → Content Provider will show a windows as shown in Fig. 8.13.

We set the name of ContentProvider to DatabaseProvider, set authority to com.example.databasetest.provider. The exported property indicates if other apps are allowed to access this ContentProvider. Enabled property indicates if this ContentProvider is enabled or not. Select both of these two options, click Finish to finish creating the ContentProvider.

Next, update DatabaseProvider as code below:

```

class DatabaseProvider : ContentProvider() {

    private val bookDir = 0
    private val bookItem = 1
    private val categoryDir = 2
    private val categoryItem = 3
    private val authority = "com.example.databasetest.provider"
    private var dbHelper: MyDatabaseHelper? = null

    private val uriMatcher by lazy {
        val matcher = UriMatcher(UriMatcher.NO_MATCH)
        matcher.addURI(authority, "book", bookDir)
        matcher.addURI(authority, "book/#", bookItem)
        matcher.addURI(authority, "category", categoryDir)
        matcher.addURI(authority, "category/#", categoryItem)
        matcher
    }
}

```

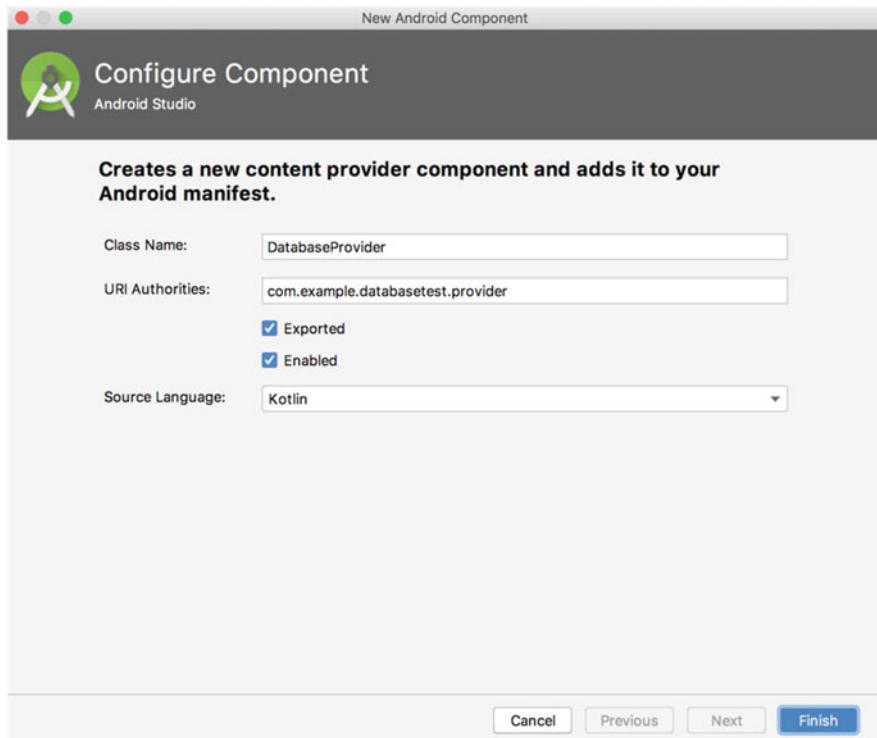


Fig. 8.13 Window to create ContentProvider

```
override fun onCreate() = context?.let {  
    dbHelper = MyDatabaseHelper(it, "BookStore.db", 2)  
    true  
} ?: false  
  
override fun query(uri: Uri, projection: Array<String>?, selection:  
String?,  
    selectionArgs: Array<String>?, sortOrder: String?) = dbHelper?.  
let {  
    // query data  
    val db = it.readableDatabase  
    val cursor = when (uriMatcher.match(uri)) {  
        bookDir -> db.query("Book", projection, selection,  
selectionArgs,  
            null, null, sortOrder)  
        bookItem -> {  
            val bookId = uri.pathSegments[1]  
            db.query("Book", projection, "id = ?", arrayOf(bookId), null,  
null,  
            sortOrder)  
        }  
        categoryDir -> db.query("Category", projection, selection,  
null,  
            sortOrder)  
    }  
}
```

```
selectionArgs,
        null, null, sortOrder)
categoryItem -> {
    val categoryId = uri.pathSegments[1]
    db.query("Category", projection, "id = ?", arrayOf
(categoryId),
        null, null, sortOrder)
}
else -> null
}
cursor
}

override fun insert(uri: Uri, values: ContentValues?) = dbHelper?.let {
    // add data
    val db = it.writableDatabase
    val uriReturn = when (uriMatcher.match(uri)) {
        bookDir, bookItem -> {
            val newBookId = db.insert("Book", null, values)
            Uri.parse("content://$authority/book/$newBookId")
        }
        categoryDir, categoryItem -> {
            val newCategoryId = db.insert("Category", null, values)
            Uri.parse("content://$authority/category/$newCategoryId")
        }
        else -> null
    }
    uriReturn
}

override fun update(uri: Uri, values: ContentValues?, selection: String?,
selectionArgs: Array<String>?) = dbHelper?.let {
    // update data
    val db = it.writableDatabase
    val updatedRows = when (uriMatcher.match(uri)) {
        bookDir -> db.update("Book", values, selection, selectionArgs)
        bookItem -> {
            val bookId = uri.pathSegments[1]
            db.update("Book", values, "id = ?", arrayOf(bookId))
        }
        categoryDir -> db.update("Category", values, selection,
selectionArgs)
        categoryItem -> {
            val categoryId = uri.pathSegments[1]
            db.update("Category", values, "id = ?", arrayOf(categoryId))
        }
        else -> 0
    }
    updatedRows
} ?: 0
```

```

override fun delete(uri: Uri, selection: String?, selectionArgs:
Array<String>?) {
    = dbHelper?.let {
        // delete data
        val db = it.writableDatabase
        val deletedRows = when (uriMatcher.match(uri)) {
            bookDir -> db.delete("Book", selection, selectionArgs)
            bookItem -> {
                val bookId = uri.pathSegments[1]
                db.delete("Book", "id = ?", arrayOf(bookId))
            }
            categoryDir -> db.delete("Category", selection, selectionArgs)
            categoryItem -> {
                val categoryId = uri.pathSegments[1]
                db.delete("Category", "id = ?", arrayOf(categoryId))
            }
            else -> 0
        }
        deletedRows
    } ?: 0
}

override fun getType(uri: Uri) = when (uriMatcher.match(uri)) {
    bookDir -> "vnd.android.cursor.dir/vnd.com.example.
    databaseprovider.book"
    bookItem -> "vnd.android.cursor.item/vnd.com.example.
    databaseprovider.book"
    categoryDir -> "vnd.android.cursor.dir/vnd.com.example.
    databaseprovider.category"
    categoryItem -> "vnd.android.cursor.item/vnd.com.example.
    databaseprovider.category"
    else -> null
}
}

```

Do not get scared by the length of code here as it is actually easy to understand because everything is covered in the last section. First, at the top of the class, we define four variables which will be used for accessing all the data in Book table, one row of data in Book table, all data in Category table, one row of data in Category table correspondingly. Then by lazy code block, we initialize UriMatcher and add the URI format that can match. By lazy block is a lazy loading technique provided by Kotlin, where the code in the block is not executed initially, but only when the uriMatcher variable is called for the first time, and the return value of last line of code in the block is assigned to uriMatcher. We will discuss more about by lazy in the Kotlin's class in this chapter.

Next are the implementations of the abstract methods. First is onCreate() which is short but has some special syntax. Here we use Getter method syntax sugar, ?: operator, let function, ?: operator and single line function syntax sugar. It first call getContext() and use ?: operator and let function to determine if its return value is empty or not: if empty then use ?: operator to return false which indicates that

ContentProvider initialization fails; if not empty then execute the code in let function. In let function, an instance of MyDatabaseHelper gets created and return true to indicate that ContentProvider initialization succeeds. With the help of ternary operators and standard function we expresses the logic in one line and can apply the single line function syntax sugar here, thus we can use equal sign to return the value directly. Other methods have similar syntax structure, and you should not need further explanation.

query() method first gets an instance of SQLiteDatabase, and based on the uri argument to determine which table user is trying to access and then call query() of SQLiteDatabase to do the actual query and return the result by an instance of Cursor. Notice that when try to access a single row of data, getPathSegments() of Uri instance is called. This method will break down the content after authority by “/” and put the results into a string array, thus the value at position 0 is the path, at position 1 is the id. After id is acquired, selection and selectionArgs can constraint the result and achieve single row query functionality.

insert() also first gets an instance of SQLiteDatabase and based on the uri argument to determine which table user is trying to add data to and use insert() in SQLiteDatabase to actually add the data. Notice that insert() need to return an URI that can represent the newly added data, thus we still need to call Uri.parse() to parse the Content URI to Uri object, and of course this content uri will end with the id of the newly added data.

Update() is very similar. It first gets an instance of SQLiteDatabase and based on the uri argument to determine which table user is trying to update data to and uses update() in SQLiteDatabase to actually update the data. The return value is the number of affected rows.

delete() still first gets an instance of SQLiteDatabase and based on the uri argument to determine which table user is trying to delete data from and uses delete() in SQLiteDatabase to actually delete the data. The return value is the number of affected rows.

The last is getType() which exactly follows the format mentioned in last section and no need to explain here again. That is everything we need for ContentProvider.

One last thing is that, ContentProvider needs to register in AndroidManifest.xml to be actually enabled. But since we use Android Studio shortcut to create the ContentProvider, this step has been automatically done by the IDE. Open AndroidManifest.xml, the code should be as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.databasetest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
```

```
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
<provider
    android:name=".DatabaseProvider"
    android:authorities="com.example.databasetest.provider"
    android:enabled="true"
    android:exported="true">
</provider>
</application>

</manifest>
```

Within the <application> tag, there is a new <provider> tag which is used to register the DatabaseProvider. The android:name property set the class name of DatabaseProvider, android:authorities property set the authority of DatabaseProvider; the enabled and exported property are auto-generated from our selection which means that DatabaseProvider can be accessed by other apps.

Now the DatabaseTest project can share data with other apps and let us try it out. First we need to uninstall the DatabaseTest app from emulator to avoid the effect coming from the existing data. Run the project to install the app again. Close DatabaseTest and create a new project ProviderTest which we will use to access the data in DatabaseTest.

First let us create the layout by modifying activity\_main.xml as shown below:

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

    <Button
        android:id="@+id/addData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add To Book" />

    <Button
        android:id="@+id/queryData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Query From Book" />

    <Button
        android:id="@+id/updateData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update Book" />
```

```

<Button
    android:id="@+id/deleteData"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Delete From Book" />

</LinearLayout>

```

The layout is simple. It has four buttons that will be used to trigger add, query, update, and delete data. Then update MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    var bookId: String? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        addData.setOnClickListener {
            // add data
            val uri = Uri.parse("content://com.example.databasetest.
provider/book")
            val values = ContentValues("name" to "A Clash of Kings",
                "author" to "George Martin", "pages" to 1040, "price" to 22.85)
            val newUri = contentResolver.insert(uri, values)
            bookId = newUri?.pathSegments?.get(1)
        }
        queryData.setOnClickListener {
            // query data
            val uri = Uri.parse("content://com.example.databasetest.
provider/book")
            contentResolver.query(uri, null, null, null, null)?.apply {
                while (moveToNext()) {
                    val name = getString(getColumnIndex("name"))
                    val author = getString(getColumnIndex("author"))
                    val pages = getInt(getColumnIndex("pages"))
                    val price = getDouble(getColumnIndex("price"))
                    Log.d("MainActivity", "book name is $name")
                    Log.d("MainActivity", "book author is $author")
                    Log.d("MainActivity", "book pages is $pages")
                    Log.d("MainActivity", "book price is $price")
                }
                close()
            }
        }
        updateData.setOnClickListener {
            // update data
            bookId?.let {
                val uri = Uri.parse("content://com.example.databasetest.
provider/
                book/$it")
                val values = ContentValues("name" to "A Storm of Swords",

```

```

        "pages" to 1216, "price" to 24.05)
        contentResolver.update(uri, values, null, null)
    }
}
deleteData.setOnClickListener {
    // delete data
    bookId?.let {
        val uri = Uri.parse("content://com.example.databasetest.
provider/
        book/$it")
        contentResolver.delete(uri, null, null)
    }
}
}
}

```

The above code adds the CRUD logic in the corresponding event handling code. When adding the data, first call Uri.parse() to parse content URI to Uri object and put the data in ContentValues object, then use insert() of ContentResolver to actually add data. Notice that insert() will return an Uri object which contains the id of the newly added data which we can get by using getPathSegments() and use it later.

When querying the data, still first call Uri.parse() to parse content URI to Uri object and then use query() of ContentResolver to actually query data. The result will be in the Cursor object. We iterate the cursor to get the result and print the data row by row.

When updating the data, first call Uri.parse() to parse content URI to Uri object and put the data in ContentValues object, then use update() of ContentResolver to actually update data. Notice that we have added an id to the end of the content URI that was returned when the data was added, in order not to affect the other rows in the Book table when calling the Uri.parse() method. This means that we only want to update the newly added row of data and no other rows of Book table will be affected.

When deleting the data, still first call Uri.parse() to parse content URI with id at the end to Uri object and then use delete() of ContentResolver to actually delete data. Since, we specified an id in the content URI thus only the row with this id will get deleted and other data in Book table will not get affected.

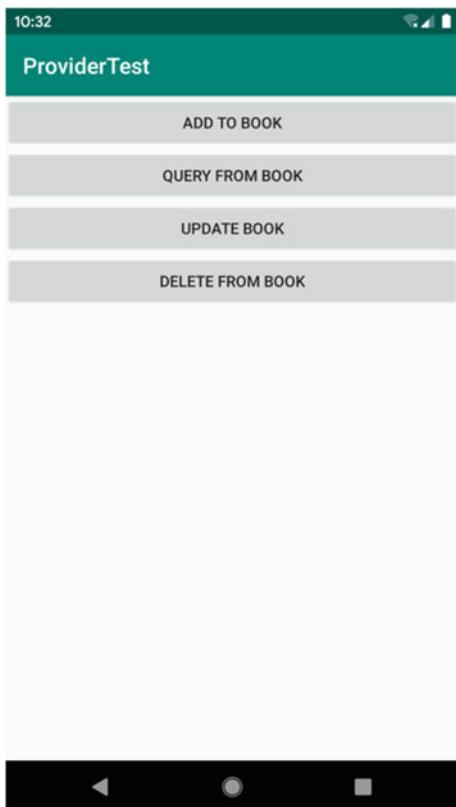
Run ProviderTest should show UI as shown in Fig. 8.14.

Click Add to Book button and data should get added to DatabaseTest database which we can verify by clicking Query From Book button, the Logcat information should be as shown in Fig. 8.15.

Then click Update Book button to update the data and click Query From Book button to verify, the result should be as shown in Fig. 8.16.

Lastly, click Delete From Book button to delete the data and after this, click Query From Book cannot show any data. This proves that we indeed can share data between apps now! Not just ProviderTest, any app can access the data in DatabaseTest and without any concern of privacy leak.

**Fig. 8.14** ProviderTest main UI



com.example.provider (504) ▾ Verbose ▾ com.example.provider D/MainActivity: book name is A Clash of Kings com.example.provider D/MainActivity: book author is George Martin com.example.provider D/MainActivity: book pages is 1040 com.example.provider D/MainActivity: book price is 22.85

**Fig. 8.15** Query added data

com.example.provider (504) ▾ Verbose ▾ com.example.provider D/MainActivity: book name is A Storm of Swords com.example.provider D/MainActivity: book author is George Martin com.example.provider D/MainActivity: book pages is 1216 com.example.provider D/MainActivity: book price is 24.05

**Fig. 8.16** Query Updated Data

That is almost everything important about ContentProvider. Let us start this chapter's Kotlin Class now to learn more Kotlin advanced topics.

## 8.5 Kotlin Class: Generics and Delegate

In this section, we will discuss two advanced topics in Kotlin: generics and delegate. We actually used generics a few times in previous chapters without systematic coverage while delegate is a brand new topic.

### 8.5.1 Basic Use of Generics

Generic is not some brand-new concept in programming. Java introduced generics as early as in version 1.5, and naturally Kotlin supports generics. However, generics in Kotlin do have some difference when compared with Java. In this section, we will learn the basic use of generics which is the same as in Java and then in Chap. 10's Kotlin Class, we will discuss the special functionality of generics belonging to Kotlin.

First, what is generics? Usually we need to specify the type for a variable while generics allows us to write code without specifying the type which makes the code more extensible.

For instance, we can put integers or strings in List because it does not specify any type but instead uses generics. Thus we can construct lists with syntax like List<Int>, List<String>, etc.

Then how do we define our own generics implementation? Let us take a look at the fundamental syntax first.

There are mainly two ways to define generics, one is to define generic class and another one is to define generic method which both use the syntax <T>. The letter T is actually not required but can be any letter or even a word, but it is a convention to use letter T here.

If we need to define a generic class, we can have the code below:

```
class MyClass<T> {  
  
    fun method(param: T) : T {  
        return param  
    }  
  
}
```

Now, MyClass is a generic class, and the method in MyClass takes param of T type and return value of T type.

When the MyClass and its method() is used, we can specify the generic type to a concrete type as shown below:

```
val myClass = MyClass<Int>()
val result = myClass.method(123)
```

If we set the MyClass generic type to be Int type, then method() can take a param of Int type and its return value is also Int type.

If we only want to define a generic method, then we can simply apply the syntax mentioned above to the method as shown below:

```
class MyClass {

    fun <T> method(param: T) : T {
        return param
    }

}
```

And we need to make some adjustment to call the method:

```
val myClass = MyClass()
val result = myClass.method<Int>(123)
```

Now we just need to specify the type when we call method(). Also Kotlin has excellent type inference mechanism and if an Int type param is passed in, it will infer that the generic type should be of Int type, thus we do not need to specifically set the type as shown below:

```
val myClass = MyClass()
val result = myClass.method(123)
```

Kotlin also allows putting constraint to the type of the generics. The code above allow you to specify any type for method(), but if you want to put constraint on it, you can do so. For instance, you can constraint that method() generic type has to be Number type as shown below:

```
class MyClass {

    fun <T : Number> method(param: T) : T {
        return param
    }

}
```

This means that method() generics has to be of Number type like Int, Float, Double, etc. If you specify the type to string, an exception will be thrown as string is not a number.

By default, all the generic types can be nullable types, this is because the default constraint is Any?. Thus if we don't want the generic type to be null, we can set the constraint to be Any.

Next, we try to apply the generic knowledge we learned in this subsection. Recall that in Sect. 6.5.1 in which we discussed higher-order functions, we built build function as shown below:

```
fun StringBuilder.build(block: StringBuilder.() -> Unit) :  
StringBuilder {  
    block()  
    return this  
}
```

This function works like apply function, but build function can only be applied to StringBuilder class while apply function can be applied to any class. Now, let us apply generics to extend the build function and make it work exactly like apply.

The idea is simple, we just need to use <T> to define build function as generic function and replace the StringBuilder to T. Create build.kt with the following code:

```
fun <T> T.build(block: T.() -> Unit) : T {  
    block()  
    return this  
}
```

That's it! Now you can just use build function as using apply function, for example, we can use build function to simplify Cursor iteration:

```
contentResolver.query(uri, null, null, null, null)? .build {  
    while (moveToNext()) {  
        ...  
    }  
    close()  
}
```

That is everything I want to discuss for Kotlin generics which is like Java generics. Next, let us take a look at another important topic of this section—delegate.

### 8.5.2 *Class Delegation and Delegated Properties*

Delegate is a design pattern, and its fundamental idea is that: if some entity cannot handle certain logic then it can delegate this logic to another assisted entity to handle.

This concept may sound strange to Java developers as there is no native support for delegation in Java, while C# and some other languages natively support delegate.

Kotlin also supports delegation and mainly in two ways: class delegation and delegated properties. Let us cover them one by one.

First, let us take a look at class delegation which fundamentally delegates the implementation of a class to another class. In the previous chapter, we used Set data structure which is similar with List except that its data is not ordered and no duplication is allowed. Set is an interface and to use it, we need to use the concrete class that implements its methods for instance HashSet. With the help of delegation, we can easily implement our own implementation class. For instance, let us define MySet class that implement Set interface as shown below:

```
class MySet<T>(val helperSet: HashSet<T>) : Set<T> {  
  
    override val size: Int  
        get() = helperSet.size  
  
    override fun contains(element: T) = helperSet.contains(element)  
  
    override fun containsAll(elements: Collection<T>) = helperSet.  
containsAll(elements)  
  
    override fun isEmpty() = helperSet.isEmpty()  
  
    override fun iterator() = helperSet.iterator()  
  
}
```

The constructor of MySet takes a param of type HashSet which works as assisting object. The implementation of Set interface methods, the corresponding method of HashSet gets called. This essentially is a way of delegation.

So why don't we just use HashSet directly if we essentially call the method in HashSet? The example above is an extreme one. There are cases that we need to write our own implementations, adding new methods besides reusing the assistant object's method and delegation can help in these scenarios.

However, what if there are too many methods in the interface? It will be very painful to call assistant object's method one by one. In Java there is no natively supported solution but in Kotlin class delegation can help resolve this problem.

The keyword for delegation is by Kotlin. We just need to add by keyword after the interface declaration, followed by the assistant object to eliminate the boilerplate code as shown below:

```
class MySet<T>(val helperSet: HashSet<T>) : Set<T> by helperSet {  
}
```

The above code works exactly as the previous example but is more succinct with the help of class delegation. If we need to override the method, then we just need to override that specific method, and all other methods will have the implementation of the assistant object by default as shown below:

```
class MySet<T>(val helperSet: HashSet<T>) : Set<T> by helperSet {  
    fun helloWorld() = println("Hello World")  
  
    override fun isEmpty() = false  
  
}
```

The highlighted code adds helloWorld() and override isEmpty() to let it return false all the time. This is apparently anti-pattern and only for demonstration purpose. Now, MySet becomes a new data structure which will never be empty and can print Hello World. All the other interface methods are the same as HashSet. This is what class delegation in Kotlin can do.

After class delegation, let us take a look at delegation properties. Its fundamental idea is also easy to understand, but to use it flexibly is tricky.

The fundamental idea for class delegation is to delegate the implementation of a class to another class while delegation properties is to delegate the implementation of a field (property) to another class.

The following example shows the syntax of delegation properties:

```
class MyClass {  
  
    var p by Delegate()  
  
}
```

The by keyword connects the p field and the Delegate instance. This means that p field's implementation is delegated to Delegate class. When p field gets read, then getValue() in Delegate will be called, and when p field gets written, setValue() in Delegate will be called.

Thus, we need to implement Delegate class as code below:

```
class Delegate {  
  
    var propValue: Any? = null  
  
    operator fun getValue(myClass: MyClass, prop: KProperty<*>) : Any? {  
        return propValue  
    }  
}
```

```
operator fun setValue(myClass: MyClass, prop: KProperty<*>,
value: Any?) {
    propValue = value
}
```

This is a template of implementation. In Delegate class, we need to implement getValue() and setValue() and use the operator keyword to declare.

getValue() takes two params: the first param is used to specify which class can delegate to this Delegate class, and here we set it to be MyClass; the second param KProperty<\*> is a property operator class in Kotlin which can be used to get the values related to property and is not used here but is required to be in the method param list. <\*> means that you don't know or don't care for the type of the generics and just want to compile successfully which is like <?> in Java. The return value can be any type based on the implementation logic, the above code is just for demonstration purpose.

setValue() is similar except that it takes three params. The first two params are the same as in getValue(), the last param is the value that will be assigned to delegation property which should be the same type of the return value of getValue().

The delegation property work flow is like this: when we assign value to p property of MyClass, then setValue() of Delegate will be called and when we try to get the value of p property, then getValue() of Delegate will be called.

If p field is declared as val then we do not need to implement setValue(). This should be easy to understand as if p is a val, then it cannot be reassigned after initialization thus there is no need to implement setValue(), we only need to implement getValue().

As mentioned previously, delegation itself is easy to understand but the tricky part is to use it flexibly. Next, let us use an example to practice how to use delegation.

### 8.5.3 *Implement Lazy Function*

When we initialized uriMatcher variable in Sect. 8.4.2, we used lazy loading which can delay the execution of the code in the by lazy code block. The code in the by lazy block will not get executed until uriMatcher variable is called for the first time.

With the understanding of Kotlin delegation, we can explore the mechanism under the hood. The syntax for by lazy is as follows:

```
val p by lazy { ... }
```

Now look at this code again, doesn't it make more sense? In fact, by lazy is not a keyword linked together. Only by is the keyword in Kotlin, and lazy is just a higher-

order function here. In the lazy function, a Delegate object is created and returned. When we call the p property, we are actually calling the getValue() method of the Delegate object, and then the getValue() method calls the Lambda expression passed in by the lazy function. The code in the expression can then be executed, and the resulting value obtained after calling the p property is the return value of the last line of code in the Lambda expression.

After demystification, it looks like Kotlin lazy load is not difficult at all, and let us implement our own lazy function.

Create Later.kt with the following code:

```
class Later<T>(val block: () -> T) { }
```

We first define a generic class Later. The Later constructor takes a param of function type, and this function does not take any param with return value of the type that Later specifies.

Next, implement getValues() with the following code:

```
class Later<T>(val block: () -> T) {
    var value: Any? = null
    operator fun getValue(any: Any?, prop: KProperty<*>): T {
        if (value == null) {
            value = block()
        }
        return value as T
    }
}
```

The highlighted code set the first param of getValue() to type of Any? which allows the delegation of Later applicable to all classes. Then it uses value variable to store the value, and if value is empty then call the function passed from constructor to get the value, otherwise return the value directly.

Since lazy load does not assign value to the property, we do not need to implement setValue().

That is everything needed for delegation properties. The code is written so that the delegate property is complete. Although we can use it immediately, it is better to define another top-level function in order to make its usage more similar to that of a lazy function. This function can be in Later.kt but should be defined outside of Later class since only the functions that are not defined within any class is top-level function as shown below:

```
fun <T> later(block: () -> T) = Later(block)
```

We define this top-level function as generic function and take a param of function type. This top-level function creates an instance of Later class and pass the function type argument to the constructor of Later class.

That is everything we need to do for the later lazy load function and can replace the lazy function directly as shown below:

```
val uriMatcher by later {
    val matcher = UriMatcher(UriMatcher.NO_MATCH)
    matcher.addURI(authority, "book", bookDir)
    matcher.addURI(authority, "book/#", bookItem)
    matcher.addURI(authority, "category", categoryDir)
    matcher.addURI(authority, "category/#", categoryItem)
    matcher
}
```

But how can we verify that the lazy loading of the later function is in effect? We can do it by the following code:

```
val p by later {
    Log.d("TAG", "run codes inside later block")
    "test later"
}
```

We add a log in the later function block. Put this code in any of the Activity and reference p field in the button click event.

You shall find that when Activity gets started, the log in later function is not triggered. Only when you click the button will the log get printed out which means that the log statement gets triggered. When you click the button again, there is no new log because the code in the later block will be only triggered once.

Notice that we only demonstrate the mechanism of how lazy function lazy load, but there are more things in lazy function which we do not cover, such as multi-threading code, empty value handling, etc. Thus, please use the lazy function provided by Kotlin in real world project.

That is everything for this chapter's Kotlin Class, it is time to retro what we have learnt in this chapter.

## 8.6 Summary and Comment

There are not many topics in this chapter and a lot of the times we use the database knowledge that we discussed from previous chapter. Thus, hopefully you should find this chapter easy to understand. In this chapter, we first discussed about Android permission mechanism and how to use runtime permission in Android 6.0 and above. Then, we focus on ContentProvider to learn how to share data across apps, and you should know not only how to access data in other apps but also how to create your own ContentProvider to share data.

However, think twice before you create ContentProvider. Only, when you really need to share data should you create ContentProvider.

In this chapter's Kotlin Class, we discussed generics and delegation. The topics are getting more and more advanced, but you should master them as they are important features of Kotlin especially generics which will be used frequently in later chapters.

After so many chapters of system knowledge, you probably feel bored. In the next chapter, let us change the course and discuss Android multimedia.

# Chapter 9

## Enrich Your App with Multimedia



Phones used to be the device that can only be used to make phone calls and send text messages. Now, phones are playing a more and more important role in our daily lives, and we can have all kinds of entertainment through a mobile phone: we can listen to music during the boring commute; we can watch movies during the trip; we can shoot photos wherever we go.

All of the miscellaneous entertainments need the support of strong multimedia functionality and Android excels in this field. It provides a series of APIs that can help us use the multimedia to create all kinds of apps. In this chapter, we are going to discuss some of the commonly used multimedia techniques.

In previous eight chapters, we have been using the emulator to run the application, however we need a real phone to test the feature. Thus, let us take a look at how to run application in Android phones.

### 9.1 Run Application on Phone

Of course, first you need an Android phone and if you don't have one, now is the time to purchase one.

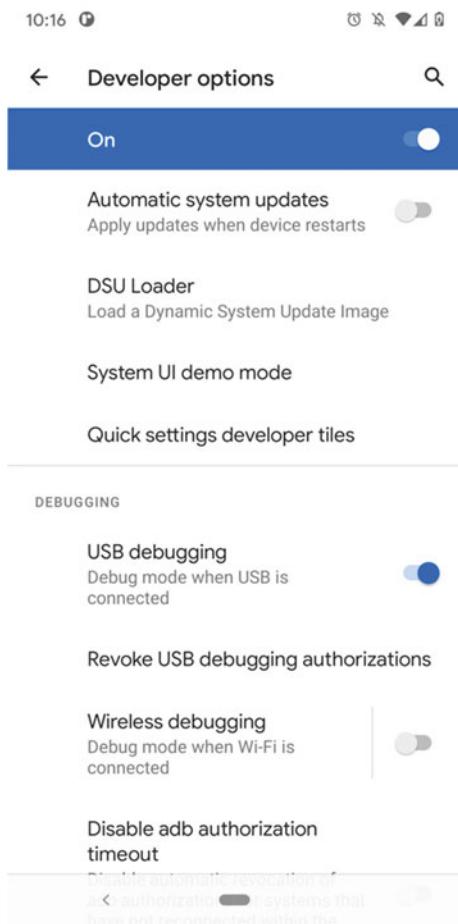
In order to run the app on the phone, we need to connect the phone to the laptop and go to Settings → System → Developer options and enable USB debugging as shown in Fig. 9.1.

Notice that since Android 4.2, the Developer options is hidden by default, and you need to go to About phone settings and continue clicking the version number a few times to show the Developer options.

If it is the first time you are connecting the phone with the computer, on the phone, a dialog will show up to enable USB debugging as shown in Fig. 9.2.

Click Always allow for this computer and click OK then next time when connecting to computer, there will be no dialog showing up.

**Fig. 9.1** Enable USB debugging



Observe Logcat, you will find that we have two devices connected, one is the emulator and another one is the phone just connected as shown in Fig. 9.3.

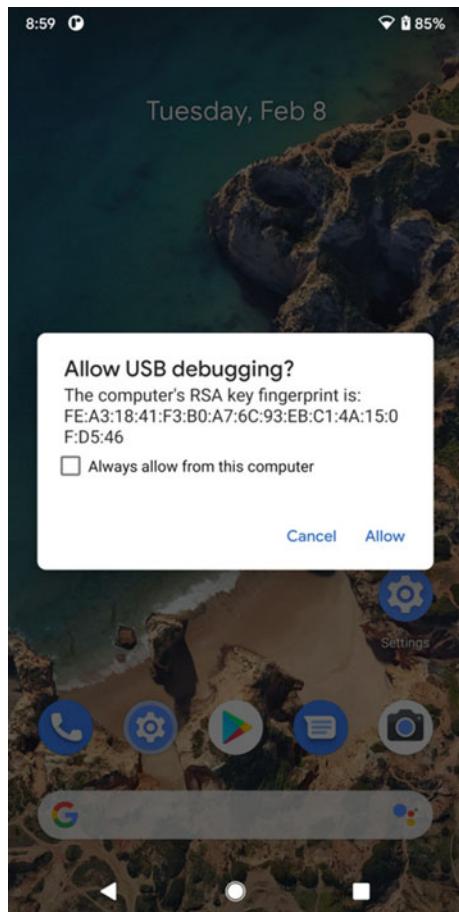
We can select the device that the app is going to run on by using the tool at the top of Android Studio toolbar as shown in Fig. 9.4.

Select Google Pixel to run the app on real device.

## 9.2 Notification

Notification used to be an Android-only feature. When app need to notify user for certain information but not in the foreground, then notification can help to deliver the message. After notification is sent, the top status bar will show an icon for notification and the dropdown menu will show the content of the notification. Notification

**Fig. 9.2** Enable USB debugging dialog



**Fig. 9.3** Connected devices

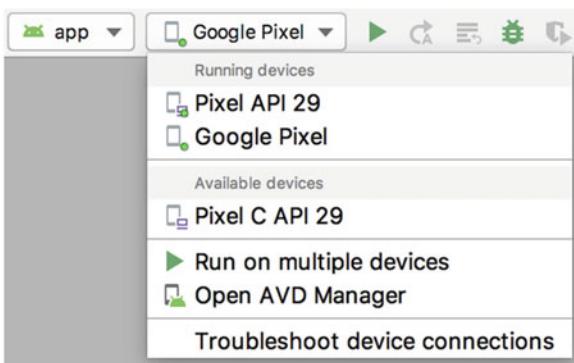


has been a success since it was launched. Even iOS also introduced similar feature after version 5.0.

### 9.2.1 Create Notification Channel

The notification was invented with good will, but certain developers abused it.

**Fig. 9.4** Select the device to run app



Every notification means another chance for opening the app, and many developers tried to send as many notifications as they can to get more opportunity to let the user see the app. From developers' perspective this seems all right, but from the user's perspective, this can be disastrous if all applications are doing the same thing.

Although Android allows user to totally block notification from the selected apps. In these notifications, not all of them should be blocked and some of them may contain the information user is interested. For example, I want to get the notification from the person I'm following but not notifications from random updates from groups. Android users used to have to make a choice to either see all or block all notifications, which was a pain point for Android notification framework.

Then Android 8.0 introduced the concept of notification channel. Each of the notification must be assigned to a channel to display, and developers can create channels for their apps freely. User has the control of the channels and determine how important are the channels and if the specific channel can ring or vibrate and turn on and off of the specific channel.

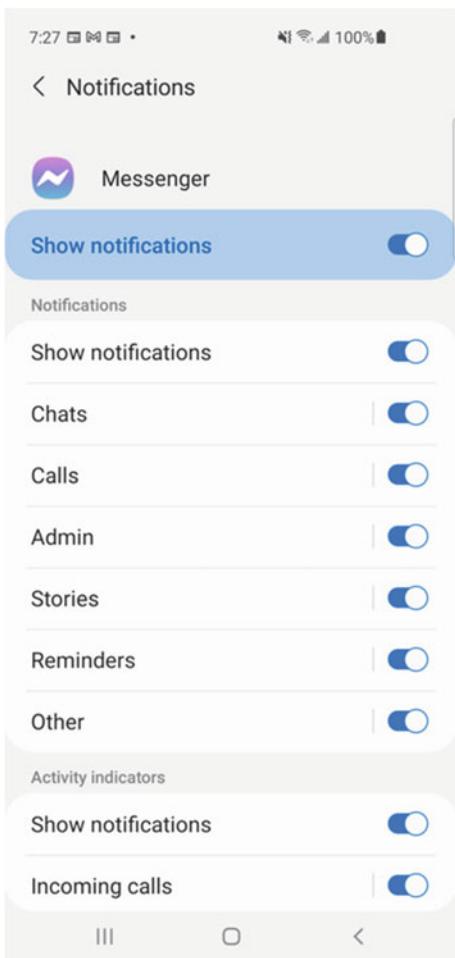
With control at this level, user can choose to receive what kind of notification they want and get rid of the unwanted notifications. Use the previous scenario as an example, Facebook has plenty of channels for the user to manage, like your friends, groups, events, etc. If you do not wish to receive any of the channels, you can just turn it off, thus you can always get the update from friends but not from other channels that you do not care.

Developers need to be considerate when creating channels. Analyze your app carefully and divide your app's notifications into groups before creating the channels. Let us take a look at Messenger's notification channel categorization as shown in Fig. 9.5.

As you can see, Messenger have very granular categorization and provide great freedom for the user to make decisions and greatly reduced the likelihood that users uninstall the app because of notification bombarding.

If our app need to send notification, we have to create our own notification channels, and next let us take a look at how to do so step by step.

**Fig. 9.5** Messenger notification categorization



First, we need `NotificationManager` to manage the notifications which can be acquired by `getSystemService()` in `Context`. This method will take a string param to determine the system service which we will pass in `Context`. `NOTIFICATION_SERVICE`. Thus to get an instance of `NotificationManager` we can use the code below:

```
val manager = getSystemService(Context.NOTIFICATION_SERVICE) as  
NotificationManager
```

Next, we need to use `NotificationChannel` to create a notification channel and call `createNotificationChannel()` in `NotificationManager` to register the notification channel. Since `NotificationChannel` and `createNotificationChannel()` were added in Android 8.0, version check needs to be added which is as shown below:

```

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val channel = NotificationChannel(channelId, channelName,
importance)
    manager.createNotificationChannel(channel)
}

```

To create a notification channel, at least three params are needed: channel ID, channel name, and importance level. Channel ID can be defined freely as long as it is a unique ID. The channel name will be shown to the user and should clearly and accurately describe this channel. The importance level mainly has values of IMPORTANCE\_HIGH, IMPORTANCE\_DEFAULT, IMPORTANCE\_LOW, IMPORTANCE\_MIN, and the order of importance is decreasing. Different importance level will determine different behavior of the notification, and we will demonstrate this later. Of course this is the initial state and user can manually change the importance of the notification channel which developers cannot revert.

### **9.2.2 Basic Use of Notification**

Now, let us take a look at how to use notifications. Notifications can be created in Activity, BroadcastReceiver, and Service, which we will cover later. Creating notification in Activity is rare as only the app is on background should we need to use notification.

We need a Builder to create Notification object, however, almost every version of Android OS will modify the notification more or less, and the instable API issue is particularly serious for notification. For instance, there was no notification channel before Android 8.0. The solution to this has been mentioned a few times previously, which is to use the compatible API in AndroidX library. Using NotificationCompat in AndroidX to create Notification object can ensure that app can work as expected in all versions of Android as shown below:

```

val notification = NotificationCompat.Builder(context, channelId)
build()

```

NotificationCompat.Builder constructor takes two params: the first is context; the second param is the channel ID which needs to match with the ID when we create the notification channel.

Of course, the above code only creates an empty Notification object and is useless. We can enrich the notification object by setting the properties before build(). Some of the basic properties are set as shown below:

```

val notification = NotificationCompat.Builder(context, channelId)
.setContentView("This is content title")
.setContentText("This is content text")
.setSmallIcon(R.drawable.small_icon)

```

```
.setLargeIcon(BitmapFactory.decodeResource(getResources(), R.drawable.large_icon))
.build()
```

These methods are self-explanatory. The title and content can be set by `setContentTitle()` and `setContentText()` correspondingly and can be seen by pulling down the notification bar. The `setSmallIcon()` method is used to set the small icon for the notification. Note that it can only be set using a pure alpha layer image, and the small icon will be displayed on the system status bar. The large icon set by `setLargeIcon()` can be seen by pulling down the notification bar.

Now, we just need to call `notify()` in `NotificationManager` to display the notification. `Notify()` takes two params: the first is the unique id of this notification; the second param is the `Notification` object, and we can pass the object created above. Thus to show a notification, we can use the following code:

```
manager.notify(1, notification)
```

That is the whole process of creating the notification, now let us use an example to demonstrate. Create `NotificationTest` project and update `activity_main.xml` as code below:

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

    <Button
        android:id="@+id/sendNotice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send Notice" />

</LinearLayout>
```

The layout here simply has `Send Notice` button to trigger sending a notification. Next update `MainActivity` as shown below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val manager = getSystemService(Context.NOTIFICATION_SERVICE) as
            NotificationManager
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val channel = NotificationChannel("normal", "Normal",
                NotificationManager.
```

```

        IMPORTANCE_DEFAULT)
        manager.createNotificationChannel(channel)
    }
    sendNotice.setOnClickListener {
        val notification = NotificationCompat.Builder(this, "normal")
            .setContentTitle("This is content title")
            .setContentText("This is content text")
            .setSmallIcon(R.drawable.small_icon)
            .setLargeIcon(BitmapFactory.decodeResource(resources,
                R.drawable.large_icon))
            .build()
        manager.notify(1, notification)
    }
}
}

```

As you can see, we first get an instance of NotificationManager, then create a notification channel and set its ID to normal. The notification channel will not be recreated again and the creation code will not negatively impact performance.

Next, the notification creation and notifying logic are added in the button event handling code as mentioned previously. Notice that since we only have channel with value normal, the channel ID passed to NotificationCompat.Builder constructor has to be normal otherwise the notification will not get displayed. You can use your own icons for the notification icons, but you can also use the images in the source code which you can find in the preface. You can just create drawable-xxhdpi folder and put these images in it.

Run the app and once MainActivity is shown, the notification channel will be created and we can validate this by opening the app settings. Click Settings → Apps & notifications → Notifications as shown in Fig. 9.6.

As you can see, there is a Normal notification channel, which we just created.

Next, go back to our app and click Send Notice button then you shall see a small icon at the left side of the system status bar as shown in Fig. 9.7.

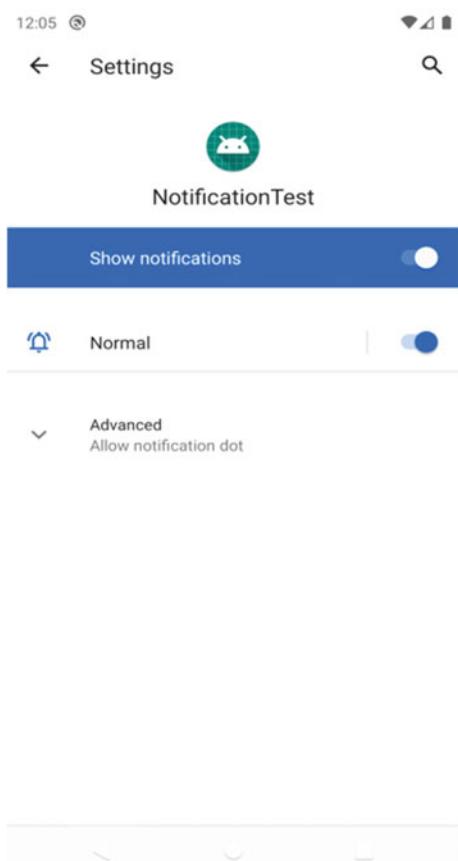
Pull down and you should see the detail content of this notification in notification bar as shown in Fig. 9.8.

If you have ever used Android phone, you might think this notification is clickable. However, if you click it, nothing will happen. In order to implement clicking the notification functions, we need corresponding logic, and we need to learn a new concept—PendingIntent.

PendingIntent is similar with Intent and can be used to start Activity, Service, and send Broadcast. The difference is that intent's action is immediate and instant while PendingIntent is pending until certain event triggered it.

There are a few static methods to get the instance of PendingIntent such as getActivity(), getBroadcast(), and getService(). These methods take the same params: the first param is Context; the second param is rarely used and we can just pass 0; the third param is an object of Intent which is used to set the intent of this PendingIntent; the fourth param is used to set the action of PendingIntent which has

**Fig. 9.6** Create notification channel



values of `FLAG_ONE_SHOT`, `FLAG_NO_CREATE`, `FLAG_CANCEL_CURRENT`, and `FLAG_UPDATE_CURRENT`, you can read the document to see what each value stands for but usually we can just pass in 0.

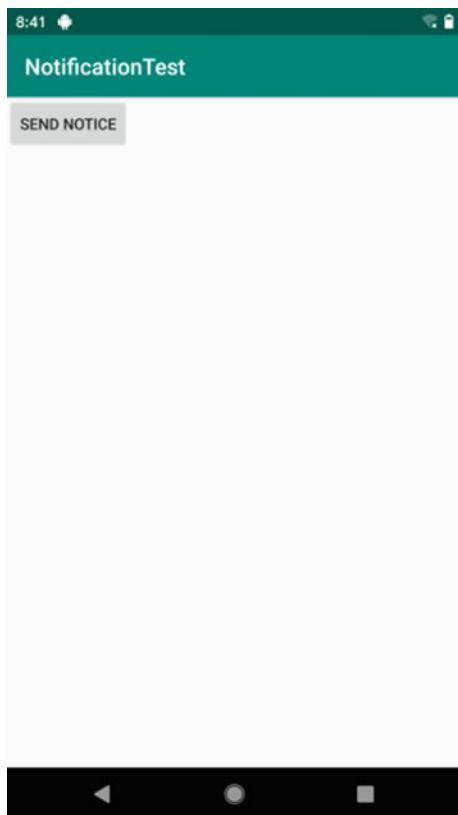
`NotificationCompat.Builder`'s `setContentIntent()` will take a param of `PendingIntent` object and the intent of the `PendingIntent` will be called when user clicks this notification.

Now, let us optimize `NotificationTest` and add click event to the notification that can start another Activity when user click it.

First, create another Activity by right clicking `com.exmaple.notificationtest` package `New → Activity Empty Activity` to create `NotificationActivity`. Modify `activity_notification.xml` as code below:

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

**Fig. 9.7** Notification small icon Fig

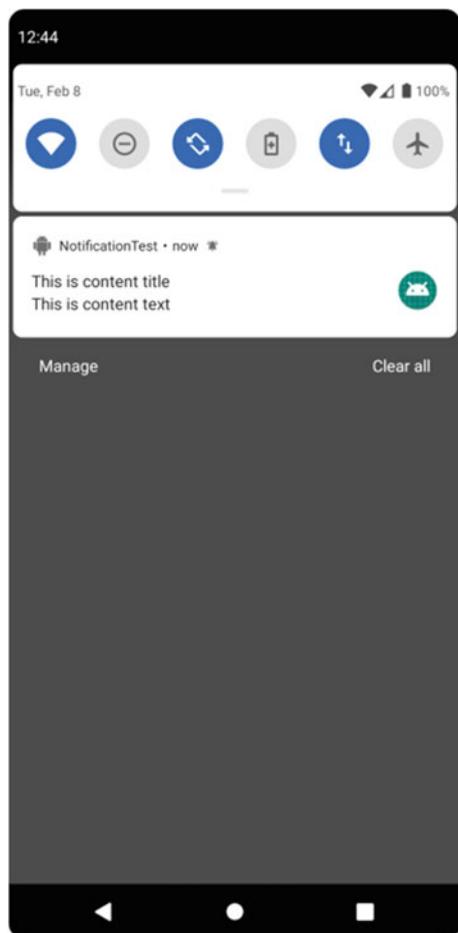


```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_centerInParent="true"  
    android:textSize="24sp"  
    android:text="This is notification layout"  
/>  
  
</RelativeLayout>
```

That's everything we need for `NotificationActivity`. Next update `MainActivity` to add click action as code below:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        sendNotice.setOnClickListener {  
            val intent = Intent(this, NotificationActivity::class.java)  
            val pi = PendingIntent.getActivity(this, 0, intent, 0)
```

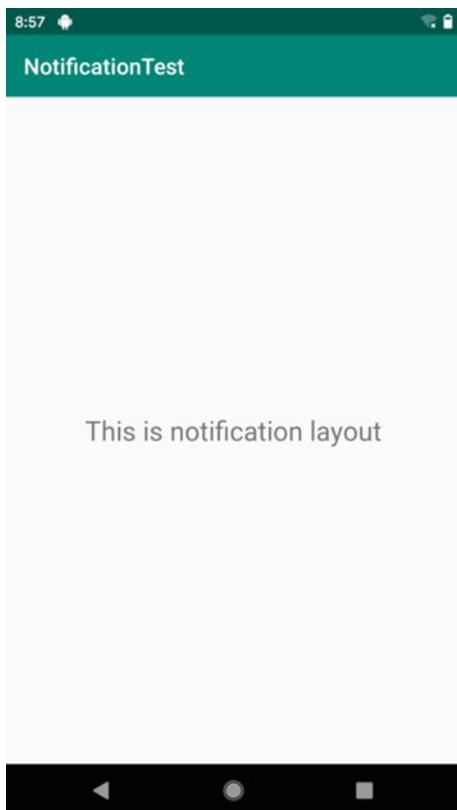
**Fig. 9.8** Notification detail content



```
val notification = NotificationCompat.Builder(this, "normal")
    .setContentTitle("This is content title")
    .setContentText("This is content text")
    .setSmallIcon(R.drawable.small_icon)
    .setLargeIcon(BitmapFactory.decodeResource(resources,
        R.drawable.large_icon))
    .setContentIntent(pi)
    .build()
manager.notify(1, notification)
}
```

}

**Fig. 9.9** Click Notification to open NotificationActivity screen



The highlighted code creates an intent object and passes it to `getActivity()` of `PendingIntent` to get an instance of `PendingIntent`, then passes this instance to `setContentIntent()` in `NotificationCompat.Builder`.

Run the app again and click Send Notice button will display a notification. Pull down the system bar and click the notification to open `NotificationActivity` screen as shown in Fig. 9.9.

You may notice that the notification icon in the system status bar is still there. The reason is because we have not cancelled this notification yet. There are two ways to cancel the notification: one is to `setAutoCancel()` inside `NotificationCompat.Builder`; another way is to call `cancel()` in `NotificationManager`.

The first method is shown as below:

```
val notification = NotificationCompat.Builder(this, "normal")
    ...
    .setAutoCancel(true)
    .build()
```

The highlighted code pass true to setAutoCancel() which means that when the notification gets clicked then it will get cancelled automatically.

The second method is shown as below:

```
class NotificationActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_notification)  
        val manager = getSystemService(Context.NOTIFICATION_SERVICE) as  
        NotificationManager  
        manager.cancel(1)  
    }  
  
}
```

The highlighted code pass 1 to cancel() which is the id of this notification when we create the notification. To cancel a specific notification, just pass the id of the notification in cancel().

### 9.2.3 Advanced Topics in Notification

Now, you know how to create and cancel notification and also how to respond to notification click event. But notification is not limited to these functions and let us explore more about notification.

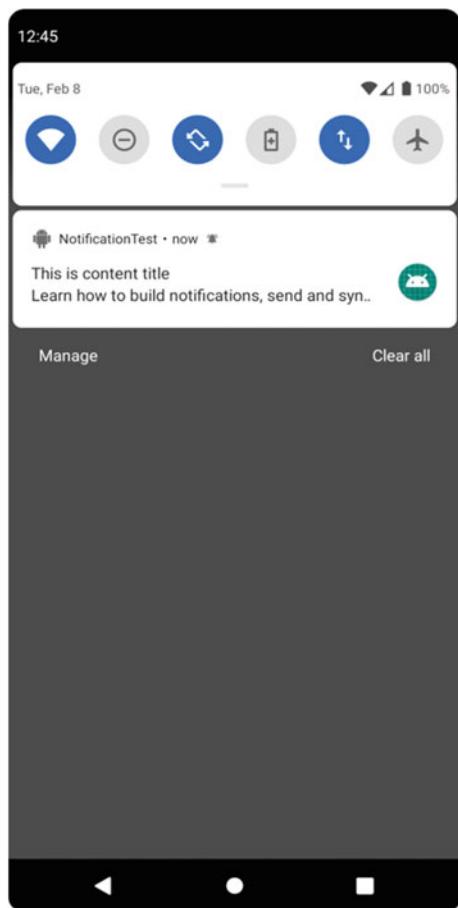
NotificationCompat.Builder provides rich API for use to create colorful notifications. But it's impossible to cover all the APIs and let us take a look at those more commonly used ones.

First, setStyle() method can help us to create rich-text notification content. This means that notification can have more elements besides text and icons. setStyle() takes a param of NotificationCompat.Style type which can be used to construct rich-text message, such as large block of text, image, and so on.

Before we apply setStyle(), let us first take a look what will happen without setStyle(). The previous notifications are short, what will happen if we make the notification as a large block of text? For instance:

```
val notification = NotificationCompat.Builder(this, "normal")  
    ...  
    .setContentText("Learn how to build notifications, send and sync  
data,  
and use voice actions. Get the official Android IDE and developer tools  
to  
build apps for Android.")  
    ...  
    .build()
```

**Fig. 9.10** Display very long text



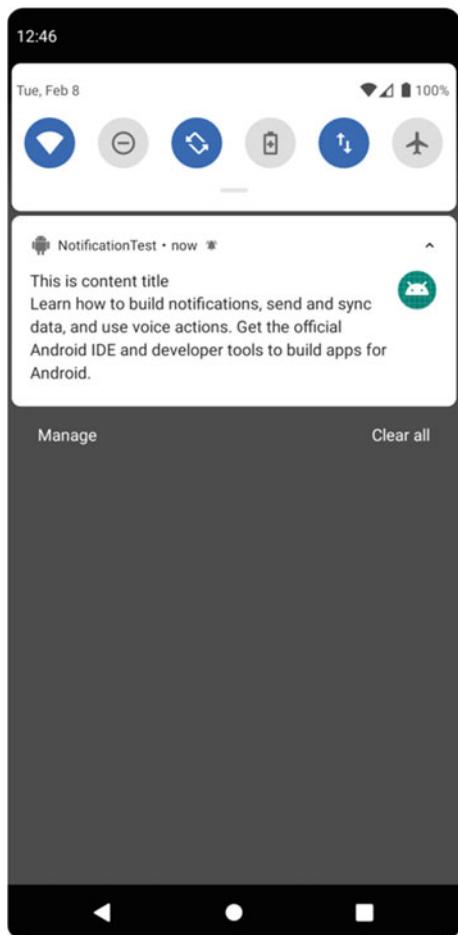
Run the app again and send the notification, the result should be as shown in Fig. 9.10.

As you can see, not all the content can display, which makes sense since the notification content should be concise and the detail should be shown in the Activity that will be opened by the notification.

However, Android does support displaying large blocks of text if needed. This is done with the help of `setStyle()` as shown below:

```
val notification = NotificationCompat.Builder(this, "normal")
    ...
    ..setStyle(NotificationCompat.BigTextStyle())
    .bigText("Learn how to
build
notifications, send and sync data, and use voice actions. Get the
official
```

**Fig. 9.11** Display long text in notification



```
        .setContentText("Android IDE and developer tools to build apps for Android."))  
.build()
```

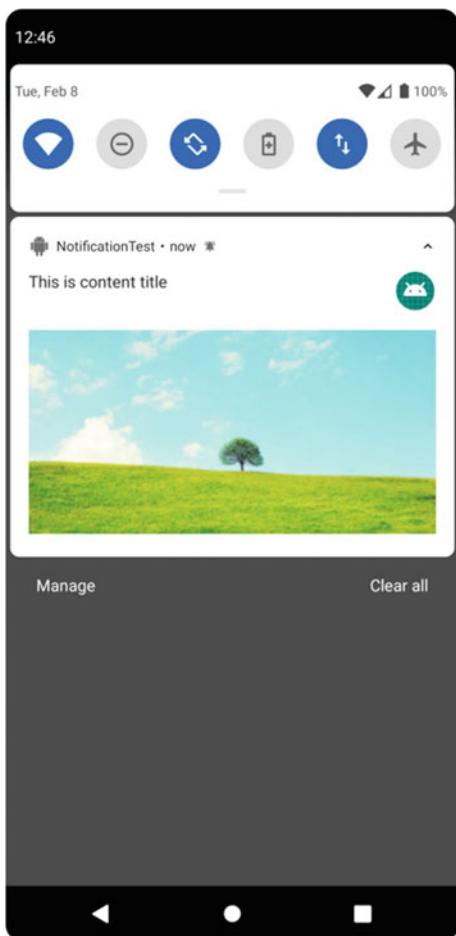
The highlighted code uses `setStyle()` to replace `setContentText()`. In `setStyle()`, we create an object of `NotificationCompat.BigTextStyle` and use its `bigText()` method to set the long text.

Run the app again and send notification, the result is as shown in Fig. 9.11.

Besides large block of text, notification can also display big picture which is used similarly as `bigText` as shown below:

```
val notification = NotificationCompat.Builder(this, "normal")  
    ...  
    .setStyle(NotificationCompat.BigPictureStyle() .bigPicture(  
        BitmapFactory.decodeResource(resources, R.drawable.  
big_image)))  
    .build()
```

**Fig. 9.12** Display big picture in notification



The highlighted code still uses `setStyle()`, but this time we create an object of `NotificationCompat.BigPictureStyle` which can be used by displaying big picture. Then, we pass the picture to its `bigPicture()` method. We can use `decodeResource()` of `BitmapFactory` to decode the image to `Bitmap` object and then pass it to `bigPicture()`.

Run the app again and send notification, the result should be as shown in Fig. 9.12.

That's it for `setStyle()`.

Next, let us take a look at how notification's importance level will affect the behavior of notification. Simply put, the higher importance level a notification has, the easier it will get the attention from user. For example, notifications from higher importance level notification channel can display banner, make some sound while

lower importance level notification can be hidden in certain circumstances, or display after more important ones.

It is worth noting that developers can only assign the importance level when creating the notification channel. User can adjust the importance level whenever they like and developers cannot do anything about it.

Since we cannot change the importance level of the existing notification channel, let us create a new notification channel to test. Update MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
            ...  
            val channel2 = NotificationChannel("important", "Important",  
                NotificationManager.IMPORTANCE_HIGH)  
            manager.createNotificationChannel(channel2)  
        }  
        sendNotice.setOnClickListener {  
            val intent = Intent(this, NotificationActivity::class.java)  
            val pi = PendingIntent.getActivity(this, 0, intent, 0)  
            val notification = NotificationCompat.Builder(this, "important")  
            ...  
        }  
    }  
}
```

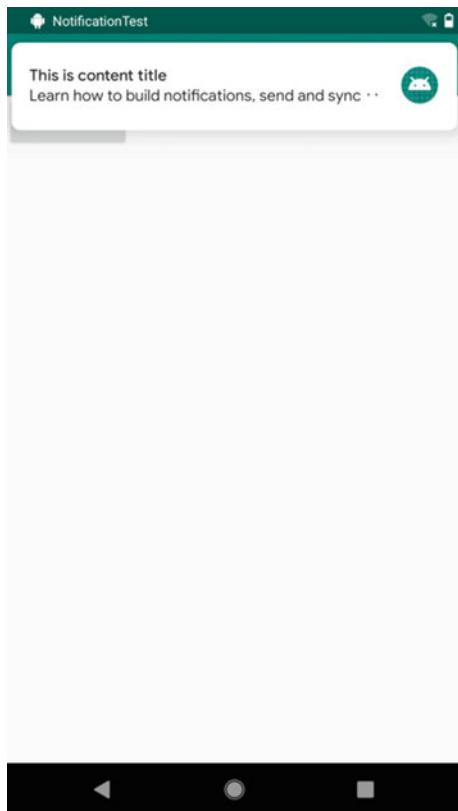
Here we set the importance level of the notification channel to “high”, indicating that this is a very important notification that users must see immediately. Run the app again and click Send notice, the result is as shown in Fig. 9.13.

As you can see, this notification will pop up a small window instead of a small icon at the system status bar, and it has more content detail to display which means that this is an important notification. Whether the user is playing game or watching movie, this notification will display at the front to attract the attention. Of course, you need to be careful about using this importance level and make sure that the notification is actually important, otherwise it may back fire.

## 9.3 Camera and Album

We often send photos when we use Messenger or Snapchat, and these photos can be from Camera or from Photos. A lot of apps will use camera or photos, and let us take a look at how to use them.

**Fig. 9.13** Send important notification



### 9.3.1 Take Photos with Camera

Let us first take a look at camera. A lot of apps allow you to take a selfie and upload in the app to be the profile picture and let's use an example to see how to use camera to take photos.

Create CameraAlbumTest project and update activity\_main.xml as code below:

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

    <Button
        android:id="@+id/takePhotoBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Take Photo" />
```

```
<ImageView  
    android:id="@+id/imageView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal" />  
  
</LinearLayout>
```

This layout file has two widgets: a Button and an ImageView. Button is used to take a photo by using the camera, and ImageView is used to display the photo.

Add the business logic to do all these in MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {  
  
    val takePhoto = 1  
    lateinit var imageUri: Uri  
    lateinit var outputImage: File  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        takePhotoBtn.setOnClickListener {  
            // Create File object to store the image  
            outputImage = File(externalCacheDir, "output_image.jpg")  
            if (outputImage.exists()) {  
                outputImage.delete()  
            }  
            outputImage.createNewFile()  
            imageUri = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {  
                FileProvider.getUriForFile(this, "com.example.  
cameraalbumtest.  
fileprovider", outputImage)  
            } else {  
                Uri.fromFile(outputImage)  
            }  
            // start Camera app  
            val intent = Intent("android.media.action.IMAGE_CAPTURE")  
            intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri)  
            startActivityForResult(intent, takePhoto)  
        }  
    }  
  
    override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
        super.onActivityResult(requestCode, resultCode, data)  
        when (requestCode) {  
            takePhoto -> {  
                if (resultCode == Activity.RESULT_OK) {  
                    // Display the photo  
                    val bitmap = BitmapFactory.decodeStream(contentResolver.  
openInputStream(imageUri))  
                }  
            }  
        }  
    }  
}
```

```
        imageView.setImageBitmap(rotateIfRequired(bitmap))
    }
}
}

private fun rotateIfRequired(bitmap: Bitmap): Bitmap {
    val exif = ExifInterface(outputImage.path)
    val orientation = exif.getAttributeInt(ExifInterface.
TAG_ORIENTATION,
        ExifInterface.ORIENTATION_NORMAL)
    return when (orientation) {
        ExifInterface.ORIENTATION_ROTATE_90 -> rotateBitmap(bitmap,
90)
        ExifInterface.ORIENTATION_ROTATE_180 -> rotateBitmap(bitmap,
180)
        ExifInterface.ORIENTATION_ROTATE_270 -> rotateBitmap(bitmap,
270)
        else -> bitmap
    }
}

private fun rotateBitmap(bitmap: Bitmap, degree: Int): Bitmap {
    val matrix = Matrix()
    matrix.postRotate(degree.toFloat())
    val rotatedBitmap = Bitmap.createBitmap(bitmap, 0, 0, bitmap.
width, bitmap.height,
        matrix, true)
    bitmap.recycle() // Recycle the bitmap object
    return rotatedBitmap
}
```

In MainActivity, the first thing is to register click event for Button, and then we can put the camera related logic in the click event handling block.

Here, we first create an object of File to store the photo taken by the camera and name the photo to output\_image.jpg and store it in SD card cache. The external cache directory is used to save temporary data which can be acquired by getExternalCacheDir(). The full path is /sdcard/Android/data/<package name>/cache. The reason to use external cache directory to save the image is because access to SD card needs runtime permission since Android 6.0, and this means that if file gets saved anywhere else in SD card, we need to ask for runtime permission and if we use external cache directory, we can skip this step. Also since Android 10.0, public SD card directories are not accessible by apps and each app can only access the private directory of its own (see Chap. 7) and SD card external cache directory.

Then, we will need to make a condition check to see if the current device's system is earlier than Android 7.0. If the condition is true, then call fromFile() of Uri to convert File object to Uri object, which is real local path of output\_image.jpg. Otherwise, call getUriFromFile() of FileProvider to convert the File object to an Uri

object that has been encapsulated. This method takes three params: the first param is an object of Context; the second param can be any unique string; and the third param is the newly created File object. The conversion is required because since Android 7.0, Uri that is using local real path is considered unsafe and will cause FileUriExposedException. FileProvider is a special ContentProvider that adopts similar data access mechanism to protect data and can selectively share the encapsulated Uri to other apps which improves app data safety.

Next, we can construct an Intent object and set the action of the object to android.media.action.IMAGE\_CAPTURE. Then, we use putExtra() of intent to specify the output path of the specified image and here we can just pass the Uri object. Then, we call startActivityForResult() to start Activity. Since it is an implicit Intent, system will find the Activity that can respond this intent to start, which is the Camera app. Then, the photo taken by the camera will be save to output\_image.jpg.

Since we use startActivityForResult() to start Activity, we will get a result in onActivityResult() after we take a photo. If photo is successfully taken then we can call decodeStream() of BitmapFactory to parse the output\_image.jpg to an object of Bitmap and set it to ImageView to display.

Notice that we add the logic to check the image direction and will rotate the image if we need to before displaying in the screen. This is because some phones make the assumption that photos will be taken in landscape mode by default and will rotate for 90 degrees when going to portrait mode.

As ContentProvider, we also need to register the FileProvider in AndroidManifest.xml as code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.cameraalbumtest">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="com.example.cameraalbumtest.
fileprovider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/file_paths" />
        </provider>
    </application>
</manifest>
```

The value of android:name property is fixed while the value of android:authorities has to be the same as the second param of FileProvider.getUriForFile(). The highlighted code also use <meta-data> tag within <provider> tag to specify the share path of Uri and references @xml/file\_paths resource which does not exist yet and let us create it now.

Right click res directory New Directory, and create xml directory, then right click xml directory → New → File to create file\_paths.xml file and update the content as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/
    android">
    <external-path name="my_images" path="/" />
</paths>
```

The external-path represents the files in the root of the external storage area; the name property can be any value; the value of path property sets the sharable path and “/” means share the whole SD card and you can also just share output\_image.jpg.

Now, run the app and click Take Photo button, and it will start Camera to take a photo as shown in Fig. 9.14. After photo is taken, click the button in the middle to go back to the screen of our app, and you should see the newly taken photo as shown in Fig. 9.15.

### 9.3.2 Select Images from Album

Sometimes, we do not need to take a new photo or just want to select from Photos app. A decent app should be able to allow user to make the decision to take a new photo or select from existing ones. Let us take a look at how to select photo from Photos or Gallery app.

Still modifying the CameraAlbumTest project, we edit the activity\_main.xml file and add a button to the layout for selecting images from the album, the code is shown below:

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

    <Button
        android:id="@+id/takePhotoBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Take Photo" />
```

**Fig. 9.14** Open camera to take photo



```
<Button  
    android:id="@+id/fromAlbumBtn"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="From Album" />  
  
<ImageView  
    android:id="@+id/imageView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal" />  
  
</LinearLayout>
```

Then, update MainActivity to add the logic to select photo from Gallery or Photos app as shown below:

```
class MainActivity : AppCompatActivity() {  
    ...  
    val fromAlbum = 2
```

**Fig. 9.15** Display photo in app



```
override fun onCreate(savedInstanceState: Bundle?) {  
    ...  
    fromAlbumBtn.setOnClickListener {  
        // Open document selector  
        val intent = Intent(Intent.ACTION_OPEN_DOCUMENT)  
        intent.addCategory(Intent.CATEGORY_OPENABLE)  
        // Only display images  
        intent.type = "image/*"  
        startActivityForResult(intent, fromAlbum)  
    }  
  
    override fun onActivityResult(requestCode: Int, resultCode:  
Int, data: Intent?) {  
        super.onActivityResult(requestCode, resultCode, data)  
        when (requestCode) {  
            ...  
            fromAlbum -> {  
                if (resultCode == Activity.RESULT_OK && data != null) {  
                    data.data?.let { uri ->  
                        // Display the selected images  
                }  
            }  
        }  
    }  
}
```

```
        val bitmap = getBitmapFromUri(uri)
        imageView.setImageBitmap(bitmap)
    }
}
}

private fun getBitmapFromUri(uri: Uri) = contentResolver
    .openFileDescriptor(uri, "r")?.use {
    BitmapFactory.decodeFileDescriptor(it.fileDescriptor)
}
...
}
```

In the From Album button click event, we construct an object of Intent and set its action to Intent.ACTION\_OPE\_DOCUMENT to open the system document Picker. Then, we set filter to the document category to only display image files. Then, we just need to call startActivityForResult(). Notice that the second argument we pass to startActivityForResult() is fromAlbum and this can make sure that when onActivityResult() is called, the branch under fromAlbum condition will be executed.

Next, we call getData() of Intent to get the Uri of the image and then call getBitmapFromUri() to convert the Uri to an object of Bitmap and display the image.

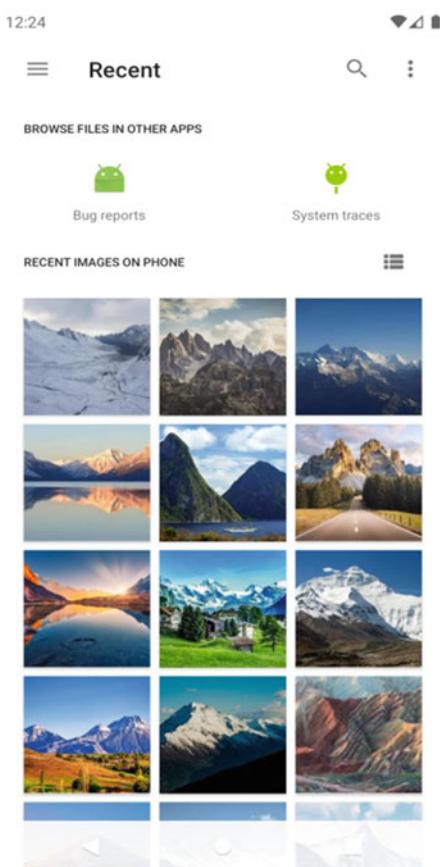
Run the app again and click From Album button to open the Picker to select images as shown in Fig. 9.16.

Select any of them and go back to our app, the selected image should be displayed as shown in Fig. 9.17.

Now, you learned how to use camera to take photos and select photos in the device. However, the implementation here is far from perfect. This is because if some images are high resolution images, then app can crash if we try to load them directly. A better approach would be to compress the image properly then load into memory. You can figure out how to compress images by searching and reading the related documents, and I will not discuss in this book.

## 9.4 Play Multi-Media Files

People use their phones to listen to music and watch movies a lot, and Android supports playing multi-media with comprehensive APIs such that developers can easily embed audio or video player in their app. Let us take a look at how to do it in this section.

**Fig. 9.16** Open file Picker

#### 9.4.1 Play Audio

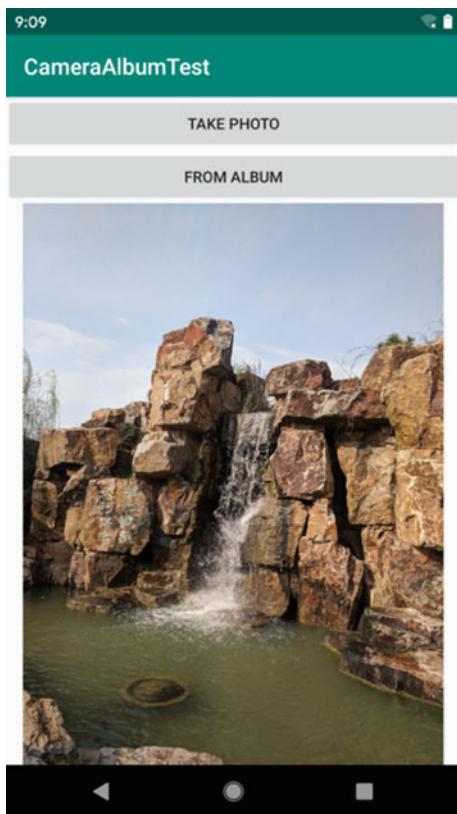
To play audio file in Android is used with MediaPlayer class which provides comprehensive methods to play with audio files of all formats and makes playing music really easy. Table 9.1 lists the commonly used methods in MediaPlayer.

Let us take a look at the process to use MediaPlayer. First, we need to create an object of MediaPlayer and use setDataSource() to set the source of the audio file and call prepare() to get MediaPlayer ready to play. Then, we can call start() to start playing audio, call pause() to pause playing and call reset() to stop playing.

Let us use an example to experiment these methods. Create PlayAudioTest project and update activity\_main.xml as code below:

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

**Fig. 9.17** Display selected image



**Table 9.1** Important Methods in MediaPlayer

| Method name     | Description  |
|-----------------|--|
| setDataSource() | Set the location of the media to be played         |
| prepare()       | Called before playing to do preparation work       |
| start()         | Start or continue playing                          |
| pause()         | Pause playing                                      |
| reset()         | Reset the MediaPlayer to newly created state       |
| seekTo()        | Play from the specified position                   |
| stop()          | Stop playing the resource                          |
| release()       | Release the related resource of MediaPlayer object |
| isPlaying()     | Determine if MediaPlayer is playing audio          |
| getDuration()   | Get the duration of the audio resource             |

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

    <Button
        android:id="@+id/play"
```

**Fig. 9.18** Put audio file in assets folder



```

        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Play" />

    <Button
        android:id="@+id/pause"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Pause" />

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

</LinearLayout>

```

The layout file has three buttons that can play, pause, and stop playing correspondingly.

MediaPlayer can be used to play the audio from network, device, and packaged in the app. For simplicity, let us take a look at how to play the audio packaged with the app.

Android Studio allows us to create assets directory to save file and sub-directories which will be packaged into the installer. Then, we can use the interface provided by AssetManager class to access the files under assets directory.

The assets directory has to be under app/src/main which is at the similar layer as java and res directories. Right click app/src/main New Directory, and type in assets in the pop up window to create the directory.

Since we need to play the audio file, I prepared music.mp3 file (check prelude for downloading link), and just put it in assets directory as shown in Fig. 9.18.

Then update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    private val mediaPlayer = MediaPlayer()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        initMediaPlayer()
        play.setOnClickListener {
            if (!mediaPlayer.isPlaying) {
                mediaPlayer.start() // start playing
            }
        }
        pause.setOnClickListener {
            if (mediaPlayer.isPlaying) {
                mediaPlayer.pause() // pause playing
            }
        }
        stop.setOnClickListener {
            if (mediaPlayer.isPlaying) {
                mediaPlayer.reset() // stop playing
                initMediaPlayer()
            }
        }
    }

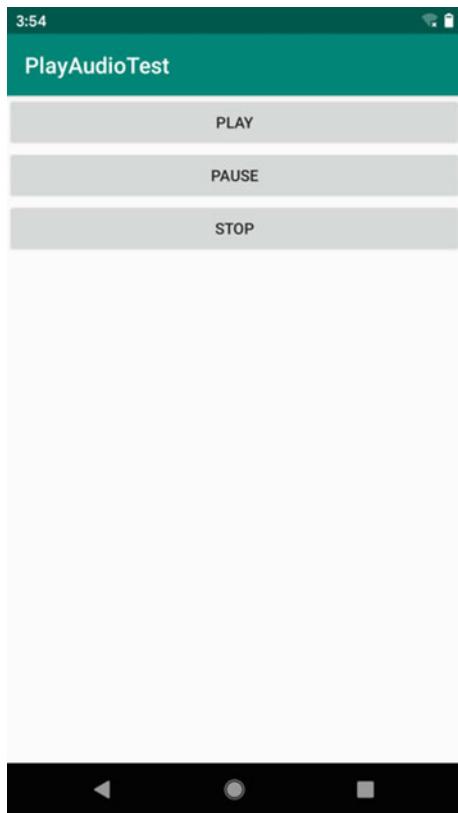
    private fun initMediaPlayer() {
        val assetManager = assets
        val fd = assetManager.openFd("music.mp3")
        mediaPlayer.setDataSource(fd.fileDescriptor, fd.startOffset, fd.length)
        mediaPlayer.prepare()
    }

    override fun onDestroy() {
        super.onDestroy()
        mediaPlayer.stop()
        mediaPlayer.release()
    }
}
```

When the class gets initialized, we create an instance of MediaPlayer and in onCreate(), we call initMediaPlayer() to initialize MediaPlayer object. In this method, we get an instance of AssetManager through getAssets() which can be used to read any assets in assets folder. Then we call openFd() to open it and then we call setDataSource() and prepare() to get ready for MediaPlayer to play.

Next, let us take a look at the click events of buttons. When clicking Play button, we need to determine if the MediaPlayer is playing audio or not, if not playing then use start() to play. When clicking Pause button, if MediaPlayer is playing audio then

**Fig. 9.19** Music player main screen



call pause() to pause playing. When the “Stop” button is clicked, if the current MediaPlayer is playing audio, the reset() method will be called to reset the MediaPlayer to the state just created, and then the initMediaPlayer() method will be called again.

Lastly in onDestory(), we need to call stop() and release() to release all the resources related to MediaPlayer.

That’s everything needed for an elementary audio player. Run the app and the main screen should be as shown in Fig. 9.19.

Click Play and you should hear the music, then click Pause, music should pause there. Click Play again will continue playing at the paused position. Click Stop, music will stop and if you click Play again, it should play from the start.

#### 9.4.2 Play Video

Playing video is not more complicated than playing audio, and usually it is done through VideoView class. This class can display video and control play, thus we can

**Table 9.2** Important methods in VideoView

| Method name    | Description                        |
|----------------|------------------------------------|
| setVideoPath() | Set the path to the video path     |
| start()        | Start or continue playing video    |
| pause()        | Pause playing video                |
| resume()       | Start playing from beginning       |
| seekTo()       | Play from specified position       |
| isPlaying()    | Determine if video is playing      |
| getDuration()  | Get the duration of the video file |

implement an elementary video player simply with this class. The use of VideoView is quite similar as MediaPlayer. Its commonly used methods are shown as in Table 9.2.

Let us use an example to learn these methods. Create PlayVideoTest project and update activity\_main.xml as code below:

```

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

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/play"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Play" />

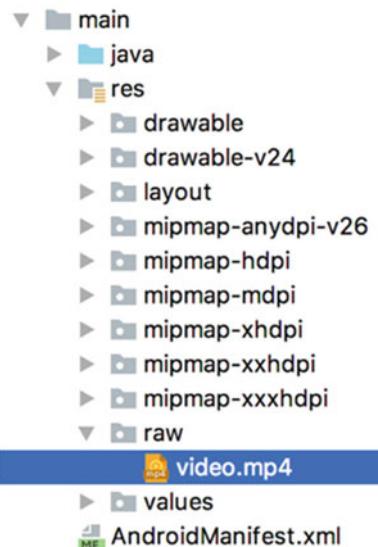
        <Button
            android:id="@+id/pause"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Pause" />

        <Button
            android:id="@+id/replay"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Replay" />

    </LinearLayout>

```

**Fig. 9.20** Put video resource in raw folder



```
<VideoView
    android:id="@+id/videoView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

</LinearLayout>
```

This layout file also has three buttons to control video: play, pause, and replay. There is a VideoView under the buttons to display the video.

Next, we need to store the video resource. Unfortunately, VideoView cannot play videos in assets folder. The res folder allow us to create a raw folder under it, and we can save the resource files like audio and video in this folder. VideoView can play the video resource in this folder.

Right click app/src/main/res New Directory, type raw in the pop up dialog to finish creating the raw folder and put the video resource in this folder. I prepared video.mp4 file which can be downloaded in the aforementioned download links. Of course, you can use your own videos. The structure should be as shown in Fig. 9.20.

Then, update MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val uri = Uri.parse("android.resource://$packageName/${R.raw.
video}")
        videoView.setVideoURI(uri)
        play.setOnClickListener {
```

```
        if (!videoView.isPlaying) {
            videoView.start() // Start play
        }
    }
    pause.setOnClickListener {
        if (videoView.isPlaying) {
            videoView.pause() // Pause play
        }
    }
    replay.setOnClickListener {
        if (videoView.isPlaying) {
            videoView.resume() // Replay
        }
    }
}

override fun onDestroy() {
    super.onDestroy()
    videoView.suspend()
}
```

The code above should be easy to understand now as it is quite similar with the previous audio player code. We first call Uri.parse() in onCreate() to get an object of Uri from video.mp4 file in raw folder. This is the required and standard way to do so. Then, we pass the uri to setVideoUri() of VideoView to initialize VideoView.

Next, let us take a look at the click events. When clicking Play button, if video is not playing then start() will be called to start playing. When clicking Pause button, if the video is playing then pause() will be called to pause playing. When clicking Replay, if the video is playing then call resume() to play from beginning.

Lastly, in onDestroy(), we can call suspend() to release all the resources from VideoView.

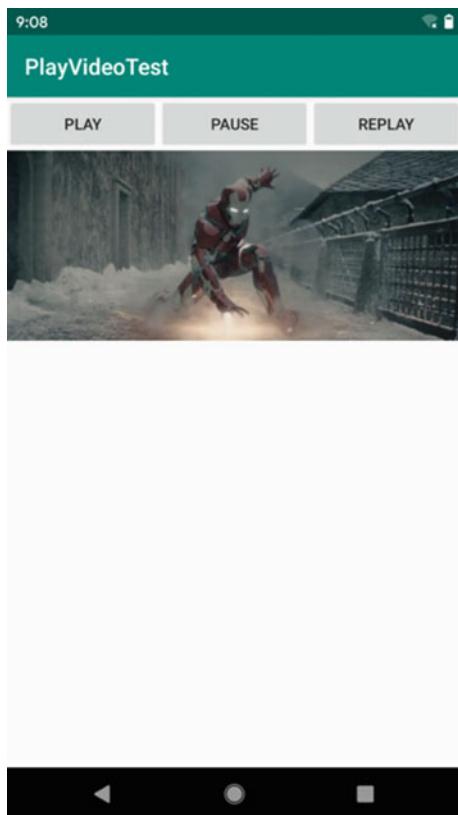
Run the app and click Play, you should see video start playing as shown in Fig. 9.21.

Click Pause to pause the playing video, and click Replay to play from start.

That's almost everything about VideoView. Why its use is so similar to MediaPlayer? This is because VideoView is just an encapsulation of MediaPlayer. Under the hood, it is MediaPlayer. Notice that VideoView has limited support to video formats and its performance also lags. Thus, do not use VideoView to implement video player that has format or performance requirements. Only use it to play some known and short videos.

That's everything I want to cover about multi-media. Time for Kotlin Class in this chapter.

**Fig. 9.21** Play video with VideoView



## 9.5 Kotlin Class: Use Infix to Improve Readability

In the previous chapters, we have used A to B syntax to construct key-value pair a few times including the Kotlin internal function `mapOf()` and the `cvOf()` function, we created in Chap. 7.

The advantage of this syntax is its readability. Compared with a function, this syntax looks more like using English. You may be wondering how this functionality is implemented, and whether “to” is a keyword in the Kotlin language. In this chapter’s Kotlin class, we will deep dive into these topics.

First of all, it is not a keyword in Kotlin. The reason that we can use syntax like A to B is because Kotlin provides an advanced syntax sugar: infix function. The infix function is not some complicated concept, it just adjusts the syntax of calling function. For instance, syntax A to B is functionally equal to `A.to(B)`.

Let us use two examples to see how to use infix starting with a simpler one.

String class has `startsWith()` function which you might have used. It is used to determine if a string starts with certain substring. For example, the condition check result of code below is true:

```
if ("Hello Kotlin".startsWith("Hello")) {  
    // business logic  
}
```

`startsWith()` is easy to use. But with infix function, we can rewrite the code to make it more readable. Create `infix.kt` with the code below:

```
infix fun String.startsWith(prefix: String) = startsWith(prefix)
```

Ignore the `infix` keyword for now, and you can tell that this is an extension function of `String` class. We add `startsWith()` functions to `Strings` which is used to determine if a string starts with some substring and its implementation is calling `startsWith()` of `String` class.

However, after adding the `infix` keyword, `startsWith()` function becomes an infix function. Besides the traditional way to call it, we can use a special syntax sugar format to call `startsWith()` function as shown below:

```
if ("Hello Kotlin" startsWith "Hello") {  
    // business logic  
}
```

From this example, we can see that infix's syntax is actually not complicated. The code above actually just calls the `startsWith()` function of `Hello Kotlin` string and pass in `Hello` string as param. However, infix function allows us to omit dot, braces, etc. and adopts a more natural language syntax to write code and make code more readable.

There are two requirements of infix function due to its special format of syntax sugar: first, infix cannot be a top-level function but has to be a member function of a class, and we can use extension function to define it to apply to a class; second, infix has to accept one and only one param which has no type limitation. You can think about why we need these two requirements and see if these requirements make sense or not.

Let us take a look at another more complicated example. Assume we have a set and if we need to determine if this set contains certain element, we can have code as shown below:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")  
if (list.contains("Banana")) {  
    // Business logic  
}
```

Simple to write, right? However, we can still make it more readable with infix function. Add the following code in `infix.kt`:

```
infix fun <T> Collection<T>.has(element: T) = contains(element)
```

We add an extension function to Collection interface. Since Collection is the interface of all the collection classes of Java and Kotlin, if we add has() to Collection, then all the sub classes of Set can use this function.

We also use the generics function from the last chapter to make has() take any type of param. The internal implementation of this function is easy, it is calling contains() of Collection interface. This means that has() has the same functionality as contains() and the only difference is that has() function has infix keyword in the definition and can be used with infix function syntax sugar.

Now, we can use the following code to determine if a Collection has certain element:

```
val list = listOf("Apple", "Banana", "Orange", "Pear", "Grape")
if (list has "Banana") {
    // Business logic
}
```

With the two examples, you should get better understanding of infix function. But you may still wonder how to implement A to B syntax. Let's take a look at the source code of to(), press Ctrl (command in Mac) and click the function to see its source code as shown below:

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

As you can see, here it uses generic function to define to() a function of A and takes a param of B type. Thus A and B can be two different types and can make the key value pair like String to integer.

The implementation of to() function simply create and return a Pair object. This means that A to B syntax actually will return a Pair object that contains data of type A and B, and mapOf() takes variable arguments of Pair type. Now, we just demystified the syntax of A to B.

We can also create our own syntax by mimicking to(). Add the following code in infix.kt:

```
infix fun <A, B> A.with(that: B): Pair<A, B> = Pair(this, that)
```

We simply replace to() to with() and keeps the implementation logic. Now, we can use with() to construct key value pairs in our project and pass the result to mapOf():

```
val map = mapOf("Apple" with 1, "Banana" with 2, "Orange" with 3, "Pear"
with 4,
"Grape" with 5)
```

Isn't it amazing? The infix function can help us create lots of interesting syntaxes and to make our code more readable.

That's it for this chapter's Kotlin Class, and next we will learn more about Git.

## 9.6 Git Time: Advanced Topics in Version Control

In the last Git Time section, we learned the basic use of Git which includes installation, create repository, and commit the local code changes. In this section, we will cover more techniques but we need to do some preparation first.

What we need to do for preparation is to create code repository for a project. Let us do it for PlayVideoTest project. Open terminal and go to the root directory of this project, then execute git init command as shown in Fig. 9.22.

That's what we need for preparation work.

### 9.6.1 Ignore Files

Now, we have created the repository, but before committing the code changes in PlayVideoTest project, we need to ask us what should be committed and what should not.

In Chap. 1, when we discuss about Android project structure, we know that the files in build folder are auto-generated during compilation, and we should not add these files in the version control, but how to do so?

Git allows user to specify the files and folders that they do not want to include in the version control. It will check .gitignore file in the repository folder, if this file exists, it will read the file line by line and exclude all the files and folders in each line from the version control. Notice that .gitignore can recognize wild card “\*”.

But we actually do not need to create .gitignore file by ourselves, Android Studio will create two .gitignore files when creating the project, one is in the root folder and another one is in the app folder. The one in the root folder is as shown in Fig. 9.23.

These files are just some configuration files generated by Android Studio, and most of the time, there is no need to add them in version control. Briefly go over the file, we can find that most of the lines are specific files and directories except \*.iml which means all files that end with \*.iml.

The .gitignore file in app module is simpler as shown in Fig. 9.24.

Since most of the files under app are the code we write, by default only the build folder will be ignored.

Of course, we can modify these two files as we want to meet our needs, for example, I don't want to keep track the tests in the app module because only I will use them, then I can modify app/.gitignore as below:

```
guolindeMacBook-Pro:~ guolin$ cd AndroidStudioProjects/AndroidFirstLine/PlayVideoTest/
guolindeMacBook-Pro:PlayVideoTest guolin$ git init
Initialized empty Git repository in /Users/guolin/AndroidStudioProjects/AndroidFirstLine/PlayVideoTest/.git/
guolindeMacBook-Pro:PlayVideoTest guolin$
```

Fig. 9.22 Create code repository

**Fig. 9.23** .gitignore file in root folder

```

1 *.iml
2 .gradle
3 /local.properties
4 /.idea/caches
5 /.idea/libraries
6 /.idea/modules.xml
7 /.idea/workspace.xml
8 /.idea/navEditor.xml
9 /.idea/assetWizardSettings.xml
10 .DS_Store
11 /build
12 /captures
13 .externalNativeBuild

```

**Fig. 9.24** .gitignore under app module

```

1 /build
2
3
4
5

```

```

/build
/src/test
/src/androidTest

```

We can do so by simply adding the highlighted two lines of text because all the test files are in these two folders. Now, we can commit the code changes, first we need to use add command to add the files as shown below:

```
git add .
```

Then, we can execute commit command to commit the code change as shown below:

```
git commit -m "First commit."
```

### 9.6.2 *Inspect Modified Content*

After the first commit, we might still need to maintain the project or add new features. The best practice is to commit the code whenever we finish certain piece of work. If a feature is complex and we may forget what we have changed before. But we do not need to worry, Git keeps all the record! Now let us take a look at how to check what has changed after last commit.

To see the changes is actually quite simple, and we just need to use status command. Type the following command in the root directory:

```
git status
```

```
[guolindeMacBook-Pro:PlayVideoTest guolin$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   app/src/main/java/com/example/playvideotest/MainActivity.kt

no changes added to commit (use "git add" and/or "git commit -a")
guolindeMacBook-Pro:PlayVideoTest guolin$ ]
```

**Fig. 9.25** Check file changes

Then Git will show that there is no changes to commit, since we just committed the code changes. Now update MainActivity in PlayVideoTest project as shown below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        play.setOnClickListener {
            ...
            Log.d("MainActivity", "video is playing")
        }
        ...
    }
}
```

Here, we add the log logic. Then execute git status command again, the result should be as shown in Fig. 9.25.

Git reminds us that MainActivity.java file has been changed, but how to see what are the changes? Then, we need diff command, as shown below:

```
git diff
```

This will display all the changes from all the modified files, if you just want to check the changes in MainActivity.java, you can add the full path of this file:

```
git diff app/src/main/java/com/example/playvideotest/MainActivity.
kt
```

The result is as shown in Fig. 9.26.

We can see all the changes by using the command. The plus sign on the left means added content. If some contents are deleted then there will be a minus sign on the left side.

```
guolindeMacBook-Pro:PlayVideoTest guolin$ git diff app/src/main/java/com/example/playvideotest/MainActivity.kt
diff --git a/app/src/main/java/com/example/playvideotest/MainActivity.kt b/app/src/main/java/com/example/playvideotest/MainActivity.kt
index 322a3d9..b3ce399 100644
--- a/app/src/main/java/com/example/playvideotest/MainActivity.kt
+++ b/app/src/main/java/com/example/playvideotest/MainActivity.kt
@@ -3,6 +3,7 @@ package com.example.playvideotest
import android.net.Uri
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
+import android.util.Log
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
@@ -16,6 +17,7 @@ class MainActivity : AppCompatActivity() {
    if (videoView.isPlaying) {
        videoView.start() // 开始播放
    }
    Log.d("MainActivity", "video is playing")
}
pause.setOnClickListener {
    if (videoView.isPlaying) {
guolindeMacBook-Pro:PlayVideoTest guolin$
```

**Fig. 9.26** Check the changes in files

```
guolindeMacBook-Pro:PlayVideoTest guolin$ git status
On branch master
nothing to commit, working tree clean
guolindeMacBook-Pro:PlayVideoTest guolin$
```

**Fig. 9.27** Verify file change reverted

### 9.6.3 Revert the Uncommitted Changes

Sometimes the new code changes are not what we want, and we decide to revert the changes before the code is committed, this can be easily done.

For example, we added one line to log in MainActivity, if we want to revert the change, then we can use checkout command as shown below:

```
git checkout app/src/main/java/com/example/playvideotest/
MainActivity.kt
```

After running this, all the uncommitted changes in MainActivity.java should be reverted. Run git status command to verify and the result is as shown in Fig. 9.27.

As you can see, there is no file change to commit and this means that revert is successful.

However, this command can only revert the files that hasn't been added and if the file has already been added, we cannot revert the changes by the command above. Let us try it.

First add log code in MainActivity and execute the following command:

```
git add .
```

```
[guolin@MacBook-Pro:PlayVideoTest guolin$ git status  
On branch master  
Changes to be committed:  
(use "git reset HEAD <file>..." to unstage)  
  
        modified:   app/src/main/java/com/example/playvideotest/MainActivity.kt  
guolin@MacBook-Pro:PlayVideoTest guolin$ ]
```

Fig. 9.28 Check file changes again

This will add all the file changes, and we can use git status to validate, the result is as shown in Fig. 9.28.

After running the checkout command again, you will find that MainActivity is still in added state and changes are not reverted.

What should we do under this circumstance? To revert the added file, we should undo add then revert the change. To undo add, we can use reset as shown below:

```
git reset HEAD app/src/main/java/com/example/playvideotest/  
MainActivity.kt
```

Then, run git status again, you will find that MainActivity.java is back to the state of not added yet, then we can use checkout to revert the changes.

#### 9.6.4 Check Commit History

After a few months of development on PayVideoTest, we probably will have hundreds of commits, you may probably forget what the changes were in each commit. It does not matter, Git will remember for us. We can use log command to check the history commits as shown below:

```
git log
```

Since we only committed once, there is not much to display, as shown in Fig. 9.29.

All the commit history records will have commit id, author, date, and description. If we add log again in MainActivity and commit the change as shown below:

```
git add .  
git commit -m "Add log."
```

Now run git log command, the result should be as shown in Fig. 9.30.

If there are too many commits and if we only want to see one of the record, then we can do so by specifying the id of this commit:

```
[guolin@MacBook-Pro:PlayVideoTest guolin$ git log  
commit ddfd7ea16002e30597ab62a575d6464826f0dd6f (HEAD -> master)  
Author: Tony <tony@gmail.com>  
Date:   Wed Aug 14 19:30:06 2019 +0800  
  
        First commit.  
guolin@MacBook-Pro:PlayVideoTest guolin$ ]
```

Fig. 9.29 Check commit history

```
[guolin@MacBook-Pro:PlayVideoTest guolin$ git log  
commit 2960da5042b2dbf1abbb3691be1b18a6f446b844 (HEAD -> master)  
Author: Tony <tony@gmail.com>  
Date:   Wed Aug 14 20:06:00 2019 +0800  
  
        Add log.  
  
commit ddfd7ea16002e30597ab62a575d6464826f0dd6f  
Author: Tony <tony@gmail.com>  
Date:   Wed Aug 14 19:30:06 2019 +0800  
  
        First commit.  
guolin@MacBook-Pro:PlayVideoTest guolin$ ]
```

Fig. 9.30 Check commit history

```
git log 2960da5042b2dbf1abbb3691be1b18a6f446b844
```

We can also check the recent n commits with param, for instance, -1 means we only want see the last commit:

```
git log -1
```

That's everything for this chapter's Git Time. Let's summarize what we learned in this chapter now.

## 9.7 Summary and Comment

In this chapter, we mainly discussed various topics of multi-media in Android, including using notification, taking photos, selecting photos, and playing audio and video file. Since the emulator is not enough to test all these, I also covered how to debug in Android phones.

In this chapter's Kotlin Class section, we learned how to use infix function and demystified mapOf() function and A to B syntax.

In this chapter's Git Time section, we covered some more advanced techniques of using Git, including ignoring file, check changes, check commits history, and so on. Now you should be able to use Git in real project but we will cover more about Git in later chapters.

Now, there is only one component left among the four main components of Android which is Service. Let us discuss this in next chapter.

# Chapter 10

## Work on the Background Service



Back to the college days, iPhone was a luxury device and only few could afford, Android was not even born yet, and Nokia dominated the world market of mobile phones. At that time, I used to enjoy Nokia's Symbian OS because it supported background task which a lot of other phones did not support. I could make phone calls or listen to music keeping QQ live on the background, how cool was it! I also used to naively think that smart phones are those that support background tasks.

Nowadays, Symbian has lost its position for a long time, Android and iOS almost take up all the smart phone market share. Between these two operating systems, iOS used to have no support to background tasks and then realized the importance of this functionality and gradually added support for it. On the contrary, Android used to support very rich background functionalities, then realized the side effects of openness and gradually put restriction on the background functionalities. Service is used to implement background related functions and is one of the four components of Android. In this chapter, we will take a look at this important component.

### 10.1 What Is Service?

Service is a solution provided by Android for background tasks. It is a good choice for the tasks that need to run for a long time while do not interact with user. Service does not depend on user interface to run and even when the app is put on background or user opens another app, Service can keep running uninterrupted.

It is worth noting that, Service does not run in an independent process but depends on the process of the app that creates the Service. When an app's process has been killed, all the Services that depends on this app's process will stop running.

Also, do not get confused by the word background. Service does not start a new thread by default, and all the code in it runs in main thread by default. This means that we need to manually create worker thread to run the business logic, otherwise,

main thread may get blocked. Thus, in the first section of this chapter, let us take a look at Android multithreading.

## 10.2 Android Multithreading

If you are familiar with Java, you should not find multithreading a strange concept. When we need to run some time consuming operations, for example, a network request may take some time because of slow internet or server cannot respond instantly. If we do not run these operations in worker thread, then main thread will be blocked which leads to poor user experience. Let us start with the basic use of thread.

### 10.2.1 Basic Use of Thread

Android multithreading is very similar to Java multithreading and uses similar syntax. For instance, to define a new thread, we only need to create a new class that extends Thread and override run() of the super class, then put the time consuming logic in this method, as shown in code below:

```
class MyThread : Thread() {  
    override fun run() {  
        //time consuming logic  
    }  
}
```

To start this thread, create an instance of MyThread and call start() to run the code of run() in worker thread as shown in code below:

```
MyThread().start()
```

Obviously, extending Thread is tightly coupled with Thread, and usually, we implement Runnable interface to define a thread as code below:

```
class MyThread : Runnable {  
    override fun run() {  
        // business logic  
    }  
}
```

Then, we need to adjust the way to start the thread as shown below:

```
val myThread = MyThread()  
Thread(myThread).start()
```

The constructor of Thread takes an argument of Runnable type and the instance of MyThread implements Runnable interface, thus we can pass it to Thread constructor. Then, we can call start() of Thread to execute the code in run() in worker thread.

If you do not want to define a new class to implement Runnable interface, you can also use Lambda expression which is more common, as shown in code below:

```
Thread {  
    // business logic  
}.start()
```

You should feel familiar to the methods mentioned above as they are also used in Java. Kotlin also provides an easier way to start a new thread, as shown below:

```
thread {  
    // business logic  
}
```

The thread here is a Kotlin top-level function, and we just need to implement the business logic in the Lambda expression without calling start().

After reviewing similarities, let us take a look at the difference.

### 10.2.2 Update UI in Worker Thread

As many other GUI libraries, Android UI is not thread safe. This means that updating the UI element has to happen in main thread, otherwise there will be exception.

Let us use an example to verify this. Create AndroidThreadTest project, and modify activity\_main.xml as shown below:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/  
res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
<Button  
    android:id="@+id/changeTextBtn"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Change Text" />  
  
<TextView  
    android:id="@+id/textView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_centerInParent="true"  
    android:text="Hello world"  
    android:textSize="20sp" />
```

```
android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original
thread that created a view hierarchy can touch its views.
```

**Fig. 10.1** Stack trace of crash

```
</RelativeLayout>
```

The layout file defines two widgets, TextView will display “Hello World” at the center of the screen. Button will change the content of TextView and we expect that click Button will change the text in TextView to “Nice to meet you.”

Next, update MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        changeTextBtn.setOnClickListener {
            thread {
                textView.text = "Nice to meet you"
            }
        }
    }
}
```

In the button click event code, we start a new worker thread, and in the worker thread, we call setText() of TextView to update the text to “Nice to meet you.” The code here simply updates UI in worker thread. Run the app and click “Change Text” button, and you shall find that app crashes. Observe the error log in logcat, you shall find that this is caused by updating UI in worker thread as shown in Fig. 10.1.

This verifies that Android indeed does not allow UI updating in worker thread. But sometimes, we need to run some time consuming tasks in worker thread and will use the result of the task to update the UI widgets. What shall we do?

Android provides the mechanism of asynchronous message handling to solve the problem perfectly. We will discuss this in next section.

Update MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {

    val updateText = 1

    val handler = object : Handler() {
        override fun handleMessage(msg: Message) {
            // Updating UI here is fine
            when (msg.what) {
                updateText -> textView.text = "Nice to meet you"
            }
        }
    }
}
```

```
        }
```

```
    }
```

```
}
```

```
override fun onCreate(savedInstanceState: Bundle?) {
```

```
    super.onCreate(savedInstanceState)
```

```
    setContentView(R.layout.activity_main)
```

```
    changeTextBtn.setOnClickListener {
```

```
        thread {
```

```
            val msg = Message()
```

```
            msg.what = updateText
```

```
            handler.sendMessage(msg) // send the Message object
```

```
        }
```

```
    }
```

```
}
```

We first define an integer variable `updateText` to represent the action of updating `TextView`. Then, we create an object of `Handler` and override the super class's `handleMessage()`, and handle the `Message` in this method. If the value of `what` field in `Message` is equal to `updateText`, then update the content of `TextView` to "Nice to meet you."

Next, let us take a look at the button click event code. Here, we do not update the UI in the worker thread but instead create an object of `Message(android.os.Message)`, and set the value of `what` field to `updateText`, then call `sendMessage()` of `Handler` to send this `Message`. Then, `Handler` will receive this `Message` and handle it in `handleMessage()`. Notice that `handleMessage()` is running in main thread, thus we can update UI here without causing exception. Then if `what` field's value is equal to `updateText`, we update the content of `TextView` to "Nice to meet you."

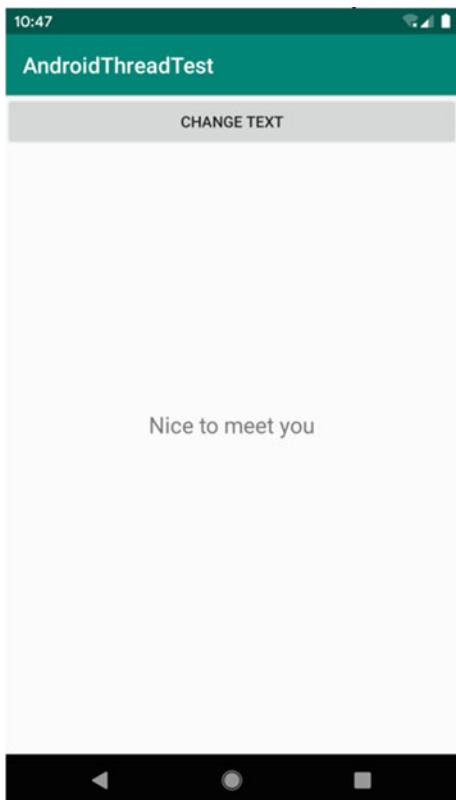
Run the app again and you should see "Hello world" at the center of screen. Click "Change Text" button and the content will be updated to "Nice to meet you," as shown in Fig. 10.2.

This is the elementary use of Android asynchronous message handling mechanism and with this mechanism we can easily solve the problem of updating UI in worker thread. Now, let us take a look at how it works under the hood.

### 10.2.3 Async Message Handling Mechanism

There are four main components involving in processing the asynchronous message: `Message`, `Handler`, `MessageQueue`, and `Looper`. We have seen `Message` and `Handler` in previous section while `MessageQueue` and `Looper` are new concept. Let me briefly cover these four components.

**Fig. 10.2** Update text success



#### 1. Message

Message is used to communicate between threads, which can carry small amount of data to share between threads. In the last section, we used what field; besides this, we can also use arg1 and arg2 to carry some integer values and use obj field to carry Object instance.

#### 2. Handler

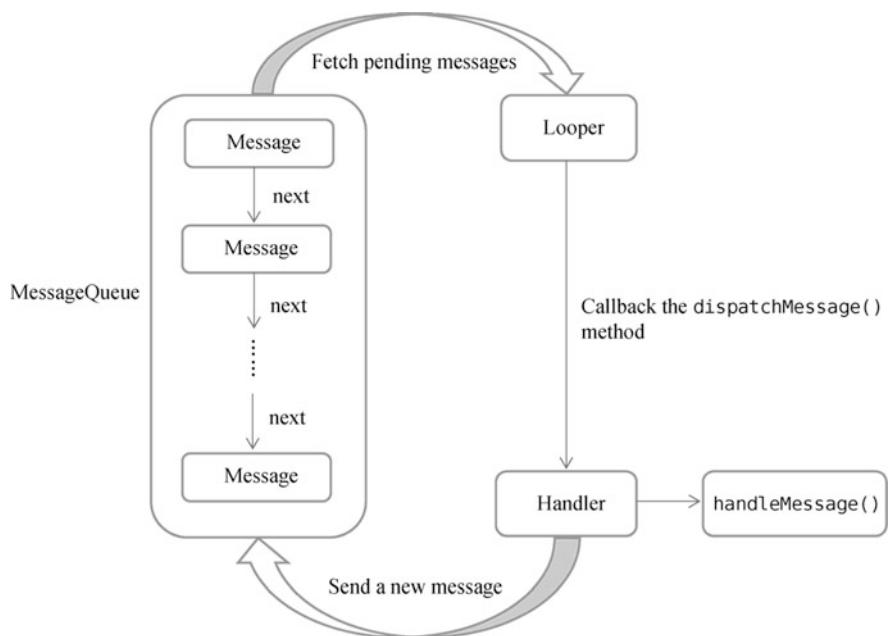
Handler is used to send and process Message. Handler's sendMessage(), post() can be used to send message and the message will eventually reach handleMessage() of Handler.

#### 3. MessageQueue

MessageQueue is used to store all the messages sent through Handler. These messages will stay in the message queue, waiting to be processed and each thread will only have one instance of MessageQueue.

#### 4. Looper

Looper is the manager of MessageQueue in each thread; after loop() of Looper gets called, thread will process the message until loop stops. Whenever there is a



**Fig. 10.3** Asynchronous message handling process diagram

message in MessageQueue, message will be handed to handleMessage() of Handler to process. Each thread will only have one instance of Looper.

After introduction to the concept of Message, Handler, MessageQueue, and Looper, let us go over the process of asynchronous message handling. First, we need to create an instance of Handler in main thread and override its handleMessage(). Then, when there is need to update UI, we create an instance of Message and send this message through Handler. Then message will be added to MessageQueue and wait to be processed, and Looper will try to get message from MessageQueue and send to handleMessage() of Handler. Since Handler is created in main thread, handleMessage() will also run in main thread. Thus, we can update UI in handleMessage(). The whole flow is shown in Fig. 10.3.

Basically the process will place the message from worker thread to main thread, and make it possible to update UI.

#### 10.2.4 Use AsyncTask

To make it easier to update UI in worker thread, Android also provides some other useful tools such as AsyncTask. With AsyncTask, even without knowledge of asynchronous message handling mechanism, it is still easy to switch from worker

thread to main thread. Of course, under the hood, `AsyncTask`'s implementation is also based on asynchronous message handling mechanism. Android just does a good job to encapsulate the implementation details.

`AsyncTask` is an abstract class, thus we need to create a sub-class to inherit it. When inheriting `AsyncTask`, we can specify three generic params as shown below:

- Params. The params needed during running `AsyncTask` and can be used in background tasks.
- Progress. The progress indicator shown in the screen when background task is running, this is a generic type param.
- Result. It is used after the task is done, if result needs to be returned, also a generic type.

Thus, the simplest way to define an `AsyncTask` can have the form as below:

```
class DownloadTask : AsyncTask<Unit, Int, Boolean>() {  
    ...  
}
```

The above code sets the first generic param to `Unit`, which means that there is no param passed to background task. The second param type is `Int`, which means that integer value is used to indicate the progress. The third generic param is `Boolean`, which means that the return value is of `Boolean` type.

The `DownloadTask` has no business logic yet, and we need to override some methods in `AsyncTask` to make it runnable. Usually we need to override the following four methods.

#### 1. `onPreExecute()`

This method is called before execution of background task to initialize UI for example, displaying a progress bar.

#### 2. `doInBackground(Params...)`

This method will run in worker thread, and we need to run the time consuming task here. After finishing, use the return statement to return the result and if the third generic param of `AsyncTask` is `Unit`, then no need to return the result. Notice that this method cannot update UI. If there is need to update UI components such as the progress of the current task, then we can call `publishProgress(Progress...)` to do it.

#### 3. `onProgressUpdate(Progress...)`

After the background task calls `publishProgress(Progress...)`, `onProgressUpdate(Progress...)` will be called and the params are passed from the background task. In this method, we can update the UI with the params.

#### 4. `onPostExecute(Result)`

After background task finishes and return, this method will be called. The result of the task will be passed to this method, and we can use the result to update

UI, for instance, displaying the result of the task, closing the progress bar, and so on.

Thus, a more complete form of AsyncTask can have form as shown below:

```
class DownloadTask : AsyncTask<Unit, Int, Boolean>() {

    override fun onPreExecute() {
        progressDialog.show() // display progress dialog
    }

    override fun doInBackground(vararg params: Unit?) = try {
        while (true) {
            val downloadPercent = doDownload() // example business logic
            method
            publishProgress(downloadPercent)
            if (downloadPercent >= 100) {
                break
            }
        }
        true
    } catch (e: Exception) {
        false
    }

    override fun onProgressUpdate(vararg values: Int?) {
        // update the download progress
        progressDialog.setMessage("Downloaded ${values[0]}%")
    }

    override fun onPostExecute(result: Boolean) {
        progressDialog.dismiss()// close the progress dialog
        // display the download result
        if (result) {
            Toast.makeText(context, "Download succeeded", Toast.
LENGTH_SHORT).show()
        } else {
            Toast.makeText(context, "Download failed", Toast.
LENGTH_SHORT).show()
        }
    }
}
```

In this DownloadTask, we execute the downloading code in doInBackground() which runs in worker thread and does not block main thread. Notice that doDownload() does not really do anything here, let us assume it can calculate the download progress and return the result. After acquiring the download progress, we can display the result, but doInBackground() runs in worker thread, no UI update can

happen here, thus we can call publishProgress() to publish the download progress and then onProgressUpdate() will be called for updating the UI.

After downloading, doInBackground() will return a Boolean result, and onPostExecute() will be called. This method also runs in main thread, and we can show the Toast message based on the download result and finish the DownloadTask.

Fundamentally, to use AsyncTask, we put the time consuming code in doInBackground(), update UI in onProgressUpdate() and wrap up in onPostExecute().

To start this task, we only need the following code:

```
DownloadTask().execute()
```

Of course, you can pass any number of params to execute(), which will be passed to doInBackground().

The AsyncTask is easier to use as we do not need to think about asynchronous message handling mechanism, no need to create a Handler to send and receive message. All we need is to publishProgress() to switch from worker thread to main thread.

## 10.3 Basic Use of Service

Let us get to the main topic of this chapter—Service. And let us start with the basic use of Service.

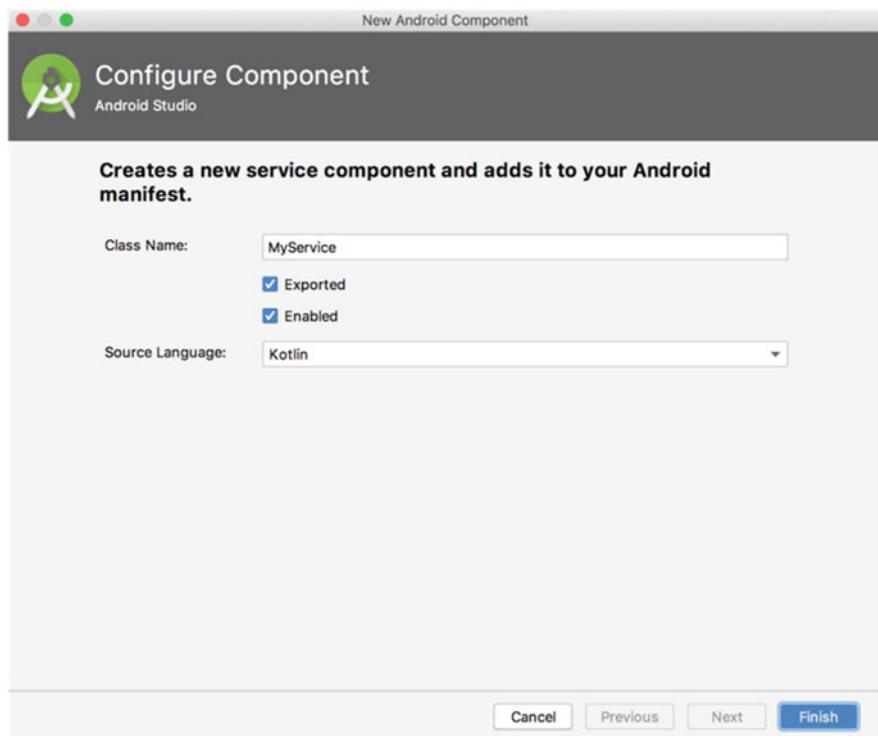
### 10.3.1 Define a Service

Let us create a new project to test Service. Create ServiceTest project and right click com.example.servicetest → New → Service → Service, a window will show up as shown in Fig. 10.4.

Here, we set the class name to MyService. Exported property determines if this service can be accessible to other apps. Enabled property determines if this service is enabled or not. Check these two properties and click Finish.

MyService should have the following code:

```
class MyService : Service() {  
  
    override fun onBind(intent: Intent): IBinder {  
        TODO("Return the communication channel to the service.")  
    }  
}
```



**Fig. 10.4** Window to create Service

You can tell that MyService inherits system Service class. This class is empty besides onBind()

which is the only abstract method in Service, thus it has to be overridden in sub class. We will cover this method later and ignore it for now.

To run the time consuming task, we need to override some other methods in Service, as shown below:

```
class MyService : Service() {  
    ...  
    override fun onCreate() {  
        super.onCreate()  
    }  
  
    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {  
        return super.onStartCommand(intent, flags, startId)  
    }  
}
```

```

override fun onDestroy() {
    super.onDestroy()
}

}

```

Here, we override `onCreate()`, `onStartCommand()`, and `onDestroy()`, which are three most commonly used methods in Service. `onCreate()` will be called when Service gets created, `onStartCommand()` will be called when Service gets started, and `onDestroy()` will be called when the Service gets destroyed.

Normally, if we want to execute certain code right after Service gets started, then we can put the code in `onStartCommand()`. And when Service gets destroyed, we should recycle the resources that will not be used in `onDestroy()`.

It is worth noting that Service needs to get registered in `AndroidManifest.xml` to be able to get enabled. And you should notice that this is a common requirement for all the four components, and you probably can also tell the Android Studio already did this for us. Open `AndroidManifest.xml`, the code should be as shown below:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.servicetest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <service
            android:name=".MyService"
            android:enabled="true"
            android:exported="true">
        </service>
    </application>

</manifest>

```

This is everything we need to define a Service.

### 10.3.2 Start and Stop Service

After Service definition is done, we need to know how to start and stop this Service. We need Intent to do them. Next, let us experiment them in `ServiceTest`.

First update `activity_main.xml` as code below:

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

    <Button
        android:id="@+id/startServiceBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start Service" />

    <Button
        android:id="@+id/stopServiceBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Stop Service" />

</LinearLayout>
```

Here, we add two buttons to start and stop Service correspondingly.  
Then update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        startServiceBtn.setOnClickListener {
            val intent = Intent(this, MyService::class.java)
            startService(intent) // start Service
        }
        stopServiceBtn.setOnClickListener {
            val intent = Intent(this, MyService::class.java)
            stopService(intent) // stop Service
        }
    }
}
```

In the click event code of “Start Service” button. We construct an object of Intent and call startService() to start MyService. In the click event code of “Stop Service,” we also construct an object of Intent and call stopService() to stop MyService. startService() and stopService() are both defined in Context class, thus we can call them directly in Activity. Also Service can stop itself by calling its method stopSelf().

Next question is how do we verify if the Service has started successfully or stopped? The simplest way is to add logging in the methods of MyServices, as shown in code below:

```

class MyService : Service() {
    ...
    override fun onCreate() {
        super.onCreate()
        Log.d("MyService", "onCreate executed")
    }

    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        Log.d("MyService", "onStartCommand executed")
        return super.onStartCommand(intent, flags, startId)
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d("MyService", "onDestroy executed")
    }
}

```

Run the app and the main screen should look like Fig. 10.5.

Click “Start Service” button and the logs in Logcat should be as shown in Fig. 10.6.

From the logs, we can see that `onCreate()` and `onStartCommand()` have been called, which means that Service has been started successfully. You can also find it by `Settings → System → Advanced → Developer options → Running services` (the path may vary depends on the device and may not even exist), as shown in Fig. 10.7.

Click “Stop Service” button and Logcat window should show log as shown in Fig. 10.8.

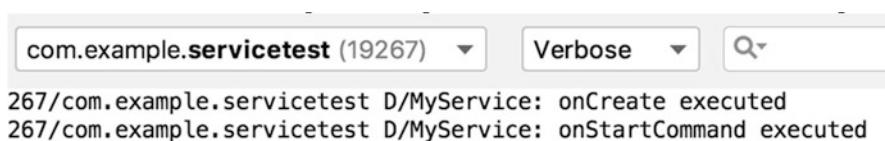
The log in Logcat proves that `MyService` indeed stopped.

The above are basic uses of starting and stopping Service. However, since Android 8.0, the background capability has been greatly weakened. Now only when the app is visible can Service run without issue, once the app is on background, Service can be recycled any time by system. Android introduced this change to prevent malicious apps from taking resources in the background for a long time and slowing down the phone. Of course, if you indeed need to run some time consuming task on the background, you can use foreground Service or WorkManager. We will discuss foreground Service shortly and will cover WorkManager in Chap. 13.

Back to the main topic. So you might wonder what is the difference between `onCreate()` and `onStartCommand()`? Since, after clicking “Start Service” button, both methods got executed.

So the difference is that `onCreate()` will be called when Service gets created for the first time while `onStartCommand()` will be called every time Service gets started. Since, it is our first time to click “Start Service” button, Service will be created and both methods will be called, then if you click “Start Service” button again, you shall find that only `onStartCommand()` will be called.

**Fig. 10.5** Main screen of ServiceTest



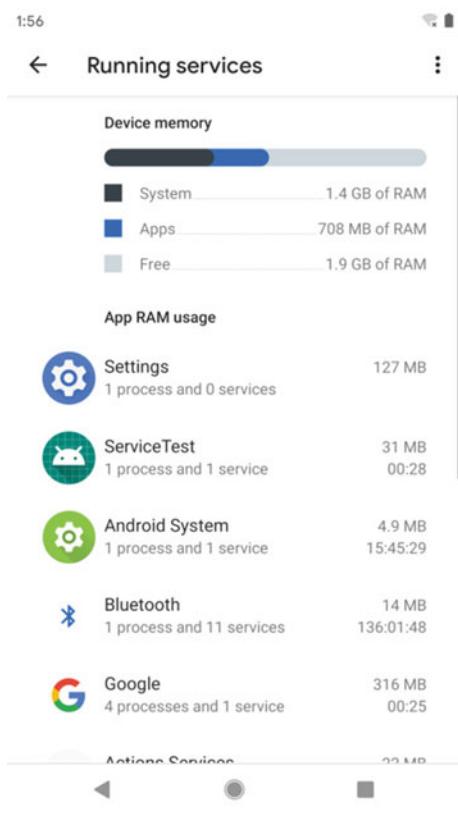
**Fig. 10.6** Start service logs

### 10.3.3 Communication Between Activity and Service

In last section, we discussed start and stop Service. You might notice that after Service gets started in Activity, there is no connection between them. Indeed, Activity cannot control what's running in Service. The situation is like Activity notify Service to run and then Service will keep running. But Activity does not know what Service is doing and the status of the Service.

What if we need to control the business logic running in Service by Activity? We need help from the method we ignored previously which is onBind().

**Fig. 10.7** Running Services list



Actions: Services | 02:30

com.example.servicetest (19267) ▾ Verbose ▾ 🔍

267/com.example.servicetest D/MyService: onDestroy executed

**Fig. 10.8** Stop Service log

For instance, assume we need to implement downloading in MyService and in the Activity we need to control when to start downloading, check the progress. The solution is to create an object of Binder to manage downloading. Update MyService as code below:

```
class MyService : Service() {

    private val mBinder = DownloadBinder()

    class DownloadBinder : Binder() {
```

```
fun startDownload() {
    Log.d("MyService", "startDownload executed")
}

fun getProgress(): Int {
    Log.d("MyService", "getProgress executed")
    return 0
}

}

override fun onBind(intent: Intent): IBinder {
    return mBinder
}
...
}
```

We create DownloadBinder class that inherits Binder and inside this class we create methods to start downloading and check the progress. Of course these two methods are just for demonstration purpose and just log some information for us to verify.

Then, we create a DownloadBinder object in MyService and return this instance in onBind(). That is all we need for MyService.

Next, let us take a look at how to call the methods in Activity. First add two buttons in layout file by modifying activity\_main.xml as shown below:

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

    ...

<Button
    android:id="@+id/bindServiceBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Bind Service" />

<Button
    android:id="@+id/unbindServiceBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Unbind Service" />

</LinearLayout>
```

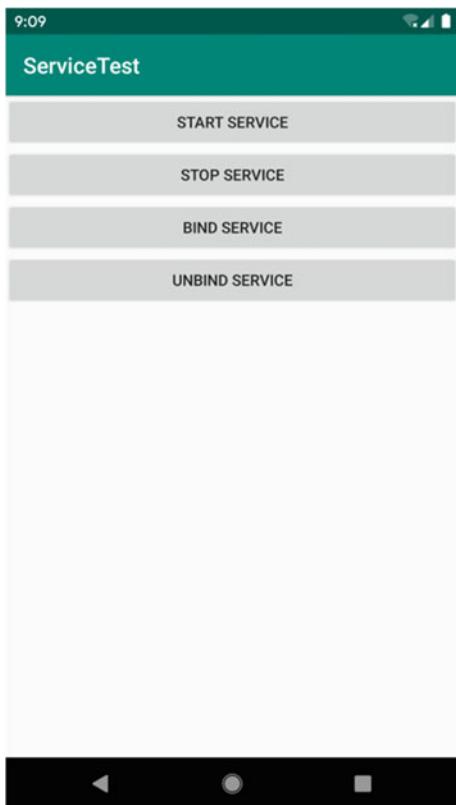
These two buttons are used to bind and unbind Service with Activity. Once Activity binds with Service, it can call the method provided by Binder of the Service. Update MainActivity as code below:

```
class MainActivity : AppCompatActivity() {  
  
    lateinit var downloadBinder: MyService.DownloadBinder  
  
    private val connection = object : ServiceConnection {  
  
        override fun onServiceConnected(name: ComponentName, service: IBinder) {  
            downloadBinder = service as MyService.DownloadBinder  
            downloadBinder.startDownload()  
            downloadBinder.getProgress()  
        }  
  
        override fun onServiceDisconnected(name: ComponentName) {}  
    }  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        ...  
        bindServiceBtn.setOnClickListener {  
            val intent = Intent(this, MyService::class.java)  
            bindService(intent, connection, Context.BIND_AUTO_CREATE) //  
            bind Service  
        }  
        unbindServiceBtn.setOnClickListener {  
            unbindService(connection) // unbind Service  
        }  
    }  
}
```

In the code above, we first create an instance of anonymous class of ServiceConnection and override onServiceConnected() and onServiceDisconnected(). onServiceConnected() will be called when Activity binds with Service successfully, while onServiceDisconnected() will be called when the process that creates Service crashes or gets killed, which is not used very often. In onServiceConnected(), we get an instance of DownloadBinder by casting. With this instance, we can connect Activity and Service as we can now call any public method inside DownBinder from Activity or, in other words, Activity can decide what Service will do. Here is just a simple test, we call startDownload() and getProgress() of DownloadBinder in onServiceConnected().

Of course, for now Activity and Service are not bound together yet as this will be done in the click event code of “Bind Service” button. Here, we also construct an

**Fig. 10.9** New main screen of ServiceTest



object of Intent and call bindService() to bind MainActivity with MyService. bindService() takes three params, the first param is the Intent object we just constructed, the second is instance of ServiceConnection and the third param is a flag and BIND\_AUTO\_CREATE means that Service will be created automatically after Activity is bind to Service. This will trigger onCreate() of MyService but onStartCommand() will not be triggered.

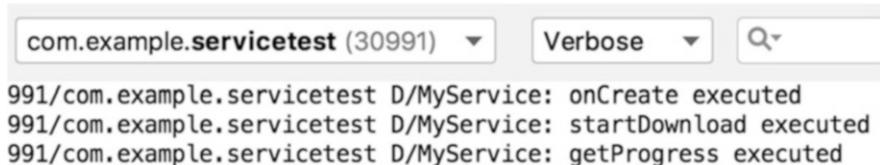
If we want to unbind Activity and Service, we just need to call unbindService() which has been implemented in “Unbind Service” button event click code.

Run the app again and the main screen should be as shown in Fig. 10.9.

Click “Bind Service” button, and Logcat should show logs as shown in Fig. 10.10.

The logs show that, firstly, onCreate() of MyService gets called and then startDownload() and getProgress() get called, which means that we indeed call the methods provided by Service from Activity successfully.

Notice that, Service is for general use which means that MyService can bind to MainActivity and any other Activity, and after binding, they can all get the same DownloadBinder instance.



The screenshot shows the Android Logcat interface. At the top, there are three dropdown menus: 'com.example.servicetest (30991)', 'Verbose', and a search icon. Below these, the log output is displayed in a monospaced font. It contains three lines of text, each starting with '991/com.example.servicetest D/MyService:'. The first line is 'onCreate executed', the second is 'startDownload executed', and the third is 'getProgress executed'.

```
991/com.example.servicetest D/MyService: onCreate executed
991/com.example.servicetest D/MyService: startDownload executed
991/com.example.servicetest D/MyService: getProgress executed
```

Fig. 10.10 Bind service logs

## 10.4 Service Life Cycle

Just as Activity and Fragment, Service also has its lifecycle and methods like `onCreate()`, `onStartCommand()`, `onBind()`, and `onDestroy()` can potentially be called during Service lifecycle.

In anywhere of the project, after `startService()` of Context is called, the corresponding Service will start and `onStartCommand()` will be called. If this Service has never been created, then `onCreate()` will be called before `onStartCommand()`. After start, Service will keep running until `stopService()` or `stopSelf()` is called or gets recycled by the system. Notice that, every `startService()` call will make `onStartCommand()` get called once, but there will only be one instance of Service. Thus no matter how many times `startService()` gets called, Service will stop by just calling `stopService()` or `stopSelf()` once.

Also, we can call Context method `bindService()` to get a persistent connection of Service which will call Service method `onBind()`. Similarly, if this Service hasn't been created before, `onCreate()` will be called before `onBind()`. Then caller can get the `IBinder` instance returned from `onBind()` and communicate with Service freely. As long as the connection between caller and Service exists, Service will keep running until it gets recycled by the system.

If `stopService()` is called after `startService()`, then `onDestroy()` will be called and Service will be destroyed. Similarly, if `unbindService()` is called after `bindService()` then `onDestroy()` will be triggered too. These two scenarios are easy to understand. However, it is totally possible that we call both `startService()` and then `bindService()`, how to destroy Service under this situation? Based on the mechanism of Android system, a Service will be in running state after being started or bind, only both of these two conditions are false can the Service gets destroyed. Thus, we need to call `stopService()` and `unbindService()` to get `onDestroy()` be called for this scenario.

That's the whole lifecycle of Service.

## 10.5 More Techniques on Service

We covered some commonly used and fundamental concepts and uses of Service. Let us take a look at some of the more advanced topics of Service.

### 10.5.1 Use Foreground Service

As mentioned previously, since Android 8.0, only when the app is on foreground and visible can Service run stably. Once the app is on background, Service can get recycled by system at any time. If you want to keep Service running, then you can try to use foreground Service. The biggest difference between foreground Service and background Service is that, it will have an icon in the system status bar and pulling down the system bar will display more details and is very much like notification as shown in Fig. 10.11.

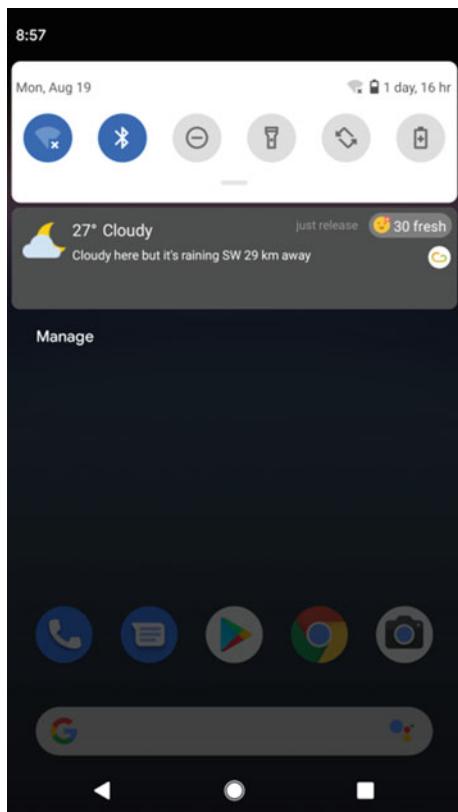
The icon in the status bar basically serves as a way of keeping the app visible on the foreground, thus system is less likely to recycle the foreground Service. Also, user can pull down the system status bar to know what apps are running, thus malicious apps cannot consume resources without letting the user know.

To create a foreground Service is not complicated at all. Modify MyService as code below:

```
class MyService : Service() {  
    ...  
    override fun onCreate() {  
        super.onCreate()  
        Log.d("MyService", "onCreate executed")  
        val manager = getSystemService(Context.NOTIFICATION_SERVICE) as  
            NotificationManager  
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
            val channel = NotificationChannel("my_service", "Foreground  
Service Notification",  
                NotificationManager.IMPORTANCE_DEFAULT)  
            manager.createNotificationChannel(channel)  
        }  
        val intent = Intent(this, MainActivity::class.java)  
        val pi = PendingIntent.getActivity(this, 0, intent, 0)  
        val notification = NotificationCompat.Builder(this, "my_service")  
            .setContentTitle("This is content title")  
            .setContentText("This is content text")  
            .setSmallIcon(R.drawable.small_icon)  
            .setLargeIcon(BitmapFactory.decodeResource(resources,  
                R.drawable.large_icon))  
            .setContentIntent(pi)  
            .build()  
        startForeground(1, notification)  
    }  
    ...  
}
```

We only need to update the code in onCreate() and the new code should look familiar as this is the notification creating code we discussed in Chap. 9. I actually also reused small\_icon and large\_icon from NotificationTest project. However, this time, I do not use NotificationManager to display the notification but instead call

**Fig. 10.11** Foreground Service



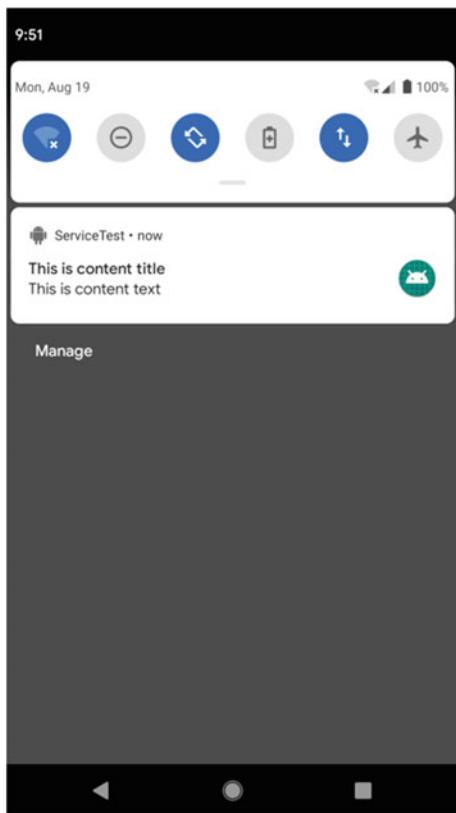
`startForeground()` after constructing the `Notification` object. This method takes two params: the first param is the id of the notification which is similar to the first param of `notify()`; the second param is the `Notification` object. Calling `startForeground()` will make `MyService` become a foreground Service and display in the system status bar.

Also, since Android 9.0, using foreground Service has to request permission in `AndroidManifest.xml` as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest">
    <uses-permission android:name="android.permission.
    FOREGROUND_SERVICE" />
    ...
</manifest>
```

Run the app again and click “Start Service” or “Bind Service” button, `MyService` will be started as foreground Service and display a notification icon in the system

**Fig. 10.12** Foreground service in system status bar



status bar. Pulling down the system status bar will reveal more details as shown in Fig. 10.12.

Now, even if you exit the app, MyService will keep running and you dont need to worry that it will get recycled. Of course the corresponding notification icon of MyService will be visible in the status bar. If user does not want the app keep running, they can manually kill the app and MyService will stop running.

Using foreground Service is as easy as this. If you still remember how to use notification from Chap. 9, this section should be easy for you.

### 10.5.2 Use IntentService

From the beginning of this chapter, we already know that by default, the code in Service will run in main thread, and if we want to run some time-consuming logic in Service, then very likely we will get ANR.

Then, we need to apply Android multithreading again. We should create a worker thread in the methods in Service to handle the time consuming logic. Thus, a standard Service can have the following form:

```
class MyService : Service() {
    ...
    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        thread {
            // business logic
        }
        return super.onStartCommand(intent, flags, startId)
    }
}
```

However, once this Service has started, it will keep running until stopService() or stopSelf() is called or gets recycled by the system. Thus, to let Service stop itself after business code is done, we can have the following code:

```
class MyService : Service() {
    ...
    override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
        thread {
            // business logic
            stopSelf()
        }
        return super.onStartCommand(intent, flags, startId)
    }
}
```

Though this is not complicated at all, there are always developers that forget to start thread or forget to call stopSelf(). In order to create asynchronous and self-stop Service, Android provides IntentService class. Let us take a look at how to use it.

Create MyIntentService class that inherits IntentService as shown in code below:

```
class MyIntentService : IntentService("MyIntentService") {

    override fun onHandleIntent(intent: Intent?) {
        // log the id of the current thread
        Log.d("MyIntentService", "Thread id is ${Thread.currentThread().name}")
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d("MyIntentService", "onDestroy executed")
    }
}
```

```
}
```

It is required to call the constructor of the super class first and pass in a random string which is only useful for debugging purpose. Then in this sub class, we need to implement abstract method `onHandleIntent()` in which we can execute time consuming logic without worrying ANR because this method is running in worker thread. To prove this, we log the id of the current thread in `onHandleIntent()`. Also, since this is an `IntentService`, it should stop after running. We override `onDestroy()` and log there to determine if Service is indeed stopped or not.

Next, edit `activity_main.xml` by adding a button to start `MyIntentService` as shown below:

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

    ...

    <Button
        android:id="@+id/startIntentServiceBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start IntentService" />

</LinearLayout>
```

Then edit `MainActivity` as code below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        startIntentServiceBtn.setOnClickListener {
            // log main thread id
            Log.d("MainActivity", "Thread id is ${Thread.currentThread().name}")
            val intent = Intent(this, MyIntentService::class.java)
            startService(intent)
        }
    }
}
```

In the click event code of “Start IntentService” button, `MyIntentService` gets started and the id of main thread gets logged which will be used to compare with

IntentService later. There is no difference between starting IntentService and the regular Service.

Do not forget that Service needs to register in AndroidManifest, as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.servicetest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        ...

        <service
            android:name=".MyIntentService"
            android:enabled="true"
            android:exported="true"/>

    </application>

</manifest>
```

Of course, you can use Android Studio tool to create IntentService. However, this will generate some code that we are not going to use at all, thus I decide to create the IntentService manually.

Run the app and the main screen should be as shown in Fig. 10.13.

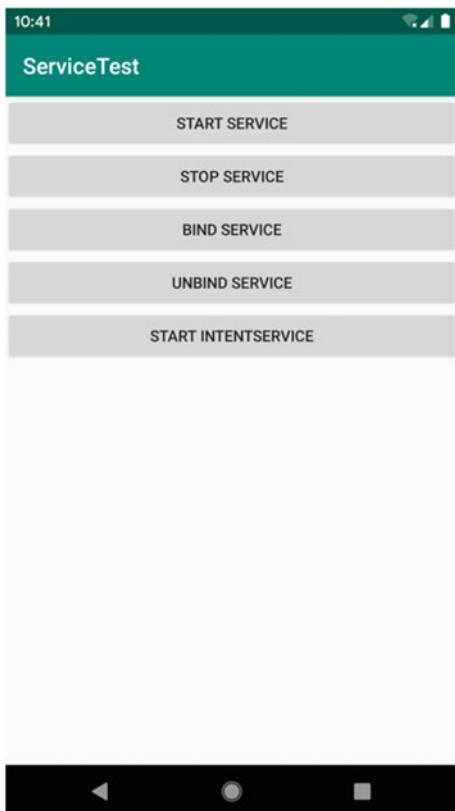
Click “Start IntentService” button and the logs in Logcat should be as shown in Fig. 10.14.

As you can see, not only the thread names where MyIntentService and MainActivity are located are different, but also the onDestroy() method is executed, which means that MyIntentService does stop automatically after it finishes running. IntentService are preferred choice for many developers because of its auto-creating worker thread and stopping feature. That’s everything about Service in this chapter. Time for Kotlin Class!

## 10.6 Kotlin Class: Advanced Topics in Generics

Do you still recall the Kotlin generics knowledge we covered in Chap. 8’s Kotlin Class? It is quite similar as generics in Java and easy to understand. However, Kotlin also provides some special features in generics, and we will discuss some of them in this chapter.

**Fig. 10.13** Update main screen of ServiceTest



```
com.example.servicetest (13701) ▾ Verbose ▾ Q▼  
701/com.example.servicetest D/MainActivity: Thread id is main  
628/com.example.servicetest D/MyIntentService: Thread id is IntentService[MyIntentService]  
701/com.example.servicetest D/MyIntentService: onDestroy executed
```

**Fig. 10.14** Logs from Starting IntentService

### 10.6.1 Reify Generic Types

There is no such concept as reified type parameters in Java, and we need to understand type erasure in Java to help us understand reified type parameter.

Before JDK 1.5, there was no generics in Java, and data structures like List can store data of any type. When we read the data, down cast is needed which is redundant and dangerous. For example, if there are strings and integers inside a List and when we try to read data from the List, casting to the same type will throw ClassCastException.

Java introduced generics in JDK 1.5. This makes data structure such as List easy to use and safe to use.

As a matter of fact, Java generics is implemented based on type erasure. It means that the type constraints of generic types only exist at compile time, and will still run according to the pre-JDK 1.5 mechanism, and the JVM will not recognize the generic types we specify in our code. For example, for List<String> collection, compilation checks if only Strings are added in this List but during runtime, JVM does not know what type of data is in this List.

The generics in majority of the languages that are based on JVM are implemented by type erasure which also includes Kotlin. This prevents us from using syntaxes like a is T or T::class.java, because the type information of T has been erased at runtime.

However, Kotlin has the concept called inline function which was discussed in Chap. 6's Kotlin Class. The code inside inline function will be replaced to the call site during compilation, which makes type erasure obsolete because the real type will replace the generic type in the inline function. This can be illustrated by Fig. 10.15.

In the end, it is actually as shown in Fig. 10.16.

Bar() is a inline function with generic type parameter, foo() calls bar() and after compilation, code in bar() will get the reified type.

This means that the generic type in inline function in Kotlin can be reified.

The syntax to reify generics requires the function to be an inline function which requires the inline modifier. Second, the reified keyword must be added where the generic is declared to indicate that the generic is to be implemented. This is shown in code below:

```
inline fun <reified T> getGenericType() {  
}
```

Type T is a reified generic type as there are inline and reified modifier. We can use this to get the reified type of the generic as shown below:

```
inline fun <reified T> getGenericType() = T::class.java
```

Even this is just one line of code, it implements something that cannot be done in Java: getGenericType() function returns the reified type. Syntax like T.class is not

**Fig. 10.15** Code replacement for inline function

```
fun foo() {  
    bar<String>()  
}  
  
inline fun <T> bar() {  
    // do something with T type  
}
```

**Fig. 10.16** Code after replacement

```
fun foo() {  
    // do something with String type  
}
```



**Fig. 10.17** Result of reified type

legit in Java, but in Kotlin, with reified type parameter, we can use syntax like `T::class.java`.

We can use the following code to test `getGenericType()`:

```
fun main() {
    val result1 = getGenericType<String>()
    val result2 = getGenericType<Int>()
    println("result1 is $result1")
    println("result2 is $result2")
}
```

Here two different generic types are specified for the `getGenericType()` function, and since the `getGenericType()` function returns the specific type of the specified generic type, here we print the returned result. Now run the `main()` function, and the result is shown in Fig. 10.17.

When the generic type is specified to `String`, type `java.lang.String` will be returned; when the generic type is specified to `Int`, then type `java.lang.Integer` will be returned.

Now, let us take a look at how to apply it in Android project.

### 10.6.2 Application of Reified Type in Android

Reified type makes syntaxes like `a is T`, `T::class.java` possible. This can be used flexibly to optimize our code. Here are some examples.

So far, we have covered all of the four main components in Android. They all need to be used with Intent except ContentProvider. For example, we can use the following code to start an activity:

```
val intent = Intent(context, TestActivity::class.java)
context.startActivity(intent)
```

Syntax like `TestActivity::class.java` can be optimized by using reified type in Kotlin.

Create `reified.kt` with code below:

```
inline fun <reified T> startActivity(context: Context) {
    val intent = Intent(context, T::class.java)
```

```

        context.startActivity(intent)
    }

```

The `startActivity()` function takes a param of `Context` type and the inline modifier and reified modifier make `T` a reified type. Now, the second param that `Intent` takes should be an `Activity` type, but since `T` is reified, we can directly pass in `T::class.java` and then use `startActivity()` of `Context` to start the activity.

With the code change above, if we want to start `TestActivity`, we can have the following code:

```
startActivity<TestActivity>(context)
```

Kotlin knows that `TestActivity` should be started. This makes code much more succinct.

However, there is still issue in `startActivity()`, because usually `Intent` will have some extra data in it when starting Activity, for instance:

```

val intent = Intent(context, TestActivity::class.java)
intent.putExtra("param1", "data")
intent.putExtra("param2", 123)
context.startActivity(intent)

```

The change above cannot pass these data. We can use higher order function which was introduced in Chap. 6 to solve this. Open `reified.kt` and add a function that override `startActivity()` as code below:

```

inline fun <reified T> startActivity(context: Context, block: Intent.()
    -> Unit) {
    val intent = Intent(context, T::class.java)
    block(intent)
    context.startActivity(intent)
}

```

The function param is defined on `Intent` class, then after instantiation of `Intent`, this function will be called and passed in the `Intent` instance. Thus `startActivity()` can pass data to `intent` in the Lambda expression as shown below:

```

startActivity<TestActivity>(context) {
    putExtra("param1", "data")
    putExtra("param2", 123)
}

```

Reified type and higher order function makes it possible to have succinct syntax as shown above, which makes starting Activity much easier.

This is just an example, and you can try to implement starting Service using the same pattern by yourself as an exercise.

### 10.6.3 Covariance and Contravariance

Covariance and contravariance are not used quite often, and I personally find them not easy to understand. However, lots of Kotlin APIs utilize covariance and contravariance, and it will be useful to learn these concepts so that we can have a deeper understanding of Kotlin.

A lot of the materials on these topics are obscure, but the fundamental idea is not that hard to understand. I will try to use as plain language as possible to explain to you.

There are two concepts, we need to know before we start. The param list in a generic type or interface is called in and the return value is out as shown in Fig. 10.18.

First, let us define three classes as shown below:

```
open class Person(val name: String, val age: Int)
class Student(name: String, age: Int) : Person(name, age)
class Teacher(name: String, age: Int) : Person(name, age)
```

Person class has name and age fields with sub classes Student and Teacher.

Is it legit that if a Student instance is passed to a method that takes param of Person type? Apparently this is legit because a Student is also a Person.

A follow up question is that, is it legit that if a function takes a param of List<Person> type and we pass in an instance of List<Student>? It looks legit but actually not allowed in Java. This is because List<Student> cannot be the sub type of List<Person> due to potential type casting issue.

We can use an example to illustrate why there can be type casting issue. Define SimpleData class as code below:

```
class SimpleData<T> {
    private var data: T? = null

    fun set(t: T?) {
        data = t
    }

    fun get(): T? {
        return data
    }
}
```

**Fig. 10.18** Definition of in and out

```
interface MyClass<T> {
    fun method(param: T): T
}
```

SimpleData is a generic type which has generic data field, set() will assign value to data and get() will return the value of data.

Assume it is legit to pass SimpleData<Student> instance to method that takes param of SimpleData<Person> type, then the following code is also legit:

```
fun main() {
    val student = Student("Tom", 19)
    val data = SimpleData<Student>()
    data.set(student)
    handleSimpleData(data) // this will throw exception, assume
    compilation can succeed
    val studentData = data.get()
}

fun handleSimpleData(data: SimpleData<Person>) {
    val teacher = Teacher("Jack", 35)
    data.set(teacher)
}
```

In main(), an instance of Student is put into SimpleData<Student> and then gets passed to handleSimpleData(). This method takes a param of SimpleData<Person> type (assume compilation succeeds), and creates a Teacher instance to replace the original data in SimpleData<Person> argument. This is legit because Teacher is a sub class of Person and it is safe to replace Person with Teacher.

But the problem arises immediately. Back in the main() method, we call the get() method of SimpleData<Student> to get the Student data it encapsulates internally, but now SimpleData<Student> actually contains an instance of Teacher, so a type conversion exception is bound to occur.

To avoid this potential type cast exception, Java makes it illegit to pass param like this. Although Student is a sub class of Person, SimpleData<Student> is not a sub type of SimpleData<Person>.

The main problem is that handleSimpleData() set a Teacher instance to SimpleData<Person>. If SimpleData on T is read only, then there is no risk of ClassCastException. Will this change make SimpleData<Student> a sub type of SimpleData<Person>?

Now, we can introduce the definition of covariance. For MyClass<T>, if A is a sub type of B and MyClass<A> is a sub type of MyClass<B>, then MyClass is covariant in T.

As just mentioned, if a generic type is read only then there is no risk of ClassCastException. To do this, all the methods in MyClass<T> cannot take param of T type. In other words, T can only be out but not in.

Edit SimpleData as code below:

```
class SimpleData<out T>(val data: T?) {
    fun get(): T? {
        return data
    }
}
```

Here we have modified the SimpleData class by adding an `out` keyword in front of the declaration of the generic `T`. This means that now `T` can only appear in the `out` position, not in position, and it also means that SimpleData is covariant on the generic `T`.

Since `T` cannot be `in`, we cannot use `set()` to assign value to data argument and so the constructor assign value to data. You might ask that why `T` is `in` for constructor? This is because of `val` modifier which makes `T` in constructor is still read only and is legit and safe. Also even if `var` modifier is used, if `private` modifier is added to make sure that `T` is immutable to outside, then this is still legit.

With this change, the following code can compile and with risk of `ClassCastException`:

```
fun main() {
    val student = Student("Tom", 19)
    val data = SimpleData<Student>(student)
    handleMyData(data)
    val studentData = data.get()
}

fun handleMyData(data: SimpleData<Person>) {
    val personData = data.get()
}
```

Since `SimpleData` is covariant, then `SimpleData<Student>` is a sub type of `SimpleData<Person>`. It is safe to pass param to `handleMyData()`.

Inside `handleMyData()`, the data in `SimpleData` is read, and since `Person` is the super type of `Student`, type casting is safe, thus there is no issue for the above code.

At this point, you've pretty much gotten the hang of covariance, but there's one last example to review. As we mentioned earlier, if a method takes a `List<Person>` argument and passes in an instance of `List<Student>`, that's not allowed in Java. Please notice my language here, this is not allowed in Java.

This is because Kotlin already makes lots of internal APIs covariant which include various collection classes and interfaces. In Chap. 2, we mentioned that `List` in Kotlin is read only, and to add data to `List`, we need to use `MutableList`. Since `List` is immutable, it is naturally covariant. Here is a simplified version of source code of `List`:

```
public interface List<out E> : Collection<E> {
    override val size: Int
    override fun isEmpty(): Boolean
    override fun contains(element: @UnsafeVariance E): Boolean
    override fun iterator(): Iterator<E>
    public operator fun get(index: Int): E
}
```

Generic type `E` has `out` modifier, which means that `List` is covariant in `E`. Notice that theoretically, if `List` is covariant in `E`, then `E` can only be `out`; however, in `contains()`, `E` is `in`.

This is not legit because of class cast exception as mentioned before. However, `contains()` only checks if the current collection has the passed in argument and will

not mutate the argument. Thus, it is actually safe to pass in E. In order to pass compilation, `@UnsafeVariance` annotation is added to inform compiler that it is safe to do so and compilation will succeed. Use this carefully and only when you're absolutely sure that this is safe.

That is covariance, and let us take a look at contravariance.

### 10.6.4 Contravariance

After discussion of covariance, contravariance should be easier to understand because these two concepts are closely related.

By definition they are the opposite. For `MyClass<T>`, if A is sub type of B and `MyClass<B>` is a sub type of `MyClass<A>`, then `MyClass` is contravariant in T. The difference between covariance and contravariance is shown in Fig. 10.19.

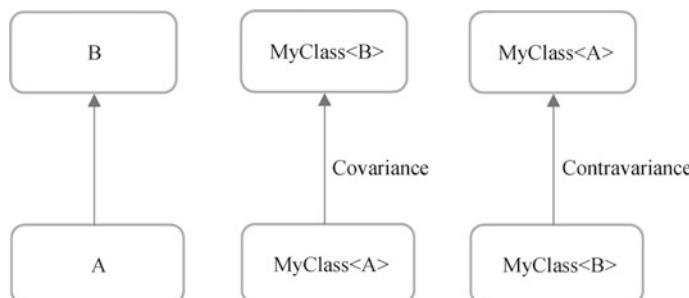
The definition sounds counterintuitive as A is a sub type of B, how come that `MyClass<B>` can be sub type of `MyClass<A>`? This can be clarified with the following example.

Define Transformer interface as code below:

```
interface Transformer<T> {
    fun transform(t: T): String
}
```

The `transform()` method takes a param of T type and return value of String type. The implementation will be done by sub types. We have the following implementation of Transformer interface:

```
fun main() {
    val trans = object : Transformer<Person> {
        override fun transform(t: Person): String {
            return "${t.name} ${t.age}"
        }
    }
}
```



**Fig. 10.19** Covariance and Contravariance comparison

```

    }
    handleTransformer(trans) // This will throw exception

fun handleTransformer(trans: Transformer<Student>) {
    val student = Student("Tom", 19)
    val result = trans.transform(student)
}

```

The val trans is an anonymous implementation of Transformer<Person> which return a string with the data inside the t argument which is of type Person. However, handleTransformer() takes a param of Transformer<Student> type and creates an instance of Student then uses the transform() in trans argument to transform the Student object to a String.

From code safety perspective, there is no issue in the above code as Student is a sub class of Person, and using the anonymous implementation to transform Student object to String is also safe, no class cast exception risk at all. However, handleTransformer() will show syntax error because Transformer<Person> is not a sub type of Transformer<Student>.

Contravariance is used to resolve this kind of problem. Edit Transformer interface as code below:

```

interface Transformer<in T> {
    fun transform(t: T): String
}

```

The in modifier means that T can only be in but not out, which means that Transformer is contravariant in T.

With this small change, the code above will compile and run without problem because now, Transformer<Person> is a sub type of Transformer<Student>.

That is how contravariance is used. Why T cannot be out? Assume if T can be out, let us see what will happen.

Edit Transformer as code below:

```

interface Transformer<in T> {
    fun transform(name: String, age: Int): @UnsafeVariance T
}

```

Now transform() takes name and age as params and the return value is T type. Since contravariance does not allow T to be out, annotation @UnsafeVariance is added to compile, which is the same as what is done in List source code.

The risk of the above code can be illustrated with the following code:

```

fun main() {
    val trans = object : Transformer<Person> {
        override fun transform(name: String, age: Int): Person {
            return Teacher(name, age)
        }
    }
}

```



**Fig. 10.20** Exception caused by misusing contravariance

```

        }
    handleTransformer(trans)
}

fun handleTransformer(trans: Transformer<Student>) {
    val result = trans.transform("Tom", 19)
}

```

This is a classic example of misusing contravariance that causes `ClassCastException`. In anonymous implementation of `Transformer<Person>`, a `Teacher` object is created with name and age arguments and returned. `Teacher` is sub class of `Person`, thus return `Teacher` object is legit here.

However, in `handleTransformer()`, `transform()` of `Transformer<Student>` is called and expects a `Student` object will be returned while `transform()` actually returns an object of `Teacher` type. This will definitely cause `ClassCastException`.

Run the code and the result should be as shown in Fig. 10.20.

The exception tells us that `Teacher` cannot be cast to `Student`.

This demonstrates that we need to follow the rules to use covariance and contravariance and only allow covariance to be in and contravariance to be out to eliminate `ClassCastException`. Although we can use `@UnsafeVariance` annotation to help us flexibly use covariance and contravariance, the risk is on us.

Last, let us see an example of application of contravariance in Kotlin internal API. A classic example is `Comparable` which is used to compare two objects. Its source code is as shown below:

```

interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

```

`Comparable` is contravariant in `T` and `compareTo()` will have implementation to actually do the comparison. Why `Comparable` interface is contravariance? Assume that we have `Comparable<Person>` that implements the logic comparing age of two `Person` objects, then the same logic can be applied to `Student` objects, thus it makes sense to make `Comparable<Person>` a sub type of `Comparable<Student>`. This is a classic use case of contravariance.

That is all for covariance and contravariance, and time to summarize what we have learned in this chapter.

## 10.7 Summary and Comment

In this chapter, we covered lots of important topics related to Service, including Android multithreading, basic use of Service, Service lifecycle, foreground Service, IntentService, etc. These should meet the needs of most scenarios, and you should be able to figure out the rest by yourself.

In this chapter's Kotlin Class, we discussed advanced topics in generics which is leap of understanding of generics. Reified type parameters is very useful in Kotlin and with examples in this book you should have already felt it and it can really help you optimize your code. Covariance and contravariance are difficult to understand but hopefully with the examples and plain language explanation, you can grasp the essence of these two concepts.

This is a monumental chapter as we finished all the four main components in Android. You are no longer a rookie Android developer and you can work independently to solve problems.

But still, everything we have done so far can run in local machine while almost all the apps in the store can connect to internet. In the next chapter, let us take a look at Android networks programming.

# Chapter 11

## Exploring New World with Network Technologies



If you cannot connect to the internet while playing with the phone, you will feel bored in a short time. Indeed, twenty-first century is the century of the internet, and nowadays, almost all the PCs, phones, tablets, and TVs are capable of connecting to the internet.

Apparently, all Android phones have access to the internet. As developers, we should make good use of the internet to make better apps. Apps like Messenger, Twitter, and Pinterest all use the internet extensively. This chapter mainly focuses on how to connect with server from client side using HTTP and parse the data returned from server which is commonly used in Android. Now let us take a look at how to do it.

### 11.1 WebView

Sometimes we need to display some website, while we do not want to open the system browser, but apparently, we cannot implement our own browser. What can we do?

Android provides WebView to help us embed a browser in our app to display web pages.

WebView is also easy to use. Let me use an example to demonstrate. Create WebViewTest and edit activity\_main.xml as code below:

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

    <WebView
        android:id="@+id/webView"
```

```

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

</LinearLayout>

```

The new control WebView is used to display the web page. We set the id of this widget and let it fill the whole screen. Next, edit MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        webView.settings.javaScriptEnabled(true)
        webView.webViewClient = WebViewClient()
        webView.loadUrl("https://www.google.com")
    }
}

```

We can set some properties of browser with getSettings() of WebView. Here we just call setJavaScriptEnabled() to enable support for JavaScript.

Next, we call setWebViewClient() of WebView and pass in an instance of WebViewClient. This will keep the web page redirecting happen inside WebView instead of opening system browser.

The last step is to use loadUrl() of WebView and pass the web URL in so that the corresponding web page can be displayed. Here we will simply display Google search page.

Notice that since we are trying to access network here and accessing network requires permission, we need to edit AndroidManifest.xml as shown below:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.webviewtest">

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

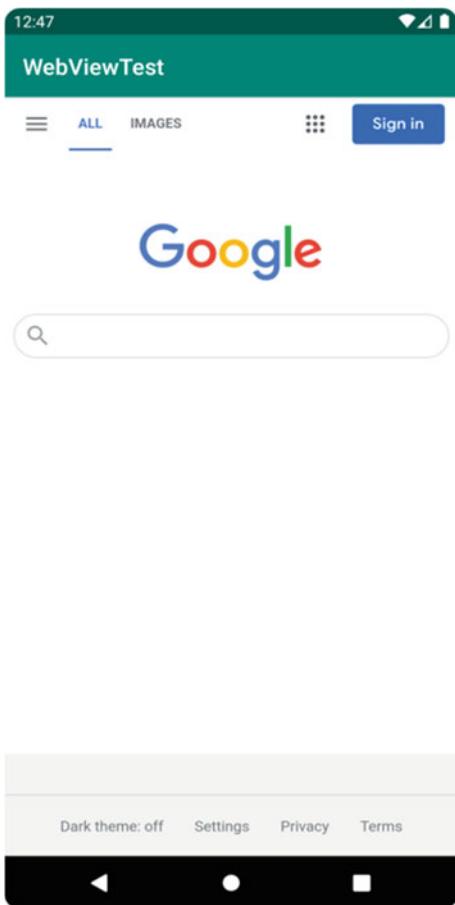
    ...
</manifest>

```

Before running the app, make sure your phone or emulator is connected to the internet. If you are using emulator, you just need to make sure that the computer in which the emulator is running on is connected to the internet. Run the app, and it should be as shown in Fig. 11.1.

WebViewTest has the basic function of a browser; it can display the Google search page, and you can also browse more web pages inside it. Of course, there are

**Fig. 11.1** Use WebView to load web page



more advanced techniques to use WebView which I will not cover as these are not something this chapter will focus on. Next, let us take a look at HTTP protocol.

## 11.2 Use HTTP to Access Network

HTTP itself is worth a book, and apparently, this book is about learning Android; thus you just need to know enough to unblock learning. The fundamental idea is very simple. Client sends a HTTP request to server, and after receiving the request, server will send back response to client, and then client just parses the data and process accordingly. Simple, right? That is the fundamental mechanism under a browser. Take the aforementioned WebView as an example, we sent a HTTP request to Google server, then server parses the request and determines that we want to access to the search page, then it will return the HTML code of that page, and WebView

will use the kernel of system browser to parse the HTML code to render the web page.

Basically, WebView helps us to send HTTP request, respond to the response of server, parse the data, and render the web page. But it does a great job to encapsulate all of these, and we cannot perceive these at all. To understand this process better, let us try to manually send HTTP request.

### 11.2.1 Use HttpURLConnection

We used to have two main methods to send HTTP request in Android: HttpURLConnection and HttpClient. But HttpClient had too many APIs and difficulty to extend; thus Android did not recommend to use this API more and more. Finally since Android 6.0, HttpClient got removed totally. Thus we will only cover the officially recommended HttpURLConnection.

To get an instance of HttpURLConnection, we just need to create an object of URL and pass in the url and call openConnection() as shown below:

```
val url = URL("https://www.baidu.com")
val connection = url.openConnection() as HttpURLConnection
```

Then we configure the method of HTTP request we will be using. There are two commonly used methods: GET and POST. Get means to get data from server, while POST means to send or post data to server. Below is a code example:

```
connection.requestMethod = "GET"
```

Next we can configure other things such as connect timeout, read timeout, and message header that server expects, etc. You can configure based on your need, and below is just an example:

```
connection.connectTimeout = 8000
connection.readTimeout = 8000
```

To get the input stream returned by server, we can call getInputStream(), and then we can read from the input stream:

```
val input = connection.getInputStream()
```

At the end we should close the HTTP connection with disconnect():

```
connection.disconnect()
```

Next, let us use an example to demonstrate how to use HttpURLConnection. Create NetworkTest project, and edit activity\_main.xml as shown below:

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

    <Button
        android:id="@+id/sendRequestBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send Request" />

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

        <TextView
            android:id="@+id/responseText"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

    </ScrollView>

</LinearLayout>
```

Notice that we use a new widget: ScrollView. This is because there is only limited space in the phone and may not be able to host all the content. With ScrollView, we can scroll to see the content that is outside the current screen. The newly added button is used to send HTTP request; TextView is used display the data returned from server.

Next edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        sendRequestBtn.setOnClickListener {
            sendRequestWithHttpURLConnection()
        }
    }

    private fun sendRequestWithHttpURLConnection() {
        // start thread to send request
        thread {
            var connection: HttpURLConnection? = null
            try {
```

```

    val response = StringBuilder()
    val url = URL("https://www.baidu.com")
    connection = url.openConnection() as HttpURLConnection
    connection.connectTimeout = 8000
    connection.readTimeout = 8000
    val input = connection.getInputStream
    // read the input stream
    val reader = BufferedReader(InputStreamReader(input))
    reader.use {
        reader.forEachLine {
            response.append(it)
        }
    }
    showResponse(response.toString())
} catch (e: Exception) {
    e.printStackTrace()
} finally {
    connection?.disconnect()
}
}

private fun showResponse(response: String) {
    runOnUiThread {
        // display the result in UI
        responseText.text = response
    }
}
}

```

In the click event code of “Send Request” button, we call sendRequestWithHttpURLConnection(), and in this method, we start a worker thread to send HTTP request by using HttpURLConnection, and the address is the search page of Google. Then we use BufferedReader to read the returned stream and pass the result to showResponse(). In showResponse(), runOnUiThread() is called, and in the Lambda expression, we display the data.

The reason why we need to call runOnUiThread() is because Android does not allow worker thread to update UI. We discussed asynchronous message handling mechanism in Sect. 10.2.3, and runOnUiThread() just encapsulates the mechanism. With the help of this method, we can update the screen with data coming from server.

That’s the whole process. But before running the app, do not forget to request for permission. Edit AndroidManifest.xml as code below:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.example.networktest">

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

```

**Fig. 11.2** Server response

```
</manifest>
```

Run the app and click “Send Request,” and the result is as shown in Fig. 11.2.

The response is not very readable which is just plain HTML code, and usually browser will parse the code to beautiful web page and present the web page to us.

To submit or post data to server, we just need to use the HTTP POST method and write the data in the output stream. Notice that data should be key-value format and use “&” to separate them. For example, if we want to send user name and password, we can have the following format:

```
connection.requestMethod = "POST"
val output = DataOutputStream(connection.getOutputStream)
output.writeBytes("username=admin&password=123456")
```

### 11.2.2 Use OkHttp

Of course, we have more options than HttpURLConnection. In today's world there are so many excellent open-source libraries for us to choose that can replace HttpURLConnection, and OkHttp is one of the best libraries.

OkHttp is developed by Square who contributes a lot for open-source community. Besides OkHttp, they also open source Retrofit, Picasso, etc. OkHttp's interfaces are easy to use, and its implementation under the hood is special too and exceeds HttpURLConnection. It is the no. 1 choice for Android developers when it comes to network library. In this section, we will take a look at how to use OkHttp. The web page for this project is <https://github.com/square/okhttp>.

To use OkHttp, we need to add dependency to OkHttp library. Edit app/build.gradle, and in dependencies closure, add the following content:

```
dependencies {  
    ...  
    implementation 'com.squareup.okhttp3:okhttp:4.1.0'  
}
```

The above dependency will download two libraries: One is OkHttp lib, and another is Okio lib which is the communication foundation lib of OkHttp. When I wrote the book, 4.1.10 was the latest version, and you can find the latest version in the website mentioned before.

To use OkHttp, we first need to create an instance of OkHttp as shown below:

```
val client = OkHttpClient()
```

Then, to send HTTP request, we need to create an object of Request:

```
val request = Request.Builder().build()
```

This is an empty request object which does not really do anything yet. We need to use other methods before build() to enrich this Request object. For example, to set the web address by url(), we can have the following code:

```
val request = Request.Builder()  
    .url("https://www.baidu.com")  
    .build()
```

Then call newCall() of OkHttpClient to create an object of Call and call its execute() to send the request and get the response from server as shown below:

```
val response = client.newCall(request).execute()
```

The Response object is the response returned from server, and we can use the following code to get the actual data:

```
val responseData = response.body?.string()
```

To send POST request, some extra steps are needed. We need to create an object of Request Body to store the params that will be posted, as code below:

```
val requestBody = FormBody.Builder()  
    .add("username", "admin")  
    .add("password", "123456")  
    .build()
```

Then pass the RequestBody object to post() in Request.Builder:

```
val request = new Request.Builder()  
    .url("https://www.baidu.com")  
    .post(requestBody)  
    .build()
```

The rest is the same as GET request; we can call execute() to send the request and get the server response.

That is all we need to start using OkHttp, and we will cover how to use OkHttp and Retrofit together later. Now let us reimplement NetworkTest with OkHttp.

There is no need to change the layout file. Edit MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        sendRequestBtn.setOnClickListener {  
            sendRequestWithOkHttp()  
        }  
    }  
    ...  
    private fun sendRequestWithOkHttp() {  
        thread {  
            try {  
                val client = OkHttpClient()  
                val request = Request.Builder()  
                    .url("https://www.baidu.com")  
                    .build()  
                val response = client.newCall(request).execute()  
                val responseData = response.body?.string()  
                if (responseData != null) {  
                    showResponse(responseData)  
                }  
            } catch (e: Exception) {  
            }  
        }  
    }  
}
```

```
        e.printStackTrace()
    }
}
}
```

We simply add sendRequestWithOkHttp() and call this method in click event of “Send Request” button. In this method, we start a worker thread, and in the worker thread, we use OkHttp to send HTTP request to Google as discussed previously. Then we still call showResponse() to display the data from server.

With little change, we can run the app again. Click “Send Request” button, and you should see the same result as shown in last section which proves that using OkHttp to send HTTP request is successful.

### 11.3 Parse XML Data

Usually, every app that needs to access network will have its own corresponding server, and we can send data to server and get data from server. One question is: what format should we use for the data to transfer in the internet? Plain text does not work because the other side does not know how to use the text at all. Thus we usually transfer formatted data which has format requirement and syntax, and the other side will follow the requirement and syntax to parse the data to get information from.

The most commonly used formats are XML and JSON. In this section, we will start with XML.

First, how can we get XML data? Now let us build a simple web server which will provide XML text, and we will have our app to access this server and parse the XML text.

It is actually quite simple to set up web server, and there are any kinds of servers that we can choose from. Here we will use Apache server. I will only demonstrate how to set up web server in Windows as Mac and Ubuntu both install Apache server by default, and you just need to start it. If you use these two OS, Google how to use it by yourself.

To set up Apache server in Windows, you need to download the Apache installer which can be found in <http://httpd.apache.org>. After downloading, double click to install as shown in Fig. 11.3.

Then click Next all the way to the window that let you input your domain name, and I will use test.com as shown in Fig. 11.4. You can use whatever name you want.

Then keep clicking Next to the window that allows you to choose the installation path, and here I choose to install under C:\Apache folder. Click Next all the way until finish installation. After installation, server will start automatically, and you can open the browser to verify it. Type in 127.0.0.1 to verify. If the web page as shown in Fig. 11.5 can show up, it means that server has been started successfully.

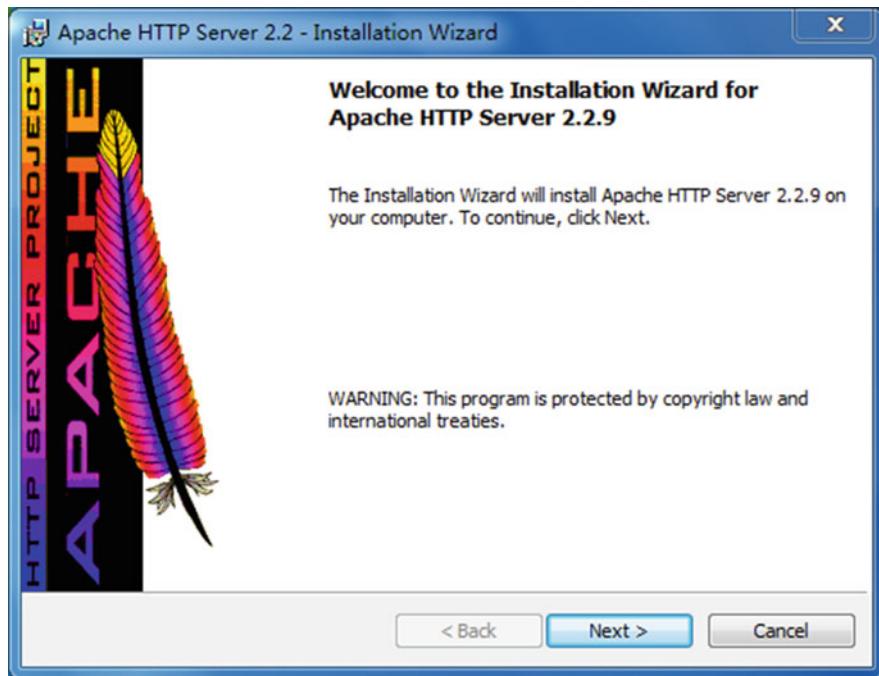


Fig. 11.3 Apache server installation wizard

Next, create `get_data.xml` in `C:\Apache\htdocs` and edit this file to add the XML content as shown below:

```
<apps>
  <app>
    <id>1</id>
    <name>Google Maps</name>
    <version>1.0</version>
  </app>
  <app>
    <id>2</id>
    <name>Chrome</name>
    <version>2.1</version>
  </app>
  <app>
    <id>3</id>
    <name>Google Play</name>
    <version>2.3</version>
  </app>
</apps>
```

Then type in `http://127.0.0.1/get_data.xml` should show content as shown in Fig. 11.6.

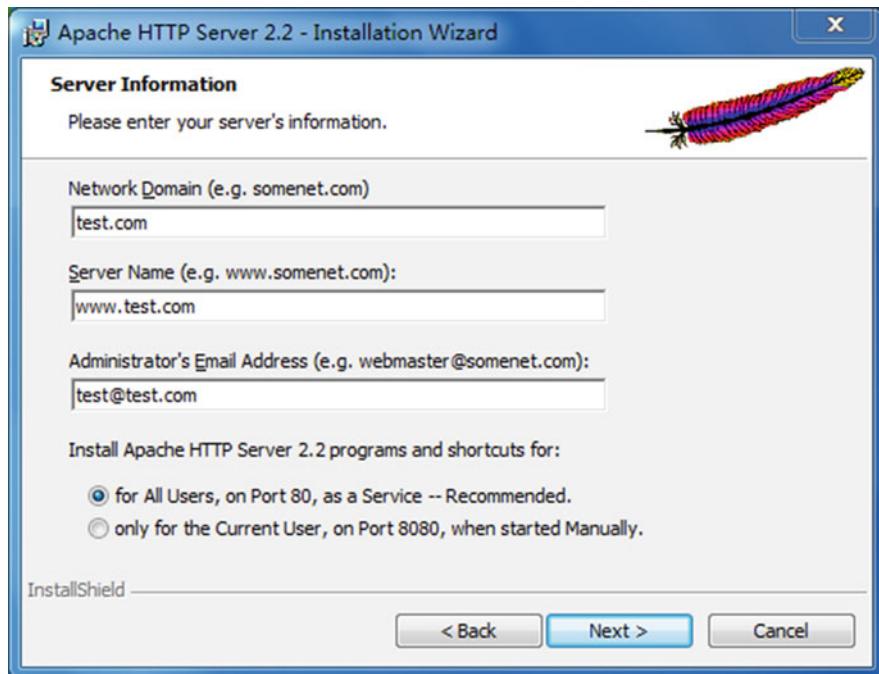


Fig. 11.4 Domain and server information

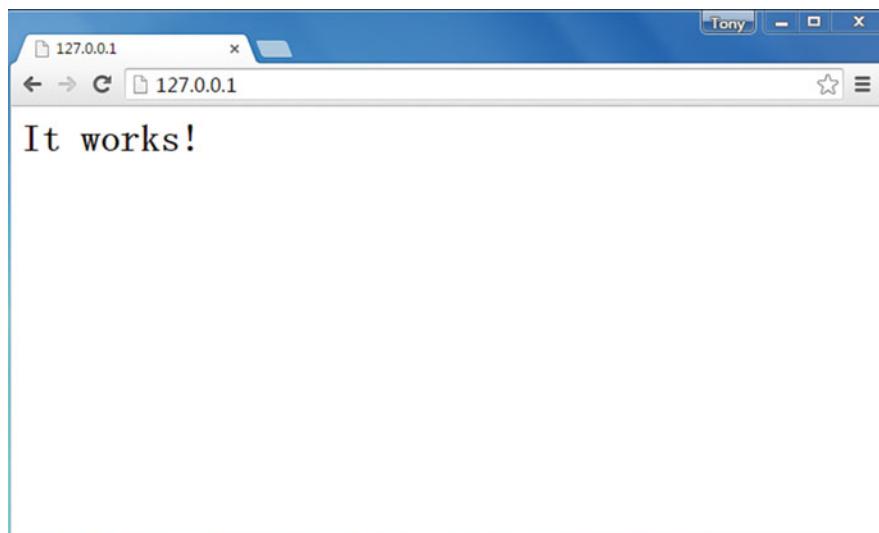


Fig. 11.5 Apache server default web page



Fig. 11.6 Verify XML data in browser

That is all the preparation work we need to do. Next, let us try to get the XML content and parse the content in our Android app.

### 11.3.1 Pull Parser

There are a few ways to parse XML data, and we will cover two commonly used ones: Pull parse and SAX parser. For simplicity, we will work on top of NetworkTest project so that we can reuse the code for network communication and focus on XML data parsing.

Since we can get the XML data, the only thing we need to do is to parse the data and get the information we want. Edit MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {
    ...
    private fun sendRequestWithOkHttp() {
        thread {
            try {
                val client = OkHttpClient()
                val request = Request.Builder()
                    // set the server address to local host
                    .url("http://10.0.2.2/get_data.xml")
                    .build()
                val response = client.newCall(request).execute()
                val responseData = response.body?.string()
                if (responseData != null) {

```

```

        parseXMLWithPull(responseData)
    }
} catch (e: Exception) {
    e.printStackTrace()
}
}
...
private fun parseXMLWithPull(xmlData: String) {
    try {
        val factory = XmlPullParserFactory.newInstance()
        val xmlPullParser = factory.newPullParser()
        xmlPullParser.setInput(StringReader(xmlData))
        var eventType = xmlPullParser.eventType
        var id = ""
        var name = ""
        var version = ""
        while (eventType != XmlPullParser.END_DOCUMENT) {
            val nodeName = xmlPullParser.name
            when (eventType) {
                // start parsing a node
                XmlPullParser.START_TAG -> {
                    when (nodeName) {
                        "id" -> id = xmlPullParser.nextText()
                        "name" -> name = xmlPullParser.nextText()
                        "version" -> version = xmlPullParser.nextText()
                    }
                }
                // complete parsing a node
                XmlPullParser.END_TAG -> {
                    if ("app" == nodeName) {
                        Log.d("MainActivity", "id is $id")
                        Log.d("MainActivity", "name is $name")
                        Log.d("MainActivity", "version is $version")
                    }
                }
            }
            eventType = xmlPullParser.next()
        }
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
}

```

First, the HTTP request address has been changed to [http://10.0.2.2/get\\_dtaa.xml](http://10.0.2.2/get_dtaa.xml), and 10.0.2.2 is the local IP address for emulator. After getting the response from server, data will not be displayed directly, and instead we will use `parseXMLWithPull()` to parse the data.

In `parseXMLWithPull`, we first create an object of `XmlPullParserFactory` and use this instance to get an object of `XmlPullParser` and call its `setInput()` to pass in the XML data to prepare for parsing. The parse process is quite simple. We can get the



The screenshot shows the Android Logcat interface. At the top, there is a search bar with the text "com.example.networktest (14325) ▾" and a dropdown menu labeled "Verbose ▾". Below the header, the log output is displayed in a monospaced font. The log entries are as follows:

```
597/com.example.networktest D/MainActivity: id is 1
597/com.example.networktest D/MainActivity: name is Google Maps
597/com.example.networktest D/MainActivity: version is 1.0
597/com.example.networktest D/MainActivity: id is 2
597/com.example.networktest D/MainActivity: name is Chrome
597/com.example.networktest D/MainActivity: version is 2.1
597/com.example.networktest D/MainActivity: id is 3
597/com.example.networktest D/MainActivity: name is Google Play
597/com.example.networktest D/MainActivity: version is 2.3
```

**Fig. 11.7** Logcat records of XML parsing result

current parse event by using `getEventType()` and then parse in a while loop until the parse event is equal to `XmlPullParser.END_DOCUMENT`. Before that parse work is not done yet, and `next()` will return the next parse event.

In the while loop, the current node's name can be acquired by using `getName()`. If the node name is equal to id, name, or version, we can call `nextText()` to get the content of this node, and after parsing an app node, we will display the content.

That is it for the whole process. But before running the app, we need to do some configuration. Starting from Android 9.0, app can only use HTTPS request by default because HTTP is considered unsafe and thus not supported. However, the Apache server we set up is using HTTP.

In order to use HTTP, we need to configure as follows. Right click res → New → Directory, and create xml folder. Then right click xml folder → New → File, and then create network\_config.xml. Edit network\_config.xml as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config cleartextTrafficPermitted="true">
        <trust-anchors>
            <certificates src="system" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

After this we can use HTTP in our app and let us test it. Run NetworkTest and click “Send Request” button, and Logcat should have the records as shown in Fig. 11.7.

As you can tell, the XML content has been successfully parsed.

### 11.3.2 SAX Parser

Pull parser is easy to use, but it is not our only choice. SAX parser is another commonly used XML parser. It is more complicated than Pull to use, but the syntax is clearer.

To use SAX parser, usually we will create a new class that inherits DefaultHandler, then override the five methods of super class as shown below:

```
class MyHandler : DefaultHandler() {

    override fun startDocument() {
    }

    override fun startElement(uri: String, localName: String, qName: String, attributes: Attributes) {
        Attributes)
    }

    override fun characters(ch: CharArray, start: Int, length: Int) {
    }

    override fun endElement(uri: String, localName: String, qName: String) {
    }

    override fun endDocument() {
    }

}
```

These five methods are self-explanatory. Among them, startElement(), characters(), and endElement() take params, and the data parsed from XML will be passed to these methods as arguments. It is worth noting that, when getting the content from node, characters() can be called multiple times and newline will also be treated as content, and we need to handle this in code.

Next, let us use SAX parser to implement the same function as with using Pull parser. Create ContentHandler class that inherits DefaultHandler and overrides the five methods from super class as shown below:

```
class ContentHandler : DefaultHandler() {

    private var nodeName = ""

    private lateinit var id: StringBuilder

    private lateinit var name: StringBuilder
```

```
private lateinit var version: StringBuilder

override fun startDocument() {
    id = StringBuilder()
    name = StringBuilder()
    version = StringBuilder()
}

override fun startElement(uri: String, localName: String, qName: String, attributes: Attributes) {
    // record the current node name
    nodeName = localName
    Log.d("ContentHandler", "uri is $uri")
    Log.d("ContentHandler", "localName is $localName")
    Log.d("ContentHandler", "qName is $qName")
    Log.d("ContentHandler", "attributes is $attributes")
}

override fun characters(ch: CharArray, start: Int, length: Int) {
    // add to corresponding StringBuilder based on the node name
    when (nodeName) {
        "id" -> id.append(ch, start, length)
        "name" -> name.append(ch, start, length)
        "version" -> version.append(ch, start, length)
    }
}

override fun endElement(uri: String, localName: String, qName: String) {
    if ("app" == localName) {
        Log.d("ContentHandler", "id is ${id.toString().trim()}")
        Log.d("ContentHandler", "name is ${name.toString().trim()}")
        Log.d("ContentHandler", "version is ${version.toString().trim()}")
    }
    // clear StringBuilder
    id.setLength(0)
    name.setLength(0)
    version.setLength(0)
}

override fun endDocument() {
}
```

Code above creates a `StringBuilder` object for `id`, `name`, and `version` correspondingly and initializes them in `startDocument()`. When parsing for a node starts, `startElement()` will be called. Its `localName` param keeps the name of the current node. Then when parsing the content in the node, `characters()` will be called, and we

can determine to add the parsed content to the corresponding `StringBuilder` object based on the name of current node. Then in `endElement()`, if the app node is parsed, we can log the content of id, name, and version. Notice that, currently, id, name, and version may contain backspace or newline, and thus we need to call `trim()` before we log the content. We also need to clear `StringBuilder` after logging; otherwise it will affect the next read.

The rest is simple. Edit `MainActivity` as code below:

```
class MainActivity : AppCompatActivity() {
    ...
    private fun sendRequestWithOkHttp() {
        thread {
            try {
                val client = OkHttpClient()
                val request = Request.Builder()
                    // set the server address to local machine
                    .url("http://10.0.2.2/get_data.xml")
                    .build()
                val response = client.newCall(request).execute()
                val responseData = response.body?.string()
                if (responseData != null) {
                    parseXMLWithSAX(responseData)
                }
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }
    ...
    private fun parseXMLWithSAX(xmlData: String) {
        try {
            val factory = SAXParserFactory.newInstance()
            val xmlReader = factory.newSAXParser().getXMLReader()
            val handler = ContentHandler()
            // set the ContentHandler instance to XMLReader
            xmlReader.contentHandler = handler
            // start parsing
            xmlReader.parse(InputSource(StringReader(xmlData)))
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

This time, we use `parseXMLWithSAX()` to parse the XML data from server. `parseXMLWithSAX()` first create an object of `SAXParserFactory` object and then get an instance of `XMLReader`, and next we pass the instance of `ContentHandler` to `XMLReader`. Finally we call `parse()` to start parsing.

Run the app again, click “Send Request” button, and check the Logcat records, and you should see the same result as in Fig. 11.7.

Beside Pull parser and SAX parser, DOM parser is also commonly used, but I will not cover it here, and you can search related material.

## 11.4 Parse JSON Data

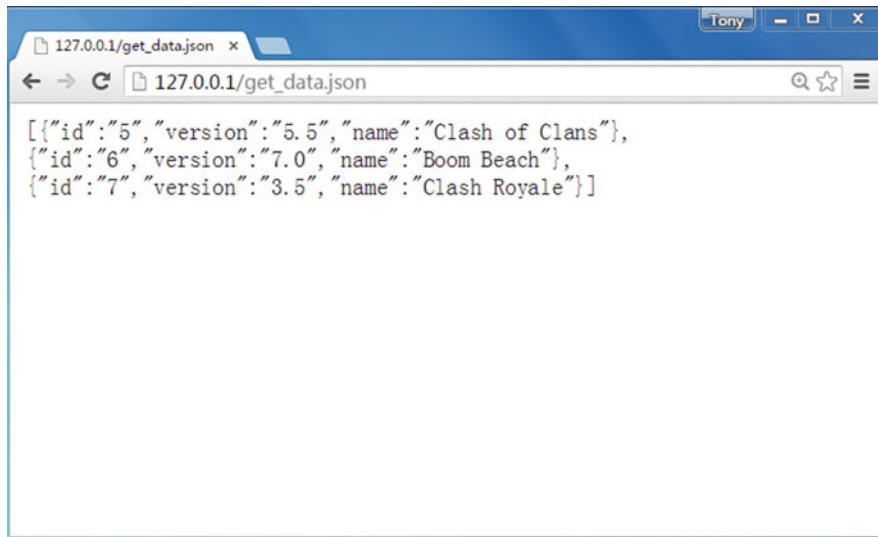
As promised, after XML, let us take a look at another important format JSON. Compared with XML, JSON is more compact and has less footprint. This means that it can be transferred faster in the internet. However, its semantics is less descriptive.

For the preparation work, we need to create get\_data.json in C:\Apache\htdocs and add the following content into this file:

```
[{"id": "5", "version": "5.5", "name": "Clash of Clans"},  
 {"id": "6", "version": "7.0", "name": "Boom Beach"},  
 {"id": "7", "version": "3.5", "name": "Clash Royale"}]
```

Now type in [http://127.0.0.1/get\\_data.json](http://127.0.0.1/get_data.json) should show the content as shown in Fig. 11.8.

After preparation for JSON data is done, let us see how to parse in Android app.



**Fig. 11.8** Verify JSON data in browser

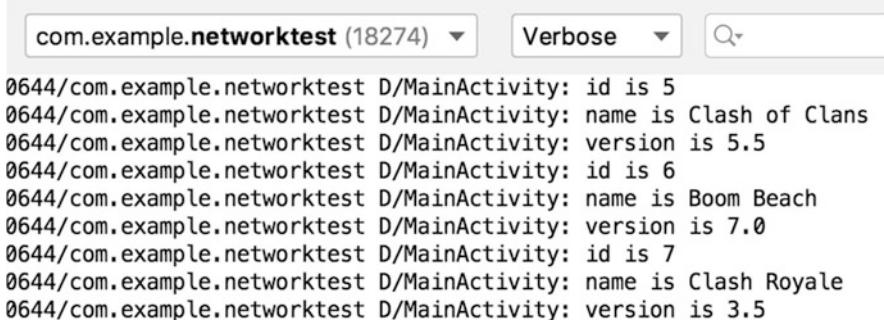
### 11.4.1 *JSONObject*

Similarly, we have plenty of choices to parse JSON data. We can use the official *JSONObject* or the open-source lib *GSON* from Google and some other open-source libs like *Jackson*, *FastJSON*, and so on. I will cover *JSONObject* and *GSON* in this book.

Edit *MainActivity* as shown below:

```
class MainActivity : AppCompatActivity() {
    ...
    private fun sendRequestWithOkHttp() {
        thread {
            try {
                val client = OkHttpClient()
                val request = Request.Builder()
                    // set address to be localhost
                    .url("http://10.0.2.2/get_data.json")
                    .build()
                val response = client.newCall(request).execute()
                val responseData = response.body?.string()
                if (responseData != null) {
                    parseJSONWithJSONObject(responseData)
                }
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }
    ...
    private fun parseJSONWithJSONObject(jsonData: String) {
        try {
            val jsonArray = JSONArray(jsonData)
            for (i in 0 until jsonArray.length()) {
                val jsonObject = jsonArray.getJSONObject(i)
                val id = jsonObject.getString("id")
                val name = jsonObject.getString("name")
                val version = jsonObject.getString("version")
                Log.d("MainActivity", "id is $id")
                Log.d("MainActivity", "name is $name")
                Log.d("MainActivity", "version is $version")
            }
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
```

First we want to change the HTTP request address to [http://10.0.2.2/get\\_data.json](http://10.0.2.2/get_data.json). Then use *parseJSONWITHJSONObject()* to parse the data. And this method is actually quite simple. The server is defined a JSON array, and here it gets passed into



The screenshot shows the Android Logcat interface. At the top, there are two dropdown menus: 'com.example.networktest (18274)' and 'Verbose'. To the right of these is a magnifying glass icon. The main area displays several log entries from the application's log:

```
0644/com.example.networktest D/MainActivity: id is 5  
0644/com.example.networktest D/MainActivity: name is Clash of Clans  
0644/com.example.networktest D/MainActivity: version is 5.5  
0644/com.example.networktest D/MainActivity: id is 6  
0644/com.example.networktest D/MainActivity: name is Boom Beach  
0644/com.example.networktest D/MainActivity: version is 7.0  
0644/com.example.networktest D/MainActivity: id is 7  
0644/com.example.networktest D/MainActivity: name is Clash Royale  
0644/com.example.networktest D/MainActivity: version is 3.5
```

**Fig. 11.9** Logcat records of parsed JSON data

a JSONArray object. Then we iterate through this JSONArray; each element is a JSONObject object, and each JSONObject object contains id, name, and version. Next, we just need to call getString() to get the text data and log them.

That is all we need to do! Run the app again and click “Send Request” button, and the result should be as shown in Fig. 11.9.

### 11.4.2 *JSON*

Using JSONObject to parse JSON is already quite simple, Google’s GSON will make this job effortless and let us take a look.

GSON is not in the official Android API which means that we need to add dependency to this lib in our project. Edit app/build.gradle by adding the following code in the dependencies closure:

```
dependencies {  
    ...  
    implementation 'com.google.code.gson:gson:2.8.5'  
}
```

The magic of GSON is that it can map JSON string to an object, and we do not need to manually parse at all.

For instance, for the following JSON format string:

```
{"name": "Tom", "age": 20}
```

We can define Person class with name and age fields, then we just need to call the following code to parse JSON to a Person object:

```
val gson = Gson()  
val person = gson.fromJson(jsonData, Person::class.java)
```

If JSON data is an array, then it is a little bit more complicated. For example, we can have the following data:

```
[{"name": "Tom", "age": 20}, {"name": "Jack", "age": 25},
 {"name": "Lily", "age": 22}]
```

Then we need TypeToken to pass the expected data type to fromJson() as shown below:

```
val typeOf = object : TypeToken<List<Person>>() {}.type
val people = gson.fromJson<List<Person>>(jsonData, typeOf)
```

That is it for the fundamental idea; now let us play with it. First create App class, add id, name, and version fields as shown below:

```
class App(val id: String, val name: String, val version: String)
```

Then edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {
    ...
    private fun sendRequestWithOkHttp() {
        thread {
            try {
                val client = OkHttpClient()
                val request = Request.Builder()
                    // set server address to localhost
                    .url("http://10.0.2.2/get_data.json")
                    .build()
                val response = client.newCall(request).execute()
                val responseData = response.body?.string()
                if (responseData != null) {
                    parseJSONwithGSON(responseData)
                }
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }
    ...
    private fun parseJSONwithGSON(jsonData: String) {
        val gson = Gson()
        val typeOf = object : TypeToken<List<App>>() {}.type
        val appList = gson.fromJson<List<App>>(jsonData, typeOf)
        for (app in appList) {
            Log.d("MainActivity", "id is ${app.id}")
            Log.d("MainActivity", "name is ${app.name}")
            Log.d("MainActivity", "version is ${app.version}")
        }
    }
}
```

```
    }  
}
```

Run the app again and click “Send Request” button, and Logcat should show the same records as shown in Fig. 11.9. That is all for XML and JSON parsing.

## 11.5 Implementing Network Callback

At this moment, you should know how to use HttpURLConnection and OkHttp to send HTTP request and parse the server response. However, our previous implementation is actually problematic. This is because most of the time, there will be more than one call site of network request which means that HTTP request can be reused, and if we write the HTTP request code in each place, it is apparently not a good idea.

Thus we should abstract the logic and put them in the same public class, and whenever we need to send network request, we just need to call the method inside the class. For example, we can have the following code:

```
object HttpUtil {  
  
    fun sendHttpRequest(address: String): String {  
        var connection: HttpURLConnection? = null  
        try {  
            val response = StringBuilder()  
            val url = URL(address)  
            connection = url.openConnection() as HttpURLConnection  
            connection.connectTimeout = 8000  
            connection.readTimeout = 8000  
            val input = connection.inputStream  
            val reader = BufferedReader(InputStreamReader(input))  
            reader.use {  
                reader.forEachLine {  
                    response.append(it)  
                }  
            }  
            return response.toString()  
        } catch (e: Exception) {  
            e.printStackTrace()  
            return e.message.toString()  
        } finally {  
            connection?.disconnect()  
        }  
    }  
}
```

Then when we need to send network request, we can have the following code:

```
val address = "https://www.baidu.com"
val response = HttpUtil.sendHttpRequest(address)
```

After getting the response, we can parse the response and process the data. However, it is worth noting that network request is time-consuming and there is no new worker thread created in sendHttpRequest(). This can block main thread when calling sendHttpRequest().

You might think that we can simply create a worker thread in sendHttpRequest() to solve this problem. But this actually does not work. Because by using a worker thread, sendHttpRequest() will finish before the time-consuming network request is done, which means the server response will not be handled.

To solve the problem, we need to use the callback mechanism. Let us take a look at how to use it.

First, we need to define an interface, for example, we can name it as HttpCallbackListener, as code below:

```
interface HttpCallbackListener {
    fun onFinish(response: String)
    fun onError(e: Exception)
}
```

We define two methods in the interface: onFinish() will be called when server successfully responds our request, and onError() will be called when there is network error. Both of these methods take param. The param of onFinish() is the response from server, and the param of onError() is error message.

Next edit HttpUtil as shown below:

```
object HttpUtil {

    fun sendHttpRequest(address: String, listener: HttpCallbackListener) {
        thread {
            var connection: HttpURLConnection? = null
            try {
                val response = StringBuilder()
                val url = URL(address)
                connection = url.openConnection() as HttpURLConnection
                connection.connectTimeout = 8000
                connection.readTimeout = 8000
                val input = connection.inputStream
                val reader = BufferedReader(InputStreamReader(input))
                reader.use {
                    reader.forEachLine {
                        response.append(it)
                    }
                }
            } catch (e: Exception) {
                listener.onError(e)
            }
        }
    }
}
```

```
        // callback onFinish()
        listener.onFinish(response.toString())
    } catch (e: Exception) {
        e.printStackTrace()
        // callback onError()
        listener.onError(e)
    } finally {
        connection?.disconnect()
    }
}
}
```

We add a param of `HttpCallbackListener` type and start a worker thread inside it to execute the network logic. Notice that worker thread cannot use return statement to return data; thus, we pass the response to `onFinish()` of `HttpCallbackListener`, and if there is exception, then pass the exception to `onError()`.

Now `sendHttpRequest` need to take a param of `HttpCallbackListener` too, as shown below:

```
HttpUtil.sendHttpRequest(address, object : HttpCallbackListener {
    override fun onFinish(response: String) {
        // handle response
    }

    override fun onError(e: Exception) {
        // handle exception
    }
})
```

Now, when server successfully responds, we can handle the response in `onFinish()`, and when there is exception we can handle the exception in `onError()`. This means that we pass the data to the caller with the help of callback.

Still using `HttpURLConnection` is complicated, and using `OkHttp` will simplify it. In `HttpUtil`, add `sendOkHttpRequest()` as shown below:

```
object HttpUtil {
    ...
    fun sendOkHttpRequest(address: String, callback: okhttp3.Callback) {
        val client = OkHttpClient()
        val request = Request.Builder()
            .url(address)
            .build()
        client.newCall(request).enqueue(callback)
    }
}
```

The okhttp3.Callback param in sendOkHttpRequest() is an interface in OkHttp lib that is similar to HttpCallbackListener. After client.newCall(), we do not call execute() as previously but instead call enqueue() and pass in the okhttp3.Callback argument. You have probably already figured out that inside enqueue(), a new worker thread will be started and the HTTP request will be executed in the worker thread and the result will be passed to okhttp3.Callback.

Now we can have the following code to use sendOKHttpRequest():

```
HttpUtil.sendOkHttpRequest(address, object : Callback {
    override fun onResponse(call: Call, response: Response) {
        // get the response
        val responseData = response.body?.string()
    }

    override fun onFailure(call: Call, e: IOException) {
        // handle exception
    }
})
```

As you can tell, the interface design of OkHttp is indeed succinct and easy to use. Notice that, whether it is HttpURLConnection or OkHttp, the callback will run in worker thread; thus we cannot update UI unless we switch the thread by runOnUiThread().

## 11.6 The Best Network Lib: Retrofit

When we talk about Android networks, we have to mention Retrofit because it is just so good. Retrofit is yet another network lib developed by Square which serves totally different purpose compared with OkHttp. OkHttp is more about lower level communication implementation, while Retrofit focus on higher level interface encapsulation. As a matter of fact, Retrofit is an application layer network communication lib based on OkHttp which can help us develop network-related functionality with OOD pattern. The project web address for Retrofit is <https://github.com/square/retrofit>.

Let start exploring this lib with creating RetrofitTest project.

### 11.6.1 Basic Use of Retrofit

First I want to talk about the fundamental ideas behind Retrofit's design. Its design is based on the following facts.

The network requests from the same app mostly go to the same server domain. This is easy to understand as any company's app should send requests to the company's server instead of querying all kinds of different servers.

Server's interfaces can be abstracted into different categories. For instance, adding new user, updating user data, and querying user data can be in one category, while adding new books, selling book, and querying available books can be in another category. Categorizing the interfaces properly can make the code structure more reasonable and improve readability and maintainability.

Lastly, developers are more used to the programming pattern that calling an interface and getting the value which is difficult to implement when calling the interface of the server. Most developers do not care about the communication details, while the traditional network libs require developers write significant amount of network-related code.

Retrofit's design put the above facts into consideration. First, we can configure a root address and use the relative path when specifying the server interface address without the need to provide the full URL.

Also, Retrofit allows us to categorize the server interfaces by defining the interfaces that are in the same category in the same interface file to make the code structure more reasonable.

Lastly, we also do not need to think about the details of network communication. We just need to declare the methods and return values in interface file and specify the corresponding server interface and params by annotation. Then when we call the method, Retrofit will send request to the corresponding server interface and parse the response to the type of return value. This makes the network operations more OOD.

Retrofit's design is mostly about the above ideas; let us use an example to explore how to use Retrofit.

To use Retrofit, we need to add the dependency in the project first. Edit app/build.gradle and add the following code in the dependencies closure:

```
dependencies {  
    ...  
    implementation 'com.squareup.retrofit2:retrofit:2.6.1'  
    implementation 'com.squareup.retrofit2:converter-gson:2.6.1'  
}
```

Since Retrofit is based on OkHttp, the first dependency will download Retrofit, OkHttp, and Okio together without specifically adding OkHttp dependency. Also Retrofit will automatically parse the JSON response to object; thus the second dependency is a Retrofit converting lib which uses GSON to parse JSON data and will download GSON lib together without specifically adding the GSON dependency. Besides GSON, Retrofit also supports other mainstream JSON parser libs such as Jackson, Moshi, etc., but without doubt, GSON is the most popular one.

We will continue using the JSON data interface in Sect. 11.4. Since Retrofit will use GSON to parse the JSON data to object, we need to add App class and add id, name, and version fields as code below:

```
class App(val id: String, val name: String, val version: String)
```

Next, we should create different interface files based on the server interface functionalities. However, since our Apache server has only one interface that is about getting JSON data, we just need to define one interface file with one method. Create AppService interface with code below:

```
interface AppService {
    @GET("get_data.json")
    fun getAppData(): Call<List<App>>
}
```

It is recommended to start the Retrofit interface file name with function category and end with Service.

There are two things in the above code that are worth noting. First is the annotation added above getAppData(). The @GET annotation means that when getAppData() is called, Retrofit will send a GET request, and the address is the param passed in the @GET annotation. Notice that we just need to pass in the relative path, and we will configure the root path later.

Second is that the type of return value of getAppData() has to be Call of Retrofit, and specifying the type of object that response should be converted to by using generics. Since the response here is JSON array of App type, we specify the generic type to be List<App>. Of course, Retrofit also provides Call Adapters to use the type of return values, for instance, by combining Retrofit and RxJava, we can define the return value to be type of Observable, Flowable, and so on, but this is beyond this book can cover.

After the definition of the AppService, let us put it into use. Add a button in activity\_main.xml as shown below:

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

    <Button
        android:id="@+id/getAppDataBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Get App Data" />

</LinearLayout>
```

The above code adds a button control, and we will handle network request logic in its click event code.

Edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

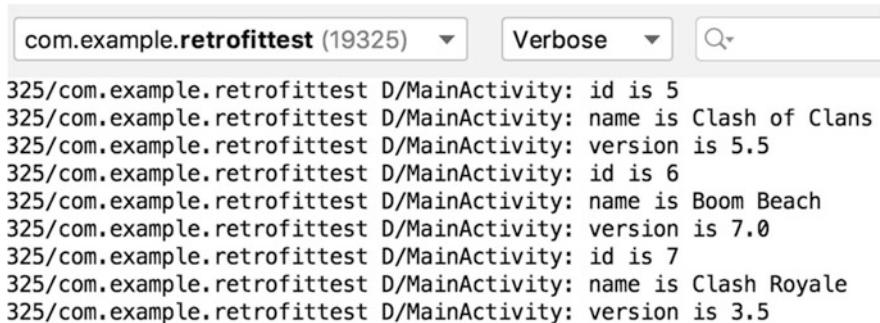
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        getAppDataBtn.setOnClickListener {
            val retrofit = Retrofit.Builder()
                .baseUrl("http://10.0.2.2/")
                .addConverterFactory(GsonConverterFactory.create())
                .build()
            val appService = retrofit.create(AppService::class.java)
            appService.getAppData().enqueue(object : Callback<List<App>> {
                override fun onResponse(call: Call<List<App>>, response: Response<List<App>>) {
                    val list = response.body()
                    if (list != null) {
                        for (app in list) {
                            Log.d("MainActivity", "id is ${app.id}")
                            Log.d("MainActivity", "name is ${app.name}")
                            Log.d("MainActivity", "version is ${app.version}")
                        }
                    }
                }

                override fun onFailure(call: Call<List<App>>, t: Throwable) {
                    t.printStackTrace()
                }
            })
        }
    }
}
```

In the “Get App Data” button click event code, an object of Retrofit gets created by using Retrofit.Builder, its baseUrl() is used to specify the root path of all the Retrofit requests, and addConverterFactory() is used to specify the converter lib when Retrofit parses the data, and here we set it to GsonConverterFactory. Notice that we have to call these two methods.

With Retrofit object, we can call its create() and pass in the corresponding Class type of the Service interface to create a delegate object. It’s OK if you are not familiar with delegate as long as you know that with delegate object, we can call the methods defined in interface, and Retrofit will call the implementation.

A Call<List<App>> object will be returned when getAppData() of AppService is called, and then after its enqueue() is called, Retrofit will send the network request based on the server address in the annotation, and the response will be passed to the Callback implementation. Notice that, when sending the request, Retrofit will automatically start worker thread, and when data is returned to Callback, Retrofit



The screenshot shows the Android Logcat interface. At the top, there are three dropdown menus: 'com.example.retrofittest (19325)', 'Verbose', and a search icon. Below these, a list of log entries is displayed:

```

325/com.example.retrofittest D/MainActivity: id is 5
325/com.example.retrofittest D/MainActivity: name is Clash of Clans
325/com.example.retrofittest D/MainActivity: version is 5.5
325/com.example.retrofittest D/MainActivity: id is 6
325/com.example.retrofittest D/MainActivity: name is Boom Beach
325/com.example.retrofittest D/MainActivity: version is 7.0
325/com.example.retrofittest D/MainActivity: id is 7
325/com.example.retrofittest D/MainActivity: name is Clash Royale
325/com.example.retrofittest D/MainActivity: version is 3.5

```

**Fig. 11.10** Data from Retrofit request and parse

will switch to main thread. During this process, we do not need to consider anything about thread switching. In `onResponse()` of `Callback`, `response.body()` will return the object after parsing by Retrofit which is of type `List<App>` here. Then we can iterate the list and log the data.

Since the server interface is still HTTP, we still need to configure with the steps introduced in Sect. 11.3.1. Copy `network_config.xml` from `NetworkTest` and paste to `RetrofitTest`. Then edit `AndroidManifest.xml` as follows:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/
    android"
        package="com.example.retrofittest">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme"
        android:networkSecurityConfig="@xml/network_config">
        ...
    </application>

</manifest>

```

The configuration here allows using plain text to send network request and declares network permission. Run `RetrofitTest` and click “Get App Data” button, and the logcat window should show result as shown in Fig. 11.10.

The parsed data shows that our code works as expected.

The above is the basic use of Retrofit. The example is simple but demonstrates the most commonly used and important part of Retrofit. After this we can take a look at some more detailed knowledge.

### 11.6.2 Process Complex Interface Address

In last section's example, we sent request to a simple server address: [http://10.0.2.2/get\\_data.json](http://10.0.2.2/get_data.json). However, in the real world, the interface addresses provided by the server are never so basic. If you take a look at the address bar in your browser, you should easily find that the addresses change a lot. In this section, we will discuss how to use Retrofit to adapt to changes.

For simplicity, first we define Data class which has id and content fields as shown below:

```
class Data(val id: String, val content: String)
```

Then for a simple request, we can have something like the following:

```
GET http://example.com/get_data.json
```

The address here is static, and with Retrofit, we can have the following code:

```
interface ExampleService {  
  
    @GET("get_data.json")  
    fun getData(): Call<Data>  
  
}
```

We covered this in last section and should be easy to understand now.

Apparently, server cannot always provide us static address, and most of the time, certain parts of the address will change, for instance, for the following address:

```
GET http://example.com/<page>/get_data.json
```

In this interface, based on <page> value, server will return different response. How to implement this in Retrofit? It is quite easy, as shown in the code below:

```
interface ExampleService {  
  
    @GET("{page}/get_data.json")  
    fun getData(@Path("page") page: Int): Call<Data>  
  
}
```

In the address specified by @GET, we use {page} as placeholder and add page param in getData(), and then use @Path("page") to declare this param. Then when we call getData() to send request, Retrofit will replace the placeholder with the value of page param to generate a legit address.

Some servers may require us to pass in a series of params, and the following is an example:

```
GET http://example.com/get_data.json?u=<user>&t=<token>
```

This is the standard format of GET request with params. The params will follow after the question mark, and each param will have key value with equal sign between them, and “&” is used to divide the params. The above address requires us to pass in the value of user and token. We can use @Path to resolve the issue though a little bit complicated. Retrofit provides support for this kind of scenario to make it easier, as shown below:

```
interface ExampleService {  
  
    @GET("get_data.json")  
    fun getData(@Query("u") user: String, @Query("t") token: String):  
        Call<Data>  
  
}
```

The above code adds user param and token param in getData() and uses @Query to annotate them. Then Retrofit will automatically add these two params in the request.

Now you should be able to handle all kinds of server interface addresses. However, HTTP has more than just GET request, while other requests like POST, PUT, PATCH, and DELETE are also commonly used. They serve distinct purposes. GET request is used to get data from server, POST request is used to submit data to server, PUT and PATCH are used to update the data in server, and DELETE request is used to delete server data.

Retrofit supports all of these types of request by annotation with @GET, @POST, @PUT, @PATCH, and @DELETE.

For instance, assume server provides the following interface address:

```
DELETE http://example.com/data/<id>
```

This kind of interface address usually means to delete data based on id, and using Retrofit we can have the following implementation:

```
interface ExampleService {  
  
    @DELETE("data/{id}")  
    fun deleteData(@Path("id") id: String): Call<ResponseBody>  
  
}
```

The above code uses `@DELETE` annotation to send DELETE request and uses `@PATH` annotation to specify the id. Then for return value declaration, the generic type of Call has been set to `ResponseBody`, what does it mean?

This is because POST, PUT, PATCH, and DELETE do not get server data as POST does; thus usually they do not need to parse the server response. Then we can use `ResponseBody` to allow Retrofit to take any type of response and do not parse the response.

What if we need to post data to server? For example, for the following interface address and data:

```
POST http://example.com/data/create
{"id": 1, "content": "The description for this data."}
```

If we use POST request, we should put the data in the body part of HTTP request which can be done through annotation as shown below:

```
interface ExampleService {
    @POST("data/create")
    fun createData (@Body data: Data) : Call<ResponseBody>
}
```

The above code declares a param of `Data` type in `createData()` and adds `@Body` annotation. Then when Retrofit sends POST request, it will convert the data in `Data` object to JSON format text and put the text into the body part of HTTP request which can be parsed by the server. The same format can be applied to PUT, PATCH, and DELETE requests.

Some server interface may require set param in the HTTP request header, for instance:

```
GET http://example.com/get_data.json
User-Agent: okhttp
Cache-Control: max-age=0
```

These header params are key-value pairs, and we can use `@Headers` annotation to declare them as shown below:

```
interface ExampleService {
    @Headers("User-Agent: okhttp", "Cache-Control: max-age=0")
    @GET("get_data.json")
    fun getData() : Call<Data>
}
```

This can only declare static header, and for dynamic header, we need to use @Header annotation as shown below:

```
interface ExampleService {

    @GET("get_data.json")
    fun getData(@Header("User-Agent") userAgent: String,
    @Header("Cache-Control") cacheControl: String): Call<Data>

}
```

Then when HTTP request is sent, Retrofit will automatically pass the value from params to User-Agent and Cache-Control to dynamically set the value of header.

That is all for using Retrofit to handle server interface addresses.

### **11.6.3 Best Practice for Retrofit Builder**

There is a problem for our practice to get the dynamic delegate of Service interface. Up to now, we follow the pattern as shown below:

```
val retrofit = Retrofit.Builder()
    .baseUrl("http://10.0.2.2/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
val appService = retrofit.create(AppService::class.java)
```

To get the delegate object of AppService, we need to use Retrofit.Builder to construct a Retrofit object and then call its create() to create delegate object. It is fine for one-time use, but not feasible if we need to repeat this for sending to all kinds of interfaces.

As a matter of fact, there is no need to repeat this process since the Retrofit object is globally usable, and we just need to pass in the corresponding Class for different Service interface when calling create(). Thus we can encapsulate this part of code to simplify getting Service interface delegate object.

Create a singleton of ServiceCreator as code below:

```
object ServiceCreator {

    private const val BASE_URL = "http://10.0.2.2/"

    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
```

```
    fun <T> create(serviceClass: Class<T>) : T = retrofit.create  
(serviceClass)  
}
```

The object keyword makes ServiceCreator a singleton class. In this class we define BASE\_URL constant to set the root path of Retrofit and use Retrofit.Builder to create an object of Retrofit. Notice that these fields are all private and invisible to other classes.

The public create() will take a param of Class type, and when it gets called, create () of Retrofit object will be called to create the delegate object of the corresponding Service interface.

Now it will be much easier to use Retrofit, for instance, to get a delegate object of AppService, and we can use the following code:

```
val appService = ServiceCreator.create(AppService::class.java)
```

Then we can call any method in AppService interface.

But there is still room to optimize. Apply the reified type parameter knowledge from previous chapter and edit ServiceCreator as code below:

```
object ServiceCreator {  
    ...  
    inline fun <reified T> create() : T = create(T::class.java)  
}
```

We define create() without param and use inline keyword to declare the method and use reified keyword for generic type which are required for reified type parameter. Then we use syntax like T::class.java and call the create() method that has Class type param.

Now we have a new way to get the delegate object of AppService, as shown below:

```
val appService = ServiceCreator.create<AppService>()
```

Now we have more succinct code with reified type parameter.

That is end of Retrofit topics, we will cover how to use Retrofit in real-world project in Chap. 15. Next let us start the Kotlin Class of this chapter, and we will discuss coroutine in Kotlin.

## 11.7 Kotlin Class: Use Coroutine for Performant Concurrent App

Coroutine is a special concept in Kotlin which does not exist in most of the programming languages.

Coroutine is similar to thread, and we can treat it as a lightweight thread. Thread is heavy and needs OS management to switch between different threads. By using coroutine, we can switch between different coroutines at language level and improve the performance of concurrency.

For instance, assume we have foo() and bar() as shown below:

```
fun foo() {  
    print(1)  
    print(2)  
    print(3)  
}  
  
fun bar() {  
    print(4)  
    print(5)  
    print(6)  
}
```

If we do not start a new thread and call foo() and bar() sequentially, then the output result will always be 123456. If we use coroutine and call foo() in coroutine A and call bar() in coroutine B, although these two methods still in the same thread, the result of running foo() and bar() is not deterministic.

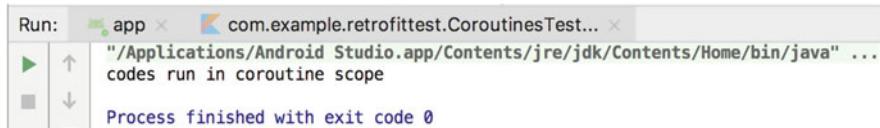
Coroutine allows us to write code that functions like multithreading but actually runs in a single thread. The suspending and resuming of the code is determined by the programming language and is independent of operating system. This greatly improved performance of concurrency code. For instance, it is impossible to start 100k threads, but it is possible to start 100k coroutines. We will verify this later.

With these basic concepts, let us take a look at how to use coroutine in Kotlin.

### 11.7.1 Basic Use of Coroutine

Coroutine is not in the Kotlin standard API but is provided as dependency lib. Thus we need to add the following dependency in app/build.gradle to use coroutine:

```
dependencies {  
    ...  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.1.1"  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
```



**Fig. 11.11** Print string in coroutine

```
android:1.1.1"
}
```

The second dependency is for Android project. In this section, we will write Kotlin code that is platform independent which does not require the second lib, but I do not want to discuss dependency the next time we use coroutine in Android project and thus add it here.

Next create CoroutinesTest.kt and define main().

The first question is how to start a coroutine? The simplest way is to use GlobalScope.launch function as code below:

```
fun main() {
    GlobalScope.launch {
        println("codes run in coroutine scope")
    }
}
```

GlobalScope.launch function can create a scope for coroutine where the code block or Lambda expression passed to launch function can run, and here we just print a string. If you run the main function, you will find that the string is not printed.

This is because GlobalScope.launch function creates a top-level coroutine which will end when the program ends. The reason why the string is not printed is because before the code in the code block gets executed, program already ended.

We can simply delay ending the program to solve the problem, as shown below:

```
fun main() {
    GlobalScope.launch {
        println("codes run in coroutine scope")
    }
    Thread.sleep(1000)
}
```

Thread.sleep() will block the main thread for 1 s; run the program again, and you should see the string gets printed as shown in Fig. 11.11.

However, this solution is still problematic because it will be suspended if the code block cannot finish running within a second. Observe the following code:

```
fun main() {
    GlobalScope.launch {
        println("codes run in coroutine scope")
```



**Fig. 11.12** Result of runBlocking

```

        delay(1500)
        println("codes run in coroutine scope finished")
    }
    Thread.sleep(1000)
}

```

We add delay() function and print a string in the console. delay() can delay the current coroutine for the time set in this function. This is different from Thread.sleep() as delay() is non-blocking and only suspends the current coroutine without affecting other coroutines. Thread.sleep() will block the current thread which will suspend all the coroutines running in this thread. Notice that delay() can only be called in the CoroutineScope or other suspending functions.

Here we suspend coroutine for 1.5 s and block the main thread for 1 s. Run the app again, and you will find that in the console the newly added string is not printed. This is because before println() gets called, app already finishes running.

Is there any way to keep the app running until the code in coroutine finishes running? We can use runBlocking function to solve this problem as shown in code below:

```

fun main() {
    runBlocking {
        println("codes run in coroutine scope")
        delay(1500)
        println("codes run in coroutine scope finished")
    }
}

```

runBlocking function creates CoroutineScope and can ensure that the current thread will be blocked until the code and child coroutines finish running. Notice that this function should only be used in test environment because it can cause performance issue in product environment.

Run the app again, and the result should be as shown in Fig. 11.12.

It shows that the two println statements got executed.

Although we can run code in coroutine now, the examples above do not show any benefits of using coroutine. This is because our code is running in the same coroutine, while the advantage of using coroutine instead of thread is in high concurrency use case.

To create multiple coroutines, we can use launch function as shown in code below:



**Fig. 11.13** Result of running multiple coroutines

```
fun main() {
    runBlocking {
        launch {
            println("launch1")
            delay(1000)
            println("launch1 finished")
        }
        launch {
            println("launch2")
            delay(1000)
            println("launch2 finished")
        }
    }
}
```

Notice that the `launch` function used above is different from `GlobalScope.launch`. It can only be called in the `CoroutineScope` and will create child coroutine in the current `CoroutineScope`. The child coroutine will end if the coroutine of the outer layer `CoroutineScope` has ended. On the other hand, `GlobalScope.launch` always creates the top-level coroutine.

The above code calls `launch` function twice and creates two child coroutines. Run the app again, and the result is as shown in Fig. 11.13.

The `println` statements did not execute in sequential order, but these two coroutines are actually running in the same thread, and the running order is determined by Kotlin instead of the OS which makes the concurrency efficiency exceptionally high.

We can use experiment to demonstrate how efficient it is with the following code:

```
fun main() {
    val start = System.currentTimeMillis()
    runBlocking {
        repeat(100000) {
            launch {
                println(".")
            }
        }
    }
    val end = System.currentTimeMillis()
    println(end - start)
}
```



**Fig. 11.14** Concurrency performance of 100k coroutines

With repeat function, we create 100k coroutines, and in each coroutine we simply print a dot, and at the end print the time to run all this. Run the app again, and the result should be as shown in Fig. 11.14.

It only takes 961 ms to execute, and if we try to start 100k threads, we will see OOM exception.

As the logic in the launch function becomes more and more complicated, we need to abstract some of the logic into a function. Then there is an issue that the code in the launch function has coroutine scope, but a function does not have coroutine scope, and how can we call suspending function like delay().

The suspend modifier can make any function to be suspending function, and suspending function can call each other as shown in the code below:

```
suspend fun printDot() {
    println(".")
    delay(1000)
}
```

With suspend modifier, we can call delay() in printDot().

However, the suspend modifier can only declare a function as suspending function and cannot provide coroutine scope. Thus you cannot call launch function in printDot() as launch function is only available in coroutine scope.

The coroutineScope function can solve this problem. The coroutineScope function is also a suspending function and can be called from any other suspending function. It will inherit the coroutine scope where it is called and creates its own child coroutine scope; thus we can provide coroutine scope to any suspending function with its help. Here is an example:

```
Run: app com.example.retrofittest.CoroutinesTest...
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
1
2
3
4
5
6
7
8
9
10
coroutineScope finished
runBlocking finished

Process finished with exit code 0
```

**Fig. 11.15** Result of coroutineScope function

```
suspend fun printDot() = coroutineScope {
    launch {
        println(".")
        delay(1000)
    }
}
```

With coroutineScope, we can call launch function in printDot().

Also coroutineScope is similar to runBlocking in a sense that it will block the current coroutine before all the code and coroutines in its coroutine scope finish running. Here is an example:

```
fun main() {
    runBlocking {
        coroutineScope {
            launch {
                for (i in 1..10) {
                    println(i)
                    delay(1000)
                }
            }
            println("coroutineScope finished")
        }
        println("runBlocking finished")
    }
}
```

The runBlocking function creates a coroutine scope, and the coroutineScope function creates its own coroutine scope in which we create a child coroutine with launch and print 1–10 every second. Then at the end of runBlocking and coroutineScope function, we print a string. Run the app again, and the result should be as shown in Fig. 11.15.

You shall see that the console will print number from 1 to 10 every second and then print the string at the end of the coroutineScope function and in the end print the string at the end of runBlocking function.

This indicates that coroutineScope function indeed blocks the current coroutine and only after its own code and sub coroutines finish running can the code after coroutineScope function gets executed.

The coroutineScope function is similar to runBlocking, but coroutineScope will only block the current coroutine and will not block other coroutine or any thread; thus there is no performance issue. On the other hand, runBlocking will block the current thread, and if you call it in the main thread, it will freeze the UI and thus not recommended in real application.

These are the basics of coroutines and let us take a look at more advanced topics for coroutine.

### ***11.7.2 More on Coroutine Scope Builder***

In the last section, we discussed a few coroutine scope builders such as GlobalScope.launch, runBlocking, launch, and coroutineScope which can create a new coroutine scope. GlobalScope.launch and runBlocking can be called anywhere, while coroutine Scope can be called in coroutine scope or suspending function, and launch function can only be called inside coroutine scope.

As mentioned previously, runBlocking will block thread, and thus it is only recommended to use it in test environment. GlobalScope.launch will create top-level coroutine scope, and thus it is also not recommended unless you specifically want to create top-level coroutine scope.

Why top-level coroutine is not recommended? This is due to its management cost. For instance, if we send network request using coroutine in an Activity, since network request is time-consuming and if user closes this Activity before server responds, then we should cancel the coroutine or ignore this network request response as Activity has been destroyed, and there is no reason to execute the callback.

How to cancel the coroutine? Both GlobalScope.launch function and launch function return an instance of Job and calling cancel() method of Job will cancel the coroutine, as shown in the code below:

```
val job = GlobalScope.launch {  
    // business logic  
}  
job.cancel()
```

However, if we create top-level coroutine all the time, then when Activity is destroyed, we need to call cancel() of all the corresponding jobs. This will make code not maintainable.

That is the reason why GlobalScope.launch is rarely used in real application. The code below demonstrates a commonly used pattern:

```
val job = Job()
val scope = CoroutineScope(job)
scope.launch {
    // business logic
}
job.cancel()
```

We create an instance of Job and pass it to CoroutineScope(). Notice that CoroutineScope() is a function though its naming looks like a class. CoroutineScope() will return an object of CoroutineScope. The syntax looks like we create an instance of CoroutineScope which may be the intentionally designed in Kotlin. With an instance of CoroutineScope, we can call its launch function to create a coroutine.

Now all the coroutines created by calling launch function of CoroutineScope will be in the scope of Job instance. We just need to call cancel() once to cancel all the coroutines in the same coroutine scope which greatly reduced coroutine management cost.

CoroutineScope() function should be used in real application, and if you just want to write some test code, runBlocking is much easier.

The launch function can create a new coroutine to execute some logic, but it cannot get the result because it always return a Job object. To get the result of the code block in the coroutine, we can use async function.

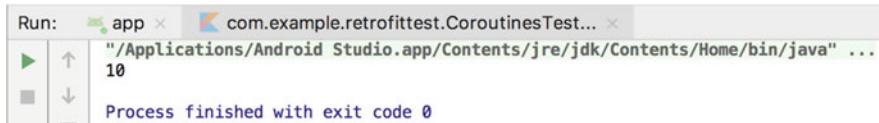
The async function can only be used in the coroutine scope. It will create a new coroutine and return a Deferred object. To get the result of the code block, we just need to call await() on the Deferred object, as shown in code below:

```
fun main() {
    runBlocking {
        val result = async {
            5 + 5
        }.await()
        println(result)
    }
}
```

The code block in the async function executes some simple arithmetic operation and use await() to get the result, and then the result will be printed in the console. Run the code again, and the result should be as shown in Fig. 11.16.

After calling async function, the code inside the code block will be executed immediately. When await() is called, if the code block has not finished running, await() will block the current coroutine until the result of async function is available.

We can use the following code to validate this:



**Fig. 11.16** Print result of async function



**Fig. 11.17** Result of async functions running sequentially

```
fun main() {
    runBlocking {
        val start = System.currentTimeMillis()
        val result1 = async {
            delay(1000)
            5 + 5
        }.await()
        val result2 = async {
            delay(1000)
            4 + 6
        }.await()
        println("result is ${result1 + result2} .")
        val end = System.currentTimeMillis()
        println("cost ${end - start} ms.")
    }
}
```

The code above has two async functions and has 1 s delay in the code block. Based on the theory above, await() will block the coroutine until code block inside async function finishes running. To prove this, we record the time stamp. Run the program again, and the result should be as shown in Fig. 11.17.

The whole process takes 2032 ms which means that these two async functions run sequentially.

This is apparently inefficient, and we can run these two async functions simultaneously to improve efficiency. Edit the code as shown below:

```
fun main() {
    runBlocking {
        val start = System.currentTimeMillis()
        val deferred1 = async {
            delay(1000)
            5 + 5
        }
        val deferred2 = async {
```



```
Run: app com.example.retrofittest.CoroutinesTest...
▶ ↑ "/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
result is 20.
cost 1029 milliseconds.

Process finished with exit code 0
```

**Fig. 11.18** Result of async functions running simultaneously

```
    delay(1000)
    4 + 6
}
println("result is ${deferred1.await() + deferred2.await()}.")
val end = System.currentTimeMillis()
println("cost ${end - start} milliseconds.")
}
}
```

Instead of using await() to get result immediately, this time we only call await() when we need the result, and the two async functions will run parallelly. Run the program again, and the result should be as shown in Fig. 11.18.

The running time becomes 1029 ms which is quite significant improvement.

Lastly, let us take a look at a special coroutine scope builder: withContext() function. This function is a suspending function, and we can treat it as a simplified version of async function. We can use it as the following example:

```
fun main() {
    runBlocking {
        val result = withContext(Dispatchers.Default) {
            5 + 5
        }
        println(result)
    }
}
```

After calling withContext(), the code block will be executed immediately, and the current coroutine will be blocked. When the code block finishes running, the result of the last line will be returned as return value of withContext() which is equivalent to val result = async{5 + 5}.await(). The only difference is that withContext() requires a thread param which I will cover in detail.

Coroutine is a lightweight concurrency concept which is like thread. Thus, we can replace the conventional thread with coroutine. But it does not mean that thread is useless. For instance, in Android, network request has to run in worker thread even if you start coroutine to send network request. If the coroutine is in the main thread, there will be exception. We can set the thread in which the coroutine should run with thread param.

There are mainly three options to choose from for this param: Dispatchers.Default, Dispatchers.IO, Dispatchers.Main. Dispatchers.Default is a standard low

concurrent thread model. When the code is computation extensive, over concurrency may actually slow down the task, and you can just use Dispatchers.Default. Dispatchers.IO is a high concurrent thread model which is good for situations that the task spends most of the time blocked and waiting, for instance, when sending network requests, to get higher concurrency, we should use Dispatchers.IO. Dispatchers.Main means that no worker thread will be started, and code will run in main thread which can only be used in Android project and is not available in pure Kotlin program.

For all the coroutine scope builders, all of the functions can set this param except coroutineScope function. Only withContext() makes this param mandatory.

Kotlin does not only improve the performance of concurrent code but also can optimize the conventional callback code to make it more succinct.

### **11.7.3 Simplifying Callback with Coroutine**

In Sect. 11.5 we discussed the callback mechanism and applied it to implement getting network request response asynchronously. The callback mechanism relies on anonymous class, while anonymous class is usually very lengthy as shown below:

```
HttpUtil.sendHttpRequest(address, object : HttpCallbackListener {
    override fun onFinish(response: String) {
        // process the response
    }

    override fun onError(e: Exception) {
        // handle exception
    }
})
```

Any place the network request is sent, such code pattern will be repeated. Is there any simpler implementation for this?

Kotlin coroutine makes it possible by using suspendCoroutine and let us take a look.

The suspendCoroutine function takes a param of Lambda expression which is only callable in the coroutine scope and can suspend the current coroutine and run the code in the Lambda expression in a normal thread. The param lust of Lambda expression will take a param of Continuation type, and its resume() or resumeWithException() can resume the coroutine.

Now let us see how to optimize the callback with suspendCoroutine. First define request() as code below:

```
suspend fun request(address: String): String {
    return suspendCoroutine { continuation ->
        HttpUtil.sendHttpRequest(address, object : HttpCallbackListener
```

```

    {
        override fun onFinish(response: String) {
            continuation.resume(response)
        }

        override fun onError(e: Exception) {
            continuation.resumeWithException(e)
        }
    })
}
}

```

`request()` is a suspending function which takes address param. Inside `request()`, `suspendCoroutine` is called which will suspend the current coroutine immediately, and the Lambda expression will run in worker thread. Then we call `Http.sendHttpRequest()` to send the network request and use the conventional callback pattern to listen to response. If request is successful, then we call `resume()` of `Continuation` to resume the suspended coroutine and pass in the response which will be the return value of `suspendCoroutine` function. If request fails, the `resumeWithException()` will resume the coroutine and pass in the exception.

The reason why the code above optimize callback while still using the conventional pattern is because there is no repeated callback implementation any more with new callsites. For instance, to get the response from Google.com we can have the following code:

```

suspend fun getGoogleResponse() {
    try {
        val response = request("https://www.google.com/")
        // process response
    } catch (e: Exception) {
        // handle exception
    }
}

```

This makes the call site much more succinct. Since `getGoogleResponse()` is a suspending function, when it calls `request()`, the current coroutine will be suspended immediately until the network request fails or succeeds. Thus even without the callback code, we can still get the response asynchronously, and even the `request fails, catch` will be called.

You might think that `getGoogleResponse()` is declared as suspending function which limits its use in coroutine scope or other suspending functions. Indeed, this is the case because `suspendCoroutine` function needs to be used with coroutines. However, with proper and thoughtful system architecture, we can easily apply the coroutine code in real application, and you will learn how to do so in Chap. 15's practice section.

In fact, `suspendCoroutine` function can optimize almost any callback, for instance, we used to use Retrofit to send network request with the following code:

```

val appService = ServiceCreator.create<AppService>()
appService.getAppData().enqueue(object : Callback<List<App>> {
    override fun onResponse(call: Call<List<App>>, response:
Response<List<App>>) {
        // process response
    }

    override fun onFailure(call: Call<List<App>>, t: Throwable) {
        // handle exception
    }
})

```

With suspendCoroutine function, we can optimize the code dramatically.

As different Service interfaces will return different types of data, we need to apply generics here. Define await() as shown in code below:

```

suspend fun <T> Call<T>.await(): T {
    return suspendCoroutine { continuation ->
        enqueue(object : Callback<T> {
            override fun onResponse(call: Call<T>, response: Response<T>) {
                val body = response.body()
                if (body != null) continuation.resume(body)
                else continuation.resumeWithException(
                    RuntimeException("response body is null"))
            }

            override fun onFailure(call: Call<T>, t: Throwable) {
                continuation.resumeWithException(t)
            }
        })
    }
}

```

This is more complicated than the request() function. First, await() is still a suspending function of generic type T, and we define it as an extension function of Call<T>; thus any Retrofit network request interface that returns Call type can call await().

Inside await(), suspendCoroutine function is used to suspend the current coroutine, and because await() is an extension function, we now has the context of Call object and can call enqueue() to let Retrofit send network request. Next, we can apply the same pattern to handle the response or exception. Notice that when onResponse() is called, when call body() to parse the response which can be null. If response is null, the above code will throw an exception, and you can make changes based on your needs.

With await(), it becomes super easy to call Retrofit's Service interface, and we can use the following code to implement the same functionality for the previous example.

```
suspend fun getAppData() {  
    try {  
        val appList = ServiceCreator.create<AppService>().getAppData()  
    await()  
        // process response  
    } catch (e: Exception) {  
        // handle exception  
    }  
}
```

There is no lengthy anonymous class implementation anymore. We just need to call await() to let Retrofit send network request and get the response directly. We can also omit the try and catch block here and have the try and catch happen at a unified interface function to make the code even more succinct.

We covered plenty of coroutine topics, and we will apply them in real application in Chap. 15. Let us summarize what we have learnt in this chapter.

## 11.8 Summary and Comment

In this section, we discussed how to use HTTP to interact with network in Android. Although Android supports many more network communication protocols, without any doubt, HTTP is the most commonly used one. We mainly use HttpURLConnection and OkHttp to send HTTP request.

Then we discussed how to parse XML and JSON data because most of the server responses are using these two formats. The parsing methods mentioned in this chapter should be sufficient for you to handle real application data parsing requirement.

Then we discussed the callback mechanism and how to use Retrofit. Retrofit is the most popular network lib for Android, and you should be proficient in using this lib.

In this chapter's Kotlin Class section, we discussed a Kotlin specific concept: coroutine which may sound strange to you before. But now you should understand this comprehensively.

That is everything I want to discuss for Android network programming, and in next chapter, we will take a look at how to make UI look beautiful with Material Design.

# Chapter 12

## Best UI Experience: Material Design in Action



For a long time, Android UI was widely seen as not beautiful or at least not as good as iOS UI. This makes some companies enforce unified UI for Android and iOS which does not make sense to me. Because for the same user, it is unlikely that they will use the same app on different OS but surely that they use different apps in the same OS. Thus having same style across different apps in the same OS is more important than unifying the style of the same app in different OS.

However, Android standard UI design style was not happily accepted by majority of people, and many companies thought they could design much better UI which caused the style of Android apps could not be unified for quite a long time. To solve this problem, Google released a brand-new design language—Material Design in Google I/O 2014.

In this chapter, we will deep dive into Material Design.

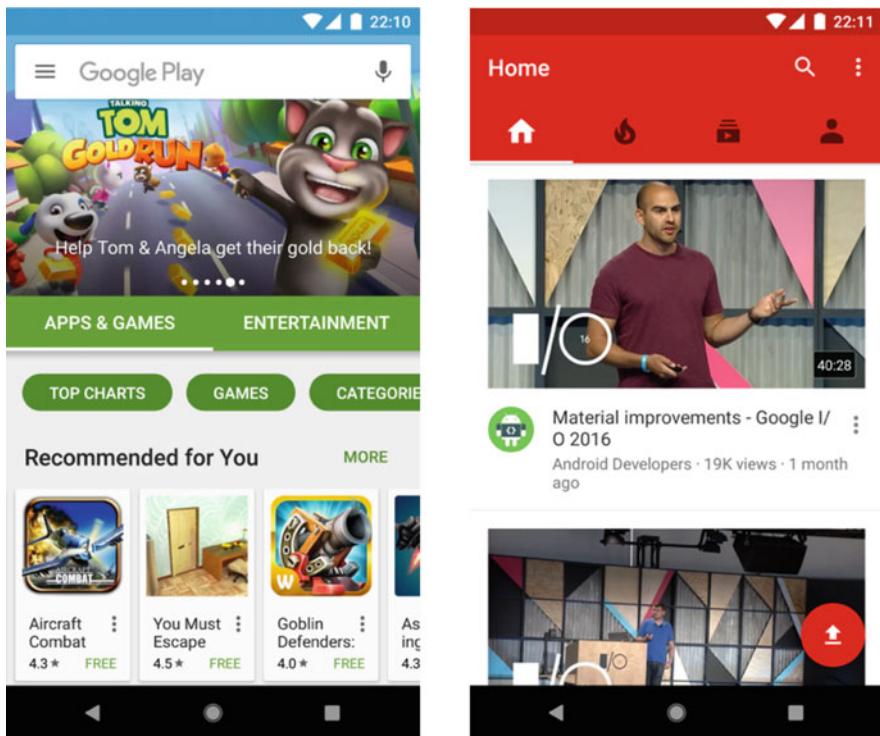
### 12.1 What Is Material Design?

Material Design is a design language created by Google design engineers based on excellent design principles with creativity and scientific theory. Google was very confident that Material Design will unify the style of Android apps because it looks really good!

To set an example, since Android 5.0, Google made all the system apps with Material Design. Figure 12.1 shows the two examples.

The left side app is Play Store, and the right side app is YouTube, and you can see that they indeed have beautiful UI.

However, after release, Material Design did not get traction immediately. This is because it is just a design guideline whose audiences are mainly designers instead of developers. Many developers might not even know what is Material Design UI, and even they understood this concept, it was difficult to implement as there was little API support.



**Fig. 12.1** Material Design style system app examples

Google realized this problem and released Design Support lib in Google I/O 2015. This lib encapsulated some of the most iconic widgets and effects in Material Design so that developers can build Material Design UI without knowing Material Design. Later, Design Support lib was renamed to Material lib to provide Material Design support to all Google products. In this chapter we will dive into Material lib and work on a Material Design project with some widgets from AndroidX lib.

Create MaterialTest project and let us begin!

## 12.2 Toolbar

Toolbar is the first widget we will discuss which is provided in AndroidX. Toolbar may sound strange, but you should be familiar with another correlated widget ActionBar.

In Sect. 4.4.1, we used to hide the native ActionBar so that we can use customized title bar. The top title bar at every Activity is ActionBar which appears in all of our examples.

ActionBar is limited to be positioned at the top of Activity and thus cannot render some Material Design effect and is not officially recommended. I will skip ActionBar and discuss the recommended Toolbar.

Toolbar inherits every functionality of ActionBar while being more flexible and can be used together with some other widgets to create Material Design effect. And now, let us take a look.

ActionBar will be used by default for a new project, and this is coming from the theme specified in the project. Open AndroidManifest.xml, and you should see the following code:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
</application>
```

The android:theme is set to AppTheme which is defined in res/values/styles.xml as shown below:

```
<resources>

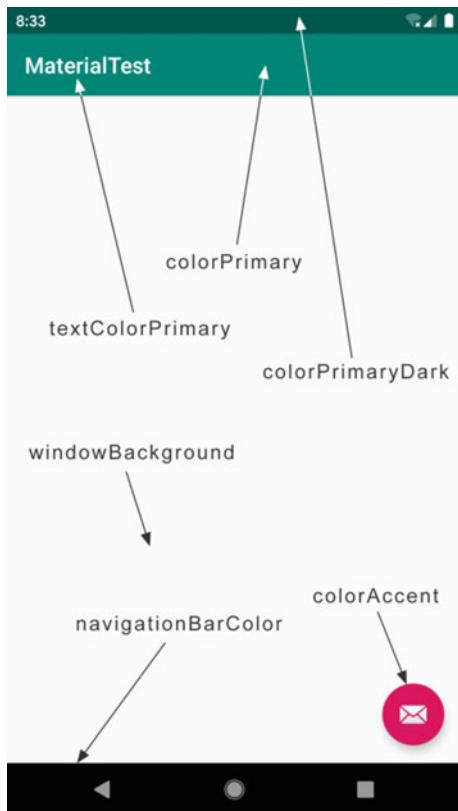
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.
DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

The AppTheme has parent theme Theme.AppCompat.Light.DarkActionBar. This is the reason why we have ActionBar in all of our projects.

To replace the ActionBar with Toolbar, we need to set a theme that is without ActionBar. Usually we choose from Theme.AppCompat.NoActionBar and Theme.AppCompat.Light.NoActionBar. I will continue to use the Light theme here as shown below:

**Fig. 12.2** How Colors work



```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

There are three colors you want to override: `colorPrimary`, `colorPrimaryDark`, and `colorAccent`. It is hard to explain with words; let me use Fig. 12.2 to illustrate.

You can tell how different colors affect different components from this image easily.

Besides these three color properties, we can also control more components' color with `textColorPrimary`, `windowBackground`, and `navigationBarColor`, etc.

colorAccent is known as secondary color and is applied to buttons, progress bar, and for highlighting something like the selected state of some widgets.

Now ActionBar is hidden; to use Toolbar to replace ActionBar, edit activity\_main.xml as code below:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="@color/colorPrimary"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

</FrameLayout>
```

In the above code, xmlns:app set a name space. The name space set by xmlns: android allows us to use android:id, android:layout\_width, etc. The xmlns:app allows us to use app:attribute and so on. The reason why we need to add xmlns: app is because a lot of the Material properties are added in newer versions of OS and does not exist in older versions. To make it compatible with older versions, we have to use app:attribute instead of android:attribute.

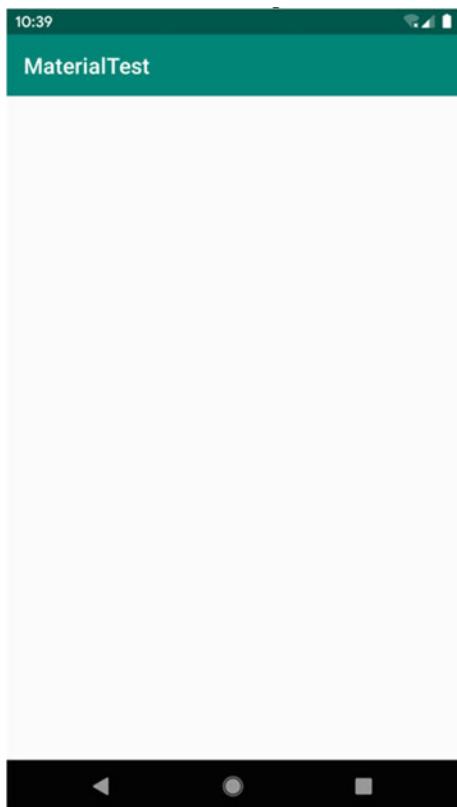
Then we define a Toolbar control which is provided by appcompat lib. We set the id, width to match\_parent, height to the height of actionBar, background color to colorPrimary. Since we set the theme of the app to Light, Toolbar will also have Light theme and the elements in Toolbar will have deeper color to distinguish from the background. But the text color change will not look good and to continue allow Toolbar to use dark theme we can set android:theme to ThemeOverlay.AppCompat. Dark.ActionBar to use dark theme Toolbar. But this introduces another problem which is the menu button (Check Sect. 3.2.5) in the Toolbar because it will have dark theme and does not look consistent. Thus we set the pop-up menu to light theme with app:popupTheme property.

If you feel puzzled, try to mix use the themes, and you will find out how it actually works.

Next, edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)
```

**Fig. 12.3** Standard toolbar

```
    }  
}
```

The highlighted code pass an instance of Toolbar to setSupportActionBar(), and this allows us to use Toolbar and make it looks and functions as ActionBar.

Run the app, and the result should be as shown in Fig. 12.3.

This looks exactly as the ActionBar we used before, but under the hood it is actually a Toolbar. Now it can render Material Design effect which we will show shortly. To edit the text in the title bar, we can edit AndroidManifest.xml as code below:

```
<application  
    android:allowBackup="true"  
    android:icon="@mipmap/ic_launcher"  
    android:label="@string/app_name"  
    android:roundIcon="@mipmap/ic_launcher_round"  
    android:supportsRtl="true"  
    android:theme="@style/AppTheme">
```

```

<activity
    android:name=".MainActivity"
    android:label="Fruits">
    ...
</activity>
</application>

```

The android:label attribute sets the content of the Toolbar. If this attribute is not set, the label of application will be used which is the app name.

We can add action button to make Toolbar more interactable. Download the resources from the link mentioned in the preface. Right click res → New → Directory and create menu folder. Then right click menu → New → Menu resource file to create toolbar.xml and add the following code:

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/backup"
        android:icon="@drawable/ic_backup"
        android:title="Backup"
        app:showAsAction="always" />
    <item
        android:id="@+id/delete"
        android:icon="@drawable/ic_delete"
        android:title="Delete"
        app:showAsAction="ifRoom" />
    <item
        android:id="@+id/settings"
        android:icon="@drawable/ic_settings"
        android:title="Settings"
        app:showAsAction="never" />
</menu>

```

The <item> tag will add the action button, and the attributes are self-explanatory.

The app:showAsAction attribute can set the position to display button, and again the app namespace is used to be compatible with lower version OS. showAsAction mainly uses the following values: always will show the button as long as there is enough space; ifRoom will show the button in Toolbar if there is enough room, if not then show the button in menu; never will only display the button in menu. Notice that action button in Toolbar only displays icon and action button in menu only displays text.

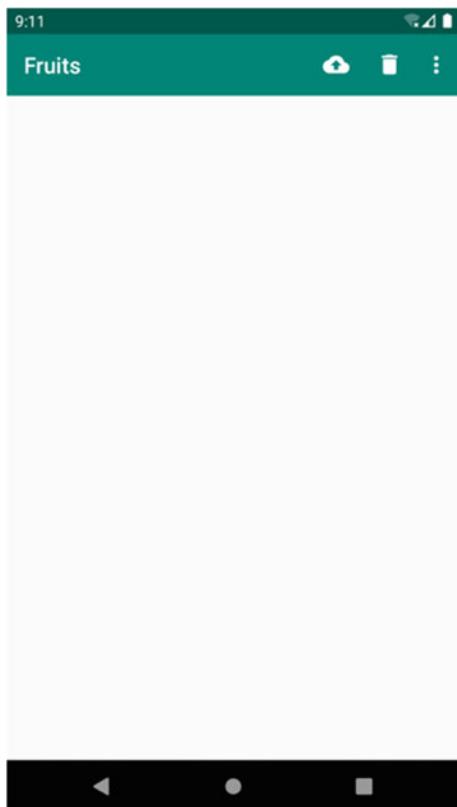
Next, we do the same as in Sect. 3.2.5. Edit MainActivity as code below:

```

class MainActivity : AppCompatActivity() {
    ...
    override fun onCreateOptionsMenu(menu: Menu?) : Boolean {
        menuInflater.inflate(R.menu.toolbar, menu)
        return true
    }
}

```

**Fig. 12.4** Toolbar with action button



```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.backup -> Toast.makeText(this, "You clicked Backup",
            Toast.LENGTH_SHORT).show()
        R.id.delete -> Toast.makeText(this, "You clicked Delete",
            Toast.LENGTH_SHORT).show()
        R.id.settings -> Toast.makeText(this, "You clicked Settings",
            Toast.LENGTH_SHORT).show()
    }
    return true
}
```

The toolbar.xml is loaded in onCreateOptionsMenu(), and button click events will be handled in onOptionsItemSelected(). Run the app again, and the result should be as shown in Fig. 12.4.

Since there is enough room to display, thus both Backup button and Delete button are displayed in the Toolbar. Settings button is set to never, and thus we need to click the menu to see it.

That is all for Toolbar for now. We will explore this widget together with other controls later.

## 12.3 Navigation Drawer

Navigation drawer is one of the most commonly seen Material Design control and widely used in Google apps like Gmail, Google Photo, etc. It looks complicated to implement, but with the tools provided by Google, we can easily create such effect.

### 12.3.1 *DrawerLayout*

The navigation drawer will hide the menu items and will slide in to display the menu items. This save screen space and has smooth animation and is the recommended in Material Design.

It will be extremely hard to implement from scratch, but with the help of *DrawerLayout* from *AndroidX*, it is super easy to build.

This layout allows two widgets inside it: The first widget is the content in the main screen, and the second widget is the content that is slide in menu or the detail. Thus we can edit *activity\_main.xml* as follows:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="@color/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </FrameLayout>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

**Fig. 12.5** Navigation drawer UI



```
    android:layout_gravity="start"
    android:background="#FFF"
    android:text="This is menu"
    android:textSize="30sp" />

</androidx.drawerlayout.widget.DrawerLayout>
```

The first widget in `DrawerLayout` is `FrameLayout` which contains the `Toolbar`; the second widget is a `TextView`. `DrawLayout` has no requirement for what kind of widget can be used here.

Notice that the second widget has to set `layout_gravity` because we need to tell the `DrawerLayout` where to put the slide in menu by set this property to left or right. Here the value of start means that it will be positioned based on language preference; if system language is from left to right such as English and Chinese, then menu will be at the left side, and for languages like Arabic, the menu will be at the right side.

That is all we need. Run the app again and drag from left side to right side to display the navigation drawer as shown in Fig. 12.5.

Swiping to left or clicking area outside the menu will close the navigation drawer and go back to the main screen. The slide in and out animation should be smooth.

Isn't it exciting that we can achieve such complicated animation with such simple edits? However, for now, the navigation drawer can only display when swiping the screen, and it may not be detected by the user. How do we let the user know there is a menu?

The recommendation from Material Design is to add a navigation button at the left side of the Toolbar. Clicking this button will also display the menu content. This will increase the awareness of the existence of the menu.

Put the `ic_menu.png` in `drawable-xxhdpi` folder and edit `MainActivity` as code below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)
        supportActionBar?.let {
            it.setDisplayHomeAsUpEnabled(true)
            it.setHomeAsUpIndicator(R.drawable.ic_menu)
        }
    }
    ...
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            android.R.id.home -> drawerLayout.openDrawer(GravityCompat.
START)
            ...
        }
        return true
    }
}
```

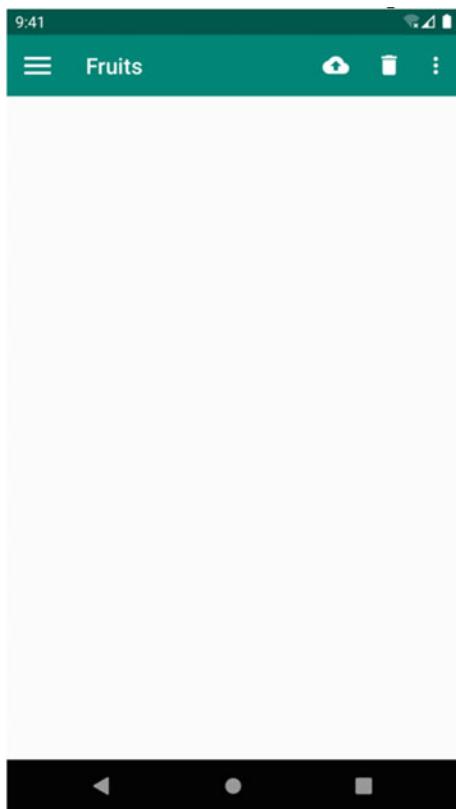
The code changes here first get an object of `ActionBar` which is implemented by `Toolbar`. Then if `ActionBar` is not null, call its `setDisplayHomeAsUpEnabled()` method to display the navigation button and call its `setHomeAsUpIndicator()` method to set the navigation button icon. As a matter of fact, the left side button on the `Toolbar` is called Home button and by default is a back arrow which means going back to the last Activity. Here we replaced the default look and functionality.

Next, handle the Home button click event in `onOptionsItemSelected()`. Home button's id should always be `android.R.id.home`. Then we can call `openDrawer()` of `DrawerLayout` to display the menu. Notice that `openDrawer()` requires a param of `Gravity` type, and we pass in `GravityCompat.START` to match definition in XML.

Run the app again, and the result should be as shown in Fig. 12.6.

User will see a button at the left side of the `Toolbar`, and once they click it, the navigation drawer will show up.

**Fig. 12.6** Display home button



### 12.3.2 NavigationView

Now let us add more content in the menu to replace the old boring TextView only menu. First let us take a look at how does Play Store's navigation drawer look like which is shown in Fig. 12.7.

Compared with the official app's navigation drawer, our current implementation is too elementary, and let us improve it step by step in this section.

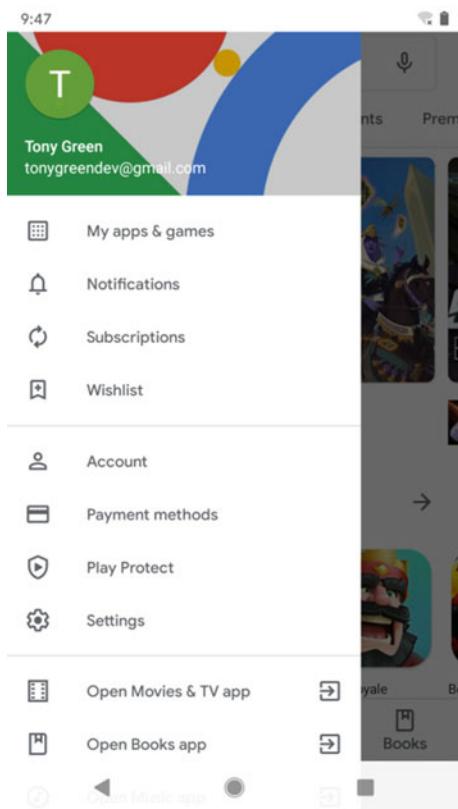
Although you can put in any widget you want into the drawer, a better choice is to use Google's native widget NavigationView. NavigationView is a widget in Material lib which was designed to meet Material Design standards and at the same time makes the implementation effortless.

Since it is in Material lib, we need to import this lib first.

Open app/build.gradle and add the following content in dependencies closure:

```
dependencies {  
    ...  
    implementation 'com.google.android.material:material:1.0.0'  
}
```

**Fig. 12.7** Navigation drawer in Play Store



```
    implementation 'de.hdodenhof:circleimageview:3.0.1'  
}
```

The first dependency is the Material lib, and the second dependency is an open-source project CircleImageView which can show image in circular shape.

We also need another two widgets: menu and headerLayout. Menu is used to display the detail menu items in NavigationView, and headerLayout is used to display the header layout in NavigationView.

First let us create menu. Put the images mentioned here in drawable-xxhdpi folder. Right click menu folder → New → Menu resource file and create nav\_menu.xml with the following code:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
    <group android:checkableBehavior="single">  
        <item  
            android:id="@+id/navCall"  
            android:icon="@drawable/nav_call"  
            android:title="Call" />  
    </item>
```

```

    android:id="@+id/navFriends"
    android:icon="@drawable/nav_friends"
    android:title="Friends" />
<item
    android:id="@+id/navLocation"
    android:icon="@drawable/nav_location"
    android:title="Location" />
<item
    android:id="@+id/navMail"
    android:icon="@drawable/nav_mail"
    android:title="Mail" />
<item
    android:id="@+id/navTask"
    android:icon="@drawable/nav_task"
    android:title="Tasks" />
</group>
</menu>
```

We embed `<group>` element in `<menu>` and set its `checkableBehavior` attribute to `single` which means that only one item can be selected at a time.

We defined five items, and the attributes should be self-explanatory enough to understand.

The headerLayout is customizable, and for simplicity we just put profile image, user name, and email address.

Talking about profile image, I prepared a pet image and put it in `drawable-xxhdpi` folder. You can use your own image, but the image better be a square because we will round the image later. Right click layout folder → New → Layout resource file and create `nav_header.xml` file. Edit this file as code below:

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/
res/android"
    android:layout_width="match_parent"
    android:layout_height="180dp"
    android:padding="10dp"
    android:background="@color/colorPrimary">

    <de.hdodenhof.circleimageview.CircleImageView
        android:id="@+id/iconImage"
        android:layout_width="70dp"
        android:layout_height="70dp"
        android:src="@drawable/nav_icon"
        android:layout_centerInParent="true" />

    <TextView
        android:id="@+id/mailText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="tonygreendev@gmail.com"
```

```
    android:textColor="#FFF"
    android:textSize="14sp" />

<TextView
    android:id="@+id/userText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_above="@+id/mailText"
    android:text="Tony Green"
    android:textColor="#FFF"
    android:textSize="14sp" />

</RelativeLayout>
```

The outer layer of this layout is a RelativeLayout with width of value match\_parent and height of value 180dp and background color of value colorPrimary.

In the RelativeLayout, we place three widgets; CircleImageView is to display image with circular shape and used just like ImageView. Here we set an image to be the profile image and set alignment to be centered. Other two TextViews are used to display user name and email address, and you should be familiar with the attributes related to RelativeLayout.

Finally, we can use NavigationView. Edit activity\_main.xml as code below:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="@color/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

    </FrameLayout>

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/navView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="start" />
```

```

    app:menu="@menu/nav_menu"
    app:headerLayout="@layout/nav_header"/>

</androidx.drawerlayout.widget.DrawerLayout>

```

The highlighted replaced the TextView with NavigationView and pass in the menu, and headerLayout was just created by app:menu and app:headerLayout attributes.

To handle the click event of menu items, edit MainActivity as code below:

```

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        setSupportActionBar(toolbar)
        supportActionBar?.let {
            it.setDisplayHomeAsUpEnabled(true)
            it.setHomeAsUpIndicator(R.drawable.ic_menu)
        }
        navView.setCheckedItem(R.id.navCall)
        navView.setNavigationItemSelectedListener {
            drawerLayout.closeDrawers()
            true
        }
    }
    ...
}

```

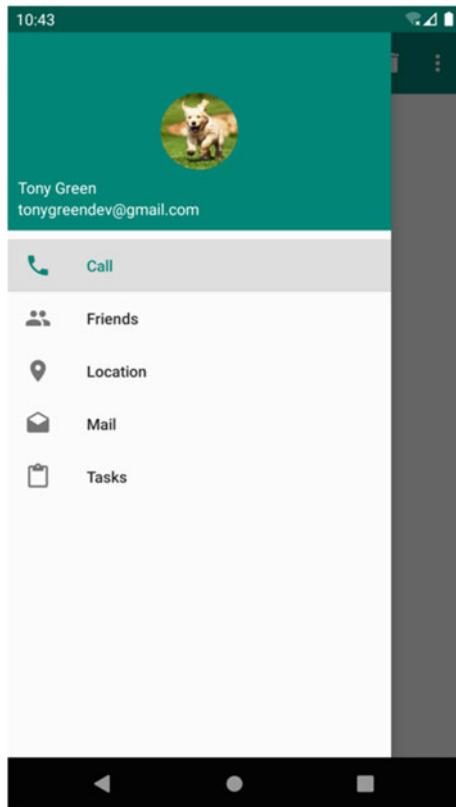
The highlighted code set Call item as default selected item and call setNavigationItemSelectedListener() to set the click event listener. When user click any menu item, the Lambda expression will be called to handle the click event. Here we simply call closeDrawers() to close the drawer and return true to indicate that event has been handled.

Run the app and click the navigation button on the left side of Toolbar, and the result should be as shown in Fig. 12.8.

Now the navigation drawer looks much better. This is why Material Design is so fascinating. It has very beautiful design principles, and if you follow the principles to create UI, the result will look beautiful.

## 12.4 FloatingActionButton and Snackbar

Material Design is not limited to 2D surface but has 3D information. Floating action button is an iconic official example of 3D design. This button does not belong to the surface of the main UI but is elevated and has a feeling of floating.



**Fig. 12.8** NavigationView UI

In this section we will discuss floating action button and also another interactable pop-up tool. We used to use `Toast` for pop-up message to notify user; however user cannot interact with `toast` message, and in this section we will take a look at how to allow user take action in the pop-up message.

#### **12.4.1 *FloatingActionButton***

`FloatingActionButton` is another widget in the `Material lib` which can help us easily create a floating action button in our app. In Fig. 12.2 we previewed floating action button which uses `colorAccent` as the color of the button by default. We can set an icon to the button to indicate its use.

Now let us begin. Put `ic_done.png` in `drawable-xxhdpi` folder. Edit `activity_main.xml` as code below:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="@color/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

        <com.google.android.material.floatingactionbutton.
FloatingActionButton
            android:id="@+id/fab"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom|end"
            android:layout_margin="16dp"
            android:src="@drawable/ic_done" />

    </FrameLayout>
    ...
</androidx.drawerlayout.widget.DrawerLayout>
```

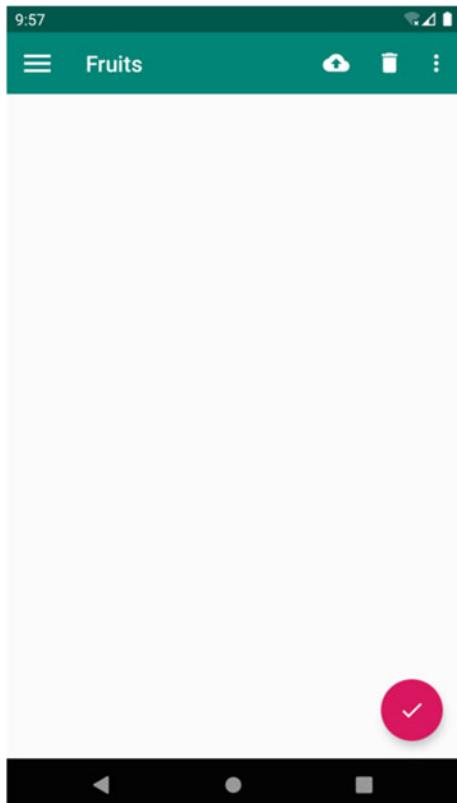
The highlighted code adds a FloatingActionButton in the main screen layout. There is no special requirement to use this widget. Its layout\_width and layout\_height attributes are set to wrap\_content, and layout\_gravity attribute is set to the bottom right of the screen. The end and start work as mentioned. The layout\_margin attribute gives some space for the widget so that it will not touch the edge of the screen. In the end the src attribute sets the icon for the FloatingActionButton.

Now run the app, and the result should be as shown in Fig. 12.9.

With simple code change, we now get a beautiful floating action button on the bottom right of the screen.

You shall find that this button has shadow effect. This is because FloatingActionButton is floating on top of the current surface, and thus there will be shadow.

We can also set the elevation of the FloatingActionButton as shown in code below:



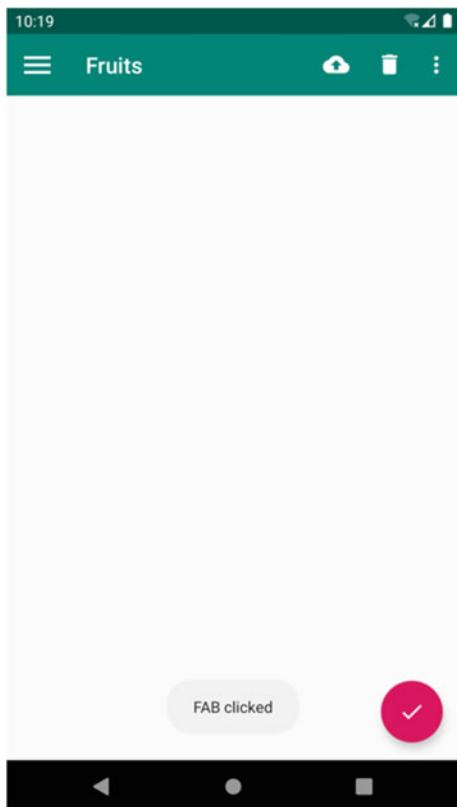
**Fig. 12.9** FloatingActionButton Effect

```
<com.google.android.material.floatingactionbutton.  
FloatingActionButton  
    android:id="@+id/fab"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="bottom|end"  
    android:layout_margin="16dp"  
    android:src="@drawable/ic_done"  
    app:elevation="8dp" />
```

The `app:elevation` attribute sets the elevation of FloatingActionButton. The larger the number is, the larger but lighter the shadow will be. However, I do not feel the difference is so significant and using the default value is enough.

Next let us see how does FloatingActionButton handle click event, after all buttons have to be clickable to be useful. Edit MainActivity as shown in code below:

**Fig. 12.10** Handle FloatingActionButton click event



```
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        ...  
        fab.setOnClickListener {  
            Toast.makeText(this, "FAB clicked", Toast.LENGTH_SHORT).show()  
        }  
        ...  
    }  
}
```

It is exactly the same as a normal button. They all use `setOnClickListener()` to set the button click listener, and here we simply show a `Toast` message.

Run the app again and click “FloatingActionButton,” and the result should be as shown in Fig. 12.10.

### 12.4.2 Snackbar

It is time to use a more advanced pop-up widget to show message to user! Let us take a look at Snackbar.

I want to make it clear that Snackbar is not to replace Toast because they are used in different scenarios. Toast is used to tell the user what is going on, but user does not need to take action. Snackbar adds a button so that user can take action. For instance, if we only show Toast message to user when they accidentally delete data, they will feel frustrated for sure. However, we can add an Undo button to let user recover the data and improve user experience.

Snackbar is easy to use too. It is very similar as using Toast except that we can add extra click event code. Edit MainActivity as shown below:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        fab.setOnClickListener { view ->
            Snackbar.make(view, "Data deleted", Snackbar.LENGTH_SHORT)
                .setAction("Undo") {
                    Toast.makeText(this, "Data restored", Toast.LENGTH_SHORT)
                        .show()
                }
                .show()
        }
        ...
    }
}
```

The make() method will create an object of Snackbar. Its first param is a View and can be any View in the current layout. Snackbar will find that outmost layer layout to display the message; the second param is the content to be displayed; the third param is the display duration of Snackbar which is similar to Toast.

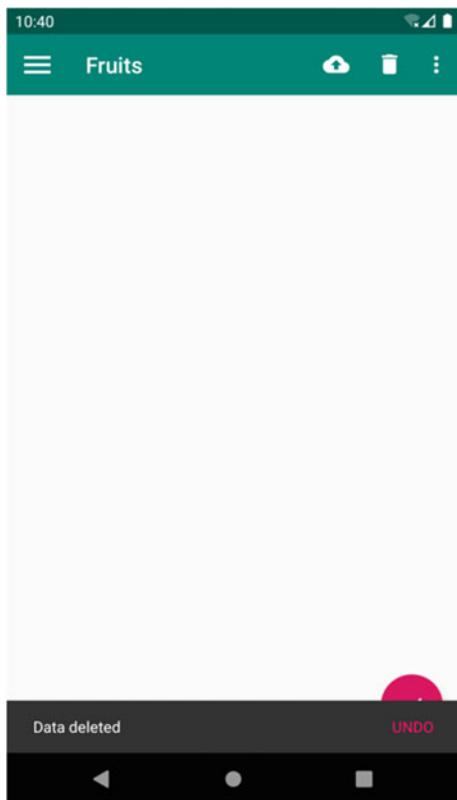
The setAction() method makes Snackbar not a static message but interactable. For simplicity we just show a Toast message to indicate that this is triggered. At last, we can use show() to display the Snackbar.

Run the app and click the floating action button, and the result should be as shown in Fig. 12.11.

As you can see, Snackbar appears at the bottom with the text we set and a clickable Undo button, and after some time, it will disappear automatically. Snackbar has animation for appearing and disappearing to enhance the visual effect.

But you should notice that Snackbar covered the floating action button when it is on the screen. Though Snackbar will disappear in a short time, we still need to fix this. With CoordinatorLayout we can solve this easily.

**Fig. 12.11** Snackbar display



#### 12.4.3 CoordinatorLayout

CoordinatorLayout is like an enhanced FrameLayout and is provided in AndroidX. Most of the time it works as a FrameLayout but with some extra Material capability.

CoordinatorLayout can listen to the events of all the controls inside it and automatically responds to the events. For instance, the Snackbar covers FloatingActionButton, and if we can let CoordinatorLayout listen to the display event of Snackbar, then it will automatically move up FloatingActionButton to ensure that it will not be covered by Snackbar.

We can simply replace the old FrameLayout to CoordinatorLayout to use it. Edit activity\_main.xml as shown in code below:

```
<androidx.drawerlayout.widget.DrawerLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:id="@+id/drawerLayout"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">
```

```
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <androidx.appcompat.widget.Toolbar  
        android:id="@+id/toolbar"  
        android:layout_width="match_parent"  
        android:layout_height="?attr/actionBarSize"  
        android:background="@color/colorPrimary"  
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"  
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />  
  
    <com.google.android.material.floatingactionbutton.  
FloatingActionButton  
        android:id="@+id/fab"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_gravity="bottom|end"  
        android:layout_margin="16dp"  
        android:src="@drawable/ic_done" />  
  
    </androidx.coordinatorlayout.widget.CoordinatorLayout>  
    ...  
</androidx.drawerlayout.widget.DrawerLayout>
```

Since CoordinatorLayout is an enhanced FrameLayout, there is no side effect for the replacement. Run the app again and click the FloatingActionButton, and the result is as shown in Fig. 12.12.

FloatingActionButton got moved up the same height as Snackbar's height and was not covered at all. When Snackbar disappears, the FloatingActionButton will move back to its original position.

Also, the reposition of FloatingActionButton has animation and synchronizes with Snackbar which makes the process very smooth.

However, Snackbar is not inside CoordinatorLayout; why its event can be listened by CoordinatorLayout?

The answer is simple. The first param we pass in make() of Snackbar specifies which View triggers Snackbar, and we passed in FloatingActionButton which is in CoordinatorLayout. Thus Snackbar's event can be listened by CoordinatorLayout. If you pass DrawerLayout to make(), then Snackbar will cover the FloatingActionButton again as DrawerLayout is not in CoordinatorLayout. Next, let us take a look at CardView.

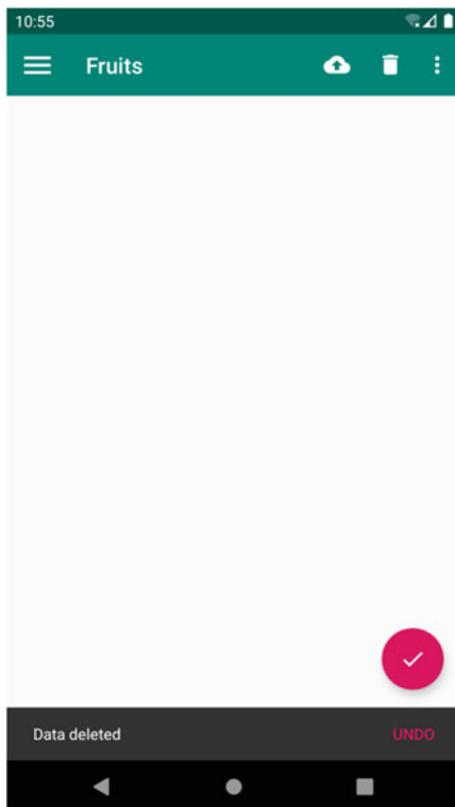


Fig. 12.12 CoordinatorLayout automatically moves up FloatingActionButton

## 12.5 CardView Layout

We've already got some Material Design effects in the UI, but the main screen is pretty empty, and I will use some fruit pictures to fill the screen. In Material Design there is card view layout style that we can use. It will make the elements in the screen looks like card which can have corner radius and shadow.

### 12.5.1 *MaterialCardView*

MaterialCardView is used to create the card view effect provided by Material lib. MaterialCardView is actually a FrameLayout too but can render corner radius and shadow and looks like 3D object.

Here is a simple example to use MaterialCardView:

```
<com.google.android.material.card.MaterialCardView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    app:cardCornerRadius="4dp"  
    app:elevation="5dp">  
    <TextView  
        android:id="@+id/infoText"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"/>  
</com.google.android.material.card.MaterialCardView>
```

The `app:cardCornerRadius` attribute sets the corner radius of the card. The `app:elevation` attribute works exactly the same as in `FloatingAcitonButton`. The `TextView` will be rendered in the card.

Apparently, we should not just display one card in the screen. To make full use of the screen, let us use `RecyclerView` to fill the full screen to implement an advanced version of fruits list.

Download the images as mentioned before and put them into the current project.

Since we need to use `RecyclerView`, we need to add the dependencies in `app/build.gradle`:

```
dependencies {  
    ...  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'com.github.bumptech.glide:glide:4.9.0'  
    annotationProcessor 'com.github.bumptech.glide:compiler:4.9.0'  
}
```

The last two dependencies added dependency to Glide which is a super powerful open-source lib to load images. It can load image from local device, network, GIF, and even local videos. The most important thing is that Glide is super easy to use, and with a few lines of code, you can load the images. The project website is <https://github.com/bumptech/glide>.

Let us start the implementation. Edit `activity_main.xml` as shown below:

```
<androidx.drawerlayout.widget.DrawerLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:id="@+id/drawerLayout"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
<androidx.appcompat.widget.Toolbar  
    android:id="@+id/toolbar"  
    android:layout_width="match_parent"
```

```
    android:layout_height="?attr/actionBarSize"
    android:background="@color/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

<com.google.android.material.floatingactionbutton.
FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="16dp"
    android:src="@drawable/ic_done" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
...
</androidx.drawerlayout.widget.DrawerLayout>
```

The above code adds a RecyclerView in CoordinatorLayout and sets its id, and set the width and height to match\_parent to fill the full screen.

Next, define a Fruit class as shown below:

```
class Fruit(val name: String, val imageId: Int).
```

There are two fields in this class: name is for fruit name, and imageid is for the image resource id.

Next, we need to set layout for the RecyclerView item. Create fruit\_item.xml under layout directory with the following code:

```
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="5dp"
    app:cardCornerRadius="4dp">

<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

<ImageView
    android:id="@+id/fruitImage"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:scaleType="centerCrop" />
```

```
<TextView  
    android:id="@+id/fruitName"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:layout_margin="5dp"  
    android:textSize="16sp" />  
</LinearLayout>  
  
</com.google.android.material.card.MaterialCardView>
```

The MaterialCardView is the top-level layout which makes every elements in RecyclerView positioned in the card. MaterialCardView is a FrameLayout; thus it is not easy to position each item, and we embed a LinearLayout to position elements.

We create an ImageView to display the image of fruit and a TextView to display the name of the fruit and let the TextView positioned in the center vertically. Notice that the scaleType attribute of ImageView was set to centerCrop because the ratios of images may not be the same, and in order to let the image fill the full ImageView, we need to keep the ratio and crop the parts that are outside of ImageView.

Next create FruitAdapter class to serve as adapter for the RecyclerView as shown in code below:

```
class FruitAdapter(val context: Context, val fruitList: List<Fruit>)  
:  
    RecyclerView.Adapter<FruitAdapter.ViewHolder>() {  
  
    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view)  
{  
        val fruitImage: ImageView = view.findViewById(R.id.fruitImage)  
        val fruitName: TextView = view.findViewById(R.id.fruitName)  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
ViewHolder {  
        val view = LayoutInflater.from(context).inflate(R.layout.  
fruit_item, parent,  
        false)  
        return ViewHolder(view)  
    }  
  
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
        val fruit = fruitList[position]  
        holder.fruitName.text = fruit.name  
        Glide.with(context).load(fruit.imageId).into(holder.  
fruitImage)  
    }  
  
    override fun getItemCount() = fruitList.size  
}
```

It is almost exactly the same as the FruitAdapter in Chap. 4 except that we now use Glide to load the image in onBindViewHolder().

As you can see it is really simple to use Glide. First pass Context, Activity, or Fragment param to Glide.with() and the call load() and pass URL, local path or resource id to it, at last, call into() to set the image to a specified ImageView.

Why do we need to use Glide this time? This is because the images here have high resolution, and without compression, we will see OOM exception. But Glide takes care of it, and we just need to load the image without worrying about any issue.

After the RecyclerView adapter is done, edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {

    val fruits = mutableListOf(Fruit("Apple", R.drawable.apple), Fruit
    ("Banana",
        R.drawable.banana), Fruit("Orange", R.drawable.orange), Fruit
    ("Watermelon",
        R.drawable.watermelon), Fruit("Pear", R.drawable.pear), Fruit
    ("Grape",
        R.drawable.grape), Fruit("Pineapple", R.drawable.pineapple),
    Fruit("Strawberry",
        R.drawable.strawberry), Fruit("Cherry", R.drawable.cherry), Fruit
    ("Mango",
        R.drawable.mango))

    val fruitList = ArrayList<Fruit>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        initFruits()
        val layoutManager = GridLayoutManager(this, 2)
        recyclerView.layoutManager = layoutManager
        val adapter = FruitAdapter(this, fruitList)
        recyclerView.adapter = adapter
    }

    private fun initFruits() {
        fruitList.clear()
        repeat(50) {
            val index = (0 until fruits.size).random()
            fruitList.add(fruits[index])
        }
    }
}
```

In MainActivity, we create a fruit list and filled with Fruit instances. Each instance stands for one kind of fruit. In initFruits(), we first clear the fruitList and



**Fig. 12.13** MaterialCardView UI

then use a random number to randomly pick fruit from the fruit list to fruitList so that every time we open the app we can see a different list. Next, to get more data to fill the screen, we use repeat() function to pick 50 fruit items.

The RecyclerView uses GridLayoutManager to layout the items. We covered LinearLayoutManager and StaggeredGridLayoutManager in Chap. 4, and this is the last missing piece of all the layout managers. There is nothing special to use GridLayoutManager; its constructor takes two params: The first is Context, and the second is the number of columns, and we set it to 2.

Run the app again, and you should see something as shown in Fig. 12.13.

As you can see each fruit is in its own card which has rounded corner and shadow. Notice that since we randomly pick fruits, so there will be some duplication.

You might notice that the Toolbar is missing, and you can actually find that it gets covered by RecyclerView. To solve this we need AppBarLayout.

### 12.5.2 AppBarLayout

The reason why RecyclerView can cover Toolbar is because RecyclerView and Toolbar are both in the CoordinatorLayout and as mentioned before CoordinatorLayout is just an advanced FrameLayout. By default, FrameLayout will put all the components at the top left corner which will cover each other. We also saw this in Sect. 4.3.3 when we were discussing FrameLayout.

Usually we can set some margin to move items around with some space, and here we can simply move down the RecyclerView with the height of the Toolbar to unblock Toolbar. However, as we are using CoordinatorLayout, we can have more elegant solution.

I will use another tool in Material lib—AppBarLayout. It is actually a vertical LinearLayout and encapsulates lots of scrolling event handling and applied the guidelines of Material Design.

Now we just need two steps to solve the problem. Step one is to embed Toolbar in AppBarLayout. Step two is to set layout behavior for the RecyclerView. Edit activity\_main.xml as code below:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.coordinatorlayout.widget.CoordinatorLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <com.google.android.material.appbar.AppBarLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content">

            <androidx.appcompat.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                android:background="@color/colorPrimary"
                android:theme="@style/ThemeOverlay.AppCompat.Dark.
                ActionBar"
                app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

        </com.google.android.material.appbar.AppBarLayout>

        <androidx.recyclerview.widget.RecyclerView
            android:id="@+id/recyclerView"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
```

**Fig. 12.14** Correct position of RecyclerView and Toolbar



```
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
/>
...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
...
</androidx.drawerlayout.widget.DrawerLayout>
```

There is not much change for the layout. We just followed the two steps above. The `appbar_scrolling_view_behavior` string is provided by Material lib.

Run the app again, and you should see that Toolbar and RecyclerView are in their right position, as shown in Fig. 12.14.

Now AppBarLayout is able to listen to the scrolling event of RecyclerView, and we can optimize it to implement Material Design effect for AppBarLayout.

When AppBarLayout received the scrolling event, the component inside it can react to the event by setting `app:layout_scrollFlags` attribute. Edit activity\_main.xml as code below:

```
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/drawerLayout"
android:layout_width="match_parent"
android:layout_height="match_parent">>

<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="@color/colorPrimary"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.
ActionBar"
            app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
            app:layout_scrollFlags="scroll|enterAlways|snap" />

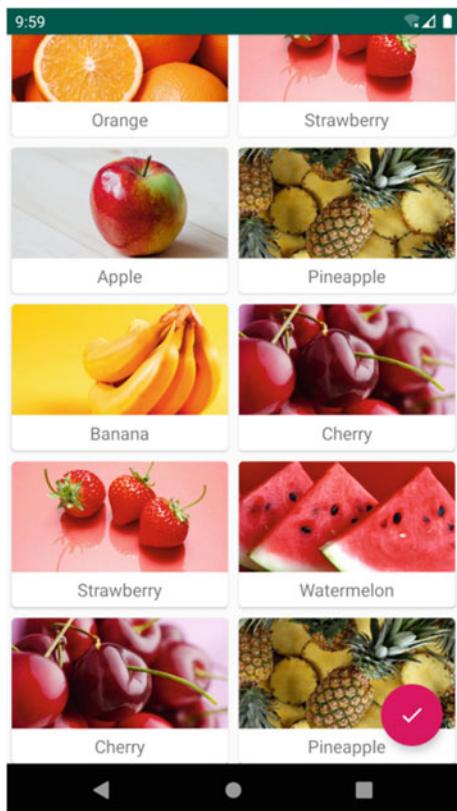
        </com.google.android.material.appbar.AppBarLayout>
        ...
    </androidx.coordinatorlayout.widget.CoordinatorLayout>
    ...
</androidx.drawerlayout.widget.DrawerLayout>
```

The highlighted code adds `app:layout_scrollFlags` attribute and sets it to `scroll|enterAlways|snap`. Among them, scroll means that when RecyclerView is scrolling up, Toolbar will scroll together and hide; enterAlways means that when RecyclerView is scrolling down, Toolbar will scroll down and appear again; snap means that before Toolbar is totally hidden or full size display, it will automatically determine if Toolbar will hide or display.

We just need this one line change. Run the app again and scroll up, and the result should be as shown in Fig. 12.15.

As you can see, as we scroll up, Toolbar disappears! And if we scroll down, Toolbar will reappear. This is because the Material Design guidelines think that when user is scrolling up, they are focusing on the content of RecyclerView, and we want max space and clean UI to optimize user experience. When user needs to act on Toolbar, they just need to scroll down to see it. This design ensures the best viewing experience and does not negatively affect any functionality. Of course, using ActionBar is impossible to implement this. Toolbar provides this possibility.

**Fig. 12.15** Scroll up to hide Toolbar



## 12.6 Pull to Refresh

Pull to refresh has been implemented in almost every app. However, the style of pull to refresh of different apps is different and does not always follow the guidelines of Material Design. Google actually has official design guidelines for pull to refresh in Material Design. We do not need to understand the guidelines because we can just use the existing components directly which follows the guidelines.

SwipeRefreshLayout is the core class to implement pull to refresh. It is also provided in AndroidX. We just need to put the component into SwipeRefreshLayout to let this component support pull to refresh. In our MaterialTest project, the component is RecyclerView.

It is fairly easy to use SwipeRefreshLayout. Edit activity\_main.xml as code below:

```
<androidx.drawerlayout.widget.DrawerLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:id="@+id/drawerLayout"
```

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

    <androidx.coordinatorlayout.widget.CoordinatorLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        ...
        <androidx.swiperefreshlayout.widget.SwipeRefreshLayout
            android:id="@+id/swipeRefresh"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_behavior="@string/
appbar_scrolling_view_behavior">

            <androidx.recyclerview.widget.RecyclerView
                android:id="@+id/recyclerView"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                app:layout_behavior="@string/
appbar_scrolling_view_behavior" />

        </androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
        ...
    </androidx.coordinatorlayout.widget.CoordinatorLayout>
    ...
</androidx.drawerlayout.widget.DrawerLayout>
```

The highlighted code change simply embeds the RecyclerView inside SwipeRefreshLayout, and then it will support pull to refresh. Notice that since RecyclerView is inside SwipeRefreshLayout, we need to move the app:layout\_behavior attribute into SwipeRefreshLayout.

But that's not the end of it yet, we still need to handle the refreshing logic. Edit MainActivity as code below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        swipeRefresh.setColorSchemeResources(R.color.colorPrimary)
        swipeRefresh.setOnRefreshListener {
            refreshFruits(adapter)
        }
    }

    private fun refreshFruits(adapter: FruitAdapter) {
        thread {
            Thread.sleep(2000)
            runOnUiThread {
                initFruits()
            }
        }
    }
}
```

```
        adapter.notifyDataSetChanged()
        swipeRefresh.isRefreshing = false
    }
}
...
}
```

The `setColorSchemeResource()` method sets the color of the progress bar, and we just use `colorPrimary` here. The `setOnRefreshListener()` sets the listener for pull to refresh event, and the Lambda expression will be called, and we can write business logic here.

Most of the cases, when users pull to refresh, we need to fetch data from network and display the data. For simplicity, we will not interact with network but call `refreshFruits()` to refresh locally. This method starts a new thread and let it sleep for 2 s. This is because local refresh is very fast, and if we do not put thread to sleep, it will finish instantly, and we cannot see the refreshing process. After 2 s, `runOnUiThread()` will switch the thread to main thread, and the `initFruits()` method will regenerate data. The `notifyDataSetChanged()` method will notify the change. In the end, we pass `false` to `setRefreshing()`, indicating that refreshing event finishes and hide the progress bar.

Run the app again and pull the screen, and you should see a progress ring appear, and release your fingers, and it will refresh as shown in Fig. 12.16.

The progress circle will only display for 2 s, and then the fruits on the screen will be updated.

We have covered a lot of Material Design examples now, but that is not the end of it. Let us take a look at another amazing Material Design effect—collapsible Toolbar.

## 12.7 Collapsible Toolbar

We are using Toolbar to implement the title bar which looks just like the conventional ActionBar. The difference is that it can respond to the scrolling event of RecyclerView to hide and display. Material Design does not limit the look of title bar, and we can customize the title bar based on our preference. In this section, we will create a collapsible title bar with the help of CollapsingToolbarLayout.

### 12.7.1 CollapsingToolbarLayout

`CollapsingToolbarLayout` is also provided in Material lib. It can enrich the UI of Toolbar with some beautiful effects.

**Fig. 12.16** Pull to refresh

However, CollapsingToolbarLayout cannot live alone. By design it has to be positioned inside AppBarLayout, and AppBarLayout has to be inside Coordinatorlayout. This means that we will apply all the knowledge we have learned in this chapter.

First, we need to create another activity to display the details of the fruit. Right click com.example.materialtest package → New → Activity → Empty Activity and create FruitActivity. Name the layout as activity\_fruit.xml, and we can start to implement the layout.

We can divide the activity\_fruit.xml into fruit title bar and fruit content details.

First let us implement the title bar. We will use Coordinatorlayout as the top level layout as shown below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">
```

```
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

There is nothing new to you but do remember to define xmlns:app namespace which is widely used in Material Design.

Next, embed AppBarLayout inside Coordinatorlayout, as code below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <com.google.android.material.appbar.AppBarLayout  
        android:id="@+id/appBar"  
        android:layout_width="match_parent"  
        android:layout_height="250dp">  
    </com.google.android.material.appbar.AppBarLayout>  
  
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Still nothing new. We set an id and set its width to match\_parent and height to 250dp. Of course you can use any height value, but after some experiments, I found 250dp makes UI look the best.

Next, embed CollapsingToolbarLayout in AppBarLayout, as code below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <com.google.android.material.appbar.AppBarLayout  
        android:id="@+id/appBar"  
        android:layout_width="match_parent"  
        android:layout_height="250dp">  
  
        <com.google.android.material.appbar.CollapsingToolbarLayout  
            android:id="@+id/collapsingToolbar"  
            android:layout_width="match_parent"  
            android:layout_height="match_parent"  
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"  
            app:contentScrim="@color/colorPrimary"  
            app:layout_scrollFlags="scroll|exitUntilCollapsed">  
        </com.google.android.material.appbar.CollapsingToolbarLayout>  
  
</com.google.android.material.appbar.AppBarLayout>  
  
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

We use a new layout CollapsingToolbarLayout. Attributes like id, layout\_width, and layout\_height are simple and no need to explain. The android:theme attribute has value of ThemeOverlay.AppCompat.Dark.ActionBar which we used in activity\_main.xml to set the theme of Toolbar. We need to set the theme to a higher level because we need to create more advanced Toolbar effect. The app:contentScrim attribute sets the background color when CollapsingToolbarLayout is collapsing and after collapsed. Since CollapsingToolbarLayout becomes a normal Toolbar after collapsing, the background color should be colorPrimary, and we will see how it looks like shortly. The app:layout\_scrollFlags also gets moved to higher level layout, and scroll means that CollapsingToolbarLayout will scroll together with the fruit details; exitUntilCollapsed means that CollapsingToolbarLayout will stay in the screen after collapsing.

Next, we can set the title bar content in CollapsingToolbarLayout as shown in code below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.appbar.AppBarLayout
        android:id="@+id/appBar"
        android:layout_width="match_parent"
        android:layout_height="250dp">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:id="@+id/collapsingToolbar"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
            app:contentScrim="@color/colorPrimary"
            app:layout_scrollFlags="scroll|exitUntilCollapsed">

            <ImageView
                android:id="@+id/fruitImageView"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:scaleType="centerCrop"
                app:layout_collapseMode="parallax" />

            <androidx.appcompat.widget.Toolbar
                android:id="@+id/toolbar"
                android:layout_width="match_parent"
                android:layout_height="?attr/actionBarSize"
                app:layout_collapseMode="pin" />

        </com.google.android.material.appbar.CollapsingToolbarLayout>
    
```

```
</com.google.android.material.appbar.AppBarLayout>  
  
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Nothing difficult for the code changes here. Notice that in order to make the UI looks better, I add some margin for MaterialCardView and TextView. MaterialCardView adds 35dp to marginTop to save some space.

That is everything we need for fruit title bar, and fruit content details, I will add a FloatingActionButton in activity\_fruit.xml to send comment. Put the ic\_comment.png in drawable-xxhdpi folder. Edit activity\_fruit.xml as code below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <com.google.android.material.appbar.AppBarLayout  
        android:id="@+id/appBar"  
        android:layout_width="match_parent"  
        android:layout_height="250dp">  
        ...  
    </com.google.android.material.appbar.AppBarLayout>  
  
    <androidx.core.widget.NestedScrollView  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        app:layout_behavior="@string/appbar_scrolling_view_behavior">  
        ...  
    </androidx.core.widget.NestedScrollView>  
  
    <com.google.android.material.floatingactionbutton.  
FloatingActionButton  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_margin="16dp"  
        android:src="@drawable/ic_comment"  
        app:layout_anchor="@+id/appBar"  
        app:layout_anchorGravity="bottom|end" />  
  
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

The added FloatingActionButton is at the same surface as AppBarLayout and NestedScrollView. The app:layout\_anchor attribute sets the anchor of FloatingActionButton. We set it to AppBarLayout to make FloatingActionButton display within the fruit title bar area. Then we use app:layout\_anchorGravity attribute to position the FloatingActionButton at the bottom right of the title bar. Other attributes are fairly easy and no need to explain further.

That is all for the whole `activity_fruit.xml`, though it is pretty lengthy, but since we divide it into two pieces and I explained all the new concepts and important things, you should have no difficulty to understand.

After UI, we need to write the function code. Edit `FruitActivity` as code below:

```
class FruitActivity : AppCompatActivity() {

    companion object {
        const val FRUIT_NAME = "fruit_name"
        const val FRUIT_IMAGE_ID = "fruit_image_id"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_fruit)
        val fruitName = intent.getStringExtra(FRUIT_NAME) ?: ""
        val fruitImageId = intent.getIntExtra(FRUIT_IMAGE_ID, 0)
        setSupportActionBar(toolbar)
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
        collapsingToolbar.title = fruitName
        Glide.with(this).load(fruitImageId).into(fruitImageView)
        fruitContentText.text = generateFruitContent(fruitName)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            android.R.id.home -> {
                finish()
                return true
            }
        }
        return super.onOptionsItemSelected(item)
    }

    private fun generateFruitContent(fruitName: String) = fruitName.
    repeat(500)

}
```

In `onCreate()` we get the fruit name and fruit image resource id from Intent and use Toolbar as ActionBar and enable the Home button. By default Home button icon is a back arrow which is what we expected for this activity, thus there is no need to change.

Next, we can fill the content in the screen. The `setTitle()` method of `CollapsingToolbarLayout` will set the fruit name as the current screen's title. Then we can use Glide to load the image into the `ImageView` of the titlebar. For the fruit content details, since it is just an example app, the `generateFruitContent()` method will simply append the fruit name 500 times to generate a long string and display it in the `TextView`.

Lastly, we handle the Home button click event in `onOptionsItemSelected()` by closing the current activity with `finish()` and return to the previous activity.

We still need to handle the click event of RecyclerView; otherwise, we cannot even open FruitActivity. Edit FruitAdapter as code below:

```
class FruitAdapter(val context: Context, val fruitList: List<Fruit>)
:
    RecyclerView.Adapter<FruitAdapter.ViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(context).inflate(R.layout.
fruit_item, parent,
        false)
        val holder = ViewHolder(view)
        holder.itemView.setOnClickListener {
            val position = holder.adapterPosition
            val fruit = fruitList[position]
            val intent = Intent(context, FruitActivity::class.java).apply {
                putExtra(FruitActivity.FRUIT_NAME, fruit.name)
                putExtra(FruitActivity.FRUIT_IMAGE_ID, fruit.imageId)
            }
            context.startActivity(intent)
        }
        return holder
    }
    ...
}
```

This key step is actually very simple. We register a click event listener to the top-level layout of `fruit_item.xml`, and in the event handler, we get the current fruit name and image source id and pass them into Intent and use `startActivity()` to start `FruitActivity`.

Run the app and click any of the fruit in the screen. For instance, after clicking grape picture, the result should be as shown in Fig. 12.17.

Yes! We just created a beautiful UI as shown in the figure. This screen has three parts: fruit title bar, fruit content details, and the FloatingActionButton. The Toolbar and fruit background image perfectly display together which ensure the space to display the image while do not affect the functionality of the Toolbar; the left arrow is used to return to the last Activity.

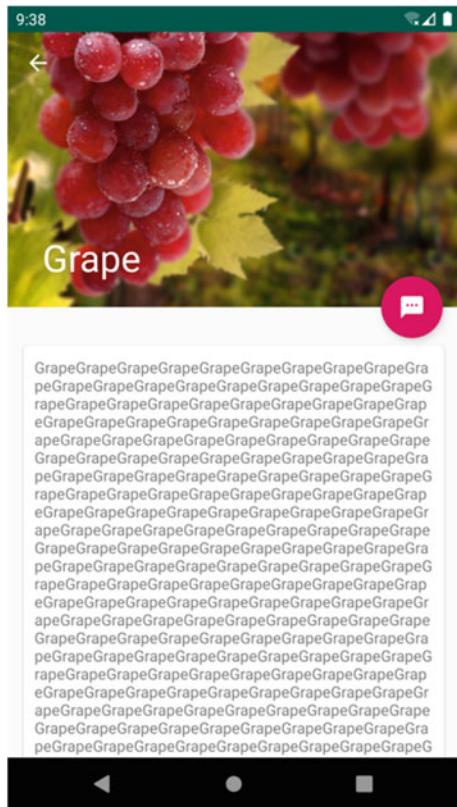
However, there is more. When you scroll up, you should see that the title on the background is becoming smaller and smaller, and image is shifting up too as shown in Fig. 12.18.

This is because when user scroll, they will focus on the content details and title bar will automatically collapse to save space.

Keep scrolling until title bar is in collapsed state as shown in Fig. 12.19.

Now, the background image is gone, and the FloatingActionButton is gone too. The title bar becomes a normal Toolbar. This is because we want to give user max

**Fig. 12.17** Fruit details screen



space to view the details. Scrolling down will reverse the animation and show UI as shown in Fig. 12.17.

We need to thank Material lib for all of these wonderful UI experiences!

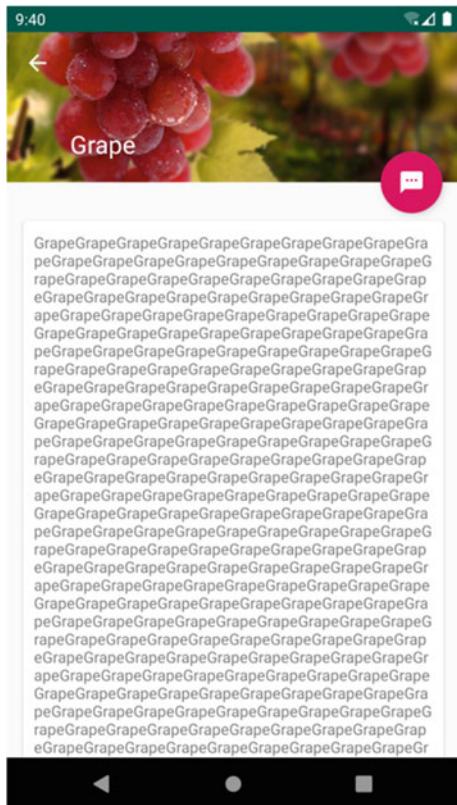
### 12.7.2 Optimizing Using of System Status Bar

There is still some room to improve for the UI. From Fig. 12.17, you can find that the background does not match with the system status bar. It will be great if we can make them share the same style.

Before Android 5.0, we could not change the background or color of the status bar, and there was no such thing as Material Design. But later versions support this, and in the book the lowest version we support is Android 5.0; thus we can make use of this capability to enhance the user experience.

To merge the background image with system status bar, we need to set android:fitsSystemWindows attribute. If we set android: fitsSystemWindows to true for CoordinatorLayout, AppBarLayout, CollapsingToolbarLayout, they can display in

**Fig. 12.18** Scroll up fruit details screen



the system status bar. Thus we need to set this attribute for our ImageView and also all the parent layouts. Edit activity\_fruit.xml as code below:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <com.google.android.material.appbar.AppBarLayout
        android:id="@+id/appBar"
        android:layout_width="match_parent"
        android:layout_height="250dp"
        android:fitsSystemWindows="true">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:id="@+id/collapsingToolbar"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
```

**Fig. 12.19** Title bar in collapsed state



```
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
        android:fitsSystemWindows="true"
        app:contentScrim="@color/colorPrimary"
        app:layout_scrollFlags="scroll|exitUntilCollapsed">

        <ImageView
            android:id="@+id/fruitImageView"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scaleType="centerCrop"
            android:fitsSystemWindows="true"
            app:layout_collapseMode="parallax" />
        ...
    </com.google.android.material.appbar.CollapsingToolbarLayout>

    </com.google.android.material.appbar.AppBarLayout>
    ...
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

We still need one more step to make it work. We need to set the status bar color to transparent in the theme. This can be done by setting the android:statusBarColor to @android:color/transparent.

Edit the theme in res/values/styles.xml as code below:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <style name="FruitActivityTheme" parent="AppTheme">
        <item name="android:statusBarColor">@android:color/transparent</item>
    </style>
</resources>
```

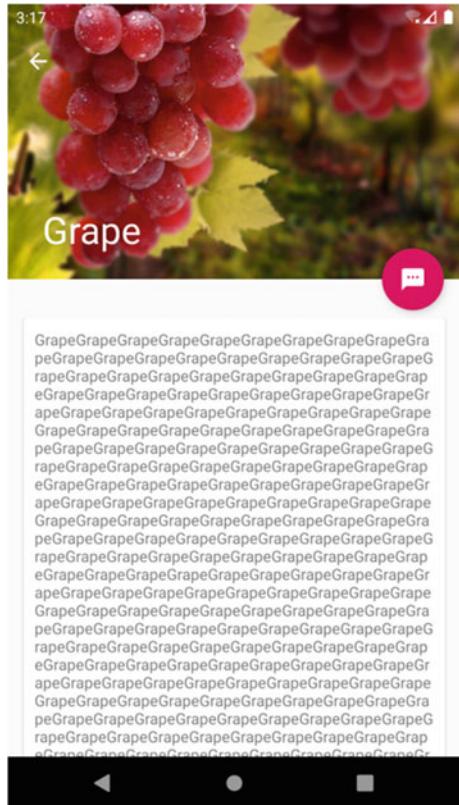
The FruitActivityTheme is specifically defined to be used by FruitActivity. Its parent theme is AppTheme which means that it inherits everything in AppTheme. Thus, we can set the status bar color in FruitActivityTheme to transparent.

To let FruitActivity use this theme, edit AndroidManifest.xml as code below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.materialtest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <activity
            android:name=".FruitActivity"
            android:theme="@style/FruitActivityTheme">
        </activity>
    </application>
</manifest>
```

**Fig. 12.20** Fusion of background image and status bar



The android:theme attribute sets FruitActivityTheme to FruitActivity, and that is what we need. Run MaterialTest again, and the result should be as shown in Fig. 12.20.

Compared with Fig. 12.17, this is a huge leap. That is all for Material Design and time for the Kotlin Class section.

## 12.8 Kotlin Class: Creating Utils

So far, we have already covered majority of the important topics in Kotlin. But we need to know how to use the knowledge flexibly.

Kotlin provides lots of syntax features that allow us extend the functionality to a great deal. APIs can become much easier to use after encapsulation. For instance, the KTX lib was created because Google wanted to simplify APIs. I introduced KTX in this book mainly to build the mindset of using Kotlin flexibly. In this section, I will demonstrate how to create easy to use utils by simplifying a few commonly used APIs.

### 12.8.1 Find Max and Min in N Numbers

To get the larger number between two numbers, we can use if statement or with the help of the Kotlin function max() as shown below:

```
val a = 10
val b = 15
val larger = max(a, b)
```

The code above is very straightforward and looks like no space to optimize. But what if we need to get the max number among three numbers? Since max() can only take two params, we have to compare the first two numbers and compare the larger result with the rest number, as shown in code below:

```
val a = 10
val b = 15
val c = 5
val largest = max(max(a, b), c)
```

This is getting complicated, and we should be able to simplify this.

We discussed vararg keyword in Chap. 7's Kotlin Class which allows the method takes any number of params of the same type. We can use it here to help solve the problem. Create Max.kt file and define max() in it as shown in code below:

```
fun max(vararg nums: Int): Int {
    var maxNum = Int.MIN_VALUE
    for (num in nums) {
        maxNum = max(maxNum, num)
    }
    return maxNum
}
```

Now with vararg keyword, max() can take any number of Integer params. The maxNum will record the current max number and initialized with the smallest integer. In the for-in loop, we iterate the nums param list, and if the current number is larger than maxNum, replace maxNum to the new larger number and in the end return maxNum.

After the above encapsulation, we can simplify the use of max() dramatically. For the same functionality, we can have the following code:

```
val a = 10
val b = 15
val c = 5
val largest = max(a, b, c)
```

Now we do not need to recursively call max() again; we can simply pass N number to max().

However, for now, max() can only be applied to integers. What if we need to get the max value of float numbers or long numbers? Of course you can override max() to receive params of different types as the internal max() does. However this will create duplicated code, and there is some better way to do it.

In Java, numbers of all types are comparable and implement Comparable interface. The same rule applies to Kotlin. With generics, we can let max() takes any number of params that implement Comparable interface, as shown in code below:

```
fun <T : Comparable<T>> max(vararg nums: T) : T {
    if (nums.isEmpty()) throw RuntimeException("Params can not be empty.")
    var maxNum = nums[0]
    for (num in nums) {
        if (num > maxNum) {
            maxNum = num
        }
    }
    return maxNum
}
```

In the code above, param T has to be a subclass of Comparable<T>. Next we check if nums param list is empty, and if true, an exception will be thrown to notify caller that max() requires param. Then we initialize maxNum to the first param in the param list and iterate the param list and update maxNum whenever there is a larger number.

With this change, max() can be applied to all types of numbers. For instance, to get the max value from three float numbers, we can have the following code:

```
val a = 3.5
val b = 3.8
val c = 4.1
val largest = max(a, b, c)
```

Now, whether the params are type of double, single, short, integer, or long, as long as they implements Comparable, max() can be applied and solves the problem once for all.

You can use the same approach to implement min() to get the min value of N numbers. This will be left as exercise for you.

### ***12.8.2 Simplifying Use of Toast***

We have been using Toast for sometime. Have you ever felt that Toast is not easy to use?

The standard way to use Toast to show some string is like this:

```
Toast.makeText(context, "This is Toast", Toast.LENGTH_SHORT).show()
```

This line of code is long, and quite often, developers forget to call the show() method which prevents Toast popping up and causes strange bugs.

Since Toast is used frequently, it is inconvenient to write this again and again, and you should consider simply how to use Toast.

Toast's makeText() takes three params: the first param is the context which is necessary; the second param is the content that needs to be displayed which can be string or string resource id; the third param is the duration time of Toast which only supports Toast.LENGTH\_SHORT and Toast.LENGTH\_LONG.

Then we can add extension function to String class and Int class and encapsulate the logic to display Toast. By doing this, we can just call the extension function to display the Toast.

Create Toast.kt with the following code:

```
fun String.showToast(context: Context) {
    Toast.makeText(context, this, Toast.LENGTH_SHORT).show()
}

fun Int.showToast(context: Context) {
    Toast.makeText(context, this, Toast.LENGTH_SHORT).show()
}
```

The String class and Int class now have their own showToast() function which both accept Context param. Inside the function, we use the standard way to display Toast but change the content to this and set the duration time to Toast.LENGTH\_SHORT.

With this change, when we need to display the Toast, we can use the code below:

```
"This is Toast".showToast(context)
```

This is more succinct than the standard way to show Toast. You can also display Toast with string resource by the following code:

```
R.string.app_name.showToast(context)
```

These two methods call the extension functions we added respectively.

Of course, the fixed duration is also an issue. The simplest solution to this is to declare a param for duration which increases the complexity of using this function again.

In Chap. 2, we discussed setting default value for params, and by applying it here, we can let showToast() support specification of duration while does not increase complexity of calling it. Edit Toast.kt as code below:

```

fun String.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}

fun Int.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}

```

The showToast() function now has a param for duration but with default value. Then we can still call showToast() as before which will have duration of Toast.LENGTH\_SHORT. If you want to use Toast.LENGTH\_LONG, simply call the following code:

```
"This is Toast".showToast(context, Toast.LENGTH_LONG)
```

This will greatly improve your dev efficiency.

### 12.8.3 Simplifying Use of Snackbar

Snackbar is used very similar with Toast but a little bit more complicated.

The standard way to use Snackbar is as shown below:

```

Snackbar.make(view, "This is Snackbar", Snackbar.LENGTH_SHORT)
    .setAction("Action") {
        // handle business logic
    }
    .show()

```

The first param for Snackbar's make() is a View, and the first param for makeText() of Toast is Context. Snackbar can set click listener with setAction(). But besides these two differences, Snackbar is very similar to Toast.

For this kind of API, there is more than one way to simplify, and I will demonstrate one that I prefer.

Since make() takes a param of View which will be used to find the top-level layout to display Snackbar, we can add an extension to View class and encapsulate the logic to display Snackbar.

Create Snackbar.kt with the following code:

```

fun View.showSnackbar(text: String, duration: Int = Snackbar.LENGTH_SHORT) {
    Snackbar.make(this, text, duration).show()
}

```

```
fun View.showSnackbar(resId: Int, duration: Int = Snackbar.LENGTH_SHORT) {
    Snackbar.make(this, resourceId, duration).show()
}
```

This is very similar to the extension functions for showToast() except that we add the extension method to View class and has param of content to display and the duration with default value. Snackbar also supports string and string resource id just as Toast; thus we override showSnackbar() for these two types of params.

Now, to use Snackbar to display some text, we can have the following code:

```
view.showSnackbar("This is Snackbar")
```

If Snackbar does not have setAction(), then the change above is enough. But as the key feature for Snackbar, it will be meaningless to use showSnackbar() without action.

The higher-order function can help here. We can let showSnackbar() to take a param of function type to fully support Snackbar. Edit Snackbar.kt as code below:

```
fun View.showSnackbar(text: String, actionText: String? = null,
    duration: Int = Snackbar.LENGTH_SHORT, block: (() -> Unit)? = null) {
    val snackbar = Snackbar.make(this, text, duration)
    if (actionText != null && block != null) {
        snackbar.setAction(actionText) {
            block()
        }
    }
    snackbar.show()
}

fun View.showSnackbar(resId: Int, actionResId: Int? = null,
    duration: Int = Snackbar.LENGTH_SHORT, block: (() -> Unit)? = null) {
    val snackbar = Snackbar.make(this, resourceId, duration)
    if (actionResId != null && block != null) {
        snackbar.setAction(actionResId) {
            block()
        }
    }
    snackbar.show()
}
```

Both of these two showSnackbar() functions take a param of function type and add a string or string resource id that will be passed to setAction(). Both of these two params have to be nullable with null as default value. Only when both of these two params are not null should we call setAction() to set the click event. Once click event happens, the function-type param will pass the event to the Lambda expression.

Now, showSnackbar() has complete Snackbar functionality, and the example we had in the beginning of this chapter can be done with the following code:

```
view.showSnackbar("This is Snackbar", "Action") {
    // handle business logic
}
```

Compared with the native Snackbar API, showSnackbar() is so much easier to use.

In this chapter's Kotlin Class, we created three util methods and applied top-level function, extension function and higher-order function, vararg keyword and default param value, and so on. Kotlin provides a lot of features for developers to extend and flexibly use APIs. The three functions here are just some simple examples; there are many things you can try out by yourself.

That is the end for this chapter's Kotlin Class section, and we will discuss Git again.

## 12.9 Git Time: Advanced Topics in Version Control

After two sections on Git, you should know how to use frequently used command in Git such as committing code etc.

Open terminal and change directory to the root directory of MaterialTest and commit your code:

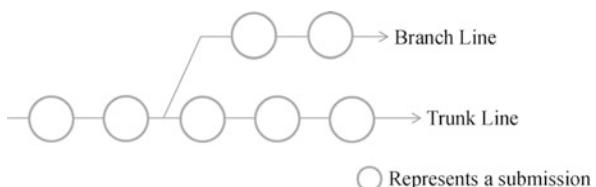
```
git init
git add .
git commit -m "First Commit."
```

This is what we need to do for preparation. Next let us see some advanced topics in Git.

### 12.9.1 Branch

Branch is an important concept in version control. Basically it is a version forks from the existing code so that code can evolve both in main branch and the forked branch without interfering each other. This can be illustrated in Fig. 12.21.

**Fig. 12.21** How branch works



```
[guolindeMacBook-Pro:MaterialTest guolin$ git branch
 * master
guolindeMacBook-Pro:MaterialTest guolin$ ]
```

Fig. 12.22 Check all branches

```
[guolindeMacBook-Pro:MaterialTest guolin$ git branch
 * master
 version1.0
guolindeMacBook-Pro:MaterialTest guolin$ ]
```

Fig. 12.23 Check branches again

You might ask why do we need to fork instead of working on the main branch. Without versions, it is OK to do so. However, with different versions, it will be problematic. For example, assume that you released version 1.0 and is working on version 1.1. However in a few weeks, user reports that there are a few serious bugs in version 1.0, and you need to fix these bugs and update version 1.0. Without branch, you will have to work on top of 1.1 which means that version 1.0 update will actually have code in version 1.1!

If you have branch, there will be no such problem at all. You just need to create a branch for version 1.0 and continue working on main branch to develop version 1.1. When there is issue detected in v1.0, you just need to fix in v1.0 branch and release the updated version and merge the updated code to main branch to ensure that main branch also has the fix.

With this in mind, let us take a look at how to use branch in Git.

To check the branches exist in the current repo, you can use git branch command, and the result is as shown in Fig. 12.22.

We haven't forked yet; thus only master branch exists in the current repo. Let's use the following command to create a branch:

```
git branch version1.0
```

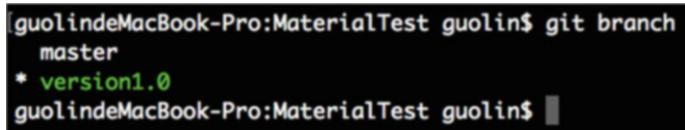
This creates a branch named version1.0, and we can run git branch command again, and the result is as shown in Fig. 12.23.

Now we can see version1.0 branch. You will notice that before master branch there is “\*”.

Mark which means that we're still working on master branch. How to switch to version1.0 branch? We can use checkout command to do so as shown below:

```
git checkout version1.0
```

Run git branch command to check again, and the result is as shown in Fig. 12.24. As you can see, we are on version1.0 branch.



```
[guolindeMacBook-Pro:MaterialTest guolin$ git branch
  master
* version1.0
guolindeMacBook-Pro:MaterialTest guolin$ ]
```

**Fig. 12.24** Branches after switch branch

Notice that the code in version1.0 and master will not interfere with each other. Thus if we fix a bug in v1.0, the bug still exists in master branch. Copy paste is definitely not a good idea, and the best solution is to use merge command as shown below:

```
git checkout master
git merge version1.0
```

This will make the code changes in v1.0 get merged to master branch. Of course, you probably will see merge conflicts, and you need to resolve the conflicts by yourself.

In the end, if we do not need v1.0, we can use the following command to delete this branch:

```
git branch -D version1.0
```

### 12.9.2 Work with Remote Repo

The true power of version control can only be revealed when there is a team instead of a single developer working on the codebase. Each developer will have a version of the code, and after they finish something, they can commit the code to server, and other members just need to sync the code from server to local machine to make sure that everybody has the same copy of code. Each team member just needs to work on their parts and collectively create an eng-heavy project.

To use Git for team collaboration, we first need a remote repo so that each team member can get the original code from and develop on their own and then sync the code to remote repo. Notice that team members should build the habit of pulling the latest code from remote repo to reduce code conflict.

Assume there is a remote repo with Git address <https://github.com/example/test.git>; then we can use the following command to download the code to local machine:

```
git clone https://github.com/example/test.git
```

To sync the changes in local machine, we can use push command as shown below:

```
git push origin master
```

The origin means the remote repo's git address, and master means sync to the master branch. The command above will sync the local changes to master branch of remote repo with address <https://github.com/example/test.git>.

To sync the remote repo's changes to local machine, Git provides two command: fetch and pull.

They have similar syntax. To use fetch, we can use the following command:

```
git fetch origin master
```

This command will sync the remote repo's code to local machine. The changes will be in origin/master branch, and we can use diff command to see what has changed:

```
git diff origin/master
```

Then we can use merge command to merge the changes in origin/master to the master branch as shown below:

```
git merge origin/master
```

The pull command is equal to running fetch and merge command together which means that it will fetch the latest code and merge to local; the command syntax is shown below:

```
git pull origin master
```

The remote repo may still seem like an abstract concept to you. It's totally fine, we will get to this again in Chap. 15.

## 12.10 Summary and Comment

Are you excited about what you learned in this chapter? In this chapter, we started with an empty project, and at the end of the chapter, we created an app with rich functionality and beautiful UI. The Material Design effects were implemented with Material lib, AndroidX, and some other open-source project.

Here the Material Design topics are more developer oriented. However, Material Design principles and guidelines are more important, and of course, this should be

the focus of designers. If you're interested in it, please refer to the Material Design official website: <https://material.io/>.

In this chapter's Kotlin Class, we did not introduce any new topics but flexibly used what we have learned so far to create three util methods. Hope this can enlighten your mind to think more about using Kotlin features creatively.

In the Git Time section, we discussed some more advanced topics in Git, and now you should have some understanding around branch and remote repo.

After 12 chapters, you should have a deep understanding of Android app development. But to create an excellent app, you need to know how to architect the code base. This is what we will discuss in next chapter—Jetpack.

# Chapter 13

## High-Quality Developing Components: Exploring Jetpack



Up to now, you should be able to build Android App independently. However, there is a huge gap between making an app and an app that has quality codebase and good architecture.

For a long time, there was no official guideline for Android projects. As long as the code is working, you have the freedom to choose how to implement. However, as developers have different understanding of Android and programing, the quality of the code varies.

As there was no official guide for writing code, some third-party architectures got introduced in Android, such as MVP, MVVM, and so on. These architectures improved the readability, maintainability, and quality of the codebase and gradually became mainstream architecture options.

Google heard this and released official architecture components library—Architecture Components to help developers write high-quality code with good architecture. In 2018, Google released Jetpack which is a suite of libraries and guidelines, and Architecture Components became part of Jetpack. Of course, Jetpack did not stay there and kept evolving and adding more components in it.

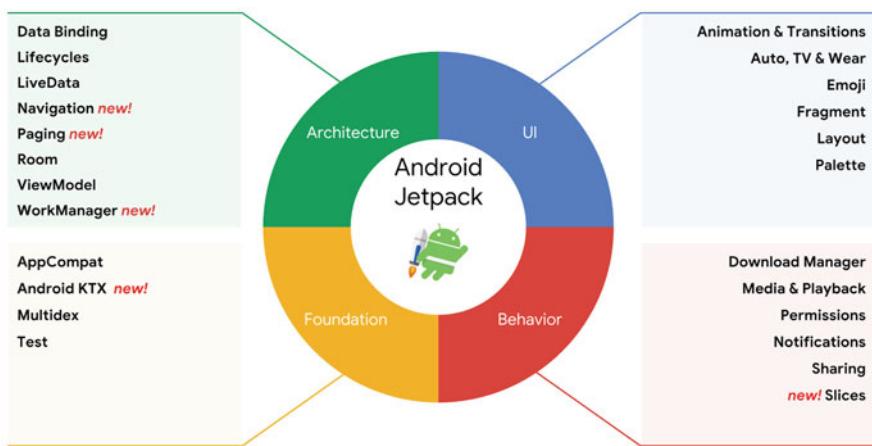
In this chapter, we will cover some important topics in Jetpack.

### 13.1 Introduction to Jetpack

Jetpack is a suite of libraries and guidelines and can help us write more succinct code, simplify the development, and reduce boilerplate code. Most of the components in Jetpack have no dependency on Android versions which means that most of them are defined in AndroidX and are backward compatible.

The structure of Jetpack is as shown in Fig. 13.1.

As you can tell from the figure above, Jetpack has lots of components which can mainly be divided into architecture, UI, behavior, and foundation. We have covered some of them like notification, permissions, and fragment.



**Fig. 13.1** Jetpack structure

Apparently, we cannot cover everything in Jetpack which will be a massive project. Among them, we care architecture the most. The recommended architecture is MVVM, and a lot of the architecture components in Jetpack is designed for MVVM. We will cover the main architecture components and will cover MVVM in Chap. 15. Create JetpackTest and let's begin.

## 13.2 ViewModel

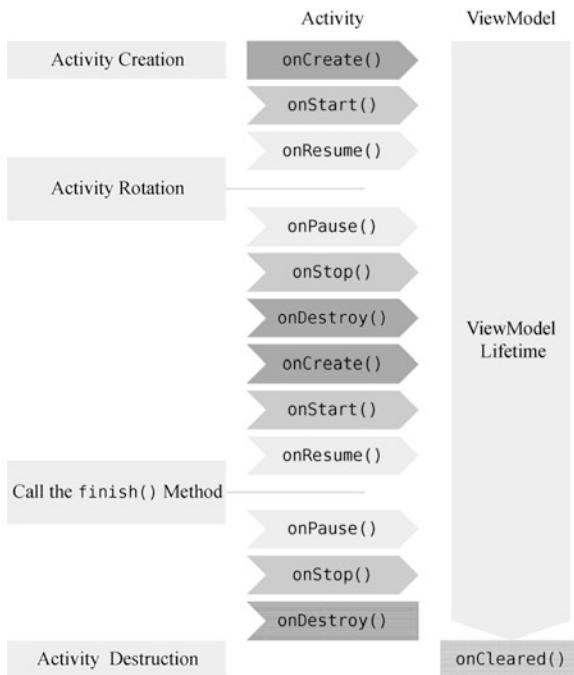
ViewModel is one of the most important components in Jetpack. With the conventional development pattern, Activity will be responsible for business logic and UI display and even handle network callbacks; thus MVP and MVVM were introduced to Android to solve this problem. It is fine to have all these logic inside Activity; however, it will not work for large project as codebase will become heavy and not maintainable due to tight coupling.

ViewModel has the data that are related to UI which used to be in Activity. This means that all the data that will be displayed should have its variable in ViewModel instead of Activity to reduce the code inside Activity.

Activity will be recreated when phone rotates, and the data inside Activity will get lost without proper saving and recovering handling. However, ViewModel has different lifecycle and will not be recreated when phone rotates. It will get destroyed only when Activity is getting destroyed. Thus, if data is in the ViewModel, even after rotating the screen, the data displayed in the screen will sustain. The lifecycle of ViewModel is as shown in Fig. 13.2.

Next, I will discuss how to use ViewModel by using a simplified counter as an example.

**Fig. 13.2** ViewModel  
LifeCycle



### 13.2.1 ViewModel Basics

Since most of components in Jetpack are in AndroidX, some of the frequently used Jetpack components will be included in the dependency when creating the Android project. But ViewModel is different, and we need to add the following dependency to app/build.gradle:

```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-extensions:2.1.0"
}
```

A recommended pattern is to create ViewModel for every Activity and fragment. Thus let us create MainViewModel for MainActivity that inherits ViewModel as shown in code below:

```
class MainViewModel : ViewModel() { }
```

As aforementioned, all the UI-related data should be inside ViewModel; thus, in order to implement a counter, we need to add counter variable in ViewModel as shown in code below:

```
class MainViewModel : ViewModel() {  
  
    var counter = 0  
  
}
```

Next, let us add a button which will increase the counter by one and display the latest value in the screen. Edit activity\_main.xml as shown in code below:

```
<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/infoText"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_gravity="center_horizontal"  
        android:textSize="32sp"/>  
  
    <Button  
        android:id="@+id/plusOneBtn"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_gravity="center_horizontal"  
        android:text="Plus One"/>  
  
</LinearLayout>
```

This layout simply adds a TextView to display the current value of the counter and a button to add the counter by one.

Next, implement the logic of counter by editing MainActivity as shown in code below:

```
class MainActivity : AppCompatActivity() {  
  
    lateinit var viewModel: MainViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        viewModel = ViewModelProviders.of(this).get(MainViewModel::  
class.java)  
        plusOneBtn.setOnClickListener {  
            viewModel.counter++  
            refreshCounter()  
        }  
    }  
}
```

```
        }
        refreshCounter()
    }

    private fun refreshCounter() {
        infoText.text = viewModel.counter.toString()
    }

}
```

Notice that we should never try to create ViewModel directly but has to get the instance of ViewModel by using ViewModelProviders with the following syntax:

```
ViewModelProviders.of(<Activity or Fragment instance>)
    .get(< corresponding ViewModel>::class.java)
```

This is because ViewModel has its own lifecycle which is longer than the Activity. If we create an instance of ViewModel in onCreate(), then every time onCreate() is called, a new ViewModel will be created which means that data cannot be persisted during screen rotation.

The rest of the code should be easy to understand. The refreshCounter() method will display the current value, and clicking the button will increase the counter by one, and refreshCounter() will be called to reflect the latest value.

Run the app, and the result should be as shown in Fig. 13.3.

Click “Plus One” button, and the value should increase as shown in Fig. 13.4.

You can rotate the screen from the toolbar at the side, and you should find that although Activity got recreated by the value of the counter was not lost as shown in Fig. 13.5.

This simple example demonstrates the fundamental use of ViewModel; however, there are some situations the above pattern is not sufficient, and let us see some more advanced use of ViewModel.

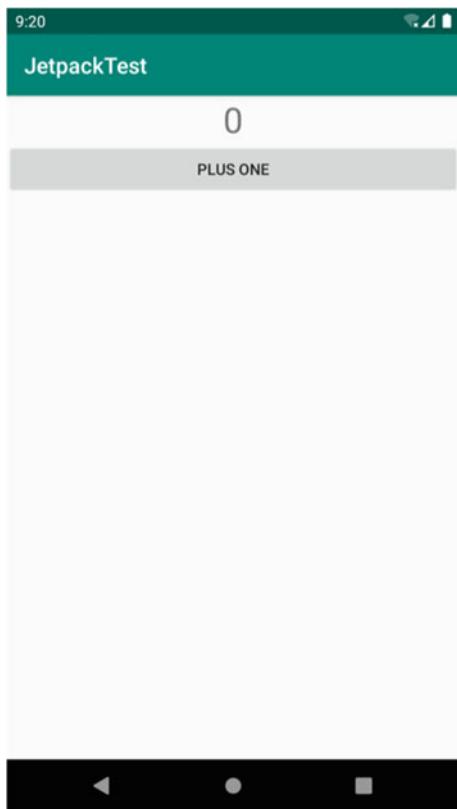
### 13.2.2 Pass Param to ViewModel

The MainViewModel has no param in the constructor. What if we need to pass some params to the constructor? Since all the instances of ViewModel are acquired through ViewModelProvider, there is no way we can pass param directly to ViewModel.

We can use ViewModelProvider.Factory to solve this. Let us use an example to demonstrate.

The current counter can persist data during screen rotation but not restarting the app. Let's upgrade the app and allow data persistence even after restarting the app.

**Fig. 13.3** Counter initial screen



To do so, we need to save the current data when exiting the app and read the data when reopening the app and pass it to MainViewModel. Thus, edit MainViewModel as shown in code below:

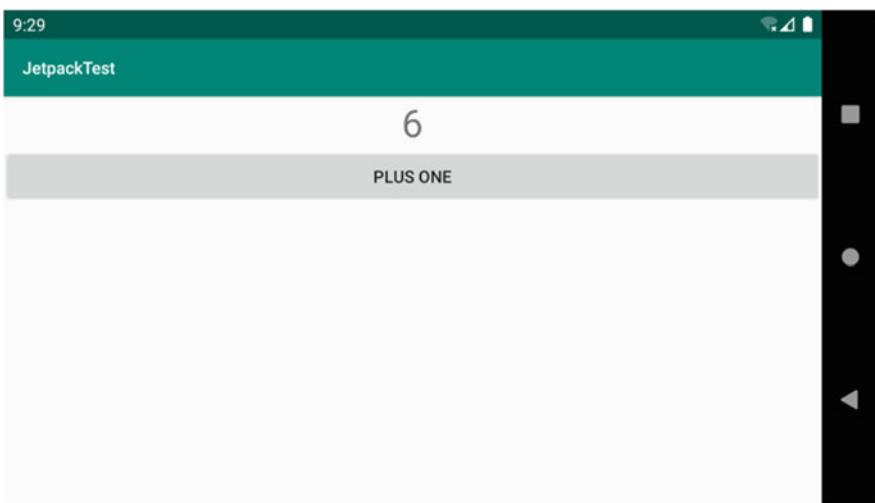
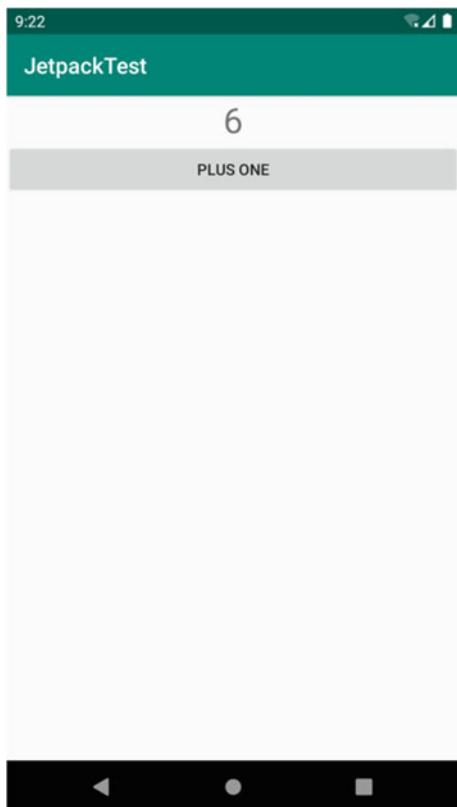
```
class MainViewModel(countReserved: Int) : ViewModel() {  
    var counter = countReserved  
}
```

The countReserved param will have the saved counter value and initialize counter with this value during initialization.

The next question is how to pass data to the constructor of MainViewModel through ViewModelProvider.Factory.

Create MainViewModelFactory class that implements ViewModelProvider.Factory interface as shown in code below:

**Fig. 13.4** Increasing counter value



**Fig. 13.5** Data in ViewModel persists during screen rotation

```

class MainViewModelFactory(private val countReserved: Int) :
ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>) : T {
        return MainViewModel(countReserved) as T
    }

}

```

The constructor of MainViewModelFactory takes countReserved param of Int type. MainViewModelFactory has to implement create(), and inside this method MainViewModel instance is created with argument countReserved. The reason why MainViewModel instance can be created here is because create() is not associated with Activity lifecycle; thus the aforementioned problem does not exist in create().

We also need to add a Clear button for users to reset the counter value to zero. Edit activity\_main.xml as shown in code below:

```

<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/clearBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="Clear"/>
</LinearLayout>

```

Edit MainActivity to implement the related logic as shown below:

```

class MainActivity : AppCompatActivity() {

    lateinit var viewModel: MainViewModel
    lateinit var sp: SharedPreferences

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        sp = getPreferences(Context.MODE_PRIVATE)
        val countReserved = sp.getInt("count_reserved", 0)
        viewModel = ViewModelProviders.of(this, MainViewModelFactory
(countReserved))
            .get(MainViewModel::class.java)
        ...
        clearBtn.setOnClickListener {
            viewModel.counter = 0
            refreshCounter()
        }
    }
}

```

```
        }
        refreshCounter()
    }

    override fun onPause() {
        super.onPause()
        sp.edit {
            putInt("count_reserved", viewModel.counter)
        }
    }
    ...
}
```

In `onCreate()`, we try to read the counter value saved previously from `SharedPreferences` instance with default value zero. The `of()` method of `ViewModelProviders` takes a param of `MainViewModelFactory` type whose constructor takes the value of read from `SharedPreferences` instance. Notice that this step is important because this is the only way to pass the countervalue to the constructor of the `MainViewModel`.

The rest of the code is simple. The counter value gets reset in “Clear” button click event code and saved in `onPause()` which ensures that whether app is killed or backgrounded, the value of counter is not lost.

Run the app again and click “Plus One” button a few times and exit the app and reopen; you will find that counter value will persist, as shown in Fig. 13.6.

Only after clicking “Clear” will the value of the counter reset to zero.

These are important topics in `ViewModel`; next, let us take a look at another important component in Jetpack—Lifecycles.

### 13.3 Lifecycles

In Android apps, a lot of the times we need to be lifecycle aware. For instance, in a certain screen, a network is sent, but before the response is received, this screen may already have been closed, and then the result should not be processed. Thus we need to be aware of Activity’s lifecycle to write proper code.

Inside Activity, it is easy to get the current state of the Activity lifecycle, and the problem is how to get the state in a non-Activity class?

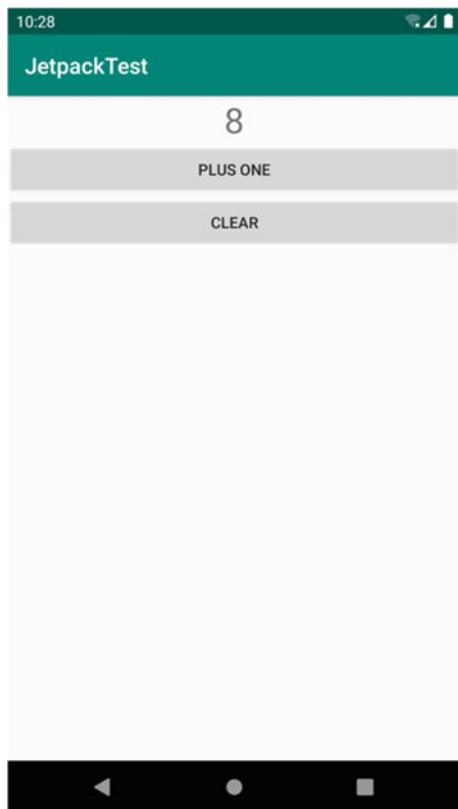
There are many solutions to this problem, for example, embed a hidden Fragment in Activity to get the lifecycle state or manually creating listeners for lifecycle events etc.

Here is an example of how to use listeners to be lifecycle aware.

```
class MyObserver {

    fun activityStart() {
    }
```

**Fig. 13.6** Counter value persists



```
fun activityStop() {  
}  
  
}  
  
class MainActivity : AppCompatActivity() {  
  
    lateinit var observer: MyObserver  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        observer = MyObserver()  
    }  
  
    override fun onStart() {  
        super.onStart()  
        observer.activityStart()  
    }  
  
    override fun onStop() {  
        super.onStop()  
    }  
}
```

```
    observer.activityStop()
}
}
```

In order to be lifecycle aware, the code above overrides the corresponding lifecycle events inside the MainActivity and then notify MyObserver. This solution works but is not an elegant solution due to the extra logic in Activity.

Lifecycle component is designed to solve this problem and can let any class to be lifecycle aware without redundant code in Activity.

Let us use an example to demonstrate how to use Lifecycles. Create MyObserver class that implements LifecycleObserver as shown below:

```
class MyObserver : LifecycleObserver {
```

LifecycleObserver does not have method that requires implementation; thus we just need to declare it.

We can define any method inside MyObserver. To be lifecycle aware, we need to add extra annotation. Define activityStart() and activityStop() as shown in code below:

```
class MyObserver : LifecycleObserver {

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    fun activityStart() {
        Log.d("MyObserver", "activityStart")
    }

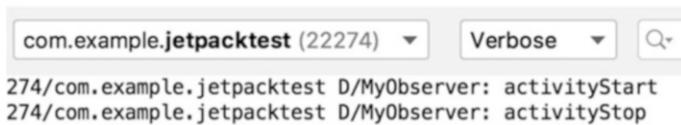
    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    fun activityStop() {
        Log.d("MyObserver", "activityStop")
    }
}
```

The @OnLifecycleEvent annotation is added to these two methods with the corresponding lifecycle events. There are seven of them: ON\_CREATE, ON\_START, ON\_RESUME, ON\_PAUSE, ON\_STOP, and ON\_DESTROY, and each of them has its own corresponding lifecycle events. There is also ON\_ANY which can match any lifecycle events.

With the above code, activityStart() and activityStop() should be triggered when onStart() and onStop() get triggered, respectively.

But it is not working yet. Because when lifecycle events happen, MyObserver is not triggered, and we do not want to add the logic one by one as before.

That is where LifecycleOwner can help. With the following syntax, MyObserver will be notified:



**Fig. 13.7** Logs in MyObserver

```
lifecycleOwner.lifecycle.addObserver(MyObserver())
```

An instance of Lifecycle can be acquired by `getLifecycle()` of LifecycleOwner. Then we just need to pass MyObserver instance to its `addObserver()` to allow the observer observe the LifecycleOwner's lifecycle.

Then what is LifecycleOwner? How to get its instance?

Of course, we can implement our own LifecycleOwner. But under most cases this is not necessary. This is because for Activity that inherits AppCompatActivity and Fragment that inherits android.fragment.app.Fragment, they are instances of LifecycleOwner. This means we can have the following code in MainActivity:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        ...
        lifecycle.addObserver(MyObserver())
    }
    ...
}
```

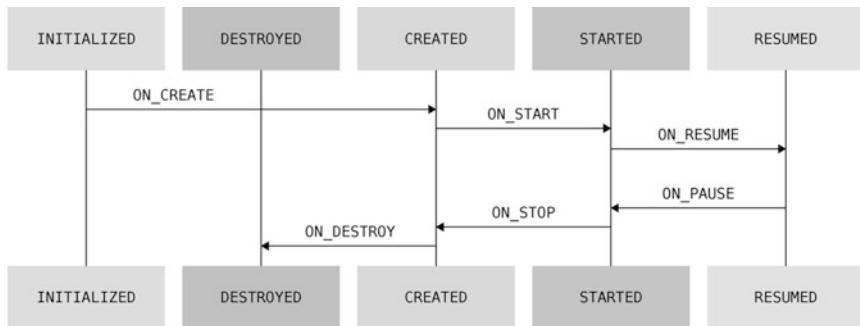
Now MyObserver can listen to Activity lifecycle events. Notice that we are using Activity as examples, but everything mentioned above can be applied to Fragment.

Run the app again, and `activityStart` will be logged. If you press Home button or Back button, `activityStop` will also be logged, as shown in Fig. 13.7.

This is how Lifecycles is usually used. MyObserver can listen to Activity lifecycle events now but cannot proactively get the lifecycle state. To solve this problem, we just need to pass the Lifecycle instance to the MyObserver constructor as shown below:

```
class MyObserver(val lifecycle: Lifecycle) : LifecycleObserver {
    ...
}
```

With the Lifecycle instance, we can get the current lifecycle state anywhere by calling `lifecycle.currentState`. This method returns an enum type which has five values: INITIALIZED, DESTROYED, CREATED, STARTED, and RESUMED. The corresponding lifecycle events of these states can be shown in Fig. 13.8.



**Fig. 13.8** Activity Lifecycle states and corresponding events

This means that when state is CREATED, then onCreate() has been called but onStart() is not called. When state is STARTED, onStart() has been called, but onResume() has not been called yet and so on so forth.

That is it for Lifecycles. Now we have covered two components in Jetpack already. However, these two components are relatively independent to each other, and in order to use them together, let us take a look at another important component in Jetpack—LiveData.

## 13.4 LiveData

LiveData is an observable component in Jetpack that can hold any type of data and notify the observer when data changes. LiveData can be used independently but is frequently used together with ViewModel. Let us see how to use LiveData through examples.

### 13.4.1 LiveData Basics

There are still issues in the simple counter we just finished. The current logic is that every click on the “Plus One” button will increase the counter value in ViewModel by one, and then this value will be reflected in the screen. This works in single thread environment. However, if ViewModel starts a worker thread to run some time consuming task, then the new data may not be reflected.

The problem is that Activity manually gets data from ViewModel instead of ViewModel notifies the data change to Activity.

You might think that we can simply pass an instance of Activity to ViewModel and then call method inside Activity. This is a bad solution because ViewModel’s

lifecycle is longer than Activity lifecycle, and if ViewModel has reference to Activity instance, then Activity will not be recycled which can cause memory leak.

The solution is to use LiveData. As mentioned before, LiveData can hold any type of data and notify the observer when data changes. This means that we should wrap counter value with LiveData and observe it in Activity so that data change can notify Activity.

To do so, edit MainViewModel as shown in code below:

```
class MainViewModel(countReserved: Int) : ViewModel() {

    val counter = MutableLiveData<Int>()

    init {
        counter.value = countReserved
    }

    fun plusOne() {
        val count = counter.value ?: 0
        counter.value = count + 1
    }

    fun clear() {
        counter.value = 0
    }
}
```

Now counter variable is an object of MutableLiveData with generic type of Int. MutableLiveData mainly has three methods to read and write data: getValue(), setValue(), and postValue(). They are self-explanatory but notice that setValue() can only be used in main thread while postValue() is used in worker thread. The above code applies syntax sugar to replace getValue() and setValue().

The init code block sets value to counter which ensures that the previously saved value can be recovered. The plusOne() method first gets the data in counter and then increases it by one and sets to counter. Notice that the data obtained by calling the getValue() method of LiveData is potentially empty, so a ?: operator, when the obtained data is empty, it will use 0 as the default count.

This is what we need to change for MainViewModel in order to use LiveData. Next, edit MainActivity as shown in code below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        plusOneBtn.setOnClickListener {
            viewModel.plusOne()
        }
        clearBtn.setOnClickListener {
            ...
        }
    }
}
```

```
        viewModel.clear()
    }
    viewModel.counter.observe(this, Observer { count ->
        infoText.text = count.toString()
    })
}

override fun onPause() {
    super.onPause()
    sp.edit {
        putInt("count_reserved", viewModel.counter.value ?: 0)
    }
}
```

It is trivial that we need to call `plusOne()` in click event of “Plus One” button and call `clear()` in click event of “Clear” button. Also we need to make corresponding change to saving data in `onPause()`.

The `observe()` method of `viewModel.counter` is the key method to observe the data change. Now counter becomes an object of `LiveData`, and any `LiveData` object can call its `observe()` method to observe the data change. This method takes two params: The first is an object of `LifecycleOwner`, and since `Activity` itself is an object of `LifecycleOwner`, thus this is passed in; the second param is an `Observer` interface, and when counter changes, the callback will be triggered, and we just need to reflect the latest value to UI.

Run the app again, and you will find it is working as before. However, current code is more proper, and there is no need to worry if a new thread will be started in `ViewModel`. Notice that if we need to set data for `LiveData` in worker thread, we have to call `postValue()` instead of `setValue()` to prevent crash.

You probably can notice that `Observer` is a functional interface with only one abstract method `onChanged()`. Then why not apply the functional API syntax as we discussed in Sect. 2.6.3 here?

This is a special case because `observe()` takes a `LifecycleOwner` param which is also a functional interface. When a Java method takes two functional interface params, either they both use the functional syntax, or both do not use functional syntax. Since the first argument is `this`, the second param cannot use the functional syntax.

However, in Google I/O 2019, Android team announced Kotlin First and promised to provide more Kotlin-oriented APIs in Jetpack. Among them, `lifecycle-livedata-ktx` is lib that is oriented to Kotlin and added syntax extension to `observe()` in v2.2.0. By adding the following dependency in `app/build.gradle`:

```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.2.0"
}
```

We can use observe() with the following syntax:

```
viewModel.counter.observe(this) { count ->
    infoText.text = count.toString()
}
```

Our example code works but still is not the recommended way to use LiveData because counter is accessible from other classes. This means that other classes can mutate counter which introduces some risk.

The recommended pattern is to only make LiveData read-only to other classes. Then classes other than ViewModel will only be able to observe the data change but cannot set value to LiveData. To do so, we can edit MainViewModel as shown in code below:

```
class MainViewModel(countReserved: Int) : ViewModel() {

    val counter: LiveData<Int>
        get() = _counter

    private val _counter = MutableLiveData<Int>()

    init {
        _counter.value = countReserved
    }

    fun plusOne() {
        val count = _counter.value ?: 0
        _counter.value = count + 1
    }

    fun clear() {
        _counter.value = 0
    }
}
```

The counter variable is renamed as \_counter with private modifier which makes it inaccessible to other classes. A new counter variable is defined to be immutable LiveData, and its get() method returns the value of \_counter.

Then when other class accesses counter, it actually gets the instance of \_counter and cannot mutate counter which ensures ViewModel's data encapsulation. This is the officially recommended pattern.

### ***13.4.2 map and switchMap***

The above pattern to use LiveData can satisfy most of the cases. However, as the project is getting more complex, there may be some requirements the above cannot

meet. In order to handle different requirements, LiveData provides two methods to transform data: map() and switchMap(). Let us take a look at these two methods.

The map() method will transform LiveData that contains the data and the LiveData that is only used to observe data. Let me use an example to demonstrate the use case.

Assume we have User class that has user name and age. It is defined as follows:

```
data class User(var firstName: String, var lastName: String, var age: Int)
```

We can create a LiveData to wrap User data in ViewModel as shown below:

```
class MainViewModel(countReserved: Int) : ViewModel() {  
  
    val userLiveData = MutableLiveData<User>()  
    ...  
}
```

This is the same as previous pattern. However, if MainActivity only needs to display the user name and has no need to display user age, then it is not a good idea to expose all fields of this LiveData.

Map() is introduced to solve this problem. It can convert the LiveData of User type to other type of LiveData; below is an example of how to use it:

```
class MainViewModel(countReserved: Int) : ViewModel() {  
  
    private val userLiveData = MutableLiveData<User>()  
  
    val userName: LiveData<String> = Transformations.map(userLiveData)  
    { user ->  
        "${user.firstName} ${user.lastName}"  
    }  
    ...  
}
```

The map() method transforms the original LiveData so that only the used data is exposed. This method takes two params: The first param is the original LiveData object, and the second param is a transformation function. For the transformation function, we simply create a string from the User object that has the user name.

Also, userLiveData has private modifier which can prevent it for being accessible from other classes which just needs to observe user name. When userLiveData changes, map() can listen to the change and execute the transformation function and notify the user name observer with the refreshed data.

This is how map() is used which should be easy to understand.

The switchMap() is used only in limited scenario but can be used more frequently.

So far, all of the LiveData objects are created in ViewModel. But this is not always possible in real project, as some LiveData objects in the ViewModel can come from some other methods.

For instance, assume we have Repository singleton with code below:

```
object Repository {
    fun getUser(userId: String): LiveData<User> {
        val liveData = MutableLiveData<User>()
        liveData.value = User(userId, userId, 0)
        return liveData
    }
}
```

This Repository class has `getUser()` method that takes a `userId` argument. In real application, we may use `userId` to query the server or local database for the corresponding `User` object, but for demonstration purpose, we simply create a new `User` object with the `userId`.

Notice that every call to `getUser()` will return a new `LiveData` object that contains `User` data.

In `MainViewModel`, we also create `getUser()` that will call `getUser()` in Repository to get the `LiveData` object as shown in code below:

```
class MainViewModel(countReserved: Int) : ViewModel() {
    ...
    fun getUser(userId: String): LiveData<User> {
        return Repository.getUser(userId)
    }
}
```

Then how to observe `LiveData` in Activity? As `getUser()` returns a `LiveData` object, can we use the following code in Activity?

```
viewModel.getUser(userId).observe(this) { user ->
}
```

This does not work because `getUser()` returns a brand-new `LiveData` object, and the above code will observe the old `LiveData` object with stale value. This means that `LiveData` is not observable in this case.

However, `switchMap()` can help solve this problem. As aforementioned, its use case is fixed: if `LiveData` object is not created inside the `ViewModel`, then `switchMap()` can transform this `LiveData` to another observable `LiveData` object.

Edit `MainViewModel` as shown in code below:

```
class MainViewModel(countReserved: Int) : ViewModel() {  
    ...  
    private val userIdLiveData = MutableLiveData<String>()  
  
    val user: LiveData<User> = Transformations.switchMap  
(userIdLiveData) { userId ->  
        Repository.getUser(userId)  
    }  
  
    fun getUser(userId: String) {  
        userIdLiveData.value = userId  
    }  
}
```

The userIdLiveData variable is defined to observe the change of userId and switchMap() switch this to another observable LiveData object.

The switchMap() method takes two params: The first param is the userIdLiveData which switchMap() will observe, and the second param is a transformation function, and notice that we have to return a LiveData object in this function because switchMap() is supposed to transform the LiveData object returned from the transformation function to another observable LiveData object. Then we just need to call getUser() in Repository to get LiveData object and return it.

So the whole process is like this: first, when getUser() of MainViewModel is called, only the userId will be set to userIdLiveData. Once userIdLiveData changes, then switchMap() that is observing the userIdLiveData will be called, and the transformation function will be called. The transformation function will call Repository.getUser() to get the real user data. At the same time, switchMap() transforms the LiveData object returned by Repository.getUser() to an observable LiveData object which Activity will observe.

To test it, edit activity\_main.xml and add “Get User” button in the screen as shown in code below:

```
<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/getUserBtn"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:layout_gravity="center_horizontal"  
        android:text="Get User"/>  
</LinearLayout>
```

Then edit MainActivity as show below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        getUserBtn.setOnClickListener {
            val userId = (0..10000).random().toString()
            viewModel.getUser(userId)
        }
        viewModel.user.observe(this, Observer { user ->
            infoText.text = user.firstName
        })
    }
    ...
}
```

The “Get User” button click event handler uses a random number as userId and call getUser() of MainViewModel to get the user information which has no return value. After data is acquired, the observe() method on the observable LiveData object will be notified, and the user name will be displayed on the screen.

Run the app again and keep clicking “Get User” button, and you should find that the number in the screen keeps changing as shown in Fig. 13.9. This because the userId is random and also means that switchMap() indeed is working.

In the above example, we have userId argument to observe, but what if there is no argument?

We can still use the pattern with some light changes. We just need to create an empty LiveData object as shown in code below:

```
class MyViewModel : ViewModel() {

    private val refreshLiveData = MutableLiveData<Any?>()

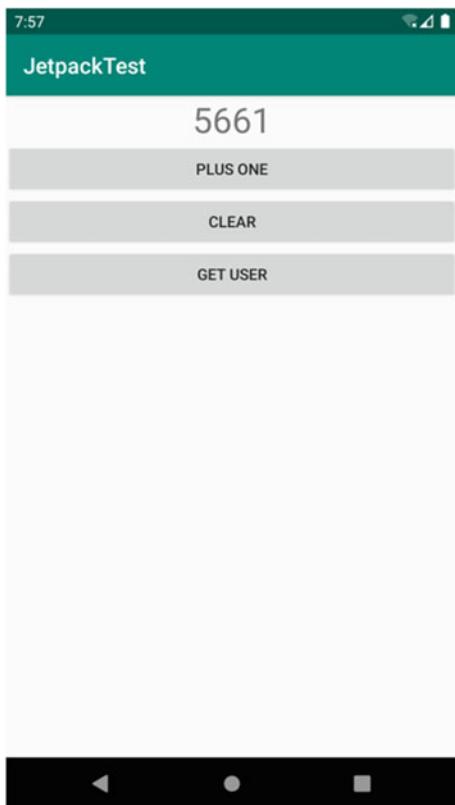
    val refreshResult = Transformations.switchMap(refreshLiveData) {
        Repository.refresh()
    }

    fun refresh() {
        refreshLiveData.value = refreshLiveData.value
    }
}
```

Here we define refresh() to be a parameterless method. As you can see, here we have defined a refresh() method without parameters, and correspondingly defined a refreshLiveData, but it does not need to specify the specific type of data contained, so here we specify the generic type of LiveData to Any?.

The key change here is that we simply get the refreshLiveData’s original value (by default it is empty) and then set it to refreshLiveData, and this can trigger data change. LiveData does not check if the new data has the same value as the previous

**Fig. 13.9** Random UserId between 0 and 9999



value; as long as setValue() or postValue() is triggered, data change event will be triggered.

Then, Activity can just observe LiveData object refreshResult, and once refresh() is called, the callback of observer will get the triggered with the latest data.

So far we have only been using LiveData and ViewModel together, and Lifecycles have no involvement yet.

But this is not the case as it depends on Lifecycles such that LiveData can be the bridge between Activity and ViewModel without risk of memory leak.

Internally, LiveData uses Lifecycles to be lifecycle aware and releases the reference to Activity when it gets destroyed thus eliminating memory leak.

Also, in order to save energy, when Activity is in invisible state (screen turned off, or covered by other Activity, etc.), the observer will not be notified with the data change in LiveData. Only when the Activity becomes visible again can the observer receive the latest data change. This performance optimization also depends on Lifecycles.

If LiveData changes multiple times while Activity is in the invisible state, then when Activity becomes visible again, only the latest data will be notified to observer, and all the data before that are considered expired and will be discarded.

That is it for important topics in LiveData.

## 13.5 Room

In Chap. 7, we discussed how to use SQLite database and used native APIs to do CRUD. These APIs are easy to use but can become messy when used in large-scale project unless with proper encapsulation. A lot of ORM frameworks have been designed to solve this problem.

Object Relational Mapping is the mapping between object-oriented language and relational database.

ORM frameworks allow developers to interact with database with object-oriented thinking pattern and eliminate SQL use under most scenarios and at the same time without worrying the messy logic of database operation.

The Android official ORM framework is Room which is also a component in Jetpack.

### 13.5.1 Use Room to CRUD

Room mainly consists of Entity, Dao, and Database which has its own responsibility.

- Entity. It is used to define the class that encapsulates the data, and each class has its corresponding table in database with the columns auto-generated by the fields in the class.
- Dao. It is short for data access object which encapsulates the operations on database. Business logic interacts with Dao layer instead of interacting directly with the database.
- Database. It defines the key information of the database including the version number and entity classes and provides instance of Dao.

Let us use examples to demonstrate what do they really mean.

Let us work on top of JetpackTest. To use Room, we need to add the following dependencies in app/build.gradle:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'

dependencies {
    ...
    implementation "androidx.room:room-runtime:2.1.0"
    kapt "androidx.room:room-compiler:2.1.0"
}
```

Besides the two Room dependencies, kotlin-kapt plugin is added because Room will generate code based on the annotation which has dependency on this plugin.

Notice that kapt is only available in Kotlin project and Java project needs to use annotationProcessor.

First let us define Entity class. We can use the existing User class as entity class. However, it is recommended that every entity class should have id field and set it as primary key. Thus we can edit User class as shown in code below:

```
@Entity
data class User(var firstName: String, var lastName: String, var age: Int) {

    @PrimaryKey(autoGenerate = true)
    var id: Long = 0

}
```

The `@Entity` annotation declares User an entity class, and the `@PrimaryKey` annotations makes id the primary key. The auto-generated param is set to true so that the primary key's value is auto-generated.

Here we only defined one entity for simplicity, and in real applications, you probably need to define multiple entity classes which have similar definition pattern and may need some connection between the entity classes.

Next, define Dao which is the most important part of Room as all the database access operations are encapsulated here.

In Chap. 7, we learned that all database operations belong to CRUD even though business logic varies. Dao makes sure that all the business logic can be done and only interact with Dao instead of the database.

To use Dao, let us create UserDao interface first. Notice that this has to be an interface which is similar to Retrofit. Then define the interface as shown in code below:

```
@Dao
interface UserDao {

    @Insert
    fun insertUser(user: User): Long

    @Update
    fun updateUser(newUser: User)

    @Query("select * from User")
    fun loadAllUsers(): List<User>

    @Query("select * from User where age > :age")
    fun loadUsersOlderThan(age: Int): List<User>

    @Delete
    fun deleteUser(user: User)
```

```

    @Query("delete from User where lastName = :lastName")
    fun deleteUserByLastName(lastName: String): Int
}

```

The `@Dao` annotation makes `UserDao` recognized as a Dao instance. Since Dao encapsulates the database operation logic, thus Room provides `@Insert`, `@Delete`, `@Update`, and `@Query` annotations.

The `@insert` annotation applied to `insertUser()` makes this method insert the `User` argument into database and return the automatically generated primary key id. The `@Update` annotation applied to `updateUser()` makes this method update the `User` argument in the database. The `@Delete` annotation makes `deleteUser()` method delete the `User` argument from database. All of these three methods require no SQL statement.

However, to query data or insert, delete, and update data without entity class, param requires SQL statement. For instance, the `loadAllUsers()` method queries all the users, and if only `@Query` is applied, Room cannot figure out what kind of data this method is going to query; thus `@Query` annotation has to contain the SQL statement. We can pass the param of the method to the SQL statement, for instance, `loadUsersOlderThan()` can query the users that are older than the specified age. Also if nonentity class param is used to insert, update, and delete data, then SQL statement is required, and `@Query` annotation has to be used instead of `@Insert`, `@Delete`, nor `@Update`; refer to `deleteUserByLastName()`.

The methods in the above code covers adding user, updating user data, querying user, and deleting user, and in real application you can follow the same pattern to create methods to meet the business needs.

The requirement to write SQL statement is not friendly, but SQL indeed can meet various business requirements, and Room supports syntax checking at compile time which means that if the SQL statement has syntax error, compiler will point out the syntax instead of throwing error at runtime.

Last part is the defining Database. This part has pattern to follow. We just need to define database version number, entity classes to include, and the Dao instance. Create `AppDatabase.kt` as shown in code below:

```

@Database(version = 1, entities = [User::class])
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

    companion object {

        private var instance: AppDatabase? = null

        @Synchronized
        fun getDatabase(context: Context): AppDatabase {
            instance?.let {

```

```
        return it
    }
    return Room.databaseBuilder(context.applicationContext,
        AppDatabase::class.java, "app_database")
        .build().apply {
    instance = this
}
}
}
```

The `@Database` annotation at the top of the `AppDatabase` specifies the version number and entity class. Between the entity classes, use comma to separate.

`AppDatabase` has to inherit `RoomDatabase` class and declared with `abstract` modifier with corresponding abstract method to get the Dao instance, for instance, `userDao()` returns an instance of `UserDao`. Only the abstract method is required, and the implementation is done by Room.

Immediately afterwards, we write a single instance pattern in the companion object structure, because in principle there should be only one instance of `AppDatabase` globally. `databaseBuilder()` takes three params. The first param has to be `applicationContext`, and no any other context can be used to prevent memory leak. We will discuss `applicationContext` details in Chap. 14. The second param is the class type of `AppDatabase`, and the third param is the database name. Last, we need to call `build()` and assign the created instance to `instance` variable and return it.

That is everything we need to make Room work, and the next thing is to test the implementation. Edit `activity_main.xml` to add four buttons for CRUD operations:

```
<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/getUserBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Get User"/>

<Button
    android:id="@+id/addDataBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Add Data"/>
```

```
<Button
    android:id="@+id/updateDataBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Update Data"/>

<Button
    android:id="@+id/deleteDataBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Delete Data"/>

<Button
    android:id="@+id/queryDataBtn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="Query Data"/>
</LinearLayout>
```

Then edit MainActivity to add the business logic in the click event code of these four buttons, as shown in code below:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        val userDao = AppDatabase.getDatabase(this).userDao()
        val user1 = User("Tom", "Brady", 40)
        val user2 = User("Tom", "Hanks", 63)
        addDataBtn.setOnClickListener {
            thread {
                userDao.insertUser(user1)
                userDao.insertUser(user2)
            }
        }
        updateDataBtn.setOnClickListener {
            thread {
                user1.age = 42
                userDao.updateUser(user1)
            }
        }
        deleteDataBtn.setOnClickListener {
            thread {
                userDao.deleteUserByLastName("Hanks")
            }
        }
        queryDataBtn.setOnClickListener {
            thread {
                for (user in userDao.loadAllUsers()) {
```

```
        Log.d("MainActivity", user.toString())
    }
}
}
...
}
```

The logic here is simple. First an instance of UserDao is acquired; two User objects are created. Then in the “Add Data” button, click event handler; insertUser() is called to insert these two User objects into database and assign the returned id value to the original User object. This is required because update and delete operations need the id value.

In the “Update Data” button, click event handler; we set user1’s name to 42 and call updateUser() to update the data in the database. In the “Delete Data” button, click event handler; deleteUserByLastName() is called to delete all the users with lastName Hanks. In the “Query Data” button, click event handler; loadAllUsers() is called to query and log all the users in the database.

Since database operations are time-consuming, Room does not allow them in main thread by default. Thus, we put the operations in the worker thread. To help testing, Room also provides an easier method as shown in code below:

```
Room.databaseBuilder(context.getApplicationContext(), AppDatabase::
class.java, "app_database")
    .allowMainThreadQueries()
    .build()
```

The allowMainThreadQueries() method allows database operations in main thread, but this is only recommended to be used in test environment.

Run the app, and UI should be as shown in Fig. 13.10.

Then click “Add Data” button, then click “Query Data,” and Logcat should show records as shown in Fig. 13.11.

This proves that two user data have been successfully inserted into database.

Next, click “Update Data” button and click “Query Data” button again; the logs in Logcat should be as shown in Fig. 13.12.

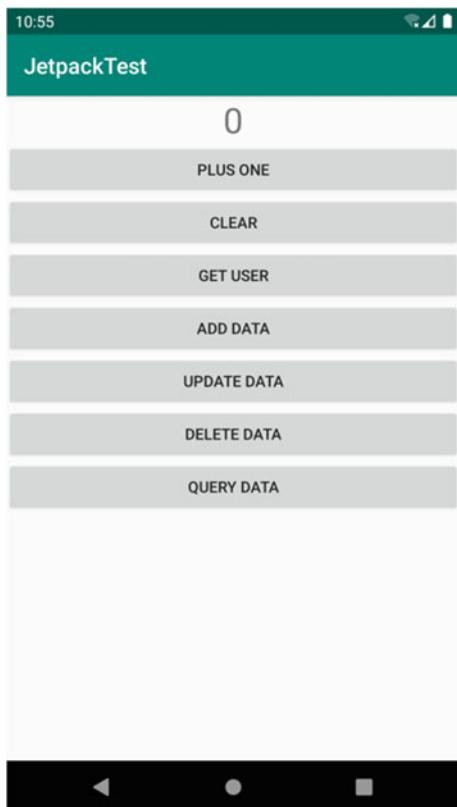
As shown in the figure, the age of the user in the first row has been updated to 42 successfully.

Lastly, click “Delete Data” button and click “Query Data” button again; the logs in Logcat should be as shown in Fig. 13.13.

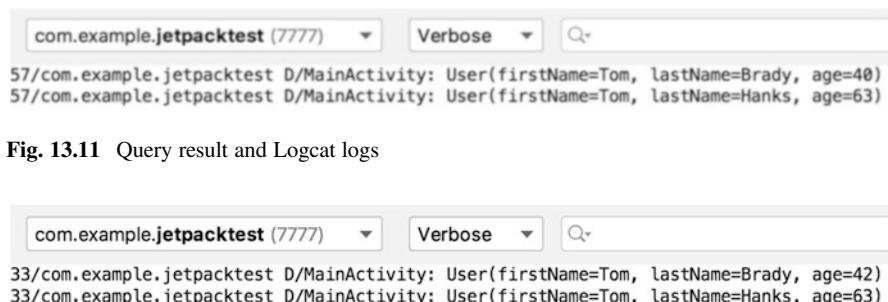
Now there is only one row of data left.

You might think using Room is not easy as using SQLiteDatabase as we need to define Entity, then define Dao, and lastly define Database. However, after these steps, we can write code with object-oriented pattern, and there is no need to consider the implementation detail of the database. In large-scale projects, Room can help you create more reasonable architecture and more maintainable; thus it becomes the most officially recommended database framework.

**Fig. 13.10** UI with CRUD buttons



**Fig. 13.11** Query result and Logcat logs



**Fig. 13.12** Query updated data



**Fig. 13.13** Query database after deletion

### 13.5.2 Room Database Upgrade

Of course, database mostly will not stay the same and need to be upgraded as the business grows. However, to upgrade database in Room is not simpler as it is with using the native SQLiteDatabase and also requires upgrading logic implementation. My open-source database framework LitePal can automatically upgrade database based on entity changes; you can Google it to see how I did it.

However, if you just want to test how to upgrade without complex business logic changes, Room provides a simple way to do so which is as shown in code below:

```
Room.databaseBuilder(context.getApplicationContext(), AppDatabase::  
class.java, "app_database")  
    .fallbackToDestructiveMigration()  
    .build()
```

The fallbackToDestructiveMigration() method will destroy the current database and recreate a new one when upgrade happens. Of course, the side effect is that all the data in the database will be lost.

This apparently works in test environment, but if this is prod environment and user loses their data, it will be devastating! Thus let us see how to properly upgrade database in Room.

Assume that we need to add a Book table, and then the first thing we need to do is to create a Book entity class as shown in code below:

```
@Entity  
data class Book(var name: String, var pages: Int) {  
  
    @PrimaryKey(autoGenerate = true)  
    var id: Long = 0  
  
}
```

The Book class has primary key id, book name, and page fields. The @Entity annotation makes it an entity class.

Next, create BookDao interface and define APIs inside it:

```
@Dao  
interface BookDao {  
  
    @Insert  
    fun insertBook(book: Book): Long  
  
    @Query("select * from Book")  
    fun loadAllBooks(): List<Book>  
  
}
```

Then edit AppDatabase to implement the upgrade logic as shown below:

```
@Database(version = 2, entities = [User::class, Book::class])
abstract class AppDatabase : RoomDatabase() {

    abstract fun userDao(): UserDao

    abstract fun bookDao(): BookDao

    companion object {

        val MIGRATION_1_2 = object : Migration(1, 2) {
            override fun migrate(database: SupportSQLiteDatabase) {
                database.execSQL("create table Book (id integer primary
                    key autoincrement not null, name text not null,
                    pages integer not null)")
            }
        }

        private var instance: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            instance?.let {
                return it
            }
            return Room.databaseBuilder(context.applicationContext,
                AppDatabase::class.java, "app_database")
                .addMigrations(MIGRATION_1_2)
                .build().apply {
                    instance = this
                }
        }
    }
}
```

The above changes upgrade the version number to 2 and add Book class to the entity class declaration list and then add bookDao() to get the instance of BookDao.

In the code block of companion object, the anonymous implementation of class Migration takes 1 and 2 as arguments which means that when database upgrades from v1 to v2, the code in this anonymous class implementation will be called. Notice the naming pattern here; MIGRATION\_1\_2 can help improve readability. Since we need to add a new table for Book, we need to write the creating table statement in migrate(). Notice that this statement has to match with the Book entity class definition; otherwise Room will throw exception.

The last thing is to add addMigrations() and pass MIGRATION\_1\_2 to it.

Now whenever we need to do any database operation, Room will automatically run the corresponding upgrade logic based on the current version number to make sure that database is up to date.

Not every upgrade needs to create a new table. Maybe we just need to add a new column in the existing table, and for this, we can simply use alter statement.

Assume that we want to add author field in the Book class, then we can add it to Book as shown below:

```
@Entity
data class Book(var name: String, var pages: Int, var author: String) {

    @PrimaryKey(autoGenerate = true)
    var id: Long = 0

}
```

Since entity fields changed, the corresponding database table has to be upgraded. Edit AppDatabase as shown in code below:

```
@Database(version = 3, entities = [User::class, Book::class])
abstract class AppDatabase : RoomDatabase() {

    ...
    companion object {
        ...
        val MIGRATION_2_3 = object : Migration(2, 3) {
            override fun migrate(database: SupportSQLiteDatabase) {
                database.execSQL("alter table Book add column author text not
null
                default 'unknown'")
            }
        }

        private var instance: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            ...
            return Room.databaseBuilder(context.applicationContext,
                AppDatabase::class.java, "app_database")
                .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
                .build().apply {
                    instance = this
                }
            }
        }
    }
}
```

Most of the steps are the same as previously; we update the version number to 3 and create the upgrading logic MIGRATION\_2\_3 and pass it to addMigrations().

It might be hard to get the SQL statement in migration() right at the first time, but no worries, app will crash and notify you the reason after the first operation on the database, and you can just fix based on the trace.

That is for Room, and let us take a look at the last Jetpack component in this chapter—WorkManager.

## 13.6 WorkManager

Android background is a complicated topic, and even I myself am totally clear about the API changes happened for each version of Android OS. In the beginning, Android background was a wild west, and Service had very high priority which was only below Activity, and you could do a lot of hacky things in Service. However, because the background functions are too open, every application wants to take up unlimited background resources, resulting in the phone's memory getting tighter and tighter, consuming power faster and faster, and becoming more and more stuck. To mitigate these issues, almost every new version of Android OS will put more restriction on background permissions.

Some of the important changes are: v4.4 makes the AlarmManager triggering not accurate any more, v5.0 introduced JobScheduler to process background tasks, v6.0 introduced Doze and App Standby mode to reduce the frequency of wake up by background, and v8.0 prohibited background use of Service and only foreground Service is allowed. Of course there are many detail changes not covered here.

The frequent API changes made it difficult for developers to create apps that are compatible in different versions of OS. In order to solve this problem, Google released WorkManager which can handle timing-related tasks. Based on the OS version, it will choose AlarmManager or JobScheduler under the hood and thus make it easier for developers. It also supports periodic tasks and chaining work and so on.

But first, I want to make it clear that WorkManager is different from Service, and there is no direct link between them too. Service is the four main components of Android and will keep running in the background until being destroyed. WorkManager is just a tool to process time-related tasks which can ensure that even app exits or phone restarts, the registered task can still get executed. This makes WorkManager a perfect tool to run periodic tasks that need to interact with server, for instance, periodic syncing data with server and so on.

Notice that the periodic tasks registered with WorkManager is not guaranteed to run on time, and this is not a bug but expected behavior. This is because in order to reduce energy consumption, it is possible that tasks that will be triggered around the same time will be triggered at the same time to reduce the frequency of waking up CPU and efficiently increase battery life. Now, let us take a look at how to use WorkManager.

### 13.6.1 WorkManager Basics

To use WorkManager, add the following dependency in app/build.gradle.

```
dependencies {  
    ...  
    implementation "androidx.work:work-runtime:2.2.0"  
}
```

That is all we need for preparation work.

We mainly need the following three steps to use WorkManager:

1. Define a background task with business logic
2. Configure the condition and restriction to run this task and create background task request
3. Enqueue the background task with enqueue() of WorkManager, and OS will run the task at the proper time

Let's follow the steps here to implement an example.

To define a background task, let us create SimpleWorker class as an example, as shown below:

```
class SimpleWorker(context: Context, params: WorkerParameters) :  
Worker(context, params) {  
  
    override fun doWork(): Result {  
        Log.d("SimpleWorker", "do work in SimpleWorker")  
        return Result.success()  
    }  
}
```

The pattern is to let the class inherit Worker class and call its only constructor and override the doWork() method with the background business logic.

The doWork() method does not run on main thread; thus it is safe to put time-consuming logic here. We simply log a string here. This method returns an object of Result, and based on the status, we can return Result.success() or Result.failure(). We can also return Result.retry() which is also a failure state, but we can use it together with setBackoffCriteria() of WorkRequest.builder to run the task again which we will discuss later.

That is all we need to define background task. The next step is to configure the background task conditions and constraints.

There are a lot of configurations we can set, and here I will demonstrate the basic configurations as shown in code below:

```
val request = OneTimeWorkRequest.Builder(SimpleWorker::class.  
java).build()
```



**Fig. 13.14** Logcat logs for SimpleWorker

We just need to pass the background task class to OneTimeWorkRequest.Builder's constructor and call build() to finish building.

OneTimeWorkRequest.Builder is the subclass of WorkRequest.Builder and is used to construct the background task request that will only run once. Another subclass PeriodicWorkRequest.Builder as its name implies can create periodic background task request. However, in order to reduce energy consumption, PeriodicWorkRequest.builder's period param cannot be less than 15 min; below is an example of this builder.

```
val request = PeriodicWorkRequest.Builder(SimpleWorker::class.java, 15,
    TimeUnit.MINUTES).build()
```

The last step is to pass the request to enqueue() of WorkManager, and the task will run when the time is right:

```
WorkManager.getInstance(context).enqueue(request)
```

To test this, let us first add “Do Work” button in activity\_main.xml as shown in code below (Fig. 13.14):

This is the basic used case of WorkManager, and next, let us take a look at some more advanced tasks.

### 13.6.2 Handling Complex Task with WorkManager

In the previous example, we successfully ran a background task, but did not control when can it run. WorkManager provides the interface to set the time and many other configurations, and now let us take a look at some of them.

To delay the execution by a specified time, we just need to use setInitialDelay() as shown in code below:

```
val request = OneTimeWorkRequest.Builder(SimpleWorker::class.java)
    .setInitialDelay(5, TimeUnit.MINUTES)
    .build()
```

The above code will run the task in 5 min, and you can choose time units from millisecond, second, minute, hour, and day.

We can also add a tag to the background task:

```
val request = OneTimeWorkRequest.Builder(SimpleWorker::class.java)
    ...
    .addTag("simple")
    .build()
```

And we can cancel the background task request with tag:

```
WorkManager.getInstance(this).cancelAllWorkByTag("simple")
```

If there is no tag, we can use id to cancel the background task request:

```
WorkManager.getInstance(this).cancelWorkById(request.id)
```

However, only one request can be cancelled with id, while all tasks with the same tag can be cancelled together which is useful in certain situations.

We can also use the following code to cancel all the background task requests:

```
WorkManager.getInstance(this).cancelAllWork()
```

As aforementioned, if doWork() returns Result.retry(), then the task can run again with setBackoffCriteria() as shown below:

```
val request = OneTimeWorkRequest.Builder(SimpleWorker::class.java)
    ...
    .setBackoffCriteria(BackoffPolicy.LINEAR, 10, TimeUnit.SECONDS)
    .build()
```

The setBackoffCriteria() method takes three params: the second and third params are used to set the time for after what time the task will run again, and the minimum value cannot be less than 10 s; the first param is used to set the criteria of retry if the task fails again. This is because if task keeps failing, then it is not meaningful to run again and again which will only drain the battery. As the failure times increase, we should increase the time for the next retry. There are two values for the first param we can choose from: LINEAR and EXPONENTIAL and are self-explanatory.

The other two results Result.success() and Result.failure() can be used as shown below:

```
WorkManager.getInstance(this)
    .getWorkInfoByIdLiveData(request.id)
    .observe(this) { workInfo ->
        if (workInfo.state == WorkInfo.State.SUCCEEDED) {
            Log.d("MainActivity", "do work succeeded")
```

```
    } else if (workInfo.state == WorkInfo.State.FAILED) {
        Log.d("MainActivity", "do work failed")
    }
}
```

The `getWorkInfoByIdLiveData()` method takes the background task request id and returns an object of `LiveData`, and we can call its `observer()` to observe the data change which is the result of the background task.

The `getWorkInfosByTagLiveData()` can observe the result of all the task requests with the same tag name which can be used above.

Next let us take a look at chaining work.

Assume we have three independent background tasks: sync data, compress data, and upload data. Now if we want to run them sequentially, we can use chaining work to implement as shown in the code below:

```
val sync = ...
val compress = ...
val upload = ...
WorkManager.getInstance(this)
    .beginWith(sync)
    .then(compress)
    .then(upload)
    .enqueue()
```

The above code is self-explanatory. The `beginWith()` method starts a chaining work, and the `then()` method can append the works. `WorkManager` only runs the task after the previous task successfully finishes running which means that if a certain task fails or is cancelled, then all the following tasks will not get executed.

That is all you need to know about `WorkManager` and also everything I'd like to discuss for Jetpack in this chapter. You should be able to build a high-quality app now, and we will practice the knowledge learnt in Chap. 15. Now it is Kotlin Class time again.

## 13.7 Kotlin Class: Use DSL to Construct Specific Syntax

DSL is short for domain-specific language which can be used to create syntax that is different from the host language to meet specific needs.

Kotlin supports DSL, and there is no fixed pattern to implement DSL in Kotlin. For instance, the syntax we created by using infix function in Chap. 9's Kotlin Class section belongs to DSL. In this section, we will take a look at how to use higher-order functions to implement DSL which is the most frequent approach to implement DSL in Kotlin.

We've been using DSL for sometimes now, for instance, to add dependencies in the project, we may add the following code in `build.gradle`:

```
dependencies {
    implementation 'com.squareup.retrofit2:retrofit:2.6.1'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.1'
}
```

Gradle is the build tool based on Groovy, and the above syntax is DSL provided by Groovy. With Kotlin, we can create the similar syntax too. Let us see how to do it.

First, create DSL.kt and define Dependency class as shown below:

```
class Dependency {

    val libraries = ArrayList<String>()

    fun implementation(lib: String) {
        libraries.add(lib)
    }

}
```

The List object contains all the dependencies, and the implementation() adds dependency to List.

Next, define dependencies higher-order function as shown in code below:

```
fun dependencies(block: Dependency.() -> Unit): List<String> {
    val dependency = Dependency()
    dependency.block()
    return dependency.libraries
}
```

The dependencies function takes a param of function type, and since this param is defined inside Dependency class, thus an instance of Dependency needs to be created to call this function. This instance will call the function type param, and thus the Lambda expression will be executed. In the end, the dependency collection is returned.

Now with this DSL design, we can use the following syntax in our project:

```
dependencies {
    implementation("com.squareup.retrofit2:retrofit:2.6.1")
    implementation("com.squareup.retrofit2:converter-gson:2.6.1")
}
```

Since dependencies function takes a function type param, thus we can pass in a Lambda expression, and since this Lambda expression has the context of Dependency class, implementation() of Dependency can be called directly to add the dependency.

Of course this is still not exactly the same as the syntax used in build.gradle because of the syntax difference between Kotlin and Groovy.



**Fig. 13.15** Print all the added dependencies

The added dependencies can also be acquired by the return value of dependencies function as shown in code below:

```
fun main() {
    val libraries = dependencies {
        implementation("com.squareup.retrofit2:retrofit:2.6.1")
        implementation("com.squareup.retrofit2:converter-gson:2.6.1")
    }
    for (lib in libraries) {
        println(lib)
    }
}
```

The return value of dependencies function is assigned to libraries variable, and we use the for-in loop to print all the dependencies. Run main(), and the result should be as shown in Fig. 13.15.

The result shows that the dependencies added with DSL syntax are all acquired.

The above syntax is more straightforward than calling implementation() method of Dependency object, and you shall find that the more dependencies to add, the more advantageous DSL syntax will be.

Let us try to implement some more complicated DSL.

You probably know that website rendering is done by parsing HTML code. HTML defines numerous tags, <table> tag is used to create table, <tr> is used to create row in table, and <td> tag defines a standard cell. The combination of these three tags can create table of any sizes.

Let us see an example. First, create test.txt with the following HTML code:

```
<table>
<tr>
    <td>Apple</td>
    <td>Grape</td>
    <td>Orange</td>
</tr>
<tr>
    <td>Pear</td>
    <td>Banana</td>
    <td>Watermelon</td>
</tr>
```



**Fig. 13.16** Table created with HTML

This defines a table with two rows and three columns. How to verify it? We can simply change the file format extension and rename it to test.html and double click this file to open it with browser; the result should be as shown in Fig. 13.16.

The border is not visible because by default the width of the border is 0.

What if we need to dynamically generate the HTML code using Kotlin? The most naïve way is to concatenate the strings which is redundant and not readable at all.

With DSL we can generate HTML with more succinct and readable code.

Define Td class in DSL.kt as shown in code below:

```
class Td {
    var content = ""

    fun html() = "\n\t\t<td>$content</td>"
}
```

The content variable is used to fill the cell, and the html() returns HTML code with <td>. Tag and set content with the content variable. Notice that in order to make the result more straightforward, I used escape here to add line and indentation. Of course this is not needed as browser will ignore new line and indentation when parsing HTML code.

Now create Tr class as shown in code below:

```
class Tr {
    private val children = ArrayList<Td>()

    fun td(block: Td.() -> String) {
        val td = Td()
        td.content = td.block()
        children.add(td)
    }

    fun html(): String {
        val builder = StringBuilder()
        builder.append("\n\t<tr>")
        for (childTag in children) {
            builder.append(childTag.html())
        }
    }
}
```

```

        builder.append("\n\t</tr>")
        return builder.toString()
    }
}

```

Tr is more complicated than Td as it can contain any number of <td>. The children variable contains all the Td object of the current Tr. The td() function takes a param of function type that is defined on Td and returns value of String type. Inside td(), an object of Td type is created, and its block() method is called, and the return value is assigned to content field of Td which means that the return value of the Lambda expression is returned to content field. Of course, this Td object needs to be added to the children collection.

The html() method inside Tr is similar to the html() in Td except that since Tr may contain multiple Td, we need to iterate the children collection to add all the Tds within <tr> and return nested HTML code.

Now we can use the following code to construct one row of data:

```

val tr = Tr()
tr.td { "Apple" }
tr.td { "Grape" }
tr.td { "Orange" }

```

But this is finished yet. Define Table class, as shown in code below:

```

class Table {
    private val children = ArrayList<Tr>()

    fun tr(block: Tr.() -> Unit) {
        val tr = Tr()
        tr.block()
        children.add(tr)
    }

    fun html(): String {
        val builder = StringBuilder()
        builder.append("<table>")
        for (childTag in children) {
            builder.append(childTag.html())
        }
        builder.append("\n</table>")
        return builder.toString()
    }
}

```

It's similar to the Tr class. The children collection in Table contains the Tr objects. The tr() method takes a function type param, and when this method is called, an object Tr will be created, and the Lambda expression will be called. The created Tr object is added to the children collection.

The `html()` method is also similar and iterate the `children` collection then add all the `Tr` object within `<table>` tag.

Now we can use the following code to construct a table:

```
val table = Table()
table.tr {
    td { "Apple" }
    td { "Grape" }
    td { "Orange" }
}
table.tr {
    td { "Pear" }
    td { "Banana" }
    td { "Watermelon" }
}
```

There is room to optimize for the above code. Define `table()` as shown below:

```
fun table(block: Table.() -> Unit): String {
    val table = Table()
    table.block()
    return table.html()
}
```

The `table()` function takes a param of function type that is defined on `Table`. When `table()` is called, an object of `Table` will be created, and the Lambda expression will be called . Finally return the HTML code by `html()` of `Table`.

Now we can use the following code to dynamically generate a table HTML code:

```
fun main() {
    val html = table {
        tr {
            td { "Apple" }
            td { "Grape" }
            td { "Orange" }
        }
        tr {
            td { "Pear" }
            td { "Banana" }
            td { "Watermelon" }
        }
    }
    println(html)
}
```

This DSL syntax is much more readable. Run `main()`, and the result should be as shown in Fig. 13.17.

Inside DSL, other Kotlin syntax can also be used. For instance, we can use loop to generate `<tr>` and `<td>` tags:



```

Run: app com.example.jetpacktest.DSLKt ...
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
<table>
  <tr>
    <td>Apple</td>
    <td>Grape</td>
    <td>Orange</td>
  </tr>
  <tr>
    <td>Pear</td>
    <td>Banana</td>
    <td>Watermelon</td>
  </tr>
</table>

Process finished with exit code 0

```

**Fig. 13.17** DSL generated HTML code



```

Run: app com.example.jetpacktest.DSLKt ...
"/Applications/Android Studio.app/Contents/jre/jdk/Contents/Home/bin/java" ...
<table>
  <tr>
    <td>Apple</td>
    <td>Grape</td>
    <td>Orange</td>
  </tr>
  <tr>
    <td>Apple</td>
    <td>Grape</td>
    <td>Orange</td>
  </tr>
  <tr>
    <td>Apple</td>
    <td>Grape</td>
    <td>Orange</td>
  </tr>
</table>

Process finished with exit code 0

```

**Fig. 13.18** HTML table code generated with loop

```

fun main() {
    val html = table {
        repeat(2) {
            tr {
                val fruits = listOf("Apple", "Grape", "Orange")
                for (fruit in fruits) {
                    td { fruit }
                }
            }
        }
        println(html)
    }
}

```

The `repeat()` function helps us to generate two rows of data, and for each row, the `for-in` loop iterate the `List` collection to fill the data for the cells. The result should be as shown in Fig. 13.18.

In this example, we used higher-order function to implement an advanced DSL, and hopefully it can help you in your professional career when you need to create DSL.

That is the end of Kotlin DSL, and let us summarize what we have learned in this chapter.

## 13.8 Summary and Comment

In this chapter, we discussed some topics in Jetpack which is a suite of libraries and guidelines. Due to its complexity, we mainly discussed the architecture component. This component can help developers to build apps with quality and well-structured code. The ViewModel, Lifecycles, and LiveData are born for MVVM, and we will use MVVM to finish a real project in Chap. 15.

Room can help us improve the architecture of database design. WorkManager provides another option for us to run background task, and keep it in mind that WorkManager is totally different from Service.

In the Kotlin Class section, we discussed a more advanced topic which is DSL without introducing any new Kotlin features but instead through the flexible use of higher-order function and created syntax that looks totally different from Kotlin. Constructing DSL is not sued frequently, but when you need it, this chapter should help you find solution.

After 13 chapters, we systematically discussed Android development knowledge. But still there are some advanced topics I want to cover in this book which will be covered in the next chapter.

# Chapter 14

## Keep Stepping Up: More Skills You Need to Know



Toward the end of the book, we've covered enough fundamental topics, and it is time to discuss some more advanced techniques.

### 14.1 Obtaining Context Globally

You have probably noticed that there are many places we need to use Context, such as displaying Toast, starting Activity, sending Broadcast, operating the database, sending notification, and so on.

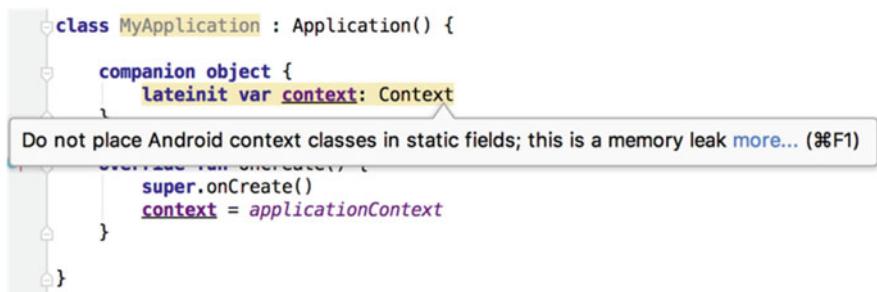
We did have any problem to get Context in our previous examples because all the actions were taken inside Activity and Activity itself is Context. However, as the architecture of the app is getting more complicated, it is likely that we need to get Context outside of Activity.

For instance, in Chap. 12's Kotlin Class section, we created `Toast.kt` and encapsulates the code to use `Toast` as shown in code below:

```
fun String.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}

fun Int.showToast(context: Context, duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(context, this, duration).show()
}
```

The `makeText()` method requires a param of `Context` type, while this code is not inside `Activity` nor `Service`; thus there is no way to get `Context` object directly. That is the reason we add `Context` param to `showToast()` and hoping that the callsite will pass in the `Context` object.



**Fig. 14.1** Memory leak warning

This solution passes the problem to the code that is calling it without actually solving the problem. Next let us take a look at how to get Context anywhere you need.

The Application class in Android will be initialized when the app is started. We can create a subclass of Application to manage some global states of the app, and global Context is one of them.

First we need to create `MyApplication` class that inherits Application as shown in code below:

```

class MyApplication : Application() {

    companion object {
        lateinit var context: Context
    }

    override fun onCreate() {
        super.onCreate()
        context = applicationContext
    }
}

```

The `context` variable is defined inside `companion object`, and `onCreate()` is overridden. Inside `onCreate()`, the context acquired by `getApplicationContext()` is assigned to `context` variable which makes it possible to get the instance of Context in a static way.

Notice that static Context can easily lead to memory leak and thus is risky to do so, and Android Studio will display warning as shown in Fig. 14.1.

However, we are not using Activity context nor Service context but Application context which only one instance exists throughout the life of app; thus there is risk of memory leak in the above code. We can use the following annotation to mute the warning:

```
class MyApplication : Application() {  
  
    companion object {  
        @SuppressLint("StaticFieldLeak")  
        lateinit var context: Context  
    }  
    ...  
}
```

Next we need to make sure that when the app starts, `MyApplication` class instead of the default `Application` class should be initialized. It can be easily done by setting it in `<application>` tag in `AndroidManifest.xml` as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/  
    android"  
        package="com.example.materialtest">  
    <application  
        android:name=".MyApplication"  
        android:allowBackup="true"  
        android:icon="@mipmap/ic_launcher"  
        android:label="@string/app_name"  
        android:roundIcon="@mipmap/ic_launcher_round"  
        android:supportsRtl="true"  
        android:theme="@style/AppTheme">  
        ...  
    </application>  
</manifest>
```

Now, wherever we need to use `Context` in the codebase, we just need to call `MyApplication.context`.

With this change, we can optimize `showToast()` as shown in code below:

```
fun String.showToast(duration: Int = Toast.LENGTH_SHORT) {  
    Toast.makeText(MyApplication.context, this, duration).show()  
}  
  
fun Int.showToast(duration: Int = Toast.LENGTH_SHORT) {  
    Toast.makeText(MyApplication.context, this, duration).show()  
}
```

Now, `showToast()` does not need to get the context through param but instead simply calls `MyApplication.context`. And we can simply call this method as shown below:

```
"This is Toast".showToast()
```

You never need to worry about getting `Context` instance any more.

## 14.2 Passing Objects with Intent

We can use Intent to start Activity and Service and send Broadcast and so on. We can add some extra data in the Intent to share value. For instance, we added the following code in FirstActivity:

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("string_data", "hello")
intent.putExtra("int_data", 100)
startActivity(intent)
```

The putExtra() method adds shared data which SecondActivity can get with code below:

```
intent.getStringExtra("string_data")
intent.getIntExtra("int_data", 0)
```

You probably have noticed that the data types that putExtra() supports are limited and customized types are not supported. Now let's see how to pass object with Intent.

### 14.2.1 *Serializable*

Using Intent to share object is usually done through Serializable and Parcelable. We will cover Serializable first.

The serialized object can be saved to local device and transferred in the network. To serialize an object, we just need the class implement the Serializable interface.

Assume Person class has name and age fields, and to serialize this class, we can have code below:

```
class Person : Serializable {
    var name = ""
    var age = 0
}
```

Person class implements Serializable interface, and then all the Person objects are serializable.

Then in FirstActivity we can pass Person object using Intent as shown below:

```
val person = Person()
person.name = "Tom"
person.age = 20
val intent = Intent(this, SecondActivity::class.java)
```

```
intent.putExtra("person_data", person)
startActivity(intent)
```

As Person implements Serializable interface, we can pass the Person instance directly to putExtra().

To get this object in SecondActivity is easy too, as shown below:

```
val person = intent.getSerializableExtra("person_data") as Person
```

The getSerializableExtra() method returns the serialized object, and by downcasting it to Person object, we successfully share object with Intent.

Notice that, after serialization and deserialization, we have two instances that have exactly the same value which means that this process does not pass the reference of the object but pass the value of the object.

### 14.2.2 *Parcelable*

Unlike Serializable, Parcelable decomposes the object and makes sure that every part is supported by Intent which makes it possible to pass the object.

To use Parcelable to pass Person, edit Person class as code below:

```
class Person : Parcelable {
    var name = ""
    var age = 0

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(name) // write name
        parcel.writeInt(age) // write age
    }

    override fun describeContents(): Int {
        return 0
    }

    companion object CREATOR : Parcelable.Creator<Person> {
        override fun createFromParcel(parcel: Parcel): Person {
            val person = Person()
            person.name = parcel.readString() ?: "" // read name
            person.age = parcel.readInt() // read age
            return person
        }

        override fun newArray(size: Int): Array<Person?> {
            return arrayOfNulls(size)
        }
    }
}
```

```

    }
}
```

Parcelable implementation is more complicated. By implementing Parcelable, we override describeContents() and writeToParcel(). We can simply return 0 for describeContents(), but in writeToParcel(), we need to call the writeXxx() of Parcel to write all the fields in Person. For instance, to write string, call writeString(), and to write integer, call writeInt(), so on so forth.

We also need to provide a companion object CREATOR which is an anonymous class implementation in Person.

Here we create an implementation of Parcelable.Creator and set its generic type to Person. Then we need to override createFromParcel() and newArray(). In createFromParcel(), we need to create an instance of Person, read its name and age field through readXxx() functions, and return it. Notice that the read order has to match the write order. The newArray() is simply called arrayOfNulls() and uses the size argument to create an empty array of Persons.

In FirstActivity we can still use the same code which is the pass Person object, but we need to make some changes when reading object in SecondActivity as shown below:

```
val person = intent.getParcelableExtra("person_data") as Person
```

Essentially we replaced getSerializableExtra() with getParcelableExtra().

Using Parcelable in this way has lots of boilerplate code, and Kotlin provides another simpler way to use Parcelable, and the prerequisite is that all the data have to be in the main constructor.

Edit Person class as code below:

```
@Parcelize
class Person(var name: String, var age: Int) : Parcelable
```

We simply move the name and age fields in the main constructor and add @Parcelize annotation to Person which greatly reduced the boilerplate code.

The Serializable is simpler than Parcelable; however, since it needs serialization and deserialization of the whole object, it is less efficient compared with Parcelable. Thus in most of the cases, Parcelable is the recommended solution to pass object with Intent.

### 14.3 Creating Your Own Logging Tool

We discussed how to use Android log in 1.4 and applied log in almost every example in this book. Though the native log is already powerful, there is still space to improve, for instance, controlling the log can be optimized.

Assume you are working on a big project, and to help debug, you added lots of logs. Now as the project is coming to the launch state, there is a problem which is the logs for debugging purpose will keep logging after release which will drag the performance of the app and potentially leak important information.

We cannot simply remove these logs as it takes time to do so, and we still need the log for maintenance purpose. Thus, we need a way to control logging such that logs will only be recorded during dev environment and turned off for prod environment.

This may seem complicated at first glance, but we can actually easily achieve this by customizing the log tool. Create LogUtil singleton as shown in code below:

```
object LogUtil {  
  
    private const val VERBOSE = 1  
  
    private const val DEBUG = 2  
  
    private const val INFO = 3  
  
    private const val WARN = 4  
  
    private const val ERROR = 5  
  
    private var level = VERBOSE  
  
    fun v(tag: String, msg: String) {  
        if (level <= VERBOSE) {  
            Log.v(tag, msg)  
        }  
    }  
  
    fun d(tag: String, msg: String) {  
        if (level <= DEBUG) {  
            Log.d(tag, msg)  
        }  
    }  
  
    fun i(tag: String, msg: String) {  
        if (level <= INFO) {  
            Log.i(tag, msg)  
        }  
    }  
  
    fun w(tag: String, msg: String) {  
        if (level <= WARN) {  
            Log.w(tag, msg)  
        }  
    }  
}
```

```

fun e(tag: String, msg: String) {
    if (level <= ERROR) {
        Log.e(tag, msg)
    }
}

```

We first define VERBOSE, DEBUG, INFO, WARN, and ERROR integers with ascending value and then define a static variable level that will have value of these five integers.

Next we define v(), d(), i(), w(), and e() log methods which call Log.v(), Log.d(), Log.i(), Log.w(), and Log.e() correspondingly, and for each of these methods, we add condition check to make sure that only log when the level is smaller than or equal to the current log level.

That is what we need to do for customizing the log tool. In the project we can use LogUtil as the native log. For instance, to log DEBUG level log, we can have the code as below:

```
LogUtil.d("TAG", "debug log")
```

To log WARN level log, we can have the following code:

```
LogUtil.w("TAG", "warn log")
```

This means that we can control the log by setting the value of level. For instance, setting level to VERBOSE will log all, and setting level to ERROR will only log the errors.

With this we can simply set level to VERBOSE during development and set level to ERROR for release version.

## 14.4 Debug Android Apps

Sometimes, we cannot find out what goes wrong by just looking at logs, and we need to debug line by line, observe the data in the memory, and so on to debug. In this section, let us take a look at how to debug using Android Studio.

Remember the app we built in Chap. 6 that can force log off? Let us use it as an example to debug Android app. The app has log in functionality and assume log in is not working, and we need to find out why this is the case.

The first step is to add breakpoint. Since we need to debug the log in code, so we should add the breakpoint inside the click event of login button. To add breakpoint, click the left side of the code as shown in Fig. 14.2.

To remove the breakpoint, click it again.



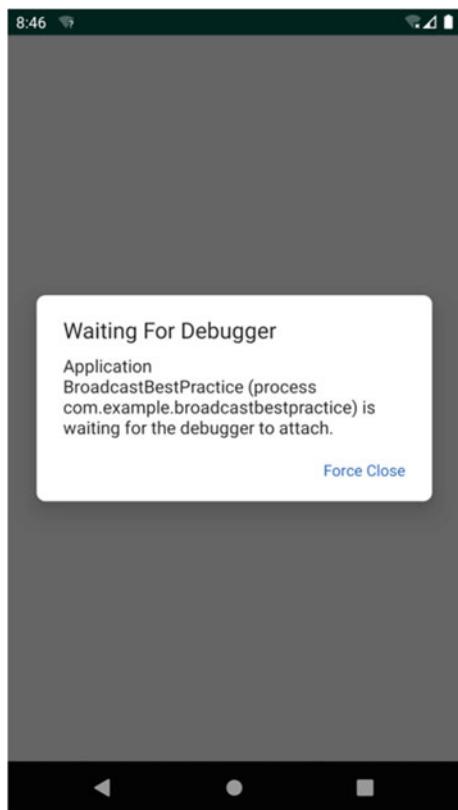
```
25
26 ● login.setOnClickListener { it: View!
27     val account = accountEdit.text.toString()
28     val password = passwordEdit.text.toString()
29     // 如果账号是admin且密码是123456, 就认为登录成功
30     if (account == "admin" && password == "123456") {
```

Fig. 14.2 Add breakpoint

Fig. 14.3 Buttons in Toolbar



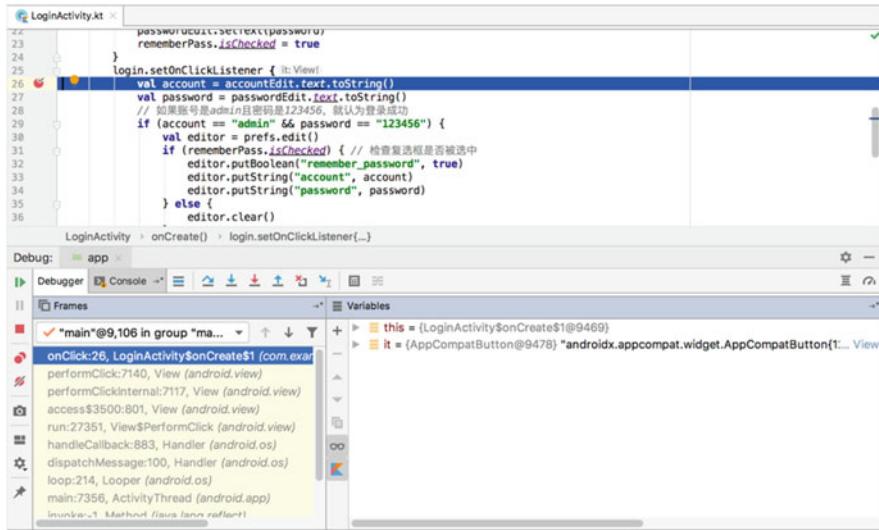
Fig. 14.4 Dialog to wait for debugger



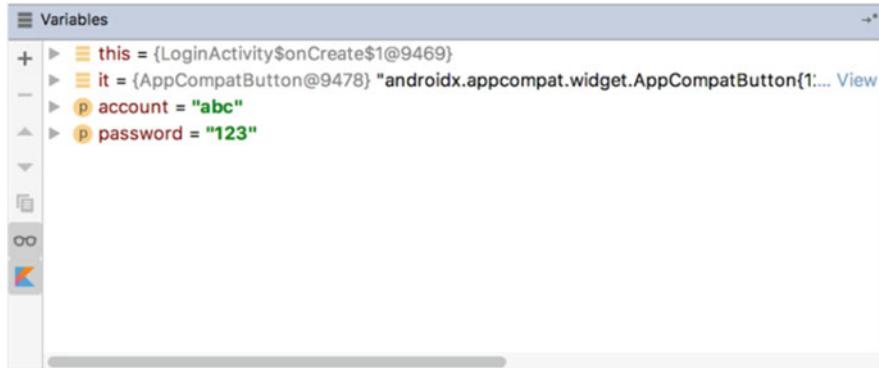
With breakpoint, we can start debugging. Clicking “Debug” button in the toolbar on the top (the rightmost button in Fig. 14.3) will run the app in debug mode.

When the app starts to run, you will see a dialog as shown in Fig. 14.4.

This dialog will disappear automatically in a short time, and then type in account and password and click “Login” button in the login screen, and then Android Studio will open the Debug window as shown in Fig. 14.5.



**Fig. 14.5** Debug window



**Fig. 14.6** Variables view

Press F8 to execute one line of code, and you shall see the data in the memory from the Variables view as shown in Fig. 14.6.

As you can see, the account and password app got from the input fields are abc and 123, respectively, while the correct account and password should be admin and 123456, and that is the reason why login failed. After identifying the problem and to stop debugging, click “Stop” button (the bottom button in Fig. 14.7) in the Debug window.

The steps above can help bugging, but app runs slowly in debug mode, and if the breakpoint is deep, then it will take some time to trigger the breakpoint. Android also

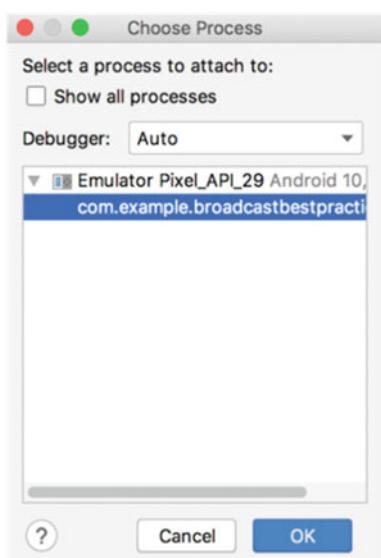
**Fig. 14.7** Stop debugging button



**Fig. 14.8** Buttons in Toolbar



**Fig. 14.9** Process selection dialog



provides another way to debug which can get the app in debug mode at any time, and let us take a look now.

Now, you can just start your app and then fill in account and password and then click “Attach Debugger to Android Process” button in the top toolbar (the rightmost button in Fig. 14.8). A process selection dialog will show up as shown in Fig. 14.9.

There is only one process to choose from which is the current app’s process. Click this process and click “OK” to get into debug mode for this process.

Next, click “Login” button in the app, and Android Studio will open the debug window, and after that it is the same process to debug. Compared with the previous approach, this is more flexible and widely used.

## 14.5 Dark Mode

The operating systems are mostly using light mode which has no problem during day time or when there is enough light. However, when there is not enough light, it may glare.

In order to make apps comfortable in dark environment, a lot of apps provide a button to switch to dark mode to change the color and brightness that is more fit in dark environment.

However, some apps support this, while some do not, and user may need to turn on dark mode in each app. To solve this problem Google released system level dark mode support in Android 10.0.

You may think that this looks not like rocket science, but why it takes so long for Android to support it? This is because only OS support is enough, and it will be useful only when majority of the apps can support it which is not easy at all.

Thus, I hope that when you build apps, you can build support for dark theme based on the requirement of Android OS. Otherwise, when user turns on dark mode and finds your app still in light mode, it will be bad user experience.

Dark mode is good for the eyes and also for the battery as it consumes less energy. Now let us take a look at how to support dark mode.

In Android 10.0 and above, user can turn on and off dark mode in Settings→Display→Dark theme. Once dark mode is turned on, the OS and internal apps turn to dark theme as shown in Fig. 14.10.

If you open the apps we created, you will find that they are still in light mode which contradicts the OS theme. Let us use the MaterialTest project we created in Chap. 12 as an example to see how to support dark theme.

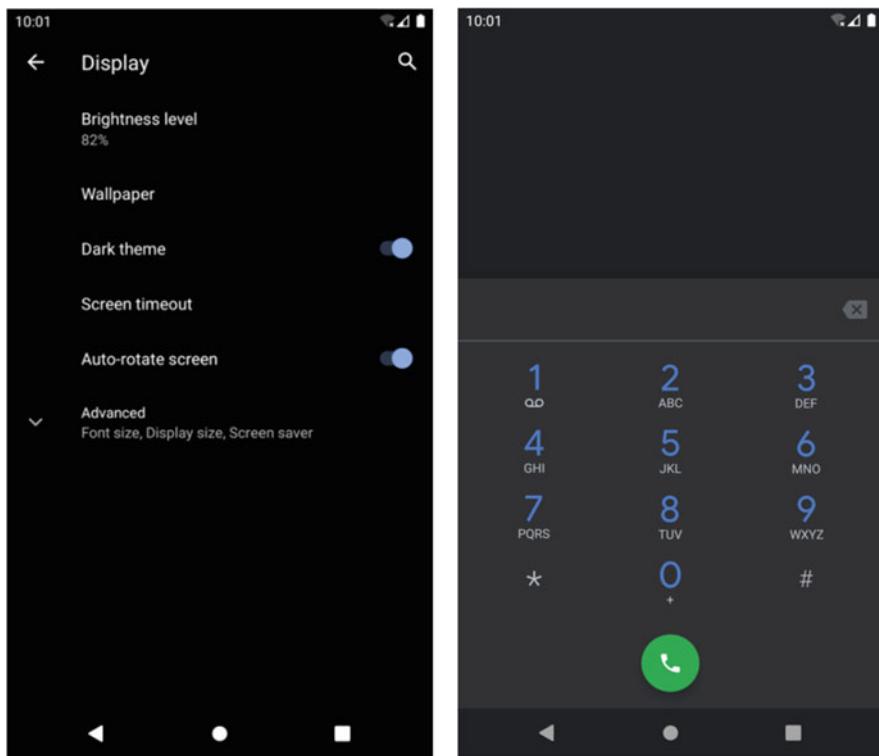
The easiest way is to use Force Dark which can make app quickly support dark theme with minimum code change. The mechanism of Force Dark is that OS will analyze every layer of View in light mode, and before the View is rendered, OS automatically converts the color to some color that fit for dark mode. Notice that only light theme apps can use this, if the current theme is dark theme, Force Dark does not work.

To use Force Dark in the project, we need to set android:forceDarkAllowed attribute which was introduced in API 29 or Android 10.0. Thus we need to make some changes for system compatibility.

Right click res→New→Directory, create values-v29 folder and right click values-v29→New→Values resource file and create styles.xml. Edit this file as code below:

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <item name="android:forceDarkAllowed">true</item>
    </style>

```



**Fig. 14.10** Settings and call app in dark mode

```
</style>  
</resources>
```

This is a copy of the previous styles.xml except the android:forceDarkAllowed attribute. Setting this attribute to true allows OS to use Force Dark, and only OS version no earlier than Android 10.0 will read values-v29.

Run MaterialTest, and the result should be as shown in Fig. 14.11.

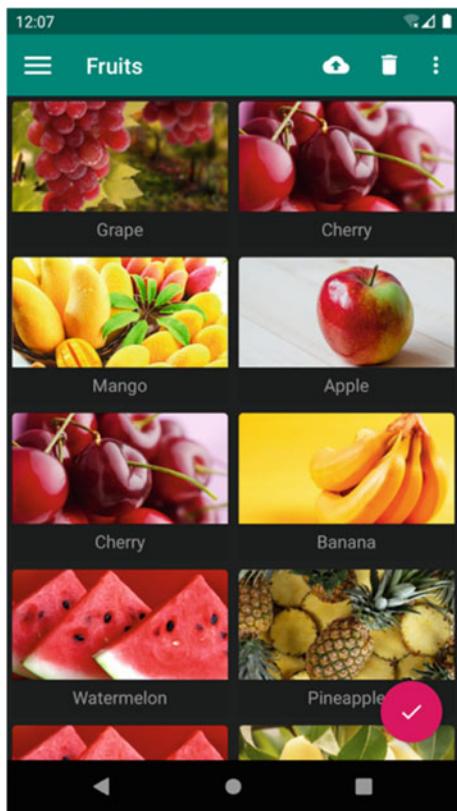
As you can tell, the dark theme here does not look very well, and the card layout effect is lost.

Force Dark is simple and brutal, and usually the result is not very impressive. Thus I do not recommend to use it but to manually implement.

There is no silver bullet to implement this, and we have to design the UI with theme change in mind which may sound complicated, but we can simplify it with some techniques.

In Chap. 12, we mentioned that AppCompat lib themes have light theme and dark theme. For instance, The Theme.AppCompat.Light.NoActionBar used in MaterialTest is light theme, and Theme.AppCompat.NoActionBar is dark theme. The default color for the widgets varies a lot under different themes.

**Fig. 14.11** Force Dark effect



If we choose DayNight theme, app will respect the theme selection in the settings. To test it, delete values-v29 folder and its contents and edit values/styles.xml as shown below:

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.DayNight.NoActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
    ...
</resources>
```

**Fig. 14.12** DayNight theme effect



The only change is that the parent theme of the AppTheme is set to Theme.AppCompat.DayNight.NoActionBar. Thus when user turns on dark mode, MaterialTest should use dark theme colors.

Run the app, and the result should be as shown in Fig. 14.12.

This looks better compared with the previous, and the card layout effect is reserved.

Notice that though most parts of the screen now are using dark theme colors, the title bar and the floating action button still use light theme. This is because they use the color defined in colors.xml as shown in code below:

```
<resources>
    <color name="colorPrimary">#008577</color>
    <color name="colorPrimaryDark">#00574B</color>
    <color name="colorAccent">#D81B60</color>
</resources>
```

This hardcoded the widget color, and DayNight cannot change the color.

**Fig. 14.13** Dark theme colors



The solution is that we can set different colors for dark theme. Right click res→New→Directory, and create values-night folder, and then right click values-night→New→values resource file and create colors.xml. Then set the colors for dark theme, as shown in code below:

```
<resources>
    <color name="colorPrimary">#303030</color>
    <color name="colorPrimaryDark">#232323</color>
    <color name="colorAccent">#008577</color>
</resources>
```

Now, in light theme, values/colors.xml will be used, and when dark theme is used, values-night/colors.xml will be used.

Run the app again, and the result should be as shown in Fig. 14.13.

If printing is color printing, the difference may be small, but in the real app, Figs. 14.12 and 14.13 look quite different.

Though we can change colors based on themes, it is not recommended to not use hardcoded colors to set widget color when DayNight theme is used. We should use

themes that can change color based on the current theme. For instance, black font color should have white background, while white font color should have black background, and we can use theme to set the background and text color; here is an example:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="?android:attr/colorBackground">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Hello world"
        android:textSize="40sp"
        android:textColor="?android:attr/textColorPrimary" />

</FrameLayout>
```

This will use the current theme to assign proper colors to the control, as shown in Fig. 14.14.

You can also execute code based on the current theme, and the following code is an example:

```
fun isDarkTheme(context: Context): Boolean {
    val flag = context.resources.configuration.uiMode and
        Configuration.UI_MODE_NIGHT_MASK
    return flag == Configuration.UI_MODE_NIGHT_YES
}
```

The isDarkTheme() method will return if the current system theme is dark theme or not.

Notice that Kotlin replaces bit operator with keywords; thus the and keyword is equal to & operator in Java, and the or keyword equals to | operator, xor equals to ^ operator.

That is it for using themes. The fundamental idea is that we should put dark theme into consideration for UI design, implementation, and testing. You can use the techniques mentioned in this section to make the process easier.

Next is the last section of Kotlin Class in this book.

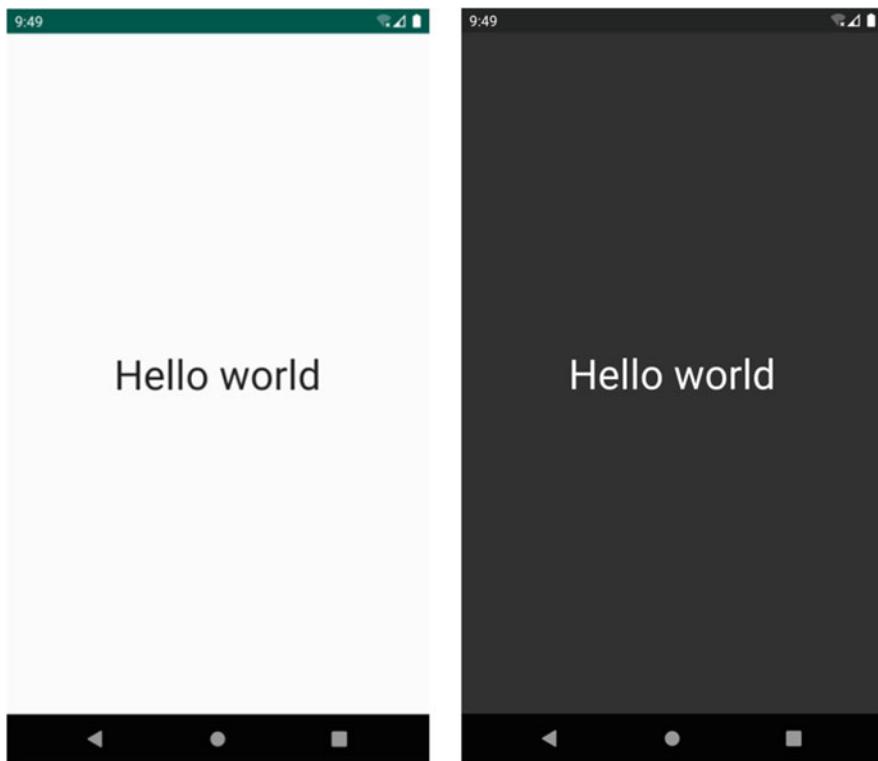


Fig. 14.14 Light theme and dark theme difference

## 14.6 Kotlin Class: Conversion Between Java and Kotlin Code

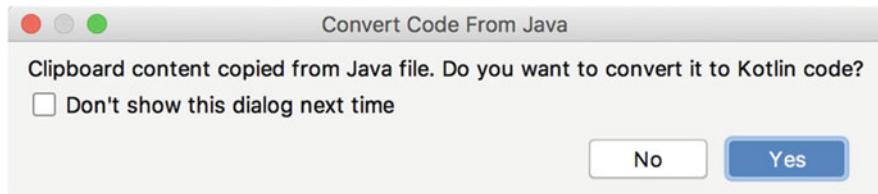
Now you should be familiar with Kotlin, and in this last section of Kotlin Class, I will not discuss any new Kotlin features but something that everyone cares how to convert between Java code and Kotlin code.

All of the example projects in this Book are written in Kotlin and do not have the problem of legacy Java code. However, there are plenty of projects that are written in Java, and what can we do to migrate the legacy codebase to Kotlin?

A naïve approach is to manually rewrite the code which is not a good idea. Android Studio can help us migrate the codebase with a click.

For instance, here is some Java code:

```
public void printFruits() {  
    List<String> fruitList = new ArrayList<>();  
    fruitList.add("Apple");  
    fruitList.add("Banana");  
    fruitList.add("Orange");
```



**Fig. 14.15** Dialog to convert Java to Kotlin

**Fig. 14.16** Auto converted code

```

fun printFruits() {
    val fruitList = ArrayList<String>()
    fruitList.add("Apple")
    fruitList.add("Banana")
    fruitList.add("Orange")
    fruitList.add("Pear")
    fruitList.add("Grape")
    for (fruit in fruitList) {
        println(fruit)
    }
}

```

If we want to convert this to Kotlin, we can copy the code and open a Kotlin file in Android Studio and paste there. Then Android Studio will have a dialog as shown in Fig. 14.15.

Click Yes, and Android Studio will convert the code for us, and the result should be as shown in Fig. 14.16.

You probably can notice that the auto conversion only converts based on syntax and does not utilize Kotlin features well. This means that the converted code is naïve and basic, and we still need to manually optimize the code. For instance, the above code can be optimized as shown in code below:

```

fun printFruits() {
    val fruitList = mutableListOf("Apple", "Banana", "Orange", "Pear",
"Grape")
    for (fruit in fruitList) {
        println(fruit)
    }
}

```

To convert a Java file to Kotlin file, we can open this Java file, and from navigation bar, click Code→Convert Java File to Kotlin File as shown in Fig. 14.17.

These are the two ways we can convert Java to Kotlin with a click of button. But how to convert Kotlin code to Java? Unfortunately, there is no way to do it in

**Fig. 14.17** Convert Java file to Kotlin file



```
class FirstActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_first)
        button.setOnClickListener { it: View!
            Toast.makeText(context: this, text: "You clicked Button", Toast.LENGTH_SHORT).show()
        }
    }
}
```

**Fig. 14.18** Use Button without findViewById()

Android Studio because Kotlin has many features that do not exist in Java which makes it hard to do such conversion.

However, we can compile the Kotlin code to bytecode and dissemble the bytecode to Java code. The dissembled code may not be able to run as normal Java but can help us understand the mechanism under the hood for Kotlin features.

For instance, kotlin-android-extensions plugin can simplify the code inside Activity by eliminating findViewById(). But how is this implemented?

An example use of this is shown in Fig. 14.18.

**Fig. 14.19** Kotlin Bytecode Window

```

1 // ====== com/example/materialtest/FirstActivi
2 // class version 50.0 (50)
3 // access flags 0x31
4 public final class com/example/materialtest/FirstActivi
5
6 // access flags 0x4
7 protected onCreate(android.os.Bundle;)
8 // annotable parameter count: 1 (visible)
9 // annotable parameter count: 1 (invisible)
10 @org.jetbrains.annotations.Nullable() // invisible
11
12 L0
13 LINENUMBER 11 L0
14 ALOAD 0
15 ALOAD 1
16 INVOKESPECIAL androidx/appcompat/app/AppCompatActiv
17 L1
18 LINENUMBER 12 L1
19 ALOAD 0
20 LDC -1300024
21 INVOKEVIRTUAL com/example/materialtest/FirstActivit
22 L2
23 LINENUMBER 13 L2
24 ALOAD 0
25 GETSTATIC com/example/materialtest/R$id.button : I
26 INVOKEVIRTUAL com/example/materialtest/FirstActivit
27 CHECKCAST android/widget/Button
28 NEW com/example/materialtest/FirstActivity$onCreate
29 DUP
30 ALOAD 0
31 INVOKESPECIAL com/example/materialtest/FirstActivit
32 CHECKCAST android/view/View$OnClickListener
33 INVOKEVIRTUAL android/widget/Button.setOnClickListener
34 L3
35 LINENUMBER 16 L3
36 RETURN
37 L4
38 LOCALVARIABLE this Lcom/example/materialtest/FirstA
39 LOCALVARIABLE savedInstanceState Landroid/os/Bundle
40 MAXSTACK = 4
41 MAXLOCALS = 2
42

```

As long as activity\_first.xml defines a button with id button, we can use button variable directly in Activity without definition nor calling findViewById() to assign value to button variable.

To understand how this is done, let us compile this to bytecode and then dissemble the bytecode to Java code.

Click Android Studio navigation bar Tools→Kotlin→Show Kotlin Bytecode and window as shown in Fig. 14.19 should pop up.

It may look confusing, but it is OK. Click “Decompile” button at the top left corner of the window to decompile the bytecode to Java code, as shown in Fig. 14.20.

In the Java code we can see that the plugin actually generates `$_findCachedViewById()` method(to prevent duplication of naming). In this method, the instance of control will be acquired through calling `findViewById()` with the passed in id value, and then `HashMap` is used to cache this instance to save searching next time.

In `onCreate()`, `$_findCachedViewById()` is called to get the instance of button, and `setOnClickListener()` is called to register the click event for the button.

Isn’t exciting to know what happens under the hood for the plugin? This technique can help us get a deeper understanding of the implementation behind many Kotlin features.

That is everything for conversion between Java and Kotlin code and is the end for the last Kotlin Class. Now you should be well prepared for most of the challenges in

```

public final class FirstActivity extends AppCompatActivity {
    private HashMap _$_findViewCache;

    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(-1300024);
        ((Button)this._$_.findCachedViewById(id.button)).setOnClickListener((OnClickListener)(OnClickListener) (it) -> {
            Toast.makeText((Context)FirstActivity.this, (CharSequence)"You clicked Button", duration: 0).show();
        });
    }

    public View _$_.findCachedViewById(int var1) {
        if (this._$_.findViewCache == null) {
            this._$_.findViewCache = new HashMap();
        }

        View var2 = (View)this._$_.findViewCache.get(var1);
        if (var2 == null) {
            var2 = this.findViewById(var1);
            this._$_.findViewCache.put(var1, var2);
        }

        return var2;
    }

    public void _$_.clearFindViewByIdCache() {
        if (this._$_.findViewCache != null) {
            this._$_.findViewCache.clear();
        }
    }
}

```

**Fig. 14.20** Decompiled Java code

real project, and to get you even more well prepared, we will practice what we have learned in next chapter. Now let us first summarize what we have learnt in this chapter.

## 14.7 Summary

We have finished 14 chapters! This also marks the end of the discussion of Kotlin and Android knowledge.

We covered most of the important topics in Android in 14 chapters. We started with setting up the development environment and discussed the four main components, UI, Fragment, data persistence, multi-media, network, Material Design, Jetpack, and so on. In this chapter, we discussed advanced topics like how to get Context globally and customize log tool, debugging, dark theme, and so on.

We also went over important topics in Kotlin in our Kotlin Class sections and created all of the example apps with Kotlin. Hopefully you should be familiar with Kotlin now.

However, we have not build an app that can use everything we have discussed yet. In the next chapter, let us practice what we have learned so far to build a weather app.

# Chapter 15

## Real Project Practice: Creating a Weather App



In this chapter we will build a functional weather app to test what we have learned. Let us name it SunnyWeather and begin the work.

### 15.1 Analysis Before Start

Before we jump into coding, we need to analyze the requirements of the app and make a list for the functions based on which we will implement one by one. I think SunnyWeather should provide the following functions:

- Search data for cities in majority of the countries
- Search weather data of majority of the cities in the world
- Capability to switch cities
- Capability to refresh the weather

Although there are only four items in the list, to implement them, we need to use UI, network, data persistence, multithreading, and so on. Good thing is that we covered all of these in previous chapters, and it should not be hard.

After function requirements, we need to consider the technical feasibility. The most important question is how can we get the data of cities in majority of the countries and get the weather data in these cities. Unfortunately, there are less and less free weather information APIs, and many of them are closed service. In this book, we will use Caiyun weather.

Caiyun weather itself is an excellent app for weather report, and we will use this app as an example and try to implement app that is similar to it. Also Caiyu opens its API that can provide the city data and its weather data in more than 100 countries. These APIs are reliably available. However, you cannot use the API for free for unlimited times. You can send 100k free requests each day which is more than enough for testing purpose.

1. Account Information    2. Token Application    3. Completion

**Developer Type \***: Individual / NPO [Switch to Enterprise](#)

**Personal Name/ NPO Name:** \* Please fill in the full name

**Contact Name:** \*

**Phone Number:** \* Please fill in the correct name & Contact Information, otherwise the auditor will not be able to review your information.

**验证码:** \*  [发送验证码](#)

**Next**

Fig. 15.1 Fill in Account Information

1. Account Information    2. Token Application    3. Completion

**App Type:**  Weather API  Translation API

**App Name:** Sunny Weather

**App URL:**   
If you are developing an application that uses an open platform data interface and is already available in the iOS / android App market, please submit the address of the store where the application is located (for Web Applications, please provide a web address; for PC applications, please provide a download address)

**App Description:** test app for first line of code  
If the application is still under development, please indicate in this field and provide the URL after the application is online.

**Submit**

Fig. 15.2 Request token

To use the API, you need to register an account, and the address for it is <https://dashboard.caiyunapp.com/>.

Then login to the account and fill in the account information as shown in Fig. 15.1.

Click Next to get the token. You do not need to fill the app link, but instead, you can just fill in the app info as it is to be built as shown in Fig. 15.2.

| List of Tokens |             |                 |       |     |          |         |         |
|----------------|-------------|-----------------|-------|-----|----------|---------|---------|
| Name           | App Type    | Token           | Stats | API | Document | Package | purpose |
| SunnyWeather   | Weather     | xbYS77OCu61K... | check | try | check    | buy     | modify  |
|                | Translation | r5f2t5qtp285... | check | try | check    | buy     | modify  |

Fig. 15.3 Token page

Submit the request and wait for approval which usually takes one workday. After getting the approval, click “My Token” page to check the token you get which should show something as shown in Fig. 15.3.

Click the token link to see the token value and the available request number.

We can use this token to use the API provided by Caiyun weather, for instance, the following address will get the information of New York.

```
https://api.caiyunapp.com/v2/place?query=beijing&token={token}
&lang=en_US
```

The query param sets the search keywords, and token param should be the token value you just get. Caiyun server will return JSON response to the sender, and for the above request, it should be something as shown below:

```
{"status": "ok", "query": "beijing",
"places": [
 {"id": "place.11643701859936750", "location":
 {"lat": 39.905, "lng": 116.39139}, "place_id": "mb-
 place.11643701859936750", "name": "Beijing, People's Republic of
 China", "formatted_address": "Beijing"}, {
 "id": "poi.352187422529", "location":
 {"lat": 40.0743725, "lng": 116.5931035}, "place_id": "mb-
 poi.352187422529", "name": "Beijing Capital International Airport,
 Airport Expressway \u673a\u573a\u9ad8, Beijing, 100621, People's
 Republic of China", "formatted_address": "Beijing Capital
 International Airport"}, {
 "id": "poi.953482743839", "location":
 {"lat": 39.5113035, "lng": 116.410748}, "place_id": "mb-
 poi.953482743839", "name": "Beijing Daxing International Airport,
 Langfang, Hebei 065000, People's Republic of
 China", "formatted_address": "Beijing Daxing International Airport"}, {
 "id": "poi.790273992530", "location":
 {"lat": 39.9448205, "lng": 116.4083415}, "place_id": "mb-
 poi.790273992530", "name": "Confucius Temple, 13 Guozijian St, Beijing,
 100007, People's Republic of China", "formatted_address": "Confucius
 Temple"}, {
 "id": "poi.738734386221", "location":
 {"lat": 31.22803125, "lng": 121.475908375}, "place_id": "mb-
 poi.738734386221", "name": "Shanghai Concert Hall, 523 Yan'an E Rd
```

```
\u5ef6\u5b89\u4e1c\u8def523\u53f7, Shanghai, 200021, People's
Republic of China", "formatted_address": "Shanghai Concert Hall"
}]
```

Status means the state of request, and ok means that request is successful. Places is a JSON array which contains the location information that server thought most related to the search keywords. Notice that location has longitude and latitude information.

After getting the information of the city, we need to get the weather data of this city which has the following address:

```
https://api.caiyunapp.com/v2.5/{token}/116.4073963,39.9041999/
realtim.json
```

We still need to pass in the token value, and then we can pass the longitude and latitude information separated with comma. Then server will return the real-time weather data in JSON format. Below is a simplified version of the response:

```
{
  "status": "ok",
  "result": {
    "realtime": {
      "temperature": 23.16,
      "skycon": "WIND",
      "air_quality": {
        "aqi": { "chn": 17.0 }
      }
    }
  }
}
```

The real-time block contains the real-time weather of the selected location. The content inside the response block should be self-explanatory.

To get weather forecast, we need to use another API interface which can be used as shown below:

```
https://api.caiyunapp.com/v2.5/{token}/116.4073963,39.9041999/
daily.json
```

It simply changes the last part from realtime.json to daily.json. This API will also return a lot of data, and here is a simplified version:

```
{
  "status": "ok",
  "result": {
    "daily": [
      "temperature": [ { "max": 25.7, "min": 20.3 }, ... ],
      ...
    ]
  }
}
```

```
"skycon": [ {"value": "CLOUDY", "date": "2019-10-20T00:00+08:00"},  
... ],  
  "life_index": {  
    "coldRisk": [ {"desc": "\u06781\u06613\u053d1"}, ... ],  
    "carWashing": [ {"desc": "\u09002\u05b9c"}, ... ],  
    "ultraviolet": [ {"desc": "\u06700\u05f31"}, ... ],  
    "dressing": [ {"desc": "\u05bd2\u051b7"}, ... ]  
  }  
}  
}
```

Most of the data are arrays, but other than that the content should be self-explanatory.

Next, we just need to parse the JSON format data which should be easy for you.

Now we know that it is feasible to build SunnyWeather, and we're ready to write code. But before that, we can make SunnyWeather an open-source app that is hosted on GitHub, and we will discuss in the last Git Time section of this book.

## 15.2 Git Time: Host Code on GitHub

You should be familiar with Git now. In this section we will take a look at how to host SunnyWeather code on GitHub.

GitHub is the largest code host and apparently uses Git to do version control. Any open-source project can host code on GitHub for free. Its address is <https://github.com/>. The front page of the official website is as shown in Fig. 15.4.

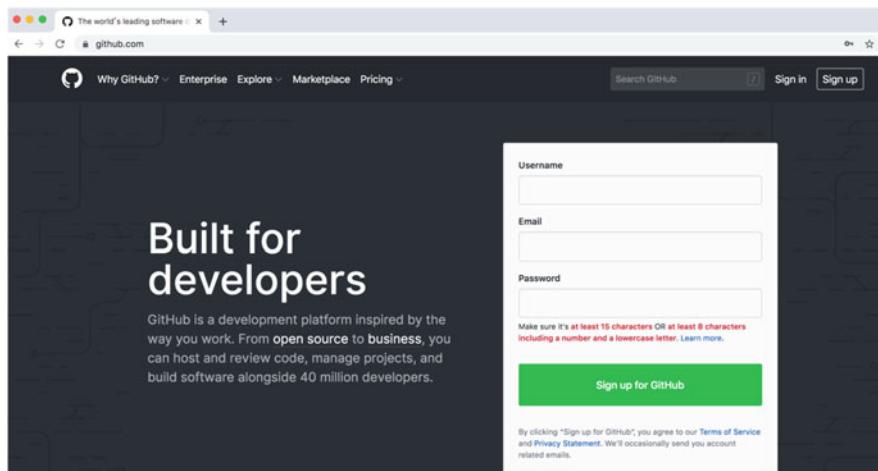


Fig. 15.4 GitHub front page

**Fig. 15.5** Create account

## Create your account

Username \*

 ✓

Email address \*

 ✓

Password \*

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter.  
[Learn more.](#)

Next: Select a plan

First, we need to register for GitHub account, click “Sign up for GitHub,” then fill the user name, email, and password as shown in Fig. 15.5.

Click “Next: Select a plan” to open the plan page, and we simply choose the free one as shown in Fig. 15.6.

Then a survey page will show up as shown in Fig. 15.7.

You can just click “Skip this step” if you are not interested.

Now you can open your email to verify and activate the account. Open GitHub again, and you should be directed to your personal GitHub page as shown in Fig. 15.8.

Click “Start a project” to create a new repository and name it “SunnyWeather,” then select “Initialize this repository with a README,” and add a .gitignore file of Android type then use Apache License 2.0 as the open-source license. This is as shown in Fig. 15.9.

Then click “Create repository” to finish creating SunnyWeather repository as shown in Fig. 15.10. The address for this repo is <https://github.com/guolindev/SunnyWeather>.

As you can see, GitHub generated .gitignore, LICENSE and README.md for us. You can edit the README.md file to edit the description of the repo.

Now we need to create SunnyWeather project in Android Studio, and the package name should be com.sunnyweather.android, as shown in Fig. 15.11.

Next, we need to clone the remote to local. Click “Clone or download” in the front page of the repo to get the Git address of remote repo as shown in Fig. 15.12.

Copy the Git address which should be <https://github.com/guolindev/SunnyWeather.git>.

Then open terminal and change directory to SunnyWeather as shown in Fig. 15.13.

Then clone the repo with command git clone <https://github.com/guolindev/SunnyWeather.git>, as shown in Fig. 15.14.

**Pick the plan that's right for you**

| Individuals  | Teams   |  |   |
|--|---|--|---|
| <br><b>Free</b><br><b>\$0 USD</b><br>Per month<br>The basics of GitHub for every developer<br><a href="#" style="background-color: #0072bc; color: white; padding: 5px 10px; text-decoration: none;">Choose Free</a>  | <br><b>Pro</b><br><b>\$7 USD</b><br>Per month<br>Pro tools for developers with advanced requirements<br><a href="#" style="background-color: #0072bc; color: white; padding: 5px 10px; text-decoration: none;">Choose Pro</a>  |  |   |
| <br><b>Team</b><br><b>\$9 USD</b><br>Per user / month<br>Starts at \$25 and includes 5 users<br>Advanced collaboration and management tools for teams<br><a href="#" style="background-color: #0072bc; color: white; padding: 5px 10px; text-decoration: none;">Choose Team</a> | <br><b>Enterprise</b><br><b>\$21 USD</b><br>Per user / month<br>Security, compliance, and deployment controls for organizations<br><a href="#" style="background-color: #0072bc; color: white; padding: 5px 10px; text-decoration: none;">Start your 14-day free trial</a> |  |   |
| <ul style="list-style-type: none"> <li>✓ Unlimited public repositories</li> <li>✓ Unlimited private repositories</li> <li>✓ Limited to 3 collaborators for private repositories</li> <li>✓ Issues and bug tracking</li> <li>✓ Project management</li> </ul>  | <ul style="list-style-type: none"> <li>✓ Includes everything in Free</li> <li>✓ Unlimited collaborators</li> <li>✓ GitHub Pages</li> <li>✓ Wikis</li> <li>✓ Protected branches</li> <li>✓ Code owners</li> <li>✓ Repository insights</li> </ul>   | <ul style="list-style-type: none"> <li>✓ Unlimited public repositories</li> <li>✓ Unlimited private repositories</li> <li>✓ Team access controls</li> <li>✓ User management and billing</li> <li>✓ Issues and bug tracking</li> <li>✓ Project management</li> <li>✓ Advanced tools and insights</li> </ul> | <ul style="list-style-type: none"> <li>✓ Includes everything in Team</li> <li>✓ Self-hosted or cloud-hosted, or both</li> <li>✓ SAML single sign-on</li> <li>✓ Access provisioning</li> <li>✓ Invoice billing</li> <li>✓ 99.95% uptime SLA</li> <li>✓ Simplified account administration</li> <li>✓ Unified search and contributions</li> <li>✓ Priority support</li> <li>✓ Advanced auditing</li> </ul> |
| Free for<br>Open source teams<br>Academic faculty  | Free for<br>Open source teams<br>Academic faculty   | Free for<br>Open source teams<br>Academic faculty  | Free for<br>Open source teams<br>Academic faculty   |

**Fig. 15.6** Select plan

Once you see the texts as shown in the figure, it means clone is successful, and .gitignore, LICENSE, and README.md have been cloned to local. Get into SunnyWeather folder and use ls -al command to check the files, and it should be as shown in Fig. 15.15.

Now we need to copy paste all the files in this folder to the directory above it so that we can add the folder in the version control. Notice that .git is a hidden folder and do not forget to copy it. The directory above the current one has .gitignore file too, and we can just replace it. After this, you can delete SunnyWeather, and the project directory should be as shown in Fig. 15.16.

Next, we need to submit the existing files to GitHub, and this can be done by the following steps:

Add all the files to version control:

```
git add .
```

Commit changes:

```
git commit -m "First commit."
```

**Fig. 15.7** Survey page

How much programming experience do you have?

|   |                                    |
|---|------------------------------------|
| None<br>I don't program at all                | A little<br>I'm new to programming |
| A moderate amount<br>I'm somewhat experienced | A lot<br>I'm very experienced      |

What do you plan to use GitHub for?  
(Select up to 3)

|                                    |                                    |                                  |
|------------------------------------|------------------------------------|----------------------------------|
|                                    |                                    |                                  |
| Learn to code                      | Learn Git and GitHub               | Host a project (repository)      |
|                                    |                                    |                                  |
| Create a website with GitHub Pages | Find and contribute to open source | School work and student projects |
|                                    |                                    |                                  |
| Use the GitHub API                 | Other                              |                                  |

I am interested in:

We'll connect you with communities and projects that fit your interests.  
For example: webapp ocaml ajax

**Complete setup**

[Skip this step](#)

Push changes to remote repo(GitHub):

```
git push origin master
```

Notice that the last step may require authentication, and you can type in the user name and password. The final result should be as shown in Fig. 15.17.

After push is done, refresh the front page of SunnyWeather, and you should see the files appear as shown in Fig. 15.18.

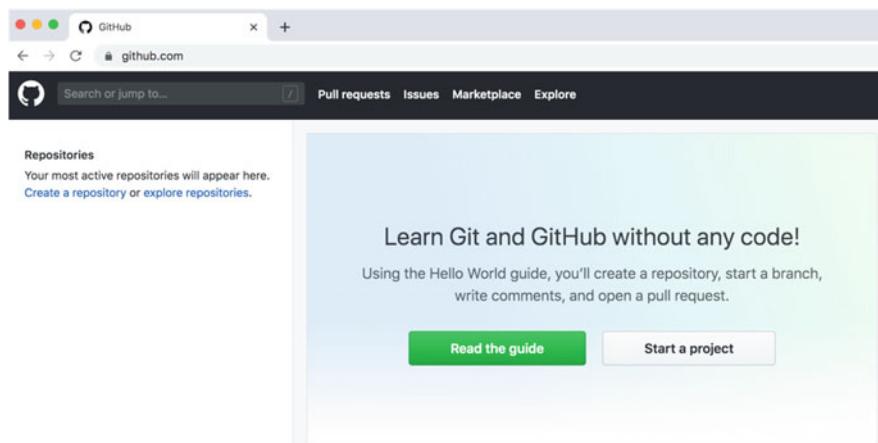


Fig. 15.8 GitHub personal page

Owner **Repository name \***

guolindev / SunnyWeather ✓

Great repository names are short and memorable. Need inspiration? How about **bookish-octo-pancake?**

Description (optional)

**Public**  
Anyone can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

**Initialize this repository with a README**  
This will let you immediately clone the repository to your computer.

Add .gitignore: **Android** ▾ | Add a license: **Apache License 2.0** ▾ ⓘ

**Create repository**

Fig. 15.9 Create repository

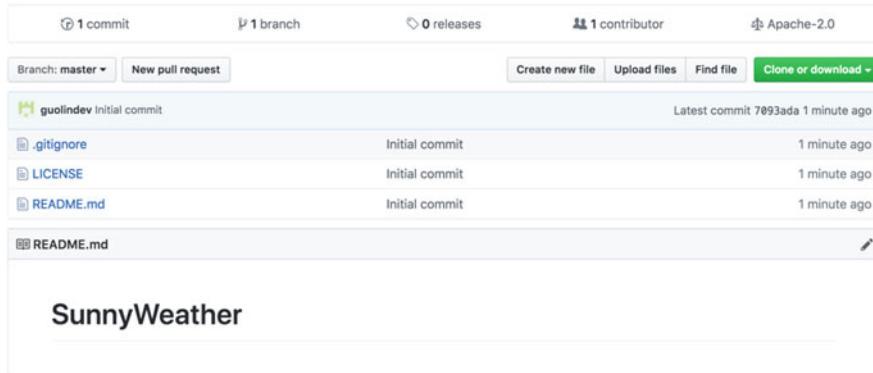


Fig. 15.10 SunnyWeather repo page

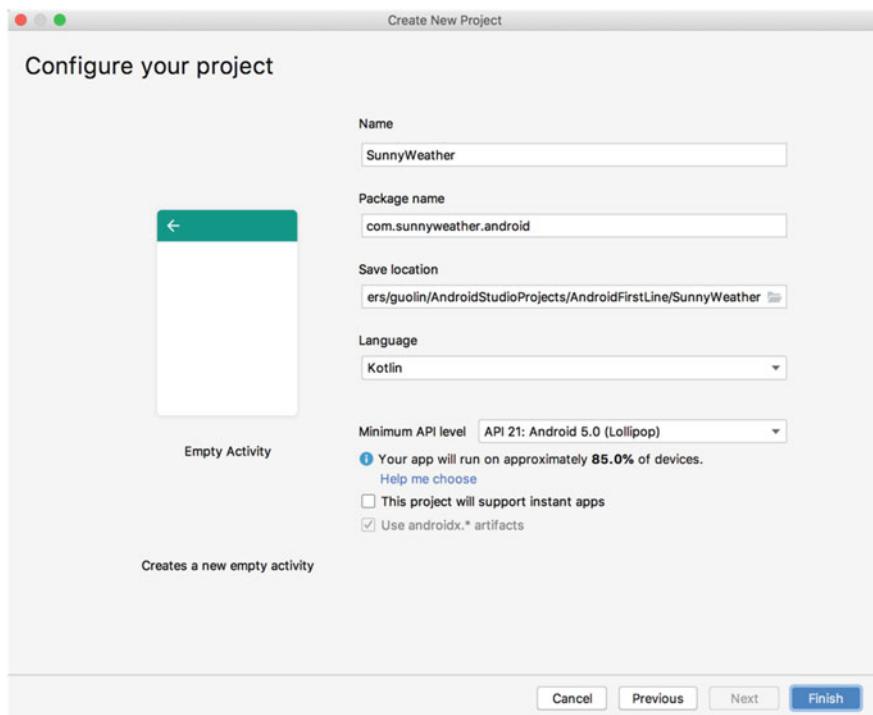
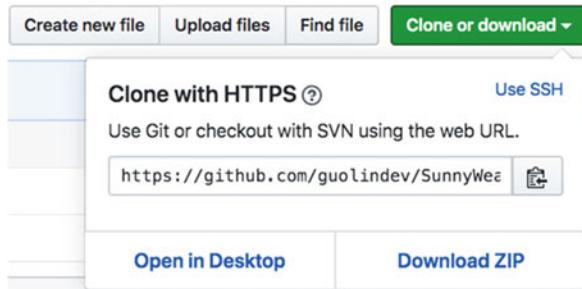


Fig. 15.11 Create SunnyWeather Project

**Fig. 15.12** Git Address of remote repo



```
[guolindeMacBook-Pro:~ guolin$ cd AndroidStudioProjects/AndroidFirstLine/SunnyWeather/
guolindeMacBook-Pro:SunnyWeather guolin$ ]
```

**Fig. 15.13** Change Directory to SunnyWeather

```
guolindeMacBook-Pro:SunnyWeather guolin$ git clone https://github.com/guolindev/SunnyWeather.git
Cloning into 'SunnyWeather'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
guolindeMacBook-Pro:SunnyWeather guolin$ ]
```

**Fig. 15.14** Clone remote repo to local

```
[guolindeMacBook-Pro:SunnyWeather guolin$ cd SunnyWeather
[guolindeMacBook-Pro:SunnyWeather guolin$ ls -al
total 40
drwxr-xr-x  6 guolin  staff   192 10 14 22:06 .
drwxr-xr-x  15 guolin  staff   480 10 14 22:06 ..
drwxr-xr-x  12 guolin  staff   384 10 14 22:06 .git
-rw-r--r--  1 guolin  staff  1002 10 14 22:06 .gitignore
-rw-r--r--  1 guolin  staff 11357 10 14 22:06 LICENSE
-rw-r--r--  1 guolin  staff    14 10 14 22:06 README.md
guolindeMacBook-Pro:SunnyWeather guolin$ ]
```

**Fig. 15.15** Check the cloned files

## 15.3 Introduction to MVVM

You probably can recall that in Chap. 13, we mentioned that a lot of the Jetpack architecture components are designed for MVVM architectural pattern. So, what is MVVM architectural pattern? How to create a project with this pattern? We will cover these in this section.

MVVM is short for Model-View-ViewModel and is an advanced architectural pattern widely used in Android apps besides similar architectural pattern such as

```
[guolin@MacBook-Pro: SunnyWeather guolin$ ls -al
total 104
drwxr-xr-x 17 guolin  staff   544 10 14 22:15 .
drwxr-xr-x 34 guolin  staff  1088 10 14 21:57 ..
drwxr-xr-x 12 guolin  staff   384 10 14 22:06 .git
-rw-r--r--  1 guolin  staff  1002 10 14 22:06 .gitignore
drwxr-xr-x  5 guolin  staff   160 10 14 21:57 .gradle
drwxr-xr-x 10 guolin  staff   320 10 14 21:57 .idea
-rw-r--r--  1 guolin  staff  11357 10 14 22:06 LICENSE
-rw-r--r--  1 guolin  staff    14 10 14 22:06 README.md
-rw-r--r--  1 guolin  staff   831 10 14 21:57 SunnyWeather.iml
drwxr-xr-x  8 guolin  staff   256 10 14 21:57 app
-rw-r--r--  1 guolin  staff   661 10 14 21:57 build.gradle
drwxr-xr-x  3 guolin  staff    96 10 14 21:57 gradle
-rw-r--r--  1 guolin  staff  1169 10 14 21:57 gradle.properties
-rwxr--r--  1 guolin  staff  5296 10 14 21:57 gradlew
-rw-r--r--  1 guolin  staff  2260 10 14 21:57 gradlew.bat
-rw-r--r--  1 guolin  staff   436 10 14 21:57 local.properties
-rw-r--r--  1 guolin  staff    47 10 14 21:57 settings.gradle
guolin@MacBook-Pro: SunnyWeather guolin$ ]
```

Fig. 15.16 SunnyWeather Project directory structure

```
[guolin@MacBook-Pro: SunnyWeather guolin$ git push origin master
Counting objects: 70, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (52/52), done.
Writing objects: 100% (70/70), 126.23 KiB | 9.02 MiB/s, done.
Total 70 (delta 0), reused 0 (delta 0)
To https://github.com/guolindev/SunnyWeather.git
  00cf416..2d871ee master -> master
guolin@MacBook-Pro: SunnyWeather guolin$ ]
```

Fig. 15.17 Sync committed changes to remote repo

MVP, MVC, and so on. With this pattern, there are three components in the app: Model is the data, View is the UI that presenting the data, while ViewModel is the bridge between data model and UI. This separates the business logic and UI presentation.

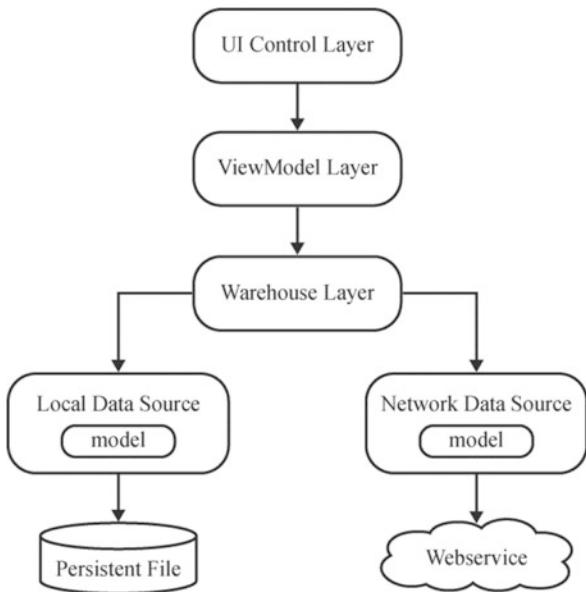
Of course, a good project architecture should also have data storage, data source, and so on. Figure 15.19 shows the diagram of project that adopts MVVM architecture.

There are a few layers. The UI layer has Activity, Fragment, layout files, and other UI-related components. ViewModel layer has the data that are related to UI and ensures that data can persist between configuration changes. It also provides

| Tony First commit. |                | Latest commit 2d871ee 4 minutes ago |
|--------------------|----------------|-------------------------------------|
| .idea              | First commit.  | 4 minutes ago                       |
| app                | First commit.  | 4 minutes ago                       |
| gradle/wrapper     | First commit.  | 4 minutes ago                       |
| .gitignore         | Initial commit | 38 minutes ago                      |
| LICENSE            | Initial commit | 38 minutes ago                      |
| README.md          | Initial commit | 38 minutes ago                      |
| build.gradle       | First commit.  | 4 minutes ago                       |
| gradle.properties  | First commit.  | 4 minutes ago                       |
| gradlew            | First commit.  | 4 minutes ago                       |
| gradlew.bat        | First commit.  | 4 minutes ago                       |
| settings.gradle    | First commit.  | 4 minutes ago                       |

**Fig. 15.18** Check committed changes in GitHub

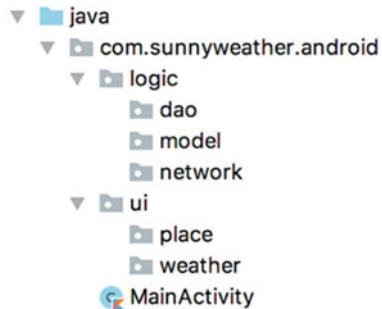
**Fig. 15.19** MVVM architectural pattern diagram



interface that UI layer can call and communicate with data storage. The data storage layer mainly determines where to get the data for the data request and then return the data to the caller. Local data source can be database, SharedPreferences, and so on, while network data resource is usually done with Webservice interface provided by the server through Retrofit.

Notice that the arrows are one-direction arrows, for instance, UI layer points to ViewModel layer. This means that UI layer has reference of ViewModel layer and the opposite is false. Also reference cannot skip layers, for instance, UI layer cannot have reference to the data storage layer. Each layer can only interact with the layer directly above or below it.

**Fig. 15.20** New project structure



Next, let us build the project by strictly following this pattern. To have a better structure, we need to create a few packages in com.sunnyweather.android as shown in Fig. 15.20.

As the names imply, logic package contains the business logic, and ui package contains UI-related code. Inside logic package, we have dao, model, and network folders which are used to store data access object, object model, and network-related code, respectively. The ui folder has place and weather folders which contains the two main screens, respectively.

We will use a lot of libs, and now we can add all the dependencies at the same time. Edit app/build.gradle and add the following dependencies in dependencies closure:

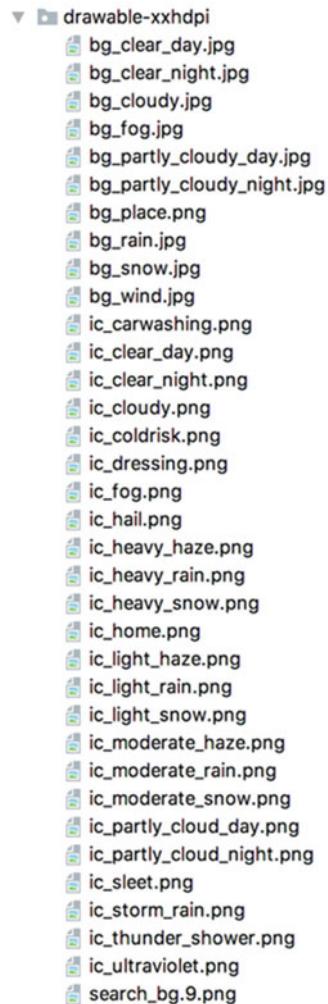
```

dependencies {
    ...
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    implementation "androidx.lifecycle:lifecycle-extensions:2.1.0"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.2.0"
    implementation 'com.google.android.material:material:1.0.0'
    implementation 'com.squareup.retrofit2:retrofit:2.6.1'
    implementation 'com.squareup.retrofit2:converter-gson:2.6.1'
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.3.0"
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
android:1.1.1"
}
    
```

We have used all of them in previous chapters, and they should look similar to you.

To make SunnyWeather more beautiful, I prepared a few pictures which you can download from the resources. Now we need to put them in drawable-xxhdpi folder as shown in Fig. 15.21.

That is all we need for the preparation work. Let us begin building the app.

**Fig. 15.21** Image resources

## 15.4 Search City Data

Based on the previous analysis, in order to get the weather data, we first need to get the data of the specific city and then get the longitude and latitude information of the city. Thus the step is to get the city data. As mentioned before, we will have business logic and UI and let us start with business logic.

### 15.4.1 Business Logic Implementation

In MVVM, since ViewModel does not have reference of Activity, we can use what we discussed in Chap. 14 to get Context globally.

Create SunnyWeatherApplication class in com.sunnyweather.android package, as shown in code below:

```
class SunnyWeatherApplication : Application() {

    companion object {
        @SuppressLint("StaticFieldLeak")
        lateinit var context: Context
    }

    override fun onCreate() {
        super.onCreate()
        context = applicationContext
    }

}
```

Then we need to set this class in AndroidManifest.xml as shown below:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="com.sunnyweather.android">
    <application
        android:name=".SunnyWeatherApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
    </application>
</manifest>
```

Now we can get Context object at any call sites by using SunnyWeatherApplication.context.

We can also put the token value in SunnyWeatherApplication to make it more accessible, as shown in code below:

```
class SunnyWeatherApplication : Application() {
    companion object {
        const val TOKEN = "fill in your token value"
        ...
    }
}
```

```
    } ...
```

Now we can start from the bottom layer as shown in Fig. 15.18. First, define data model. Create PlaceResponse.kt in logic/model package with the following code:

```
data class PlaceResponse(val status: String, val places: List<Place>)

data class Place(val name: String, val location: Location,
    @SerializedName("formatted_address") val address: String)

data class Location(val lng: String, val lat: String)
```

The classes and fields are defined based on the search result which is JSON data in Sect. 15.1. However, some fields' name in JSON do not follow Kotlin naming conventions; thus `@SerializedName` annotation is used to create mapping between JSON fields and Kotlin fields.

Once we have the data model, we can start working on networks. First, define a Retrofit interface to access the city search API. Create PlaceService interface in logic/network package with the following code:

```
interface PlaceService {
    @GET("v2/place?token=${SunnyWeatherApplication.TOKEN}
    &lang=zh_CN")
    fun searchPlaces(@Query("query") query: String):
    Call<PlaceResponse>
}
```

The `@GET` annotation will make Retrofit send a GET request to the address specified in the annotation whenever `searchPlaces()` is called. Most of the params for API for searching city data do not change except query argument; thus we used `@Query` annotation to pass value to this param. The other two params do not change and are hardcoded in `@GET` annotation.

The return value of `searchPlaces()` is declared as `Call<PlaceResponse>`; thus Retrofit will automatically parse the response of JSON data to `PlaceResponse` object.

To use `PlaceService` interface, we need to create a Retrofit creator. Create `ServiceCreator` singleton in logic/network package as shown below:

```
object ServiceCreator {
    private const val BASE_URL = "https://api.caiyunapp.com/"

    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
```

```

    .addConverterFactory(GsonConverterFactory.create())
    .build()

    fun <T> create(serviceClass: Class<T>) : T = retrofit.create
    (serviceClass)

    inline fun <reified T> create() : T = create(T::class.java)

}

```

This Retrofit creator follows the pattern we discussed in Sect. 11.6.3, so it should not be difficult to understand.

Next, we need to encapsulate all the network request API by defining a unified network data access interface. Create `SunnyWeatherNetwork` singleton in logic/network package with the following code:

```

object SunnyWeatherNetwork {

    private val placeService = ServiceCreator.create(PlaceService::
    class.java)

    suspend fun searchPlaces(query: String) = placeService.
    searchPlaces(query).await()

    private suspend fun <T> Call<T>.await(): T {
        return suspendCoroutine { continuation ->
            enqueue(object : Callback<T> {
                override fun onResponse(call: Call<T>, response: Response<T>) {
                    val body = response.body()
                    if (body != null) continuation.resume(body)
                    else continuation.resumeWithException(
                        RuntimeException("response body is null"))
                }

                override fun onFailure(call: Call<T>, t: Throwable) {
                    continuation.resumeWithException(t)
                }
            })
        }
    }
}

```

This is an important class that demonstrates lots of advanced techniques. Let us take a deep dive into it.

First, the `ServiceCreator` creates a dynamic delegate object of `PlaceService` interface. Then the `searchPlaces()` function calls the `searchPlaces()` method defined in `PlaceService` to send the request to search city data.

In order to make the code more succinct, we applied the technique discussed in Sect. 11.7.3 to simplify the Retrofit callback. Since we need to use coroutines, we define an await() function and declare searchPlaces() to be a suspending function. The implementation of await() has been discussed in Sect. 11.7.3.

Now, when searchPlaces() of SunnyWeatherNetwork is called, Retrofit will send the network request and block the current coroutine. Once server response is received, await() will parse the data to data model and return the value and unblock the current coroutine. The searchPlaces() will return the return value of await() to the caller.

That is all we need for networks, and we can start to work on data storage. As aforementioned, this layer determines if the requested data should be acquired from local data source or from network and then return the data. Thus, it is functioning like a middle layer between the data request layer and cache; if data is not cached locally, then get from network, and otherwise just return the cached data.

But I do not think we need to cache the city data in our app. Create Repository singleton as the unified interface for data storage layer with code shown below:

```
object Repository {

    fun searchPlaces(query: String) = liveData(Dispatchers.IO) {
        val result = try {
            val placeResponse = SunnyWeatherNetwork.searchPlaces(query)
            if (placeResponse.status == "ok") {
                val places = placeResponse.places
                Result.success(places)
            } else {
                Result.failure(RuntimeException("response status is
                    ${placeResponse.status}"))
            }
        } catch (e: Exception) {
            Result.failure<List<Place>>(e)
        }
        emit(result)
    }
}
```

In order to return the data acquired asynchronously to the caller, usually the methods defined in the data storage layer will return a LiveData object. We discussed LiveData in Sect. 13.4, and here some new techniques are applied. The liveData() method is provided by lifecycle-livedata-ktx, and it can automatically create and return a LiveData object and provide context of a suspending function. This means that we can call any suspending function in liveData(). Here, searchPlaces() of SunnyWeatherNetwork is called to get the city data, and if the response state is ok, then Result.success() is used to get the city data array, and if not Result.failure() is called to create an exception and emit this exception with emit(). The emit()

method is similar to `setValue()` in `LiveData`, but since we cannot get the `LiveData` object here, `lifecycle-livedata-ktx` provides this alternative.

Notice that the thread param type for `liveData()` is `Dispatchers.IO` so that the code inside it runs in worker thread. Android does not allow time-consuming operations like network request, and database operation runs in main thread; thus it is necessary to switch thread in data storage layer.

The last step for business logic layer is to define the `ViewModel` layer. This layer serves as the bridge between business logic and UI layer and more toward the business logic layer. Since `ViewModel` usually has 1-1 mapping with `Activity` or `Fragment`, thus we put them together.

Create `PlaceViewModel` in `ui/place` package with the code below:

```
class PlaceViewModel : ViewModel() {  
  
    private val searchLiveData = MutableLiveData<String>()  
  
    val placeList = ArrayList<Place>()  
  
    val placeLiveData = Transformations.switchMap(searchLiveData) {  
        query ->  
            Repository.searchPlaces(query)  
    }  
  
    fun searchPlaces(query: String) {  
        searchLiveData.value = query  
    }  
}
```

`PlaceViewModel` also defines `searchPlaces()`, but this method does not call `searchPlaces()` method in the data storage layer but instead pass the `query` argument to `searchLiveData` and call `switchMap()` of `Transformations` to observe this object. Without doing this the `LiveData` object returned from data storage layer is not observable which we have discussed in Sect. 13.4.2. When `searchPlaces()` is called, the corresponding transformation function of `switchMap()` will be called. In the transformation function, we just need to call the `searchPlaces()` method defined in data storage layer to send the network request and transform the returned `LiveData` object to a `LiveData` object that is observable by the `Activity`.

The `placeList` collection caches the city data that is presented on the screen because per the MVVM guideline, we should put these data in `ViewModel` so that it is not lost after configuration changes. We will use this collection later in the UI code.

That is the end of business logic, and now `SunnyWeather` is able to search the city data, and let us start to build the UI.

### 15.4.2 UI Implementation

Building UI usually starts with creating layout, and since we will reuse searching city data, it is not recommended to place this in the Activity but should be in the Fragment so that we can reuse it.

Create fragment\_place.xml in res/layout folder with the following code:

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

    <ImageView
        android:id="@+id/bgImageView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:src="@drawable/bg_place"/>

    <FrameLayout
        android:id="@+id/actionBarLayout"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:background="@color/colorPrimary">

        <EditText
            android:id="@+id/searchPlaceEdit"
            android:layout_width="match_parent"
            android:layout_height="40dp"
            android:layout_gravity="center_vertical"
            android:layout_marginStart="10dp"
            android:layout_marginEnd="10dp"
            android:paddingStart="10dp"
            android:paddingEnd="10dp"
            android:hint="Input Address"
            android:background="@drawable/search_bg"/>
    </FrameLayout>

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@+id/actionBarLayout"
        android:visibility="gone"/>
</RelativeLayout>
```

There are mainly two parts in this layout. The EditText view provides user a search box, while the RecyclerView displays the search results. The ImageView

provides a background to make the app looks more beautiful but not really functional component.

For simplicity, I will hardcode all the texts which are not recommended as the proper way is to define the strings in strings.xml and reference them in the layout.

Since we use RecyclerView, we need to define its item view. Create place\_item.xml in layout folder with the following code:

```
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="130dp"
    android:layout_margin="12dp"
    app:cardCornerRadius="4dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="18dp"
        android:layout_gravity="center_vertical">

        <TextView
            android:id="@+id/placeName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="?android:attr/textColorPrimary"
            android:textSize="20sp"/>

        <TextView
            android:id="@+id/placeAddress"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:textColor="?android:attr/textColorSecondary"
            android:textSize="14sp"/>

    </LinearLayout>

</com.google.android.material.card.MaterialCardView>
```

The MaterialCardView is the parent layout which makes all the items in RecyclerView display in card. Inside the card, there are two TextViews to display the location name and detail address.

After defining the item view, we need to create adapter for the RecyclerView. Create PlaceAdapter in ui/place folder. Make this adapter inherits Recyclerview.Adapter and specify the generic type to PlaceAdapter.ViewHolder, as shown in code below:

```
class PlaceAdapter(private val fragment: Fragment, private val placeList: List<Place>) :  
    RecyclerView.Adapter<PlaceAdapter.ViewHolder>() {  
  
    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view)  
    {  
        val placeName: TextView = view.findViewById(R.id.placeName)  
        val placeAddress: TextView = view.findViewById(R.id.placeAddress)  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
        ViewHolder {  
        val view = LayoutInflater.from(parent.context).inflate(R.layout.  
            place_item,  
            parent, false)  
        return ViewHolder(view)  
    }  
  
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
        val place = placeList[position]  
        holder.placeName.text = place.name  
        holder.placeAddress.text = place.address  
    }  
  
    override fun getItemCount() = placeList.size  
}
```

This is the standard way to create RecyclerView adapter, and we have implemented similar code several time; thus there is no need for explanation here.

After this, we need to implement the Fragment. Create PlaceFragment in ui/place folder and make it inherit the Fragment in AndroidX, as shown in code below:

```
class PlaceFragment : Fragment() {  
  
    val viewModel by lazy { ViewModelProviders.of(this).get  
(PlaceViewModel::class.java) }  
  
    private lateinit var adapter: PlaceAdapter  
  
    override fun onCreateView(inflater: LayoutInflater, container:  
        ViewGroup?,  
        savedInstanceState: Bundle?): View? {  
        return inflater.inflate(R.layout.fragment_place, container, false)  
    }  
  
    override fun onActivityCreated(savedInstanceState: Bundle?) {  
        super.onActivityCreated(savedInstanceState)  
        val layoutManager = LinearLayoutManager(activity)  
        recyclerView.layoutManager = layoutManager
```

```

        adapter = PlaceAdapter(this, viewModel.placeList)
        recyclerView.adapter = adapter
        searchPlaceEdit.addTextChangedListener { editable ->
            val content = editable.toString()
            if (content.isNotEmpty()) {
                viewModel.searchPlaces(content)
            } else {
                recyclerView.visibility = View.GONE
                bgImageView.visibility = View.VISIBLE
                viewModel.placeList.clear()
                adapter.notifyDataSetChanged()
            }
        }
        viewModel.placeLiveData.observe(this, Observer{ result ->
            val places = result.getOrNull()
            if (places != null) {
                recyclerView.visibility = View.VISIBLE
                bgImageView.visibility = View.GONE
                viewModel.placeList.clear()
                viewModel.placeList.addAll(places)
                adapter.notifyDataSetChanged()
            } else {
                Toast.makeText(activity, "Cannot find any result",
                    Toast.LENGTH_SHORT).show()
                result.exceptionOrNull()?.printStackTrace()
            }
        })
    }
}

```

The PlaceViewModel instance is acquired through late init by using lazy function. This is recommended as we can use viewModel variable anywhere in the class without worrying the initialization of this variable.

The fragment\_place layout is inflated in onCreateView() which is the standard way of using Fragment.

Inside onActivityCreated(), we set the LayoutManager and adapter to the RecyclerView and use the placeList collection in PlaceViewModel as data source. Then we call addTextChangedListener() of EditText to listen the changes of the search box. When the text inside the search bar changes, we pass the new text to searchPlaces() of PlaceViewModel so that a new request for city data will be sent. When the search box is empty, the RecylerView will be hidden, and background image will show up.

After sending request, we need to get the response which needs LiveData. Here we observe placeLiveData in PlaceViewModel, and when data changes, the Observer interface implementation will be called. Then, in the callback, if the data is not empty, it will be added to the placeList collection and notify PlaceAdapter to refresh the UI, and if the data is empty, something wrong happens, and a Toast message is shown and print the exception reason.

That is what we need to do to search city data. However, Fragment cannot display in the UI directly, and we need to attach it to some Activity. Edit activity\_main.xml as shown in code below:

```
<FrameLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <fragment  
        android:id="@+id/placeFragment"  
        android:name="com.sunnyweather.android.ui.place.PlaceFragment"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent" />  
  
</FrameLayout>
```

This FrameLayout simply adds the PlaceFragment and fill the whole layout.

Since we already defined search bar in PlaceFragment, there is no need to use the native ActionBar. Edit res/values/styles.xml as shown below:

```
<resources>  
  
    <!-- Base application theme. -->  
    <style name="AppTheme" parent="Theme.AppCompat.Light.  
NoActionBar">  
        ...  
    </style>  
  
</resources>
```

Before we can run the app, we need to ask for permissions. Edit AndroidManifest.xml as shown below:

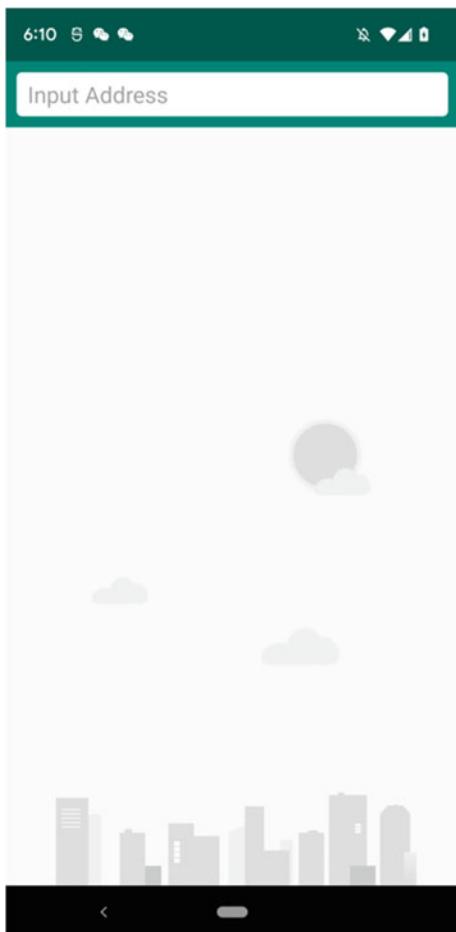
```
<manifest xmlns:android="http://schemas.android.com/apk/res/  
android"  
    package="com.sunnyweather.android">  
  
    <uses-permission android:name="android.permission.INTERNET" />  
    ...  
</manifest>
```

We need to add this permission request because we get the data through network. Run the app, and the initial screen should look like Fig. 15.22.

Then we can search any city in the world, and the corresponding data will display on the screen as shown in Fig. 15.23.

Though only the name and address are shown, we actually got the longitude and latitude information in the response which we can use for weather forecast.

**Fig. 15.22** Initial screen of PlaceFragment



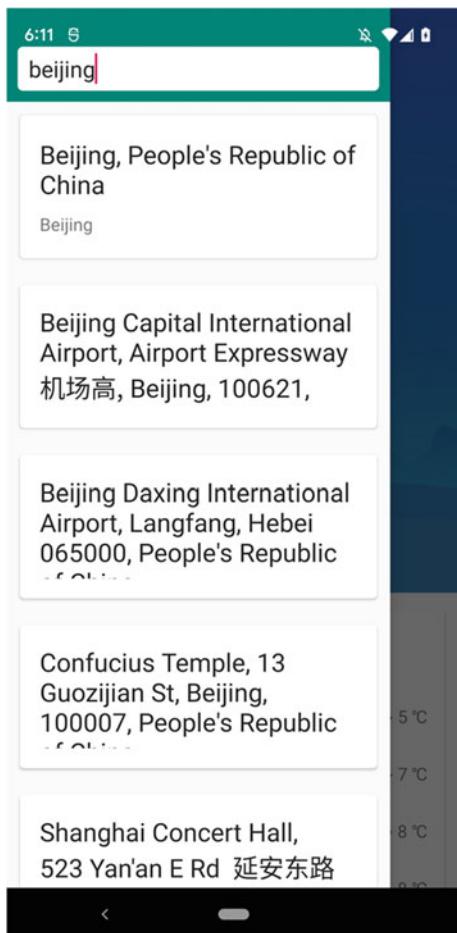
It seems that we can simply put the code inside Fragment instead of abstracting into layers. It indeed will work and is the way most starters will do. However, as the app is getting more complicated, the Activity and Fragment's size will increase and gradually becomes not maintainable. The MVVM architecture makes the structure of the codebase easy to understand and easy to extend, and you should see this when building the weather forecast functionality.

That is everything we need to for Phase I. Let's commit the code. First, add all the files in version control:

```
git add .
```

Then commit the code:

```
git commit -m "Implement search for city data"
```

**Fig. 15.23** Data for Beijing

Lastly, sync the changes to remote repo:

```
git push origin master
```

OK! That is it for Phase I and let us start Phase II.

## 15.5 Display Weather Data

To display the weather data, we also need to implement the business logic and UI and still start with business logic.

### 15.5.1 Business Logic Implement

The Caiyu API returns various weather data, and we cannot display all of them; thus I will select some more important ones to display to user.

The simplified JSON response is as shown below:

```
{
  "status": "ok",
  "result": {
    "realtime": {
      "temperature": 23.16,
      "skycon": "WIND",
      "air_quality": {
        "aqi": { "chn": 17.0 }
      }
    }
  }
}
```

Then we should create data model based on this JSON. Create `RealtimeResponse.kt` with the following code:

```
data class RealtimeResponse(val status: String, val result: Result) {

  data class Result(val realtime: Realtime)

  data class Realtime(val skycon: String, val temperature: Float,
                     @SerializedName("air_quality") val airQuality: AirQuality)

  data class AirQuality(val aqi: AQI)

  data class AQI(val chn: Float)

}
```

Notice that all the data classes are defined inside `RealtimeResponse` class to prevent name conflicts with other data models.

The simplified JSON response for weather forecast data is as shown below:

```
{
  "status": "ok",
  "result": {
    "daily": [
      "temperature": [ { "max": 25.7, "min": 20.3}, ... ],
      "skycon": [ { "value": "CLOUDY", "date": "2019-10-20T00:00+08:00"}, ... ],
      "life_index": {
        "coldRisk": [ { "desc": "\u06781\u06613\u053d1" }, ... ],
        ...
      }
    ]
  }
}
```

```

    "carWashing": [ {"desc": "\u9002\u5b9c"}, ... ],
    "ultraviolet": [ {"desc": "\u6700\u5f31"}, ... ],
    "dressing": [ {"desc": "\u5bd2\u51b7"}, ... ]
}
}
}
}
```

The specialty of this JSON response is that all the data are arrays, and each element in the array is 1 day's data. In the data model, we can use List collection to map the JSON array. Create DailyResponse.kt in logic/model folder with the following code:

```

data class DailyResponse(val status: String, val result: Result) {

    data class Result(val daily: Daily)

    data class Daily(val temperature: List<Temperature>, val skycon:
List<Skycon>,
                    @SerializedName("life_index") val lifeIndex: LifeIndex)

    data class Temperature(val max: Float, val min: Float)

    data class Skycon(val value: String, val date: Date)

    data class LifeIndex(val coldRisk: List<LifeDescription>, val
carWashing:
                    List<LifeDescription>, val ultraviolet:
List<LifeDescription>,
                    val dressing: List<LifeDescription>)

    data class LifeDescription(val desc: String)

}
```

We also define all the data model classes inside DailyResponse, and as you can tell, RealtimeResponse also defines Result class, and by putting them into different classes, there is no conflict between them.

We also need to define Weather class in logic/model folder to encapsulate Realtime and Daily objects, as shown in code below:

```
data class Weather(val realtime: RealtimeResponse.Realtime, val
daily: DailyResponse.Daily)
```

After definition of the data model, we need to implement network-related code. Now, you should find that with the layered structure, each step is very clear.

Now, define a Retrofit interface that can access the weather data API. Create WeatherService interface in logic/network folder, as shown in code below:

```

interface WeatherService {

    @GET("v2.5/${SunnyWeatherApplication.TOKEN}/{lng},{lat}/
realtime.json")
    fun getRealtimeWeather(@Path("lng") lng: String, @Path("lat") lat:
String):
        Call<RealtimeResponse>

    @GET("v2.5/${SunnyWeatherApplication.TOKEN}/{lng},{lat}/daily.
json")
    fun getDailyWeather(@Path("lng") lng: String, @Path("lat") lat:
String):
        Call<DailyResponse>

}

```

The `getRealtimeWeather()` and `getDailyWeather()` are self-explanatory, and both use `@GET` annotation to specify the API interface and use `@Path` annotation to pass in the longitude and latitude information. The return values are `Call<RealtimeResponse>` and `Call<DailyResponse>`.

Next, we need to encapsulate the `WeatherService` interface in `SunnyWeatherNetwork`. Edit `SunnyWeatherNetwork` as shown in code below:

```

object SunnyWeatherNetwork {

    private val weatherService = ServiceCreator.create
    (WeatherService::class.java)

    suspend fun getDailyWeather(lng: String, lat: String) =
        weatherService.getDailyWeather(lng, lat).await()

    suspend fun getRealtimeWeather(lng: String, lat: String) =
        weatherService.getRealtimeWeather(lng, lat).await()
    ...
}

```

The encapsulation to `WeatherService` is almost the same as to `PlaceService` which proves the benefits of layered structure again as we can extend new functionality easily.

After network implementation, we need to implement the data storage layer. Edit `Repository` as shown below:

```

object Repository {
    ...
    fun refreshWeather(lng: String, lat: String) = liveData
    (Dispatchers.IO) {
        val result = try {
            coroutineScope {
                val deferredRealtime = async {

```

```
        SunnyWeatherNetwork.getRealtimeWeather(lng, lat)
    }
    val deferredDaily = async {
        SunnyWeatherNetwork.getDailyWeather(lng, lat)
    }
    val realtimeResponse = deferredRealtime.await()
    val dailyResponse = deferredDaily.await()
    if (realtimeResponse.status == "ok" && dailyResponse.status == "ok") {
        val weather = Weather(realtimeResponse.result.realtime,
                               dailyResponse.result.daily)
        Result.success(weather)
    } else {
        Result.failure(
            RuntimeException(
                "realtime response status is ${realtimeResponse.status}" +
                "daily response status is ${dailyResponse.status}"
            )
        )
    }
}
} catch (e: Exception) {
    Result.failure<Weather>(e)
}
emit(result)
}
}
```

In the data storage layer, we use refreshWeather() to refresh weather both for real-time weather and forecast. This is because it is redundant to call two methods to get the weather data for the caller and it is better we can encapsulate this in data storage layer.

We can run the requests to get the real-time data and forecast data at the same time; thus we can use two async functions to send the requests and call await() to ensure that only both requests succeed can we continue. Since async function can only be called in coroutine scope, coroutineScope function is called.

Next, after getting RealtimeResponse and DailyResponse, if both of their response states are ok, then these two objects will be encapsulated into one Weather object, and Result.success() method will encapsulate this Weather object. If the state is not ok, Result.failure() will be called to encapsulate the error information, and emit() is called to emit the exception.

There is room to optimize here. Since we used coroutine to simplify network callback and every network request may throw exception, we have to try and catch for each every one of the network requests. This apparently is duplicate, and as mentioned before, we can handle this in one place, and we can do it with the following code:

```

object Repository {

    fun searchPlaces(query: String) = fire(Dispatchers.IO) {
        val placeResponse = SunnyWeatherNetwork.searchPlaces(query)
        if (placeResponse.status == "ok") {
            val places = placeResponse.places
            Result.success(places)
        } else {
            Result.failure(RuntimeException("response status is
${placeResponse.status}"))
        }
    }

    fun refreshWeather(lng: String, lat: String) = fire(Dispatchers.IO) {
        coroutineScope {
            val deferredRealtime = async {
                SunnyWeatherNetwork.getRealtimeWeather(lng, lat)
            }
            val deferredDaily = async {
                SunnyWeatherNetwork.getDailyWeather(lng, lat)
            }
            val realtimeResponse = deferredRealtime.await()
            val dailyResponse = deferredDaily.await()
            if (realtimeResponse.status == "ok" && dailyResponse.status ==
"ok") {
                val weather = Weather(realtimeResponse.result.realtime,
                    dailyResponse.result.daily)
                Result.success(weather)
            } else {
                Result.failure(
                    RuntimeException(
                        "realtime response status is ${realtimeResponse.status}" +
                            "daily response status is ${dailyResponse.status}"
                    )
                )
            }
        }
    }

    private fun <T> fire(context: CoroutineContext, block: suspend () ->
Result<T>) =
        liveData<Result<T>>(context) {
            val result = try {
                block()
            } catch (e: Exception) {
                Result.failure<T>(e)
            }
            emit(result)
        }
    }
}

```

The key change here is `fire()` which is a higher-order function defined based on `liveData()` params. Inside `fire()` and `liveData()` are called, and in code block of

liveData(), try and catch are used. The try block will run the Lambda expression and emit the Lambda expression evaluated value.

Notice that the code block in liveData() has the context of suspending function, while inside the Lambda expression, there is no suspending function context although Lambda expression is running in the suspending function. To solve this issue, we need to add the suspend modifier to make the Lambda expression also has the suspending function context.

The rest is simple, and we can simply replace the liveData() function in searchPlaces() and refreshWeather() to fire() and remove the try and catch and emit() usage. Now the data storage layer is much more succinct.

The last step for logic layer is to define ViewModel layer. Create WeatherViewModel in ui/weather package as shown below:

```
class WeatherViewModel : ViewModel() {  
  
    private val locationLiveData = MutableLiveData<Location>()  
  
    var locationLng = ""  
  
    var locationLat = ""  
  
    var placeName = ""  
  
    val weatherLiveData = Transformations.switchMap(locationLiveData)  
    { location ->  
        Repository.refreshWeather(location.lng, location.lat)  
    }  
  
    fun refreshWeather(lng: String, lat: String) {  
        locationLiveData.value = Location(lng, lat)  
    }  
}
```

This class is also extremely simple. The refreshWeather() method will refresh weather data and encapsulate the longitude and latitude information to a Location object and assign it to locationLiveData object then call switchMap() of Transformations to observe this object. Inside the transformation function of switchMap() method, the refreshWeather() defined in data storage layer is called; thus the LiveData object can be transformed to a LiveData object that is observable by the Activity.

The locationLng, locationLat, and placename are variables that UI layer will use to present. By putting them in ViewModel, we can make sure that they can survive the configuration change.

That is for business logic layer; next we need to implement the UI.

### 15.5.2 Implement UI Layer

First, we need to create an Activity that presents weather data. Right click ui/weather package→New→Activity→Empty Activity and create WeatherActivity then set the layout to be activity\_weather.xml.

Since all the weather data will display in one screen, activity\_weather.xml will be very long. In order to make the code more clear, I will apply what we have discussed in Sect. 4.4.1 to write the different parts of the component in different layout files and use these layout in activity\_weather.xml.

Right click res/layout→New→Layout resource file, create now.xml as the real-time weather layout, and add the following code:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/nowLayout"
    android:layout_width="match_parent"
    android:layout_height="530dp"
    android:orientation="vertical">

    <FrameLayout
        android:id="@+id/titleLayout"
        android:layout_width="match_parent"
        android:layout_height="70dp">

        <TextView
            android:id="@+id/placeName"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="60dp"
            android:layout_marginEnd="60dp"
            android:layout_gravity="center"
            android:singleLine="true"
            android:ellipsize="middle"
            android:textColor="#fff"
            android:textSize="22sp" />

    </FrameLayout>

    <LinearLayout
        android:id="@+id/bodyLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:orientation="vertical">

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

```
        android:layout_gravity="center_horizontal"
        android:textColor="#fff"
        android:textSize="70sp" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_marginTop="20dp">

        <TextView
            android:id="@+id/currentSky"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_color="#fff"
            android:textSize="18sp" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="13dp"
            android:layout_color="#fff"
            android:textSize="18sp"
            android:text="|" />

        <TextView
            android:id="@+id/currentAQI"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="13dp"
            android:layout_color="#fff"
            android:textSize="18sp" />

    </LinearLayout>

</LinearLayout>

</RelativeLayout>
```

The layout has two parts. The header part only has a TextView to display the city name. The bottom part is the weather data layout and has a few TextViews to display the temperature, sky, and air quality information.

Next, create forecast.xml to display the weather forecast with the following code:

```
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="15dp"
    android:layout_marginRight="15dp"
```

```
    android:layout_marginTop="15dp"
    app:cardCornerRadius="4dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="15dp"
            android:layout_marginTop="20dp"
            android:layout_marginBottom="20dp"
            android:text="Forecast"
            android:textColor="?android:attr/textColorPrimary"
            android:textSize="20sp"/>

        <LinearLayout
            android:id="@+id/forecastLayout"
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="wrap_content">
        </LinearLayout>

    </LinearLayout>

</com.google.android.material.card.MaterialCardView>
```

The parent layout is MaterialCardView so that everything will be in a card. The TextView display the title, and the LinearLayout will contain the forecasts which will add the views based on the data returned from server.

Now we need to define an item view layout for forecast. Create forecast\_item.xml with the following code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="15dp">

    <TextView
        android:id="@+id/dateInfo"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_weight="4" />

    <ImageView
        android:id="@+id/skyIcon"
```

```
    android:layout_width="20dp"
    android:layout_height="20dp" />

<TextView
    android:id="@+id/skyInfo"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="3"
    android:gravity="center" />

<TextView
    android:id="@+id/temperatureInfo"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_weight="3"
    android:gravity="end" />

</LinearLayout>
```

This item view layout contains three TextViews and one ImageView, and they will display the date, weather icon, sky, low temperature, and high temperature, respectively.

Next create life\_index.xml to display the life index with the following code:

```
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="15dp"
    app:cardCornerRadius="4dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginStart="15dp"
            android:layout_marginTop="20dp"
            android:text="life index"
            android:textColor="?android:attr/textColorPrimary"
            android:textSize="20sp" />

        <LinearLayout
            android:layout_width="match_parent"
```

```
    android:layout_height="wrap_content"
    android:layout_marginTop="20dp">

    <RelativeLayout
        android:layout_width="0dp"
        android:layout_height="60dp"
        android:layout_weight="1">

        <ImageView
            android:id="@+id/coldRiskImg"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:layout_marginStart="20dp"
            android:src="@drawable/ic_coldrisk" />

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:layout_toEndOf="@+id/coldRiskImg"
            android:layout_marginStart="20dp"
            android:orientation="vertical">

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="12sp"
                android:text="Cold Risk" />

            <TextView
                android:id="@+id/coldRiskText"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_marginTop="4dp"
                android:textSize="16sp"
                android:textColor="?android:attr/textColorPrimary" />
        </LinearLayout>

    </RelativeLayout>

    <RelativeLayout
        android:layout_width="0dp"
        android:layout_height="60dp"
        android:layout_weight="1">

        <ImageView
            android:id="@+id/dressingImg"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
```

```
        android:layout_marginStart="20dp"
        android:src="@drawable/ic_dressing" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_toEndOf="@+id/dressingImg"
        android:layout_marginStart="20dp"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="12sp"
            android:text="Clothes" />

        <TextView
            android:id="@+id/dressingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp"
            android:textSize="16sp"
            android:textColor="?android:attr/textColorPrimary" />
    </LinearLayout>

</RelativeLayout>

</LinearLayout>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="20dp">

    <RelativeLayout
        android:layout_width="0dp"
        android:layout_height="60dp"
        android:layout_weight="1">

        <ImageView
            android:id="@+id/ultravioletImg"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:layout_marginStart="20dp"
            android:src="@drawable/ic_ultraviolet" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
```

```
        android:layout_toEndOf="@+id/ultravioletImg"
        android:layout_marginStart="20dp"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="12sp"
            android:text="UV" />

        <TextView
            android:id="@+id/ultravioletText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp"
            android:textSize="16sp"
            android:textColor="?android:attr/textColorPrimary" />
    </LinearLayout>

</RelativeLayout>

<RelativeLayout
    android:layout_width="0dp"
    android:layout_height="60dp"
    android:layout_weight="1">

    <ImageView
        android:id="@+id/carWashingImg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_marginStart="20dp"
        android:src="@drawable/ic_carwashing" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:layout_toEndOf="@+id/carWashingImg"
        android:layout_marginStart="20dp"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="12sp"
            android:text="Car Wash" />

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

```
        android:layout_marginTop="4dp"
        android:textSize="16sp"
        android:textColor="?android:attr/textColorPrimary" />
    </LinearLayout>

</RelativeLayout>

</LinearLayout>

</com.google.android.material.card.MaterialCardView>
```

This layout has lots of code but actually not complicated at all. It basically defines grid with four cells to display the index for cold risk, clothes, real-time UV, and car wash index. All of the cells are the same. Each cell has one ImageView to display the icon, one TextView to display title, and another TextView to display the index. It should be easy to understand this layout.

Next, we need to import them into activity\_weather.xml as shown in code below:

```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/weatherLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scrollbars="none"
    android:overScrollMode="never"
    android:visibility="invisible">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

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

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

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

    </LinearLayout>

</ScrollView>
```

Since there are a lot of contents in this screen, ScrollView is used to allow user scroll and see the contents that are outside of the screen. As only one direct child layout is allowed inside ScrollView, we add another vertical LinearLayout to import all the layouts defined previously.

Notice that we need to hide the ScrollView initially; otherwise it does not look good to show it without any data. After weather data request is successful, we will show the ScrollView.

After finishing UI layouts, we need to implement WeatherActivity. Before that we need to create a function that can map some weather information(CLOUDY, WIND etc) to Sky object. Create Sky.kt in logic/model with the following code:

```
class Sky (val info: String, val icon: Int, val bg: Int)

private val sky = mapOf(
    "CLEAR_DAY" to Sky("Clear", R.drawable.ic_clear_day, R.drawable.
bg_clear_day),
    "CLEAR_NIGHT" to Sky("Clear", R.drawable.ic_clear_night, R.
drawable.bg_clear_night),
    "PARTLY_CLOUDY_DAY" to Sky("Cloudy", R.drawable.
ic_partly_cloud_day,
        R.drawable.bg_partly_cloudy_day),
    "PARTLY_CLOUDY_NIGHT" to Sky("Cloudy", R.drawable.
ic_partly_cloud_night,
        R.drawable.bg_partly_cloudy_night),
    "CLOUDY" to Sky("Cloudy", R.drawable.ic_cloudy, R.drawable.
bg_cloudy),
    "WIND" to Sky("Windy", R.drawable.ic_cloudy, R.drawable.bg_wind),
    "LIGHT_RAIN" to Sky("Light Rain", R.drawable.ic_light_rain, R.
drawable.bg_rain),
    "MODERATE_RAIN" to Sky("Moderate Rain", R.drawable.
ic_moderate_rain, R.drawable.bg_rain),
    "HEAVY_RAIN" to Sky("Heavy Rain", R.drawable.ic_heavy_rain, R.
drawable.bg_rain),
    "STORM_RAIN" to Sky("Rain Storm", R.drawable.ic_storm_rain, R.
drawable.bg_rain),
    "THUNDER_SHOWER" to Sky("Thunder Shower", R.drawable.
ic_thunder_shower, R.drawable.bg_rain),
    "SLEET" to Sky("Sleet", R.drawable.ic_sleet, R.drawable.bg_rain),
    "LIGHT_SNOW" to Sky("Light Snow", R.drawable.ic_light_snow, R.
drawable.bg_snow),
    "MODERATE_SNOW" to Sky("Moderate Snow", R.drawable.
ic_moderate_snow, R.drawable.bg_snow),
    "HEAVY_SNOW" to Sky("Heavy Snow", R.drawable.ic_heavy_snow, R.
drawable.bg_snow),
    "STORM_SNOW" to Sky("Snow Storm", R.drawable.ic_heavy_snow, R.
drawable.bg_snow),
    "HAIL" to Sky("Hail", R.drawable.ic_hail, R.drawable.bg_snow),
    "LIGHT_HAZE" to Sky("Light Haze", R.drawable.ic_light_haze, R.
drawable.bg_fog),
    "MODERATE_HAZE" to Sky("Moderate Haze", R.drawable.
ic_moderate_haze, R.drawable.bg_fog),
    "HEAVY_HAZE" to Sky("Heavy Haze", R.drawable.ic_heavy_haze, R.
drawable.bg_fog),
    "FOG" to Sky("Foggy", R.drawable.ic_fog, R.drawable.bg_fog),
    "DUST" to Sky("Dusty", R.drawable.ic_fog, R.drawable.bg_fog)
)
```

```
fun getSky(skycon: String): Sky {
    return sky[skycon] ?: sky["CLEAR_DAY"] !!
}
```

The sky class is the data model and contains info, icon, and bg fields. The mapOf() function maps the weather info to the corresponding text, icon, and background. But it is difficult to get all the icons and backgrounds; thus similar weather types will be using the same icon and background. The getSky() method will get the Sky object based on weather code. That is all for the transformation function.

Next, we can request the weather data and display it in the UI. Edit WeatherActivity as code below:

```
class WeatherActivity : AppCompatActivity() {

    val viewModel by lazy { ViewModelProviders.of(this).get
    (WeatherViewModel::
        class.java) }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_weather)
        if (viewModel.locationLng.isEmpty()) {
            viewModel.locationLng = intent.getStringExtra("location_lng")
        }
        if (viewModel.locationLat.isEmpty()) {
            viewModel.locationLat = intent.getStringExtra("location_lat")
        }
        if (viewModel.placeName.isEmpty()) {
            viewModel.placeName = intent.getStringExtra("place_name") ?: ""
        }
        viewModel.weatherLiveData.observe(this, Observer { result ->
            val weather = result.getOrNull()
            if (weather != null) {
                showWeatherInfo(weather)
            } else {
                Toast.makeText(this, "Failed to get weather data",
                    Toast.LENGTH_SHORT).show()
                result.exceptionOrNull()?.printStackTrace()
            }
        })
        viewModel.refreshWeather(viewModel.locationLng, viewModel.
        locationLat)
    }

    private fun showWeatherInfo(weather: Weather) {
        placeName.text = viewModel.placeName
        val realtime = weather.realtime
        val daily = weather.daily
        // fill now.xml with data
    }
}
```

```

    val currentTempText = "${realtime.temperature.toInt()} °C"
    currentTemp.text = currentTempText
    currentSky.text = getSky(realtime.skycon).info
    val currentPM25Text = "Air Quality Index ${realtime.airQuality.
aqi.chn.toInt()}"
    currentAQI.text = currentPM25Text
    nowLayout.setBackgroundResource(getSky(realtime.skycon).bg)
    // fill forecast.xml with data
    forecastLayout.removeAllViews()
    val days = daily.skycon.size
    for (i in 0 until days) {
        val skycon = daily.skycon[i]
        val temperature = daily.temperature[i]
        val view = LayoutInflater.from(this).inflate(R.layout.
forecast_item,
            forecastLayout, false)
        val dateInfo = view.findViewById(R.id.dateInfo) as TextView
        val skyIcon = view.findViewById(R.id.skyIcon) as ImageView
        val skyInfo = view.findViewById(R.id.skyInfo) as TextView
        val temperatureInfo = view.findViewById(R.id.temperatureInfo) as
TextView
        val simpleDateFormat = SimpleDateFormat("yyyy-MM-dd", Locale.
getDefault())
        dateInfo.text = simpleDateFormat.format(skycon.date)
        val sky = getSky(skycon.value)
        skyIcon.setImageResource(sky.icon)
        skyInfo.text = sky.info
        val tempText = "${temperature.min.toInt()} ~ ${temperature.max.
toInt()} °C"
        temperatureInfo.text = tempText
        forecastLayout.addView(view)
    }
    // fill life_index.xml with data
    val lifeIndex = daily.lifeIndex
    coldRiskText.text = lifeIndex.coldRisk[0].desc
    dressingText.text = lifeIndex.dressing[0].desc
    ultravioletText.text = lifeIndex.ultraviolet[0].desc
    carWashingText.text = lifeIndex.carWashing[0].desc
    weatherLayout.visibility = View.VISIBLE
}
}

```

In onCreate(), we first get the longitude and latitude from Intent and assign them to the variables in WeatherViewModel; then observe the weatherLiveData, and when the response is received, call showWeatherInfo() to parse the result and display; lastly, call refreshWeather() in WeatherViewModel to refresh the request.

As for the logic in the showWeatherInfo() method, this is relatively simple. It actually gets the data from the Weather object and displays it on the corresponding control. Notice that, to display the forecast, for-in loop is used to iterate the data of each day and dynamically load and add forecast\_item.xml and set the data. Although the response contains multiple days' data for life index, we only need to show the

current day's data; thus we get the data at position 0. After setting the data, do not forget to make ScrollView visible.

After this, we need to implement the logic to switch from search screen to the weather information screen. Edit PlaceAdapter as shown below:

```
class PlaceAdapter(private val fragment: Fragment, private val
placeList: List<Place>) :
    RecyclerView.Adapter<PlaceAdapter.ViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.
place_item,
            parent, false)
        val holder = ViewHolder(view)
        holder.itemView.setOnClickListener {
            val position = holder.adapterPosition
            val place = placeList[position]
            val intent = Intent(parent.context, WeatherActivity::class.
java).apply {
                putExtra("location_lng", place.location.lng)
                putExtra("location_lat", place.location.lat)
                putExtra("place_name", place.name)
            }
            fragment.startActivity(intent)
        }
        return holder
    }
    ...
}
```

We register a click listener to the root layout of place\_item.xml, and in the click event, we get the longitude and latitude information and also the name of the location, and then pass them to Intent and start WeatherActivity by calling startActivity() of Fragment.

### 15.5.3 Record City Selection

Now we can search and display the weather information; however we do not keep the selection anywhere. This means that when user reopen the app, they need to search again which is not acceptable. In this section, we will implement keep record of the selection.

Obviously, we need to use data persistence here, and since the search selection is not relational data, we do not need to use database, and SharedPreferences is enough here.

However, even though SharedPreferences is simple to use, we will still follow MVVM pattern and do not write the code in UI control layer.

First, create PlaceDao singleton in logic/dao package with the following code:

```
object PlaceDao {

    fun savePlace(place: Place) {
        sharedPreferences().edit {
            putString("place", Gson().toJson(place))
        }
    }

    fun getSavedPlace(): Place {
        val placeJson = sharedPreferences().getString("place", "")
        return Gson().fromJson(placeJson, Place::class.java)
    }

    fun isPlaceSaved() = sharedPreferences().contains("place")

    private fun sharedPreferences() = SunnyWeatherApplication.context.
        getSharedPreferences("sunny_weather", Context.MODE_PRIVATE)

}
```

In PlaceDao, we encapsulate a few necessary save, read, and write interfaces. In savePlace(), we use GSON to transform Place object to a JSON string and save the string.

Read is the opposite operation. In getSavedPlace(), we get the JSON string from SharedPreferences and use GSON to parse the JSON string to Place object and return it.

After encapsulation PlaceDao, we can implement the logic in data storage layer. Edit Repository as shown in code below:

```
object Repository {
    ...
    fun savePlace(place: Place) = PlaceDao.savePlace(place)

    fun getSavedPlace() = PlaceDao.getSavedPlace()

    fun isPlaceSaved() = PlaceDao.isPlaceSaved()

}
```

This layer only encapsulates the methods in PlaceDao. This is for simplicity.

Since this is related to PlaceViewModel, we still need to add another layer of encapsulation in PlaceViewModel as shown in code below:

```
class PlaceViewModel : ViewModel() {
    ...
    fun savePlace(place: Place) = Repository.savePlace(place)
```

```

    fun getSavedPlace() = Repository.getSavedPlace()

    fun isPlaceSaved() = Repository.isPlaceSaved()

}

```

Since these interfaces in data do not start new thread, we can just call them without using LiveData.

After implementing the save, read, and write functionalities, we can apply them in the code. First, edit PlaceAdapter as shown in code below:

```

class PlaceAdapter(private val fragment: PlaceFragment, private val
placeList:
    List<Place>) : RecyclerView.Adapter<PlaceAdapter.ViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
    val view = LayoutInflater.from(parent.context).inflate(R.layout.
place_item,
        parent, false)
    val holder = ViewHolder(view)
    holder.itemView.setOnClickListener {
        val position = holder.adapterPosition
        val place = placeList[position]
        val intent = Intent(parent.context, WeatherActivity::class.
java).apply {
            putExtra("location_lng", place.location.lng)
            putExtra("location_lat", place.location.lat)
            putExtra("place_name", place.name)
        }
        fragment.viewModel.savePlace(place)
        fragment.startActivity(intent)
        fragment.activity?.finish()
    }
    return holder
}
...
}

```

There are mainly two changes. First, replace the Fragment in PlaceAdapter main constructor to PlaceFragment so that the PlaceviewModel can be referenced. Second, in onCreateViewHolder(), after clicking the item view, before opening WeatherActivity, savePlace() of PlaceViewModel is called to save the selection.

After saving the data, let us implement read the data. Edit PlaceFragment as shown in code below:

```

class PlaceFragment : Fragment() {
    ...
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)

```

```

    if (viewModel.isPlaceSaved()) {
        val place = viewModel.getSavedPlace()
        val intent = Intent(context, WeatherActivity::class.java).apply
    {
        putExtra("location_lng", place.location.lng)
        putExtra("location_lat", place.location.lat)
        putExtra("place_name", place.name)
    }
    startActivity(intent)
    activity?.finish()
    return
}
...
}

```

Inside the PlaceFragment, if there is saved result, then read the data and parse it to Place object and pass its data to WeatherActivity so that once a user opens the app, they can see the last selection without searching and selecting again.

Run the app again. Search and select “Beijing” again, then exit the app, then exit app and reopen the app, and you should see the weather screen with the latest weather data.

OK, that is what we need to do for Phase II, and let us commit the code.

```

git add .
git commit -m "Add display weather data"
git push origin master

```

## 15.6 Manual Refresh and Switch City

Now you probably can notice that after selecting one city, there is no way to search other cities even after exiting the app. Thus, let us add the capability to switch cities and manual refresh to get the latest weather data.

### 15.6.1 *Manual Refreshing Weather*

The weather data in the screen will not change after being opened. Usually user will try pull to refresh, and we will implement this in this section.

First, edit activity\_weather.xml as shown in code below:

```

<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/swipeRefresh"

```

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

    <ScrollView
        android:id="@+id/weatherLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:overScrollMode="never"
        android:scrollbars="none"
        android:visibility="invisible">
        ...
    </ScrollView>

</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

The code changes put the ScrollView into a SwipeRefreshLayout which makes ScrollView can be pulled.

Then edit WeatherActivity to add the refresh logic as shown below:

```
class WeatherActivity : AppCompatActivity() {

    val viewModel by lazy { ViewModelProviders.of(this).get
(WeatherViewModel::
    class.java) }

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        viewModel.weatherLiveData.observe(this, Observer { result ->
            val weather = result.getOrNull()
            if (weather != null) {
                showWeatherInfo(weather)
            } else {
                Toast.makeText(this, "Failed to get weather data",
                    Toast.LENGTH_SHORT).show()
                result.exceptionOrNull()?.printStackTrace()
            }
            swipeRefresh.isRefreshing = false
        })
        swipeRefresh.setColorSchemeResources(R.color.colorPrimary)
        refreshWeather()
        swipeRefresh.setOnRefreshListener {
            refreshWeather()
        }
    }

    fun refreshWeather() {
        viewModel.refreshWeather(viewModel.locationLng, viewModel.
locationLat)
        swipeRefresh.isRefreshing = true
    }
}
```

```
 } ...
```

The code changes abstract the refreshing code into refreshWeather() which calls refreshWeather() of WeatherViewModel and then sets isRefreshing property of SwipeRefreshLayout to true to display the refresh progress bar. In onCreate(), setColorSchemeResources() of SwipeRefreshLayout is called to set the color of the refresh progress bar which we set to colorPrimary defined in colors.xml. The setOnRefreshListener( ) set the listener for pull to refresh event and calls refreshWeather() to refresh weather data.

Do not forget to set isRefreshing to false after getting the response to hide the refresh progress bar.

### 15.6.2 Switching Cities

To switch cities, we need to be able to search cities first which has been finished in Sect. 15.4, and Fragment was used so that we can reuse it. Now we just need to import this Fragment to add the switch cities function.

Apparently, we cannot let the new Fragment to cover the weather information, and we can use the navigation drawer in Sect. 12.3 and put the Fragment into the drawer so that it will be hidden when not needed, and when user needs to switch cities, they can slide in the drawer.

Based on the Material Design guideline, we need to add a button to switch cities in the header layout so that user is aware that they can swipe. Edit now.xml as shown below:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/nowLayout"  
    android:layout_width="match_parent"  
    android:layout_height="530dp"  
    android:orientation="vertical">  
  
<FrameLayout  
    android:id="@+id/titleLayout"  
    android:layout_width="match_parent"  
    android:layout_height="70dp"  
    android:fitsSystemWindows="true">  
  
<Button  
    android:id="@+id/navBtn"  
    android:layout_width="30dp"  
    android:layout_height="30dp"  
    android:layout_marginStart="15dp"  
    android:layout_gravity="center_vertical"  
    android:background="@drawable/ic_home" />
```

```
    ...
  </FrameLayout>
  ...
</RelativeLayout>
```

The above code adds a button on the left side to switch cities.

Then edit activity\_weather.xml to add in the drawer as shown in code below:

```
<androidx.drawerlayout.widget.DrawerLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/drawerLayout"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipeRefresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    ...
  </androidx.swiperefreshlayout.widget.SwipeRefreshLayout >

  <FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:clickable="true"
    android:focusable="true"
    android:background="@color/colorPrimary">

    <fragment
      android:id="@+id/placeFragment"
      android:name="com.sunnyweather.android.ui.place.
PlaceFragment"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:layout_marginTop="25dp" />

  </FrameLayout>

</androidx.drawerlayout.widget.DrawerLayout>
```

The above code put the SwipeRefreshLayout into a DrawerLayout. Inside this layout, the first child control displays the content in the main screen. The second control displays the content of the drawer; thus, we add the search fragment here. To prevent the search box overlap with the system status bar, some margin on the top is added.

Next, we need to add the control logic for the drawer in WeatherActivity. Edit WeatherActivity as shown in code below:

```

class WeatherActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ...
        navBtn.setOnClickListener {
            drawerLayout.openDrawer(GravityCompat.START)
        }
        drawerLayout.addDrawerListener(object : DrawerLayout.
DrawerListener {
            override fun onDrawerStateChanged(newState: Int) {}

            override fun onDrawerSlide(drawerView: View, slideOffset: Float)
            {}

            override fun onDrawerOpened(drawerView: View) {}

            override fun onDrawerClosed(drawerView: View) {
                val manager = getSystemService(Context.INPUT_METHOD_SERVICE)
                    as InputMethodManager
                manager.hideSoftInputFromWindow(drawerView.windowToken,
                    InputMethodManager.HIDE_NOT_ALWAYS)
            }
        })
    }
    ...
}

```

In the click event of switch button, `openDrawer()` is called to open the drawer. We need to listen to the state of `DrawerLayout` as when the drawer is hidden, IME needs to be hidden too. Otherwise if drawer is closed but IME is still shown, it will block the view.

We added the condition check in `PlaceFragment` that if there is selection in the `SharedPreferences`, then `WeatherActivity` will be opened. After `PlaceFragment` is put into `WeatherActivity`, this logic will infinitely open activity. Thus we need to edit `PlaceFragment` with the following change:

```

class PlaceFragment : Fragment() {
    ...
    override fun onActivityCreated(savedInstanceState: Bundle?) {
        super.onActivityCreated(savedInstanceState)
        if (activity is MainActivity && viewModel.isPlaceSaved()) {
            val place = viewModel.getSavedPlace()
            val intent = Intent(context, WeatherActivity::class.java).apply
            {
                putExtra("location_lng", place.location.lng)
                putExtra("location_lat", place.location.lat)
                putExtra("place_name", place.name)
            }
            startActivity(intent)
            activity?.finish()
        }
    }
}

```

```
        return
    }
    ...
}
```

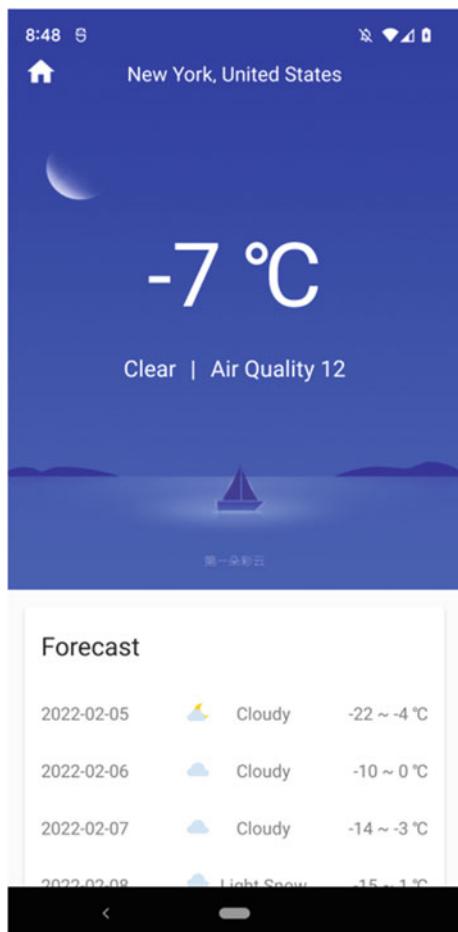
```
}
```

We only want to open WeatherActivity if the current screen is MainActivity.

After switching the city, we need to update the corresponding logic too which will happen in PlaceAdapter as we need to refresh the weather data.

Obviously, we also need to determine which Activity PlaceFragment is in. Edit PlaceAdapter as shown in code below:

```
class PlaceAdapter(private val fragment: PlaceFragment, private val
placeList:
List<Place>) : RecyclerView.Adapter<PlaceAdapter.ViewHolder>() {
    ...
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.
place_item,
                parent, false)
        val holder = ViewHolder(view)
        holder.itemView.setOnClickListener {
            val position = holder.adapterPosition
            val place = placeList[position]
            val activity = fragment.activity
            if (activity is WeatherActivity) {
                activity.drawerLayout.closeDrawers()
                activity.viewModel.locationLng = place.location.lng
                activity.viewModel.locationLat = place.location.lat
                activity.viewModel.placeName = place.name
                activity.refreshWeather()
            } else {
                val intent = Intent(parent.context, WeatherActivity::class.
java).
apply {
                putExtra("location_lng", place.location.lng)
                putExtra("location_lat", place.location.lat)
                putExtra("place_name", place.name)
            }
                fragment.startActivity(intent)
                activity?.finish()
            }
            fragment.viewModel.savePlace(place)
        }
        return holder
    }
    ...
}
```

**Fig. 15.24** Switch button**Forecast**

|            |  |            |             |
|------------|--|------------|-------------|
| 2022-02-05 |  | Cloudy     | -22 ~ -4 °C |
| 2022-02-06 |  | Cloudy     | -10 ~ 0 °C  |
| 2022-02-07 |  | Cloudy     | -14 ~ -3 °C |
| 2022-02-08 |  | Light Snow | 15 ~ 1 °C   |

If PlaceFragment is in WeatherActivity, then close the drawer and assign WeatherViewModel new values and refresh the weather data. If PlaceFragment is in MainActivity, reuse the previous logic.

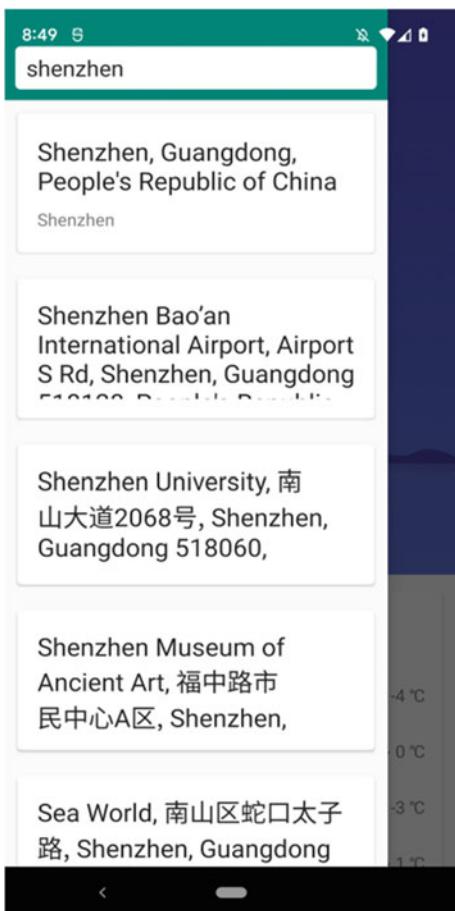
That is all we need to switch the city. Run the app again, and the result should be as shown in Fig. 15.24.

Now the title bar has a button to switch the city. Clicking the button or swiping from left side will display the drawer, and we can search and switch city here. It should be as shown in Fig. 15.25.

After selecting new city, the drawer should close automatically, and the weather information should be updated accordingly.

That is the end for Phase III. Now let us commit the code changes

**Fig. 15.25** Open drawer to search and switch



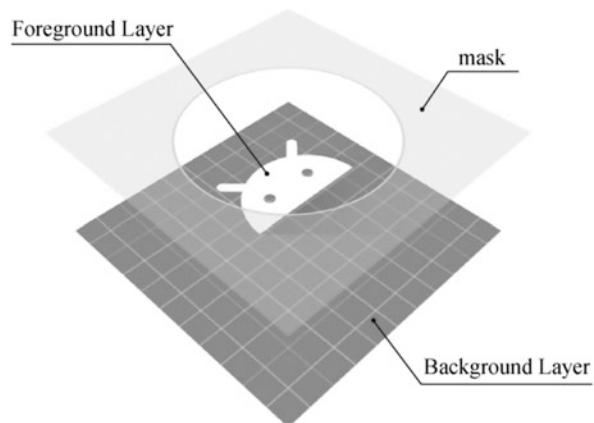
```
git add .
git commit -m "Add switching city and manual refresh weather. "
git push origin master
```

## 15.7 Create App Icon

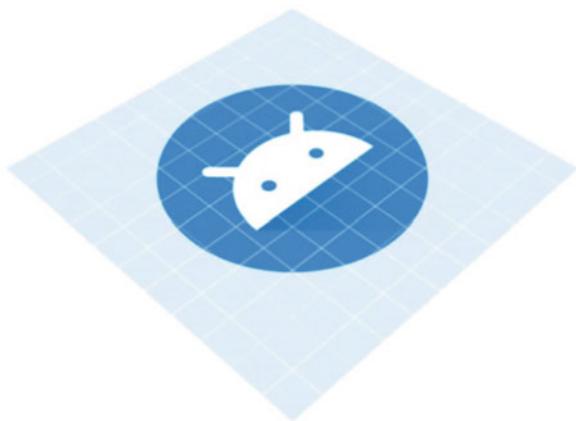
We cannot release SunnyWeather now, and one of the reasons is that we do not even have an icon for it yet. It is not a good idea to use the default icon, and in this section we will discuss how to make app icon.

A while ago, the icon images should be saved into folders of corresponding resolutions. However, since Android 8.0, Google does not recommend using single image as the app icon but should have the background asset and foreground asset. The foreground should display the logo of app, and the background should be pure

**Fig. 15.26** V8.0 and above icon layers



**Fig. 15.27** Processed icon



background which is only allowed to define color and texture, and no shape can be defined.

Then who will define the shape of the icon? Google gives the right to OEMs. OEMs will add another mask layer above the foreground and background which can be any shape as they wish. This makes all the icons have the same shape in the same OEM's phone. This can be illustrated in Fig. 15.26

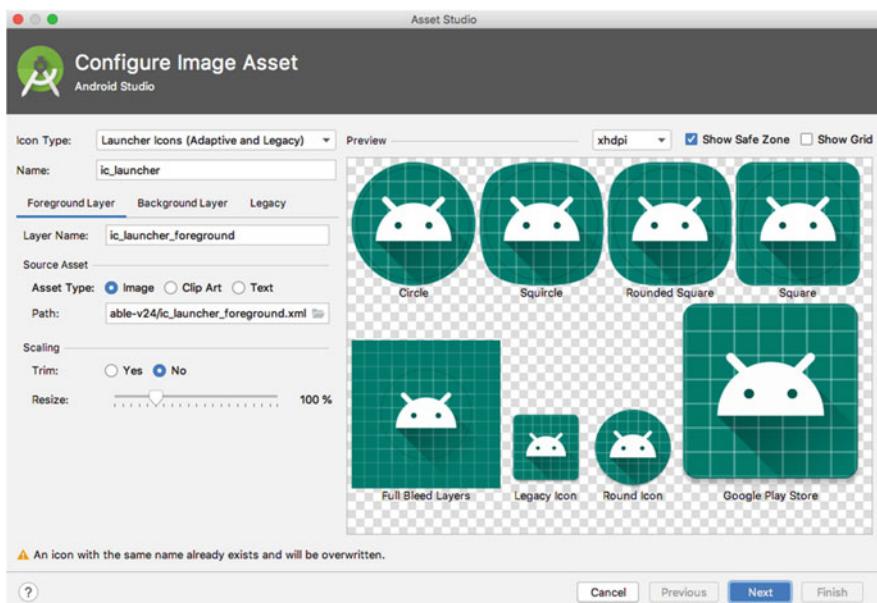
As you can tell, if the mask is round shape, then after processing, we will get round icon which is shown in Fig. 15.27.

Now let us start to make our own icon. I prepared an image as the foreground layer which is Logo(download from the prelude or GitHub link). As I'm not an artist, I made it really simple, as shown in Fig. 15.28.

Then we can use the Asset Studio in Android Studio to create icon that is compatible to different versions of OS. Click navigation bar File→New→Image Asset to open Asset Studio as shown in Fig. 15.29.

The left side is the action panel, and the right side is the preview view.

**Fig. 15.28** Foreground of icon

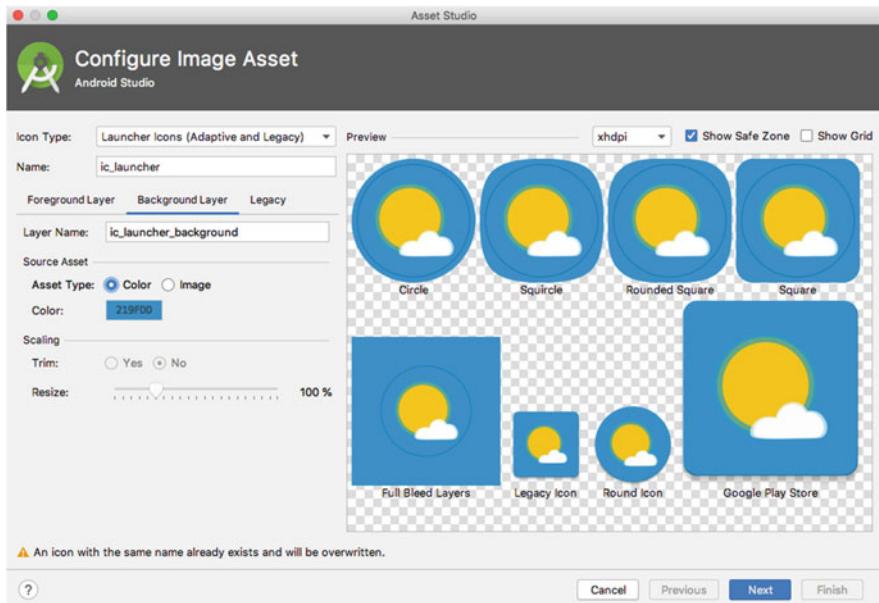


**Fig. 15.29** Asset Studio UI

Keep Icon Type as it is as the default value means creating icon that is compatible with V8.0 and earlier versions of OS. Keep the Name as ic\_launcher too so that the auto-generated icon will be replaced by this. There are three tabs for us to edit the corresponding asset. Legacy is used to edit the old OS's icon.

The preview displays the final result of the icon which covers all the possible shapes. Notice that there is a circle in each preview which is called safe zone. We need to make sure that foreground is inside safe zone; otherwise, OEM mask may crop part of the logo.

Select Logo in Foreground Layer and resize it using the Resize bar to make sure the content of Logo is in the safe zone. Then, in Background Layer, select “Color” as the Asset Type and use #219FDD as the background color. The final preview is as shown in Fig. 15.30.



**Fig. 15.30** Icon preview

Now it should be able to adapt to different masks.

Click “Next” to open the window to confirm icon path, click “Finish” to finish creating the icon. All the icon-related files will be generated and saved in different mipmap folders which are as shown in Fig. 15.31.

Notice that in mipmap-anydpi-v26, it is a xml file instead of images. Phones with Android 8.0 and above will use file in this directory as icon. Open `ic_launcher.xml`, we should see the following code:

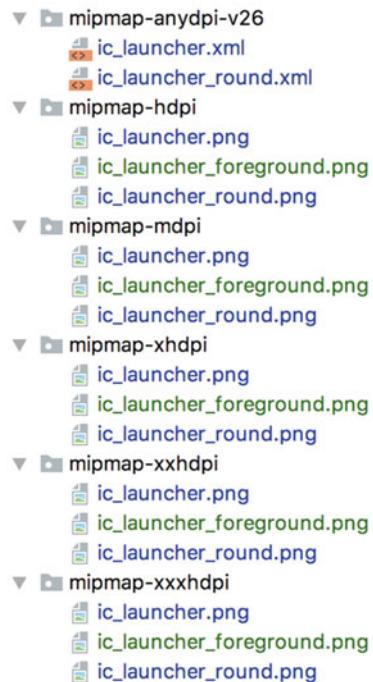
```
<adaptive-icon xmlns:android="http://schemas.android.com/apk/res/android">
    <background android:drawable="@color/ic_launcher_background"/>
    <foreground android:drawable="@mipmap/ic_launcher_foreground"/>
</adaptive-icon>
```

This is the standard way to make icons adaptive to Android 8.0 and above. The `<adaptive-icon>` tag defines `<background>` tag to set the background with the color we set previously and defines `<foreground>` tag to set the foreground of the icon which is image we prepared.

This `ic_launcher.xml` is referenced by `AndroidManifest.xml`. Open this file, and you should see the following code:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.sunnyweather.android">
```

**Fig. 15.31** Files in mipmap folders

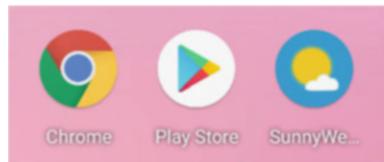


```

...
<application
    android:name=".SunnyWeatherApplication"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
</application>
</manifest>
  
```

The `android:icon` attribute in `<application>` tag is used to set the app icon, and here the value is `@mipmap/ic_launcher`. With this, for Android 8.0 and above versions, `ic_launcher.xml` in `mipmap-anydpi-v26` will be used as app icon. For Android 7.0 and lower versions, the `ic_launcher.png` in different mipmap folders will be used. The `android:roundIcon` attribute only applies to Android 7.1 which is a transitional version, and we do not need to do extra work for it. Run the app again, and the launcher should show the new icon as shown in Fig. 15.32.

The SunnyWeather icon is round in Pixel emulator which is the same as other icons. If you run the app in different phone, the shape may change.

**Fig. 15.32** Icon in launcher**Fig. 15.33** keystore file

```
> Task :app:signingReport
Variant: debugUnitTest
Config: debug
Store: /Users/guolin/.android/debug.keystore
Alias: AndroidDebugKey
```

Also you should notice that SunnyWeather is too long and cannot display all text. To rename it to something shorter, open res/values/string.xml and edit the following code:

```
<resources>
    <string name="app_name" >SunnyWeather</string>
</resources>
```

At the end, do not forget to submit changes.

```
git add .
git commit -m "Update app icon. "
git push origin master
```

That is all we need to do!

## 15.8 Generate Signed APK

We use Android Studio to install the app, and what actually happens is that Android Studio will compile the code and package everything into an APK file and transfer this file to phone, and then install this APK. All the APKs are treated as installer just as EXE file in Windows.

APK file has to be signed so that it can be installed. The reason why we did not sign and still could install is that Android Studio used a default keystore file to automatically sign the APK. Click the right tool bar in Android Studio Gradle→Project Name→app→Tasks→android and double click “signingReport,” and the result should be as shown in Fig. 15.33.

Using debug.keystore file to sign the APK only works for development purpose, and we need a real keystore file to sign the APK so that it can be released. Let us take a look at how to do so.

### 15.8.1 Generating with Android Studio

To use Android Studio to sign the APK. Click Build→Generate Signed Bundle/APK and dialog as shown in Fig. 15.34 will pop up.

The bundle file is used in Google Play store. With this file, Google Play can let the device only download necessary resources. For instance, a high resolution phone does not need to use the image sources that only used for low resolution devices; a device with arm architecture does not need to download the so(C/C++ lib) file used in X86 architecture. Thus, Bundle file can significantly minimize the footprint of the App. The drawback of this file is that you cannot directly install it in the phone nor can it be used in app stores other than Google Play.

After selecting the format, the rest process is the same. Take APK file as an example. Click “Next,” then a window to ask us fill the path and password will show up, as shown in Fig. 15.35.

Since we do not have keystore file yet, click “Create new,” and a new dialog will pop up to fill the necessary information to create keystore. It should look like in Fig. 15.36.

Make sure that the value for Validity is long enough, and I set it to 50 years. Then click “OK,” and the filled in information in this dialog will be automatically transferred to the new dialog as shown in Fig. 15.37.

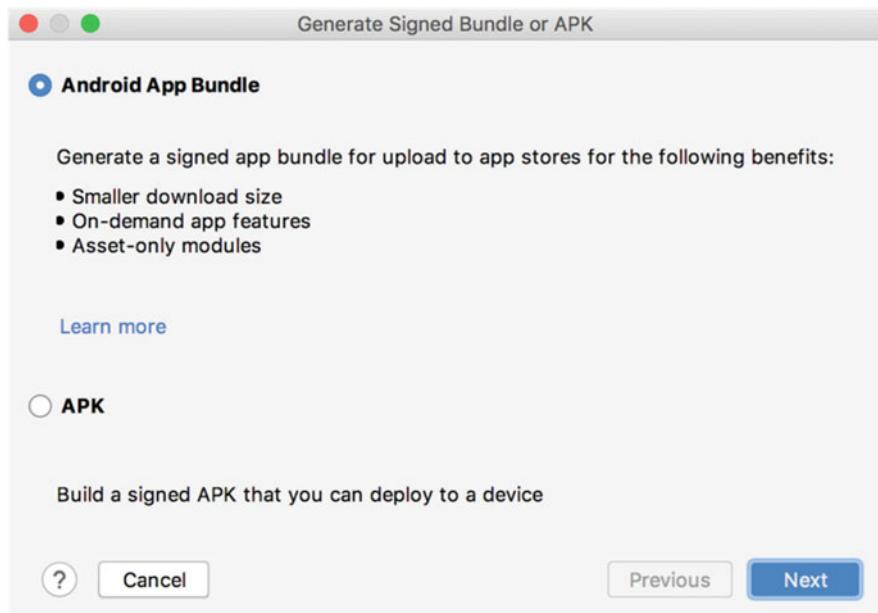
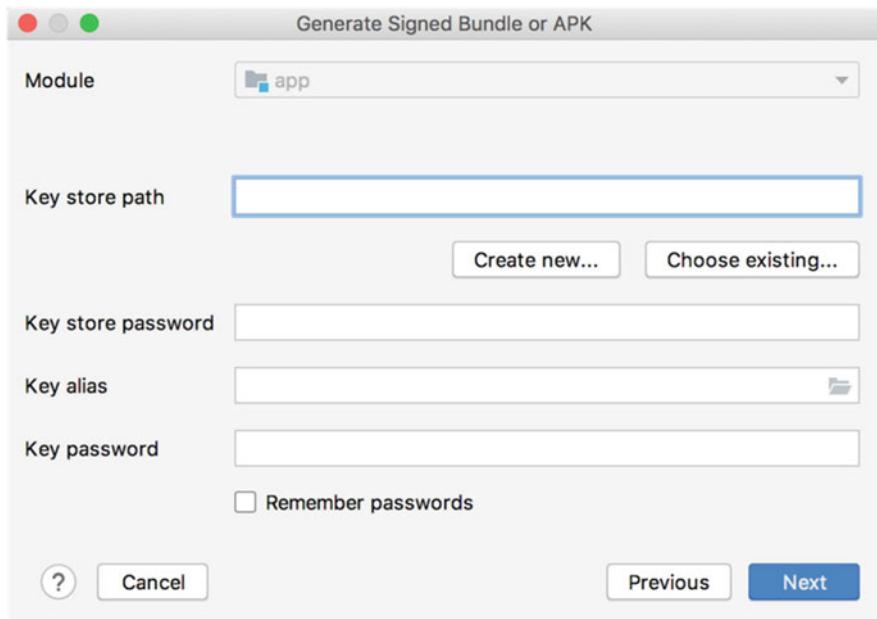


Fig. 15.34 Dialog to generate Bundle/APK



**Fig. 15.35** Dialog to generate bundle or APK

If you do not want to type password again, just check “Remember passwords.” Then click “Next,” and the dialog to select APK destination will pop up as shown in Fig. 15.38.

Keep the destination as it is, and select “release” for build type because we are creating the release APK file. Make sure to check V1 and V2 to make signature compatible with all versions of OS.

Click “Finish” and wait for some time. After APK is generated, a message as shown in Fig. 15.39 will pop up at the bottom right.

Click locate to check the generated APK file as shown in Fig. 15.40.

This app-release.apk can be directly installed or uploaded to app stores. If we selected Android App Bundle, we will get a .aab file which is the preferred format for Google Play.

### 15.8.2 Generating with Gradle

We can also use Gradle to generate the signed APK and let us take a look how to do it.

Gradle is an advanced build tool, and we benefit a lot from it in our projects. For instance, to add dependency lib, we can just add a statement in dependencies instead of manual downloading.

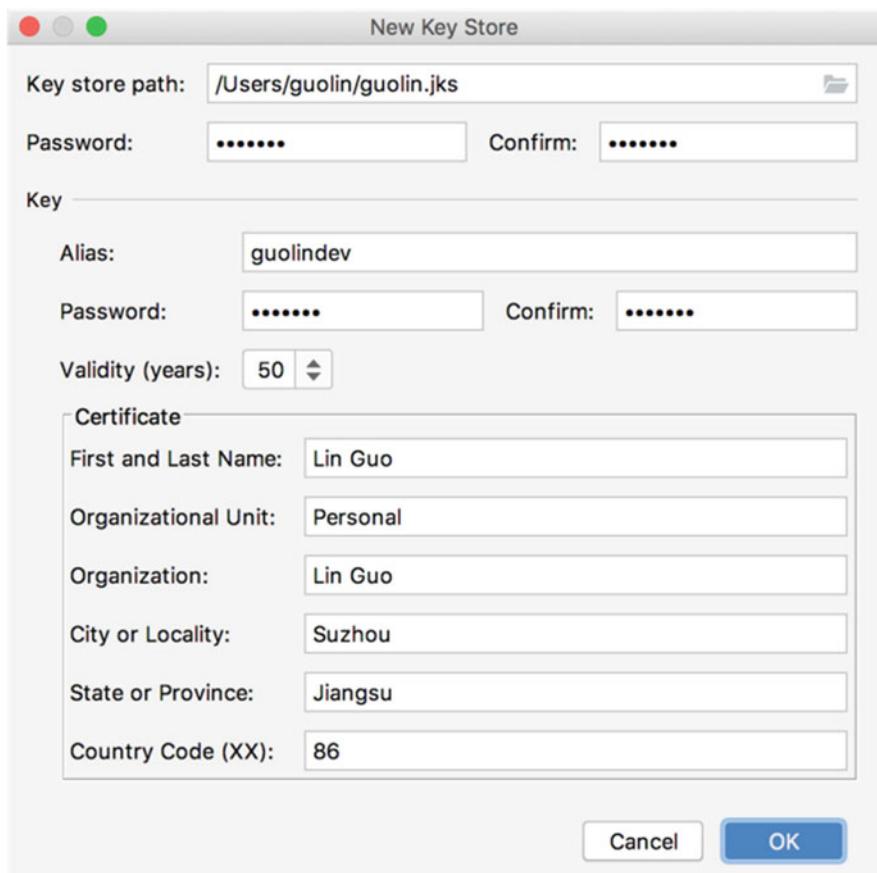


Fig. 15.36 keystore information

To use Gradle to generate signed APK, first, edit app/build.gradle and add the following code in android closure:

```
android {  
    compileSdkVersion 29  
    defaultConfig {  
        applicationId "com.sunnyweather.android"  
        minSdkVersion 21  
        targetSdkVersion 29  
        versionCode 1  
        versionName "1.0"  
        testInstrumentationRunner "androidx.test.runner.  
AndroidJUnitRunner"  
    }  
    signingConfigs {  
        config {
```

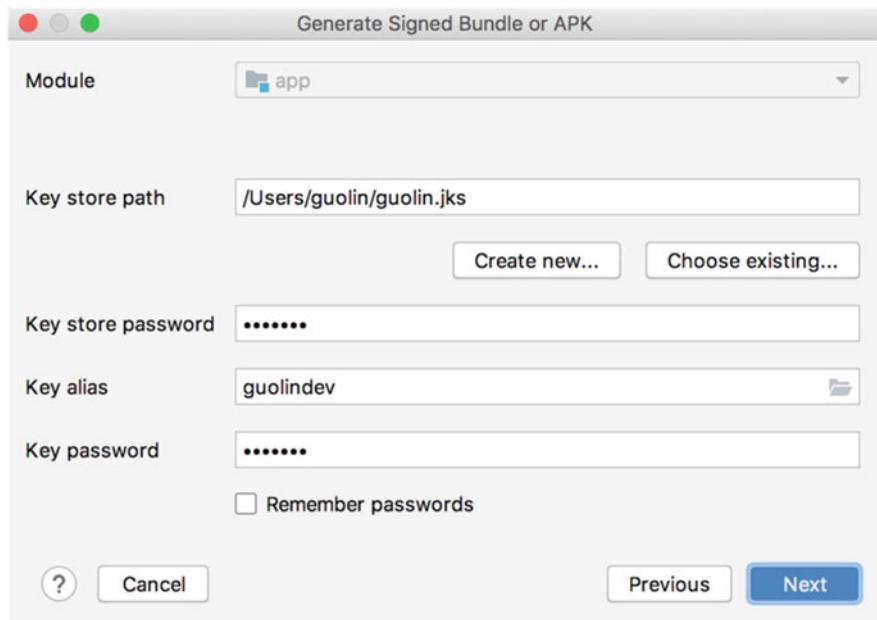


Fig. 15.37 Auto fill

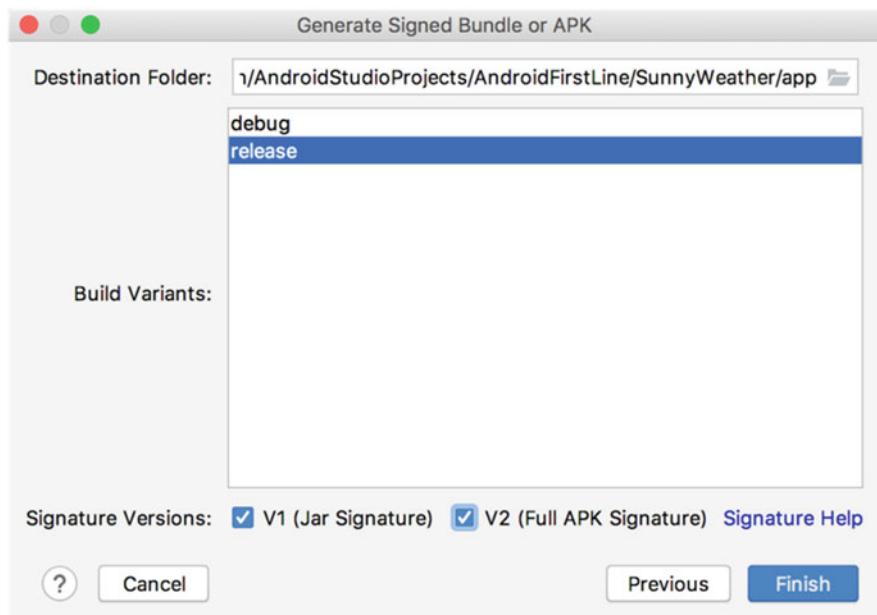
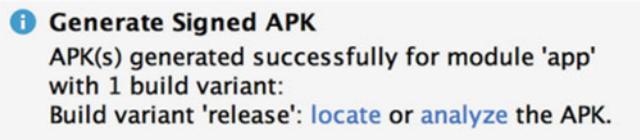


Fig. 15.38 Select file destination



**Fig. 15.39** Generate signed APK success message

| 名称              | 修改日期   | 大小     | 种类    |
|-----------------|--------|--------|-------|
| app-release.apk | 下午1:12 | 4.9 MB | 文稿    |
| output.json     | 下午1:12 | 234 字节 | 纯文本文稿 |

**Fig. 15.40** Generated APK file

```

storeFile file('/Users/guolin/guolin.jks')
storePassword '1234567'
keyAlias = 'guolindev'
keyPassword '1234567'
}
}
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                    'proguard-rules.pro'
    }
}

```

We add signingConfigs closure inside android closure and add config closure inside signingConfigs closure. Then we configure the keystore information inside config closure. The fields in the closure should be self-explanatory.

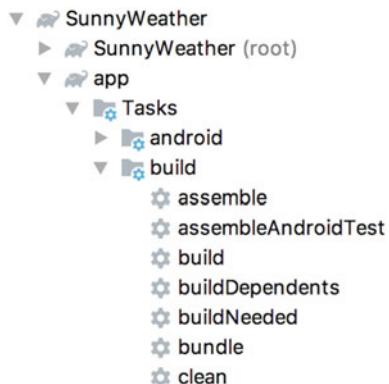
After configuration, we just need to use this configuration when generating the release version APK. Edit app/build.gradle as shown below:

```

android {
    ...
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
                        'proguard-rules.pro'
            signingConfig signingConfigs.config
        }
    }
}

```

**Fig. 15.41** Internal Gradle tasks



```

BUILD SUCCESSFUL in 8s
53 actionable tasks: 7 executed, 46 up-to-date
21:28:08: Task execution finished 'assemble'.
  
```

**Fig. 15.42** Run assemble task

```

}
}
```

We added the configuration in release closure that is in buildTypes closure. Then when generating the release APK, the signing configuration will be used to sign.

How to generate APK file now? There are many internal Gradle tasks in Android Studio, and there is a task to generate APK. Right click Gradle→Project Name→app→Tasks→build, as shown in Fig. 15.41.

The assemble task is used to generate APK file and will generate debug and release versions at the same time. Double click to run this task, and the result should be as shown in Fig. 15.42.

After successfully running assemble, APK files are in app/build/outputs/apk, as shown in Fig. 15.43.

The app-release.apk in release folder is now signed.

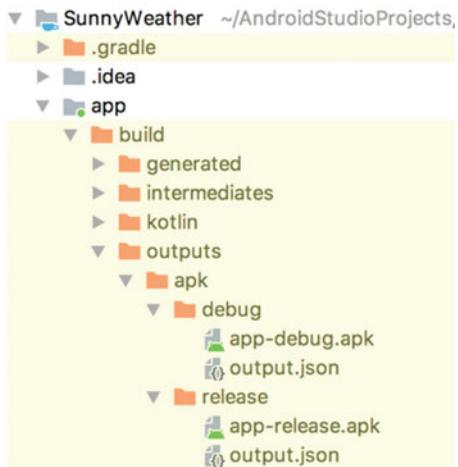
Notice that all the information in keystore is in plain text in build.gradle which is not safe especially considering that SunnyWeather is an open-source project which basically leaks the password of the keystore file. The recommended way is to save these sensitive information in an independent file and read this file in build.gradle.

To follow this pattern, we can add the following content in gradle.properties file under the Android Studio project root directory:

```

KEY_PATH=/Users/guolin/guolin.jks
KEY_PASS=1234567
ALIAS_NAME=guolindev
ALIAS_PASS=1234567
  
```

**Fig. 15.43** Generated APK files



This file is dedicated to store the global key-value pairs. And we save the keystore information in key-value pairs. To read this in build.gradle, edit app/build.gradle as shown below:

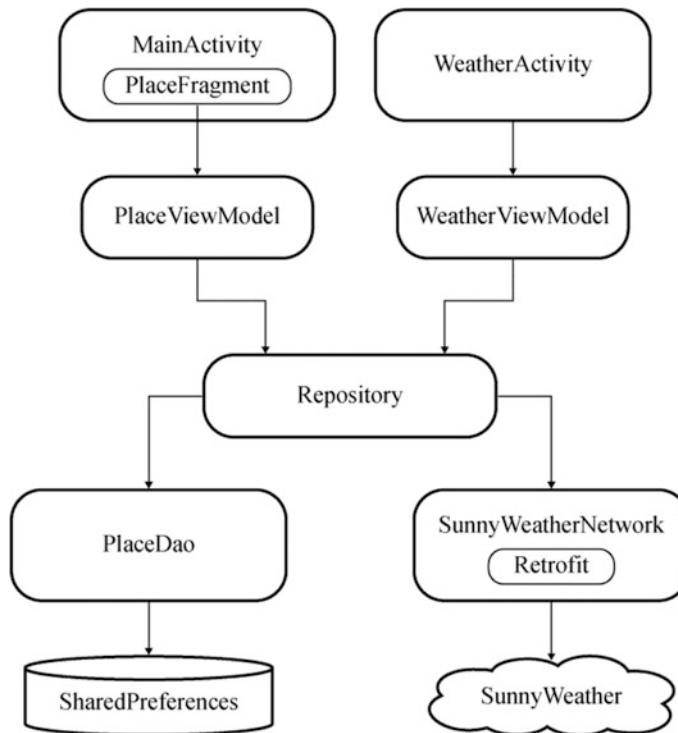
```
android {
    ...
    signingConfigs {
        config {
            storeFile file(KEY_PATH)
            storePassword KEY_PASS
            keyAlias ALIAS_NAME
            keyPassword ALIAS_PASS
        }
    }
    ...
}
```

We just need to replace the plain text with the keys. And if gradle.properties is safe, there is no need to worry about leakage, we can do so to add this file in .gitignore and only keep this file in local machine.

## 15.9 More To Do

That is all for this chapter, and our project strictly follows the MVVM pattern. The architecture of Sunny Weather is illustrated in Fig. 15.44, and it should be crystal clear to you now.

As you can tell, SunnyWeather has a clear and layered architecture. Remember that project with good architectural design can always be illustrated with concise and



**Fig. 15.44** SunnyWeather architecture



**Fig. 15.45** Fork button in GitHub

clear architecture diagram, and the badly designed and messy ones will not be able to have a clear diagram.

But even now, SunnyWeather is still far from perfect. It only has some basic functionality, and here are things you can consider to improve:

- Provide more comprehensive weather information as we only display a small portion of the response
- Enable multi-city display at the same time
- Enable background update with configurable update frequency
- Dark mode

As SunnyWeather is hosted in GitHub, you can fork a branch to work on.

To do so, login your GitHub account and open the frontpage of SunnyWeather: <https://github.com/guolindev/SunnyWeather>, and you should see “Fork” button at the top right corner, as shown in Fig. 15.45.

Click “Fork,” and you will get a copy under your account. Use GIT clone to get a copy in local machine, and then you can make any changes as you wish.